Technical University of Munich
Faculty for Informatics

Evaluation of dimension-wise Error Estimates using the Spatially Adaptive Combination Technique

Bachelor's Thesis

by
Vivian Haller

Technical University of Munich
Faculty for Informatics

Evaluation of dimension-wise Error Estimates using the Spatially Adaptive Combination Technique

Evaluierung dimensionsabhängiger Fehlerschätzer in der räumlich-adaptiven Kombinationstechnik

Bachelor's Thesis

Finalised at the
Chair of Scientific Computing in Computer Science I5
Faculty for Informatics
Technical University of Munich

Author: Vivian Haller
Supervisor: Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisor: M.Sc. Michael Obersteiner

Submission date: May 15, 2019

I would like to specially thank my parents for making my studies and everything else possible, as well as my advisor, Michael Obersteiner, for his continued patience and support.

Vivian Haller, May 1st, 2019
Munich

I hereby confirm that this work is my own and that no materials or sources apart from those documented have been utilised.

Munich, May 15, 2019 _____

Vivian Haller

Abstract

In [10], a novel approach for numerical quadrature has been developed, based on a spatially adaptive variant of the sparse grid combination technique [7]. In this thesis, the resulting procedure, henceforth referred to as the split-extend scheme, has been subject to a slight modification regarding one of its fundamental operations, essentially reducing its consumption of function evaluations per application. This modified procedure, restricted to piecewise linear basis functions, has been put to the test with regard to a selection of widely used functions taken from [5] and [8], some of which have already been employed in [10]. In several cases, the corresponding results have either shown a noticeably increased performance or proved to be at least on par with the basic scheme, which indicates that the method itself might be worthy of further investigation and improvement.

Das in [10] eingeführte, auf einer räumlich adaptiven Variante der in [7] konzipierten Kombinationstechnik für dünnbesetzte Gitter basierende, im Folgenden als Split-Extend-Methode bezeichnete numerische Quadraturverfahren wurde im Zuge dieser Abschlussarbeit einer leichten Modifikation im Hinblick auf eine seiner beiden grundlegenden Operationen unterzogen, welche eine substantielle Reduktion der nötigen Funktionsauswertungen pro Ausführung mit sich zieht. Diese Variante wurde mit einer Einschränkung auf stückweise lineare Basisfunktionen an einer Auswahl gebräuchlicher, [5] und [8] entnommener Funktionen zur Ausführung gebracht, welche bereits in [10] zu Testzwecken eingesetzt wurden. In einigen Fällen sollten sich die entsprechenden Resultate als deutlich besser herausstellen oder sich zumindest auf Augenhöhe mit jenen des unmodifizierten Verfahrens befinden, was zu zusätzlichen Nachforschungen und Verbesserungen anregen mag.

Contents

Notations

Symbols	Description
$\mathbb{N}$,$\mathbb{N}_0$	The set of non-negative integers without/including 0.
$[d]$	The positive integers less than or equal to $d \in \mathbb{N}$.
$\mathbb{R}$, $\mathbb{R}^d$	The set of real numbers/vectors with real components.
$\boldsymbol{l}, \boldsymbol{j}$	Vectors in $\mathbb{R}^d$, usually multi-indices in $\mathbb{N}_0^d$. $\boldsymbol{l}$ corresponds to a grid's level and $\boldsymbol{j}$ to a point's/basis function's index.
l_i	The i-th component of $\boldsymbol{l}$.
$\mathbb{1}$	The one vector, i.e., $(1, \ldots, 1) \in \mathbb{R}^d$.
$\boldsymbol{e}_i$	The k-th standard unit vector in $\mathbb{R}^d$.
Ω	Subset of $\mathbb{R}^d$, typically the objective function's domain.
$f : \Omega \to \mathbb{R}$	The objective function, typically integrable.
$\overline{\Omega}$, $\partial\Omega$	Closure and boundary of Ω in $\mathbb{R}^d$.
$\lvert \cdot \rvert$	A norm on $\mathbb{R}^d$ (modulus in the scalar case).
$\lvert \cdot \rvert_1$	1-norm on $\mathbb{R}^d$; one has $\lvert\boldsymbol{l}\rvert_1 := \sum_{i\in[d]} \lvert l_i\rvert$.
$\lvert \cdot \rvert_\infty$	Max-norm on $\mathbb{R}^d$; one has $\lvert\boldsymbol{l}\rvert_\infty := \max_{i\in[d]} \lvert l_i\rvert$.
$L^p(\Omega)$	Lebesgue space of p-integrable functions on Ω, $p \geq 1$.
$\lVert \cdot \rVert_p$	p-norm on $L^p(\Omega)$; one has $\lVert f\rVert_p := \left(\int_\Omega \lvert f(x)\rvert^p dx\right)^{\frac{1}{p}}$ for $1 \leq p < \infty$.
$\operatorname{supp}(f)$	Support of f, i.e., $\operatorname{supp}(f) := \overline{\{x \in \Omega \lvert f(x) \neq 0\}}$.
$C^k(\Omega)$	Space of k-times continuously differentiable functions, $k \in \mathbb{N}_0 \cup \{\infty\}$.
Df, $\frac{\partial}{\partial x_i} f$	Total differential and i-th partial derivative of $f \in C^k(\Omega)$, $k \geq 1$.
$\mathcal{D}(\Omega)$	Space of test functions $C_0^\infty = \{f \in C^\infty(\Omega) \lvert \ \operatorname{supp}(f) \subset \Omega \text{ compact}\}$.
$W^{k,p}, W_0^{k,p}, H^k, H_0^k$	Different kinds of Sobolev spaces, $k \in \mathbb{N}_0, p \geq 1$.
$\lVert \cdot \rVert_{k,p}$	Sobolev norms.
$D^{(\boldsymbol{\alpha})} f$	Weak derivative of $f \in W^{k,p}(\Omega)$ w.r.t. multi-index $\boldsymbol{\alpha} \in \mathbb{N}_0^d$, $\lvert\boldsymbol{\alpha}\rvert_1 \leq k$.
$\Omega_{\boldsymbol{l}}$	Anisotropic full grid of level $\boldsymbol{l}$.
$h_l, h_{\boldsymbol{l}}, h_n$	Mesh widths in grids.
ϕ, $\phi_{l,j}$, $\phi_{\boldsymbol{l},\boldsymbol{j}}$	Piecewise linear hat functions.
$x_{l,j}, \boldsymbol{x}_{\boldsymbol{l},\boldsymbol{j}}$	Grid points associated with nodal basis functions.
$\operatorname{span}_{\mathbb{R}}(M)$	Smallest $\mathbb{R}$-linear subspace containing M w.r.t. set inclusion $\subset$.
$V \oplus V'$	Direct sum of linear spaces V, V'.
$\dim(V)$	The dimension of a linear space V.
$V_l, V_{\boldsymbol{l}}, V_n^{(\infty)}, V$	Spaces spanned by nodal point bases.
$\mathcal{I}_l, \mathcal{I}_{\boldsymbol{l}}$	Index sets related to the hierarchical basis' construction.
$W_l, W_{\boldsymbol{l}}$	Increment spaces.
$\alpha_{l,j}, \alpha_{\boldsymbol{l},\boldsymbol{j}}$	Coefficients in the hierarchical representation.
$\mathcal{O}$	Landau notation, also used as an index set.
$V_n^{(1)}, V_{0,n}^{(1)}$	Sparse grid spaces corresponding to in-/homogeneous boundary conditions.
$Q_l f$	Conventional sparse grid quadrature.
$f_l, f_{\boldsymbol{l}}, f_n, f_n^c$	The piecewise linear interpolant of f. f_n^c is constructed using the standard combination technique.
$\boldsymbol{\tau}$	Truncation parameter s.th. $\tau_i \in \mathbb{N}_0 \cup \{-1\}, i \in [d]$ used in the truncated combination technique.
$\operatorname{argmax}_{i\in[d]} x_i$	If not unique, the smallest $k \in [d]$ s.th. $x_k = \max_{i\in[d]} x_i$.
$\mathcal{I}, \mathcal{A}, \mathcal{O}$	General index sets.

Introduction

When numerically integrating a scalar objective function $f : [a, b] \to \mathbb{R}$, one typically approximates its integral by a linear combination of (positive) weights w_i and evaluations of f on $n \in \mathbb{N}$ distinct points $x_0, \ldots, x_{n-1} \in [a, b]$, the abscissas, which may be fixed beforehand (by using e.g. an equidistant discretisation of $[a, b]$):

$$\int_{[a,b]} f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i).$$

A natural way to extend such schemes into higher dimensions, e.g. onto some domain $\Omega \subset \mathbb{R}^d$, would consist of using a certain discretisation in each dimension. However, this approach quickly succumbs to the so-called curse of dimensionality, as the total number of points would lie in $\mathcal{O}(n^d)$ and thus nowadays become computationally infeasible for $n > 4$ (cf. [11, p.5]).
Sparse grids, which require significantly fewer points of evaluation, allow to mitigate the curse to a certain extent while maintaining the error decay of full grids up to a logarithmic factor. Their construction is based on a sparse tensor product decomposition, which dates back to numerical integration of partial differential equations [12] (cf. [11, p.5]). In this thesis, such a tensor product construction is reclusively performed for piecewise linear basis functions, which sort of generalises strategies like Archimedes' quadrature.
As sparse grids tend to require more effort in implementation than full grids, since common full grid methods would need to be adapted, the so-called sparse grid combination technique comes in handy. It allows to obtain sparse grids by linearly combining so-called anisotropic full grids of varying resolutions, effectively enabling the use of both common full grid methods and parallelisation techniques, as one may process each grid separately and finally combine the individual solutions. As remarked in [11, p.5], this technique, which has been developed in [7], has found its way into a plethora of applications in such diverse fields as e.g. economics, regression, classification and uncertainty quantification.
As static grids prove insufficient when the objective function f shows highly varying characteristics in different parts of its domain Ω, adaptive approaches are required. While there already exist multiple dimension-adaptive procedures (e.g. by Gerstner and Griebel [6]) that make use of the combination technique, there have not yet occurred noteworthy breakthroughs in utilising it in a spatially adaptive way in higher dimensions (cf. [10, p.2]). The so-called split-extend scheme introduced in [10] is a spatially adaptive procedure based on the sparse grid combination technique, which can, among other problems, be applied to numerical quadrature, whence it will be the main focus of this thesis.

The contents of this work are presented in the following order: In the first section, the necessary theoretical concepts, such as weakly differentiable functions, the piecewise linear hierarchical basis, sparse grid spaces and the sparse grid combination technique are compiled. For more detailed discussions and the proofs of the statements given in the text, one may consider the corresponding sources, mainly [3], [4] and [11]. The second section shortly distinguishes between dimensional and spatial adaptivity and subsequently presents the split-extend scheme, i.e. the split and extension operations as well as the error and benefit estimators, with a restriction to quadrature rules based on piecewise linear basis functions. This section almost completely relies on [10] with respect to the split-extend scheme and furthermore to [6] and [11], regarding the fundamental concepts of adaptivity. The third section compiles the most important grid functionality of the **SpACE** framework [9] and moreover presents the contributions regarding the single-dimensional split operation made by the author. The text concludes with a series of test cases, most of them taken from Genz' test functions [5], whose results are also compared to those made in [10].

1 Theoretical Foundations

This section compiles some basic concepts and results needed to describe and discuss the adaptive integration procedure in the following sections, regarding mainly weakly differentiable functions, (piecewise linear) hierarchical basis, (generalised) sparse grids and the (truncated) combination technique.

1.1 Weak derivatives and Sobolev spaces

Let $\Omega \subset \mathbb{R}^d$ be an open set.
$f \in L^p(\Omega)$ is called **weakly differentiable** with derivative $D^{(\boldsymbol{\alpha})}f$, if there exists a $g \in L^p(\Omega)$ such that

$$\int_\Omega g(x)\psi(x)dx = (-1)^{|\boldsymbol{\alpha}|_1} \int_\Omega f(x) D^{(\alpha)}\psi(x)dx \quad \forall \psi \in \mathcal{D}(\Omega).$$

Hereby $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}_0^n$ denotes a multi-index and one defines

$$\frac{\partial^{|\boldsymbol{\alpha}|_1}}{\partial x_1^{\alpha_1}, \ldots, \partial x_n^{\alpha_n}}\psi := D^{(\boldsymbol{\alpha})}\psi.$$

Provided that such a g exists, it is unique (cf. [1]) and one sets $D^{(\boldsymbol{\alpha})}f = g$. This definition allows for a specialisation of the spaces $L^p(\Omega)$ by including the notion of differentiability (cf. [13, Def. 1.3.1]):

Definition 1.1 (Sobolev spaces)**.** *Let $k \in \mathbb{N}_0$ and $f \in L^p(\Omega)$, such that the weak derivatives $D^{(\alpha)}f$ exist for all multi-indices $\boldsymbol{\alpha}$, $|\boldsymbol{\alpha}|_1 \leq k$. One defines*

$$\|f\|_{k,p} := \begin{cases} \left(\sum_{|\boldsymbol{\alpha}|_1 \leq k} \|D^{(\boldsymbol{\alpha})}f\|_p^p\right)^{\frac{1}{p}} : 1 \leq p < \infty \\ \max_{|\boldsymbol{\alpha}|_1 \leq k} \|D^{(\boldsymbol{\alpha})}f\|_\infty : p = \infty \end{cases}$$
$$\text{and } W^{k,p}(\Omega) := \left\{ f \in L^p(\Omega) \middle| \|f\|_{k,p} < \infty \right\}$$

$\|\cdot\|_{k,p}$ defines a norm on $W^{k,p}(\Omega)$ and in the case $k = 0$ one has $D^{(\alpha)}f = D^{(0)}f = f$, as well as $\|f\|_{k,p} = \|f\|_p$, thus $W^{k,p}(\Omega) = L^p(\Omega)$. If $p = 2$, it is conventional to use the notation $H^k(\Omega) := W^{k,p}(\Omega)$. The spaces defined above are complete (cf. [13, Theorem 1.3.2]):

Theorem 1.2. *For $k \in \mathbb{N}_0, 1 \leq p \leq \infty$ the Sobolev space $(W^{k,p}(\Omega), \|\cdot\|_{k,p})$ is a Banach space.*

These spaces do not consist of functions in the general sense, but merely of equivalence classes of functions that coincide on sets of Lebesgue measure zero. Even though there exists a continuous representative in every such class if certain prerequisites are met (cf. [2, p.67]), it is not straightforward to combine boundary conditions with this notion, as the boundary of a domain $\Omega \subset \mathbb{R}^d$, provided that it fulfills certain smoothness conditions, is a $(n-1)$-dimensional submanifold and thus has measure zero. Therefore, another kind of spaces has been introduced, namely $W_0^{k,p} := \overline{C_0^\infty(\Omega)}$, where the closure is taken with regard to $\|\cdot\|_{k,p}$. Analogously to the previous definitions, H_0^k abbreviates $W_0^{k,2}$. If $V \subset \mathbb{R}^d$ is an open and bounded set such that $\Omega \subset V$ is compact in V and $\partial\Omega$ fulfils certain smoothness conditions, there exists a linear and continuous extension operator, called the **trace**,

$$T : W^{1,p}(\Omega) \to W_0^{1,p}(V),$$

in a way that $(Tf)|_\Omega = f$ for $f \in W^{1,p}(\Omega)$. As the practical approaches discussed in later sections of this text always require pointwise evaluations and the above function spaces are only needed for the compilation of theoretical results in this chapter, further details are omitted.

1.2 Hierarchical Basis

This section deals with a possible solution to the problem of interpolating a function that satisfies certain smoothness conditions using a piecewise linear approach. The first step will consist in defining discrete approximation spaces and a suitable set of basis functions, proceeding from one-dimensional domains to the multidimensional case via a tensor product approach. Moreover, a few important results regarding the approximation quality of these spaces with respect to the norms $\|\cdot\|_2$ and $\|\cdot\|_\infty$ are to be presented, while restricting the discussion to the case of homogeneous boundary conditions.
In what follows, especially in the case of a multi-dimensional domain $\Omega \subset \mathbb{R}^d$, multi-indices

$$\boldsymbol{l} = (l_1, \ldots, l_d), \boldsymbol{j} = (j_1, \ \ldots, j_d) \in \mathbb{N}_0^d,$$

will denote the **level** of a grid/grid point/basis function and the **index**, i.e. the location of a grid point/basis function, respectively. When dealing with grids, one generally assumes to have a bounded and rectangular domain $\Omega \subset \mathbb{R}^d$. For convenience, the following discussion is limited to the hypercube $\Omega = [0,1]^d$, since this setting can be achieved via a suitable rescaling and therefore imposes no loss of generality. Given a multi-index $\boldsymbol{l} \in \mathbb{N}_0^d$, the **anisotropic full grid** of mesh-width $h_{\boldsymbol{l}} := 2^{-\boldsymbol{l}} := (2^{-l_1}, \ldots, 2^{-l_d})$ on Ω is given by the set $\Omega_{\boldsymbol{l}}$ that contains the points

$$x_{\boldsymbol{l},\boldsymbol{j}} = (x_{l_1,j_1}, \ldots, x_{l_d,j_d}) := (j_1 \cdot h_1, \ldots, j_d \cdot h_d) := \boldsymbol{j} \cdot \boldsymbol{l} \qquad \text{for } \boldsymbol{0} \leq \boldsymbol{j} \leq 2^{\boldsymbol{l}},$$

where the inequalities are meant to be understood component-wise. Furthermore, a grid is called **block-adaptive**, if it has a block structure comprised of regular grids. The individual grids may have non-uniform resolutions, as depicted in figure 3.

