Fakultät für Informatik
Lehrstuhl für Wissenschaftliches Rechnen

Learning from Data with Geometry-Aware Sparse Grids

Kilian Michael Röhner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Helmut Seidl
Prüfer der Dissertation:	1.	Prof. Dr. Hans-Joachim Bungartz
	2.	Prof. Dr. Markus Hegland

Die Dissertation wurde am 03.08.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30.09.2020 angenommen.

Abstract

In data mining, high-dimensional and data-intensive problems are raising the bar of what models have to cope with. The runtime of many methods grows exponentially with the number of dimensions or does not scale well with increasing dataset sizes. Within the variety of techniques that can cope with both, sparse grids offer reliable error estimates and traceability of model evaluations. While they have proven to perform well for moderate dimensionalities, they are in their traditional form not applicable to problems with several hundreds or even thousands of dimensions.

We accept this challenge by introducing geometry-aware sparse grids. This variant of sparse grids considers not only the relevance of single dimensions but also takes into account the interactions between dimensions. The resulting grids span spaces with thousands of dimensions in which we consider probability density functions. We discuss how to efficiently compute those density estimations and shift the heavy computational effort offline while maintaining the flexibility of adapting the model to the problem at runtime. Exploiting the structure of the established combination grid scheme, we speed up the learning by several orders of magnitude. To handle big data problems, the SG++ data mining pipeline is introduced. It includes a batch-wise parallel density estimation learner that builds up our classifier.

Density estimation forms the basic building block for our Bayes' classifier. We investigate refinement strategies in the context of classification for (geometry-aware) sparse grids and introduce a dimensional-adaptive combigrid scheme for image classification. The performance of this approach with regard to accuracy, runtime and flexibility is shown at the example of classification of the MNIST and CIFAR datasets, two cases that could not be tackled with sparse grid methods so far. Whilst we don't reach the precisions other techniques such as neural networks achieve, our models offer easy traceability and transparent insights into the datasets.

Contents

1. Introduction

Machine Learning undertook a rapid development in the last decade. Many problems thought impossible to solve computationally were tackled successfully using data-driven algorithms. This went hand in hand with the systematic collection of data in all areas of life. While some said that “data is the gold of the 21st century” [21], others stated that “data is the new oil” [70]. All agree that the gathered data is a valuable asset to economy and society and that modern and efficient algorithms are required to reveal its potential. In this thesis, we present novel and efficient sparse grid-based techniques to access this value.

While the idea to employ sparse grids for data-driven problems is not new, it is certainly fresher than both the sparse grids technique and the development of data-driven algorithms in general. The former started with the works of Smolyak [91] in the 60s and was then brought up again by Zenger [106] in the 90s. It stems from numerical discretization of high-dimensional problems such as partial differential equations. The *curse of dimensionality* [7] is the exponential growth of degrees of freedom with the number of dimensions, when a full grid is chosen to approximate a high-dimensional function. With the *sparse grid technique*, this curse of dimensionality is mitigated by only including the degrees of freedom that are likely to contribute most to the solution. Applying hierarchical basis functions instead of a nodal basis functions allows for an a priori graduation of the degrees of freedom by their expected contribution to the solution.

Sparse grids have already proven to be applicable to many problem classes from the world of data mining. In [29], they have been applied to regression and binary classification problems with varying dataset sizes and dimensionalities up to 34. With the grid structure adapted to the problem via spatial adaptivity, the approximation results were even better as shown in [79]. Employing sparse grids for density estimation has been discussed before [78]. It suits as a building block for both clustering [77] and classification [76]. Much work has been published since and in-between those mentioned to address specific problem classes, more efficient algorithms and related techniques (such as the combination grid technique), among them also ideas how to approach image data [52].

One of the most exciting topics in machine learning is image classification. Training a model with previously labeled image data, it generalizes the visual attributes. For new images, the label is then predicted. Neural networks [66] have shown to be applicable to a wide range of image classification tasks [94]. In this thesis, we research on the application of sparse grids to image classification and related problems. One of the key advantages of sparse grid based data mining is the traceability and transparency of the learning process. The model parameters we learn directly relate to the problem space and codify it explicitly. Such methods are currently in high demand as they allow to formulate robust and legal compliant model accuracies. For example, the European Parliament recently voted for explainable algorithms in automated decision-making systems [71].

The formal introduction to hierarchical grid-based function representation is given in Chap. 2. There, we provide a structured approach to the construction and characteristics of component grids and regular sparse grids. With the data at hand, we adapt our sparse grid models to the problem. Spatial adaptivity [79] allows for a finer resolution in certain areas of the problem space, yielding spatially adaptive sparse grids. Dimensional adaptivity [42] regulates the resolution in each dimension individually, resulting in dimensional adaptive sparse grids. Based on the grids we define, we then turn to different choices of hierarchical basis functions with local support. Together, the grid points and the basis functions span the sparse grid space, which we search the approximation for. We present the *kinked linear basis functions*, a novel type of basis functions derived from the *modified linear basis functions* in this chapter. It allows us to perform the high-dimensional image classification tasks with a considerably higher accuracy than with all other basis functions investigated. Related to the sparse grid technique, we also show the combination grid technique [39] allowing for an inherent parallelization scheme based on anisotropic full grids.

How we employ spatially adaptive sparse grids for data-driven problems is presented in Chap. 3. The mathematical foundations we require to flexibly handle sparse grids in various settings is laid in this chapter. Starting with the density estimation approach, we show how to unify both spatial adaptivity and computational speedup using an offline/online scheme, which becomes only possible by the techniques we present. Thereby, the offline/online scheme is proven to be applicable as an incremental learning scheme with spatially adaptive sparse grids. This allows us to shift the heavy computational effort of acquiring the density estimation to an offline stage by using matrix decomposition techniques tailored to the linear systems at hand. Based on the density estimation, we then build up a Bayes classifier [40] allowing multi-class classification. For this to integrate with spatially adaptive sparse grids, we propose refinement indicators for classification taking into account one sparse grid per class.

To fully exploit the computational capability of modern hardware, we investigate how to parallelize the learning algorithms in Chap. 4. A parallelization scheme including multiple interlocking layers is proposed. On the first layer, we parallelize over the batches (data parallelism). Subsequent, the computation is parallelized with the combination grid technique (model parallelization). To exploit the tensor operations performed when training the machine learning model, we finally employ ScaLAPACK [12] (distributed linear algebra). The potentials of those parallelization layers is discussed and evaluated. In the end, this enables us to both learn big data problems and increase the model complexity in reasonable computational time.

Exploiting the geometric properties of image datasets is key when tackling them with sparse grids. How to use the a priori knowledge of this problem class with geometry-aware sparse grids is presented in Chap. 5. Employing sparse grids to tackle image classification problems with more than 1,000 dimensions becomes only possible by further thinning the sparse grid. We propose approaches to do so for both grayscale images and color channel images, optionally creating a data hierarchy to take into account both fine-grained and coarse-grained features of the image data. Consequently, the application of geometry-aware sparse grids to different image classification benchmark datasets is discussed.

We finalize the thesis with software aspects in Chap. 6. Making the outcomes from this publicly funded science accessible to the public and the scientific community in a user-friendly fashion was of import during this project. The resulting *data mining pipeline* is presented from both a user- and developer perspective.

2. Hierarchical Grid-based Function Representation

For a function f on $\Omega = [0,1]^d$ (where $d \in \mathbb{N}$ is the dimensionality) with $f\colon \Omega \to \mathbb{R}$, a common task in numerics is to find a discrete representation $\tilde{f}\colon \Omega \to \mathbb{R}$ of f on a discrete grid $\mathcal{G}$, so that

$$f(\boldsymbol{x}) \approx \tilde{f}(\boldsymbol{x}) = \sum_{p \in \mathcal{G}} \alpha_p \cdot \phi_p(\boldsymbol{x}) \tag{2.1}$$

holds. $p \in \mathcal{G}$ is called a grid point and $\phi_p\colon \Omega \to \mathbb{R}$ its corresponding basis function, which is weighted with $\alpha_p \in \mathbb{R}$ to obtain the discrete approximation.

With the data problems we have in mind, it is essential that we deal with the curse of dimensionality and choose the degrees of freedom, which form our grid, accordingly. The resulting sparse grid structure we are about to introduce and the a priori unknown distribution of the data in Ω brings us to approximation structures that allow to be spatially adapted to the problem at hand. In turn, this is the reason we employ hierarchical bases. While a nodal basis could offer computational benefits through the local support of the corresponding basis functions, it proves unsuitable to deal with the requirements of spatial adaptivity. Thus, we construct the grids and corresponding basis functions with the goal to obtain a hierarchical structure.

The algorithms we run on the grids require a structured input. We see how to introduce grids in one dimension that are of structured form. Each point has a well-defined support node – grids with unstructured nodes are not used in this thesis. The one-dimensional structure then leads to multi-dimensional grids via the tensor product approach.

Most concepts we discuss in this chapter are not new. With grid-based linear interpolation dating back to the ancient Greeks, sparse grids have been discovered by Smolyak [91] in the 60s and brought back to life by Zenger [106] in the 90s. Notable literature includes also the 2004 article in *Acta Numerica* [18] and the dissertation by Dirk Pflüger [79]. For a complete overview on sparse grids, we refer to the latter two. Our contribution to approximation theory is limited to the kinked linear basis function presented in Sec. 2.2.3.

In this chapter, we discuss different hierarchical grid setups for both the one-dimensional case ($d = 1$) and the multi-dimensional case ($d > 1$) in Sec. 2.1. We see how to mitigate the curse of dimensionality by employing sparse grids, which contain significantly less points than full grids. Then, we investigate different types of basis functions ϕ in Sec. 2.2. The three types we present differ in how the basis function extrapolates towards the boundary of Ω. Of those three, the *kinked linear basis function* is a novel approach to extrapolate towards the boundary. It works well in high-dimensional cases when the function value at the boundary cannot assumed to be zero, which proves valuable in the course of this thesis. Finally, in Sec. 2.3 we show how the grids and basis functions are combined to construct the function spaces that we search the approximation for.

2.1. Hierarchical Grids

The grid we use to represent our function consists of the supporting points p for the interpolant $\tilde{f}$. In the following, we first discuss one-dimensional subspace grids and component grids. Then, multi-dimensional grids are discussed, where both full grids and sparse grids are introduced.

2.1.1. One-dimensional Hierarchical Grids

The one-dimensional problems we consider live on $\Omega^1 := [0, 1]$. To construct one-dimensional grids, we start by defining one-dimensional grid points:

Definition 2.1.1 (One-dimensional grid point)
A one-dimensional grid point p defined via its level $l \in \mathbb{N}$ and index $i \in \mathbb{N}$ (whereas $i < 2^l$ has to hold) is given as a tuple

$$p := (l, i) \tag{2.2a}$$

and the set of all one-dimensional grid points is thus

$$\mathbb{G}^1 = \left\{ (l, i) \in \mathbb{N} \times \mathbb{N} \;\middle|\; i < 2^l \right\} . \tag{2.2b}$$

The absolute coordinate of a grid point $\texttt{coord}^1$ given by

$$\begin{aligned} \texttt{coord}^1 \colon \mathbb{G}^1 &\to \Omega^1 , \\ (l, i) &\mapsto \frac{i}{2^l} \end{aligned} \tag{2.2c}$$

is located at the center of its support $\texttt{support}^1$ given by

$$\begin{aligned} \texttt{support}^1 \colon \mathbb{G}^1 &\to \mathcal{P}(\Omega^1) \\ (l, i) &\mapsto \left[\frac{i-1}{2^l}, \frac{i+1}{2^l} \right] . \end{aligned} \tag{2.2d}$$

A one-dimensional grid $\mathcal{G}$ is then defined as a discrete set of one-dimensional grid points. For two grid points $p_1 = (l_1, i_1)$ and $p_2 = (l_2, i_2)$ with $l_1 < l_2$, we denote p_1 as the *coarser* grid point and p_2 as the *finer* grid point.

To help with the construction of regular grids, we continue with the smallest building block of our grids and define the one-dimensional subspace grid:

Definition 2.1.2 (One-dimensional subspace grid)
The one-dimensional subspace grid of level $l \in \mathbb{N}$ is given by

$$\texttt{subspaceGrid}^1_l := \left\{ (l,i) \in \mathbb{G}^1 \;\middle|\; i \bmod 2 = 1 \right\} . \tag{2.3}$$

From now on, we only consider grid points with odd indices as defined in Def. 2.1.2. Note that all the interiors of the supports of the grid points in each subspace grid are pairwise disjoint (they only overlap at most at their boundaries) and that the union of all grid points' supports cover Ω^1:

$$\forall p, q \in \texttt{subspaceGrid}^1_l \text{ with } p \neq q :$$
$$\text{interior}(\texttt{support}^1(p)) \cap \text{interior}(\texttt{support}^1(p)) = \varnothing\,, \tag{2.4a}$$

and

$$\Omega^1 = \bigcup_{p \in \texttt{subspaceGrid}^1_l} \texttt{support}^1(p)\,. \tag{2.4b}$$

A hierarchy of the defined subspace grids is inherently obtained, subspace grid of level l is parent to the subspace grid of level $l+1$.

With the help of the subspace grids, we now define one-dimensional hierarchical grids:

Definition 2.1.3 (One-dimensional hierarchical grid)
The one-dimensional hierarchical grid of level $l \in \mathbb{N}$ is given by

$$\texttt{componentGrid}^1_l := \bigcup_{j \in [l]} \texttt{subspaceGrid}^1_j\,. \tag{2.5}$$

Note that for the one-dimensional case, there is no difference between a component grid, a full grid or a regular sparse grid. For clarity, we refer to the grids defined in Def. 2.1.3 as *component grids*. A visualization of the component grid of level 4 and its construction through the subspaces is given in Fig. 2.1.

Every grid point except for the root given by

$$\texttt{root}^1 := (1,1) \tag{2.6}$$

has a parent grid point, and every grid point has exactly two child grid points.

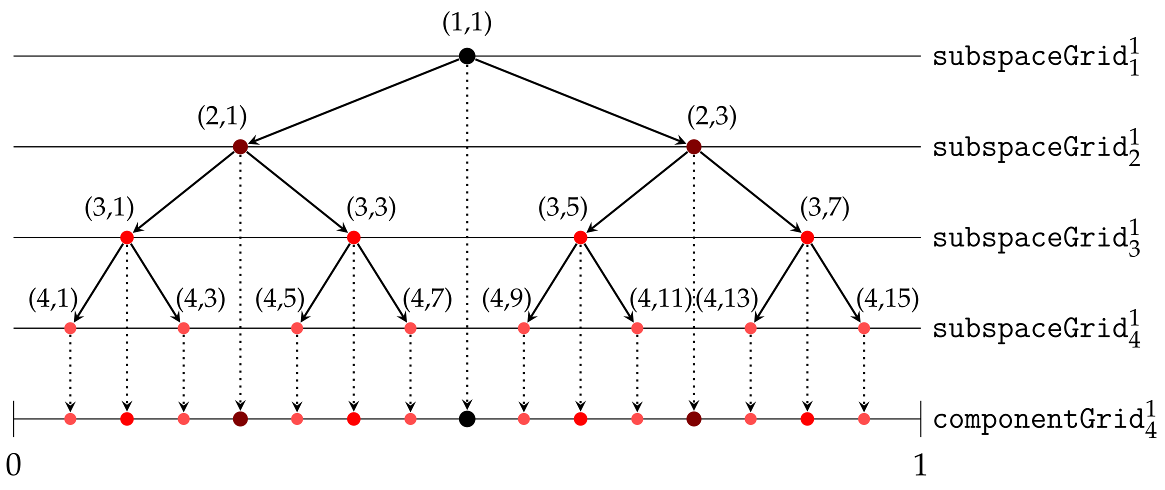


Figure 2.1.: `componentGrid`$_4^1$ is constructed through the union of the subspaces from level 1 to 4. This construction is visualized via the dotted arrows connecting the grid points in the subspaces with the corresponding grid points in `componentGrid`$_4^1$. In the subspace grids, each grid point is labeled with its level and index. The continuous arrows denote the parent-child hierarchy between the grid points of the different subspaces.

Definition 2.1.4 (Parents and ancestors of one-dimensional grid points)
The one-dimensional parent a of grid point (l, i) is given by

$$\begin{aligned}\texttt{parent}^1 \colon \mathbb{G}^1 &\to \mathbb{G}^1\,,\\ (l,i) &\mapsto \begin{cases}\texttt{root}^1\,, & (l,i) = \texttt{root}^1\,,\\ (l-1, \texttt{parent-index}(i))\,, & \text{else},\end{cases}\end{aligned} \tag{2.7a}$$

where `parent-index` is given by

$$\begin{aligned}\texttt{parent-index} \colon \mathbb{N} &\to \mathbb{N}\,,\\ i &\mapsto \begin{cases}\frac{i+1}{2}\,, & i \bmod 4 = 1\,,\\ \frac{i-1}{2}\,, & i \bmod 4 = 3\,.\end{cases}\end{aligned} \tag{2.7b}$$

With $\texttt{parent}_0^1(p) := p$, we then recursively define for $k \in \mathbb{N}$

$$\begin{aligned}\texttt{parent}_k^1 \colon \mathbb{G}^1 &\to \mathbb{G}^1\,,\\ p &\to \texttt{parent}^1(\texttt{parent}_{k-1}^1(p))\end{aligned} \tag{2.7c}$$

to end up with

$$\begin{aligned}\texttt{ancestors}^1 \colon \mathbb{G}^1 &\to \mathcal{P}(\mathbb{G}^1)\,,\\ p &\to \left\{\texttt{parent}_k^1(p) \;\middle|\; k \in \mathbb{N}\right\}\end{aligned} \tag{2.7d}$$

and

$$\begin{aligned}\texttt{ancestors}^{1^*} \colon \mathbb{G}^1 &\to \mathcal{P}(\mathbb{G}^1)\,, \\ p &\to \left\{ \texttt{parent}^1_k(p) \;\middle|\; k \in \mathbb{N}_0 \right\}\,.\end{aligned} \tag{2.7e}$$

$\texttt{parent}^1_k(p)$ contains the parents of degree k of p. With p being its own parent of degree 0, the parents of degree 1 are the intermediate parents, the parents of degree 2 its grandparents and so forth. $\texttt{ancestors}^1(p)$ is then the union over all parents starting from degree 1 and eventually also containing $\texttt{root}^1$. The difference between $\texttt{ancestors}^1(p)$ and $\texttt{ancestors}^{1^*}(p)$ is that the latter also contains p itself whereas the former does not.

Definition 2.1.5 (Children and descendants of one-dimensional grid points)
Equivalently, the grid point (l,i) has the two children:

$$\begin{aligned}\texttt{left-child}^1 \colon \mathbb{G}^1 &\to \mathbb{G}^1\,, \\ (l,i) &\mapsto (l+1, 2i-1)\end{aligned} \tag{2.8a}$$

and

$$\begin{aligned}\texttt{right-child}^1 \colon \mathbb{G}^1 &\to \mathbb{G}^1\,, \\ (l,i) &\mapsto (l+1, 2i+1)\,.\end{aligned} \tag{2.8b}$$

All children of p are given by

$$\begin{aligned}\texttt{children}^1 \colon \mathbb{G}^1 &\to \mathcal{P}(\mathbb{G}^1)\,, \\ p &\mapsto \left\{ \texttt{left-child}^1(p), \texttt{right-child}^1(p) \right\}\,.\end{aligned} \tag{2.8c}$$

With $\texttt{children}^1_0(p) := \{p\}$, we then recursively define for $k \in \mathbb{N}$

$$\begin{aligned}\texttt{children}^1_k \colon \mathbb{G}^1 &\to \mathcal{P}(\mathbb{G}^1)\,, \\ p &\mapsto \bigcup_{q \in \texttt{children}^1_{k-1}(p)} \texttt{children}^1(q)\,.\end{aligned} \tag{2.8d}$$

to end up with

$$\begin{aligned}\texttt{descendants}^1 \colon \mathbb{G}^1 &\to \mathcal{P}(\mathbb{G}^1)\,, \\ p &\mapsto \bigcup_{k \in \mathbb{N}} \texttt{children}^1_k(p)\,.\end{aligned} \tag{2.8e}$$

and

$$\begin{aligned}\texttt{descendants}^{1^*} : \mathbb{G}^1 &\to \mathcal{P}(\mathbb{G}^1)\,, \\ p &\mapsto \bigcup_{k \in \mathbb{N}_0} \texttt{children}^1_k(p)\,. \end{aligned} \tag{2.8f}$$

Similar to the definition of parents and ancestors, $\texttt{children}^1_k(p)$ contains all children of p of degree k. With p being its own child of degree 0, the children of degree 1 are the intermediate children (given by $\texttt{children}^1(p)$), the children of degree 2 are the grandchildren and so forth. $\texttt{descendants}^1(p)$ and $\texttt{descendants}^{1^*}(p)$ contain all those hierarchical children of degree 1 up to infinity, rendering those sets infinite. The difference between the two is, that $\texttt{descendants}^{1^*}(p)$ also contains p itself whereas $\texttt{descendants}^1(p)$ does not. The parent-child relations are denoted with continuous arrows in Fig. 2.1.

Boundary Grid Points Usually, $\texttt{root}^1$ at the center of the domain is the root of the parent-child tree. However, for some algorithms, it is necessary to define two special grid points.

Definition 2.1.6 (Boundary grid points)
At the left boundary of Ω^1, the virtual grid point $\texttt{left-boundary} := (0, 0)$ with coordinate $\texttt{coord}^1(\texttt{left-boundary}) = 0$ is situated. At the right boundary of Ω^1, the virtual grid point $\texttt{right-boundary} := (0, 1)$ with coordinate $\texttt{coord}^1(\texttt{right-boundary}) = 1$ is situated. We set:

$$\texttt{right-child}^1(\texttt{left-boundary}) = \texttt{right-boundary}\,, \tag{2.9a}$$

$$\texttt{parent}^1(\texttt{right-boundary}) = \texttt{left-boundary}\,, \tag{2.9b}$$

$$\texttt{left-child}^1(\texttt{right-boundary}) = \texttt{root}^1\,, \tag{2.9c}$$

$$\texttt{parent}^1(\texttt{root}^1) = \texttt{right-boundary} \tag{2.9d}$$

and

$$\mathcal{B}^1 := \{\texttt{left-boundary}, \texttt{right-boundary}\}\,. \tag{2.9e}$$

`left-boundary` does not have a left child and `right-boundary` does not have a right child. Also, `left-boundary` does not have a parent.

Thus, the boundary points serve as ancestors to the root point. They can be used to model boundary function values unequal of zero, but the resulting grid structures are heavily subject to the curse of dimensionality. In this thesis, we do not use boundary grid points to model non-zero boundaries. Instead, we use them to determine the set of geometric neighbors of a given grid, as we see later in Alg. 5.

Neighborhood of Grid Points Two grid points are neighbors of each other, if no other grid point lies between them. We formalize this by defining the relation $\texttt{neighbor}^1$ as

Definition 2.1.7 (Neighborhood of grid points)
For a grid $\mathcal{G} \subset \mathbb{G}^1$, the relation $\texttt{neighbor}^1$ is given as

$$\begin{aligned}\texttt{neighbor}^1_{\mathcal{G}} := \Big\{(p_1, p_2) \in \Big(\mathcal{G} \cup \mathcal{B}^1\Big) \times \Big(\mathcal{G} \cup \mathcal{B}^1\Big) \;\Big|\; & p_1 \neq p_2, \\ \nexists p_3 \in \mathcal{G} : \Big(& \texttt{coord}^1(p_1) < \texttt{coord}^1(p_3) < \texttt{coord}^1(p_2) \\ & \text{or } \texttt{coord}^1(p_2) < \texttt{coord}^1(p_3) < \texttt{coord}^1(p_1)\Big)\Big\}\,.\end{aligned} \tag{2.10}$$

Obviously, the definition of geometric neighbors is different from the parent-child relationship. Only for a leaf grid point, we know that one of its geometric neighbors is the direct parent. The other geometric neighbor is not determined so easily. However, we see in Sec. 3.3.2 that systematically iterating through the parent-child hierarchy determines the neighborhood of each grid point.

2.1.2. Multi-dimensional Hierarchical Grids

For the d-dimensional case (with $d \in \mathbb{N}$), we consider the space $\Omega = [0,1]^d$. All the multi-dimensional problems we consider live on Ω. We again start by defining d-dimensional grid points:

Definition 2.1.8 (Grid point)
A d-dimensional grid point p defined via its level $\boldsymbol{l} \in \mathbb{N}^d$ and index $\boldsymbol{i} \in \mathbb{N}^d$ (whereas $i_j < 2^{l_j}$ has to hold $\forall j \in [d]$) is given as

$$p := (\boldsymbol{l}, \boldsymbol{i})\,. \tag{2.11a}$$

The set of all possible d-dimensional grid points $\mathbb{G}_d$ is thus

$$\mathbb{G}_d := \Big\{(\boldsymbol{l}, \boldsymbol{i}) \in \mathbb{N}^d \times \mathbb{N}^d \;\Big|\; i_j < 2^{l_j},\ \forall j \in [d]\Big\} \tag{2.11b}$$

and we call a finite $\mathcal{G} \subset \mathbb{G}_d$ a d-dimensional grid. The projection of $p \in \mathbb{G}_d$ on the jth dimension $\texttt{proj}$ is the one-dimensional grid point given by

$$\begin{aligned}\texttt{proj}\colon \mathbb{G}_d \times \mathbb{N} &\to \mathbb{G}^1\,, \\ ((\boldsymbol{l}, \boldsymbol{i}), j) &\mapsto (l_j, i_j)\end{aligned} \tag{2.11c}$$

which allows us to obtain the absolute coordinate `coord` of p as

$$\begin{aligned}\texttt{coord}\colon \mathbb{G}_d &\to \Omega\,,\\ p &\mapsto \bigtimes_{j\in[d]} \texttt{coord}^1(\texttt{proj}(p,j))\end{aligned} \tag{2.11d}$$

located at the center of its support `support` given by

$$\begin{aligned}\texttt{support}\colon \mathbb{G}_d &\to \mathcal{P}(\Omega)\,,\\ p &\mapsto \prod_{j\in[d]} \texttt{support}^1(\texttt{proj}(p,j))\,.\end{aligned} \tag{2.11e}$$

So, a d-dimensional grid point p can be interpreted as the combination of d one-dimension grid points p_j ($j \in [d]$) whereas the projection of p on dimension j yields p_j.

2.1.2.1. Subspace Grids and Component Grids

To construct the multi-dimensional hierarchical grids, we start by generalizing the one-dimensional subspace grids to multi-dimensional, general subspace grids:

Definition 2.1.9 (Subspace grid)
In d dimensions, the subspace grid of level $\hat{\boldsymbol{l}} \in \mathbb{N}^d$ is given by

$$\texttt{subspaceGrid}_{d,\hat{\boldsymbol{l}}} := \left\{ p \in \mathbb{G}_d \;\middle|\; \forall j \in [d] :\ \texttt{proj}(p,j) \in \texttt{subspaceGrid}^1_{\hat{l}_j} \right\}\,. \tag{2.12}$$

The grid points combined in one subspace are of the same level concerning their hierarchical structure. All level-vectors of the points in one subspace grid are identical. The grid points differ only in their indices, were they take any value permitted by Def. 2.1.2.

Again, we observe what we already saw for the one-dimensional case in Eq. 2.4, that the interiors of the supports in each subspace grid are pairwise disjoint and that the union of all grid points' supports covers Ω:

$$\begin{aligned}&\forall p,q \in \texttt{subspaceGrid}_{d,\hat{\boldsymbol{l}}} \text{ with } p \neq q :\\ &\qquad \text{interior}(\texttt{support}(p)) \cap \text{interior}(\texttt{support}(q)) = \varnothing\,,\end{aligned} \tag{2.13a}$$

and

$$\Omega = \bigcup_{p\in\texttt{subspaceGrid}_{d,\hat{\boldsymbol{l}}}} \texttt{support}(p)\,. \tag{2.13b}$$

We also note that a hierarchy of the subspace grids is inducted by the subspace grids' level-vectors: For a subspace grid s with level-vector $\boldsymbol{l}$, we call the subspace grid s'

with level-vector $\boldsymbol{l}'$ the child in jth dimension of s, if

$$l'_k = \begin{cases} l_k\,, & k \neq j \\ l_k - 1\,, & k = j\,. \end{cases} \quad \forall k \in [d]\ . \tag{2.14}$$

Obviously, each subspace grid has d children, one in each dimension.

Now, we also define the multi-dimensional component grids:

Definition 2.1.10 (Component grid)
In d dimensions, the component grid of level $\hat{\boldsymbol{l}} \in \mathbb{N}^d$ is given by

$$\texttt{componentGrid}_{d,\hat{\boldsymbol{l}}} := \bigcup_{\substack{\boldsymbol{l}\in\mathbb{N}^d \\ \forall j\in[d]:\, l_j \in [\hat{l}_j]}} \texttt{subspaceGrid}_{d,\boldsymbol{l}}\,. \tag{2.15}$$

A component grid is also called *anisotropic full grid*. In each dimension, the distance of two neighboring grid points is dependent on the level in this dimension. This means that said distance in dimension j is the same for all pairs of neighbors. Thus, the resulting grid is of a hyperrectangular form embedded in Ω.

In Fig. 2.2, the construction of the two-dimensional component grid with level-vector $(2, 3)$ from the subspace scheme is shown. It is now important to distinguish between the level-vector of a component grid (or subspace grid) and the level-vector of a grid point. The former denotes the resolution of the component grid in each dimension whereas the latter denotes the fineness of the point in each dimension.

2.1.2.2. Parent-Child Relations between Grid Points

In contrast to the one-dimensional setting, each multi-dimensional grid point can have up to d parents. The grid point is already at the root level in a dimension if the respective entry in the level-vector is one. In this case, there is no parent in that dimension.

Definition 2.1.11 (Parents and ancestors of a point)
The parent in dimension j (with $j \in [d]$) of a point $p = (\boldsymbol{l}, \boldsymbol{i})$ with $l_j > 1$ is given by

$$\texttt{parent}\colon \mathbb{G}_d \times \mathbb{N} \to \mathbb{G}_d\,,$$
$$((\boldsymbol{l},\boldsymbol{i})\,, j) \mapsto \begin{cases} (\boldsymbol{l},\boldsymbol{i})\ , & l_j = 1\,, \\ \big((l_1,\ldots,l_{j-1},l_j-1,l_{j+1},\ldots,l_d)\,, \\ \quad (i_1,\ldots,i_{j-1},\texttt{parent-index}(i_j),i_{j+1},\ldots,i_d)\big)\,, & \text{else.} \end{cases} \tag{2.16a}$$

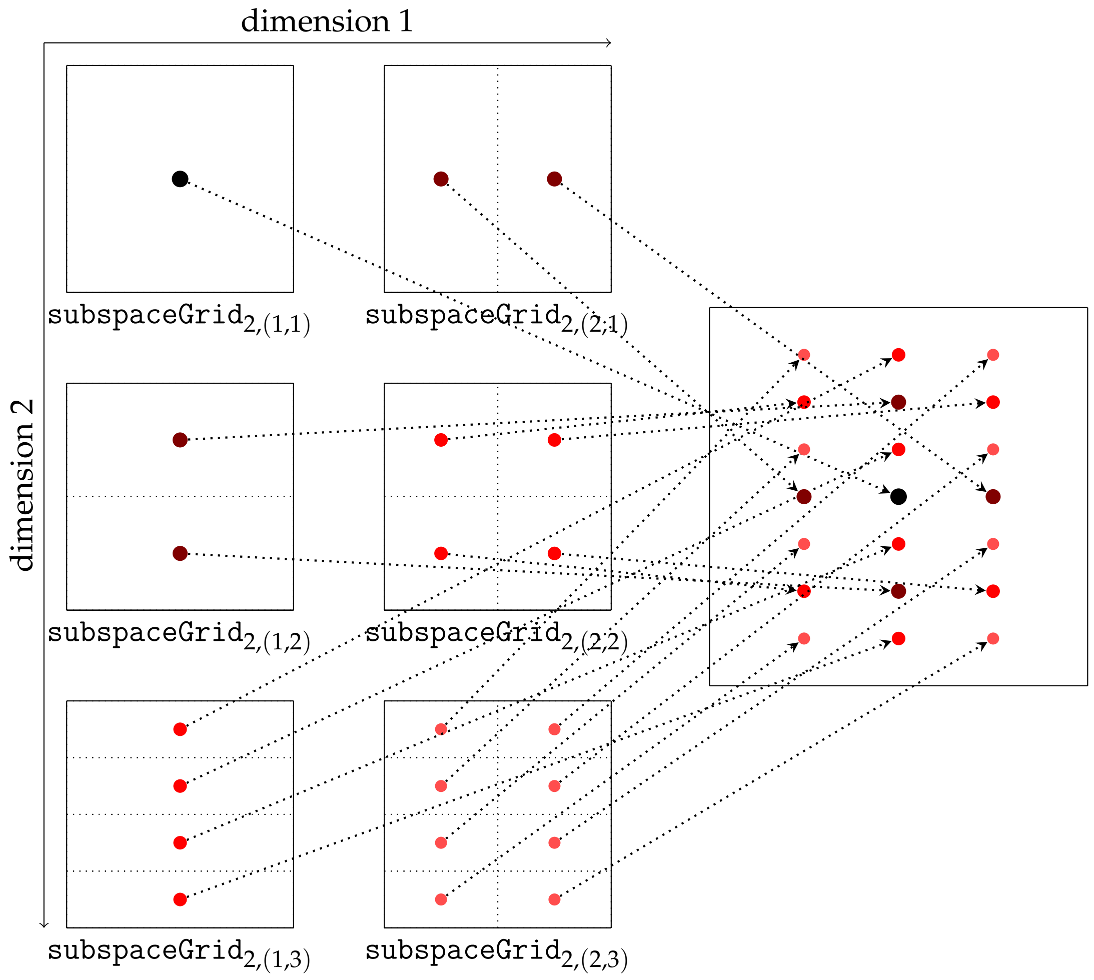


Figure 2.2.: Construction of `componentGrid`$_{2,(2,3)}$ via the six two-dimensional subspace grids of level $(1,1)$, $(1,2)$, $(1,3)$, $(2,1)$, $(2,2)$ and $(2,3)$. In the visualization of the subspaces, the supports of the grid points are depicted with dotted lines. The arrows link the grid points in the subspace grids and the corresponding grid points in `componentGrid`$_{2,(2,3)}$.