1.2.1 1-dimensional case

A common choice is given by so-called hat functions. Each basis function is generated from the standard hat function ϕ,

$$\phi(x) := \begin{cases} 1 - |x| : x \in [-1,1] \\ 0 : \text{otherwise} \end{cases}$$

by means of translation and dilation: For any level $l \in \mathbb{N}$, an index j, $1 \leq j \leq 2^l - 1$ and step size $h_l = 2^{-l}$, define

$$\phi_{l,j}(x) := \phi\left(\frac{x - jh_l}{h_l}\right) = \phi(2^l x - j) \tag{1}$$

and the corresponding space $V_l := \text{span}_{\mathbb{R}} \left\{\phi_{l,j} | 1 \leq j \leq 2^l - 1\right\}$. Note that only inner grid points $x_{l,j}$ are taken into account in this first definition. This corresponds to homogeneous boundary conditions and will therefore generally not be suitable for interpolating functions that have non-zero values at the boundary of the domain, here at the points 0 and 1. Hence, an extra level $l = 0$ may be introduced, containing two basis functions $\phi_{0,0}, \phi_{0,1}$. This is depicted in figure 1. As will be pointed out when discussing the higher-dimensional case, it is usually better to regard $\phi_{0,0}, \phi_{0,1}$ and $\phi_{1,0}$ as all belonging to the first level, to avoid the introduction of additional subspaces or to use different basis functions altogether.
$\phi_{l,j}$ is centered at $x_{l,j} = ih_l$ and has compact support $\text{supp}_{\phi_{l,j}} = [x_{l,j} - h_l, x_{l,j} + h_l]$. By defining the hierarchical index sets $\mathcal{I}_l := \left\{j \in \mathbb{N} | 1 \leq j \leq 2^l - 1, j \text{ odd}\right\}$, the hierarchical increment spaces W_l are given by

$$W_l := \text{span}_{\mathbb{R}} \left\{\phi_{l,j} | j \in \mathcal{I}_l\right\} \text{ and it holds } V_n = \bigoplus_{l \leq n} W_l \text{ for any } n \in \mathbb{N}. \tag{2}$$

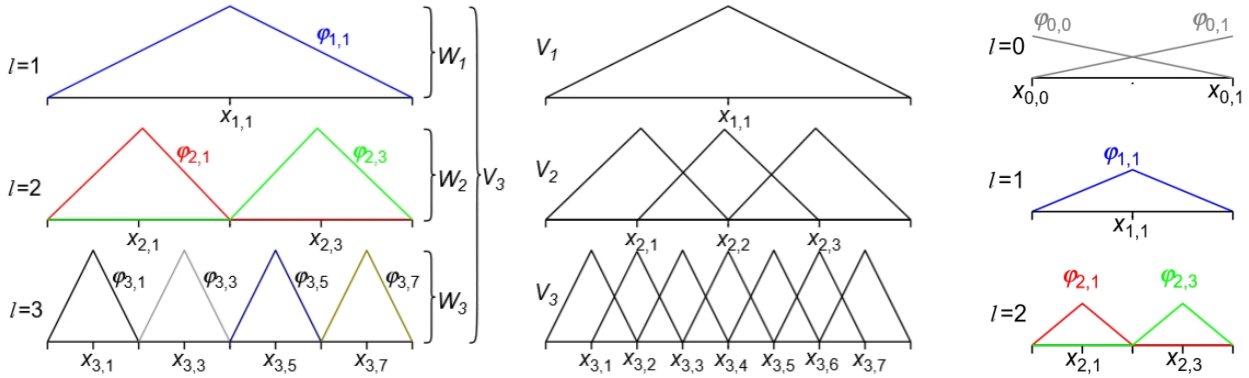


Figure 1: Comparison of hierarchical basis functions (left) and nodal basis functions (middle) $\phi_{l,j}$, as well as their corresponding grid points $x_{l,j}$ on levels $l \leq 3$ (taken from [11, p.9]). In order to deal with inhomogeneous boundary conditions, one may add an additional level $l = 0$ (right, taken from [11, p.14])

As figure 1 illustrates, all basis functions $\phi_{l,j}$ in W_l have the same size, shape and their supports are mutually disjoint. Due to (2), any function $f \in V_n$ can be represented via

$$f = \sum_{l \leq n} \sum_{j \in \mathcal{I}_l} \alpha_{l,j} \phi_{l,j}, \tag{3}$$

where the hierarchical coefficients $\alpha_{l,i}$ fulfil the relation

$$\alpha_{l,j} = f(x_{l,j}) - \frac{f(x_{l,j} - h_l) + f(x_{l,j} + h_l)}{2} = f(x_{l,j}) - \frac{f(x_{l-1,(j-1)/2}) + f(x_{l-1,(j+1)/2})}{2}. \tag{4}$$

(3), (4) and figure 2 justify the alternative term **hierarchical surpluses** commonly used when referring to the coefficients $\alpha_{l,i}$.

1.2.2 d-dimensional case

Given an anisotropic grid $\Omega_{\boldsymbol{l}}$, the corresponding space $V_{\boldsymbol{l}}$ of piecewise d-linear functions is defined by

$$V_{\boldsymbol{l}} := \operatorname{span}_{\mathbb{R}} \left\{ \phi_{\boldsymbol{l},\boldsymbol{j}} \middle| \mathbb{1} \leq \boldsymbol{j} \leq 2^{\boldsymbol{l}} - \mathbb{1} \right\}$$

via the tensor product approach $\phi_{\boldsymbol{l},\boldsymbol{j}}(x) := \prod_{i=1}^{d} \phi_{l_i,j_i}(x_i)$, $x \in \mathbb{R}^d$, using the one-dimensional hat functions ϕ_{l_i,j_i} from (1). In particular, they are linearly independent and each basis function $\phi_{\boldsymbol{l},\boldsymbol{j}}$ has an associated grid point $\boldsymbol{x}_{\boldsymbol{l},\boldsymbol{j}}$, at which it is centered [3]. Note, that $V_{\boldsymbol{l}}$ does only contain inner grid points and thus corresponds to the setting of homogeneous boundary conditions. Generalising the one-dimensional approach (2), one defines the d-dimensional hierarchical basis via the index set

$$\mathcal{I}_{\boldsymbol{l}} := \left\{ \boldsymbol{j} \in \mathbb{N}^d \middle| \mathbb{1} \leq \boldsymbol{j} \leq 2^{\boldsymbol{l}} - \mathbb{1}, j_k \text{ odd for } k \in [d] \right\}, \quad \text{resulting in } W_{\boldsymbol{l}} := \left\{ \phi_{\boldsymbol{l},\boldsymbol{j}} \middle| \boldsymbol{j} \in \mathcal{I}_{\boldsymbol{l}} \right\},$$

the increment spaces. As in the one-dimensional case (2), the supports of the basis functions $\phi_{\boldsymbol{l},\boldsymbol{j}} \in W_{\boldsymbol{l}}$ are mutually disjoint and $V_{\boldsymbol{l}}$ can be written as a direct sum of the increment spaces $W_{\boldsymbol{l}}$:

$$V_{\boldsymbol{l}} = \bigoplus_{\boldsymbol{k} \leq \boldsymbol{l}} W_{\boldsymbol{k}} = \operatorname{span}_{\mathbb{R}} \bigcup_{\boldsymbol{k} \leq \boldsymbol{l}} \left\{ \phi_{\boldsymbol{k},\boldsymbol{j}} \middle| \boldsymbol{j} \in \mathcal{I}_{\boldsymbol{k}} \right\} \tag{5}$$

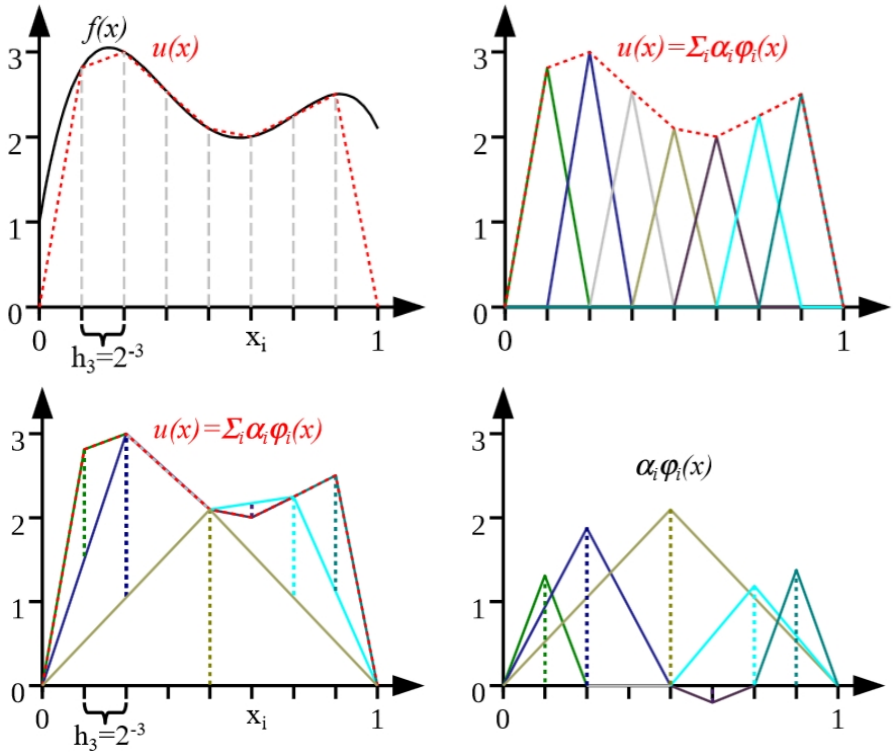


Figure 2: Piecewise linear interpolation of a scalar function $f : \mathbb{R} \to \mathbb{R}$ on $\Omega = [0, 1]$ using the nodal basis (first row, taken from [11, p.7]) and the 1d hierarchical basis (second row, taken from [11, p.9]). Homogeneous boundary conditions are assumed.

and one has obtained the d-dimensional hierarchical basis. Note that the dimension of the increment space $W_{\boldsymbol{l}}$ is $\dim(W_{\boldsymbol{l}}) = |\mathcal{I}_{\boldsymbol{l}}| = 2^{|\boldsymbol{l}-\mathbb{1}|_1}$. By setting $V_n^{(\infty)} := \bigoplus_{|\boldsymbol{k}|_\infty \leq n} W_{\boldsymbol{k}}$ for $n \in \mathbb{N}$ and $V := \bigoplus_{\boldsymbol{l} \in \mathbb{N}^d} W_{\boldsymbol{l}}$, one has $\lim_{n\to\infty} V_n^{(\infty)} = \bigcup_{n=1}^{\infty} V_n^{(\infty)} = V$, since $V_n^{(\infty)} \subset V_{n+1}^{(\infty)}$ for any $n \in \mathbb{N}$ and additionally, V is dense in $(H^1(\Omega), \|\cdot\|_{1,2})$, as remarked in [3, 3.16]. When arranging the subspaces, as depicted in figure 3 for two dimensions, the space $V_n^{(\infty)}$ corresponds to a quadratic sector of selected subspaces. Its dimension is $\dim(V_n^{(\infty)}) = (2^n \pm 1)^d \in \mathcal{O}(2^{dn}) = \mathcal{O}(h_n^{-d})$ [3, 3.31], again depending on the choice of boundary conditions. Using these definitions, each $f \in V_n$ has unique representations

$$f = \sum_{|\boldsymbol{l}|_\infty \leq n} \sum_{\boldsymbol{j} \in \mathcal{I}_{\boldsymbol{l}}} \alpha_{\boldsymbol{l},\boldsymbol{j}} \cdot \phi_{\boldsymbol{l},\boldsymbol{j}} = \sum_{|\boldsymbol{l}|_\infty \leq n} f_{\boldsymbol{l}} \tag{6}$$

where $f_{\boldsymbol{l}} \in W_{\boldsymbol{l}}$ and the coefficients $\alpha_{\boldsymbol{l},\boldsymbol{j}} \in \mathbb{R}$ are obtained from f by applying a generalisation of the one-dimensional scheme (4) mentioned in the previous subsection,

$$\alpha_{\boldsymbol{l},\boldsymbol{j}} = \prod_{k=1}^{d} \left[-\frac{1}{2} \quad 1 \quad -\frac{1}{2} \right]_{x_{l_k}, j_k, l_k}, \tag{7}$$

which corresponds to a d-dimensional stencil for a linear combination of nodal values [3, 3.23]. Now, a few selected properties of the hierarchical basis are to be compiled; for the sake of simplicity, this compilation restricts itself to homogeneous Dirichlet boundary conditions and thus functions $f \in H_0^2(\Omega)$. The underlying question is, how large the contributions of the individual increment spaces $W_{\boldsymbol{l}}$ to the piecewise d-linear interpolant of an $f \in H_0^2(\Omega)$ can be and of which order the interpolation error will be (cf. [4, p.7-8], [3, L3.5]):

Lemma 1.3. *Let $f \in H_0^2(\Omega)$ and let $f_n \in V_n^{(\infty)}$, $n \in \mathbb{N}$, denote the piecewise d-linear interpolant of f: Then $\|f - f_n\|_2 \in \mathcal{O}(h_n^2)$.*

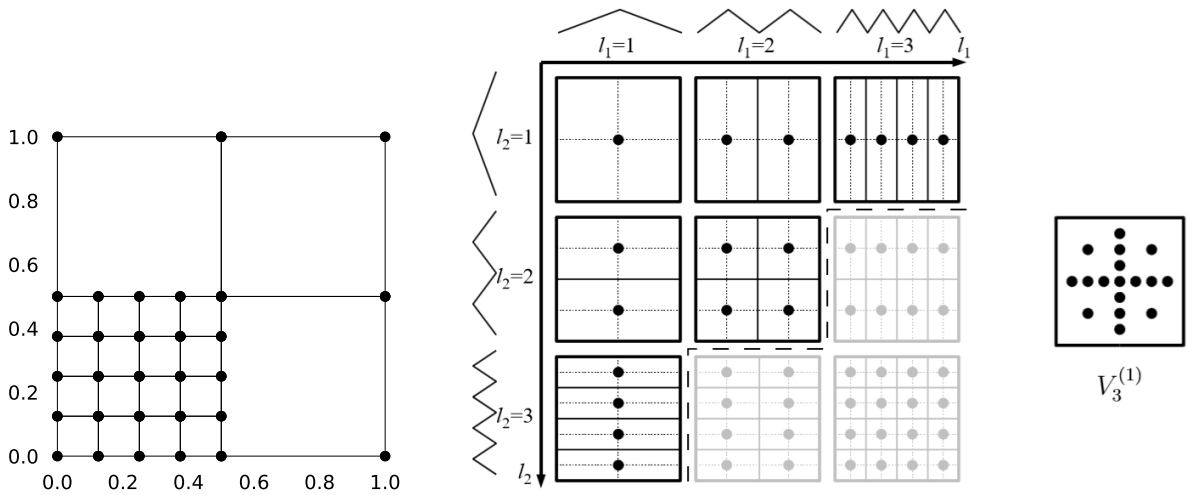


Figure 3: A block-adaptive grid (left, taken from [10, p.7]) and an illustration of the subspace selection in the definition of $V_{0,3}^{(1)}$ in 2d (right, taken from [11, p.13]).

It is an immediate conclusion, that the spaces $V_n^{(\infty)}$ are not satisfactory from a computational point of view, since they have $\mathcal{O}(h_n^{-d})$ degrees of freedom, while achieving second order error decay. Hence, it has been the goal to construct other finite-dimensional approximation spaces that have a better ratio of spent grid points versus accuracy, while restricting the candidates to spaces of the form $\bigoplus_{\boldsymbol{l}\in\mathcal{I}} W_{\boldsymbol{l}}$ for some finite $\mathcal{I} \subset \mathbb{N}^d$, which has to be optimised. This problem has been discussed in [3], using e.g. tools from discrete optimisation, which lead to sparse grids.

1.2.3 Sparse grid spaces

The underlying idea of sparse grids consists of neglecting hierarchical basis functions of small support, which therefore in total do not contribute that much to the hierarchical representation of f (cf. [4, p. 9]). By defining the sparse grid space via

$$V_n^{(1)} := \bigoplus_{|\boldsymbol{l}|_1 \leq n+d-1} W_{\boldsymbol{l}}, \tag{8}$$

one calls the corresponding grid a sparse grid. The underlying scheme of subspace selection consists of a triangular (2D) or more generally a simplicial sector (see figure 3), as opposed to the square in the case of $V_n^{(\infty)}$.
Analogously, any function $f \in V_n^{(1)}$ has unique representations $f = \sum_{|\boldsymbol{l}|_1 \leq n+d-1} \sum_{\boldsymbol{j}\in\mathcal{I}_{\boldsymbol{l}}} \alpha_{\boldsymbol{l},\boldsymbol{j}} \cdot \phi_{\boldsymbol{l},\boldsymbol{j}} = \sum_{|\boldsymbol{l}|_1 \leq n+d-1} f_{\boldsymbol{l}}$. In the case, that one only considers functions $f \in V_n^{(1)}$ that are zero on the boundary, a conventional way to define the sparse grid space is via

$$V_{0,n}^{(1)} := \bigoplus_{\substack{|\boldsymbol{l}|_1 \leq n+d-1 \\ \boldsymbol{l}>0}} W_{\boldsymbol{l}}.$$

Sparse grids were originally introduced this way by Zenger [14], the original idea and application to numerical integration is due to Smolyak [12]. Let $f : [0,1]^d \to \mathbb{R}$ and $(Q_l)_l$ denote a series of (scalar) quadrature formulas of level $l \in \mathbb{N}$ on $n_l \in \mathbb{N}$ abscissas $x_{l,1}, \ldots, x_{l,n_l} \in [0,1]$ having weights $w_{l,1}, \ldots, w_{l,n_l} \in \mathbb{R}$ such that $n_l < n_{l+1}$:

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

For any univariate real-valued function $g(x)$ and $l \in \mathbb{N}$, define the linear mapping $\Delta_k g \in \mathbb{R}$ via $\Delta_k g := (Q_k - Q_{k-1})g$, where $Q_0 g := 0$. Then, the **conventional sparse grid quadrature** Q_l for d-dimensional functions f on level $l \in \mathbb{N}$ is obtained by setting

$$Q_l f := \sum_{|\boldsymbol{k}|_1 \leq l+d-1} (\Delta_{k_1} \otimes \cdots \otimes \Delta_{k_d}) f, \tag{9}$$

see [6, p.4] for specific details. The next results are intended to briefly motivate the advantages of sparse grids over regular grids, especially with regard to the number of grid points used versus the approximation quality (cf. [4, p.11]):

Theorem 1.4. *Let $f_n \in V_{0,n}^{(1)}$ denote the interpolant of $f \in H_0^2(\Omega)$. Then the interpolation error w.r.t. the L^2-norm fulfills $\|f - f_n\|_2 = \mathcal{O}(h_n^2 \log^{d-1}(h_n^{-1}))$.*

Since full grids provide an interpolation error of $\mathcal{O}(h_n^2)$ with regard to $\|\cdot\|_2$, the quality of approximation deteriorates slightly when switching to sparse grids. However, this disadvantage is compensated by the fact, that the number of grid points and henceforth the computational and storage requirements drastically decrease (cf. [4, p.12]):

Theorem 1.5. *The number of inner grid points of $V_n^{(1)}$, $|V_{0,n}^{(1)}|$, lies in $\mathcal{O}(h_n^{-1} \cdot \log^{d-1}(h_n^{-1}))$.*

In the case, that there are no degrees of freedom on the boundary of the domain, this result shows that it is possible to mitigate the curse of dimensionality to some extent when using sparse grids instead of regular full grids. As remarked in [3], the corresponding subspace selection is optimal with regard to the norms $\|\cdot\|_2$ and $\|\cdot\|_\infty$, thus further improvement is impossible unless the problem setting is modified. An example of a sparse grid in two dimensions is depicted in figure 4. It is worth noting, that introducing the zero level basis functions in order to deal with inhomogeneous boundary conditions implies that the overwhelming majority of the grid points is located on $\partial\Omega$, especially in higher dimensions. Even though the number of grid points in $V_n^{(1)}$ and $V_{0,n}^{(1)}$ have equal asymptotic order, $|V_n^{(1)}|$ often seems too large to be practically applicable in higher dimensions (cf. [11, p.12-16]). It may even be useful to omit boundary grid points entirely and to instead modify the interior basis functions to extrapolate towards $\partial\Omega$ [11, p.16].