All parents of p are then given by

$$\begin{aligned}\texttt{parents}\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \{\texttt{parent}(p,j) \quad | \quad j \in [d]\}\ .\end{aligned} \tag{2.16b}$$

With $\texttt{parents}_0(p) := \{p\}$, we then recursively define for $k \in \mathbb{N}$

$$\begin{aligned}\texttt{parents}_k\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \bigcup_{q\in\texttt{parents}_{k-1}(p)} \texttt{parents}(q)\end{aligned} \tag{2.16c}$$

to end up with

$$\begin{aligned}\texttt{ancestors}\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \bigcup_{k\in\mathbb{N}} \texttt{parents}_k(p)\end{aligned} \tag{2.16d}$$

and

$$\begin{aligned}\texttt{ancestors}^*\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \bigcup_{k\in\mathbb{N}_0} \texttt{parents}_k(p)\,.\end{aligned} \tag{2.16e}$$

Similar to Def. 2.1.4, $\texttt{parents}_k(p)$ contains the parents of degree k of p. $\texttt{ancestors}(p)$ and $\texttt{ancestors}^*(p)$ then contain the parents of all degrees with the $\texttt{ancestors}^*(p)$ also containing p itself.

Analogously, we define the children of a point. Each d-dimensional grid point has $2d$ children, two in each dimension.

Definition 2.1.12 (Children of a point)
The children in dimension j (with $j \in [d]$) of a point $p = (\boldsymbol{l}, \boldsymbol{i})$ are given by

$$\begin{aligned}\texttt{left-child}\colon \mathbb{G}_d \times \mathbb{N} &\to \mathbb{G}_d\,,\\ ((\boldsymbol{l},\boldsymbol{i})\,,j) &\mapsto \big((l_1,\ldots,l_{j-1},l_j+1,l_{j+1},\ldots,l_d)\,,\\ &\qquad (i_1,\ldots,i_{j-1},2i_j-1,i_{j+1},\ldots,i_d)\big)\end{aligned} \tag{2.17a}$$

and

$$\begin{aligned}\texttt{right-child}\colon \mathbb{G}_d \times \mathbb{N} &\to \mathbb{G}_d\,,\\ ((\boldsymbol{l},\boldsymbol{i})\,,j) &\mapsto \big((l_1,\ldots,l_{j-1},l_j+1,l_{j+1},\ldots,l_d)\,,\\ &\qquad (i_1,\ldots,i_{j-1},2i_j+1,i_{j+1},\ldots,i_d)\big)\ .\end{aligned} \tag{2.17b}$$

All children of p are then given by

$$\begin{aligned}\texttt{children}\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \{\texttt{left-child}(p,j), \texttt{right-child}(p,j) \quad | \quad j \in [d]\}\;.\end{aligned} \tag{2.17c}$$

With $\texttt{children}_0(p) := \{p\}$, we then recursively define for $k \in \mathbb{N}$

$$\begin{aligned}\texttt{children}_k\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \bigcup_{q \in \texttt{children}_{k-1}(p)} \texttt{children}(q)\end{aligned} \tag{2.17d}$$

to end up with

$$\begin{aligned}\texttt{descendants}\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \bigcup_{k \in \mathbb{N}} \texttt{children}_k(p)\end{aligned} \tag{2.17e}$$

and

$$\begin{aligned}\texttt{descendants}^*\colon \mathbb{G}_d &\to \mathcal{P}(\mathbb{G}_d)\,,\\ p &\mapsto \bigcup_{k \in \mathbb{N}_0} \texttt{children}_k(p)\,.\end{aligned} \tag{2.17f}$$

Similar to Def. 2.1.5, $\texttt{children}_k(p)$ contains all children of degree k of p. $\texttt{descendants}(p)$ and $\texttt{descendants}^*(p)$ then contain the children of all degrees of p with $\texttt{descendants}^*(p)$ also including p itself. Because the recursive construction of the children always yields further additional grid points, those two sets are infinite. Thus, we use those sets only by intersecting them with the grids we are actually working with.

2.1.2.3. Neighborhood of Grid Points

We want to identify the geometrically neighboring grid points. In each dimension, every grid point $p = (\boldsymbol{l}, \boldsymbol{i})$ can have a left neighbor and a right neighbor. However, if no other grid points are aligned left (or right) of p, there is no neighbor in that direction. In this case, it is important for some algorithms to treat the boundary projections in those dimensions as virtual neighbors. For a grid point p and dimension j, we thus define the boundary projections.

Definition 2.1.13 (Boundary projections of a grid point)
For a grid $\mathcal{G} \subset \mathbb{G}_d$, a point $p \in \mathbb{G}_d$ and a dimension j (with $j \in [d]$), the boundary

projections of p in dimension j are given via

$$\begin{aligned}\texttt{boundary-projection-left}_{\mathcal{G}}\colon \mathbb{G}_d \times \mathbb{N} &\to \mathbb{N}_0 \times \mathbb{N}_0\,,\\ ((\boldsymbol{l},\boldsymbol{i}),j) &\mapsto \big((l_1,\dots,l_{j-1},0,l_{j+1},\dots,l_d)\,,\\ &\qquad (i_1,\dots,i_{j-1},0,i_{j+1},\dots,i_d)\big)\,,\end{aligned} \tag{2.18a}$$

and

$$\begin{aligned}\texttt{boundary-projection-right}_{\mathcal{G}}\colon \mathbb{G}_d \times \mathbb{N} &\to \mathbb{N}_0 \times \mathbb{N}_0\,,\\ ((\boldsymbol{l},\boldsymbol{i}),j) &\mapsto \big((l_1,\dots,l_{j-1},0,l_{j+1},\dots,l_d)\,,\\ &\qquad (i_1,\dots,i_{j-1},1,i_{j+1},\dots,i_d)\big)\,.\end{aligned} \tag{2.18b}$$

The set of all boundary projections for $\mathcal{G}$ is then

$$\begin{aligned}\mathcal{B}_{\mathcal{G}} = \big\{&\texttt{boundary-projection-left}_{\mathcal{G}}(p,j), \texttt{boundary-projection-right}_{\mathcal{G}}(p,j)\\ &\mid\ p \in \mathcal{G},\, j \in [d]\big\}\,.\end{aligned} \tag{2.18c}$$

With the boundary projections well defined, for a set of grid points $\mathcal{G} \subset \mathbb{G}_d$, a point $p \in \mathcal{G}$ and a dimension j, we now look at the grid points `left-aligned`, which are aligned left of p in dimension j, and `right-aligned`, which are aligned right of p in dimension j:

$$\begin{aligned}\texttt{left-aligned}_{\mathcal{G}}\colon \mathcal{G} \times \mathbb{N} &\to \mathcal{P}(\mathcal{G} \cup \mathcal{B}_{\mathcal{G}})\,,\\ ((\boldsymbol{l},\boldsymbol{i}),j) &\mapsto \Big\{(\boldsymbol{l}',\boldsymbol{i}') \in \mathbb{G}_d \ \Big|\ \texttt{coord}^1\big(\texttt{proj}_{(\boldsymbol{l}',\boldsymbol{i}'),j}\big) < \texttt{coord}^1\big(\texttt{proj}_{(\boldsymbol{l},\boldsymbol{i}),j}\big)\\ &\qquad \text{and } \forall k \in [d]\setminus\{j\}:\ \texttt{proj}_{(\boldsymbol{l},\boldsymbol{i}),k} = \texttt{proj}_{(\boldsymbol{l}',\boldsymbol{i}'),k}\Big\}\\ &\qquad \cup \big\{\texttt{boundary-projection-left}_{\mathcal{G}}((\boldsymbol{l},\boldsymbol{i}),j)\big\}\end{aligned} \tag{2.19a}$$

and

$$\begin{aligned}\texttt{right-aligned}_{\mathcal{G}}\colon \mathcal{G} \times \mathbb{N} &\to \mathcal{P}(\mathcal{G} \cup \mathcal{B}_{\mathcal{G}})\,,\\ ((\boldsymbol{l},\boldsymbol{i}),j) &\mapsto \Big\{(\boldsymbol{l}',\boldsymbol{i}') \in \mathbb{G}_d \ \Big|\ \texttt{coord}^1\big(\texttt{proj}_{(\boldsymbol{l}',\boldsymbol{i}'),j}\big) > \texttt{coord}^1\big(\texttt{proj}_{(\boldsymbol{l},\boldsymbol{i}),j}\big)\\ &\qquad \text{and } \forall k \in [d]\setminus\{j\}:\ \texttt{proj}_{(\boldsymbol{l},\boldsymbol{i}),k} = \texttt{proj}_{(\boldsymbol{l}',\boldsymbol{i}'),k}\Big\}\\ &\qquad \cup \big\{\texttt{boundary-projection-right}_{\mathcal{G}}((\boldsymbol{l},\boldsymbol{i}),j)\big\}\,.\end{aligned} \tag{2.19b}$$

Ultimately, we are now able to define the neighbors and the neighborhood of a grid point in the context of a set of grid points.

Definition 2.1.14 (Neighborhood of grid points)
For a grid $\mathcal{G} \subset \mathbb{G}_d$ and a point $p \in \mathcal{G}$, we define the neighbors of p in dimension j (with $j \in [d]$) as

$$\begin{aligned}\texttt{neighbor}_{\mathcal{G}} \colon \mathcal{G} \times \mathbb{N} \times \{\text{left}, \text{right}\} &\to \mathcal{G} \cup \mathcal{B}_{\mathcal{G}}\,, \\ (p, j, y) &\mapsto \begin{cases} \underset{q \in \texttt{left-aligned}_{\mathcal{G}}(p,j)}{\arg\max} \quad \texttt{coord}^1(\texttt{proj}_{q,j})\,, & y = \text{left}\,, \\ \underset{q \in \texttt{right-aligned}_{\mathcal{G}}(p,j)}{\arg\min} \quad \texttt{coord}^1(\texttt{proj}_{q,j})\,, & y = \text{right}\,. \end{cases}\end{aligned} \tag{2.20a}$$

The neighborhood of p is then

$$\begin{aligned}\texttt{neighborhood}_{\mathcal{G}} \colon \mathcal{G} &\to \mathcal{P}(\mathcal{G} \cup \mathcal{B}_{\mathcal{G}})\,, \\ p &\mapsto \left\{ \texttt{neighbor}_{\mathcal{G}}(p, j, y) \;\middle|\; j \in [d]\,, y \in \{\text{left}, \text{right}\} \right\}\,.\end{aligned} \tag{2.20b}$$

We notice that either the point itself or its neighbor is a leaf grid point towards its partner. Thus, the corresponding child in this dimension and direction is not present in the grid. Generally, we want to emphasize that the neighbors and the neighborhood of grid points are only well defined in the context of an actual grid that this point is part of. While boundary projections of a point are independent of the grid this point is part of, the geometric neighbors change with the grid, even if the point in question remains the same.

2.1.2.4. Regular Full Grids and Regular Sparse Grids

The construction of a regular hierarchical full grid results directly from the definition of a component grid:

Definition 2.1.15 (Regular full grid)
In d dimensions, the regular full grid of level $L \in \mathbb{N}$ is given by

$$\texttt{fullGrid}_{d,L} := \texttt{componentGrid}_{d,\mathbf{l}}\,, \tag{2.21}$$

where $l_j = L \;\forall j \in [d]$.

Thus, a regular full grid is a component grid with uniform level-vector.

To construct a regular sparse grid, we refer to the previously defined subspace grids:

Definition 2.1.16 (Regular sparse grid)
In d dimensions, a regular sparse grid of level $L \in \mathbb{N}$ is given by

$$\texttt{sparseGrid}_{d,L} := \bigcup_{\substack{\boldsymbol{l} \in \mathbb{N}^d \\ \|\boldsymbol{l}\|_1 < L+d}} \texttt{subspaceGrid}_{d,\boldsymbol{l}} . \tag{2.22}$$

Interestingly, $\texttt{root}_d := ((1,\ldots,1),(1,\ldots,1))$ being the d-dimensional root point, it holds that

$$\texttt{sparseGrid}_{d,L} = \bigcup_{0 \le k \le L-1} \texttt{children}_k(\texttt{root}_d) . \tag{2.23}$$

So, the sparse grid is either constructed by aggregating the subspace grids with corresponding levels or by starting with the root point and adding the children up to the respective degree.

The construction of both a regular full grid and a regular sparse grid of level 3 in two dimensions from the subspace scheme is depicted in Fig. 2.3. It can be seen that the subspace grids containing more grid points but with smaller support are added to the full grid but not to the sparse grid. With increasing level, only those points are added to the sparse grid that have the next-largest support size, which are those points that are expected to contribute most to the problem. In [18] it is shown that the size of $\texttt{sparseGrid}_{d,L}$ (which is proportional to the cost of any grid-based function approximation) is given by

$$\left|\texttt{sparseGrid}_{d,L}\right| = \sum_{j=0}^{L-1} 2^j \binom{d-1+j}{d-1} \in \mathcal{O}\left(2^L L^{d-1}\right) . \tag{2.24}$$

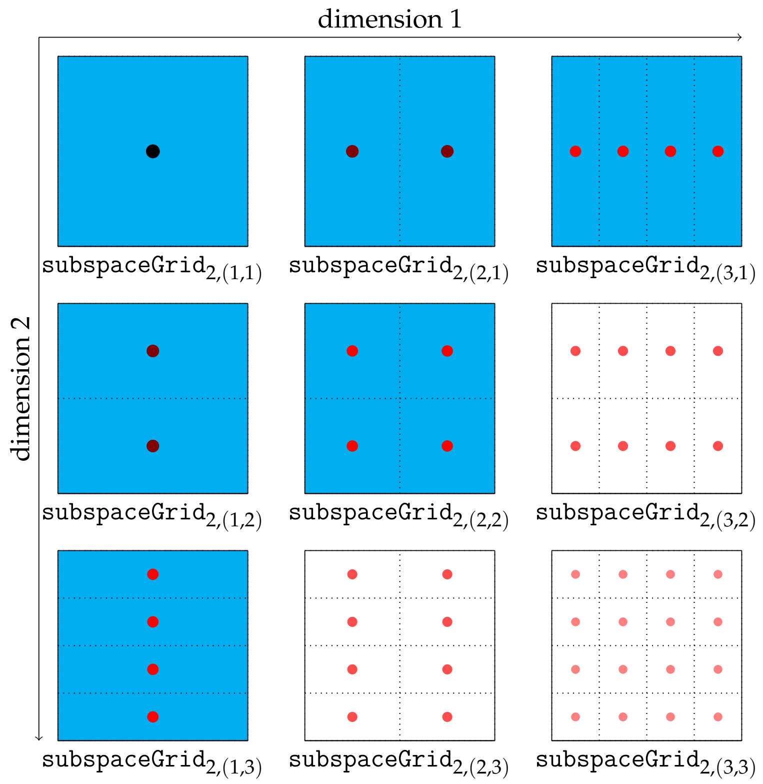
which grows much slower in d than the size of a full grid given by

$$\left|\texttt{fullGrid}_{d,L}\right| = \left(2^L - 1\right)^d \in \mathcal{O}\left(2^{d \cdot L}\right) . \tag{2.25}$$

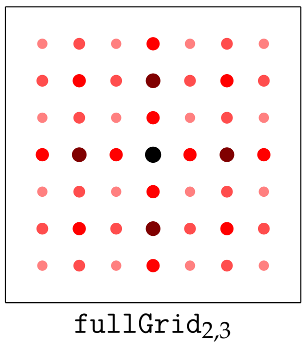
Thus, the full grid contains more degrees of freedom and thus, approximating an unknown function with a full grid is more precise than with a sparse grid. However, the grid points we omit in the sparse grid compared to the full grid are those with small support, i.e. the points that are expected to contribute less to the solution than the ones with larger support. Indeed, with increasing level, the grid points that are being added to the regular sparse grid all have the same support size. In contrast, the points being added to the full grid are of mixed support size. So, in the context of hierarchical grids, regular sparse grids offer a systematic approach of constructing an approximation.

2.1.2.5. Adaptive Sparse Grids

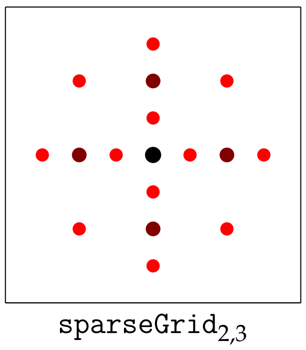
After we initially learned a dataset with an a priori grid, we might want to adapt the grid to the data and redo the learning step. Two categories of adapting the grid exist:


dimension 1
dimension 2
$\texttt{subspaceGrid}_{2,(1,1)}$
$\texttt{subspaceGrid}_{2,(2,1)}$
$\texttt{subspaceGrid}_{2,(3,1)}$
$\texttt{subspaceGrid}_{2,(1,2)}$
$\texttt{subspaceGrid}_{2,(2,2)}$
$\texttt{subspaceGrid}_{2,(3,2)}$
$\texttt{subspaceGrid}_{2,(1,3)}$
$\texttt{subspaceGrid}_{2,(2,3)}$
$\texttt{subspaceGrid}_{2,(3,3)}$

(a) The two-dimensional subspace scheme of level 3. All subspace grids form the full grid (Subfig. b), the subspace grids in blue form the sparse grid of level 3 (Subfig. c).



(b) The two-dimensional full grid of level 3. The grid contains 49 points.



(c) The two-dimensional sparse grid of level 3. The grid contains 17 points.

Figure 2.3.: The two-dimensional subspace scheme in Subfig. a shows, how both the two-dimensional full grid (Subfig. b) and sparse grid (Subfig. c) of level 3 are constructed.

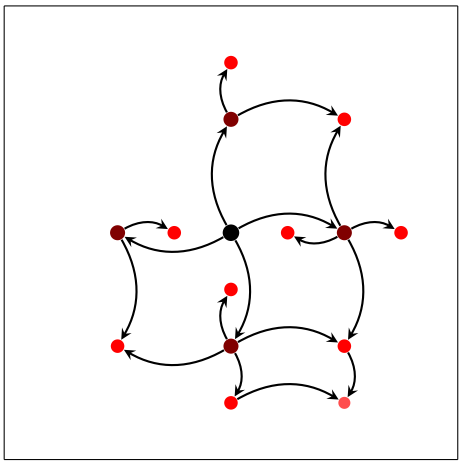


Figure 2.4.: A spatially adaptive sparse grid in two dimensions. The parent-child relationships are depicted as arrows from the parents to their respective children.

With spatial adaptivity [79], we identify grid points that lie in parts of the domain we want to know more about and refine the grid in the area of those points. In contrast, with dimensional adaptivity [37] we identify dimensions in which we desire a finer resolution and then add subspace grids that have a finer level in those dimensions. Those concepts of spatial and dimensional adaptivity are presented in the following.

2.1.2.5.1 Spatially Adaptive Sparse Grids

When representing a function with an a priori spare grid $\mathcal{G}$, we might want to adapt the grid to the problem at hand. During this process of adding grid points (refinement) and removing grid points (coarsening), the grid changes from a regular sparse grid to a spatially adaptive sparse grid, or in general, just a *sparse grid*. Let's give a formal definition of what we understand as a sparse grid:

> **Definition 2.1.17 (Sparse grid)**
> A grid $\mathcal{G} \in \mathbb{G}_d$ that satisfies
>
> $$\forall p \in \mathcal{G} : \texttt{ancestors}(p) \subseteq \mathcal{G} \tag{2.26}$$
>
> is called a sparse grid.

So, as long as the hierarchical ancestors of all grid points in $\mathcal{G}$ are in $\mathcal{G}$ as well, the sparse grid property is fulfilled. It occurs that this also holds for the full grids per Def. 2.1.15. Indeed, starting with $\texttt{root}_d$ before adding and removing grid points in the convenient order, an arbitrary full grid can be constructed. So, in order to be as flexible with the definition as needed, those cases are not explicitly excluded from being a sparse grid too. An example of a (spatially adaptive) sparse grid is shown in Fig. 2.4 together with the parent-child relations. Every point in the grid is reached by recursively visiting the children, starting with $\texttt{root}_d$.

To decide which areas in Ω require a finer or a coarser resolution, we develop problem-based indicators that assign a score to each grid point, which we use to select candidates for refinement or coarsening.

Refinement When refining a grid point $p \in \mathcal{G}$, we add its children and all of their ancestors to the grid. For this, we define the operator

$$\begin{aligned} \texttt{refine}_{\mathcal{G}} \colon \mathcal{G} &\to \mathcal{P}(\mathbb{G}_d)\,, \\ p &\mapsto \mathcal{G} \cup \bigcup_{q \in \texttt{children}(p)} \texttt{ancestors}^*(q)\,. \end{aligned} \tag{2.27}$$

Coarsening Upon coarsening a grid point $p \in \mathcal{G}$, we remove it and all of its children from the grid. This operation is given by the operator

$$\begin{aligned} \texttt{coarsen}_{\mathcal{G}} \colon \mathcal{G} &\to \mathcal{P}(\mathbb{G}_d)\,, \\ p &\mapsto \mathcal{G} \setminus \texttt{descendants}^*(p)\,. \end{aligned} \tag{2.28}$$

When $\mathcal{G}$ is a sparse grid per Def. 2.1.17 and $p \in \mathcal{G}$, both $\texttt{refine}_{\mathcal{G}}(p)$ and $\texttt{coarsen}_{\mathcal{G}}(p)$ return sparse grids too.

The decision which grid points to refine depends on the problem setting at hand. [79] proposed criteria that work well in the case of function interpolation or quadrature. However, looking at data mining tasks such as classification requires refinement criteria tailored to the problem. We propose and discuss such criteria for classification in Sec. 3.3.2.

2.1.2.5.2 Dimensional Adaptivity

When mining certain datasets, we may desire different resolutions for different dimensions in our grid. To achieve that, we add entire hierarchical subspace grids to the grid. $\texttt{subspaceGrid}_{d,\boldsymbol{l}}$ is allowed to be added to the grid as long as all $\texttt{subspaceGrid}_{d,\boldsymbol{l}'}$ with $\boldsymbol{l}' < \boldsymbol{l}$ are also added or present. We use this method especially when employing the combination grid technique (ref. Sec. 2.3.2).

2.2. Basis Functions

To complete the grid-based function representation, we need to discuss the basis functions ϕ from Eq. 2.1. First, we define the atomic building blocks necessary to construct different basis function types. The three types we look at are the linear basis functions (Sec. 2.2.1), the modified linear basis functions (Sec. 2.2.2) and the kinked linear basis functions (Sec. 2.2.3). We propose the latter type because it allows high-dimensional approximations to extrapolate towards a non-zero boundary without oscillation. All of the three basis function types live on the support of their associated grid points. They only differ on how function values towards the boundary of the domain are extrapolated. The term "linear" might be misleading, as all the three types of basis functions are piecewise linear (or the be more precise, piecewise affine). Because the denomination "linear" is used in the literature, we also stick to it.

To prepare the tools we later need, we also specifically look at the inner product of two basis functions for all types in their respective sections.

As the most primitive building block, we define the left (augmenting) and right (decreasing) part of the standard hat function centered at zero:

$$\begin{aligned} \varphi_l \colon \mathbb{R} &\to [0,1]\,, \\ x &\mapsto \begin{cases} x+1\,, & x \in [-1,0]\,, \\ 0\,, & \text{else}\,, \end{cases} \\ \varphi_r \colon \mathbb{R} &\to [0,1]\,, \\ x &\mapsto \begin{cases} -x+1\,, & x \in [0,1]\,, \\ 0\,, & \text{else}\,. \end{cases} \end{aligned} \tag{2.29}$$

This serves us to define the mother of all hat functions as

$$\begin{aligned} \varphi \colon \mathbb{R} &\to [0,1]\,, \\ x &\mapsto \begin{cases} 1\,, & x = 0\,, \\ \varphi_l(x) + \varphi_r(x)\,, & \text{else}\,. \end{cases} \end{aligned} \tag{2.30}$$

Now, we are ready to construct the linear basis functions, the modified linear basis functions and the kinked linear basis functions.

2.2.1. Linear Basis

To obtain the linear basis function for a grid point p, we center φ from Eq. 2.30 at $\texttt{coord}_p$ and scale it to $\texttt{support}_p$.

Definition 2.2.1 (One-dimensional linear basis function)
Let $p = (l, i)$ be a one-dimensional grid point. The one-dimensional linear basis

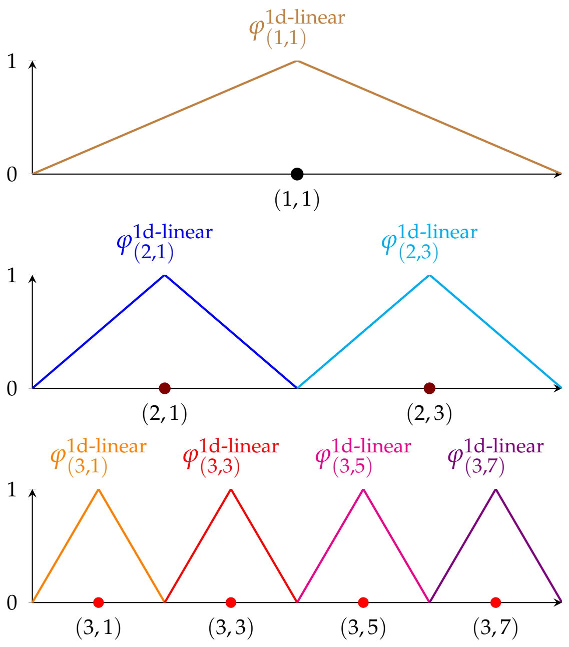


Figure 2.5.: One-dimensional linear basis functions up to level 3 constructed to dilatation and shifting of φ. It extrapolates to zero towards the boundary on all levels.

function for p is then given by

$$\begin{aligned} \varphi_p^{\text{1d-linear}} \colon \Omega^1 &\to [0,1]\,, \\ x &\mapsto \varphi(2^l x - i)\,. \end{aligned} \tag{2.31}$$

A visualization of the one-dimensional linear basis functions for all grid points in $\texttt{componentGrid}_3^1$ is found in Fig. 2.5. The peak value of $\varphi_p^{\text{1d-linear}}$ located at $\texttt{coord}_p^1$ is always 1 and the left and right branch are scaled so that they reach 0 at the boundary of $\texttt{support}_p^1$. Thus, we use the linear basis function only if we assume that the function value is zero at all boundaries of Ω.

This definition generalizes straightforward to the multi-dimensional case:

Definition 2.2.2 (Linear basis function)
Let $p \in \mathbb{G}_d$ be a d-dimensional grid point. The linear basis function for p is then given by

$$\begin{aligned} \varphi_p^{\text{linear}} \colon \Omega &\to [0,1]\,, \\ \boldsymbol{x} &\mapsto \prod_{j=1}^{d} \varphi_{\texttt{proj}_{p,j}}^{\text{1d-linear}}(x_j)\,. \end{aligned} \tag{2.32}$$

A visualization of the two-dimensional linear basis functions for all grid points in $\texttt{componentGrid}_{2,(2,2)}$ is found in Fig. 2.6. It can be seen that the peaks of the so called "pagodas" are located directly above the grid points. While this is true for the linear

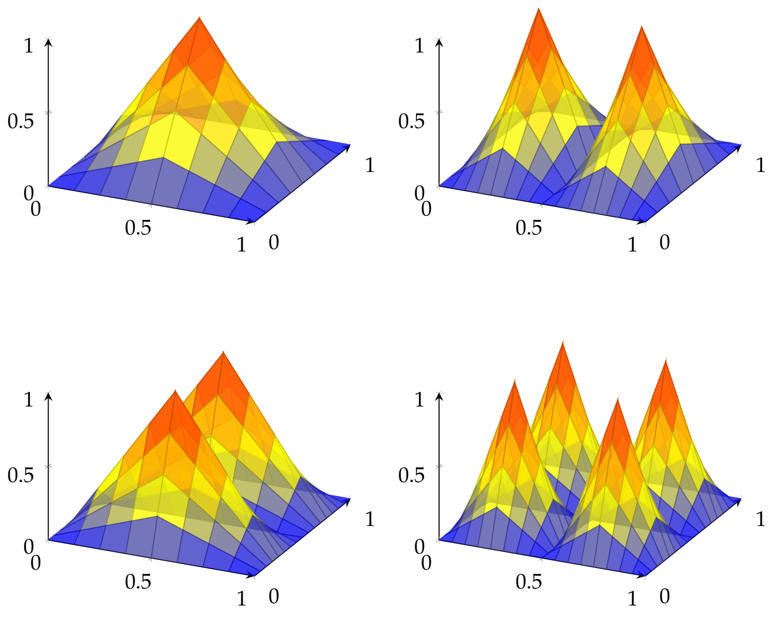


Figure 2.6.: Two-dimensional linear basis functions up to level 2. We call the shape of one basis function *pagoda*. It is noteworthy that this shape differs from a pyramid. For the pagoda, the cut along a diagonal is a polynomial of degree d.

basis functions, there are other basis functions (e.g. polynomial basis functions) whose peaks are dislocated from the grid point. For details we refer to [79].

Inner Product With $p_1, p_2 \in \mathbb{G}_d$, we want to calculate $\left\langle \varphi_{p_1}^{\text{linear}}, \varphi_{p_2}^{\text{linear}} \right\rangle_{L_2}$. It holds that

$$\left\langle \varphi_{p_1}^{\text{linear}}, \varphi_{p_2}^{\text{linear}} \right\rangle_{L_2} = \int_0^1 \varphi_{p_1}^{\text{linear}}(\boldsymbol{x}) \cdot \varphi_{p_2}^{\text{linear}}(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \prod_{j=1}^{d} \int_0^1 \varphi_{\texttt{proj}_{p_1,j}}^{\text{1d-linear}}(x) \cdot \varphi_{\texttt{proj}_{p_2,j}}^{\text{1d-linear}}(x) \mathrm{d}x \,. \tag{2.33}$$

Thus, we are looking at two one-dimensional points $q_1 = (l_1, i_1)$ and $q_2 = (l_2, i_2)$ and calculate

$$r_{q_1,q_2}^{\text{linear}} := \left\langle \varphi_{q_1}^{\text{1d-linear}}, \varphi_{q_2}^{\text{1d-linear}} \right\rangle_{L_2} = \int_0^1 \varphi_{q_1}^{\text{1d-linear}}(x) \cdot \varphi_{q_2}^{\text{1d-linear}}(x) \mathrm{d}x \,. \tag{2.34}$$

It holds that

$$r_{q_1,q_2}^{\text{linear}} = \begin{cases} \frac{2^{1-l_1}}{3}\,, & q_1 = q_2\,, \\ \frac{1-2^{l_1}\left|\frac{i_2}{2^{l_2}} - \frac{i_1}{2^{l_1}}\right|}{2^{l_2}}\,, & l_1 < l_2 \text{ and } \max\left\{\frac{i_1-1}{2^{l_1}}, \frac{i_2-1}{2^{l_2}}\right\} < \min\left\{\frac{i_1+1}{2^{l_1}}, \frac{i_2+1}{2^{l_2}}\right\}, \\ \frac{1-2^{l_2}\left|\frac{i_1}{2^{l_1}} - \frac{i_2}{2^{l_2}}\right|}{2^{l_1}}\,, & l_2 < l_1 \text{ and } \max\left\{\frac{i_1-1}{2^{l_1}}, \frac{i_2-1}{2^{l_2}}\right\} < \min\left\{\frac{i_1+1}{2^{l_1}}, \frac{i_2+1}{2^{l_2}}\right\}, \\ 0\,, & \text{else}\,. \end{cases} \tag{2.35}$$

For the proof, see Sec. A.1.1.