1.2.4 Combination technique

The sparse grid combination technique provides another way of obtaining a sparse grid representation of a given function by linearly combining multiple anisotropic full component grids. It grants a computational and implementation-wise benefit over working directly in the hierarchical basis, since there is no need to implement anything else than full grid operations and all computations on different component grids can be performed independently from each other, which enables parallelised approaches.
In the case of the standard combination technique, one considers all grids $\Omega_{\boldsymbol{l}}$ where $|\boldsymbol{l}|_1 = n-q$, $q \in [d-1] \cup \{0\}$ and $\boldsymbol{l} \geq 0$. Using the piecewise d-linear nodal basis functions $\phi_{\boldsymbol{l},\boldsymbol{j}}$ on any component grid $\Omega_{\boldsymbol{l}}$, the corresponding interpolant $f_{\boldsymbol{l}}$ of a function f on this grid is

$$f_{\boldsymbol{l}} = \sum_{\boldsymbol{j} \leq 2^{\boldsymbol{l}}} \alpha_{\boldsymbol{l},\boldsymbol{j}} \phi_{\boldsymbol{l},\boldsymbol{j}}.$$

These individual grid-wise interpolants are then combined in the following way, which is illustrated in figure 4 for $d = 2, n = 4$:

$$f_n^c := \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{|\boldsymbol{l}|_1 = n-q} f_{\boldsymbol{l}}. \tag{10}$$

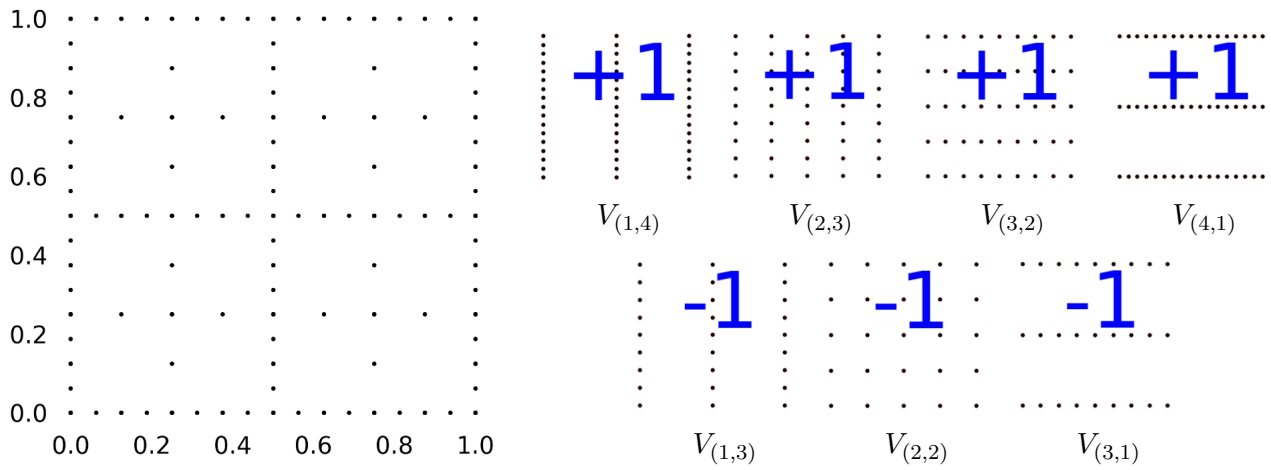


Figure 4: Example of a sparse grid in 2d generated by means of the standard combination technique (10) for $n = 4$, $d = 2$ assuming inhomogeneous boundary conditions. On the right, the component grids arising in the combination are displayed along with their coefficients (coloured in blue).

Note, that in the original article [7], only those grids $\Omega_{\boldsymbol{l}}$ that fulfil $|\boldsymbol{l}|_1 = n + (d-1) - q$, $q \in [d-1] \cup \{0\}$ and $\boldsymbol{l} > 0$ were considered for the combination procedure. Again, this is only a matter of definition, starting either with minimal level zero, as in (10) or with level one, as in [7]. It is a noteworthy fact, that the combined interpolant f_n^c (10) lies in $V_n^{(1)}$ and is identical to the sparse grid interpolant f_n [4, p.17]:

Lemma 1.6. *For any function $f : \Omega \to \mathbb{R}$, it holds $f_n^c = f_n \in V_n^{(1)}$.*

When turning to higher dimensions, it will become necessary to develop adaptive methods that consider special features of the problem at hand. As remarked in [11, p.18], the standard combination technique has been primarily utilised for dimensionally adaptive procedures.
In what follows, the so-called **truncated combination technique** (cf. [10, p.5]) will be used. It differs from the standard combination technique in the use of a truncation parameter $\boldsymbol{\tau}$, $\tau_k \in \mathbb{N}_0 \cup \{-1\}$ for each dimension $k \in [d]$. By defining the index sets

$$\mathcal{I}_{l,q} := \left\{ \boldsymbol{l} \in \mathbb{N}_0^d \middle| |\boldsymbol{l}|_1 = l + d - q - 1 + \sum_{i=1}^{d} \tau_i, \;\; \boldsymbol{l} > \boldsymbol{\tau} \right\}$$

for $n \in \mathbb{N}_0$ and $q \in [d-1] \cup \{0\}$, one enforces that each component grid that appears in the combination scheme

$$f_n^c = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{\boldsymbol{l} \in \mathcal{I}_{n,q}} f_{\boldsymbol{l}} \tag{11}$$

has a level of at least $\mathbb{1} + \boldsymbol{\tau}$. One derives the standard combination technique by setting $\boldsymbol{\tau} = 0$. The role of the truncation parameter $\boldsymbol{\tau}$ and the level l are illustrated in figure 5.

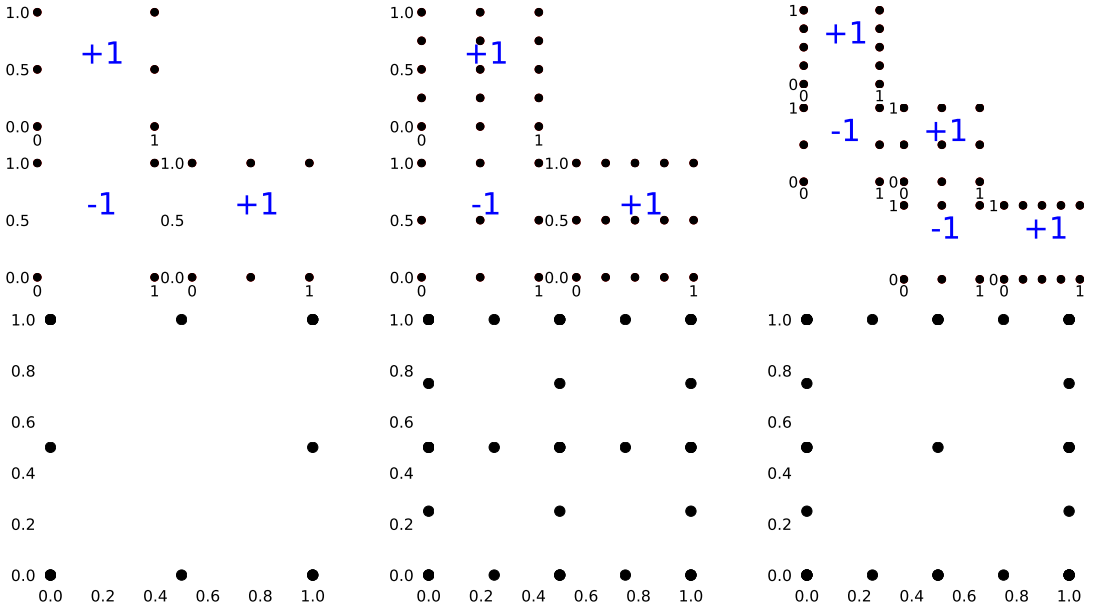


Figure 5: Grids arising from different truncation parameters $\boldsymbol{\tau}$ and levels l in the truncated combination technique (11) in two dimensions: Starting off with $\boldsymbol{\tau} = -\mathbb{1}$ and $l = 1$ on the left, increasing $\boldsymbol{\tau}$ to 0 and leaving the level constant yields the scheme depicted in the middle, whereas an increase in level to $l = 1$ and an unchanged truncation parameter result in the combination of grids on the right. All grid's coefficients are coloured in blue.

2 Adaptivity

If one needs to integrate a function which does not satisfy given smoothness requirements or shows great variation in its characteristics, adaptive refinement is one possible solution. As presented in the previous section, sparse grids are statically generated by making an a priori choice of those subspaces, which are the most relevant with regard to the interpolation error under certain smoothness conditions. Adaptive refinement aims at deciding a posteriori, which subspaces or grid points are to be refined using local error estimators. This section follows [6], [11] and [10].

2.0.1 Dimensional adaptivity

Not all subspaces arising in a combination contribute similarly to the overall approximation quality, which has been utilised in the construction of sparse grids. But instead of an a priori, statically optimal selection of a fixed simplicial subset of subspaces as in their construction, dimensionally adaptive approaches seek to identify and quantify important dimensions in which to extend the selection of subspaces. One theoretical foundation for such algorithms is given by Kolmogorov's superposition theorem, by means of which a high-dimensional function can be approximated by sums of lower-dimensional functions [6, p.3]. As an example, the so-called dimensionally adaptive quadrature by Gerstner and Griebel [6] will be presented briefly.
As mentioned above, the simplicial subspace selection of the sparse grid construction in (8) is relaxed in the following way: Up to now, only multi-indices $\boldsymbol{l} \in \mathbb{N}_0^d$ contained in the standard simplex have been considered, i.e. those that fulfilled $|\boldsymbol{l}|_1 \leq n + d - 1$. From now on, an index set $\mathcal{I} \subset \mathbb{N}^d$ is **admissible**, if

$$\forall\ \boldsymbol{l} \in \mathcal{I}, k \in [d] : l_k > 1 \Rightarrow \boldsymbol{l} - \boldsymbol{e}_k \in \mathcal{I}$$

holds, i.e., $\mathcal{I}$ is downward closed. Admissible index sets correspond to valid **generalised sparse**

grids, which are constructed via $\oplus_{\boldsymbol{l}\in\mathcal{I}} c_{\boldsymbol{l}}\Omega_{\boldsymbol{l}}$, where $c_{\boldsymbol{l}} := \sum_{\boldsymbol{i}\in\mathcal{I}:\boldsymbol{l}\leq\boldsymbol{i}\leq\boldsymbol{l}+\mathbb{1}}(-1)^{|\boldsymbol{i}-\boldsymbol{l}|_1}$ (cf. [10, p.5]). In accordance to (9), the conventional generalised sparse grid quadrature is defined as

$$Q_{\mathcal{I}}f := \sum_{\boldsymbol{k}\in\mathcal{I}}(\Delta_{k_1}\otimes\cdots\otimes\Delta_{k_d})f \tag{12}$$

for $f:\Omega\to\mathbb{R}$ and a sequence of quadrature formulas $(Q_l)_l$ similar to (9) (cf. [6, p.6]).
The basic idea of the algorithm is to iteratively add indices to $\mathcal{I}$ in such a way, that it remains admissible throughout the whole process and that the overall error made by approximating the analytic integral with the sum $\sum_{\boldsymbol{l}\in\mathcal{I}}\Delta_{\boldsymbol{l}}f$ of the differential integrals is reduced. An error estimate $g_{\boldsymbol{l}}$ is associated with each index $\boldsymbol{l}\in\mathcal{I}$, which may depend on $\Delta_{\boldsymbol{l}}f$, as well as other indicators, e.g. the number of function evaluations required for the computation of $\Delta_{\boldsymbol{l}}f$. The set of current indices $\mathcal{I}$ is partitioned into two sets $\mathcal{A}$ and $\mathcal{O}$, which are referred to as **active**, respectively **old** indices. The distinction between active and old is made as follows: $\mathcal{A}$ contains all $\boldsymbol{l}\in\mathcal{I}$, whose error estimates $g_{\boldsymbol{l}}$ have been computed, but none of the error estimates of their forward neighbours have. Whenever an index $\boldsymbol{l}$ is added to $\mathcal{I}$, it is indeed added to $\mathcal{A}$ and the error estimates $g_{\boldsymbol{k}}$ of its forward neighbours $\boldsymbol{k}\in\{\boldsymbol{l}+\boldsymbol{e}_i|i\in[d]\}$ are computed, since those are considered possible candidates for further additions to $\mathcal{I}$. The sum $\eta := \sum_{\boldsymbol{l}\in\mathcal{A}} g_{\boldsymbol{l}}$ of error estimates belonging to indices in $\mathcal{A}$ is used as an indicator for the global error. Initially, the procedure commences by setting $\mathcal{A}=\{\mathbb{1}\}$ and $\mathcal{O}=\emptyset$.
In each iteration, an index $\boldsymbol{l}$ which has the largest error estimate $|g_{\boldsymbol{l}}|$ among all active indices is moved to $\mathcal{O}$, its error estimate is subtracted from the global indicator η and all indices $\boldsymbol{k}$ corresponding to its admissible forward neighbours are added to $\mathcal{A}$ and have their respective error estimates $g_{\boldsymbol{k}}$ computed and added to η, while their differential integrals $\Delta_{\boldsymbol{k}}f$ are added to the current integral approximation r. If in some iteration the global error indicator η lies below a given tolerance ϵ, the algorithm stops.
Figure 6 provides an exemplary illustration of this dimensionally adaptive procedure in 2D, a pseudo-code implementation is given by algorithm 1 [6, p.7]. It is a noteworthy remark, that the sparse grid combination technique can also be used with generalised sparse grids (cf. [10, p.5]).

Algorithm 1: Dimension-adaptive quadrature [6]

```
1  O ← ∅;
2  A ← {𝟙};
3  r ← Δ_𝟙 f;
4  η ← g_𝟙;
5  while η > ϵ do
6  |   k ← argmax_{l∈A} |g_j|;
7  |   A ← A \ {k};
8  |   O ← O ∪ {k};
9  |   η ← η − g_k;
10 |   foreach i ∈ [d] do
11 |   |   k' ← k + e_i;
12 |   |   if ∀j ∈ [d] : k' − e_j ∈ O then
13 |   |   |   A ← A ∪ {k'};
14 |   |   |   r ← r + Δ_{k'} f;
15 |   |   |   η ← η + g_{k'};
16 |   |   end
17 |   end
18 end
19 return r;
```

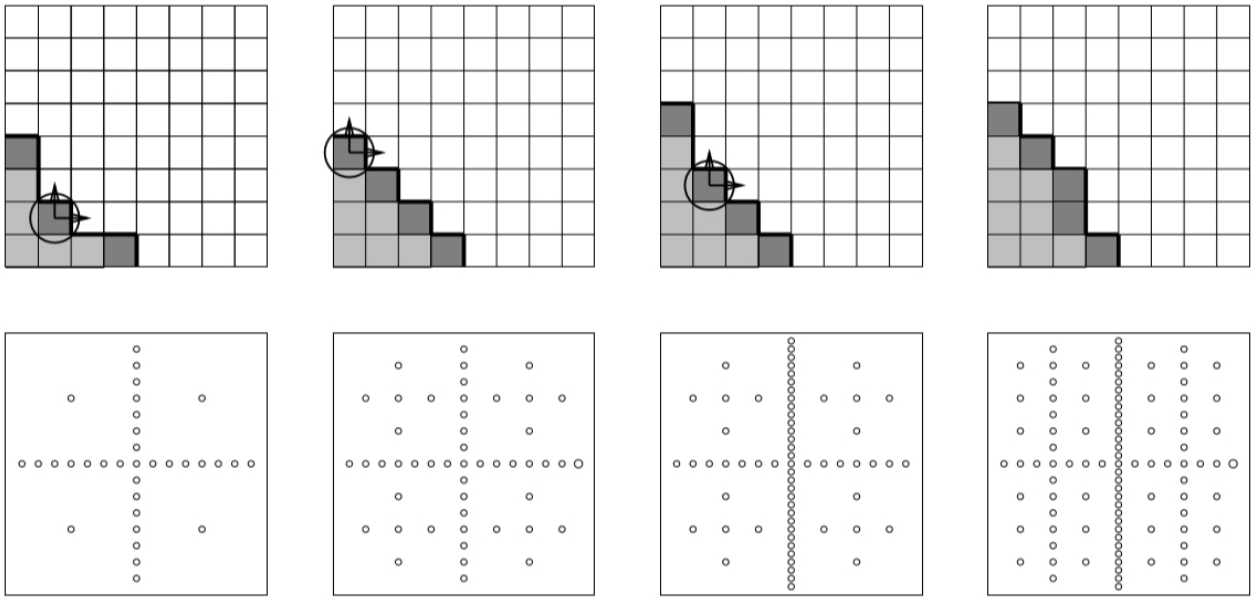


Figure 6: Dimensional adaptivity: Exemplary behaviour of algorithm 1 (taken from [6, p.9]): The first row depicts the different states of the index set $\mathcal{I}$, whose active/old indices are coloured dark/light grey, respectively. Encircled indices have largest error estimates among all active indices. The second row shows the corresponding generalised sparse grids. The midpoint rule has been used as underlying quadrature rule (cf. [6, p.9]).

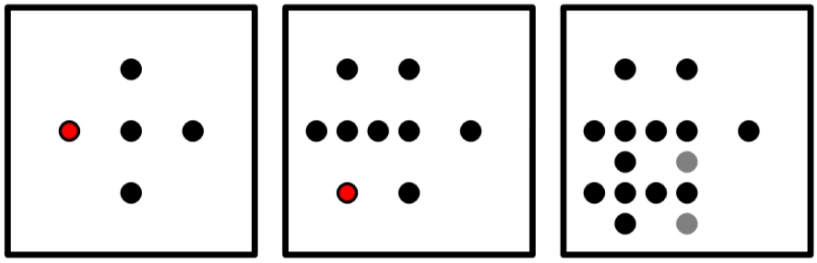


Figure 7: Successive spatial refinement steps of two red-coloured points in a regular grid using homogeneous boundary conditions. In the second refinement (middle), two points are added, whose hierarchical parents (coloured in grey) are not contained in the grid. Those are successively added, as indicated in the rightmost image (taken from [11, p.21]).