2.2.2. Modified Linear Basis

With the linear basis, the assumed value of f at the boundary of Ω is zero. However, this assumption might not be valid in many settings. One method to model non-zero values at the boundaries is to introduce grid points at the boundary and define basis functions anchored there. This technique increases the number of grid points (thus the model size) significantly with growing d, which is only feasible for lower dimensional problems. However, the resulting number of grid points is already too high for moderate dimensionalities which is why we instead use modified linear basis functions [79] to represent non-zero values at the boundaries.

Definition 2.2.3 (One-dimensional modified linear basis function)
Let $p = (l, i)$ be a one-dimensional grid point. The one-dimensional modified linear basis function for p is then given by

$$\begin{aligned} \varphi_p^{\text{1d-modlinear}} \colon \Omega^1 &\to [0,2]\,, \\ x &\mapsto \begin{cases} 1\,, & (l,i) = (1,1)\,, \\ 2 \cdot \varphi_r(2^{l-1}x)\,, & l > 1, i = 1\,, \\ 2 \cdot \varphi_l(2^{l-1}(x-1))\,, & l > 1, i = 2^l - 1\,, \\ \varphi_p^{\text{1d-linear}}(x)\,, & \text{else.} \end{cases} \end{aligned} \tag{2.36}$$

The modified linear basis for all grid points in $\texttt{componentGrid}_3^1$ is depicted in Fig. 2.7. On the first level, we model the value at the center of Ω as the base value of f over the whole domain and use the left-most and right-most grid points at each level to correct values towards the boundary. All basis functions for grid points from $(l, 2)$ to $(l, 2^l - 2)$ are equipped with the linear basis functions (see Sec. 2.2.1).

The generalization to more dimensions is (as in the linear case) given via the tensor product approach:

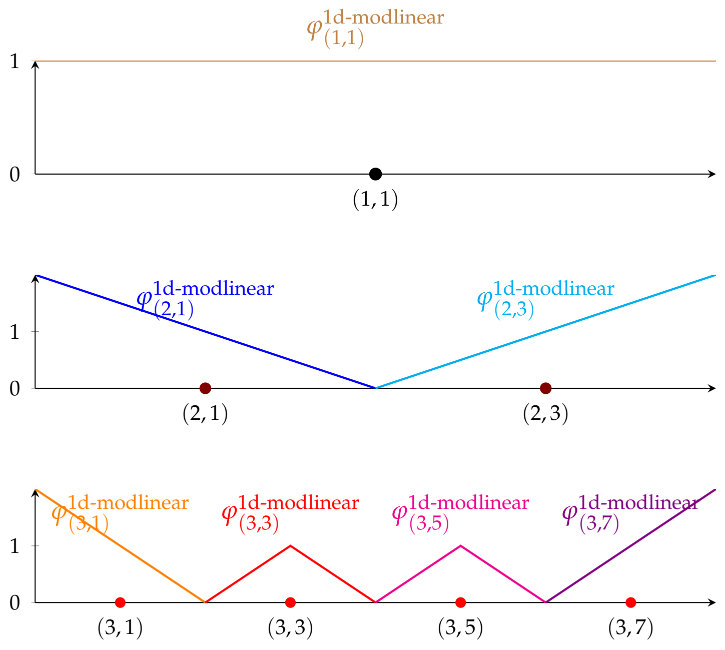


Figure 2.7.: One-dimensional modified linear basis functions up to level 3. In contrast to the linear basis functions, the modified linear basis is constant 1 on level 1 and otherwise extrapolates to a value of 2 at the boundary.

Definition 2.2.4 (Modified linear basis function)
Let $p \in \mathbb{G}_d$ be a d-dimensional grid point. The modified linear basis function for p is then given by

$$\begin{aligned} \varphi_p^{\text{modlinear}} &: \Omega \to [0, 2^d]\,, \\ \boldsymbol{x} &\mapsto \prod_{j=1}^{d} \varphi^{\text{1d-modlinear}}_{\texttt{proj}_{p,j}}(x_j)\,. \end{aligned} \tag{2.37}$$

A visualization of all two-dimensional modified linear basis functions up to level 3 is given in Fig. 2.8. For a grid point p at a corner (i.e. a grid point $(\boldsymbol{l}, \boldsymbol{i})$ with $l_j \geq 2$ and $i_j = 1$ or $i_j = 2^{l_j} - 1 \ \forall j \in [d]$), the value towards the respective corner of Ω extrapolates to 4. In higher dimensionalities d, the value in the multi-dimensional corner extrapolates to 2^d. While this is acceptable for low-dimensional problems, it might lead to problems in high-dimensional settings. The extrapolated value in the region close to the corner can explode even for very small values of α_p which might lead to highly oscillating behavior when adding additional grid points via spatial adaptivity.

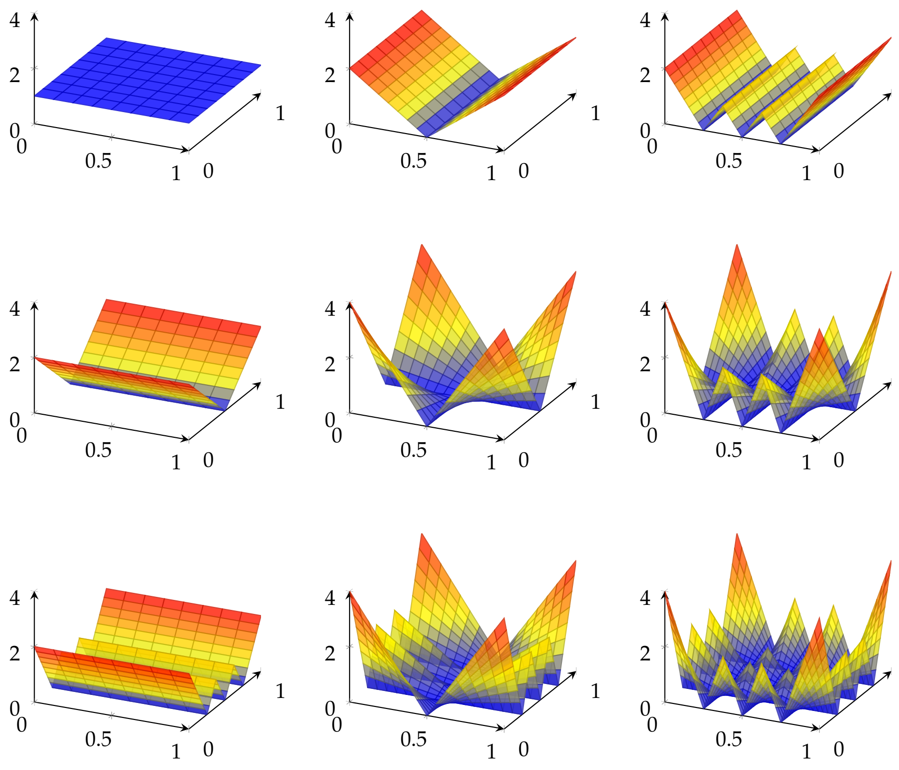


Figure 2.8.: Two-dimensional modified linear basis functions up to level 3. Due to the extrapolation to level 2 for the one-dimensional case, the modified linear basis extrapolates to 2^d in the corners for the d-dimensional case. The modified linear basis differs only from the linear basis for the grid points adjacent to the boundary of Ω.

Inner Product Again, with $p_1, p_2 \in \mathbb{G}_d$, we want to derive a closed form for $\left\langle \varphi_{p_1}^{\text{modlinear}}, \varphi_{p_2}^{\text{modlinear}} \right\rangle_{L_2}$. As for the linear case, it holds that

$$\left\langle \varphi_{p_1}^{\text{modlinear}}, \varphi_{p_2}^{\text{modlinear}} \right\rangle_{L_2} = \prod_{j=1}^{d} \int_0^1 \varphi_{\mathtt{proj}_{p_1,j}}^{\text{1d-modlinear}}(x) \cdot \varphi_{\mathtt{proj}_{p_2,j}}^{\text{1d-modlinear}}(x) \mathrm{d}x \,, \tag{2.38}$$

and we are again looking at two one-dimensional points $q_1 = (l_1, i_1)$ and $q_2 = (l_2, i_2)$ and calculate

$$r_{q_1,q_2}^{\text{modlinear}} := \left\langle \varphi_{q_1}^{\text{1d-modlinear}}, \varphi_{q_2}^{\text{1d-modlinear}} \right\rangle_{L_2} = \int_0^1 \varphi_{q_1}^{\text{1d-modlinear}}(x) \cdot \varphi_{q_2}^{\text{1d-modlinear}}(x) \mathrm{d}x \,. \tag{2.39}$$

It holds, that,

$$r_{q_1,q_2}^{\text{modlinear}} = \begin{cases} 1\,, & l_1 = l_2 = 1\,, \\ \frac{2^{3-l_1}}{3}\,, & q_1 = q_2,\, l_1 > 1,\, (i_1 = 1 \text{ or } i_1 = 2^{l_1} - 1)\,, \\ 2^{1-l_2}\,, & l_1 = 1,\, (i_2 = 1 \text{ or } i_2 = 2^{l_2} - 1)\,, \\ 2^{1-l_1}\,, & l_2 = 1,\, (i_1 = 1 \text{ or } i_1 = 2^{l_1} - 1)\,, \\ 2^{-l_2}\,, & l_1 = 1,\, 1 < i_2 < 2^{l_2} - 1\,, \\ 2^{-l_1}\,, & l_2 = 1,\, 1 < i_1 < 2^{l_1} - 1\,, \\ \frac{4}{2^{l_2}} \left(1 - \frac{1}{3} 2^{l_1 - l_2}\right)\,, & l_1 < l_2,\, \left(i_1 = i_2 = 1 \text{ or } \left(i_1 = 2^{l_1} - 1,\, i_2 = 2^{l_2} - 1\right)\right)\,, \\ \frac{4}{2^{l_1}} \left(1 - \frac{1}{3} 2^{l_2 - l_1}\right)\,, & l_2 < l_1,\, \left(i_1 = i_2 = 1 \text{ or } \left(i_1 = 2^{l_1} - 1,\, i_2 = 2^{l_2} - 1\right)\right)\,, \\ \frac{2 - 2^{l_1 - l_2} i_2}{2^{l_2}}\,, & l_1 < l_2,\, \left(i_1 = 1,\, 1 < i_2 < \frac{2^{l_2}}{2^{l_1}}\right)\,, \\ \frac{2 - 2^{l_2 - l_1} i_1}{2^{l_1}}\,, & l_2 < l_1,\, \left(i_2 = 1,\, 1 < i_1 < \frac{2^{l_1}}{2^{l_2}}\right)\,, \\ \frac{2 - 2^{l_1} + 2^{l_1 - l_2} i_2}{2^{l_2}}\,, & l_1 < l_2,\, i_1 = 2^{l_1} - 1,\, \frac{2^{l_1 + l_2} - 2^{l_2}}{2^{l_1}} < i_2 < 2^{l_2} - 1\,, \\ \frac{2 - 2^{l_2} + 2^{l_2 - l_1} i_1}{2^{l_1}}\,, & l_2 < l_1,\, i_2 = 2^{l_2} - 1,\, \frac{2^{l_2 + l_1} - 2^{l_1}}{2^{l_2}} < i_1 < 2^{l_1} - 1\,, \\ r_{q_1,q_2}^{\text{linear}}\,, & \text{else}\,. \end{cases} \tag{2.40}$$

For the proof, see Sec. A.1.2.

2.2.3. Kinked Linear Basis

With the modified linear basis, we are able to approximate boundary function values unequal to zero. However, the extrapolated values near the boundary might grow high, because the basis function value of 1 at the position of the grid point is extended to 2 at the boundary. Especially at points that are close to multiple boundaries, this value might explode. Thus, we propose a new basis function, the **kinked linear basis**.

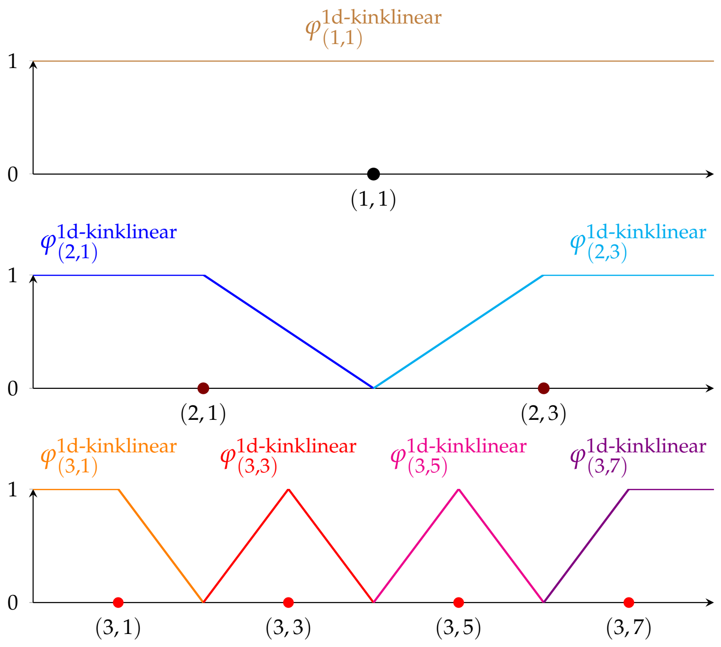


Figure 2.9.: One-dimensional kinked linear basis functions up to level 3. At the boundaries, it extrapolates to 1, which makes it different from both the linear basis and the modified linear basis.

Definition 2.2.5 (One-dimensional kinked linear basis function)
Let $p = (l, i)$ be a one-dimensional grid point. The one-dimensional kinked linear basis function for p is then given by

$$\varphi_p^{\text{1d-kinklinear}} \colon \Omega^1 \to [0,1]\,,$$
$$x \mapsto \begin{cases} 1\,, & (l,i) = (1,1)\,, \\ 1\,, & \left(i = 1 \text{ and } x < 2^{-l}\right) \text{ or} \\ & \left(i = 2^l - 1 \text{ and } x > \left(2^l - 1\right) \cdot 2^{-l}\right)\,, \\ \varphi_p^{\text{1d-linear}}(x)\,, & \text{else.} \end{cases} \tag{2.41}$$

This definition equals to

$$\varphi_p^{\text{1d-kinklinear}}(x) = \min\left\{1, \varphi_p^{\text{1d-modlinear}}(x)\right\}\,, \tag{2.42}$$

thus capping the value of the modified linear basis function at 1.

The kinked linear basis for all grid points in $\texttt{componentGrid}_3^1$ is depicted in Fig. 2.9. In contrast to Fig. 2.7, it can be seen that the value of the kinked linear basis is at most 1, whereas it grows up to 2 for the modified linear basis.

The generalization to more dimensions is (as in the linear and modified linear case) given via the tensor product approach:

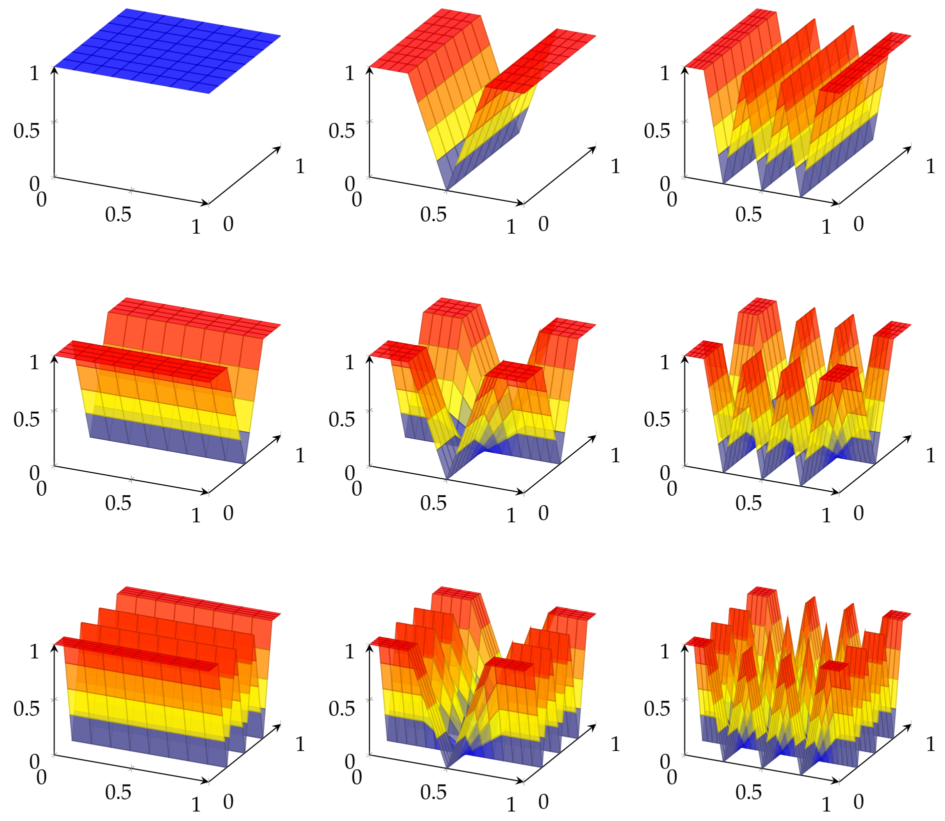


Figure 2.10.: Two-dimensional kinked linear basis functions up to level 3. Although is doesn't extrapolate to zero at the boundaries, the value in the corners doesn't grow exponentially with the dimensionality (in contrast to the modified linear basis).

Definition 2.2.6 (Kinked linear basis function)
Let $p \in \mathbb{G}_d$ be a d-dimensional grid point. The kinked linear basis function for p is then given by

$$\begin{aligned} \varphi_p^{\text{kinklinear}} \colon \Omega &\to [0,1]\,, \\ \boldsymbol{x} &\mapsto \prod_{j=1}^{d} \varphi_{\mathtt{proj}_{p,j}}^{\text{1d-kinklinear}}(x_j)\,. \end{aligned} \tag{2.43}$$

The kinked linear basis for all grid points in $\texttt{componentGrid}_{2,(3,3)}$ is depicted in Fig. 2.10. Again, in contrast to Fig. 2.8 where the value for the two-dimensional modified linear basis grows up the 4 in the corners, the kinked linear basis reaches values of at most 1, independent of the number of dimensions.

Inner Product Again, with $p_1, p_2 \in \mathbb{G}_d$, we want to derive a closed form for $\left\langle \varphi_{p_1}^{\text{kinklinear}}, \varphi_{p_2}^{\text{kinklinear}} \right\rangle_{L_2}$. As for the linear and modified linear case, it holds that

$$\left\langle \varphi_{p_1}^{\text{kinklinear}}, \varphi_{p_2}^{\text{kinklinear}} \right\rangle_{L_2} = \prod_{j=1}^{d} \int_0^1 \varphi_{\mathtt{proj}_{p_1,j}}^{\text{1d-kinklinear}}(x) \cdot \varphi_{\mathtt{proj}_{p_2,j}}^{\text{1d-kinklinear}}(x) \mathrm{d}x \,, \tag{2.44}$$

and we are again looking at two one-dimensional points $q_1 = (l_1, i_1)$ and $q_2 = (l_2, i_2)$ and calculate

$$r_{q_1,q_2}^{\text{kinklinear}} := \left\langle \varphi_{q_1}^{\text{1d-kinklinear}}, \varphi_{q_2}^{\text{1d-kinklinear}} \right\rangle_{L_2} = \int_0^1 \varphi_{q_1}^{\text{1d-kinklinear}}(x) \cdot \varphi_{q_2}^{\text{1d-kinklinear}}(x) \mathrm{d}x \,. \tag{2.45}$$

It holds, that,

$$r_{q_1,q_2}^{\text{kinklinear}} = \begin{cases} 1\,, & l_1 = l_2 = 1\,, \\ \frac{2^{2-l}}{3}\,, & q_1 = q_2,\, l_1 > 1,\, (i_1 = 1 \text{ or } i_1 = 2^{l_1} - 1)\,, \\ 3 \cdot 2^{-l_1-1}\,, & l_1 > l_2,\, \left(\left(i_2 = 1,\, i_1 < \dfrac{2^{l_1}}{2^{l_2}} \right) \text{ or } \right. \\ & \left. \left(i_2 = 2^{l_2} - 1,\, i_1 > \dfrac{2^{l_1}\left(2^{l_2} - 1\right)}{2^{l_2}} \right) \right)\,, \\ 3 \cdot 2^{-l_2-1}\,, & l_2 > l_1,\, \left(\left(i_1 = 1,\, i_2 < \dfrac{2^{l_2}}{2^{l_1}} \right) \text{ or } \right. \\ & \left. \left(i_1 = 2^{l_1} - 1,\, i_2 > \dfrac{2^{l_2}\left(2^{l_1} - 1\right)}{2^{l_1}} \right) \right)\,, \\ r_{q_1,q_2}^{\text{linear}}\,, & \text{else}\,. \end{cases} \tag{2.46}$$

For the proof, see Sec. A.1.3.

2.3. Function Spaces and Approximation

Bringing grids and basis functions together, we now discuss how to obtain function approximations.

2.3.1. Grid-based Function Space and Approximant

For every type of basis function ϕ and set of grid points $\mathcal{G}$, we construct the corresponding function space $\mathcal{V}_{\mathcal{G}}^{\phi}$:

Definition 2.3.1 (Function space)
Let $\mathcal{G} \subset \mathbb{G}_d$ be any set of grid points. For $p \in \mathcal{G}$ and a family of basis functions ϕ, ϕ_p denotes the basis function at p of this family. The function space $\mathcal{V}_{\mathcal{G}}^{\phi}$ is given by

$$\mathcal{V}_{\mathcal{G}}^{\phi} := \operatorname{span}\left\{\phi_p \;\middle|\; p \in \mathcal{G}\right\} . \tag{2.47}$$

If, for example, the task is to find an interpolant of f, there exists a unique solution $\widetilde{f} \in \mathcal{V}_{\mathcal{G}}^{\phi}$ that satisfies

$$\forall p \in \mathcal{G} : \widetilde{f}(\texttt{coord}_p) = f(\texttt{coord}_p) \tag{2.48}$$

and the challenge is to find the hierarchical surpluses α_p via the process called *hierarchization* such that

$$\widetilde{f}(\boldsymbol{x}) = \sum_{p \in \mathcal{G}} \alpha_p \phi_p(\boldsymbol{x}) . \tag{2.49}$$

It has been shown [18] that with certain smoothness assumptions for f, the ratio between cost (number of grid points) and error (difference of f and $\widetilde{f}$ under a specific norm) is optimal when choosing as $\mathcal{G}$ a regular sparse grid and that the curse of dimensionality is mitigated with this choice. Since we need those results only marginally, the reader is referred to [18] for details.

For the data-driven problems we face, rather than searching for interpolants, we want to find $\widetilde{f} \in \mathcal{V}_{\mathcal{G}}^{\phi}$ that describes an unknown distribution function (see Eq. 3.4).

Looking back at Eq. 2.1, we now have the tools at hand to construct the approximation of f.

Definition 2.3.2 (Grid-based approximant)
Let $\mathcal{G} \subset \mathbb{G}_d$ be any set of grid points and ϕ a family of basis functions well defined for $\mathbb{G}_d$. A $\widetilde{f} \in \mathcal{V}_{\mathcal{G}}^{\phi}$ is an approximation of f with ϕ on $\mathcal{G}$ if $\widetilde{f} \approx f$ and we write $\widetilde{f}$ as

$$\widetilde{f}(\boldsymbol{x}) = \sum_{p \in \mathcal{G}} \alpha_p \cdot \phi_p(\boldsymbol{x}) . \tag{2.50}$$

For specific grids, the function spaces and the approximations residing in them have specific names:

Definition 2.3.3 (Full grid and sparse grid approximation)
Let φ be either φ^{linear}, $\varphi^{\text{modlinear}}$ or $\varphi^{\text{kinklinear}}$.

- An approximation $f^{d,\varphi}_{\texttt{fullgrid},l} \in \mathcal{V}^{\varphi}_{\texttt{fullGrid}_{d,l}}$ of f is called a *full grid solution* of f.
- An approximation $f^{d,\varphi}_{\texttt{sparsegrid},l} \in \mathcal{V}^{\varphi}_{\texttt{sparseGrid}_{d,l}}$ of f is called a *sparse grid solution* of f.

In Chap. 3, we discuss in detail how to construct an approximation of a probability density estimation for any dataset residing in Ω. Whereas the choice between a full grid and a sparse grid is of major importance concerning the resulting number of degrees of freedoms, the methods we present there can be tackled with both techniques.

2.3.2. Combination Grid Technique

Solving problems with the sparse grid method instead of the full grid method requires application engineers to dive deep into algorithms and data structures that handle sparse grids appropriately. Full grid solver implementations exist for a long time and are used by lots of applications, which is why they are very mature and include lots of features that are time consuming to rebuild when switching to sparse grids. For example, parallelization on regular sparse grids is not easily done. One way to exploit both the good cost vs. approximation error ratio of sparse grids and the maturity and accessibility of full grid solvers is the combination grid technique [39]. If the setting allows it, the problem is first computed on many component grids separately, which are then combined together to yield the final solution. The idea for the two-dimensional case is depicted in Fig. 2.11. Each involved component grid's solution is weighted with a factor and the final solution is the sum over all weighted component solutions. We define the combination grid technique in the following:

Definition 2.3.4 (Combination grid technique)
Let $f_{\boldsymbol{l}}(\boldsymbol{x}) \in \mathcal{V}^{\varphi}_{\texttt{componentGrid}_{d,\boldsymbol{l}}}$ be an approximation of f on $\texttt{componentGrid}_{d,\boldsymbol{l}}$, where φ is either φ^{linear}, $\varphi^{\text{modlinear}}$ or $\varphi^{\text{kinklinear}}$. The approximation of f given by

$$f(\boldsymbol{x}) \approx f^{d,\varphi}_{\texttt{combi},l}(\boldsymbol{x}) = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{\|\boldsymbol{l}'\|_1 = l+d-1-q} f_{\boldsymbol{l}'}(\boldsymbol{x}) \tag{2.51}$$

is called the combination grid technique solution of level l of f.

For some problems such as interpolation, it has been shown [39] that

$$f^{d,\varphi}_{\texttt{combi},l} = f^{d,\varphi}_{\texttt{sparsegrid},l} \tag{2.52}$$

holds. In two dimensions, the factor $(-1)^q \binom{d-1}{q}$ resolves to weighting the components on the main diagonal of the component grid scheme with $+1$ and weighting the components on the first subdiagonal with -1, as depicted in Fig. 2.11. All other components receive a factor of 0 and are not required to compute the combination grid solution. For d dimensions, the factors on the main diagonal (which is a $d-1$-dimensional hyperplane) are still weighted with $+1$, whereas the factors for the subdiagonals grow more complex. Also, not one subdiagonal but $d-1$ additional subdiagonals (as long as the level is high enough) are required.

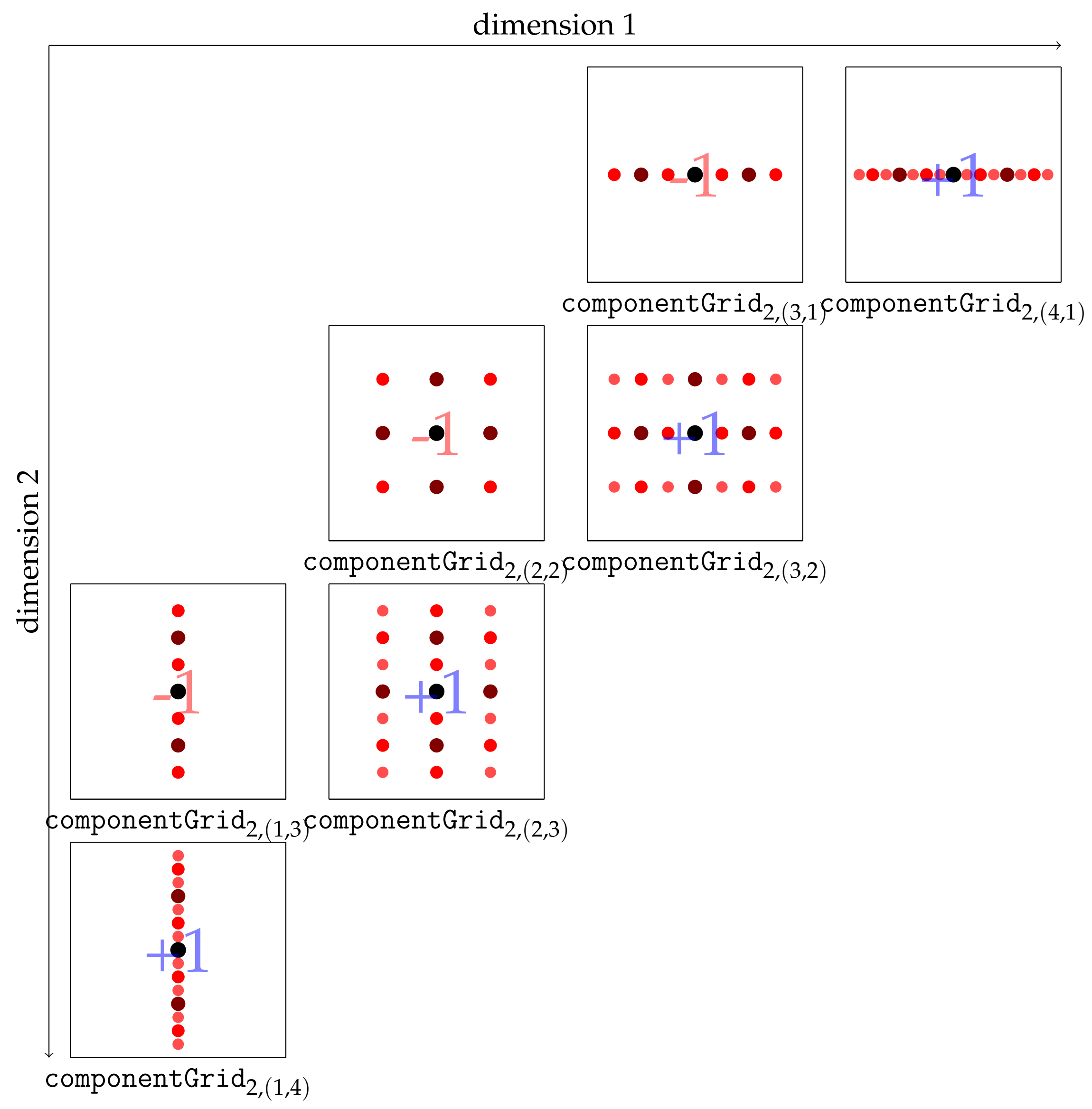

dimension 1
dimension 2
-1
+1
componentGrid$_{2,(3,1)}$
componentGrid$_{2,(4,1)}$
-1
+1
componentGrid$_{2,(2,2)}$
componentGrid$_{2,(3,2)}$
-1
+1
componentGrid$_{2,(1,3)}$
componentGrid$_{2,(2,3)}$
+1
componentGrid$_{2,(1,4)}$

Figure 2.11.: Two-dimensional combination grid technique scheme of level 4. The components in the level 4 diagonal are weighted with a factor of 1 for the combination grid technique solution, the components in the level 3 diagonal are weighted with a factor of -1.