2.0.2 Spatial adaptivity

Here, the goal is to decide which points of a subspace would contribute the most and to subsequently add those, i.e., one estimates the error for each point and then decides whether to refine it or not. The refinement may consist of adding the hierarchical children from the given point's next level to the grid. This often requires special amendments, as the process of calculating the surpluses renders it necessary that all hierarchical parents of a given grid point need also be contained in the grid. Thus, those parents are often added recursively during the refinement of a given point, cf. figure 7. Whence, as remarked in [10, p.6], the total number of points added per refinement may exceed $2 \cdot d$.
Furthermore, as remarked in [11, p.22], it may be circumstantially better to refine more than one point per iteration.
The split-extend scheme developed in [10], on which the main focus will remain until the end of this thesis, is a representative of the class of spatially adaptive methods.

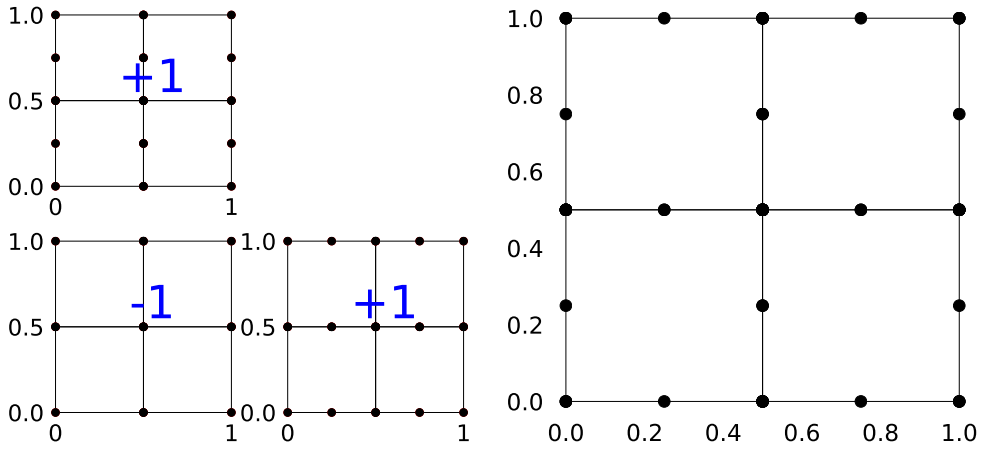


Figure 8: Initial refinement structure obtained using the truncated combination technique with $l = 1$, $\boldsymbol{\tau} = 0$. The combination coefficients are coloured in blue (taken from [10, p.8]).

2.1 The Split-Extend scheme

Introduced in [10], this approach combines the truncated sparse grid combination technique with spatial adaptivity, operating on block-adaptive grids. Even though this text restricts its use to quadrature and piecewise linear basis functions, the underlying algorithm and the existing **SpACE** framework [9] can as well be tailored onto multiple problem settings, e.g. approximatively solving partial differential equations.
One starts with an initial refinement structure which is given by the standard combination technique where $l = 1$ and $\boldsymbol{\tau} = 0$, cf. figure 8.
The grid's block structure is visualised by fully drawn lines and will be further referred to as the **refinement graph**. Individual blocks in the refinement graph are called **sub-areas** and each of them is identified via a unique index i, as well as a local level l_i and a local truncation parameter $\boldsymbol{\tau}_i$.
In sub-areas where the objective function's characteristics vary significantly, the resolution of the block-adaptive grid resulting from the combination will be gradually increased. The underlying idea of increasing the resolution of a sub-area of the grid in the truncated combination technique is to increase the resolution of that sub-area in each component grid used in the combination. However, as the refinement of a large sub-area with many grid points is rather expensive, one would first try to narrow down the area to be refined as far as possible.
There are two refinement operations defined in [10], namely **extensions** and **splits**. Both serve different purposes; the split steps are primarily used to narrow down the areas to be refined, while the extension steps' purpose lies in economically increasing the resolution of an area.
As will be pointed out in the following discussion of these operations, they correspond to the solution of a combination scheme for each sub-area i where its local truncation parameter $\boldsymbol{\tau}_i$ and local level l_i depend on and vary due to the choice of the actual refinement operations.
In the end of each refinement step (which may involve multiple sub-areas), one consistent combination scheme of (global) level l_{global} and truncation parameter $\boldsymbol{\tau}_{\text{global}}$ is generated from the local refinements of the sub-areas and each of the involved component grids is also defined over the whole domain.

2.1.1 The split operation

A split operation is performed on sub-area i by incrementing its truncation parameter $\boldsymbol{\tau}_i$: $\boldsymbol{\tau}_i := \boldsymbol{\tau}_i^{\text{old}} + \mathbb{1}$. This can be alternatively interpreted as splitting the sub-area into 2^d smaller congruent sub-areas, denoted by children(i), whose truncation parameter and structure is the

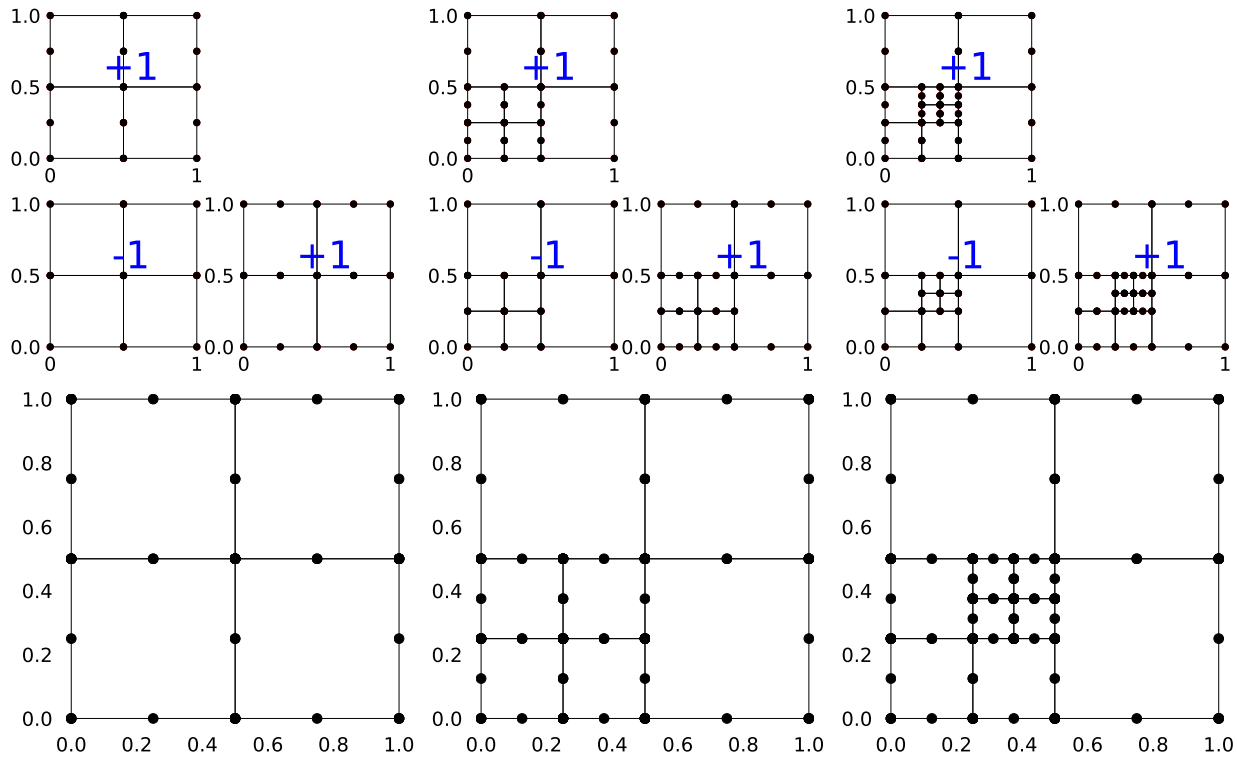


Figure 9: Exemplary split operations (taken from [10, p.9]), from left to right: Initial grid structure, grid structure after a split operation has been applied to the lower left sub-area and after the successive splitting of the newly created sub-area in the upper right corner of the previously refined sub-area.

same as that of i: $\boldsymbol{\tau}_j = \boldsymbol{\tau}_i^{\text{old}} = \boldsymbol{\tau}_i - \mathbb{1}$ for $j \in \text{children}(i)$. Furthermore i is defined as their parent, i.e. $\text{parent}(j) := i$ for $j \in \text{children}(i)$. Each split operation applied onto i doubles the number of grid points of i in each dimension. From a visual standpoint, the application of a split operation onto a sub-area corresponds to adding a grid patch to the sub-area, which has the same size as the sub-area and the same structure as the initial grid, cf. figure 9. Since split sub-areas are smaller in size, subsequent refinements are enabled to be performed more selectively with regard to the objective function's local characteristics. Among the disadvantages of the split operation one has the substantial increase in grid points per application as well as the fact that the truncation parameter being repeatedly incremented while the number of component grids is kept constant, would lead to a grid structure which increasingly resembles a full grid. As a remedy to this, the extend operation will be described in short notice.
One may view the initial structure of the grid as resulting from a split operation applied to the leftmost grid structure depicted in figure 5. Note that therefore all children j in the initial grid have the same truncation parameter $\boldsymbol{\tau}_j = -\mathbb{1}$. This implies that there will not occur any grids with truncation parameter 0, which makes it reasonable to set the truncation parameter in the global combination scheme to $\boldsymbol{\tau}_{\text{global}} := -\mathbb{1}$ (cf. [10, p.8]).

2.1.2 Single-dimensional splitting

As the split operation turns out to become quite expensive when applied extensively in higher dimensions, a single dimensional version has been implemented, which is the main contribution made in this thesis. The advantage of this restriction is that the total number of points added per split is decreased.
Basically, each sub-area i is associated to d other sub-areas, referred to as its twins. Additionally, the so-called twin error $\epsilon_{i,j} = |I_{l_i,i} - I_{l_j,j}|$, which is simply the modulus of the difference of their

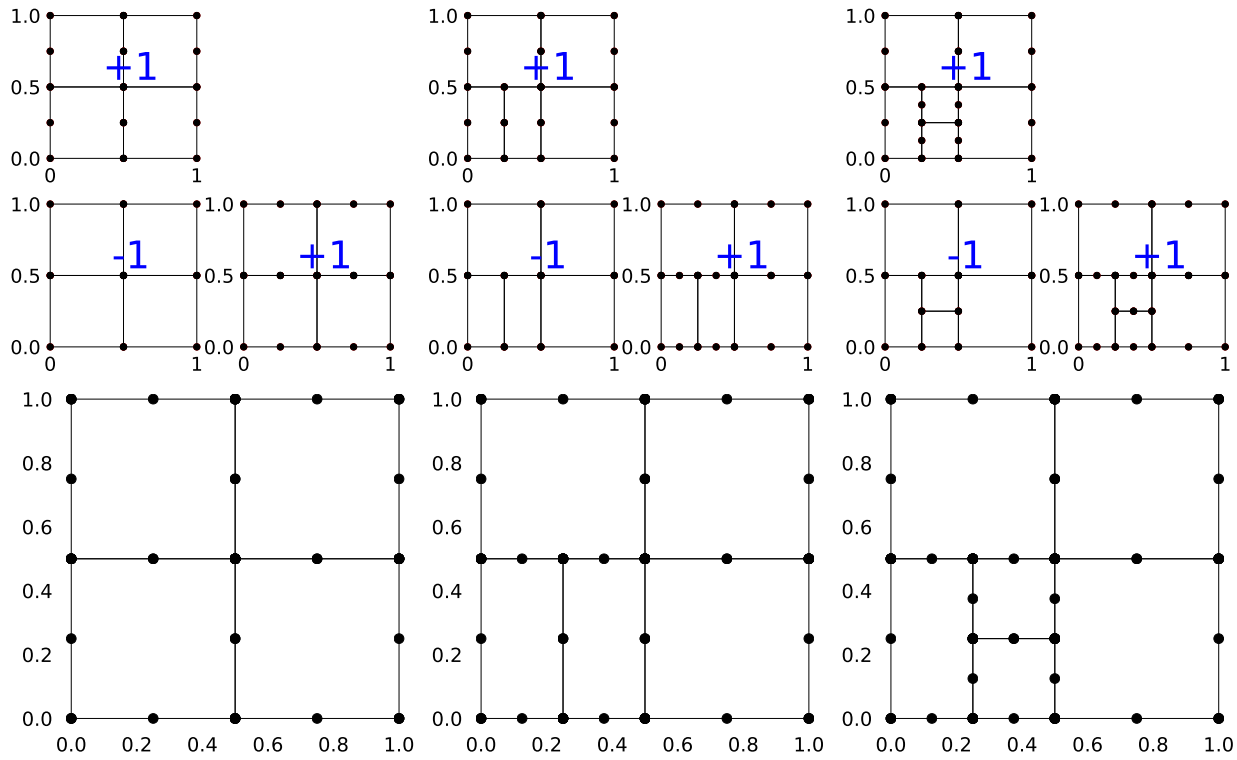


Figure 10: Exemplary single-dimensional split operations, from left to right: Initial grid structure, grid structure after a split operation in dimension 1 has been applied to the lower left sub-area and after the successive splitting in dimension 2 of the second newly created sub-area in right half of the previously refined sub-area.

integral approximations is stored for each twin j of i. In the initial refinement structure, all twins correspond to the actual neighbours in the respective dimension, i.e., a sub-area and its twin in dimension $k \in [d]$ share a facet.
Prior to a single dimensional split operation on i, the smallest dimension of maximal twin error, $d_{\max} := \operatorname{argmax}_{j \in [d]} \epsilon_{i,j}$ is determined. During the successive split step in dimension $d_{\max}$ applied on i, two instead of 2^d new children i_1, i_2 are created, cf. figure 10. As before, one sets $\text{parent}(i_1) = \text{parent}(i_2) = i$. While they inherit the set of twins in dimensions $k' \in [d] \setminus \{k\}$ from their parent, they are set to be each other's twin in dimension k.
It is quite important to notice, that the twin errors in dimensions $k' \in [d] \setminus \{k\}$ would still correspond to the parent's twin errors between areas twice as large. For this reason, to avoid unwanted behaviour like one single splitting dimension being dominantly preferred over all other refinement dimensions, each of those inherited twin errors is halved.
The residual functionality akin to error and benefit estimations has remained unchanged and the corresponding code is displayed in the following section.

2.1.3 The extend operation

Complementary to the split operation, the extend operation may increase the number of component grids and thus add new subspaces. This results from incrementing the level l_i of a sub-area i, i.e. $l_i := l_i^{\text{old}} + 1$, whereas its truncation parameter $\boldsymbol{\tau}_i$ remains unchanged. An exemplary step is depicted in figure 11.
Since one reclusively modifies local levels, there might exist different sub-areas that each define an overall inconsistent number of component grids due to their possibly unequal local levels. Therefore all the schemes corresponding to the modified sub-areas are merged while employing the so-called **coarsening** process. As pointed out in [10], this would not be necessary if the

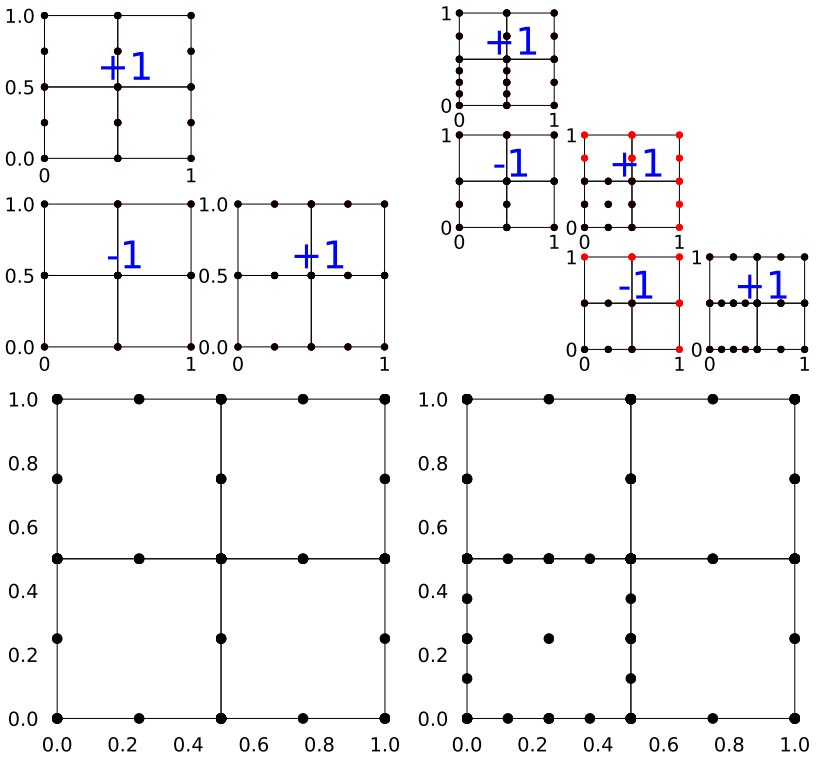


Figure 11: Exemplary extension operation (taken from [10, p.11]), from left to right: Starting from the initial grid structure ($l_i = 1$ for all $i = 0, \ldots, 3$) an extend operation is applied to the lower left sub-area: Its local level is increased to 2, which sets the global level $l_{\text{global}} := 2$ and increases the number of component grids in the global combination scheme to 5. As the remaining sub-areas have strictly smaller local levels, they need coarsening. All grid points coloured in red are not included in the final quadrature, as they only serve the purpose of having a grid covering the whole domain while their contributions have already been accounted for by the other component grids in the global combination scheme.

procedure were only to be used for solving quadrature problems, as the integral is a linear operator. Since the procedure has been conceived to be of wider application, the process has been included nonetheless.
When merging, the global combination scheme's truncation parameter is set to $\boldsymbol{\tau}_{\text{global}} := -\mathbb{1}$, its level is defined as the maximum level occurring among all sub-areas to be refined, i.e. $l_{\text{global}} := \max\{l_i | i \in \text{refinement}\}$. Subsequently, one iterates over all sub-areas for each component grid in the global combination scheme and coarsens the level vector $\boldsymbol{l}$ of the currently regarded component grid for each sub-area i whose local level l_i is strictly smaller than l_{global} by decrementing a largest entry of $\boldsymbol{l}$ $(l_{\text{global}} - l_i)$ times, which yields a coarsened level vector $\boldsymbol{l}_{i,\text{coarse}}$ for each sub-area i. Thus, a list of coarsened level vectors is generated for each sub-area i, one per component grid in the global combination scheme.
As several component grids might generate the same coarsened level vector, and it is undesirable to include the corresponding points' contribution multiple times, only one of those is included in the final combination. Similarly to generalised sparse grids, level vectors $\boldsymbol{l}_{i,\text{coarse}}$ with negative entries are deemed invalid and are therefore excluded. The given quadrature rule uses only those points corresponding to the selected coarsened level vectors $\boldsymbol{l}_{i,\text{coarse}}$.

2.1.4 Error estimators

In order to decide which sub-areas to refine next, each sub-area in the refinement graph is assigned a scalar error indicator. All those sub-areas whose error indicator lies within a certain

margin of the maximum error indicator in the present refinement structure are chosen for refinement.
In [10], an error estimator ϵ for the split-extend scheme is defined via $\epsilon_i := |I_{\text{old},i} - I_{\text{new},i}|$ for any sub-area i, which compares the current integral approximation $I_{\text{new},i}$ on i with a previous approximation $I_{\text{old},i}$. In particular, $I_{\text{new},i} := I_{l_i,i}$ denotes the approximation which results from the evaluation of the quadrature rule corresponding to the current refinement structure of sub-area i of local level l_i.
$I_{\text{old},i}$ is determined by the operation involved in the creation of sub-area i, i.e. $I_{\text{old},i} \in \{I^{\text{split}}_{\text{old},i}, I^{\text{extend}}_{\text{old},i}\}$. If it resulted from an extension, which would correspond to an increment of the local level l_i and an unchanged refinement graph, one would have $I_{\text{old},i} = I^{\text{extend}}_{\text{old},i} = I_{l_i-1,i}$.
On the other hand, if it is a newly created child of a sub-area parent(i) resulting from a split operation, there are no previous approximations available, whence a previous integral approximation of sub-area parent(i) of level l_i, denoted by $I_{l_i,\text{parent}(i)}$, is considered:

$$I_{l_i,\text{parent}(i)} := \sum_{\boldsymbol{k} \in \text{points}(l_i,\text{parent}(i))} w_{\boldsymbol{k}} f(x_{\boldsymbol{k}})$$

Herein points$(l_i, \text{parent}(i))$ is an abbreviation for the set of indices of all those points, which sub-area parent(i) would contain, if its local level were equal to l_i. In the first approach developed in [10], one calculates the previous approximation for sub-area i by restricting the above sum to those points $x_{\boldsymbol{k}}$ which lie in sub-area i, henceforth referred to as $x_{\boldsymbol{k}} \in \text{area}(i)$:

$$I^{\text{split}1}_{\text{old},i} := \sum_{\boldsymbol{k} \in \text{points}(l_i,\text{parent}(i)) \cap \text{area}(i)} w^*_{\boldsymbol{k}} f(x_{\boldsymbol{k}}) \tag{13}$$

The corresponding weight $w^*_{\boldsymbol{k}}$ of a point $x_{\boldsymbol{k}}$ only differs from $w_{\boldsymbol{k}}$ if it lies on the boundary of sub-area i, as it may thus be contained in more than one sub-area originating from parent(i). As such, its weight is divided uniformly among all children of parent(i) containing the respective point, which breaks down to setting

$$w^*_{\boldsymbol{k}} := \frac{w_{\boldsymbol{k}}}{|\{i \in \text{children}(\text{parent}(i)) | \boldsymbol{k} \in \text{area}(i)\}|}.$$

A second approach would consist in splitting the parent's previous integral approximation uniformly among all children, yielding

$$I^{\text{split}2}_{\text{old},i} := I_{l,\text{parent}(i)}/2^d. \tag{14}$$

The third approach presented in [10] is restricted to the case of piecewise linear basis functions, as the objective function's evaluations in the set points$(l_i, \text{parent}(i))$ are utilised to bilinearly interpolate at grid points in sub-area i, which in general would not preserve the order of approximation attained by using higher order basis functions. Thus

$$I^{\text{split}3}_{\text{old},i} := I_{\Gamma_{l_i,i}(l_i,\text{parent}(i))}, \tag{15}$$

where $\Gamma_{l_i,i}(l_i, j)$ is defined as the operator that linearly interpolates function values at grid points in component grids of sub-area i using function values at points belonging to sub-area j.
Moreover, it is a noteworthy fact, that this third approach may require additional function evaluations, as all grid points corresponding to level l_i in sub-area parent(i) are considered. As remarked in [10, p.12], the most promising error estimates based on (13), (14) and (15) for piecewise linear basis functions resulted from combining (13) and (15) by taking their minimum, i.e.,

$$I^{\text{split}}_{\text{old},i} := \min\{I^{\text{split}1}_{\text{old},i}, I^{\text{split}3}_{\text{old},i}\} \tag{16}$$

In general, when using higher order quadrature rules, respectively basis functions, bilinear interpolation or filtering approaches would not preserve the underlying order and thus other error estimates have to be employed.
With respect to piecewise linear basis functions, each sub-area i has now been assigned an error estimate ϵ_i. The benefit β_i of refining this sub-area is defined by $\beta_i := \frac{\epsilon_i}{|\text{points}(i,l_i)|}$ [10, p.13]. The sub-area which offers the largest benefit as well as all those sub-areas whose benefit is at least $\beta_i\gamma$ for some fixed parameter $\gamma \in \,]0,1[$ are chosen for the proximal refinement.

2.1.5 Split or extension?

If a sub-area has been selected as a candidate for refinement, one ought to make a reasonable guess, which of the two previously compiled strategies would prove more suitable. Splits are typically useful when the objective function's characteristics require a focus on small subsets of the domain, e.g. when the function is zero on most of the domain, but has a peak in a corner; it would not be of much use to increase the global resolution of the grid but only that of a small sub-area which contains the peak.
As to the aforementioned guess, [10] estimate the error reduction due to a split and an extension per sub-area to be refined and consequently choose the operation which yields better results. If sub-area i is to be refined, the approximation $I_{l_i-1,\text{parent}(i)}$ is used as a reference value for the analytic integral on sub-area parent(i).
Note that sub-area i, which has the associated integral approximation $I_{l_i,i}$, can be obtained from sub-area parent(i), provided that is has local level $l_i - 1$, by executing one split and one extension. $I_{l_i-1,\text{parent}(i)}$ is always available, as the initial refinement structure with $\boldsymbol{\tau}_{\text{global}} = 0$ results from a split and the local level l_i of sub-area i is positive, as $l_i > \tau_{\text{global},i}$ [10, p.13].
If a split step is applied to parent(i), assuming it has local level $l_i - 1$, the approximation $I_{l_i-1,i}$ for i is compared relatively to the current approximation $I_{l_i,i}$:

$$\epsilon_i^{\text{split}} := \frac{|I_{l_i,i} - I_{l_i-1,i}|}{|I_{l_i,i}|}. \tag{17}$$

As extension steps leave the refinement graph unmodified and in most cases add fewer points to the grid than splits, filtering and bilinear interpolation are used once more for piecewise linear basis functions to define $\epsilon_i^{\text{extend}}$.
Since among consecutive extensions the first tends to be the most effective with regard to error reduction [10, p.14], k steps are performed instead of just one, such that k is maximum while satisfying

$$3 \cdot |\Pi_i(l_i - 1 + k, \text{parent}(i))| \geq |\text{points}(l_i - 1, i)|.$$

$\Pi_i(l, j)$ denotes the operator that outputs the set of all grid points in sub-area j, assuming it has local level l, that are also contained in sub-area i [10, p.12]. As remarked in [10, p.14], such a k is typically maximum in the way that the number of points added in the k extension steps applied on parent(i), i.e. $|\Pi_i(l_i - 1 + k, \text{parent}(i))|$, is strictly smaller than the number of points that would be added due to a split step applied on parent(i), i.e. $|\text{points}(l_i - 1, i)|$. The third approach (15) of bilinear interpolation is included as well and once again the minimum of both relative errors is taken, i.e.,

$$\epsilon_i^{\text{extend}} := \frac{\min\{|I_{l_i,i} - \Pi_i(l_i - 1 + k, \text{parent}(i))|, |I_{l_i,i} - \Gamma_{l_i,i}(l_i - 1 + k, \text{parent}(i))|\}}{|I_{l_i,i}|} \tag{18}$$

Using the convenient definitions

$$\begin{aligned} n_i^{\text{split}} &:= |\text{points}(l_i - 1, i)|, \\ n_i^{\text{extend}} &:= |\Pi_i(l_i - 1 + k, \text{parent}(i))|, \\ n_i^{\text{reference}} &:= \begin{cases} |\text{points}(l_i - 1, \text{parent}(i))| : \text{if the grid is nested} \\ 0 : \text{otherwise} \end{cases} \end{aligned}$$

made in [10, p.14], the operation's costs on sub-area i are given as products of errors and points that are to be spent by the respective operation: $\alpha_i^\theta := \epsilon_i^\theta(n_i^\theta - n_i^{\text{reference}})$, $\theta \in \{\text{split}, \text{extend}\}$. The algorithm then performs the operation which has lower estimated cost.
Analogously to the previously discussed error estimates with regard to sub-areas i, filtering and bilinear interpolation are not combined with higher order quadrature formulas, respectively basis functions, and require different cost estimators.

2.1.6 The algorithm

The input consists of:

- Vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d$ which encode the endpoints of the intervals that define the rectangular domain Ω,
- a tolerance $\epsilon > 0$,
- the objective function $f : \Omega \to \mathbb{R}$ and
- an optional reference solution for $\int_\Omega f(x)dx$.

In the algorithm, one first initialises the level $l_{\text{global}} := 1$, the truncation parameter $\boldsymbol{\tau}_{\text{global}} := -\mathbb{1}$ and finally constructs the global combination scheme. Then, the initial refinement structure as depicted in figure 8 is created by a split operation, whence all sub-areas i originating from this process have local level $l_i = 1$ and truncation parameter $\boldsymbol{\tau}_i = -\mathbb{1}$. Subsequently, the integral approximation corresponding to the current refinement structure is calculated by iteratively accumulating the respective contributions of all sub-areas in the refinement graph.
As several sub-areas may have local levels strictly smaller than l_{global}, level vectors of certain component grids might require coarsening. Thus for each sub-area the coarsened level vectors of all component grids are calculated and utilised to derive the integral approximation on it, using a predefined grid, which defines for each dimension $k \in [d]$ (cf. [10, p.15]):

- A quadrature formula (e.g. Newton-Cotes),
- the number of points n_k,
- the position p_j for $j \in [n_k]$ and
- the weights $w_{j,k}$

that correspond to level l_k.
In [10, p.15] the assumption is made, that uniform grids as well as $2^{n_k} + 1$ points are used in all dimensions k, which conforms to the notions introduced in section 1. According to the tensor product approach also described there, the weight corresponding to the point $p_{\boldsymbol{l},\boldsymbol{j}}$ is calculated as the product $w_{\boldsymbol{l},\boldsymbol{j}} := \prod_{k \in [d]} w_{j_k,k}$. As the integral approximations for each sub-area and component grid have been calculated in this fashion, they are combined with regard to the global combination scheme, i.e., scaled with the component grid's coefficient $c_{\boldsymbol{l}}$ and added up. The result of this step, the integral approximation for the current refinement structure, is compared to the value obtained by evaluating the reference solution on the current refinement structure. If no reference solution has been specified, the sum of all hierarchical surpluses is used to estimate the error [10, p.16].

If it is smaller than the given tolerance ϵ, the algorithm terminates. In the converse situation, the sub-area i of highest expected benefit β_i is specified and all those sub-areas, whose estimated benefit exceeds $\beta_i\gamma$, are selected for refinement after their current integral approximation is removed from the temporary result. For each of those sub-areas i, the expected costs $\alpha_i^{\text{split}}, \alpha_i^{\text{extend}}$ are calculated and subsequently the cheaper operation is performed. In succession to the completion of the refinement, the previous outdated sub-areas are discarded. The following pseudo-code representation of the algorithm (cf. [10, p.16]) concludes this section.

Algorithm 2: Split-Extend scheme

Data: Rectangular domain $\Omega \subset \mathbb{R}^d$ given by $\boldsymbol{a}, \boldsymbol{b}$, tolerance $\epsilon > 0$, refinement parameter $\gamma \in \,]0, 1]$, objective function f, (optional) reference solution
Result: Approximation of $\int_\Omega f(x)dx$

```
l_global ← 1;
τ_global ← −𝟙;
Initialise global combination scheme with l_global, τ_global;
Create 2^d sub-areas i having l_i = 1, τ_i = −𝟙 in the initial split;
do
    integral ← 0;
    foreach component grid g ∈ combination scheme do
        foreach sub-area a in the current refinement do
            levelvec ← Coarsened level vector of g with regard to a;
            if levelvec ≥ 0 and not duplicate then
                integral += g.integrate(f, a, g) · g.coefficient;
            end
        end
    end
    error ← |integral−reference| / |reference|;
    β_max ← max{β_i | i ∈ refinement};
    subareas ← {i | β_i ≥ γβ_max};
    foreach sub-area a ∈ subareas do
        Calculate costs α^split, α^extend;
        if α^extend > α^split then
            if splitSingleDim then
                d_max ← argmax_{k∈[d]} |a.twinerrors[k]|;
                children ← a.split_single_dim(d_max);
            else
                children ← a.split();
            end
            foreach j ∈ children do
                l_j ← l_a;
                τ_j ← τ_a;
            end
        else
            a_new ← a.extend();
            l_a_new ← l_a + 1;
            τ_a_new ← τ_a;
        end
    end
while error ≥ ε;
```

3 Implementation

In this section, all contributions to **SpACE** [9] concerning the single dimensional split are compiled and their functionality is explained briefly. When the need arises, previously existing constructs are thematised as well.

3.1 Grids in SpACE

Grids in general are implemented as subclasses of

```
class Grid(object):
```

which has among its attributes a boolean flag **boundary**, which indicates whether one has grid points on the boundary or not, as well as two d-dimensional vectors **a**, **b**, which define the borders of the domain $\Omega = [a_1, b_1] \times \cdots \times [a_d, b_d]$ (not restricted to $[0, 1]^d$). A few selected member functions, which offer the functionality listed in the previous section, are

```
def getCoordinate(self, indexvector)
```

which returns the Cartesian coordinates of a point $\boldsymbol{x}_{\boldsymbol{l},\boldsymbol{j}}$ when supplied with the index vector $\boldsymbol{j}$ and

```
def get_weights(self)
```

which returns all weights according to the underlying quadrature rule, which are defined by the subclasses of the **Grid** class. Several types of grids are supported by the framework, e.g. **TrapezoidalGrid** and **ClenshawCurtisGrid**, which correspond to equidistant mesh combined with the trapezoidal rule and to Clenshaw-Curtis quadrature on Tchebychev abscissas, respectively.

3.2 Contributions regarding the single-dimensional split

The instances of class **RefinementObjectExtendSplit**, which inherits from the **RefinementObject** class implemented to model sub-areas, correspond to the type of sub-areas utilised in the split-extend scheme.

```
1  class RefinementObjectExtendSplit(RefinementObject)
2     def __init__(self, start, end, grid, ... , splitSingleDim=True):
3        ...
4        # twin array
5        self.twins = [None] * self.dim
6        ...
7        # twin errors
8        self.twinErrors = [None] * self.dim
9        ...
```

Its constructor has been supplied with an additional boolean parameter **splitSingleDim** (line 1) which indicates whether single-dimensional or full-dimensional splits are to be employed. In the constructor's body, two arrays **twins** and **twinErrors** of size d are initialised with the **None** data type (lines 5 and 8). They will be used to store a reference to the refinement object's twin in each dimension, respectively the error that arises from the comparison of their integral approximations.
The next addition to the code consists of the method which performs a split in a single dimension on the given refinement object, namely

```
1  def split_area_single_dim(self, d):
2     midpoint = self.grid.get_mid_point(self.start[d], self.end[d], d)
3     sub_area_array = []
4     for i in range(2):
5        start_sub_area = list(self.start)
6        end_sub_area = list(self.end)
7        start_sub_area[d] = start_sub_area[d] if i == 0 else midpoint
8        end_sub_area[d] = midpoint if i == 0 else end_sub_area[d]
9        parent_info = ErrorInfo(parent=self, last_refinement_split=True)
10       new_refinement_object = RefinementObjectExtendSplit(
11                       start=start_sub_area,
12                       end=end_sub_area, grid=self.grid,
13                       self.numberOfRefinementsBeforeExtend,
14                       parent_info=parent_info,
15                       coarseningValue=self.coarseningValue,
16                       needExtendScheme=self.needExtendScheme,
17                       automatic_extend_split=self.automatic_extend_split,
18                       splitSingleDim=self.splitSingleDim)
19       new_refinement_object.twins = list(self.twins)
20       new_refinement_object.twinErrors =
21               list([t * 0.5 if t is not None else t for t in self.twinErrors])
22       new_refinement_object.twinErrors[d] = None
23       self.children.append(new_refinement_object)
24       sub_area_array.append(new_refinement_object)
25 sub_area_array[0].set_twin(d, sub_area_array[1])
26 return sub_area_array
```

The argument **d** in line 1 indicates the split dimension. One commences by calculating the mean of start and end point in the dimension specified by **d** in line 2. Subsequently, one defines the two children of the given sub-area resulting from the split by their start and end points in lines 7 to 8. Apart from the **d**'th coordinate, those remain unchanged.
One proceeds by instantiating a new refinement object with the given boundary data, adding it to the **sub_area_array** (lines 10-18). The variable **parent_info** contains mainly error estimates and properties like the operation's benefits, local level of the parent, the type of the last operation performed on it and so forth. The d-dimensional vectors **start_sub_area** and **end_sub_area** define the borders of the refinement object, i.e., $[s_1, e_1] \times \cdots \times [s_d, e_d]$. The integer **number_of_refinements_before_extend** states, how many split operations may be performed on this refinement object, until only extensions are used in further steps. **coarseningValue** indicates, how often the underlying sub-area needs to be coarsened, according to the combination scheme, while the number of splits performed on the parent is stored in **needExtendScheme**. **automatic_extend_split** is a boolean parameter, which enables or disables the automatic decision-making, which refinement operation to apply.
The newly created refinement object's twins are simply inherited from the parent, except for the **d**'th entry, which will be set after both children have been created. Since python would just store references to the existing arrays instead of creating new ones, which could in turn cause inconsistencies and overwriting, list constructors are applied (line 19). As has been mentioned in the previous section, each of the inherited twin errors is halved, in order to avoid imbalanced behaviour regarding the choice of the split dimensions (lines 20-21).
Finally, the newly created refinement object is added to its parent's list of children and to the set of newly created refinement objects to be returned (line 24).
Now, each of the children is set to be the other's twin in dimension **d** and they are returned in

the **sub_area_array** (lines 25-26).
This method is followed up by a few short helper methods, which have already been made use of in the previous method:

```
1  def set_twin(self, d, twin):
2      self.twins[d] = twin
3      twin.twins[d] = self
```

set_twin receives a dimension **d** and a reference to another refinement object, which is defined as the twin in dimension **d** of the refinement object on which it is called.

```
1  def set_twin_error(self, d, twinError):
2      twin = self.twins[d]
3      twin.twinErrors[d] = self.twinErrors[d] = twinError
```

set_twin_error sets the twin error of **self** and its twin in dimension **d** to the given value **twinError**.

```
1  def get_split_dim(self):
2      return np.argmax(self.twinErrors)
```

get_split_dim returns the dimension in which the twin error is maximum. If it is not unique, the first occurrence is returned.
The second noteworthy change occurred in the class **SpatiallyAdaptiveExtendScheme**, which comprises substantial parts of the functionality needed for the split-extend scheme. Again, a boolean parameter indicates, if full- or single-dimensional splits are performed.

```
1  class SpatiallyAdaptiveExtendScheme(SpatiallyAdaptivBase):
2      def __init__(self, a, b, ... , split_single_dim=True, operation=None, ...):
3          ...
4          self.split_single_dim = split_single_dim
5          ...
```