The component's solutions can be obtained in parallel, because they are independent of each other. Although the total number of grid points in the combination grid technique exceeds the number of grid points in the regular sparse grid, this leads to a computational speedup of which we make use of in Sec. 4.3. In fact, we are going to show there that we even obtain a computational speedup with the combination grid technique if we still handle the components in a serial fashion.

The combination grid technique allows for dimensional adaptivity. If we want to increase the resolution only in certain dimensions, we add the component grids to the scheme that have a higher level in such a dimension on the condition, that all predecessors of the additional component grid are already present in the scheme. In this case, the computation of the factors is more complex and we employ the algorithm given in [69] to compute them efficiently.

3. Grid-based Density Estimation and Classification

Density estimation and classification are fundamental data mining tasks. For many problem settings, they are well-studied and well-understood already. However, for problems of mediocre to high dimensionality involving large datasets, some traditional methods suffer from certain drawbacks. First, techniques such as kernel density estimation [72] are not applicable very well to problems of high dimensionality due to the curse of dimensionality [93]. Second, evaluating such a density estimator also grows in linear complexity with the number of training samples, which renders the application to big data infeasible [36]. Sparse grid-based methods cope with both of those issues [75], first by mitigating the curse of dimensionality and second by the model complexity being independent of the training dataset size.

In this chapter, we propose an incremental grid-based batch learning scheme for both density estimation (unsupervised learning) and classification (supervised learning) supporting spatial grid adaptivity. With the grid-based approach, we explicitly codify the problem space. The model is explainable because the learned weights at the grid nodes directly represent the dataset and thus the problem at hand. This gives an edge to the grid-based approach which is hard to achieve for many other models, especially those obtained by deep learning [67, 85].

Fortunately, grid-based density estimation has already been the focus of the research in [76], where an offline/online scheme has been introduced. This scheme shifts the heavy computational effort of obtaining the probability density estimation to an offline phase. Decomposing and storing the system matrix of the problem-corresponding system of linear equations allows to obtain the density estimation as the solution of this linear system fast. We adapt this offline/online scheme so that we are also able to modify the underlying grid to the problem at hand. Also, we show how an incremental learning scheme is obtained by splitting the dataset into batches.

Sec. 3.2 focuses on density estimation. We propose the mentioned incremental batch-wise learning scheme. After setting the problem context and preparing the mathematical tools and fundamental techniques, we discuss the challenge of combining the offline/online scheme and spatial adaptivity. After that, we investigate the linear system stemming from the grid-based density estimation in detail with attention to updating this system as a consequence of grid refinement and coarsening. How to reflect those updates in the matrix decompositions used for the offline/online scheme is explored in detail. To achieve the desired speed-up during the online phase, we propose to employ both the Cholesky decomposition and a tridiagonal decomposition [34] which both make good use of the properties of the linear system. In particular, we show how those two matrix decompositions are updated according to the grid changes resulting from spatial adaptivity.

In Sec. 3.3, we turn to the task of classification. With the density estimation toolbox prepared and ready, we discuss how the Bayes classifier builds on top of probability density estimations of the data separated by class. We then show how the incremental batch-wise learning scheme from Sec. 3.2 transfers to classification. With the way cleared for spatial adaptivity in combination with the offline/online scheme, we propose a grid-refinement strategy for classification. This strategy is tailored to the setup of the many-grid classification model resulting from the use of the Bayes classifier. It exploits both the properties of sparse grids and the problem-specific attributes of many-grid classification.

However, before we start with density estimation, let us first focus on data in general and what has to be considered when tackling datasets with grid-based methods.

3.1. Data Mining with Sparse Grids: Prerequisites

Before starting any learning task, we first need to check whether the grid-based methods we have in mind are applicable to the problem. In this section, we characterize the data we want to work with and discuss the scope of problems we are able to tackle with the grid-based methods.

Problem Characteristics In the data context, a point in the problem space represents a data sample. It is attributed by characteristics that we call *dimensions*, which are the individual measurable quantities of the observable. For our grid-based methods to be able to work with the data, we foremost require all dimensions to be real-valued. With d dimensions at hand, this enables us to map the data to the unit hypercube $\Omega = [0,1]^d$, which is where the grid points and corresponding basis functions defined in Sec. 2.1 and Sec. 2.2 reside. This data preprocessing step is necessary for most datasets, because the data we want to get insight on is hardly always already situated in Ω from the start.

If a dimension consists of discrete values that are subject to a linear order (for example the income of a person, which is a natural number), we are also able to map it onto $[0,1]$. However, for categorical dimensions (for example the nationality of a person), we are not able to find a mapping to $[0,1]$ that coincides with the linear order of $[0,1]$. Thus, we cannot tackle such datasets with our grid-based methods (unless we remove said dimensions from the data).

Dimensionality A key characteristic of a dataset is the number of dimensions, which we also call the dimensionality. Although in theory, there are no limitations to it when tackling the dataset with the grid-based methods we have in mind, we target a certain range.

On principle, it is possible to apply the grid-based methods we investigate in this thesis to one-dimensional problems. However, in such cases, we cannot profit from the sparse grid structure, which is designed to cope with the curse of dimensionality, which only kicks in at $d > 1$. Considering an upper bound, sparse grids have been applied to problems with several hundreds of dimensions in the past [50]. We are going to look even further than that and apply the methods to problems with dimensionalities in the thousands. To achieve that, we propose some modifications to standard sparse grids, which make such high dimensionalities possible. However, even then, there is a limit to the number of dimensions when using our methods in practice. Where those limits are situated depends on the problem class and becomes clear with the development of this thesis.

Big Data The term *Big Data* is a bit fuzzy, but there are some criteria (the **V**-criteria [61]) that apply to problems attributed with it. Speakers and authors differ on how many V's should be listed. Validity, value, variety, velocity, veracity, visualization, volume, volatility and vulnerability are just some of the most mentioned keywords. In this thesis, the two attributes volume and velocity are of import.

- **Volume:** The dataset contains many samples.

  Sparse grid methods shine, when the number of data points considerably exceeds the number of grid points. While other methods suffer from exploding runtimes in the case of many data samples, our grid-based methods are designed such that the evaluation complexity of the models is always independent of the number of samples learned.

- **Velocity:** New data is generated fast.

  The algorithms we propose can cope with data streams and we employ online learning in parallel settings to be able to keep pace with high velocity data streams.

Notation With $\Omega = [0,1]^d$ we denote the problem space. Thus, for the unsupervised learning problems we look at, a dataset is written as

$$\mathcal{M} = \{\boldsymbol{x}_j \mid j \in [M], \boldsymbol{x}_j \in \Omega\} . \tag{3.1}$$

$M \in \mathbb{N}$ denotes the size of the dataset $\mathcal{M}$ with x_j ($j \in [M]$) being a sample of $\mathcal{M}$. Additionally, for the supervised learning problems, we associate a label $y_j \in \mathcal{K}$ to every sample x_j, yielding

$$\mathcal{M} = \left\{ \left(x_j, y_j\right) \quad \middle| \quad j \in [M], x_j \in \Omega, y_j \in \mathcal{K} \right\} . \tag{3.2}$$

During the data mining process, we distinguish three different datasets for our problem: The *training dataset*, the *validation dataset* and the *test dataset*:

- **Training dataset:** Denoted with $\mathcal{M}_{\texttt{train}}$, this is the data we directly learn on and train the model to.
- **Validation dataset:** Denoted with $\mathcal{M}_{\texttt{val}}$, we don't use this data to directly train the model with but to validate its quality during the training process. Also, we use it to find good values for certain hyperparameters during learning. To evaluate the model quality with previously trained data would not yield honest and useful results, as the model already knows ground truth about it. Technically, we split off some portion of $\mathcal{M}_{\texttt{train}}$ to be $\mathcal{M}_{\texttt{val}}$ before we start training the model, because $\mathcal{M}_{\texttt{val}}$ is usually not separately provided within the data problem.
- **Test dataset:** Denoted with $\mathcal{M}_{\texttt{test}}$, this dataset is used to score the model quality objectively. The test dataset and the scores obtained by scoring the model with it cannot be used (or even known) by the routines training the model as to not raise doubts about the achieved accuracies.

3.2. Density Estimation

Density Estimation is an important tool in data analysis and statistics [100]. Many methods exist to estimate the density of a given dataset [103]. It is used as a building block for other data mining methods such as classification [76] and clustering [77] and is employed to gain insight into data by sampling and visualization [90]. Given $\Omega = [0,1]^d$ and a countable d-dimensional dataset $\mathcal{M} \subset \Omega$ of size $|\mathcal{M}| = M \in \mathbb{N}$, the goal is to find a probability density function $f\colon \Omega \mapsto \mathbb{R}$ that indicates for every $x \in \Omega$ the relative likelihood that a sample of $\mathcal{M}$ has the value of x. Due to its data-driven nature, this translates to a optimization problem of finding f in a given function space. Ideally, f fits to the dataset as well as it is smooth at the same time. An example is shown in Fig. 3.1.

Histograms [73] have been used to agglomerate data points of similar value in bins. Their construction is intriguingly simple but they miss some properties such as being continuous and depending on an initial choice of bins. Nowadays, there exist a wide variety of binning methods for density estimation [87] that overcome the first drawback of not being continuous, whereas they either still require some predefined bins or some heavy computations in order to determine a good choice without expert knowledge of the dataset. If such a knowledge is already available, parametric methods are a good choice for density estimation and yield very accurate probability density estimation

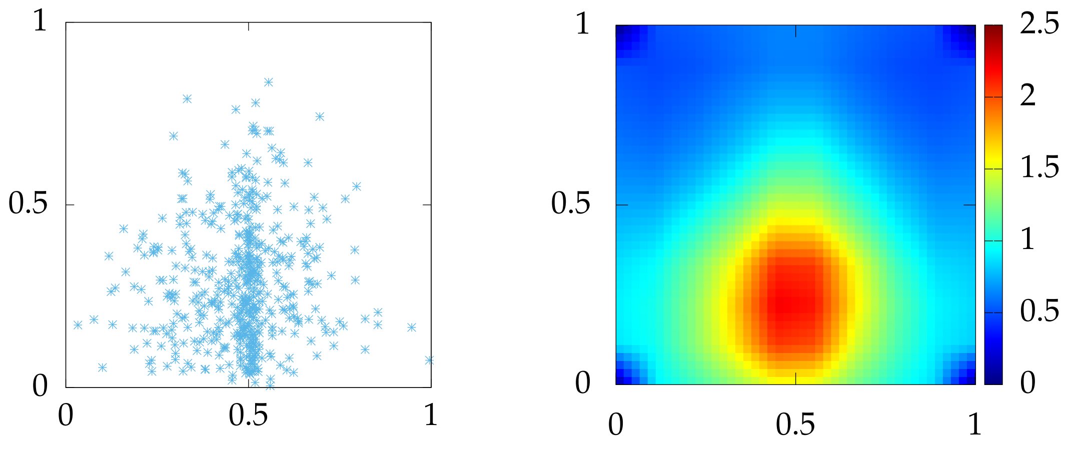


Figure 3.1.: On the right, a density estimation of the dataset on the left is visualized.

functions for lower dimensional problems. However, for real-world data problems, such knowledge rarely exists prior to analyzing the data. Also, with increasing d, the number of bins grows exponentially (curse of dimensionality), which makes this class of methods only feasible for low dimensional problems. Thus, we look at non-parametric methods [24], a prominent example being kernel density estimation (KDE), which builds f by summing up kernel functions centered at the data points. A major drawback of KDE is the linear complexity of $\mathcal{O}(M)$ for evaluating f at a given point $\boldsymbol{x} \in \Omega$. As the role of big data problems cannot be overrated in the current data science world, such methods become infeasible very quickly for large datasets.

The method we want now to focus on is grid-based density estimation. Centering the basis functions at specific grid points to build up a density estimation offers an explainable data model. Apart from quickly evaluating every possible point in the problem space, we can also explain the structure of the density estimation by looking at the coefficients of the chosen basis. With $\delta_{\boldsymbol{x}}$ being the Dirac delta function centered at $\boldsymbol{x}$, we define a highly overfitted guess of f as $f_\epsilon = \frac{1}{M}\sum_{\boldsymbol{x}\in\mathcal{M}} \delta_{\boldsymbol{x}}$. In [43] a multivariate spline-smoothing density method has been introduced, which aims to find f in a function space $\mathcal{V}$ as

$$f = \underset{\tilde{f}\in\mathcal{V}}{\arg\min} \int\limits_{\Omega} \left(\tilde{f}(\boldsymbol{x}) - f_\epsilon(\boldsymbol{x})\right) \mathrm{d}\boldsymbol{x} + \lambda \left\|\Lambda f\right\|_{L_2}^2 . \tag{3.3}$$

The left summand ensures that f fits to the available data points whereas the right summands serves as a smoothness constraint for f. Then, the variational equation associated with Eq. 3.3 for all $\psi \in \mathcal{V}$ formulates as

$$\int\limits_{\Omega} f(\boldsymbol{x})\psi(\boldsymbol{x})\mathrm{d}\boldsymbol{x} + \lambda \int\limits_{\Omega} \Lambda f(\boldsymbol{x})\ \Lambda\psi(\boldsymbol{x})\ \mathrm{d}\boldsymbol{x} = \frac{1}{M}\sum_{\boldsymbol{x}\in\mathcal{M}} \psi(\boldsymbol{x}) . \tag{3.4}$$

While the authors in [43] use this method with $\mathcal{V}$ being the full grid function space, it has been applied with a sparse grid function space as well in [78]. Since we want to both employ sparse grids and the combination technique, we only restrict the choice of $\mathcal{V}$ to a hierarchical function space as defined in Eq. 2.47 where we write the probability density function f as

$$f(\boldsymbol{x}) = \sum_{k \in \mathcal{V}} \alpha_k \varphi_k(\boldsymbol{x}) \tag{3.5}$$

with the Ritz-Galerkin method [26]. Since the ansatz function space is the same as the test function space, with $N := |\mathcal{V}|$ being the number of grid points and $\boldsymbol{R}, \boldsymbol{C} \in \mathbb{R}^{N \times N}$, $\boldsymbol{\alpha}, \boldsymbol{b} \in \mathbb{R}^N$, Eq. 3.4 translates to

$$(\boldsymbol{R} + \lambda \boldsymbol{C})\boldsymbol{\alpha} = \boldsymbol{b} \tag{3.6}$$

where

$$R_{ij} := \langle \varphi_i, \varphi_j \rangle_{L_2} := \int_{\Omega} \varphi_i(\boldsymbol{x}) \varphi_j(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \tag{3.7}$$

denotes the overlap of two basis functions in the grid. The root point has an overlap with every other point, whereas finer grid points only overlap with few other points. With growing level of the sparse grid, $\boldsymbol{R}$ gets more sparse. However, $\boldsymbol{R}$ is less sparse with growing dimension because for high dimensionality, even points on finer levels still overlap with many other points.

With the term $\lambda \boldsymbol{C}$ we control the regularization in order to limit the growth of the hierarchical surpluses:

$$C_{ij} = \langle \Lambda \varphi_i, \Lambda \varphi_j \rangle_{L_2} \,. \tag{3.8}$$

Plugging Eq. 3.5 back into the right summand of Eq. 3.3 as

$$\lambda \left\| \Lambda \sum_{k \in \mathcal{V}} \alpha_k \varphi_k(\boldsymbol{x}) \right\|_{L_2}^2 , \tag{3.9}$$

we see that, for many problems, it simplifies to

$$\lambda \sum_{k \in \mathcal{V}} \alpha_k^2 , \tag{3.10}$$

yielding a variant of Eq. 3.6, where $\boldsymbol{C} = \boldsymbol{I}$:

$$(\boldsymbol{R} + \lambda \boldsymbol{I})\boldsymbol{\alpha} = \boldsymbol{b} \,. \tag{3.11}$$

Employing Eq. 3.11 instead of Eq. 3.6 drastically reduces the effort to compute f and is used in the following to obtain the probability density function. With the hyperparameter λ, we control how much we want our grid to fit to the data points versus how smooth the sparse grid function is.

On the right hand side of Eq. 3.11, we add the contributions of all data points for each basis function:

$$b_i = \frac{1}{|\mathcal{M}|} \sum_{\boldsymbol{x} \in \mathcal{M}} \varphi_i(\boldsymbol{x}) \,. \tag{3.12}$$

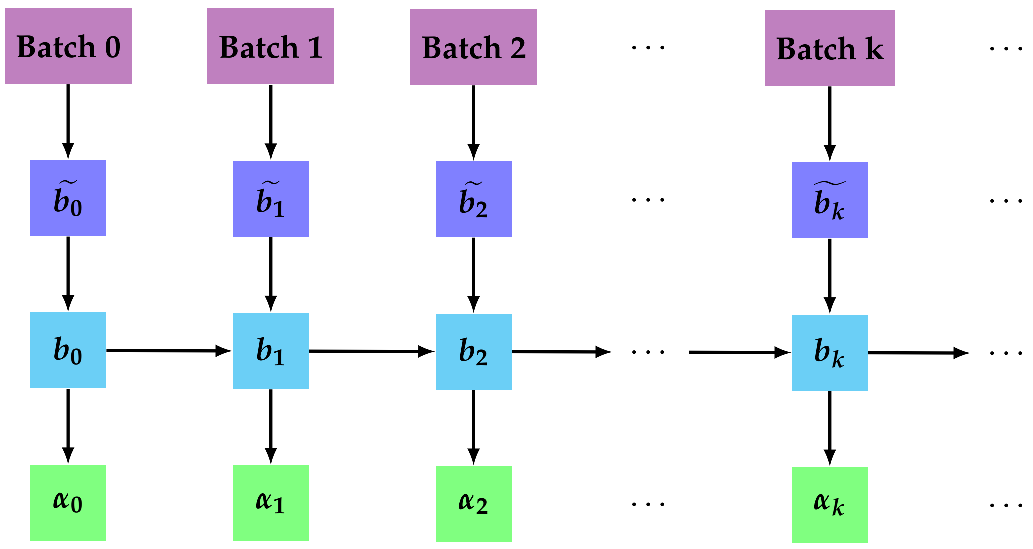


Figure 3.2.: Incremental batch-learning process. From each batch k, the corresponding right-hand side $\widetilde{\boldsymbol{b}_k}$ is obtained, which, together with $\boldsymbol{b}_{k-1}$, yields $\boldsymbol{b}_k$. The solution $\boldsymbol{\alpha}_k$ of Eq. 3.11 for all training data up to batch k is then computed from $\boldsymbol{b}_k$.

The system matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ is also denoted as

$$\boldsymbol{A} := \boldsymbol{R} + \lambda \boldsymbol{I}. \tag{3.13}$$

3.2.1. Incremental Batch Learning Process

In case of a large dataset, the learning process is transformed to a batch learning, mini-batch learning or even a online learning process. Incremental batch-learning has been proposed for various other density estimation methods [59, 98]. In the sparse grid context, online learning has been used for regression [53]. After the learning of each part of the dataset, the obtained model increment is combined with the hitherto existing state of the model. This process is sketched in Fig. 3.2. It allows us to extend the learning process with the following features:

- **Grid adaptivity.** Between or during learning the model increment for a dataset batch, we analyze the grid to see if we want to adapt it to the problem. An adaptation is either coarsening the grid by removing points from it or refining the grid by adding additional points. We discuss how the model is adapted during learning for those cases in Sec. 3.2.2.
- **Concept drift.** Depending on how the model increment is incorporated into the hitherto existing model, we optionally include information about an (implicit) concept drift, e.g. by weighting the model increment disproportionately high to the hitherto existing model.
- **Limiting the memory consumption.** The computation of Eq. 3.12 involves the evaluation of all basis functions of the grid at all data points. Splitting this

evaluation into multiple batches allows to only hold a portion of the dataset in the memory, thus limiting the required resources proportional to the batch size.

- **Parallelization over the data.** The solution of Eq. 3.11 is optionally run in parallel for different data batches. After solving Eq. 3.11 for a batch, its partial solution is incorporated into the central, overall solution. For details, we refer to Chap. 4.

As a side effect, our incremental approach helps the explainability of the model. Not only is the end result transparent, the continuous improvement of the model towards its final stage is also traceable via the evolution of the surpluses.

With m_k, we denote the data points of batch k and denote $\mathcal{M}_k$ as the data points processed up to batch k:

$$\mathcal{M}_k = \bigcup_{j\in[k]} m_j \,. \tag{3.14}$$

$\widetilde{\boldsymbol{b_k}}$ denotes the right-hand side of Eq. 3.11 for the batch k:

$$\widetilde{b_{k}}_i = \frac{1}{|m_k|} \sum_{\boldsymbol{x}\in m_k} \varphi_i(\boldsymbol{x}) \,. \tag{3.15}$$

Of course, we are interested in the quantities $\boldsymbol{b_k}$ and $\boldsymbol{\alpha_k}$, which denote the right-hand-side and solution of Eq. 3.11 of all data points up to batch k:

$$b_{ki} = \frac{1}{|\mathcal{M}_k|} \sum_{\boldsymbol{x}\in \mathcal{M}_k} \varphi_i(\boldsymbol{x}) \,, \tag{3.16}$$

$$\boldsymbol{A\alpha_k} = \boldsymbol{b_k} \,. \tag{3.17}$$

In order to implement the incremental batch learning scheme, we show that for $k > 1$

$$\boldsymbol{b_k} = \frac{|\mathcal{M}_{k-1}|}{|\mathcal{M}_k|}\boldsymbol{b_{k-1}} + \frac{|m_k|}{|\mathcal{M}_k|}\widetilde{\boldsymbol{b_k}} \tag{3.18}$$

holds, per

$$\begin{aligned}
b_{ki} &= \frac{1}{|\mathcal{M}_k|} \sum_{\boldsymbol{x}\in \mathcal{M}_k} \varphi_i(\boldsymbol{x}) \\
&= \frac{1}{|\mathcal{M}_k|} \left(\sum_{\boldsymbol{x}\in \mathcal{M}_{k-1}} \varphi_i(\boldsymbol{x}) + \sum_{\boldsymbol{x}\in m_k} \varphi_i(\boldsymbol{x}) \right) \\
&= \frac{1}{|\mathcal{M}_k|} \cdot \frac{|\mathcal{M}_{k-1}|}{|\mathcal{M}_{k-1}|} \sum_{\boldsymbol{x}\in \mathcal{M}_{k-1}} \varphi_i(\boldsymbol{x}) + \frac{1}{|\mathcal{M}_k|} \cdot \frac{|m_k|}{|m_k|} \sum_{\boldsymbol{x}\in m_k} \varphi_i(\boldsymbol{x}) \\
&= \frac{|\mathcal{M}_{k-1}|}{|\mathcal{M}_k|} \cdot \frac{1}{|\mathcal{M}_{k-1}|} \sum_{\boldsymbol{x}\in \mathcal{M}_{k-1}} \varphi_i(\boldsymbol{x}) + \frac{|m_k|}{|\mathcal{M}_k|} \cdot \frac{1}{|m_k|} \sum_{\boldsymbol{x}\in m_k} \varphi_i(\boldsymbol{x}) \\
&= \frac{|\mathcal{M}_{k-1}|}{|\mathcal{M}_k|} b_{k-1_i} + \frac{|m_k|}{|\mathcal{M}_k|} \widetilde{b_{k}}_i \,.
\end{aligned}$$

Thus, we construct the correct right-hand side incrementally by processing the data batches without having to store all hitherto seen training samples.

Concept Drift The standard batch learning scheme is performed under the assumption that the ordering of the data batches is not important for the solution. However, in some datasets, the ordering plays a role, e.g. when the data is ordered by time and the concept of the data changes with time. One example of such a dataset is the daily average temperature on earth for the last century. More recent data points predict today's temperature more accurately due to the increasing average temperature, which is reflected in the newer data points. Thus, it is desireable to weight new data points higher than older ones instead of treating them all equally. To this end, we define $\beta \in [0,1]$ as the minimal learning rate and alter Eq. 3.18 as

$$\boldsymbol{b_k} = \min\left((1-\beta)^{|m_k|}, \frac{|\mathcal{M}_{k-1}|}{|\mathcal{M}_k|}\right)\boldsymbol{b_{k-1}} + \max\left(1-(1-\beta)^{|m_k|}, \frac{|m_k|}{|\mathcal{M}_k|}\right)\widetilde{\boldsymbol{b_k}}. \tag{3.19}$$

For a value of $\beta = 0$, Eq. 3.19 corresponds to Eq. 3.18. If we choose $\beta > 0$, every data point at least weights with a portion of β into the solution whereas the old points only weight $1-\beta$. In order to make the weighting invariant to the size of the currently processed batch, the current batch is weighted with $1-(1-\beta)^{|m_k|}$ and the old batch with $(1-\beta)^{|m_k|}$.

Online Learning The batch learning process presented allows for every batch to contain an arbitrary number of data points. In online learning, the data points are processed one by one. For our methods, this is not feasible as it introduces too much overhead. Nevertheless, we want to mention, that it is very well possible to employ a pure online learning strategy with the presented methods.

3.2.2. Model Adaption during Learning

During the learning process, we adapt the initial model to the dataset at hand. When performing grid refinement, more points are added to the grid and the matrix $\boldsymbol{R}$ (and respectively $\boldsymbol{A}$) is extended by additional rows and columns. For grid coarsening, rows and columns of $\boldsymbol{R}$ (and respectively $\boldsymbol{A}$) are deleted. In addition, during the early stages of learning we might want to optimize the regularization parameter λ.

3.2.2.1. Grid Refinement

In order to adapt the model best to our data, we add points to the grid depending on where f is not yet a satisfactory estimate for the density estimation. First, we need to identify grid points that are good candidates to be refined by employing various refinement indicators. Then, as soon as we know which grid points are to be added to the grid, we need to adapt the data structures representing our model to reflect the additions.

Refinement Indicators We are interested in finding regions in Ω where f is not yet a good fit to $\mathcal{M}$ in order to employ spatial adaptivity in those areas. The hierarchical parent-child structure of the grid we defined in Sec. 2.1.2 allows us to achieve this spatial adaptivity [79]. Thereby, we identify current leaf grid points where the approximation is not yet satisfying and then refine the model by adding their children to the grid.

A simple indicator we use to identify those refinement candidates is the value of the hierarchical surplus α at the grid point. A high absolute value of α indicates that the error in this area has not yet converged, so we rank the potential refinement candidates (those grid points that still have missing children) by absolute value of the surplus. We refer to this strategy as **surplus-based refinement indicator** [80] and write it as

$$s_p^{\texttt{REF-surplus}} = \left|\alpha_p\right| . \tag{3.20}$$

However, this indicator is prone to overfitting because newly added child grid points can build up even higher surpluses than their parents if regularization fails in those areas. Those children are then ranked high as refinement candidates during the next refinement step and we keep adding grid points of increasingly finer levels to the model, all residing in a small area of Ω.

One way out is to penalize finer grid points by multiplying the surpluses with their respective basis functions' volume. This strategy, referred to as **surplus-volume-based refinement indicator** [80] is given by

$$s_p^{\texttt{REF-surplus-volume}} = \left|\alpha_p\right| \cdot \texttt{support}_p . \tag{3.21}$$

With this indicator, a coarser grid point is refined prior to a finer grid point if they have the same surplus value and we are no longer in danger of overfitting into small regions of Ω.

Modifying the Linear System When refining the grid of size N_a to a grid of size N_{a+1} ($N_{a+1} = N_a + n_a, n_a > 0$, thus adding n_a points to the grid), the linear system Eq. 3.11 changes. We look at what happens if only one additional grid point is added. For multiple new grid points, the steps are performed n_a times. The index of the new grid point is $N_a + 1$.

$\boldsymbol{R}^{(N_a)} \in \mathbb{R}^{N_a \times N_a}$ is extended to $\boldsymbol{R}^{(N_a+1)} \in \mathbb{R}^{(N_a+1)\times(N_a+1)}$ as

$$\boldsymbol{R}^{(N_a+1)} = \begin{pmatrix} \boldsymbol{R}^{(N_a)} & \boldsymbol{r}^{(N_a+1)} \\ \boldsymbol{r}^{(N_a+1)\top} & \langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} \end{pmatrix} , \tag{3.22}$$

whereas $\boldsymbol{r}^{(N_a+1)} \in \mathbb{R}^{N_a}$ is the vector containing the inner products of the basis function of the new grid point with every hitherto existing grid point:

$$r^{(N_a+1)}{}_i = \langle \varphi_i, \varphi_{N_a+1} \rangle_{L_2} . \tag{3.23}$$

Together with the regularization term ($\boldsymbol{I}^{(k)} \in \mathbb{R}^{k \times k}$ denoting the identity matrix of size k), $\boldsymbol{A}^{(N_a)} \in \mathbb{R}^{N_a \times N_a}$ is extended to $\boldsymbol{A}^{(N_a+1)} \in \mathbb{R}^{(N_a+1)\times(N_a+1)}$ as

$$\begin{aligned} \boldsymbol{A}^{(N_a+1)} &= \boldsymbol{R}^{(N_a+1)} + \lambda \boldsymbol{I}^{(N_a+1)} \\ &= \begin{pmatrix} \boldsymbol{R}^{(N_a)} + \lambda \boldsymbol{I}^{(N_a)} & \boldsymbol{r}^{(N_a+1)} \\ \boldsymbol{r}^{(N_a+1)^\mathsf{T}} & \langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{A}^{(N_a)} & \boldsymbol{r}^{(N_a+1)} \\ \boldsymbol{r}^{(N_a+1)^\mathsf{T}} & \langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda \end{pmatrix} . \end{aligned} \tag{3.24}$$

We assume that the grid refinement takes place between the processing of batch $l-1$ and batch l. The right-hand-side $\boldsymbol{b}_{l-1}$ has to be adapted as well after any grid refinement. First, the size of $\boldsymbol{b}_{l-1}$ is extended by n_a new elements. We have no choice but to initialize them to zero. However, it doesn't suffice to further treat the newly added grid points the same way as the hitherto existing grid points in Eq. 3.19 as the new grid points haven't "seen" as many data points as the hitherto existing grid points. Thus, the weighting has to be done individually for each grid point depending on how many data points have been processed for this particular grid point. For $i > N_a$, we set

$$\begin{aligned} m_{k,i} &:= \varnothing\,, \text{ for } k < l\,, \\ \mathcal{M}_{k,i} &:= \bigcup_{j \le k} m_{j,i}\,, \text{ for } k > 0\,, \\ b_{l-1_i} &:= 0\,. \end{aligned}$$

To correctly compute the right-hand-side for the new grid points, we alter Eq. 3.19 further for $k \geq l$:

$$\begin{aligned} b_{k_i} := &\min\left((1-\beta)^{|m_{k,i}|}, \frac{|\mathcal{M}_{k-1,i}|}{|\mathcal{M}_{k,i}|}\right) b_{k-1_i} \\ &+ \max\left(1-(1-\beta)^{|m_{k,i}|}, \frac{|m_{k,i}|}{|\mathcal{M}_{k,i}|}\right) \widetilde{b_{k_i}}\,. \end{aligned} \tag{3.25}$$

3.2.2.2. Grid Coarsening

Opposite to refinement, we can also remove points from the grid in order to reduce the model size and thus, the cost of computing the probability density estimation. To this end, we first need to identify grid points that are good candidates for removal and must then adapt the data structures according to the changes of grid points, similar to what we have seen for the refinement procedure.

Coarsening Indicators We are interested in finding regions in Ω where we can spare grid points, because f would be a good fit to the data even without them. For each grid point p, the surplus value α_p indicates which error is corrected on the probability density estimation in this specific point. If α_p is low, we know that the estimate is still good in this area even without this point being in the grid. Thus, analogous to the surplus-based refinement indicator, we define the **surplus-based coarsening indicator** [79] as the score that yields us the grid points that currently contribute least to f being a good fit to the data based on the absolute value of their surpluses:

$$s_p^{\texttt{COARS-surplus}} = \left|\alpha_p\right|^{-1} . \tag{3.26}$$

However, this indicator disregards that the contribution of a grid point to f also depends on the volume of its support. Thus, we also take into account this quantity and define the **surplus-volume-based coarsening indicator** [79] as

$$s_p^{\texttt{COARS-surplus-volume}} = \left(\left|\alpha_p\right| \cdot \texttt{support}_p\right)^{-1} . \tag{3.27}$$

With this indicator, the grid points that currently contribute least to f in terms of accuracy are scored highest and are thus removed first from the model.