Among the offered functionality, one has the initial splitting of the domain performed in the method **initialize_refinement**. As the full-dimensional split neither required to set the twins accordingly, nor to initialise the twin errors, these features have both been added.

```
1  def initialize_refinement(self):
2      ...
3      if self.noInitialSplitting:
4      ...
5      else:
6          self.root_cell = RefinementObjectExtendSplit(
7                           np.array(self.a),
8                           np.array(self.b),
9                           self.grid,
10                          self.numberOfRefinementsBeforeExtend,
11                          None, 0, 0,
12                          automatic_extend_split=self.automatic_extend_split,
13                          splitSingleDim=self.split_single_dim)
14         if self.split_single_dim:
15             new_refinement_objects = [self.root_cell]
16             for d in range(self.dim):
17                 temp = []
```

```
18              for area in new_refinement_objects:
19                 temp.extend(area.split_area_single_dim(d))
20              new_refinement_objects = temp
21              for area in new_refinement_objects:
22                 integral = 0
23                 for component_grid in self.scheme:
24                    integral_area, a, b = self.evaluate_area(
25                                   self.f,
26                                   area,
27                                   component_grid)
28                    integral += integral_area * component_grid.coefficient
29                 area.integral = integral
30           for i in range(2**self.dim):
31              area = new_refinement_objects[i]
32              for d in range(self.dim-1):
33                 twin =
34                  new_refinement_objects[(i+2**(self.dim-1)) % 2**(self.dim-d)]
35                 area.set_twin(d, twin)
36                 if area.twinErrors[d] is None:
37                    area.set_twin_error(d, abs(area.integral - twin.integral))
38              if area.twinErrors[self.dim-1] is None:
39                 area.set_twin_error(
40                         self.dim-1,
41                         abs(area.integral - area.twins[self.dim-1].integral))
42              area.parent_info.parent = self.root_cell
43     else:
44        new_refinement_objects = self.root_cell.split_area_arbitrary_dim()
45     self.refinement = RefinementContainer(
46                              new_refinement_objects,
47                              self.dim,
48                              self.errorEstimator)
```

The extended else branch commences with the creation of a refinement object named **root_cell**, which corresponds to the original domain prior to the initial split (lines 6-13). An array **new_refinement_objects** is initialised which initially only contains **root_cell**. It is however designated to hold the soon to be created refinement objects corresponding to the initial refinement structure.
For each dimension, one after another, a split in this dimension is performed on all refinement objects currently accumulated in the temporary array **temp** (lines 18-19) and the resulting refinement objects replace the former content of **new_refinement_objects** (line 20). After all d dimensions have been iterated through, all refinement objects resulting from the initial split of the domain have been created. Each of those refinement objects is then assigned an approximation of the objective function's integral on the corresponding sub-area with regard to the global combination scheme by successively adding up all contributions of the component grids of this sub-area (lines 21-29).
Only in this initial split, all twins are set to be the actual neighbours in each dimension and all twin errors are exactly calculated, which is achieved by the indexing in lines 32 to 41. This is considered to be of importance, as the first choices of splitting dimensions may affect later decisions in the procedure.
In the full-dimensional case, i.e., **split_single_dim** being **false**, a regular split is performed on the initial domain (lines 43-44). Finally, the refinement objects created by either one of the

split strategies set as new refinement (lines 45-48).
As noted in the discussion of the method **split_area_single_dim**, one assigns **None** to those components of the twin error array to indicate that they require an update, if single-dimensional splits are utilised. This update is realised in the method **do_refinement**, which is also a member of class **RefinementObjectExtendSplit**:

```
1 def do_refinement(self, area, position):
2     ...
3     if self.split_single_dim:
4         for d in range(area.dim):
5             if area.twinErrors[d] is None:
6                 twinError = self.operation.get_twin_error(d, area)
7                 area.set_twin_error(d, twinError)
8     ...
```

In case of single-dimensional splits (line 3), one simply assigns current twin errors to those dimensions in $[d]$, which have been assigned **None** due to the assignment of new twins during a previous split operation (lines 4-7). For structural sake, the corresponding functionality of calculating twin errors has been moved to a subclass of **GridOperation**, which itself has mostly abstract members:

```
class GridOperation(object):
```

This class' use is by no means restricted to integration, which again hints at the flexibility of the **SpACE** framework. A subclass contains the method **get_twin_error**, which is called to calculate the twin error of a refinement object i in dimension **d** by comparing $I_{l_i,i}$ to $I_{l_j,j}$, if j corresponds to i's twin in dimension **d**:

```
1 def get_twin_error(self, d, area):
2     return abs(area.integral - area.twins[d].integral)
```

4 Testing

In this last section, the performance of the modified split-extend scheme featuring single-dimensional split operations is evaluated by means of several test cases, some among those presented in [10]. The underlying functions are part of a selection included in Genz' package of test functions [5], as well as one example from [8]. The chapter is concluded by an overall summary of the results, as well as an outlook on potential future improvements of the approach.

4.1 Test functions

The general definitions of the functions $f_1, \dots, f_5 : [0,1]^d \to \mathbb{R}$ are given in the following paragraphs. Plots for particularly parameterised representatives are depicted in figure 12. Comparisons between full- and single-dimensional splitting approaches for two-dimensional cases are restricted to a visual inspection of the corresponding sparse grid structures generated by the schemes displayed in figures 13, 14, 15 and 16. Error plots with regard to the 2-dimensional grids have been arranged in figure 17. All results of higher-dimensional test cases have been compiled in figure 18, figure 19 and figure 20. To comply with the previous sections, only linear basis functions have been utilised. Throughout the whole section, the tolerance will be denoted by ϵ, conforming to the notation in section 2. Moreover, $\delta \in \mathbb{N}_0$ denotes the maximum splitting level, if such a restriction has been set beforehand.

$$f_1(\boldsymbol{x}) := \begin{cases} 0 : x \geq p \\ e^{-\sum_{i=1}^{d} c_i x_i} : \text{otherwise} \end{cases} \tag{19}$$

f_1 is a discontinuous function, as it behaves like a positive exponential on $[0, p_1] \times \cdots \times [0, p_d]$, while it is identically zero on the complement of this set in $[0,1]^d$. The components of $\boldsymbol{c} \in \mathbb{R}^d$ determine the overall growth/decay in the respective coordinates. Exemplary plots ($d = 2$) for $p = \binom{0.3}{0.3}$ and $c = \binom{1}{2}$, respectively $p = \binom{0.6}{0.6}$ and $c = \binom{10}{1}$ are depicted in figure 12 (a), respectively (b). The block-adaptive sparse grids arising from both schemes are displayed and discussed in figures 13.

$$f_2(\boldsymbol{x}) := e^{-\sum_{i=1}^{d} c_i |x_i - p_i|}, \quad f_3(\boldsymbol{x}) := e^{-\sum_{i=1}^{d} c_i^2 (x_i - p_i)^2} \tag{20}$$

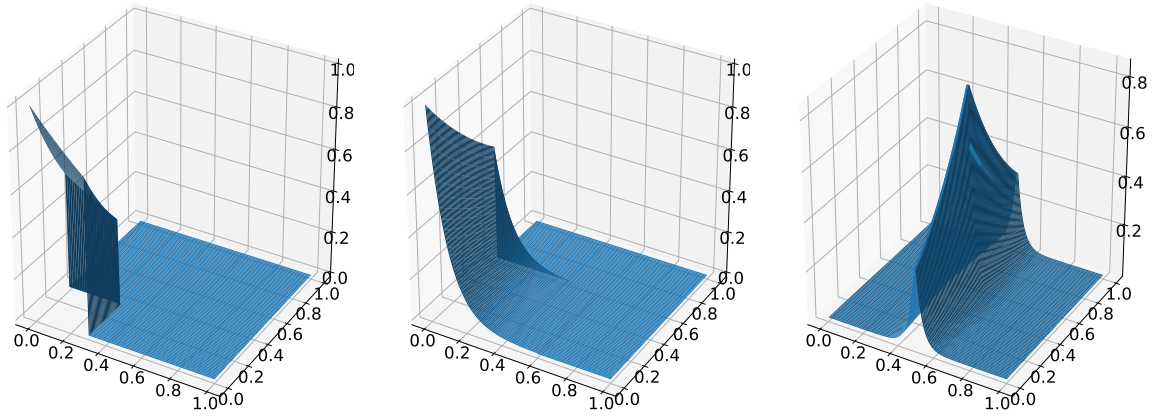
f_2 is a continuous, radially symmetric exponential centered at $p \in [0,1]^d$. $\boldsymbol{c}$ controls its growth/decay. f_3 has similarities to a Gaussian centered at p. In contrast to f_2, it is either constant or gradually decaying for increasingly large arguments and any choice of c. Additionally, it is an element of $C^\infty(\mathbb{R}^n)$. In figure 12, f_2 has been plotted for $p = \binom{0.5}{0.5}$, $c = \binom{30}{2}$ in (c), $p = \binom{0.2}{0.2}$, $c = \binom{10}{1}$ in (d) and $p = \binom{0.5}{0.5}$, $c = \binom{10}{20}$ in (e). f_3 has been visualised for $p = \binom{0.5}{1}$, $c = \binom{1000}{11}$ in figure 12 (f) and for $p = \binom{0.5}{0.5}$ $c = \binom{200}{0.1}$ in figure 12 (g). The grid structure resulting from the split-extend scheme have been collected in figures 14 and 15.

$$f_4(\boldsymbol{x}) := \frac{10^{-d}}{\prod_{i=1}^{d} \left(\frac{1}{x_i^2} + (x_i - p_i)^2 \right)} \tag{21}$$

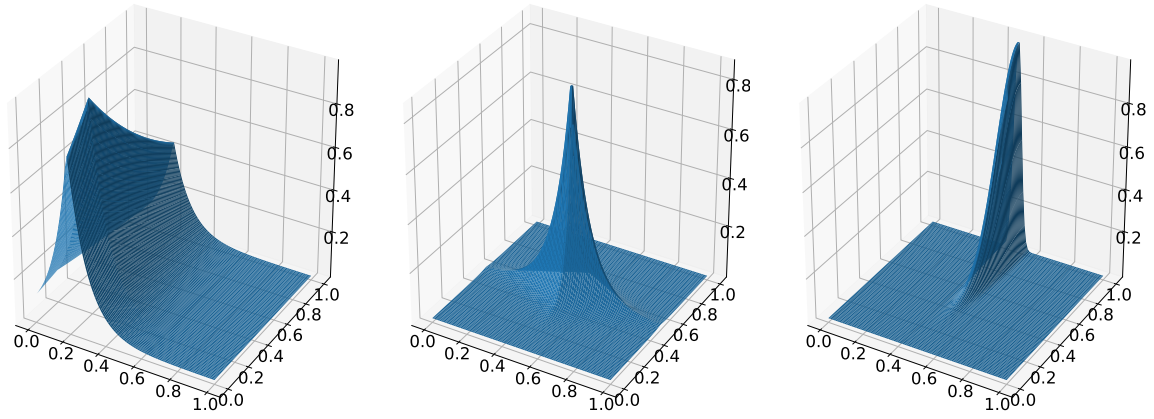
Plots of variants of the rational function f_4 are depicted in figure 12 (h) for $p = \binom{0.2}{0.2}$, $c = \binom{20}{2}$, as well as in (i) for $p = \binom{0.7}{0.7}$, $c = \binom{10}{6}$. The corresponding grids resulting from the two refinement schemes are compiled in figure 16.

$$f_5(\boldsymbol{x}) := \left(\sum_{i=1}^{d} c_i x_i \right)^{-d-1} \tag{22}$$

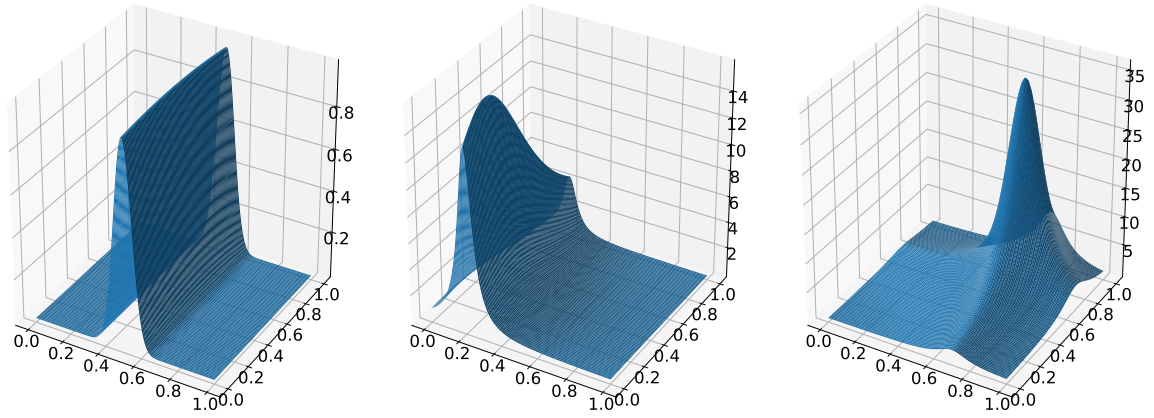
f_5 has not been plotted, but 5-dimensional test cases have been included in figure 18 and 20.



(a) $f_1, p = \binom{0.3}{0.3}, c = \binom{1}{2}$ (b) $f_1, p = \binom{0.6}{0.6}, c = \binom{10}{1}$ (c) $f_2, p = \binom{0.5}{0.5}, c = \binom{30}{2}$

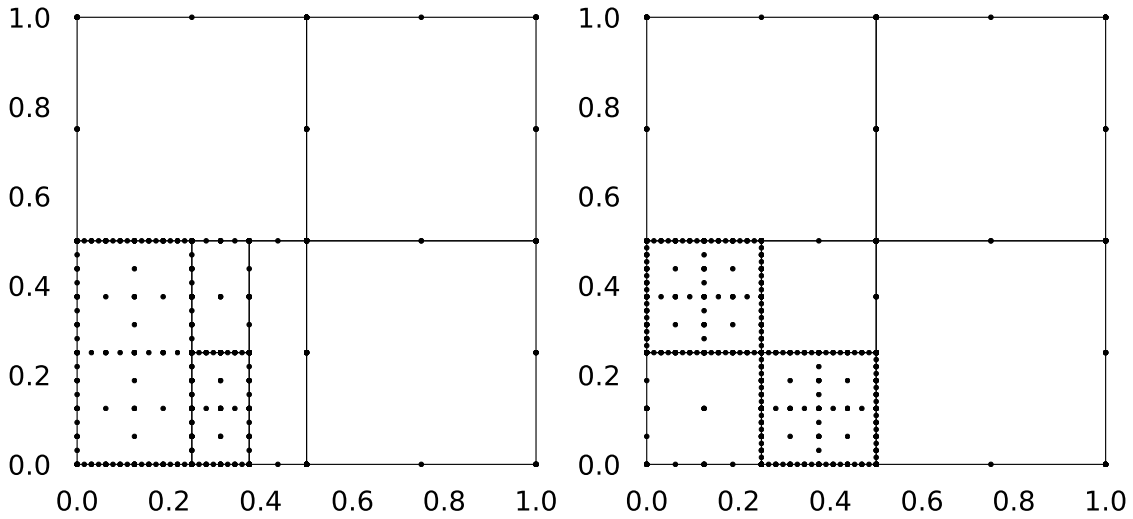


(d) $f_2, p = \binom{0.2}{0.2}, c = \binom{10}{1}$ (e) $f_2, p = \binom{0.5}{0.5}, c = \binom{10}{20}$ (f) $f_3, p = \binom{0.5}{1}, c = \binom{1000}{11}$

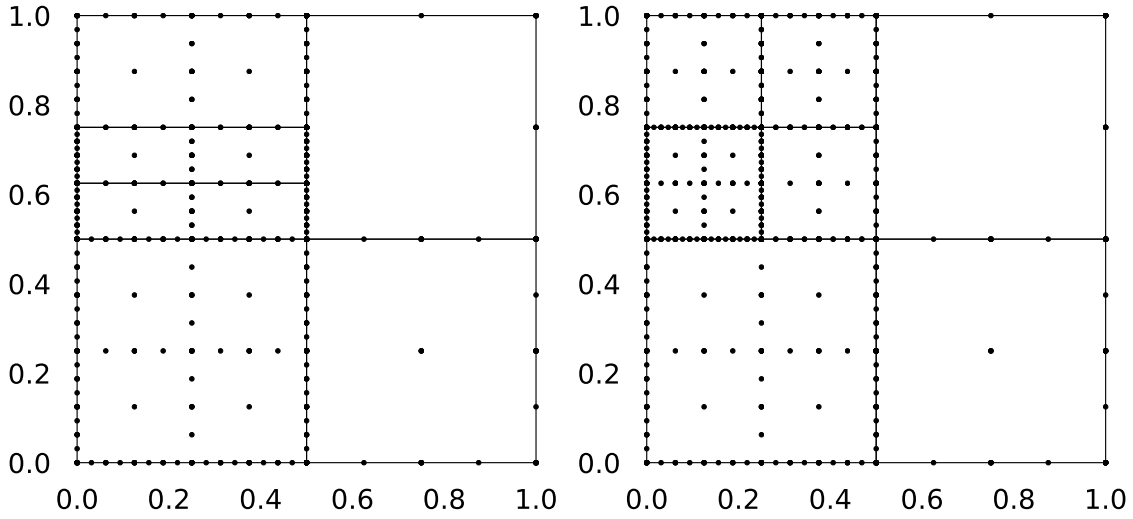


(g) $f_3, p = \binom{0.5}{0.5}, c = \binom{200}{0.1}$ (h) $f_4, p = \binom{0.2}{0.2}\ c = \binom{20}{2}$ (i) $f_4, p = \binom{0.7}{0.7}, c = \binom{10}{6}$

Figure 12: Plots of several two-dimensional variants of some employed test functions not contained in the set of test cases presented in [10]. Subfigures (e) and (f) serve as negative examples, in which full-dimensional split operations prove more effective and suitable due to a smaller number of distinct point evaluations. In all other cases (2d), the scheme applying single-dimensional splits performs at least as well, if not substantially better than the original procedure.