Modifying the Linear System Upon coarsening a grid of size N_a to a grid of size N_{a+1} ($N_{a+1} = N_a - n_a, n_a > 0$, thus removing n_a points from the grid), the linear system Eq. 3.11 changes as well. Similar to the refinement case, we look at what happens if only one point is removed from the grid. The steps are performed n_a times if multiple grid points are to be removed. The index of the removed grid point is $j \in [N_a]$.

$\boldsymbol{R}^{(N_a)} \in \mathbb{R}^{N_a \times N_a}$ is shortened to $\boldsymbol{R}^{(N_a \setminus j)} \in \mathbb{R}^{(N_a-1)\times(N_a-1)}$ as

$$\boldsymbol{R}^{(N_a\setminus j)} = \begin{pmatrix} \boldsymbol{R}^{(N_a)}_{(1:j-1;1:j-1)} & \boldsymbol{R}^{(N_a)}_{(1:j-1;j+1:N_a)} \\ \boldsymbol{R}^{(N_a)}_{(j+1:N_a;1:j-1)} & \boldsymbol{R}^{(N_a)}_{(j+1:N_a;j+1:N_a)} \end{pmatrix} . \tag{3.28}$$

$\boldsymbol{I}^{(k)}$ being the identity matrix of size k, $\boldsymbol{A}^{(N_a)} \in \mathbb{R}^{N_a \times N_a}$ is shortened to $\boldsymbol{A}^{(N_a\setminus j)} \in \mathbb{R}^{(N_a-1)\times(N_a-1)}$ as

$$\boldsymbol{A}^{(N_a\setminus j)} = \boldsymbol{R}^{(N_a\setminus j)} + \lambda \boldsymbol{I}^{(N_a-1)} . \tag{3.29}$$

Again, we assume, that grid coarsening takes place between the processing of batch $l-1$ and batch l. The right-hand-side $\boldsymbol{b_{k-1}}$ has to be adapted as well after any grid coarsening. To this end, element j is removed from $\boldsymbol{b_{l-1}}$

$$\widehat{\boldsymbol{b}}_{\boldsymbol{l-1}} = \begin{pmatrix} \boldsymbol{b}_{\boldsymbol{l-1}(1:j-1)} \\ \boldsymbol{b}_{\boldsymbol{l-1}(j+1:N_a)} \end{pmatrix} . \tag{3.30}$$

3.2.2.3. Optimization of λ

Depending on the number of grid points and the number of data points or rather the relation of both quantities, the regularization parameter λ is optimized in order to fit the data best to the grid at hand. The larger the size of the training dataset, the better but as a thumb rule, we like to have at least as many data points as grid points in the training phase. In Fig. 3.3 the test error is shown for the density estimation of a two-dimensional artificial dataset generated from the beta distribution [74] using different basis functions, different levels and different values of λ. Whereas for lower levels (1 to 5), the test error is minimal if no regularization is applied, the optimal regularization value λ^{opt} is at $\lambda^{\text{opt}} \approx 10^{-2}$.

Search Strategy N being the number of grid points in the model and M being the number of training data points, a good initial guess is $\lambda_0 = \frac{N}{M}10^{-2}$. From there, we search for an optimal value of λ, whereas

$$\mathcal{I}_{\texttt{search}} = \left[10\lambda_0, 10^{-4}\lambda_0\right] \tag{3.31}$$

is deemed a good search space. We split off 10% of the training dataset as validation dataset $\mathcal{M}_{\texttt{val}}$ and compute the right-hand side of Eq. 3.11 for both $\mathcal{M}_{\texttt{val}}$ and $\mathcal{M}_{\texttt{train}}$ as $\boldsymbol{b}_{\texttt{val}}$ and $\boldsymbol{b}_{\texttt{train}}$. Using $\boldsymbol{b}_{\texttt{train}}$ and a potential value of λ, we then solve Eq. 3.11 for $\boldsymbol{\alpha}_\lambda$ and compute the quality with

$$\|\boldsymbol{R}\boldsymbol{\alpha}_\lambda - \boldsymbol{b}_{\texttt{val}}\|_2 \,, \tag{3.32}$$

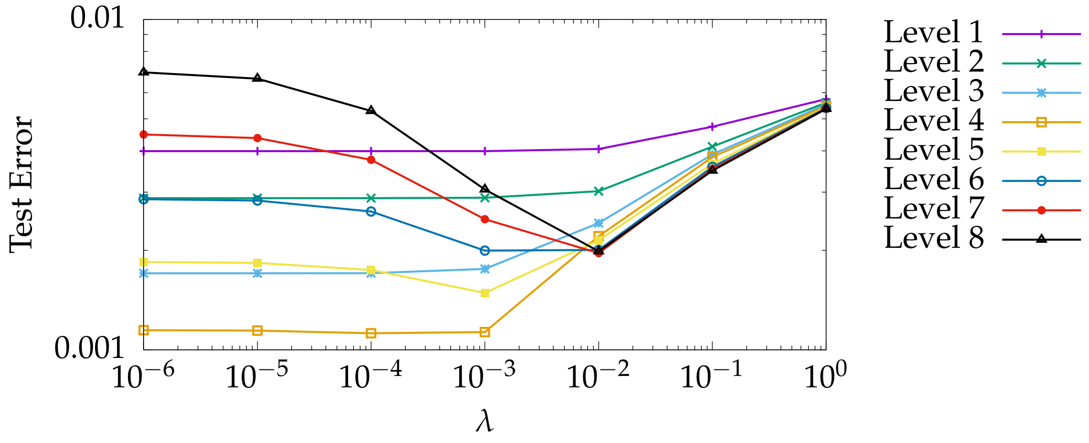
following the proposed method in [75]. We use the Golden-section search [54] to find the minimum in $\mathcal{I}_{\texttt{search}}$ most efficiently.

Modifying the Linear System Changing λ to $\widehat{\lambda}$ affects the linear system Eq. 3.11 as follows:

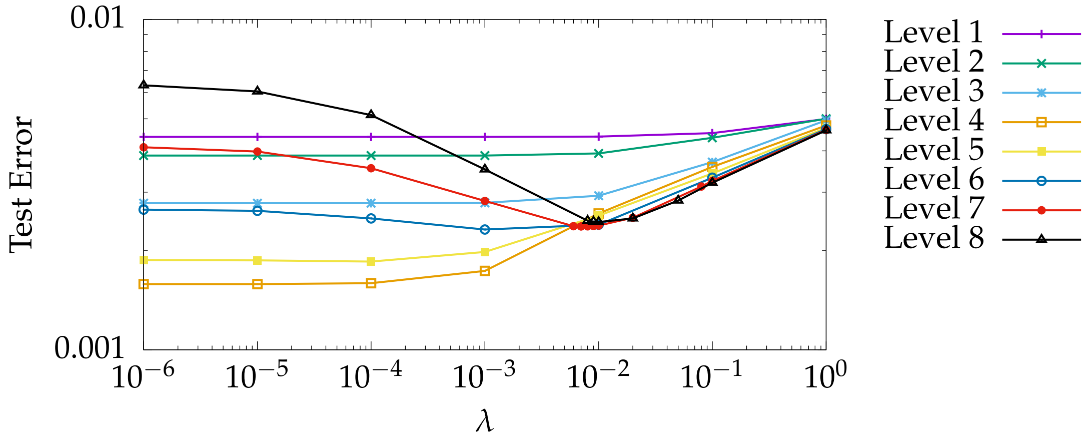
$$\widehat{\boldsymbol{A}} = \boldsymbol{A} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\,. \tag{3.33}$$

3.2.3. Fitting the Model to the Dataset

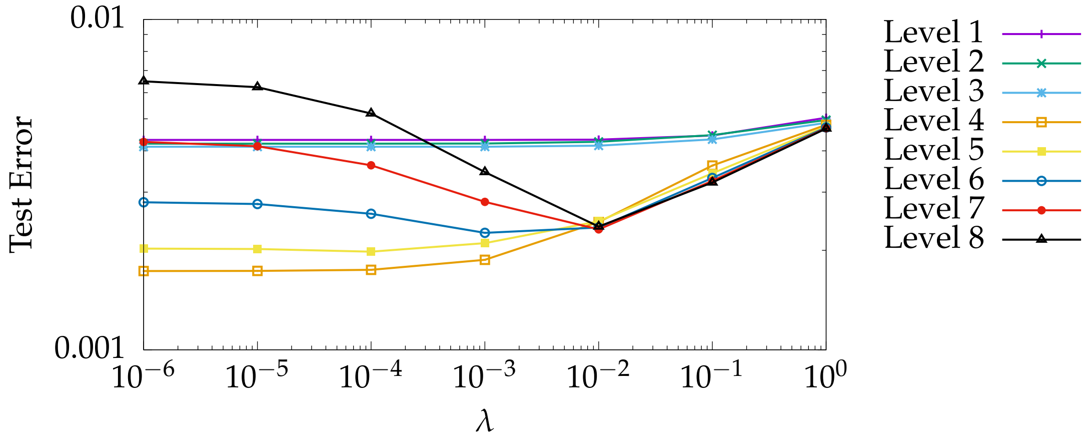
Solving Eq. 3.11 directly takes cubic time in the number of grid points. For larger grids, this is not feasible to compute even on modern hardware, which leaves us two options: Either choose some iterative solver such as conjugate gradients (CG) [47], which approximates the solution of the linear system or split the process into an offline phase and an online phase, which allows us to shift the major computational effort to the offline phase to be faster during the online phase. After noting upon some helpful properties of the linear system, we investigate both options.



(a) Linear basis functions.



(b) Modified linear basis functions.



(c) Kinked linear basis functions.

Figure 3.3.: Optimization of λ at the example of a density estimation of a two-dimensional artificial dataset (400 training data points generated from a strongly skewed beta distribution with $\alpha = \beta = 2$) for different basis functions and levels of the grid. For smaller levels (1 to 5), the error is lower if λ is low, because there are much more grid points than data points and thus, no regularization is required. For higher levels (starting with level 6), the optimal value of λ is $\lambda^{\text{opt}} \approx 10^{-2}$. For $\lambda < \lambda^{\text{opt}}$, the model overfits to the training data. In contrast, the model is over-regularized for when $\lambda > \lambda^{\text{opt}}$.
Raw data for this figure: Sec. C.1.

3.2.3.1. Attributes of the Linear System

We show that the matrices $\boldsymbol{R}$ and $\boldsymbol{A}$ are symmetric positive definite (SPD), which helps us later when factorizing $\boldsymbol{A}$.

Symmetry Per Eq. 3.7, we show that $\boldsymbol{R}$ is symmetric:

$$\begin{aligned}\forall\,(i,j) \in [N]^2 :\; R_{ij} = \left\langle \varphi_i, \varphi_j \right\rangle_{L_2} &= \int_\Omega \varphi_i(\boldsymbol{x})\varphi_j(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \\ &= \int_\Omega \varphi_j(\boldsymbol{x})\varphi_i(\boldsymbol{x})\mathrm{d}\boldsymbol{x} = \left\langle \varphi_j, \varphi_i \right\rangle_{L_2} = R_{ji}\,.\end{aligned} \tag{3.34}$$

Thus, when explicitly computing $\boldsymbol{R}$, we only need to compute the entries for $i \geq j$ and save almost half of the memory and half of the computing time in contrast to not using the symmetry.

Adding $\lambda\boldsymbol{I}$ to $\boldsymbol{R}$ does preserve the symmetry since only the diagonal is affected by this addition. Thus, $\boldsymbol{A}$ is also symmetric.

Positive Definite To show that $\boldsymbol{R}$ and $\boldsymbol{A}$ are positive definite as well, we use the definition of a Gramian matrix [35]:

> **Definition 3.2.1 (Gramian matrix)**
> For a set of vectors $\boldsymbol{v_1}, \dots \boldsymbol{v_n}$ in an inner product space, the matrix $\boldsymbol{G} \in \mathcal{K}^{n\times m}, G_{ij} = \langle \boldsymbol{v_i}, \boldsymbol{v_j} \rangle$ is called the *Gramian matrix*.

Thus, $\boldsymbol{R}$ is the Gramian matrix associated to the L_2-inner-product over the basis functions φ. With [9] we conclude that $\boldsymbol{R}$ is positive definite. $\lambda\boldsymbol{I}$ only has positive diagonal entries, which yields that $\boldsymbol{A}$ as the sum of $\boldsymbol{R}$ and $\lambda\boldsymbol{I}$ is positive definite as well.

Sparsity The sparsity of $\boldsymbol{R}$ is depicted in Fig. 3.4 for dimensions from 1 to 10 for increasing level. We observe a decreasing sparsity with increasing dimension if the level is fixed. This stems from entries of lower value in the level-vectors of the grid points for higher dimensions and thus more overlap with of the supports of the corresponding basis functions with other grid points' supports. On the other hand, the sparsity increases with increasing level, because more grid points have a smaller support which does not overlap with a lot of other grid points. However, these results show, that in general, we cannot assume $\boldsymbol{R}$ (or $\boldsymbol{A}$) being sparse enough to justify the research into sparse matrix methods or investigate other storage schemes. Those would only unfold their potentials for datasets with low dimensionality and high level, which are only a corner case in this thesis.

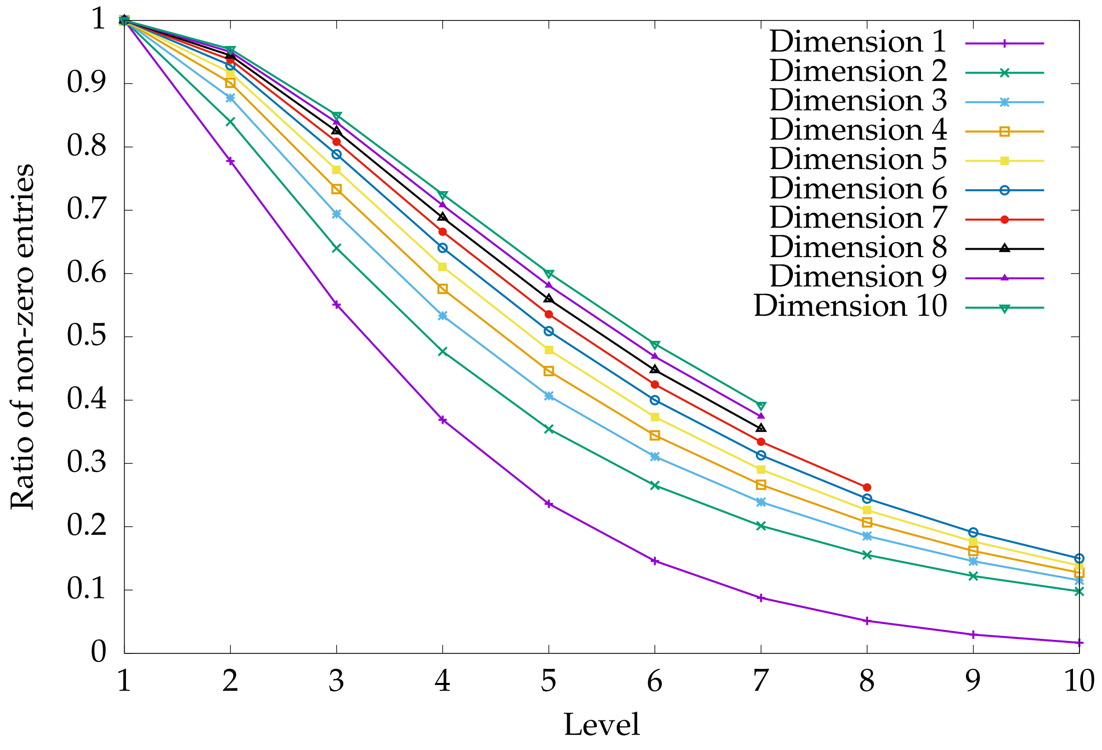


Figure 3.4.: Sparsity of $\boldsymbol{R}$ for dimensions between 1 and 10 for increasing level. With increasing dimension, the sparsity decreases when the level is fixed due to more dimensions, in which the basis functions can overlap. With increasing level, the sparsity increases due to smaller support of the basis functions which leads to less overlap with other basis functions.
Raw data for this figure: Tab. C.4.

3.2.3.2. Conjugate Gradients

The conjugate gradients (CG) method [47] allows for an iterative approximation of the solution of a system of linear equations, if the corresponding system matrix is symmetric and positive-definite. This applies to $\boldsymbol{A}$ as discussed in Sec. 3.2.3.1, so we employ the scheme to solve Eq. 3.11. In case of a batch learning process, the convergence is sped up by the fact that $\boldsymbol{\alpha_k}$ is a good initial guess for $\boldsymbol{\alpha_{k+1}}$. Thus, a good first residuum $\boldsymbol{r} \in \mathbb{R}^N$ is $\boldsymbol{r} = \boldsymbol{b_k} - \boldsymbol{A}\boldsymbol{\alpha_k}$. In each iteration, the algorithm executes a fixed number of matrix-vector products. With k being the number of iterations, the computational cost is $\mathcal{O}\left(k \cdot N^2\right)$. In-between processing the data batches, all model updates such as optimization of λ, grid refinement or grid coarsening are performed by altering $\boldsymbol{A}$ and $\boldsymbol{b}$ directly as discussed in Sec. 3.2.2.

3.2.3.3. Split into Offline and Online Phases

We observe that the right hand side of the linear system Eq. 3.11 depends on the data points whereas the left hand side only contains information about the grid and the regularization. This allows us to precompute the system matrix $\boldsymbol{A}$ of the left hand side and factorize it during an offline phase at the computation cost of $\mathcal{O}(N^3)$ operations. Having $\boldsymbol{A}$ already at hand in a decomposed form, we compute the solution of the linear system in $\mathcal{O}(N^2)$ when the data is actually available. Also, we are able to reuse the matrix factorizations for all problems with the same characteristics that are given by

- the dimensionality of the data d,
- the initial grid level l, and
- the regularization parameter λ.

So we only need to decompose the system matrix for (d, l, λ) once in order to profit from it every time we learn a dataset with those parameters. Similar approaches have been used in online learning settings [5].

During the course of learning the density, we might want to perform grid refinement or coarsening to adapt the grid to the data at hand. This results in a modified $\boldsymbol{R}$ and extended $\boldsymbol{I}$, which then also has to be reflected in the matrix decomposition of $\boldsymbol{A}$. Also, in order to find a good balance between fitting the training data and obtaining a smooth density estimation, we might want optimize λ before we start with grid refinement. The resulting offline/online scheme is depicted in Fig. 3.5. There are several possibilities

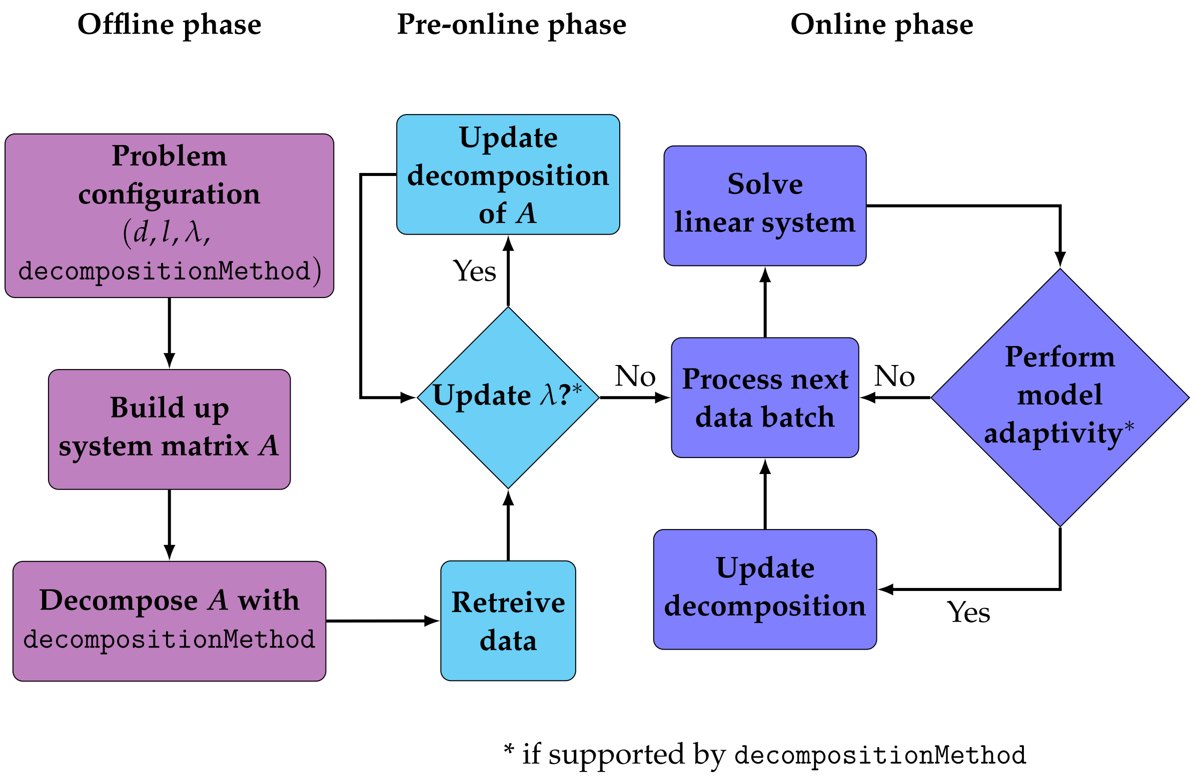

Offline phase
Pre-online phase
Online phase
Problem configuration (d, l, λ, decompositionMethod)
Build up system matrix A
Decompose A with decompositionMethod
Retreive data
Update λ?*
Yes
No
Update decomposition of A
Process next data batch
Solve linear system
Perform model adaptivity*
No
Yes
Update decomposition
* if supported by decompositionMethod

Figure 3.5.: The flowchart of the offline/online scheme. During the offline phase (violet), $\boldsymbol{A}$ is build up and decomposed. At the early stages of training in the pre-online phase (cyan), λ is optimized. During the online phase (blue), the model is trained and adapted to the data.

of how to factorize $\boldsymbol{A}$. In [76], the eigendecomposition (Sec. 3.2.3.3.1) and the LU decomposition were proposed. To better exploit the attributes of $\boldsymbol{A}$, we propose to use the Cholesky decomposition (Sec. 3.2.3.3.2) instead of the LU decomposition. This factorization method also allows us to update the decomposition after the model has been spatially adapted to the data. The same is true for the tridiagonal decomposition (Sec. 3.2.3.3.4) with the addition, that we can also optimize λ prior to updating the decomposition after spatially adapting the model. In problem settings where $\boldsymbol{A}$ is sparse, the incomplete Cholesky decomposition (Sec. 3.2.3.3.3) speeds up both the offline and the online phase even more, at the cost of accuracy. We present those techniques in the following and conclude with their comparison regarding the features and the runtimes for the different tasks when performing training and testing in Sec. 3.2.3.3.5.

3.2.3.3.1 Eigendecomposition

In [76], an eigendecomposition of $\boldsymbol{A}$ was introduced as

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^{\mathsf{T}}\,, \tag{3.35}$$

with $\boldsymbol{V} \in \mathbb{R}^{N\times N}$ an orthonormal matrix and $\boldsymbol{D} \in \mathbb{R}^{N\times N}$ a diagonal matrix. Such a factorization exists because $\boldsymbol{A}$ is a Gramian matrix (see Sec. 3.2.3.1) and the basis functions φ are linearly independent.

Offline Phase The factors $\boldsymbol{V}$ and $\boldsymbol{D}$ are obtained using Golub-Kahan bidiagonalization and QR-reduction [34] in $\mathcal{O}\left(N^3\right)$. Then, the inverse of $\boldsymbol{D}$ is computed in linear time ($\mathcal{O}\left(N\right)$), so that the system is later solved by multiplying $\boldsymbol{b}$ from the right to the inverse of $\boldsymbol{A}$.

Online Phase During the online phase, the system of linear equations Eq. 3.11 is solved for $\boldsymbol{\alpha}$ using the inverse of the eigendecomposition as well as updating the factorization according to changes to the model. In [76], a method to change λ during the online phase was presented. However, they did not present an efficient method to also incorporate changes to the model after grid refinement or coarsening (see Eq. 3.24 and Eq. 3.29). All algorithms during the online phase need to be in $\mathcal{O}\left(N^2\right)$, but recomputing the factors $\boldsymbol{V}$ and $\boldsymbol{D}$ after an adaption of the grid would require $\mathcal{O}\left(N^3\right)$. Thus, the eigendecomposition is not suitable for scenarios where the a priori grid without any spatial adaptivity is not sufficient.

Solving the System With this decomposition, due to $\boldsymbol{V}^{-1} = \boldsymbol{V}^{\mathsf{T}}$, it is straightforward to solve Eq. 3.11 for $\boldsymbol{\alpha}$ by computing three matrix-vector products as

$$\boldsymbol{\alpha} = \boldsymbol{A}^{-1}\boldsymbol{b} = \left(\boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^{\mathsf{T}}\right)^{-1}\boldsymbol{b} = \left(\boldsymbol{V}^{\mathsf{T}^{-1}}\boldsymbol{D}^{-1}\boldsymbol{V}^{-1}\right)\boldsymbol{b} = \left(\boldsymbol{V}\boldsymbol{D}^{-1}\boldsymbol{V}^{\mathsf{T}}\right)\boldsymbol{b} \tag{3.36}$$

which is in $\mathcal{O}\left(N^2\right)$.

Optimizing λ With the data at hand, λ is tuned to find a good balance between fidelity and smoothness. The changes reflected in Eq. 3.33 are also applied to the factors of the eigendecomposition as

$$\begin{aligned}
\widehat{\boldsymbol{A}} &= \boldsymbol{A} + (\widehat{\lambda} - \lambda)\boldsymbol{I} \\
&= \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^\mathsf{T} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\boldsymbol{V}\boldsymbol{V}^\mathsf{T} \\
&= \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^\mathsf{T} + \boldsymbol{V}(\widehat{\lambda} - \lambda)\boldsymbol{I}\boldsymbol{V}^\mathsf{T} \\
&= \boldsymbol{V}\left(\boldsymbol{D} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\right)\boldsymbol{V}^\mathsf{T} ,
\end{aligned} \tag{3.37}$$

so $\boldsymbol{V}$ remains unchanged and all there is left to do, is to compute the inverse of $\widehat{\boldsymbol{D}} = \boldsymbol{D} + (\widehat{\lambda} - \lambda)\boldsymbol{I}$ in order to be able to employ Eq. 3.36 again. As $\widehat{\boldsymbol{D}}$ is obviously a diagonal matrix, computing it's inverse is in $\mathcal{O}\left(N\right)$ and the process of optimizing for λ (assuming that each possible value of λ requires at least one solve of Eq. 3.11) is in $\mathcal{O}\left(N^2\right)$.

Refinement and Coarsening As discussed, in [76] no technique was presented to incorporate changes to the model into the factors of $\boldsymbol{A}$. Thus, in the following, we look at alternative factorizations of $\boldsymbol{A}$ which allow for exactly that.

3.2.3.3.2 Cholesky Decomposition

While in [76], a *LU* decomposition was proposed, we use the symmetry of $\boldsymbol{A}$ to factorize it using the Cholesky decomposition. Not only does it save memory, because instead of two factors for the *LU* decomposition, only one factor for the Cholesky decomposition has to be stored, but also is the decomposition twice as fast. So now, we are looking to factorize the system Matrix $\boldsymbol{A}$ as

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\mathsf{T} , \tag{3.38}$$

whereas $\boldsymbol{L} \in \mathbb{R}^{N \times N}$ is a lower triangular matrix. There exists a unique decomposition of that form because $\boldsymbol{A}$ is symmetric and positive definite (see Sec. 3.2.3.1). Therefore, we can compute the Cholesky decomposition for any grid configuration (d, l, λ). The following methods of factorizing and updating the system matrix using the Cholesky decomposition were implemented into SG++ in a bachelor project [89].

Offline Phase The construction of $\boldsymbol{L}$ is done stepwise by constructing the Cholesky factors $\boldsymbol{L_k} \in \mathbb{R}^{k \times k}$ for $\boldsymbol{A_k} := \boldsymbol{A}_{(1:k;1:k)}$ with $\boldsymbol{A_k} \in \mathbb{R}^{k \times k}$ so that

$$\boldsymbol{A_k} = \boldsymbol{L_k}\boldsymbol{L_k}^\mathsf{T} . \tag{3.39}$$

A recursive approach is chosen, where $\boldsymbol{L}_k$ is computed using the Cholesky factor $\boldsymbol{L}_{k-1}$ of $\boldsymbol{A}_{k-1}$. We begin with

$$\boldsymbol{L}_1 = \left(\sqrt{A_{11}}\right) . \tag{3.40}$$

Obviously, $\boldsymbol{L}_1$ is a unique, invertible, and lower diagonal matrix with only positive diagonal elements and $\boldsymbol{A}_1 = \boldsymbol{L}_1\boldsymbol{L}_1^\top$. For the recursive step, we partition $\boldsymbol{A}_k$ and $\boldsymbol{L}_k$ as

$$\begin{aligned} \boldsymbol{A}_k &= \begin{pmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{a}_k \\ \boldsymbol{a}_k^\top & d_k \end{pmatrix} \qquad \text{and} \\ \boldsymbol{L}_k &= \begin{pmatrix} \boldsymbol{L}_{k-1} & \boldsymbol{0} \\ \boldsymbol{l}_k^\top & \delta_k \end{pmatrix} . \end{aligned}$$

Comparing the entries in

$$\begin{pmatrix} \boldsymbol{A}_{k-1} & \boldsymbol{a}_k \\ \boldsymbol{a}_k^\top & d_k \end{pmatrix} = \boldsymbol{A}_k = \boldsymbol{L}_k\boldsymbol{L}_k^\top = \begin{pmatrix} \boldsymbol{L}_{k-1} & \boldsymbol{0} \\ \boldsymbol{l}_k^\top & \delta_k \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{L}_{k-1}^\top & \boldsymbol{l}_k \\ \boldsymbol{0} & \delta_k \end{pmatrix} = \begin{pmatrix} \boldsymbol{L}_{k-1}\boldsymbol{L}_{k-1}^\top & \boldsymbol{L}_{k-1}\boldsymbol{l}_k \\ \boldsymbol{l}_k^\top\boldsymbol{L}_{k-1}^\top & \boldsymbol{l}_k^\top\boldsymbol{l}_k + \delta_k^2 \end{pmatrix}$$

yields

$$\boldsymbol{l}_k = \boldsymbol{L}_{k-1}^{-1}\boldsymbol{a}_k \quad \text{and} \tag{3.41a}$$

$$\delta_k = \sqrt{d_k - \boldsymbol{l}_k^\top\boldsymbol{l}_k} . \tag{3.41b}$$

Inductively, assuming that $\boldsymbol{L}_{k-1}$ is a unique, invertible, and lower diagonal matrix with only positive diagonal elements, then $\boldsymbol{l}_k$ is unique. In order to show that $\boldsymbol{L}_k$ is a unique, invertible, and lower diagonal matrix with only positive diagonal elements too, all we need to do is show that δ_k from Eq. 3.41b is well defined and positive. We use that $\boldsymbol{A}_k$ is positive definite and take $\boldsymbol{x}_k \in \mathbb{R}^k$ as the solution of $\boldsymbol{L}_{k-1}^\top\boldsymbol{x}_k = -\boldsymbol{l}_k$ to deduce

$$\begin{aligned} 0 &< \begin{pmatrix} \boldsymbol{x}_k \\ 1 \end{pmatrix}^\top \boldsymbol{A}_k \begin{pmatrix} \boldsymbol{x}_k \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{x}_k^\top & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{L}_{k-1}\boldsymbol{L}_{k-1}^\top & \boldsymbol{L}_{k-1}\boldsymbol{l}_k \\ \boldsymbol{l}_k^\top\boldsymbol{L}_{k-1}^\top & d_k \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_k \\ 1 \end{pmatrix} \\ &= \boldsymbol{x}_k^\top\boldsymbol{L}_{k-1}\boldsymbol{L}_{k-1}^\top\boldsymbol{x}_k + \boldsymbol{x}_k^\top\boldsymbol{L}_{k-1}\boldsymbol{l}_k + \boldsymbol{l}_k^\top\boldsymbol{L}_{k-1}^\top\boldsymbol{x}_k + d_k \\ &= d_k - \boldsymbol{l}_k^\top\boldsymbol{l}_k . \end{aligned} \tag{3.42}$$

In the end, we set

$$\boldsymbol{L} := \boldsymbol{L}_N . \tag{3.43}$$

Online Phase The tasks during the online phase are to solve the system of linear equations Eq. 3.11 for $\boldsymbol{\alpha}$ using the computed Cholesky factor and updating the Cholesky factor according to the changes to the model. It is not feasible to compute the decomposition from scratch every time the grid is adapted. Therefore, we need a way to incorporate the changes directly into the decomposed form of $\boldsymbol{A}$. While it is easy to reflect those changes in $\boldsymbol{A}$, it is not trivial to keep track of them in the decomposed form $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\top$. All algorithms during the online phase are in $\mathcal{O}\left(N^2\right)$.