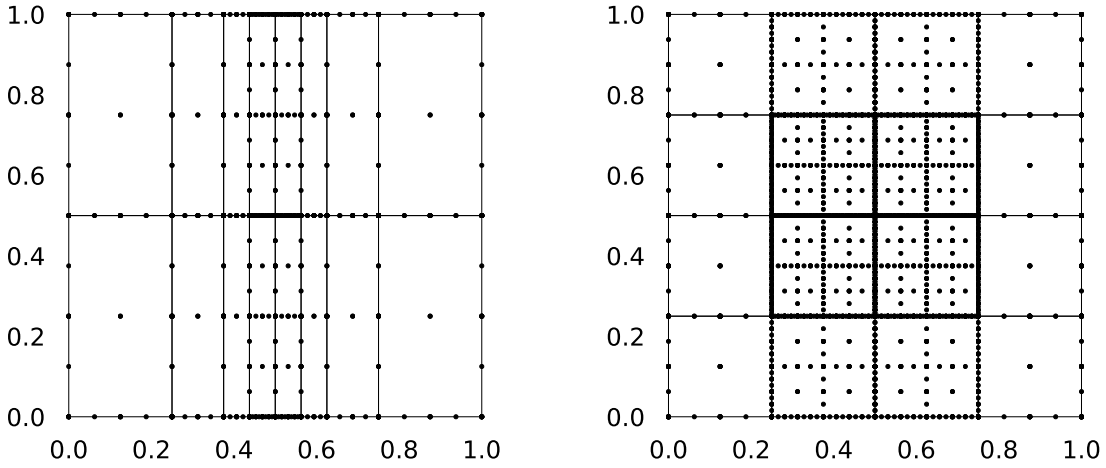


(a) $f_1, p = \binom{0.3}{0.3}, c = \binom{1}{2}$, $\epsilon = 5 \cdot 10^{-3}$, $\delta = 3$, **141** vs. **189** distinct point evaluations.
As the support of the objective function is concentrated in $[0, 0.3]^2$, both procedures refine only the lower left quadrant. In this case, the less strict regularity of the single-dimensional split operation yields only a visual advantage over the full-dimensional method. Interestingly, a minor reduction of ϵ to 0.0049 prompted a devastatingly inefficient behaviour in the latter, which enormously increases the resolution of both quadratic subregions in the lower left quadrant, while the grid created by the single-dimensional method remained unchanged.

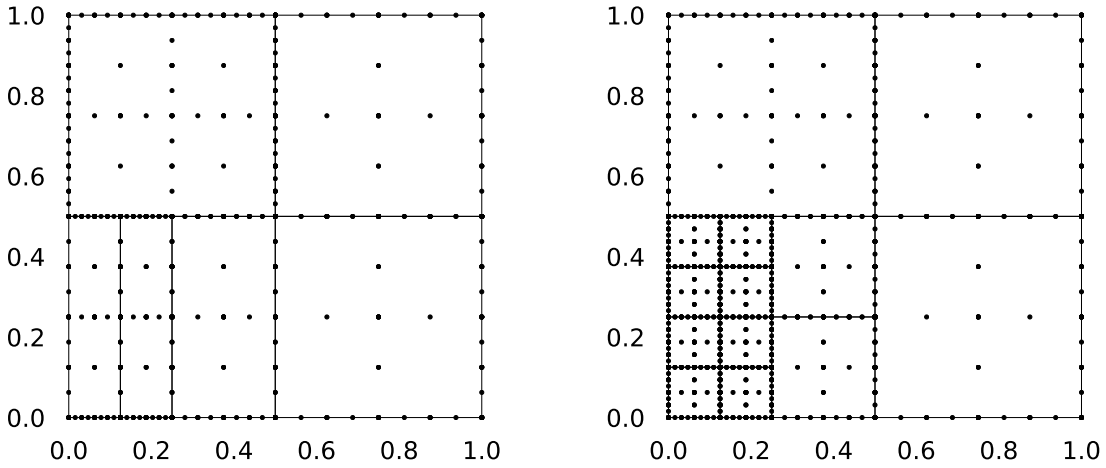


(b) $f_1, p = \binom{0.6}{0.6}, c = \binom{10}{1}$, $\epsilon = 10^{-2}$, $\delta = 3$, **184** vs. **240** distinct point evaluations.
Similar to the above case, the higher spatial flexibility of the single-dimensional procedure does only provide a visual advantage over the full-dimensional strategy here, as both procedures require similar numbers of point evaluations.

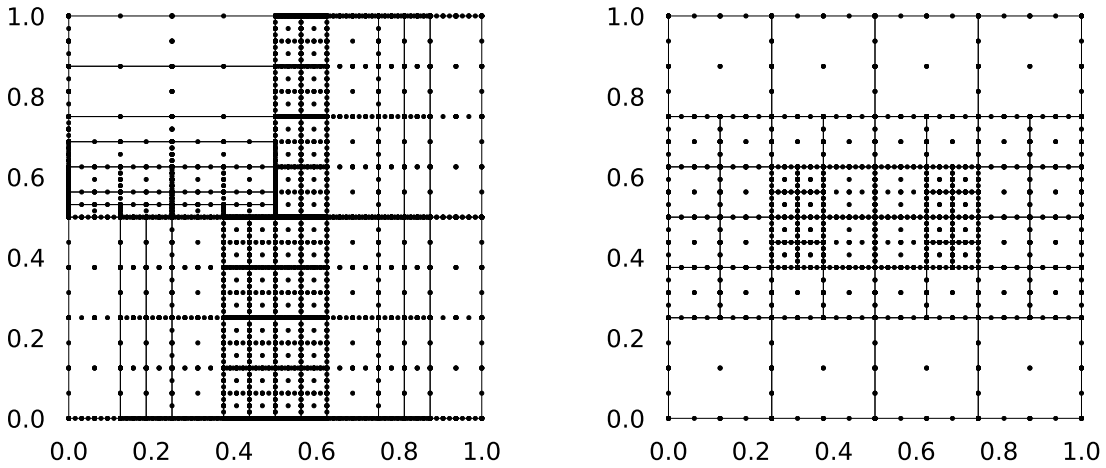
Figure 13: Grids resulting from variants of the function in (19), single-dimensional split operations on the left, full-dimensional split operations on the right.



(a) $f_2, p = \binom{0.5}{0.5}, c = \binom{30}{2}$, $\epsilon = 10^{-2}$, $\delta = 2$, **233** vs. **929** distinct point evaluations.
The single-dimensional strategy apparently allows for a refinement structure better suited to catch the shape of the elongated peak's support here.

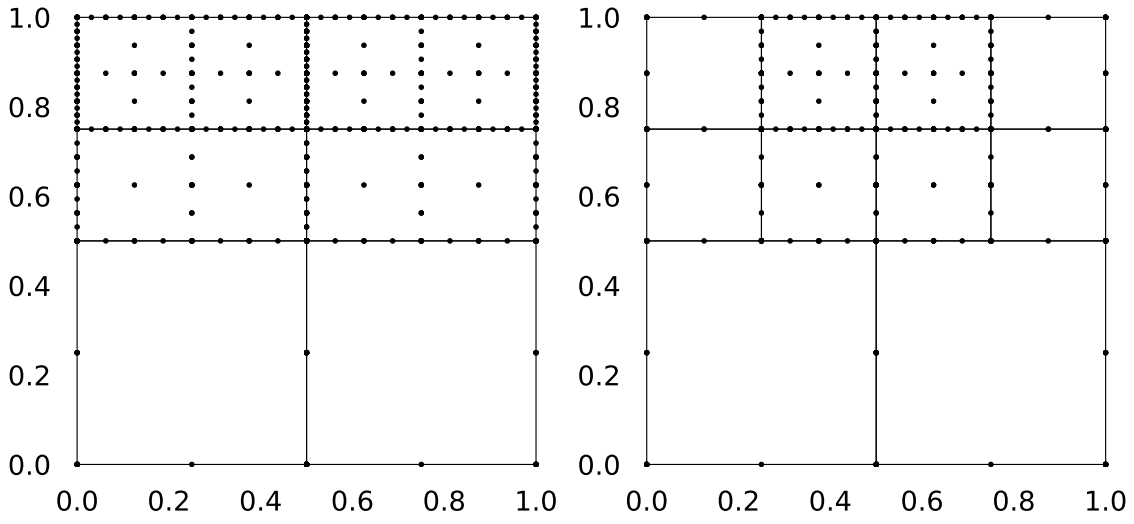


(b) $f_2, p = \binom{0.2}{0.2}, c = \binom{10}{1}$, $\epsilon = 10^{-3}$, $\delta = 2$, **205** vs. **369** distinct point evaluations.
Comparable to case (b) in figure 13, a slightly better visual adaption by the single-dimensional scheme, yet no immense improvements in efficiency.

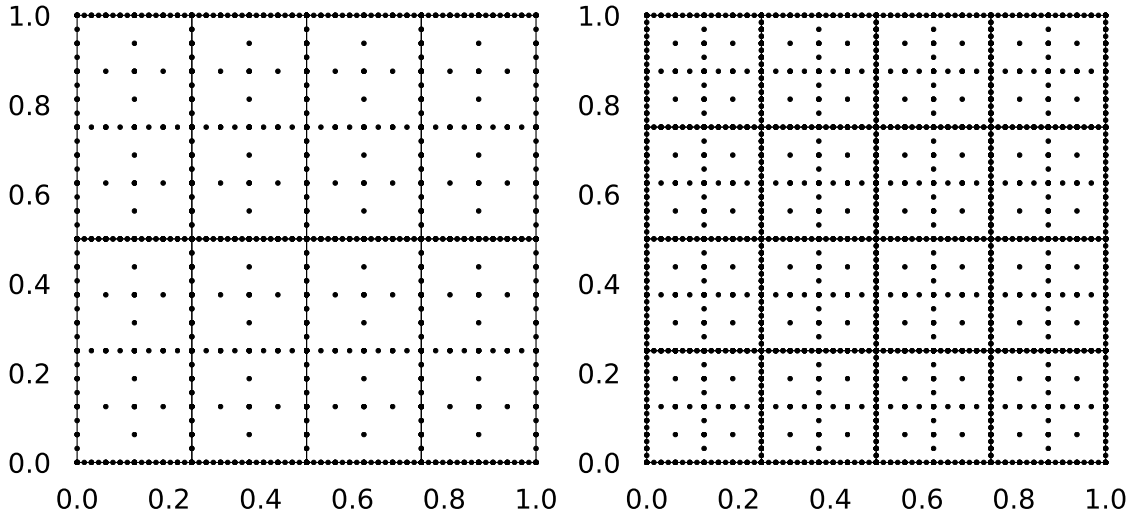


(c) $f_2, p = \binom{0.5}{0.5}, c = \binom{10}{20}$, $\epsilon = 10^{-2}$, $\delta = 2$, **1473** vs. **545** distinct point evaluations.
Here, the full-dimensional scheme clearly outperforms the single-dimensional method, which seemingly has difficulties catching the symmetry of the objective function. This may be due to the choice of initially splitting in y-dimension in the upper left quadrant.

Figure 14: Grids resulting from variants of the first function in (20), single-dimensional split operations on the left, full-dimensional split operations on the right.

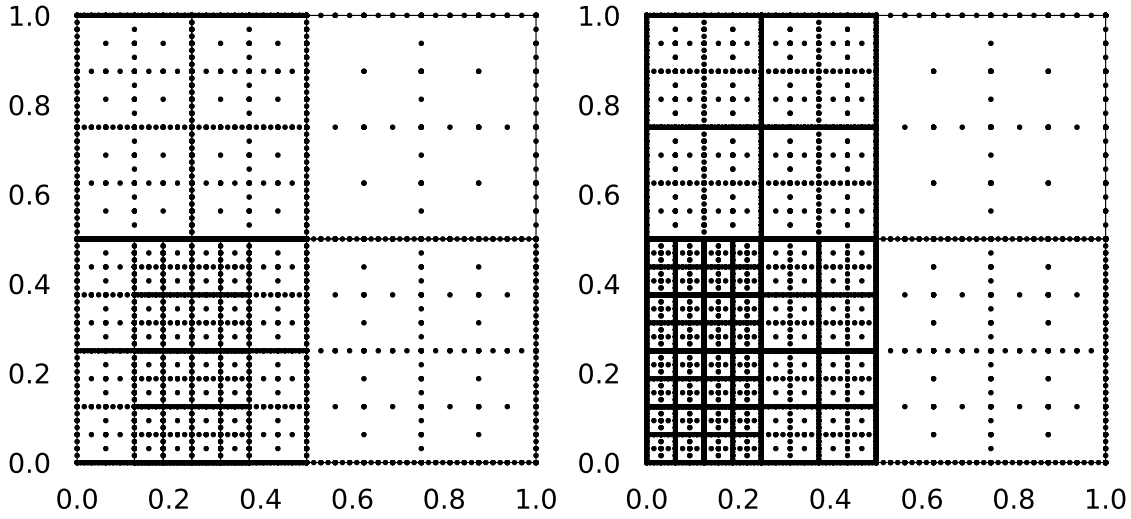


(a) $f_3, p = \binom{0.5}{1}, c = \binom{1000}{11}$, $\epsilon = 10^{-2}$, $\delta = 3$, **201** vs. **109** distinct point evaluations.
In this case, the single-dimensional method's surprising choice to split in y-direction rather than in x-direction prompts more densely refined sub-areas which would not be necessary in the upper outward corners of the domain, as the function is zero there. The full-dimensional method managed to detect this, which resulted in a slightly better performance.

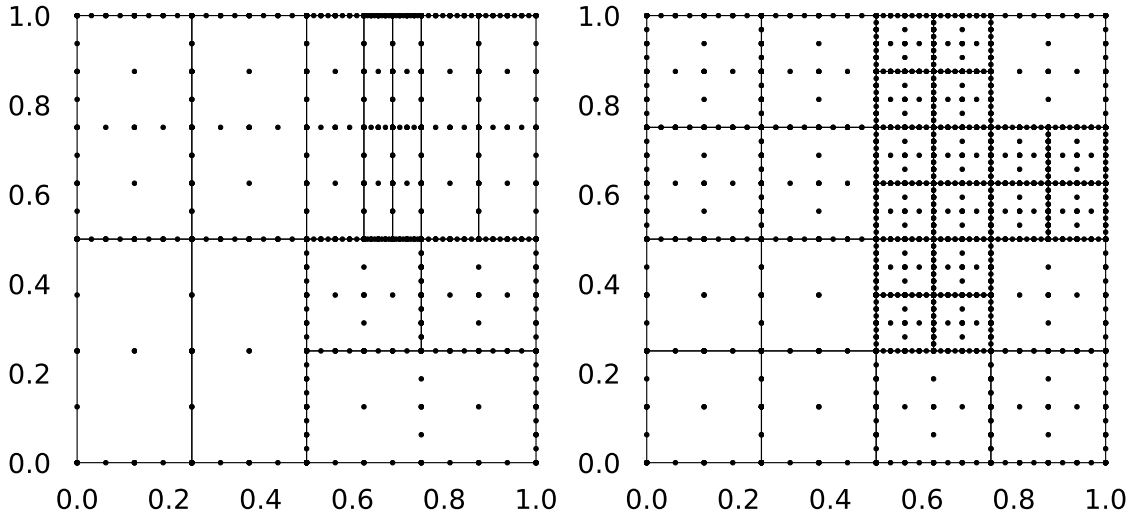


(b) $f_3, p = \binom{0.5}{0.5}, c = \binom{200}{0.1}$, $\epsilon = 10^{-4}$, $\delta = 3$, **481** vs. **897** distinct point evaluations.
Both procedures perform in a comparable fashion, the single-dimensional split procedure having a slight advantage due to the increased side length of the sub-areas in y-direction, which allows for a lower resolution in this dimension, since the objective function varies comparatively highly in x-, but not in y-dimension.

Figure 15: Grids resulting from variants of the second function in (20), single-dimensional split operations on the left, full-dimensional split operations on the right.

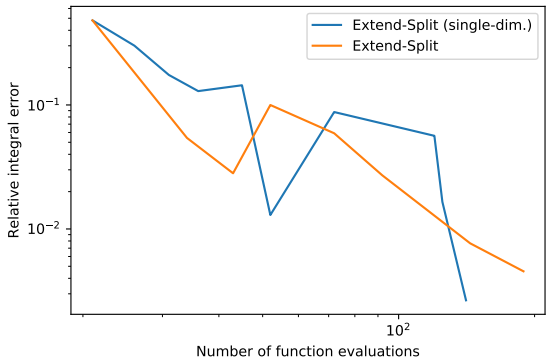


(a) $f_4, p = \binom{0.2}{0.2}$ $c = \binom{20}{2}$, $\epsilon = 10^{-4}$, δ arbitrary, **1345** vs. **2369** distinct point evaluations.
While there is no notable change on the right half of the domain, the single-dimensional scheme managed to adapt remarkably better to the stretched peak of the objective function along the middle of the lower left quadrant. Additionally, the function varies greatly in x-direction on the lower left quadrant, which the grid resulting from the procedure employing single-dimensional split steps also catches.

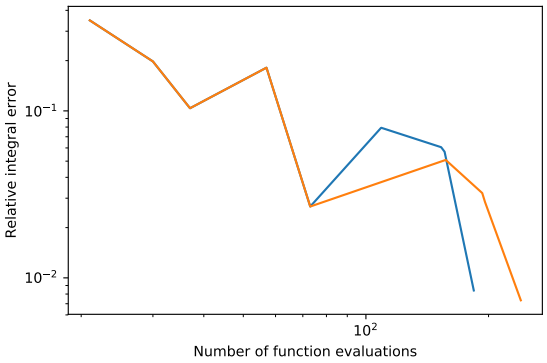


(b) $f_4, p = \binom{0.7}{0.7}, c = \binom{10}{6}$, $\epsilon = 10^{-3}$, δ arbitrary, **297** vs. **617** distinct point evaluations.
No too many differences, although again a slightly better adaptation by the single-dimensional method. This case is quite similar to examples (b) in both figure 13 and figure 14.

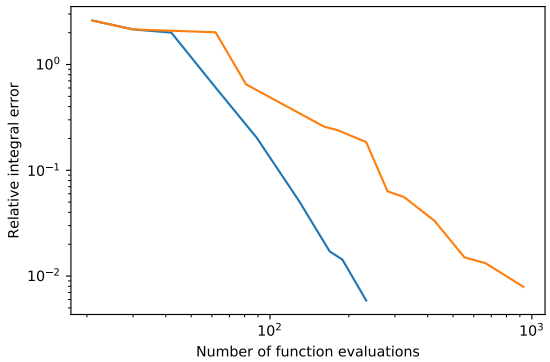
Figure 16: Grids resulting from variants of the function in (21), single-dimensional split operations on the left, full-dimensional split operations on the right.



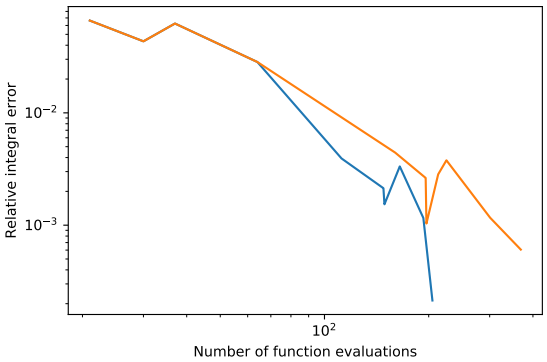
(a) $f_1, p = \binom{0.3}{0.3}, c = \binom{1}{2}$, $\epsilon = 0.5 \cdot 10^{-2}$, $\delta = 3$.