Solving the System In order to solve Eq. 3.11 for $\boldsymbol{\alpha}$, we calculate:

$$\boldsymbol{\alpha} = \boldsymbol{A}^{-1}\boldsymbol{b} = (\boldsymbol{L}\boldsymbol{L}^{\top})^{-1}\boldsymbol{b} = \boldsymbol{L}^{\top^{-1}}\boldsymbol{L}^{-1}\boldsymbol{b} \tag{3.44}$$

in two steps:

$$\begin{aligned} \boldsymbol{\alpha}_{\texttt{fw}} &= \boldsymbol{L}^{-1}\boldsymbol{b} && \text{forward substitution,} && (3.45a)\\ \boldsymbol{\alpha} &= \boldsymbol{L}^{\top^{-1}}\boldsymbol{\alpha}_{\texttt{fw}} && \text{backward substitution.} && (3.45b) \end{aligned}$$

In the serial case, both algorithms are in $\mathcal{O}\left(N^2\right)$ which already speeds up the online phase compared to solving Eq. 3.11 directly. With enough computational power at hand, they are easily parallelized to achieve a linear runtime complexity ($\mathcal{O}\left(N\right)$).

Optimizing λ With data points at hand, we are looking to find a good value for λ. The changes reflected in Eq. 3.33 should also be applied to $\boldsymbol{L}$:

$$\widehat{\boldsymbol{A}} = \boldsymbol{A} + (\widehat{\lambda} - \lambda)\boldsymbol{I} = \boldsymbol{L}\boldsymbol{L}^{\top} + \sum_{j=1}^{N} \boldsymbol{v_i}\boldsymbol{v_i}^{\top}. \tag{3.46}$$

The summands $\boldsymbol{v_i}\boldsymbol{v_i}^{\top}$ are the corresponding rank-1 matrices that are added to $\boldsymbol{A}$. To reflect those additions in $\boldsymbol{L}$, N rank-1 updates (for $\widehat{\lambda} > \lambda$) or rank-1 downdates (for $\widehat{\lambda} < \lambda$) have to be performed, of which each is in $\mathcal{O}\left(N^2\right)$. Thus, changing $\boldsymbol{L}$ after a change of the regularization parameter is in $\mathcal{O}\left(N^3\right)$, which is not feasible during the online phase.

Thus, we need to precompute the Cholesky factorization for multiple values of λ during the offline phase in order to be able to optimize for λ during the online phase. In [19], it has already been observed that the factors of the LU decomposition are non-trivial to derive after a small perturbation such as the optimization of λ presents. We conclude that this also poses a disadvantage for the Cholesky decomposition compared to the Eigendecomposition and the later discussed tridiagonal decomposition.

Refinement Adding a point to the grid of size N_a extends $\boldsymbol{A}^{(\boldsymbol{N_a})}$ by one row and one column, as can be seen in Eq. 3.24. We show how to construct the Cholesky factor $\boldsymbol{L}^{(\boldsymbol{N_a+1})} \in \mathbb{R}^{(N_a+1)\times(N_a+1)}$ of $\boldsymbol{A}^{(\boldsymbol{N_a+1})}$ directly from the Cholesky factor $\boldsymbol{L}^{(\boldsymbol{N_a})} \in \mathbb{R}^{N_a\times N_a}$ of $\boldsymbol{A}^{(\boldsymbol{N_a})}$ [92] as:

$$\boldsymbol{L}^{(\boldsymbol{N_a+1})} = \begin{pmatrix} \boldsymbol{L}^{(\boldsymbol{N_a})} & \boldsymbol{0} \\ \boldsymbol{v}^{\top} & s \end{pmatrix}, \tag{3.47}$$

with $\boldsymbol{v} \in \mathbb{R}_a^N$ and $s \in \mathbb{R}$. This allows us to not compute the Cholesky decomposition from scratch which is in $\mathcal{O}\left(N_a^3\right)$ but to perform the model update in $\mathcal{O}\left(N_a^2\right)$ for each new grid point instead. With Eq. 3.24, It holds that

$$\begin{aligned}\boldsymbol{A}^{(N_a+1)} &= \begin{pmatrix} \boldsymbol{A}^{(N_a)} & \boldsymbol{r}^{(N_a+1)} \\ \boldsymbol{r}^{(N_a+1)^\top} & \langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{L}^{(N_a)} & \boldsymbol{0} \\ \boldsymbol{v}^\top & s \end{pmatrix} \begin{pmatrix} \boldsymbol{L}^{(N_a)^\top} & \boldsymbol{v} \\ \boldsymbol{0} & s \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{L}^{(N_a)} \boldsymbol{L}^{(N_a)^\top} & \boldsymbol{L}^{(N_a)} \boldsymbol{v} \\ \boldsymbol{v}^\top \boldsymbol{L}^{(N_a)^\top} & \boldsymbol{v}^\top \boldsymbol{v} + s^2 \end{pmatrix} . \end{aligned} \tag{3.48}$$

From that, we derive:

$$\boldsymbol{v} = \boldsymbol{L}^{(N_a)^{-1}} \boldsymbol{r}^{(N_a+1)} , \tag{3.49a}$$

$$\begin{aligned} s &= \sqrt{\langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda - \boldsymbol{v}^\top \boldsymbol{v}} \\ &= \sqrt{\langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda - \boldsymbol{r}^{(N_a+1)^\top} \boldsymbol{L}^{(N_a)^{-1^\top}} \boldsymbol{L}^{(N_a)^{-1}} \boldsymbol{r}^{(N_a+1)}} . \end{aligned} \tag{3.49b}$$

Eq. 3.49a is valid because $\boldsymbol{L}^{(N_a)}$ is invertible and for Eq. 3.49b, we need to show that

$$\langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda \geq \boldsymbol{r}^{(N_a+1)^\top} \boldsymbol{L}^{(N_a)^{-1^\top}} \boldsymbol{L}^{(N_a)^{-1}} \boldsymbol{r}^{(N_a+1)} \tag{3.50}$$

so that the term under the square root does not become negative. We use *Sylvester's law of inertia* [95] to show that if Eq. 3.50 is violated, $\boldsymbol{A}^{(N_a+1)}$ would violate the positive definite property. Since we know that there exists a Cholesky decomposition for $\boldsymbol{A}^{(N_a+1)}$ (as well as for $\boldsymbol{A}^{(N_a)}$), this cannot be the case and Eq. 3.50 is true.

Computing $\boldsymbol{v}$ is in $\mathcal{O}\left(N_a^2\right)$ because we need to invert the lower triangular matrix $\boldsymbol{L}^{(N_a)}$ and multiply $\boldsymbol{r}^{(N_a+1)}$ to it. Then, computing s is in $\mathcal{O}\left(N_a\right)$. In total, updating the Cholesky factor after adding one point to the grid of size N_a is performed with $\mathcal{O}\left(N_a^2\right)$ operations.

Coarsening Removing a point from the grid of size N_a reduces $\boldsymbol{A}^{(N_a)}$ by one row and one column, as can be seen in Eq. 3.29. The index of the removed grid point is denoted with j. We show how to construct the Cholesky factor $\boldsymbol{L}^{(N_a \setminus j)} \in \mathbb{R}^{(N-1)\times(N-1)}$ of $\boldsymbol{A}^{(N_a \setminus j)} \in \mathbb{R}^{(N-1)\times(N-1)}$ from the Cholesky factor $\boldsymbol{L}^{(N_a)}$ of $\boldsymbol{A}^{(N_a)}$. To this end, we either permute the j-th row and column towards the first row and column until they become the first row and column, or we permute the j-th row and column towards the last row and column until they become the last row and column. Which direction we choose is investigated in the following.

When permuting towards the last row/column, we are looking to reconstruct the

Cholesky factor $\boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)}$ of

$$\begin{aligned}
\boldsymbol{A}_{j\to\text{last}}^{(N_a\setminus j)} &= \begin{pmatrix} \boldsymbol{R}_{(1:j-1;1:j-1)}^{(N_a)} & \boldsymbol{R}_{(1:j-1;j+1:N_a)}^{(N_a)} & \boldsymbol{R}_{(1:j-1;j)}^{(N_a)} \\ \boldsymbol{R}_{(j+1:N_a;1:j-1)}^{(N_a)} & \boldsymbol{R}_{(j+1:N_a;j+1:N_a)}^{(N_a)} & \boldsymbol{R}_{(j+1:N_a;j)}^{(N_a)} \\ \boldsymbol{R}_{(j;1:j-1)}^{(N_a)} & \boldsymbol{R}_{(j;j+1:N_a)}^{(N_a)} & R_{(j;j)}^{(N_a)} \end{pmatrix} + \lambda \boldsymbol{I}^{(N_a)} \\
&= \begin{pmatrix} \boldsymbol{A}^{(N_a\setminus j)} & \boldsymbol{v} \\ \boldsymbol{v}^\top & R_{(j;j)}^{(N_a)} + \lambda \end{pmatrix}
\end{aligned} \tag{3.51}$$

from $\boldsymbol{L}^{(N_a)}$ (with $\boldsymbol{v} = \begin{pmatrix} \boldsymbol{R}_{(1:j-1;j)}^{(N_a)} \\ \boldsymbol{R}_{(j+1:N_a;j)}^{(N_a)} \end{pmatrix}$). $\boldsymbol{A}_{j\to\text{last}}^{(N_a\setminus j)}$ is obtained from $\boldsymbol{A}^{(N_a)}$ by applying a permutation matrix $\boldsymbol{P}_{j\to\text{last}} \in \{0,1\}^{N_a\times N_a}$ to it:

$$\boldsymbol{A}_{j\to\text{last}}^{(N_a\setminus j)} = \boldsymbol{P}_{j\to\text{last}}^\top \boldsymbol{A}^{(N_a)} \boldsymbol{P}_{j\to\text{last}}\,. \tag{3.52}$$

Applying $\boldsymbol{P}_{j\to\text{last}}^\top$ to $\boldsymbol{L}^{(N_a)}$ from the left results in (at most) $N_a - j$ non-zero entries above the main diagonal. To get rid of these entries, we apply a series of givens rotations [34] $\boldsymbol{U_i}$ to the result. Since every $\boldsymbol{U_i}$ is orthogonal, so is the product

$$\boldsymbol{U} = \prod_i \boldsymbol{U_i}\,. \tag{3.53}$$

We obtain the Cholesky factor $\boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)}$ as

$$\boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)} = \boldsymbol{P}_{j\to\text{last}}^\top \boldsymbol{L}^{(N_a)} \boldsymbol{U}\,. \tag{3.54}$$

We show, that $\boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)}$ is indeed the Cholesky factor of $\boldsymbol{A}_{j\to\text{last}}^{(N_a\setminus j)}$ per

$$\begin{aligned}
\boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)} \boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)\top} &= \boldsymbol{P}_{j\to\text{last}}^\top \boldsymbol{L}^{(N_a)} \boldsymbol{U} \cdot \boldsymbol{U}^\top \boldsymbol{L}^{(N_a)\top} \boldsymbol{P}_{j\to\text{last}} \\
&= \boldsymbol{P}_{j\to\text{last}}^\top \boldsymbol{L}^{(N_a)} \boldsymbol{L}^{(N_a)\top} \boldsymbol{P}_{j\to\text{last}} \\
&= \boldsymbol{P}_{j\to\text{last}}^\top \boldsymbol{A}^{(N_a)} \boldsymbol{P}_{j\to\text{last}} \\
&= \boldsymbol{A}_{j\to\text{last}}^{(N_a\setminus j)}\,.
\end{aligned}$$

Finally, with Eq. 3.51 we obtain $\boldsymbol{L}^{(N_a\setminus j)}$ by just deleting the last row and the last column from $\boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)}$:

$$\boldsymbol{L}^{(N_a\setminus j)} = \boldsymbol{L}_{j\to\text{last}\,(1:N-1;1:N-1)}^{(N_a\setminus j)}\,. \tag{3.55}$$

Alternatively to permuting the j-th row and column to the end, we can also permute it towards the first row/column. In this case, we are looking to reconstruct the Cholesky factor $\boldsymbol{L}_{j\to\text{first}}^{(N_a\setminus j)}$ of

$$\begin{aligned}
\boldsymbol{A}_{j\to\text{first}}^{(N_a\setminus j)} &= \begin{pmatrix} R_{(j;j)}^{(N_a)} & \boldsymbol{R}_{(j;1:j-1)}^{(N_a)} & \boldsymbol{R}_{(j;j+1:N_a)}^{(N_a)} \\ \boldsymbol{R}_{(1:j-1;j)}^{(N_a)} & \boldsymbol{R}_{(1:j-1;1:j-1)}^{(N_a)} & \boldsymbol{R}_{(1:j-1;j+1:N_a)}^{(N_a)} \\ \boldsymbol{R}_{(j+1:N_a;j)}^{(N_a)} & \boldsymbol{R}_{(j+1:N_a;1:j-1)}^{(N_a)} & \boldsymbol{R}_{(j+1:N_a;j+1:N_a)}^{(N_a)} \end{pmatrix} + \lambda \boldsymbol{I}^{(N_a)} \\
&= \begin{pmatrix} R_{(j;j)}^{(N_a)} + \lambda & \boldsymbol{v}^\top \\ \boldsymbol{v} & \boldsymbol{A}^{(N_a\setminus j)} \end{pmatrix}
\end{aligned} \tag{3.56}$$

from $\boldsymbol{L}^{(N_a)}$ (with $\boldsymbol{v} = \begin{pmatrix} \boldsymbol{R}_{(1:j-1;j)}^{(N_a)} \\ \boldsymbol{R}_{(j+1:N_a;j)}^{(N_a)} \end{pmatrix}$). Analogous to the procedure above where $\boldsymbol{L}_{j\to\text{last}}^{(N_a\setminus j)}$ is constructed with the help of the permutation matrix $\boldsymbol{P}_{j\to\text{last}}$ and a series of givens rotations, $\boldsymbol{L}^{(N_a\setminus j)}$ can also be constructed through $\boldsymbol{L}_{j\to\text{first}}^{(N_a\setminus j)}$ by looking at the analogous permutation matrix $\boldsymbol{P}_{j\to\text{first}}$ and an analogous series of givens rotations.

To obtain $\boldsymbol{L}^{(N_a\setminus j)}$, we cannot just delete the first row and first column from $\boldsymbol{L}_{j\to\text{first}}^{(N_a\setminus j)}$, because comparing the entries in

$$\begin{aligned}
&\begin{pmatrix} R_{(j;j)}^{(N_a)} + \lambda & \boldsymbol{v}^\top \\ \boldsymbol{v} & \boldsymbol{A}^{(N_a\setminus j)} \end{pmatrix} = \boldsymbol{A}_{j\to\text{first}}^{(N_a\setminus j)} = \boldsymbol{L}_{j\to\text{first}}^{(N_a\setminus j)} \boldsymbol{L}_{j\to\text{first}}^{(N_a\setminus j)\top} \\
&= \begin{pmatrix} s & \boldsymbol{0} \\ \boldsymbol{w} & \overline{\boldsymbol{L}}^{(N_a\setminus j)} \end{pmatrix} \begin{pmatrix} s & \boldsymbol{w}^\top \\ \boldsymbol{0} & \overline{\boldsymbol{L}}^{(N_a\setminus j)\top} \end{pmatrix} = \begin{pmatrix} s^2 & s\boldsymbol{w}^\top \\ s\boldsymbol{w} & \boldsymbol{w}\boldsymbol{w}^\top + \overline{\boldsymbol{L}}^{(N_a\setminus j)} \overline{\boldsymbol{L}}^{(N_a\setminus j)\top} \end{pmatrix}
\end{aligned} \tag{3.57}$$

with $s \in \mathbb{R}$ and $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^{N_a - 1}$ yields, that

$$\boldsymbol{A}^{(N_a\setminus j)} = \boldsymbol{w}\boldsymbol{w}^\top + \overline{\boldsymbol{L}}^{(N_a\setminus j)} \overline{\boldsymbol{L}}^{(N_a\setminus j)\top}, \tag{3.58}$$

which means so far, we have obtained the Cholesky factor of $\boldsymbol{A}^{(N_a\setminus j)} - \boldsymbol{w}\boldsymbol{w}^\top$, but not of $\boldsymbol{A}^{(N_a\setminus j)}$.

However, we obtain $\boldsymbol{L}^{(N_a\setminus j)}$ by changing $\overline{\boldsymbol{L}}^{(N_a\setminus j)}$ according to what happens when we perform a rank-1 update on $\boldsymbol{A}^{(N_a\setminus j)} - \boldsymbol{w}\boldsymbol{w}^\top$ by adding $\boldsymbol{w}\boldsymbol{w}^\top$ to it. We notice, that finding an orthogonal matrix $\overline{\boldsymbol{U}} \in \mathbb{R}^{N\times N}$, so that

$$\begin{pmatrix} \overline{\boldsymbol{L}}^{(N_a\setminus j)} & \boldsymbol{w} \end{pmatrix} \overline{\boldsymbol{U}} = \begin{pmatrix} \boldsymbol{L}^{(N_a\setminus j)} & \boldsymbol{0} \end{pmatrix} \tag{3.59}$$

suffices, because

$$\begin{aligned}
\boldsymbol{L}^{(N_a\setminus j)} \boldsymbol{L}^{(N_a\setminus j)\top} &= \begin{pmatrix} \boldsymbol{L}^{(N_a\setminus j)} & \boldsymbol{0} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{L}^{(N_a\setminus j)\top} \\ \boldsymbol{0} \end{pmatrix} \\
&= \begin{pmatrix} \overline{\boldsymbol{L}}^{(N_a\setminus j)} & \boldsymbol{w} \end{pmatrix} \overline{\boldsymbol{U}} \cdot \overline{\boldsymbol{U}}^\top \begin{pmatrix} \overline{\boldsymbol{L}}^{(N_a\setminus j)\top} \\ \boldsymbol{w}^\top \end{pmatrix}
\end{aligned}$$

$$
\begin{aligned}
&= \begin{pmatrix} \overline{L}^{(N_a \backslash j)} & w \end{pmatrix} \begin{pmatrix} \overline{L}^{(N_a \backslash j)^\top} \\ w^\top \end{pmatrix} \\
&= \overline{L}^{(N_a \backslash j)} \overline{L}^{(N_a \backslash j)^\top} + w w^\top \\
&= A^{(N_a \backslash j)} \,.
\end{aligned}
$$

$\begin{pmatrix} \overline{L}^{(N_a \backslash j)} & w \end{pmatrix}$ is a lower diagonal matrix with an additional column of non-zero elements appended to it's right. So, we construct $\overline{U}$ as a series of givens rotations that removes those N_a non-zero elements one by one. Each such a givens rotation is expressed as an orthogonal matrix $\overline{U_i} \in \mathbb{R}^{N_a \times N_a}$, so the product

$$\overline{U} = \prod_i \overline{U_i} \tag{3.60}$$

is orthogonal too.

The question remains, when it is better to construct $L^{(N_a \backslash j)}$ via $L_{j \to \text{last}}^{(N_a \backslash j)}$ and when to construct it via $L_{j \to \text{first}}^{(N_a \backslash j)}$. In both cases, we need to construct and apply a series of givens rotations to get rid of the non-zeros above the main diagonal in the permuted Cholesky factor. For $L_{j \to \text{last}}^{(N_a \backslash j)}$, the number of such non-zero entries decreases the closer j is to N_a. For $L_{j \to \text{first}}^{(N_a \backslash j)}$, the number of such non-zero entries decreases the closer j is to 1, but we also need to construct and apply the givens rotations to reflect the above mentioned rank-1 update. To find out where the break-even point lies, we test both methods on the example of system matrix stemming from a grid of dimension 5 and level 5 (resulting in 1,471 grid points) and measure the execution times for coarsening each point of that grid. For the testing platform we used the workstation specified in Sec. B.1. The results are shown in Fig. 3.6. The first observation we make is that permuting towards the end is more expensive for grid points of smaller index than for those of higher index, whereas the reverse holds for permuting towards the front. Also, we observe, that permuting towards the end of the matrix is faster independent of the index j of the grid point in question. We postulate the theory, that this holds not only for this grid configuration but in general. To validate this claim, we test the executing time of coarsening the first grid points with both methods on the example of grids with various size. In Tab. 3.1, we see that indeed, permuting towards the end is always faster. For grid points of higher index, the execution times differ even more for the two methods. Thus in total, we conclude that permuting towards the end is to be preferred for all settings and we choose this technique as the default one from now on.

3.2.3.3.3 Incomplete Cholesky Decomposition

Upon close inspection of R, we note that depending on the dimension and level of the sparse grid, for a significant number of pairs $(i, j) \in [N]^2$:

$$R_{ij} = 0 \,. \tag{3.61}$$

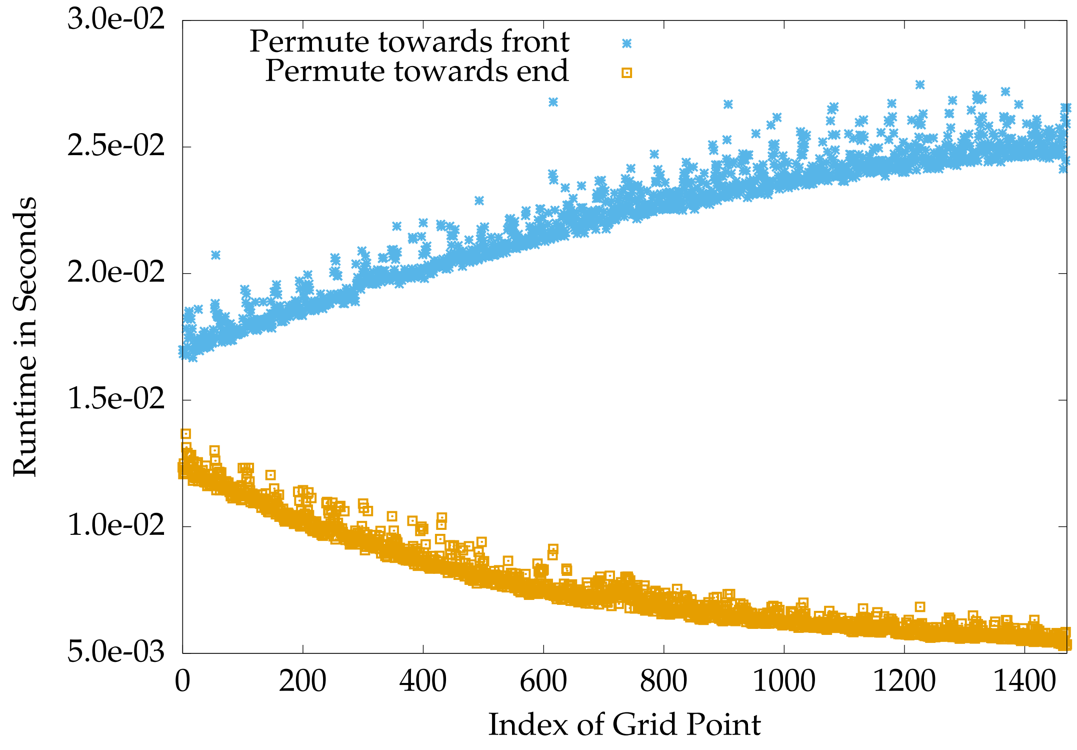


Figure 3.6.: Runtimes for updating the Cholesky decomposition when coarsening grid points by permuting towards the front (blue curve) vs. permuting towards the end (orange curve). Permuting towards the end is faster for grid points of all indices.
Raw data for this figure: Tab. C.5.

In fact, it holds that

$$R_{ij} = 0 \Leftrightarrow \text{ The supports of } \varphi_i \text{ and } \varphi_j \text{ do not overlap}\,. \tag{3.62}$$

The ratio of zeros in $\boldsymbol{R}$ (and thus in $\boldsymbol{A}$) grows with increasing level and fixed dimension but decreases with increasing dimension and fixed level, as depicted in Fig. 3.4. Thus, for lower dimensional problems, $\boldsymbol{A}$ is called *sparse* for higher levels of the sparse grid. We benefit from such a sparse matrix by computing the incomplete Cholesky decomposition [34] instead of the exact Cholesky decomposition.

To this end, we define the *sparsity pattern* $S_{\boldsymbol{M}}$ of a positive matrix $\boldsymbol{M} \in \mathbb{R}^{N\times N}$ as:

$$S_{\boldsymbol{M}} = \left\{ (i,j) \;\middle|\; (i,j) \in [N]^2, M_{ij} > 0 \right\}\,. \tag{3.63}$$

The incomplete Cholesky factorization approximates $\boldsymbol{A}$ instead of representing it exactly:

$$\boldsymbol{A} \approx \mathring{\boldsymbol{L}}\mathring{\boldsymbol{L}}^{\mathsf{T}}\,, \tag{3.64}$$

where $\mathring{\boldsymbol{L}} \in \mathbb{R}^{N\times N}$ is a sparse lower triangular matrix, which we call the incomplete Cholesky factor. Furthermore, we set

$$S_{\mathring{\boldsymbol{L}}} = \{(i,j) \in S_{\boldsymbol{A}} \;\mid\; i \geq j\} \tag{3.65}$$

Table 3.1.: Runtimes in seconds for updating the Cholesky decomposition when deleting the first grid point from the grid. Permuting towards the end is always faster than permuting towards the front for the tested grid sizes.

No. of Grid Points	Runtime permuting to front	Runtime permuting to end
71	$2.6 \cdot 10^{-5}s$	$1.3 \cdot 10^{-5}s$
111	$6.8 \cdot 10^{-5}s$	$2.7 \cdot 10^{-5}s$
209	$2.9 \cdot 10^{-4}s$	$1.8 \cdot 10^{-4}s$
351	$8.9 \cdot 10^{-4}s$	$5.0 \cdot 10^{-4}s$
769	$4.9 \cdot 10^{-3}s$	$2.1 \cdot 10^{-3}s$
1,471	$0.017s$	$0.012s$
2,561	$0.069s$	$0.038s$
4,159	$0.14s$	$0.085s$
5,503	$0.27s$	$0.18s$
10,625	$2.0s$	$1.3s$
18,943	$5.1s$	$3.2s$

and enforce exactness for the decomposition only at the elements of the sparsity pattern as

$$\forall (i,j) \in S_{\mathring{L}} : A_{ij} \overset{!}{=} \left(\mathring{\boldsymbol{L}}\mathring{\boldsymbol{L}}^\top\right)_{ij} = \sum_{k=1}^{\min(i,j)} \mathring{L}_{ik}\mathring{L}_{jk} \,. \tag{3.66}$$

Having the sparsity pattern $S_{\mathring{L}}$ already at hand, we could think about not only using it for decreasing the computational runtime of the decomposition but also to decrease the storage requirements of $\mathring{\boldsymbol{L}}$ by employing sparse matrix data structures. However, as we see in Alg. 1 and Alg. 2, a dense matrix data structure allows us to speed up computations by employing vectorization techniques when solving Eq. 3.11 for $\boldsymbol{\alpha}$. Thus, the data structures for both the Cholesky decomposition and the incomplete Cholesky decomposition are based on dense matrices. The following methods of factorizing and updating the system matrix using the incomplete Cholesky decomposition were implemented into SG++ in a student project [64].

Offline Phase To solve Eq. 3.66, a fixed point iteration is employed. We use that due to construction of $S_{\mathring{L}}$ (Eq. 3.65) it holds that $\min\{i,j\} = j$. Due to the symmetry of $\boldsymbol{A}$, we only consider the case where $i \geq j$ and Eq. 3.66 yields

$$A_{ij} = \sum_{k=1}^{\min\{i,j\}} \mathring{L}_{ik}\mathring{L}_{jk} = \sum_{k=1}^{j} \mathring{L}_{ik}\mathring{L}_{jk} = \left(\sum_{k=1}^{j-1} \mathring{L}_{ik}\mathring{L}_{jk}\right) + \mathring{L}_{ij}\mathring{L}_{jj} \,. \tag{3.67}$$

In case of $i = j$, this breaks down to

$$\mathring{L}_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} \mathring{L}_{ik}^2} \,, \tag{3.68a}$$

and for $i > j$

$$\mathring{L}_{ij} = \frac{A_{ij} - \sum_{k=1}^{j-1} \mathring{L}_{ik}\mathring{L}_{jk}}{\mathring{L}_{jj}} . \tag{3.68b}$$

Finally, we write the fix point iteration in $p \in \mathbb{N}$ for $i \geq j$ as

$$\mathring{L}_{ij}^{(0)} = A_{ij} , \tag{3.69a}$$

$$\mathring{L}_{ij}^{(p+1)} = \begin{cases} \sqrt{c} , & i = j , \\ \frac{c}{\mathring{L}_{jj}^{(p)}} , & \text{else,} \end{cases} \\ \text{with } c = A_{ij} - \sum_{k=1}^{j-1} \mathring{L}_{ik}^{(p)} \mathring{L}_{jk}^{(p)} . \tag{3.69b}$$

Stopping criteria are either a predefined maximum number of iterations, convergence, or sufficient approximation quality of the density estimation. In one sweep p of the fix point iteration, the elements $\mathring{L}_{ij}^{(p)}$ can be computed in parallel. If they are computed sequentially in a row-wise order, the incomplete Cholesky decomposition is obtained after the first sweep.

In the worst case, the runtime complexity of the incomplete Cholesky factorization is $\mathcal{O}\left(N^3\right)$. Computationally, we only profit from this approach if $\left|S_{\mathring{L}}\right| \ll N^2$.

Online Phase The tasks during the online phase are solving the system of linear equations Eq. 3.11 for $\boldsymbol{\alpha}$ using the computed incomplete Cholesky factor and updating the incomplete Cholesky factor according to model changes. Although it could be computational feasible to compute the incomplete Cholesky decomposition from scratch every time the grid is adapted, we incorporate the changes reusing the previously computed incomplete Cholesky factor.

Solving the system To solve Eq. 3.11, we could directly employ Eq. 3.45a and Eq. 3.45b from the Cholesky decomposition. However, since the factor itself is already just an approximation, we compute α as an approximation using a parallelized Jacobi method [10].

For the forward substitution step, we directly use the fact that we want to access elements of $\mathring{\boldsymbol{L}}$ row-wise. Therefore, we use node-level parallelization as well as SIMD vectorization as depicted in Alg. 1. In a standard backward substitution routine, the elements of $\mathring{\boldsymbol{L}}$ would need to be accessed column-wise, which renders the vectorization inefficient due to the row-wise storage of the elements. Thus, we reorganize the sequence in which the result of the backward substitution is computed by iterating over $\mathring{\boldsymbol{L}}$ the same way we did in the forward substitution (ref. Alg. 1) and storing

Algorithm 1 Forward propagation with an incomplete Cholesky factor

Precondition: N is the current number of grid points
Precondition: `sweeps` is the number of Jacobi iterations
Precondition: $\mathring{\boldsymbol{L}}$ is the (lower triangular) incomplete Cholesky factor of size $N \times N$
Precondition: $\boldsymbol{b}$ is the right-hand-side of Eq. 3.11 of size N
Precondition: $\boldsymbol{y}$ is a vector of size N (initially containing zeros) were the solution of the forward propagation is stored into

```
function ForwardPropagation($N$, sweeps, $\mathring{\boldsymbol{L}}, \boldsymbol{b}$)
    for sweep ← 1 to sweeps do
        parallel for $i \leftarrow 1$ to $N$ do                ▷ Node-level parallelization
            $s \leftarrow 0.0$
            vectorized for $j \leftarrow 1$ to $i$ do               ▷ SIMD vectorization
                $s \leftarrow s + \mathring{L}_{ij} \cdot y_i$
            $y_i \leftarrow \frac{b_i - s}{L_{ii}}$
    return $\boldsymbol{y}$
```

intermediary results in a vector (instead of a scalar value). In a second parallelized loop, we then compute the final solution α of Eq. 3.11. This algorithm is depicted in Alg. 2.

Optimizing λ When λ is changed as described in Sec. 3.2.2.3, we take the stale incomplete Cholesky factor $\mathring{\boldsymbol{L}}$ as an initial guess for the fix point iteration (Eq. 3.69) [3] of $\widetilde{\mathring{\boldsymbol{L}}}$ and run it again until convergence or a sufficient accuracy is reached. The following initialization is used instead of Eq. 3.69a:

$$\widetilde{\mathring{L}}_{ij}^{(0)} = \mathring{L}_{ij} \,. \tag{3.70}$$

Refinement When adding a point to the grid of size N_a per Eq. 3.24, $S_{A^{(N_a)}}$ is extended and thus, so is $S_{\mathring{L}^{(N_a)}}$. Now, we could just employ the techniques introduced in Sec. 3.2.3.3.2 (Eq. 3.49) to compute $\mathring{\boldsymbol{L}}^{(N_a+1)}$ from $\mathring{\boldsymbol{L}}^{(N_a)}$. Instead, we compute the new elements of the incomplete Cholesky factor the same way the old ones were computed. In order to solve Eq. 3.66, it suffices to apply the fix point iteration Eq. 3.69 to the new elements of $S_{\mathring{L}^{(N_a+1)}}$, because the previously computed elements of $\mathring{\boldsymbol{L}}^{(N_a)}$ do not depend on the new ones as we see in Eq. 3.68.

Coarsening When removing a point (with index j) from the grid per Eq. 3.29, $S_{A^{(N_a)}}$ is reduced and thus, so is $S_{\mathring{L}^{(N_a)}}$. We have two choices:

1. Employ the techniques introduced in Sec. 3.2.3.3.2 (Eq. 3.51 or Eq. 3.56), or

Algorithm 2 Backward propagation with an incomplete Cholesky factor

Precondition: N is the current number of grid points
Precondition: `sweeps` is the number of Jacobi iterations
Precondition: $\mathring{\boldsymbol{L}}$ is the (lower triangular) incomplete Cholesky factor of size $N \times N$
Precondition: $\boldsymbol{y}$ is a vector of size N containing the solution of the forward propagation
Precondition: $\boldsymbol{\alpha}$ is a vector of size N (initially containing zeros) were the solution of the backward propagation is stored into

```
function BackwardPropagation($N$, sweeps, $\mathring{\boldsymbol{L}}$, $\boldsymbol{y}$)
    for sweep ← 1 to sweeps do
        $\boldsymbol{s} \leftarrow \{0.0\}^N$        ▷ Initialize $\boldsymbol{s}$ as $N$-dimensional vector containing zeros
        parallel for $i \leftarrow 1$ to $N$ do        ▷ Node-level parallelization
            vectorized for $j \leftarrow 1$ to $i$ do        ▷ SIMD vectorization
                $s_j \leftarrow s_j + \mathring{L}_{ij} \cdot \alpha_i$
        parallel for $i \leftarrow 1$ to $N$ do        ▷ Second (node-level) parallel for-loop aggregating the results
            $\alpha_i \leftarrow \frac{y_i - s_k}{L_{ii}}$
    return $\boldsymbol{\alpha}$
```

2. drop all the columns $j \dots N_a$ from $\mathring{\boldsymbol{L}}^{(N_a)}$ and recompute them according to Eq. 3.68 and the new $S_{\mathring{\boldsymbol{L}}^{(N_a \setminus j)}}$.

Especially when removing multiple points from the grid, the second option is faster when we directly remove all the columns from the end up to the smallest j.

3.2.3.3.4 Tridiagonal Decomposition

With the tridiagonal decomposition [34], we are looking to factorize $\boldsymbol{A}$ as

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^{\mathsf{T}}, \tag{3.71}$$

with $\boldsymbol{Q} \in \mathbb{R}^{N \times N}$ being orthogonal and $\boldsymbol{T} \in \mathbb{R}^{N \times N}$ being a symmetric, tridiagonal matrix. Such a decomposition exists for every quadratic and symmetric matrix, $\boldsymbol{A}$ obviously fulfills those criteria. This decomposition is derived from the QR-decomposition

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}, \tag{3.72}$$

where the Hessenberg-matrix $\boldsymbol{R} \in \mathbb{R}^{N \times N}$ is of upper-triangular form with an additional sub-diagonal. Due to the symmetry of $\boldsymbol{A}$, applying $\boldsymbol{Q}$ also from the right (after obtaining it) conserves the previously obtained zeros on the lower triangular part of the matrix, yielding $\boldsymbol{T} = \boldsymbol{R}\boldsymbol{Q}^{\mathsf{T}}$.

We choose the tridiagonal decomposition, because we desire an easy way to compute $\boldsymbol{\alpha} = \boldsymbol{A}^{-1}\boldsymbol{b}$ later. Thus, for all steps of the process, we keep in mind that we always require an efficient algorithm to obtain $\boldsymbol{A}^{-1}$. The following methods of factorizing and updating the system matrix using the tridiagonal decomposition were implemented into SG++ in a bachelor project [14].

Offline Phase There exist multiple methods to compute the orthogonal factor $\boldsymbol{Q}$:

1. The Gram-Schmidt process [34],
2. Givens rotations [34],
3. or Householder transformations [48].

We chose the latter one, because it possesses a good numeric stability.

First, the reflection matrix $\boldsymbol{H_1}$ is computed to reflect the first column of $\boldsymbol{A}$ onto a multiple of a unit vector. In the following, $\boldsymbol{H_i}$ is computed to reflect the ith column of $(\boldsymbol{H_{i-1}} \dots \boldsymbol{H_1}\boldsymbol{A})_{i:N;i:N}$ until $i = N - 2$. Since all $\boldsymbol{H_i}$ are orthogonal, so is the product

$$\boldsymbol{Q} := \prod_{i=1}^{N-2} \boldsymbol{H_i}\,, \tag{3.73}$$

and

$$\boldsymbol{T} := \boldsymbol{Q}^\mathsf{T}\boldsymbol{A}\boldsymbol{Q}\,. \tag{3.74}$$

With $\boldsymbol{Q}$ and $\boldsymbol{T}$ at hand, we notice that computing the inverse $\boldsymbol{A}^{-1}$ is inexpensive:

$$\boldsymbol{A}^{-1} = \left(\boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^\mathsf{T}\right)^{-1} = \left(\boldsymbol{Q}^\mathsf{T}\right)^{-1}\boldsymbol{T}^{-1}\boldsymbol{Q}^{-1} = \boldsymbol{Q}\boldsymbol{T}^{-1}\boldsymbol{Q}^\mathsf{T}\,, \tag{3.75}$$

so in order to obtain $\boldsymbol{A}^{-1}$, we only need to invert $\boldsymbol{T}$. This is achieved by solving N equations

$$\boldsymbol{T}\boldsymbol{x_i} = \boldsymbol{e_i} \quad \forall i \in [N]\,, \tag{3.76}$$

whereas $\boldsymbol{x_i} \in \mathbb{R}^N$ denotes the ith column of $\boldsymbol{T}^{-1}$. Due to the symmetric and tridiagonal properties of $\boldsymbol{T}$, each Eq. 3.76 takes $\mathcal{O}(N)$, so the inverse is obtained in $\mathcal{O}\left(N^2\right)$.

Online Phase During the online phase, the system of linear equations Eq. 3.11 is solved for $\boldsymbol{\alpha}$ using the inverse of the tridiagonal decomposition as well as updating the factorization according to changes to the model. It is not feasible to compute and invert the decomposition from scratch every time the grid is adapted. We present a method that allows us to optimize λ at the beginning of the learning process in the online phase without having to recompute the factorization from scratch for each possible value of λ. After that, we are able to perform refinement and coarsening on the grid by computing additive factors for the inverse of the previously obtained triangular decomposition. With $\boldsymbol{A}^{-1}$ always at hand, we are able to solve Eq. 3.11 for $\boldsymbol{\alpha}$ anytime using matrix-vector product routines. Again, all algorithms during the online phase are in $\mathcal{O}\left(N^2\right)$.

To help with the notation of the changes to the model, we define $\boldsymbol{A}^{-1}_{\text{orig}\to \boldsymbol{k}} \in \mathbb{R}^{k\times k}$ as the original inverted system matrix $\boldsymbol{A}^{-1} = \boldsymbol{Q}\boldsymbol{T}^{-1}\boldsymbol{Q}^{\mathsf{T}}$ of size $N \times N$ extended to size $k \times k, k \geq N$ and filled up with zeros at the additional entries. After n_a grid points have been added to the original N grid points and $N_a = N + n_a$, we are looking to hold the inverse of $\boldsymbol{A}^{(N_a)}$ as

$$\boldsymbol{A}^{(N_a)^{-1}} = \boldsymbol{A}^{-1}_{\text{orig}\to \boldsymbol{N_a}} + \boldsymbol{B}_{\boldsymbol{n_a}}\,, \tag{3.77}$$

whereas $\boldsymbol{B}_{\boldsymbol{n_a}} \in \mathbb{R}^{N_a\times N_a}$ and $\boldsymbol{B}_{\boldsymbol{0}} = \boldsymbol{0}$.

Solving the System With the grid currently consisting of $N_a = N + n_a$ points, to solve Eq. 3.11 for $\boldsymbol{\alpha}$, we compute

$$\boldsymbol{\alpha} = \left(\boldsymbol{A}^{-1}_{\text{orig}\to \boldsymbol{N_a}} + \boldsymbol{B}_{\boldsymbol{n_a}}\right)\boldsymbol{b} = \boldsymbol{A}^{-1}_{\text{orig}\to \boldsymbol{N_a}}\boldsymbol{b} + \boldsymbol{B}_{\boldsymbol{n_a}}\boldsymbol{b}\,, \tag{3.78}$$

which consists of several matrix-vector products and is thus in $\mathcal{O}\left(N^2\right)$.

Optimizing λ As soon as data points are at hand, λ is tuned to find a good balance between fitting the model to the data, but not overfitting to it. The changes reflected in Eq. 3.33 should also be applied to the factors of the tridiagonal decomposition:

$$\begin{aligned}
\widehat{\boldsymbol{A}}^{-1} &= \left(\boldsymbol{A} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\right)^{-1} \\
&= \left(\boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^{\mathsf{T}} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\right)^{-1} \\
&= \left(\boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^{\mathsf{T}} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\boldsymbol{Q}\boldsymbol{Q}^{\mathsf{T}}\right)^{-1} \\
&= \left(\boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^{\mathsf{T}} + \boldsymbol{Q}(\widehat{\lambda} - \lambda)\boldsymbol{I}\boldsymbol{Q}^{\mathsf{T}}\right)^{-1} \\
&= \left(\boldsymbol{Q}\left(\boldsymbol{T} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\right)\boldsymbol{Q}^{\mathsf{T}}\right)^{1} \\
&= \left(\boldsymbol{Q}^{\mathsf{T}}\right)^{-1}\left(\boldsymbol{T} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\right)^{-1}\boldsymbol{Q}^{-1} \\
&= \boldsymbol{Q}\left(\boldsymbol{T} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\right)^{-1}\boldsymbol{Q}^{\mathsf{T}}\,.
\end{aligned} \tag{3.79}$$

Thus, the factor $\boldsymbol{Q}$ remains untouched and we only need to modify the diagonal entries of $\boldsymbol{T}$ as

$$\widehat{\boldsymbol{T}} := \boldsymbol{T} + (\widehat{\lambda} - \lambda)\boldsymbol{I}\,. \tag{3.80}$$

As $\widehat{\boldsymbol{T}}$ remains a symmetric, tridiagonal matrix, computing it's inverse is still in $\mathcal{O}\left(N^2\right)$ (ref. Eq. 3.76). In order to evaluate the quality of the new regularization term, Eq. 3.11 is solved for $\boldsymbol{\alpha}$ by computing

$$\boldsymbol{\alpha} = \boldsymbol{Q}\widehat{\boldsymbol{T}}^{-1}\boldsymbol{Q}^{\mathsf{T}}\boldsymbol{b}\,, \tag{3.81}$$

which is achieved in $\mathcal{O}\left(N^2\right)$ as well. Those steps are repeated as often as necessary with the total process taking $\mathcal{O}\left(N^2\right)$ operations as required for the online phase.

Refinement Originally having started with a grid of N points, we assume that we have already added n_a (with $N_a = N + n_a$) additional points and are now looking to add the $(n_a + 1)$th point. This new addition extends $\boldsymbol{A}^{(N_a)}$ by one row and one column, as depicted Eq. 3.24. $\boldsymbol{A}^{(N_a+1)}$ is expressed as a combination of two rank-1 updates as

$$
\begin{aligned}
\boldsymbol{A}^{(N_a+1)} &= \begin{pmatrix} \boldsymbol{A}^{(N_a)} & \boldsymbol{r}^{(N_a+1)} \\ \boldsymbol{r}^{(N_a+1)^\top} & \langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda - 1 + 1 \end{pmatrix} \\
&= \begin{pmatrix} \boldsymbol{A}^{(N_a)} & \boldsymbol{0} \\ \boldsymbol{0} & 1 \end{pmatrix} + \boldsymbol{r}'^{(N_a+1)} \boldsymbol{e}_{N_a+1}^\top + \boldsymbol{e}_{N_a+1} \boldsymbol{r}'^{(N_a+1)^\top} ,
\end{aligned}
\tag{3.82a}
$$

with

$$
\begin{aligned}
&\boldsymbol{r}'^{(N_a+1)} \in \mathbb{R}^{N_a+1} , \\
&r_i'^{(N_a+1)} := \begin{cases} r_i^{(N_a+1)} , & 1 \le i \le N_a , \\ \frac{\langle \varphi_{N_a+1}, \varphi_{N_a+1} \rangle_{L_2} + \lambda - 1}{2} , & i = N_a + 1 . \end{cases}
\end{aligned}
\tag{3.82b}
$$

In order to incorporate those two outer product summands into the model, the Sherman–Morrison formula [88] is used (twice):

Theorem 3.2.2 (Sherman–Morrison formula)
For $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$, $\left(\boldsymbol{M} + \boldsymbol{u}\boldsymbol{v}^\top\right)$ is invertible if and only if $1 + \boldsymbol{v}^\top \boldsymbol{M}^{-1} \boldsymbol{u} \neq 0$. Then, it holds

$$
\left(\boldsymbol{M} + \boldsymbol{u}\boldsymbol{v}^\top\right)^{-1} = \boldsymbol{M}^{-1} + \boldsymbol{W} \tag{3.83a}
$$

with

$$
\boldsymbol{W} \in \mathbb{R}^{n \times n}, \boldsymbol{W} := -\frac{\boldsymbol{M}^{-1} \boldsymbol{u} \boldsymbol{v}^\top \boldsymbol{M}^{-1}}{1 + \boldsymbol{v}^\top \boldsymbol{M}^{-1} \boldsymbol{u}} . \tag{3.83b}
$$

So, the factors $\boldsymbol{B}_{n_a}$ of Eq. 3.77 are computed inductively with Eq. 3.83 so that Eq. 3.77 always holds. For the $n_a + 1$th additional point, we first obtain $\boldsymbol{B}'_{n_a+1} \in \mathbb{R}^{(N+n_a+1) \times (N+n_a+1)}$ from $\boldsymbol{B}_{n_a}$ reflecting the first addition in Eq. 3.82a and then obtain $\boldsymbol{B}_{n_a+1}$ from $\boldsymbol{B}'_{n_a+1}$ reflecting the second addition in Eq. 3.82a.

$\boldsymbol{M}$ in Eq. 3.83 needs to be invertible, thus we artificially added a 1 at the new diagonal element of the extended system matrix in Eq. 3.82a and subtract it again via $r_{N_a+1}'^{(N_a+1)}$ in

Eq. 3.82b. We now look at its inverse:

$$\begin{aligned}\begin{pmatrix} A^{(N_a)} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix}^{-1} &= \begin{pmatrix} {A^{(N_a)}}^{-1} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} \\ &= \begin{pmatrix} & & & 0 \\ & A^{-1}_{\text{orig}\to N_a} + B_{n_a} & & \vdots \\ & & & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix} = A^{-1}_{\text{orig}\to N_a+1} + \underbrace{\begin{pmatrix} B_{n_a} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix}}_{:=\widetilde{B_{n_a}}} .\end{aligned} \tag{3.84}$$

Now, we handle the first of the two additions $\begin{pmatrix} A^{(N_a)} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} + r'^{(N_a+1)} e^{\mathsf{T}}_{N_a+1}$ per Eq. 3.83 like

$$\begin{aligned}&\left(\begin{pmatrix} A^{(N_a)} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} + r'^{(N_a+1)} e^{\mathsf{T}}_{N_a+1} \right)^{-1} \\ =& A^{-1}_{\text{orig}\to N_a+1} + \underbrace{\widetilde{B_{n_a}} - \frac{\left(A^{-1}_{\text{orig}\to N_a+1} + \widetilde{B_{n_a}} \right) r'^{(N_a+1)} e^{\mathsf{T}}_{N_a+1} \left(A^{-1}_{\text{orig}\to N_a+1} + \widetilde{B_{n_a}} \right)}{1 + e^{\mathsf{T}}_{N_a+1} \left(A^{-1}_{\text{orig}\to N_a+1} + \widetilde{B_{n_a}} \right) r'^{(N_a+1)}}}_{:=B'_{n_a+1}} ,\end{aligned}$$

and we write B'_{n_a+1} as

$$\begin{aligned} B'_{n_a+1} &= \widetilde{B_{n_a}} - \frac{\left(A^{-1}_{\text{orig}\to N_a+1} r'^{(N_a+1)} + \widetilde{B_{n_a}} r'^{(N_a+1)} \right) \left(\overbrace{e^{\mathsf{T}}_{N_a+1} A^{-1}_{\text{orig}\to N_a+1}}^{=0} + e^{\mathsf{T}}_{N_a+1} \widetilde{B_{n_a}} \right)}{1 + \underbrace{e^{\mathsf{T}}_{N_a+1} A^{-1}_{\text{orig}\to N_a+1} r'^{(N_a+1)}}_{=0} + e^{\mathsf{T}}_{N_a+1} \widetilde{B_{n_a}} r'^{(N_a+1)}} \\ &= \widetilde{B_{n_a}} - \frac{\left(A^{-1}_{\text{orig}\to N_a+1} r'^{(N_a+1)} + \widetilde{B_{n_a}} r'^{(N_a+1)} \right) e^{\mathsf{T}}_{N_a+1} \widetilde{B_{n_a}}}{1 + e^{\mathsf{T}}_{N_a+1} \widetilde{B_{n_a}} r'^{(N_a+1)}} .\end{aligned} \tag{3.85}$$

For the second addition $\left(\begin{pmatrix} A^{(N_a)} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} + r'^{(N_a+1)} e^{\mathsf{T}}_{N_a+1} \right) + e_{N_a+1} {r'^{(N_a+1)}}^{\mathsf{T}}$ of Eq. 3.82a, we continue with

$$\begin{aligned}&\left(\left(\begin{pmatrix} A^{(N_a)} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} + r'^{(N_a+1)} e^{\mathsf{T}}_{N_a+1} \right) + e_{N_a+1} {r'^{(N_a+1)}}^{\mathsf{T}} \right)^{-1} \\ =& A^{-1}_{\text{orig}\to N_a+1} + \underbrace{B'_{n_a+1} - \frac{\left(A^{-1}_{\text{orig}\to N_a+1} + B'_{n_a+1} \right) e_{N_a+1} {r'^{(N_a+1)}}^{\mathsf{T}} \left(A^{-1}_{\text{orig}\to N_a+1} + B'_{n_a+1} \right)}{1 + {r'^{(N_a+1)}}^{\mathsf{T}} \left(A^{-1}_{\text{orig}\to N_a+1} + B'_{n_a+1} \right) e_{N_a+1}}}_{:=B_{n_a+1}} ,\end{aligned}$$

and we write $\boldsymbol{B}_{n_a+1}$ as

$$\boldsymbol{B}_{n_a+1} = \boldsymbol{B}'_{n_a+1}$$

$$- \frac{\left(\overbrace{\boldsymbol{A}^{-1}_{\text{orig}\to N_a+1}\boldsymbol{e}_{N_a+1}}^{=\boldsymbol{0}} + \boldsymbol{B}'_{n_a+1}\boldsymbol{e}_{N_a+1}\right)\left(\overbrace{\boldsymbol{r}'^{(N_a+1)^\top}\boldsymbol{A}^{-1}_{\text{orig}\to N_a+1}}^{=\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a+1}\boldsymbol{r}'^{(N_a+1)}\right)^\top} + \boldsymbol{r}'^{(N_a+1)^\top}\boldsymbol{B}'_{n_a+1}\right)}{1 + \underbrace{\boldsymbol{r}'^{(N_a+1)^\top}\boldsymbol{A}^{-1}_{\text{orig}\to N_a+1}\boldsymbol{e}_{N_a+1}}_{=0} + \boldsymbol{r}'^{(N_a+1)^\top}\boldsymbol{B}'_{n_a+1}\boldsymbol{e}_{N_a+1}}$$

$$= \boldsymbol{B}'_{n_a+1} - \frac{\boldsymbol{B}'_{n_a+1}\boldsymbol{e}_{N_a+1}\left(\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a+1}\boldsymbol{r}'^{(N_a+1)}\right)^\top + \boldsymbol{r}'^{(N_a+1)^\top}\boldsymbol{B}'_{n_a+1}\right)}{1 + \boldsymbol{r}'^{(N_a+1)^\top}\boldsymbol{B}'_{n_a+1}\boldsymbol{e}_{N_a+1}}. \tag{3.86}$$

We benefit from the last rearrangement in Eq. 3.86 computationally, because $\boldsymbol{A}^{-1}_{\text{orig}\to N_a+1} \cdot \boldsymbol{r}'^{(N_a+1)}$ has already been computed in Eq. 3.85. Thus, all we need to compute for the second addition is $\boldsymbol{r}'^{(N_a+1)^\top}\boldsymbol{B}'_{n_a+1}$.

In total, we showed that the inverse of $\boldsymbol{A}^{(N_a)}$ can be partitioned as depicted in Eq. 3.77. The computations we need to undertake in Eq. 3.85 and Eq. 3.86 are matrix-vector products, which are computed in $\mathcal{O}\left(N_a^2\right)$.

Coarsening Originally having started with a grid of N points, we assume that we have already added n_a (with $N_a = N + n_a$) additional points and are now looking to remove the grid point with index j. This removal shrinks $\boldsymbol{A}^{(N_a)}$ by one row and one column, as depicted Eq. 3.29. We define $\boldsymbol{A}'^{(N_a\setminus j)} \in \mathbb{R}^{N_a\times N_a}$ as

$$\boldsymbol{A}'^{(N_a\setminus j)} := \begin{pmatrix} \boldsymbol{A}^{(N_a)}_{(1:j-1;1:j-1)} & \boldsymbol{0} & \boldsymbol{A}^{(N_a)}_{(1:j-1;j+1:N_a)} \\ \boldsymbol{0} & 1 & \boldsymbol{0} \\ \boldsymbol{A}^{(N_a)}_{(j+1:N_a;1:j-1)} & \boldsymbol{0} & \boldsymbol{A}^{(N_a)}_{(j+1:N_a;j+1:N_a)} \end{pmatrix} \tag{3.87}$$

and are looking to obtain $\boldsymbol{A}^{(N_a\setminus j)}$ via $\boldsymbol{A}'^{(N_a\setminus j)}$ per

$$\boldsymbol{A}^{(N_a\setminus j)} = \begin{pmatrix} \boldsymbol{A}'^{(N_a)}_{(1:j-1;1:j-1)} & \boldsymbol{A}'^{(N_a)}_{(1:j-1;j+1:N_a)} \\ \boldsymbol{A}'^{(N_a)}_{(j+1:N_a;1:j-1)} & \boldsymbol{A}'^{(N_a)}_{(j+1:N_a;j+1:N_a)} \end{pmatrix}. \tag{3.88}$$

Similar to Eq. 3.82, $\boldsymbol{A}'^{(N_a\setminus j)}$ is also expressed as a combination of two rank-1 updates per

$$\boldsymbol{A}'^{(N_a\setminus j)} = \boldsymbol{A}^{(N_a)} - \boldsymbol{r}'^{(N_a\setminus j)}\boldsymbol{e}_j^\top - \boldsymbol{e}_j\boldsymbol{r}'^{(N_a\setminus j)^\top}, \tag{3.89a}$$

with

$$\begin{aligned} &\boldsymbol{r}'^{(N_a\backslash j)} \in \mathbb{R}^{N_a}\,, \\ &r_i'^{(N_a\backslash j)} := \begin{cases} \left\langle \varphi_i, \varphi_j \right\rangle_{L_2}, & i \neq j\,, \\ \frac{\left\langle \varphi_j, \varphi_j \right\rangle_{L_2} + \lambda - 1}{2}, & i = j\,. \end{cases} \end{aligned} \tag{3.89b}$$

To perform those two subtractions, we are going to use the Sherman-Morrison formula [88] (twice) again (Eq. 3.83) to reflect the update on the inverse of $\boldsymbol{A}^{(N_a)}$ by defining an additive component $\boldsymbol{B}_{n_a\backslash j} \in \mathbb{R}^{N_a \times N_a}$ such that:

$$\boldsymbol{A}'^{(N_a\backslash j)^{-1}} = \boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}_{n_a\backslash j}\,. \tag{3.90}$$

However, a rank-1 update affecting rows and columns in the original system matrix $\boldsymbol{A}$ is not feasible, as it would require a complete recomputation of the decomposition of $\boldsymbol{A}^{(N_a\backslash j)}$ because $\boldsymbol{A}^{-1}$ (and $\boldsymbol{A}$) are only available in decomposed form. Thus, we can only remove points from the grid, whose information is solely stored in $\boldsymbol{B}_{n_a}$. Those are points that have been previously added during a refinement step. Points that have originally been in the grid cannot be coarsened in $\mathcal{O}\left(N^2\right)$ when employing the tridiagonal decomposition. So for now, we require that

$$j \overset{!}{>} N\,. \tag{3.91}$$

For removing the point with index j, we first obtain $\boldsymbol{B}'_{n_a\backslash j} \in \mathbb{R}^{N_a \times N_a}$ from $\boldsymbol{B}_{n_a}$ reflecting the first subtraction in Eq. 3.89a and then obtain $\boldsymbol{B}_{n_a\backslash j}$ from $\boldsymbol{B}'_{n_a\backslash j}$ reflecting the second subtraction in Eq. 3.89a.

Again, $\boldsymbol{M}$ in Eq. 3.83 needs to be invertible, which is why in the definition of $\boldsymbol{A}'^{(N_a\backslash j)}$ (Eq. 3.87) the value of 1 is forced at the jth diagonal entry, which is then constructed via the definition of $r_j'^{(N_a\backslash j)}$ in Eq. 3.89b.

Let's first see how to handle the first subtraction of $\boldsymbol{A}^{(N_a)} - \boldsymbol{r}'^{(N_a\backslash j)}\boldsymbol{e}_j^\top$ in Eq. 3.89a per Eq. 3.83, knowing that $\boldsymbol{A}^{(N_a)^{-1}} = \boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}_{n_a}$ holds (Eq. 3.77):

$$\begin{aligned} &\left(\boldsymbol{A}^{(N_a)} - \boldsymbol{r}'^{(N_a\backslash j)}\boldsymbol{e}_j^\top\right)^{-1} \\ =&\boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \underbrace{\boldsymbol{B}_{n_a} + \frac{\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}_{n_a}\right)\boldsymbol{r}'^{(N_a\backslash j)}\boldsymbol{e}_j^\top\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}_{n_a}\right)}{1 - \boldsymbol{e}_j^\top\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}_{n_a}\right)\boldsymbol{r}'^{(N_a\backslash j)}}}_{:=\boldsymbol{B}'_{n_a\backslash j}}\,, \end{aligned}$$

and we write $\boldsymbol{B}'_{n_a\setminus j}$ as

$$\boldsymbol{B}'_{n_a\setminus j} = \boldsymbol{B}_{n_a} + \frac{\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\boldsymbol{r}'^{(N_a\setminus j)} + \boldsymbol{B}_{n_a}\boldsymbol{r}'^{(N_a\setminus j)}\right)\left(\overbrace{\boldsymbol{e}_j^\top\boldsymbol{A}^{-1}_{\text{orig}\to N_a}}^{=\boldsymbol{0}} + \boldsymbol{e}_j^\top\boldsymbol{B}_{n_a}\right)}{1 - \underbrace{\boldsymbol{e}_j^\top\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\boldsymbol{r}'^{(N_a\setminus j)}}_{=0} - \boldsymbol{e}_j^\top\boldsymbol{B}_{n_a}\boldsymbol{r}'^{(N_a\setminus j)}}$$

$$= \boldsymbol{B}_{n_a} + \frac{\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\boldsymbol{r}'^{(N_a\setminus j)} + \boldsymbol{B}_{n_a}\boldsymbol{r}'^{(N_a\setminus j)}\right)\boldsymbol{e}_j^\top\boldsymbol{B}_{n_a}}{1 - \boldsymbol{e}_j^\top\boldsymbol{B}_{n_a}\boldsymbol{r}'^{(N_a\setminus j)}}. \tag{3.92}$$

The second subtraction of $\left(\boldsymbol{A}^{(N_a)} - \boldsymbol{r}'^{(N_a\setminus j)}\boldsymbol{e}_j^\top\right) - \boldsymbol{e}_j\boldsymbol{r}'^{(N_a\setminus j)^\top}$ in Eq. 3.89a then follows as

$$\left(\left(\boldsymbol{A}^{(N_a)} - \boldsymbol{r}'^{(N_a\setminus j)}\boldsymbol{e}_j^\top\right) - \boldsymbol{e}_j\boldsymbol{r}'^{(N_a\setminus j)^\top}\right)^{-1}$$

$$= \boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \underbrace{\boldsymbol{B}'_{n_a\setminus j} + \frac{\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}'_{n_a\setminus j}\right)\boldsymbol{e}_j\boldsymbol{r}'^{(N_a\setminus j)^\top}\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}'_{n_a\setminus j}\right)}{1 - \boldsymbol{r}'^{(N_a\setminus j)^\top}\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a} + \boldsymbol{B}'_{n_a\setminus j}\right)\boldsymbol{e}_j}}_{:=\boldsymbol{B}_{n_a\setminus j}}$$

and we write $\boldsymbol{B}_{n_a\setminus j}$ as

$$\boldsymbol{B}_{n_a\setminus j} = \boldsymbol{B}'_{n_a\setminus j} + \frac{\left(\overbrace{\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\boldsymbol{e}_j}^{=\boldsymbol{0}} + \boldsymbol{B}'_{n_a\setminus j}\boldsymbol{e}_j\right)\left(\overbrace{\boldsymbol{r}'^{(N_a\setminus j)^\top}\boldsymbol{A}^{-1}_{\text{orig}\to N_a}}^{=\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\boldsymbol{r}'^{(N_a\setminus j)}\right)^\top} + \boldsymbol{r}'^{(N_a\setminus j)^\top}\boldsymbol{B}'_{n_a\setminus j}\right)}{1 - \underbrace{\boldsymbol{r}'^{(N_a\setminus j)^\top}\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\boldsymbol{e}_j}_{=0} - \boldsymbol{r}'^{(N_a\setminus j)^\top}\boldsymbol{B}'_{n_a\setminus j}\boldsymbol{e}_j}$$

$$= \boldsymbol{B}'_{n_a\setminus j} + \frac{\boldsymbol{B}'_{n_a\setminus j}\boldsymbol{e}_j\left(\left(\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\boldsymbol{r}'^{(N_a\setminus j)}\right)^\top + \boldsymbol{r}'^{(N_a\setminus j)^\top}\boldsymbol{B}'_{n_a\setminus j}\right)}{1 - \boldsymbol{r}'^{(N_a\setminus j)^\top}\boldsymbol{B}'_{n_a\setminus j}\boldsymbol{e}_j}. \tag{3.93}$$

We benefit from the last rearrangement in Eq. 3.93 computationally, because $\boldsymbol{A}^{-1}_{\text{orig}\to N_a}\cdot\boldsymbol{r}'^{(N_a\setminus j)}$ has already been computed in Eq. 3.92. Thus, all we need to compute for the second addition is $\boldsymbol{r}'^{(N_a\setminus j)^\top}\boldsymbol{B}'_{n_a\setminus j}$. The computations we need to undertake in Eq. 3.85 and Eq. 3.86 are matrix-vector products, which are obtained in $\mathcal{O}\left(N_a^2\right)$.