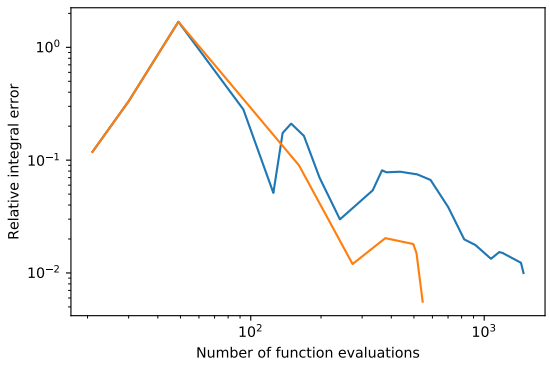
(b) $f_1, p = \binom{0.6}{0.6}, c = \binom{10}{1}$, $\epsilon = 10^{-2}$, $\delta = 3$.



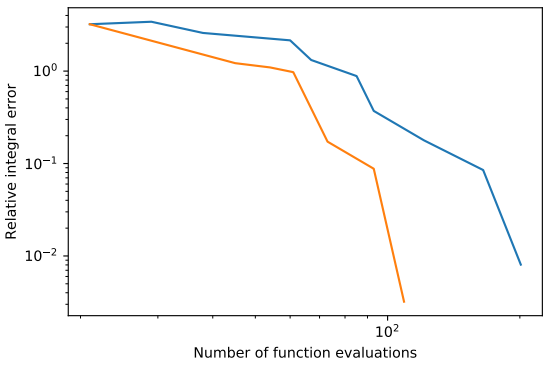
(c) $f_2, p = \binom{0.5}{0.5}, c = \binom{30}{2}$, $\epsilon = 10^{-2}$, $\delta = 2$.



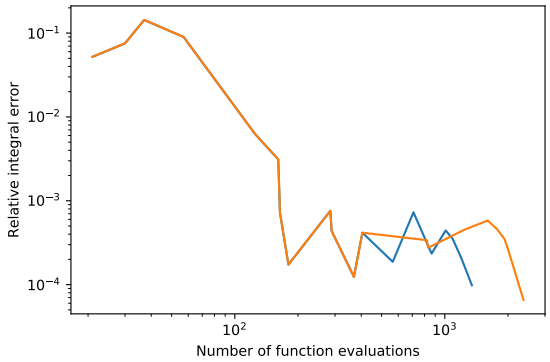
(d) $f_2, p = \binom{0.2}{0.2}, c = \binom{10}{1}$, $\epsilon = 10^{-3}$, $\delta = 2$.



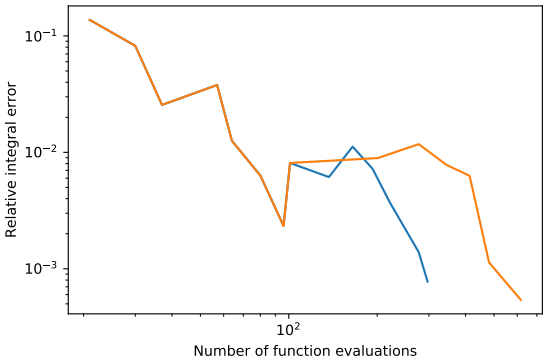
(e) $f_2, p = \binom{0.5}{0.5}, c = \binom{10}{20}$, $\epsilon = 10^{-2}$, $\delta = 2$.



(f) $f_3, p = \binom{0.5}{1}, c = \binom{1000}{11}$, $\epsilon = 10^{-2}$, $\delta = 3$.

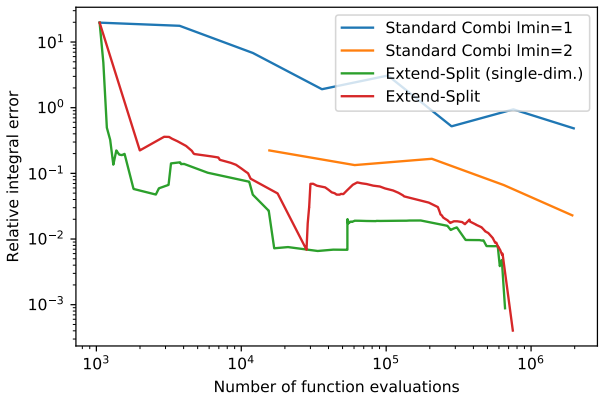


(g) $f_4, p = \binom{0.2}{0.2}$ $c = \binom{20}{2}$, $\epsilon = 10^{-4}$, δ arbitrary.

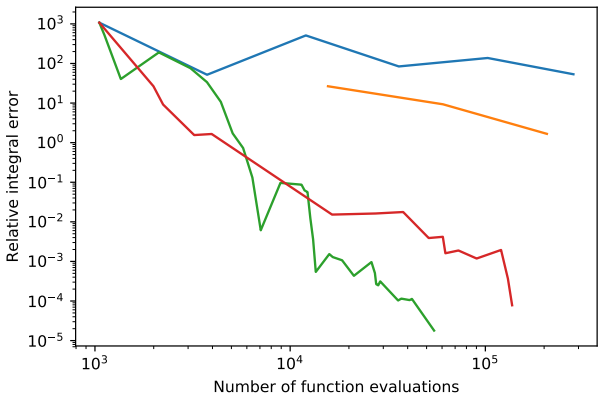


(h) $f_4, p = \binom{0.7}{0.7}, c = \binom{10}{6}$, $\epsilon = 10^{-3}$, δ arbitrary.

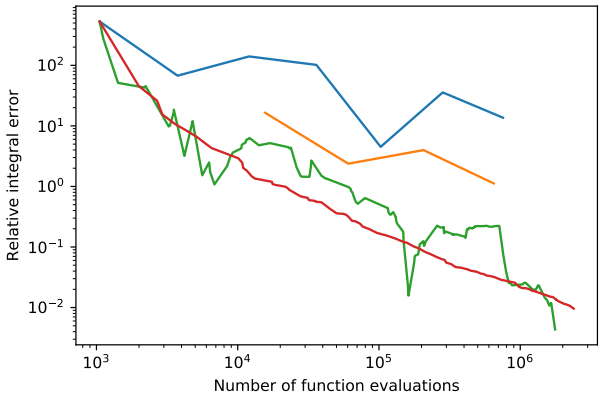
Figure 17: Error plots corresponding to the block-adaptive grid structures from figures (13), (14), (15) and (16), respectively the functions depicted in figure 12, except for the one in (g). The overall impression is, that at least in 2d, no immensely significant performance-related differences seem to exist between both schemes.



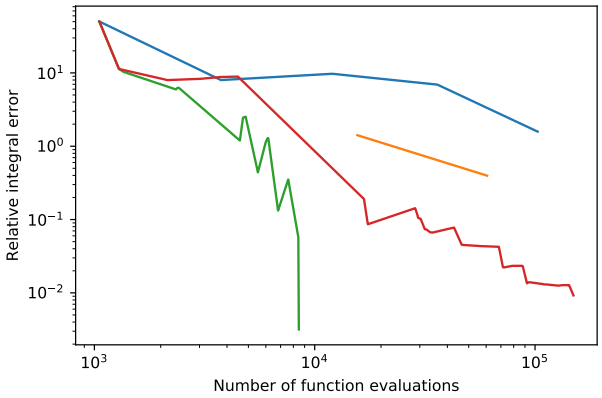
(a) f_1, where $d = 5$, $p = 0.2 \cdot \mathbb{1}$, $c = (1, 2, 3, 4, 5)^T$, $\epsilon = 10^{-3}$ and $\delta = 5$:
661140 vs. **748288** distinct evaluations.
As far as this test case is concerned, both schemes perform nearly identically. In comparison to the graphs in (b) and (c), the error corresponding to the single-dimensional variant consistently bounds the error of the full-dimensional variant from below.



(b) f_3, where $d = 5$, $p = 0.99 \cdot \mathbb{1}$, $c = 100 \cdot (1, 2, 3, 4, 5)^T$, $\epsilon = 10^{-4}$ and $\delta = 5$:
54629 vs. **137136** distinct evaluations.
After reaching the mark of 10^4 point evaluations, the single-dimensional variant seems to perform at least one order better until the tolerance is reached.

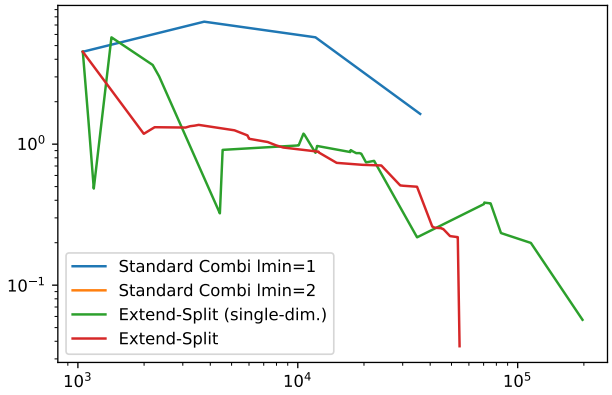


(c) f_4, where $d = 5$, $p = 0.99 \cdot \mathbb{1}$, $c = 10 \cdot (1, 2, 3, 4, 5)^T$, $\epsilon = 10^{-2}$ and $\delta = 3$:
1765987 vs. **2390888** distinct evaluations.
Like in (a), both schemes perform similarly. Interestingly, the error corresponding to the full-dimensional method decays in a much more continuous fashion than that of its single-dimensional counterpart.

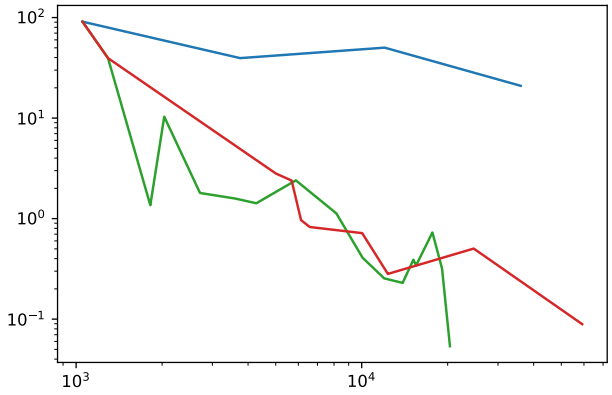


(d) f_5, where $d = 5$, $p = 0.99 \cdot \mathbb{1}$, $c = (1, 2, 3, 4, 5)^T$, $\epsilon = 10^{-2}$ and $\delta = 3$:
8471 vs. **149392** distinct evaluations.
This test case may serve as one of the best examples to hint at the potential of single-dimensional split steps in higher dimensions among all those presented, although this may be due to a fortunate choice of the objective function's parameters.

Figure 18: Some of the 5d test cases included in [10], namely the functions (19), the second function in (20), (21) and (22). Judging from those examples only, the single-dimensional variant appears to behave at least as efficiently as the underlying procedure.

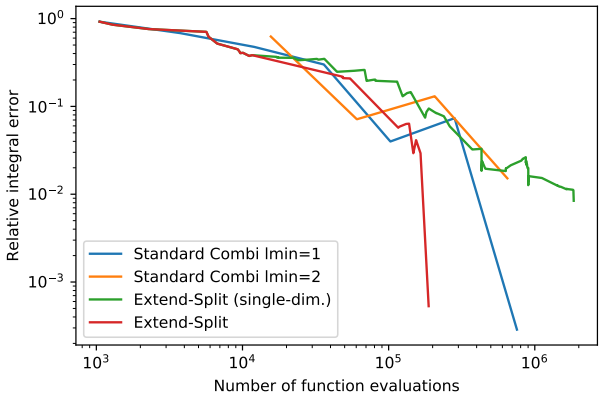


(a) f_1, where $d = 5$, $p = 0.3 \cdot \mathbb{1}$, $c = (1, 2, 4, 8, 16)^T$, $\epsilon = 10^{-1}$ and δ was arbitrary:
197151 vs. **54416** distinct evaluations.
In this example, the objective function's support has uniform length in all dimensions and thus a regularity in its structure which, comparable to the two-dimensional case, puts the full-dimensional scheme at a minor advantage.

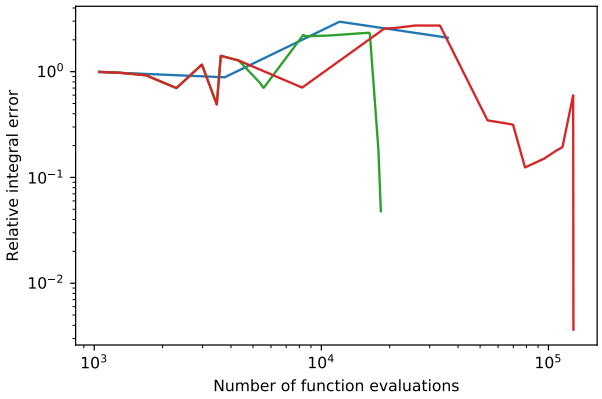


(b) f_1, where $d = 5$, $p = 10^{-1} \cdot (1, 1, 3, 3, 3)^T$, $c = (1, 2, 4, 8, 16)^T$, $\epsilon = 10^{-1}$, and δ was arbitrary:
20369 vs. **59167** distinct evaluations.
Here, on the other hand, a slight decrease in the lengths of $\text{supp}(f)$ in the first two coordinates turns the tide and the roles of both procedures are practically reversed.

Figure 19: Additional test cases with regard to function (19).



(a) f_4, where $d = 5$, $p = 0.2 \cdot \mathbb{1}$, $c = (2, 2, 10, 20, 50)^T$, $\epsilon = 10^{-2}$, and $\delta = 5$:
1846784 vs. **187842** distinct evaluations.
Note, that the full-dimensional procedure and even the standard combination technique literally outperform the single-dimensional scheme in this example. As a noteworthy remark, the execution of the single-dimensional variant involved a high number of splits, the majority of them in dimensions 2 to 4, which were however not always as effective in reducing the global error.



(b) f_4, where $d = 5$, $p = 10^{-1} \cdot (3, 3, 4, 5, 5)^T$, $c = (2, 2, 10, 20, 50)^T$, $\epsilon = 10^{-1}$, and δ was arbitrary:
18275 vs. **128806** distinct evaluations. Alike to the examples (a) and (b) in figure 19, a slight decrease in uniformity among the components of p provokes a different outcome for both strategies, as the single-dimensional variant shows similar tendencies in the beginning, but reaches the required tolerance much earlier. The majority of the splits were performed in dimension 1, 3 and 4.

Figure 20: Additional test cases with regard to function (21).

4.2 Conclusion

Judging by the results that have been presented in this section, the augmentation of the split-extend scheme, respectively the underlying spatially adaptive combination technique with the single-dimensional splitting operation may in comparison achieve a noteworthy increase in efficiency with regard to the number of function evaluations needed, if the integrand is suitable. Namely, if the objective function's largest variations are restricted to a proper subset of the $[d]$ dimensions, as in case of the functions illustrated in figure 12 (b), (c), (d) and (g) in 2d, the single-dimensional procedure seems to display its largest potential. In cases of functions on rectangular, full-dimensional supports like in the examples (e) and (f) in figure 12, the full-dimensional splitting strategy has been perceived to be both more suitable from a geometrical viewpoint, as well as of greater practical use in several instances. From the test cases on functions in higher dimensions, especially the examples (b) and (d) in figure 18, one gathers that the advantages provided by the utilisation of single-dimensional split operations may become even more prominent in those dimensions.
In contrast to those positive results, analogous to the two-dimensional cases, one may naturally construct a plethora of counterexamples, such as the one depicted in figure 19 (a) and figure 20 (a), in which the modified split-extend scheme performs quite poorly in comparison to the original procedure. Notice, that in figure 20 (a) even the standard combination technique demonstrated a higher performance than the adaptive scheme based on single-dimensional split steps.
Conversely, this can also be stated for the basic scheme, which is a simple consequence of the general "greedy" refinement strategy. As no additional information with regard to the objective function is included, e.g. its derivatives in case of differentiability, it seems to be too much to expect more of the scheme under such general circumstances. Not having sufficiently smooth integrands may also be considered a realistic setting in numerous fields of application, such as uncertainty quantification. Yet it is significant to remark, that almost all of the previously discussed test cases relied more or less heavily on the scheme being supplied with an exact reference solution, in order to produce better error estimations. If one had only used the hierarchical surpluses for error estimation, the convergence properties might have notably deteriorated, as the author noticed on several occasions when running even small test cases in two dimensions. However, as the overall approach is of a rather novel kind, further improvements with regard to error- and benefit-estimators are not unlikely to be achieved and might elevate the general performance of the scheme. It may on the other hand be of interest to consider higher-order quadrature rules, respectively basis functions, alongside with further error estimators described in [10], perhaps even particularly tailored to the single-dimensional splitting strategy.
For now, one would do best to conduct further testing, especially in higher dimensions, i.e. $d > 5$, as the method is expected to become increasingly potent.

List of Figures

References

[1] Martin Brokate. *Functional Analysis*. Lecture notes, Technische Universität München, 2017.

[2] Kersting G. Brokate M. *Maß und Integral*. Birkhäuser, 2010.

[3] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.

[4] Jochen Garcke. *Sparse grids in a nutshell*. Springer, 2013.

[5] Alan Genz. A package for testing multiple integration subroutines. 01 1987.

[6] Thomas Gerstner and Michael Griebel. Dimension-adaptive tensor-product quadrature. *Computing*, 71(1):65–87, September 2003.

[7] Michael Griebel, Michael Schneider, and Christoph Zenger. A combination technique for the solution of sparse grid problems. 1992.

[8] John Jakeman and Stephen Roberts. Local and dimension adaptive sparse grid interpolation and quadrature. 09 2011.

[9] Michael Obersteiner. Spatially adaptive combination environment. `https://gitlab.lrz.de/ga34suk/SpACE`.

[10] Michael Obersteiner and Hans-Joachim Bungartz. A spatially adaptive sparse grid combination technique for numerical quadrature. In *Sparse Grids and Applications - Munich 2018*. Springer Verlag, Jul 2018.

[11] Dirk Michael Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Dissertation, Technische Universität München, München, 2010.

[12] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Dokl. Akad. Nauk SSSR*, 148(5):1042–1045, September 1963.

[13] Brenner S. und Scott L. *The Mathematical Theory of Finite Element Methods*. Springer New York, 2008.

[14] Christoph Zenger. Sparse grids. In Wolfgang Hackbusch, editor, *Parallel Algorithms for Partial Differential Equations*, volume 31 of *Notes on Numerical Fluid Mechanics*, pages 241–251. Vieweg, 1991.