$\boldsymbol{B}_{n_a\setminus j}$'s jth row and column are now zero except for the jth diagonal element, which is 1. Also, $\boldsymbol{A}^{-1}_{\text{orig}\to N_a}$'s jth row and column are zero in all elements, thus deleting the jth row and column from it directly yields $A^{-1}_{\text{orig}\to N_a-1}$. So, we construct B_{n_a-1} from $B_{n_a}\backslash j$ by setting

$$B_{n_a-1} := \begin{pmatrix} B_{n_a}\backslash j_{(1:j-1;1:j-1)} & B_{n_a}\backslash j_{(1:j-1;j+1:N_a)} \\ B_{n_a}\backslash j_{(j+1:N_a;1:j-1)} & B_{n_a}\backslash j_{(j+1:N_a;j+1:N_a)} \end{pmatrix} \tag{3.94}$$

and obtain $A^{(N_a\backslash j)^{-1}}$ as

$$A^{(N_a\backslash j)^{-1}} = A^{-1}_{\text{orig}\to N_a-1} + B_{n_a-1} =: A^{(N_a-1)^{-1}}. \tag{3.95}$$

For the refinement and coarsening process, it is now irrelevant which kind of model update (refinement or coarsening) has last been applied. $A^{-1}_{\text{orig}\to N_a} + B_{n_a}$ is used transparently by both methods as stale inverse of the system matrix from which to perform the update. The partitioning of the inverse of $A^{(N_a)}$ as depicted in Eq. 3.77 always holds.

3.2.3.3.5 Comparison of the Matrix Decomposition Methods

When employing the offline/online scheme for grid-based density estimation, the choice of the matrix decomposition technique depends on the scope of operation as well as the asymptotic complexity and the concrete runtime of the algorithms. Both aspects are discussed in the following paragraphs.

Feature Comparison A feature-wise comparison of the introduced matrix decomposition methods is shown in Tab. 3.2. Of the presented matrix decomposition methods, all allow for a solve of Eq. 3.11 for α in quadratic time, which was the motivation to factorize A in the first place. In all cases, the solve is optionally parallelized given a suitable hardware platform.

The factorization of the matrix in the offline phase has cubic complexity for all decomposition techniques except for incomplete Cholesky, which provides the factor in quadratic time. However, since the incomplete Cholesky decomposition only approximates the Cholesky factor, subsequently performing forward and backward substitution to obtain α also yields only an approximation and not the exact solution.

Looking at spatial adaptivity, the eigendecomposition does not allow for the grid to be changed. Basing the model only on an a priori, dataset independent grid does not yield good results, which is why this decomposition is not feasible to use in the context of grid-based density estimation. All other techniques support grid adaptivity, whereat the coarsening of grid points with the tridiagonal decomposition is limited to points that have previously been added to the model in a refinement step. Thus, points from the initial a priori grid cannot be removed from the model when employing the tridiagonal decomposition.

	Eigen	Cholesky	Incomplete Cholesky	Tridiagonal
Exactness of Decomposition	✓	✓	✗	✓
Complexity of Factorization	$\mathcal{O}\left(N^3\right)$	$\mathcal{O}\left(N^3\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^3\right)$
Complexity of Solve for $\boldsymbol{\alpha}$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$
Solve for $\boldsymbol{\alpha}$ parallelizable	✓	✓	✓	✓
Complexity of optimizing λ	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^3\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$
Complexity of adding one point to the grid	$\mathcal{O}\left(N^3\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$
Complexity of removing one point from the grid	$\mathcal{O}\left(N^3\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$	$\mathcal{O}\left(N^2\right)$ for points previously added to the grid

Table 3.2.: Comparison of the matrix factorization techniques used to handle the batch-wise grid-based density estimation.

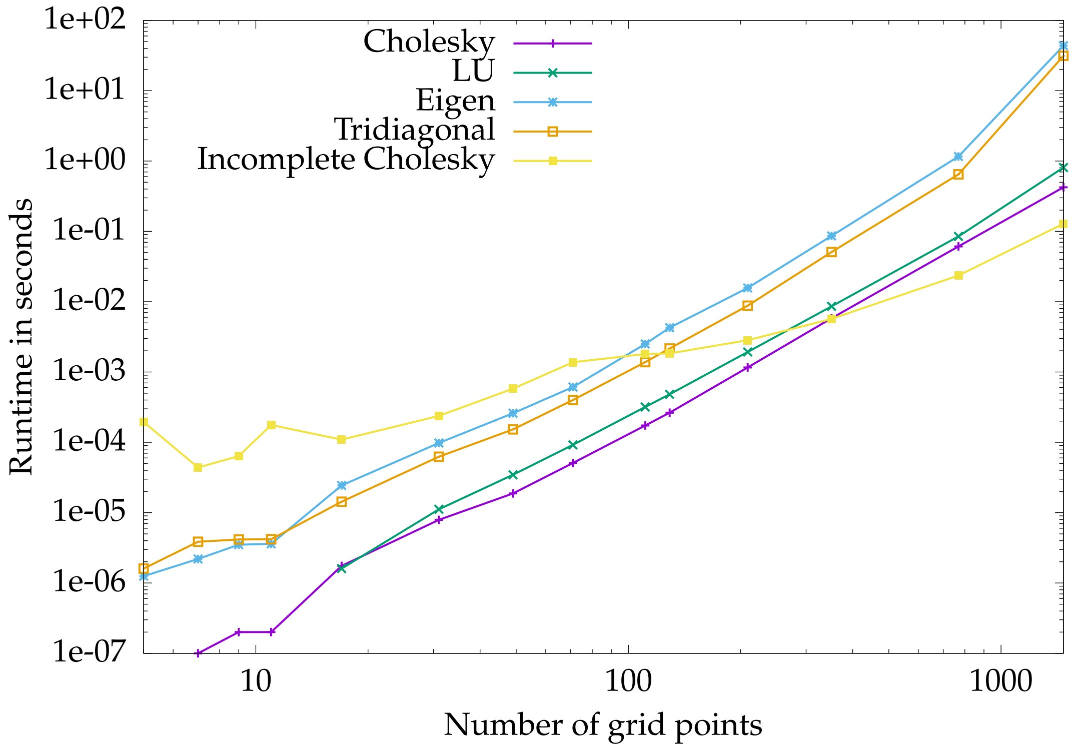


Figure 3.7.: Runtime comparison for factorizing the system matrix for the presented matrix decomposition techniques. For smaller grids, factorizing the matrix with the incomplete Cholesky decomposition is slowest, but the contrary is true for larger grids. Due to exploiting the symmetric and positive definite attributes of the linear system, obtaining the Cholesky decomposition is faster than obtaining the *LU* decomposition and obtaining the tridiagonal decomposition is faster than obtaining the eigendecomposition.
Raw data for this figure: Tab. C.6.

Optimizing the regularization parameter λ is performed in linear time with the eigendecomposition, though the quadratic complexity of the tridiagonal and the incomplete Cholesky decomposition are also acceptable for this step of the online phase. Only with the Cholesky decomposition, the regularization parameter λ cannot be efficiently optimized.

Runtime Comparison In the following, we compare the runtimes of the presented decomposition techniques for the different tasks of factorizing the matrix, solving the system and modifying the decomposition. All tests were executed on the workstation platform (ref. Sec. B.1).

Factorizing the Matrix The runtimes for factorizing the matrix with each of the presented decomposition techniques is shown in Fig. 3.7 for different grid configurations. For the decomposition itself, only the number of grid points rather than the dimensionality and level are of importance for the runtime. We observe that the Cholesky

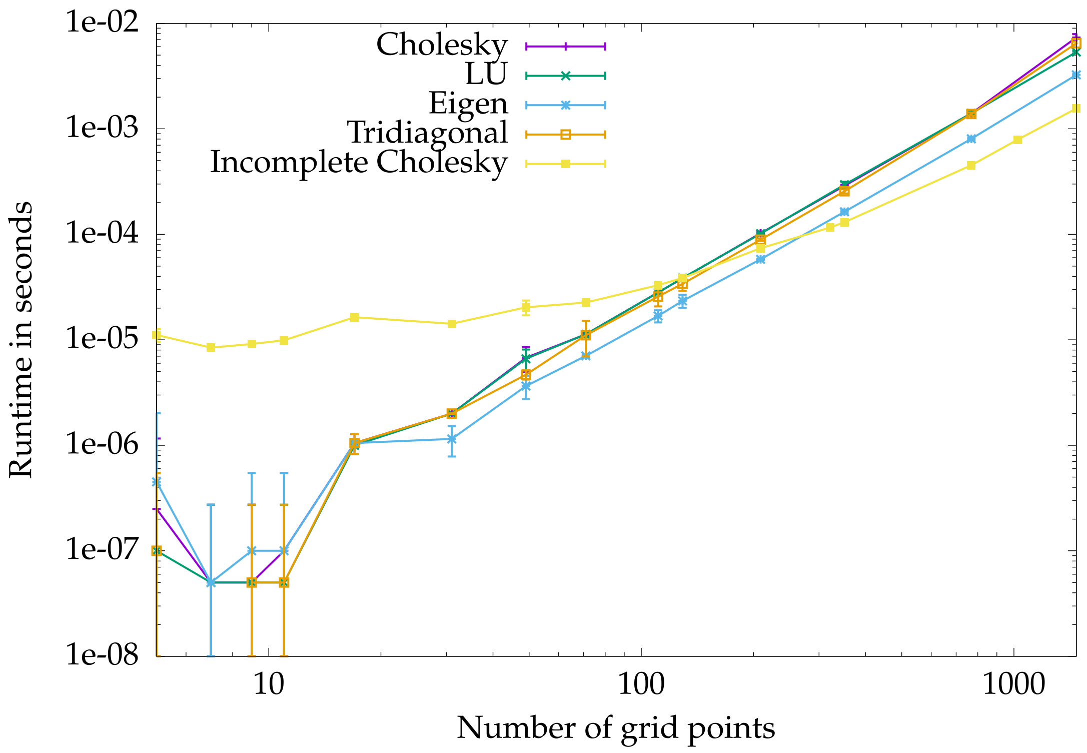
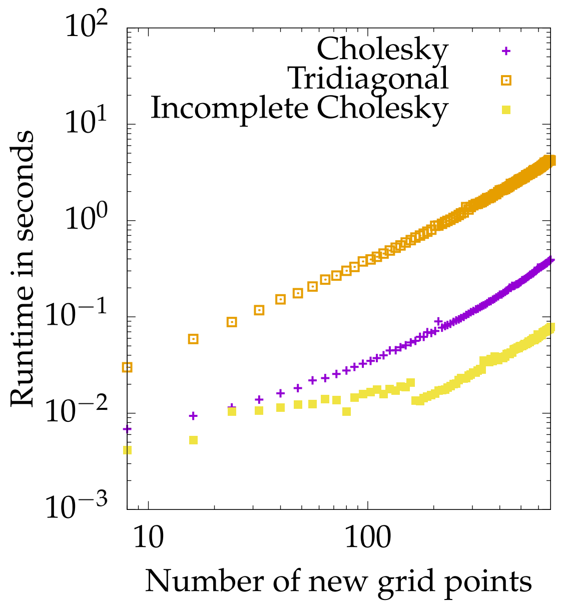


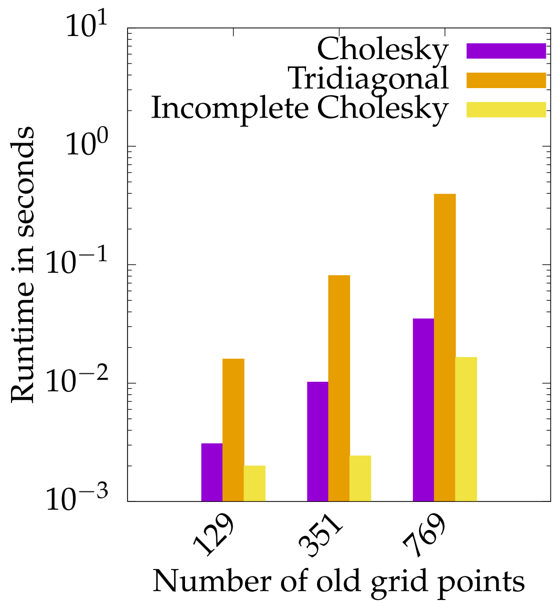
Figure 3.8.: Runtime comparison for solving the linear system. Solving with the incomplete Cholesky decomposition is slowest for small grid but quickly becomes the fastest methods with increasing grid size. Among the other techniques, solving with the eigendecomposition is fastest with the other methods not far behind.
Raw data for this figure: Tab. C.7.

decomposition runs approximately twice as fast than its legacy partner, the *LU* decomposition. We already expected this, because with the Cholesky decomposition, we exploit that the system matrix is symmetric and positive definite which the *LU* decomposition does not. Also, we observe that the tridiagonal decomposition is obtained approximately twice as fast than its legacy partner, the eigendecomposition, which is also the case due to exploiting the symmetry of the matrix. However, we also see that all those four techniques grow with the same rate. Only the incomplete Cholesky decomposition grows with a smaller rate. For a small number of grid points, it performs worse than the counterparts because the iterative scheme introduces some computational overhead. However, with increasing grid size it quickly becomes faster than all other techniques, making it a good choice if obtaining a decomposition fast is the priority.

Solving the System The runtimes for solving the system with the matrix already factorized is shown in Fig. 3.8. While the runtimes fluctuate for small grids, the asymptotic runtime for increasing grid size becomes apparent soon. Again, the incomplete Cholesky decomposition is slowest for small grids but outperforms the other techniques with increasing grid size as expected. Among the exact methods, solving the



(a) Varying the number of new grid points.



(b) Varying the number of initial grid points.

Figure 3.9.: Runtime comparison for matrix decomposition update after refinement. In the first scenario (Subfig. a), the initial grid consists of 769 grid points and we vary the number of points being added to the grid. The incomplete Cholesky decomposition is updated fastest, but the Cholesky decomposition still beats the tridiagonal decomposition by far. The same is observed in the second scenario (Subfig. b), where we refine grids of different initial size by adding 103 new points and measuring the runtimes for updating the decompositions again. Raw data for this figure: Tab. C.8 and Tab. C.9.

system with the eigendecomposition is fastest with the other methods not far behind. In total, the time spent for solving the system is so small for all implemented decomposition techniques that it is insignificant compared to constructing the right-hand side of the equation when processing a data batch.

Refinement Runtime measurements for updating the matrix decomposition after the grid has been refined are shown in Fig. 3.9. Of course, only the three techniques supporting grid adaptivity are under investigation. We compare two setups: In the first setup (Fig. 3.9a) the size of the initial grid is fixed (769 grid points) and we vary the number of grid points that are being added to the grid. In the second setup (Fig. 3.9a) the number of points being added corresponds to 103 for different initial grid sizes. We observe that the incomplete Cholesky decomposition is updated fastest, whereas the Cholesky decomposition comes second before the tridiagonal decomposition. For all tested grid layouts and decomposition methods, the runtimes are feasible to invest during the online phase. However, the asymptotic increase of the runtimes raises the expectation that updating the tridiagonal decomposition is costly to execute for grids with several thousands of grid points.

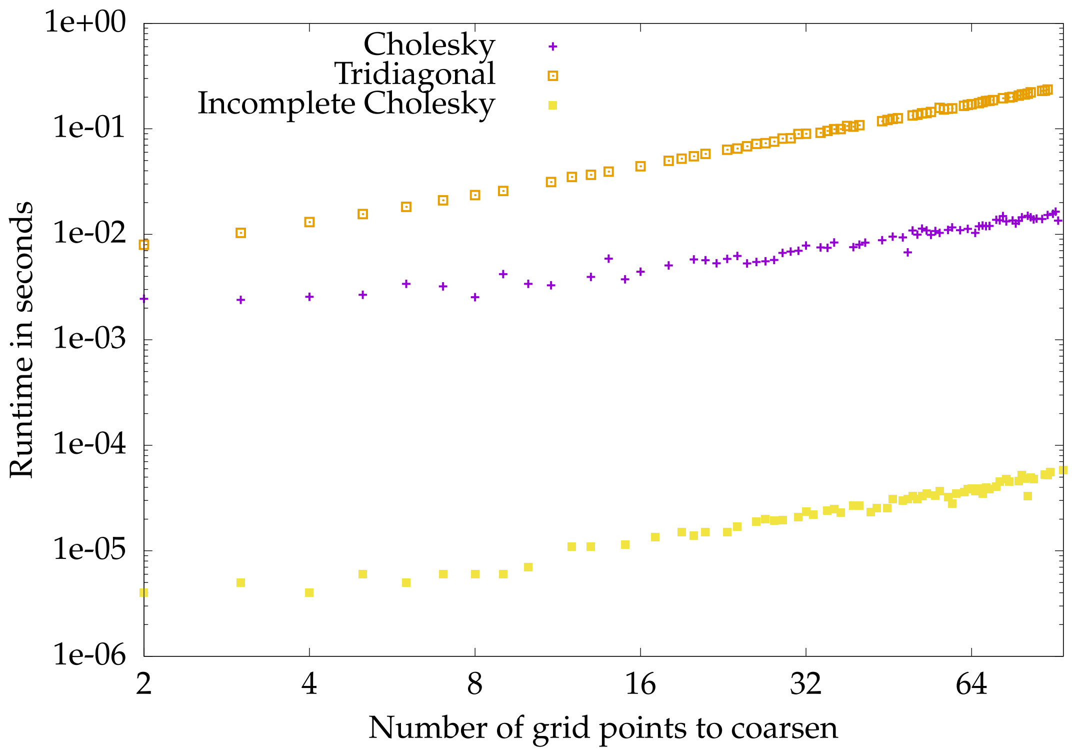


Figure 3.10.: Runtime comparison for matrix decomposition update after coarsening. The size of the initial grid is fixed to 809 grid points and we vary the number of grid points that are being removed from the grid. As for refinement, updating the incomplete Cholesky decomposition is fastest because we only need to delete the according row and column from the factor. For the exact decompositions, the update for the Cholesky decomposition is significantly faster than the update to the tridiagonal decomposition.
Raw data for this figure: Tab. C.10.

Coarsening We measure the runtimes of updating the decomposition after removing various numbers of grid points from a grid of initial size 809 and display them in Fig. 3.10. Again, the tests are limited to the methods supporting coarsening. The indices of the points to be coarsened are chosen randomly, but we limit ourselves to grid points that have previously been added to the grid. Thereby, we can also run the update for the tridiagonal decomposition, which only supports to coarsen these points. Updating the incomplete Cholesky decomposition after a refinement means to just remove the corresponding row and column from the factor which is why this method is fastest by several orders of magnitude. For the other two, the picture is similar to the refinement experiments: Updating the Cholesky decomposition is significantly faster than updating the tridiagonal decomposition. Both methods still achieve feasible runtimes to be employed during the online phase. For larger grids, updating the tridiagonal decomposition is expensive and needs to be considered carefully when configuring the adaptivity settings.

3.2.4. Conclusions

We proposed an incremental learning scheme that allows to employ spatial adaptivity in-between learning data batches. Applying this scheme results in explainable models of the data. Also, we showed how to employ matrix decomposition techniques that allow to be updated after model adaptions. This enables us to use the offline/online scheme together with spatial adaptivity. Thus, we successfully removed the crucial obstacle which prevented the use of the offline/online scheme for problems that require to be adapted to the model at runtime. Note, that those problems could have been tackled before by employing the conjugate gradients technique (ref. Sec. 3.2.3.2), but with far worse runtimes as the comparisons in [76] show. Also, we could have tackled them with regular sparse grids, whereupon the accuracy would have been unsatisfactory. Combining spatial adaptivity with the offline/online scheme now enables us to approximate the density of such datasets with sparse grids both fast and accurate.

3.3. Classification

Another common task in data mining is classification, where we predict the labels of previously unseen data points after training with a set of pre-labeled data. For a discrete set of classes $\mathcal{K}$, the countable training dataset is given by

$$\mathcal{M} \subset \Omega \times \mathcal{K} \tag{3.96}$$

where for $(\boldsymbol{x}, y) \in \mathcal{M}$, y is denoted the label of $\boldsymbol{x}$. The classifier we are looking to learn from $\mathcal{M}$ is

$$c_{\mathcal{M}} \colon \Omega \to \mathcal{K}\,. \tag{3.97}$$

One of the oldest methods to solve this problem are decision trees, but other methods have proven to perform well in different scenarios such as support vector machines [15], logistic regression [101] or kernel estimators [72]. Most notably, neural networks [66] in various facets have gained huge attention in both science and industry in recent years. To perform classification, we build up on top of the density estimation discussed in Sec. 3.2 and employ the Bayes classifier [40]. The idea is to estimate the density of each class separately and then use the maximum a posteriori rule to pick the class of an unclassified sample. In itself, this method is independent of the specific technique used to construct the density estimations. Thus, on the first glance, for the construction of the classifier as we provide it in Sec. 3.3.1, it is inconsequential that we employ sparse grids for the density estimation. However, when it comes to adapting the classifier to the problem, the sparse grid structure is of advantage, as we see in Sec. 3.3.2.

3.3.1. Density Estimation based Bayes Classifier

For our work, we look at the Bayes classifier [68], which has been proven to minimize the probability of misclassified data [22]. Therefore, we state Bayes' theorem:

> **Theorem 3.3.1 (Bayes' theorem)**
> For events A and B, the conditional probability $P(A|B)$ is given by:
> $$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \,. \tag{3.98}$$

This allows us to write the Bayes classifier as

$$c_{\mathcal{M}}(\boldsymbol{x}) = \arg\max_{y \in \mathcal{K}} P(y|\boldsymbol{x}) = \arg\max_{y \in \mathcal{K}} \frac{P(\boldsymbol{x}|y)P(y)}{P(\boldsymbol{x})} \,. \tag{3.99}$$

As $P(\boldsymbol{x})$ is independent of y and we are only interested in the $\arg\max$ over y, we simplify Eq. 3.99 further to

$$c_{\mathcal{M}}(\boldsymbol{x}) = \arg\max_{y \in \mathcal{K}} P(\boldsymbol{x}|y)P(y) \,. \tag{3.100}$$

With the density estimation method presented in Sec. 3.2, the conditional probability $P(\boldsymbol{x}|y)$ is estimated for each class by separating $\mathcal{M}$ like

$$\mathcal{M}_y := \{\boldsymbol{x} \quad | \quad (\boldsymbol{x}, y) \in \mathcal{M}\} \tag{3.101}$$

into the sub-training datasets for each class y and calculating $P(\boldsymbol{x}|y) := f_y(\boldsymbol{x})$ with $\mathcal{M}_y$ as defined in Eq. 3.5. The prior is obtained via $P(y) := |\mathcal{M}_y|$, so our density based classification develops from Eq. 3.100 to

$$c_{\mathcal{M}}(\boldsymbol{x}) = \arg\max_{y \in \mathcal{K}} f_y(\boldsymbol{x}) \cdot |\mathcal{M}_y| \,. \tag{3.102}$$

$c_{\mathcal{M}}(\boldsymbol{x})$ is the class label returned at point $\boldsymbol{x}$. We call this class also the *dominant class* at this point.

We emphasize the many-grid setup of the classifier. For each class y, a separate grid-based density estimation and thus, a separate grid is constructed. In total, the classification model consists of $|\mathcal{K}|$ grids and for each grid the corresponding system matrix, its decomposition, the right-hand side, and the sparse grid solution $\boldsymbol{\alpha_y}$.

This many-grid setup is key for our construction of an explainable classifier. We already discussed that the grid-based density estimations obtained for each class allow for a transparent data model. With the Bayes classifier building up on that, its result is explainable on both levels of its structure – the individual classes represented by a grid-based density estimation for each class and the relation of the classes via the *minimum-risk* criterion.

In order to employ a batch learning scheme, the methods presented in Sec. 3.2.1 are employed for each f_y and the models are then adapted according to Sec. 3.2.2. The entire learning process for the batch-wise classification is shown in Alg. 3. The evaluation of

Algorithm 3 Batch-wise classification with grid-based density estimation

Precondition: $\mathcal{M}_k \subset \Omega \times \mathcal{K}$ is the countable, labeled training dataset for batch k

```
function TrainClassifier
    Assemble $\boldsymbol{A}$ from Eq. 3.13 with given a priori grid configuration
    decomp $\leftarrow$ Factorize $\boldsymbol{A}$ according to the desired decomposition method
    for all classes $y$ do
        decomp$_y$ $\leftarrow$ decomp
        numSamples$_y$ $\leftarrow 0$
    for all batches $k$ do
        Retrieve $\mathcal{M}_k$
        for all classes $y \in \mathcal{K}$ do
            $\mathcal{M}_{k,y} \leftarrow \{\boldsymbol{x} | (\boldsymbol{x}, y) \in \mathcal{M}_k\}$
            if $k = 1$ then
                $\lambda \leftarrow$ Result of the optimization of the regularization parameter for $f_y$ according to Sec. 3.2.2.3
                Update Eq. 3.11 according to the new $\lambda$
            $\boldsymbol{b_{k,y}} \leftarrow$ Compute the right-hand-side of Eq. 3.11 with $\mathcal{M}_{k,y}$ per Eq. 3.25
            $\boldsymbol{\alpha_{k,y}} \leftarrow$ Solution of Eq. 3.11 with $\boldsymbol{b_{k,y}}$ per Sec. 3.2.3
            numSamples$_y$ $\leftarrow$ numSamples$_y$ $+ |\mathcal{M}_{k,y}|$
        Model changes: Identify model changes for all $f_y$ under simultaneous consideration of all $\boldsymbol{\alpha_{k,y}}$    ▷ This step is discussed in detail in Sec. 3.3.2
        for all classes $y$ do
            Incorporate model changes into Eq. 3.11 per Sec. 3.2.2
            Modify decomp$_y$ according to Sec. 3.2.3.3
```

a classifier trained this way is depicted in Alg. 4. For class y, the grid associated with it is denoted with $\mathcal{G}_y$ and the surpluses associated with it as $\boldsymbol{\alpha_y}$. $p_{y,i}$ is the ith grid point in $\mathcal{G}_y$ and $\alpha_{y,i}$ is the surplus at $p_{y,i}$.

3.3.2. Refinement for Classification

Just as for density estimation, we want the model to adapt to the dataset also in classification. Whereas we had only one sparse grid as model for building the probability density estimator, in the classification setting with K classes, the model consists of K probability density functions, one for each class. Thus, the refinement indicators we

Algorithm 4 Evaluation of the Bayes classifier

Precondition: $\mathcal{K}$ are the class labels
Precondition: $\boldsymbol{x} \in \Omega$ is the data point to be classified
Precondition: f_y is the probability density function for class y
Precondition: $\texttt{numSamples}_y$ are the number of samples processed for each class y

```
function EvaluateClassifier
    bestClass ← null
    bestValue ← $-\infty$
    for all classes $y \in \mathcal{K}$ do
        classValue$_y$ ← $f_y(\boldsymbol{x}) \cdot$ numSamples$_y$
        if classValue$_y$ > bestValue then
            bestClass ← $y$
            bestValue ← classValue$_y$
    return bestClass
```

employed for density estimation (ref. Sec. 3.2.2.1) are not applicable to the classification setting.

The goal of refining the model is to increase the classification accuracy, which means, reducing the number of misclassified points. Thus, we want to search for areas in Ω where we can improve the accuracy by refining grid points. Those areas are the ones where Eq. 3.102 yields close values to the dominant class also for other classes, thus areas where

$$\exists y' \in \mathcal{K} \text{ with } y' \neq y := c_{\mathcal{M}}(\boldsymbol{x}) : \quad f_y(\boldsymbol{x}) \cdot \left|\mathcal{M}_y\right| \approx f_{y'}(\boldsymbol{x}) \cdot \left|\mathcal{M}_{y'}\right| , \tag{3.103}$$

which cover also the areas where the dominant class change, thus small connected $\omega \subset \Omega$ where

$$\exists x_1, x_2 \in \omega : \quad c_{\mathcal{M}}(\boldsymbol{x_1}) \neq c_{\mathcal{M}}(\boldsymbol{x_2}) . \tag{3.104}$$

To obtain refinement candidates, we are looking to score all leaf grid points of all classes. A leaf grid point is a grid point, whose children are not yet all contained in the grid. As a result, the grid points with the highest scores are the best refinement candidates.

Collecting Geometric Neighbors The obvious idea to find viable ω from Eq. 3.104 is to directly look at neighboring grid points and check if the dominant class changes between the two. When the dominant class changes between two neighboring grid points, to better localize the exact boundary of the class change, we add the grid point in between those two points. Because the individual grids of the classes may diverge during the course of our batch learning process, to simplify notation and computation, we define the union of the grids from all classes:

$$\mathcal{G}_{\text{total}} := \bigcup_{y \in \mathcal{K}} \mathcal{G}_y . \tag{3.105}$$

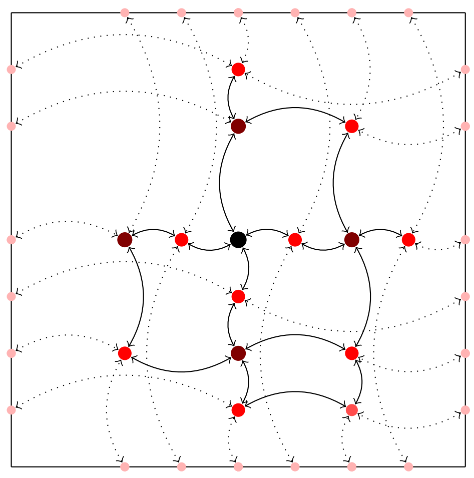


Figure 3.11.: Geometric neighbor relations in a two-dimensional spatially adaptive sparse grid. The 18 neighbor relations involving two grid points are marked with continuous arrows, whereas the 24 relations involving a grid point and a boundary projection point are marked with dotted arrows.

Every grid point has exactly two neighbors in each dimension. The geometric neighbor relations at the example of a two-dimensional sparse grid are visualized in Fig. 3.11. N being the number of grid points implies that the number of neighbor relations is in $\mathcal{O}(N \cdot d)$. If two grid points are neighbors of each other, one of those points is refinable in the direction of the other but not vice versa. The point that is refinable is called *leaf* in this direction. To iterate through all possible neighbor relations, we thus look at the leaf grid points (those are the ones that are refinable after all) and add their respective children in directions of dominant class changes. In Alg. 5, the iteration through the sparse grid and the collection of neighbor relation candidates is depicted.

Algorithm 5 Refinement for classification

Precondition: $d \in \mathbb{N}$ is the dimensionality
Precondition: $\mathcal{G}_{\text{total}} \subset \mathbb{G}_d$ is the sparse grid from Eq. 3.105
Precondition: $c_{\mathcal{M}}$ is the classifier
Precondition: `refinementCandidates`$[y]$ is the list of refinement candidates for class y

```
function InitializeNeighbors($d$)
    rootLevel ← $\{1\}^d$
    rootIndex ← $\{1\}^d$
    initialNeighbors ← $\varnothing$
    for $j$ ← 1 to $d$ do
        level ← copyOf(rootLevel)
        level[$j$] ← 0
        leftIndex ← copyOf(rootIndex)
        leftIndex[$j$] ← 0
        rightIndex ← copyOf(rootIndex)
        initialNeighbors$(j,$ left$)$ $\leftarrow$ (level, leftIndex)
        initialNeighbors$(j,$ right$)$ $\leftarrow$ (level, rightIndex)
    return initialNeighbors

function ADAPTNEIGHBORS(neighbors, $j$, newLevel, newIndex)
    newNeighbors $\leftarrow$ copyOf(neighbors)
    for all neighbor $\in$ newNeighbors do
        neighbor.level$[j]$ $\leftarrow$ newLevel
        neighbor.index$[j]$ $\leftarrow$ newIndex
    return newNeighbors

function PROCESSNEIGHBORS(leafPoint, neighborPoint, $j$, $y$)
    classLeaf $\leftarrow$ $c_{\mathcal{M}}$(coord(leafPoint))
    classNeighbor $\leftarrow$ $c_{\mathcal{M}}$(coord(neighborPoint))
    if classLeaf $\neq$ classNeighbor then
        score $\leftarrow$ SCORE(leafPoint, neighborPoint, classLeaf, classNeighbor)
        refinementCandidates[classLeaf].add(score, leafPoint, $j$, $y$)
        refinementCandidates[classNeighbor].add(score, leafPoint, $j$, $y$)
function STEPDOWN($d$, minDim, gridPoint, neighbors)
    for $j \leftarrow 1$ to $d$ do
        newLevel $\leftarrow$ gridPoint.level$[j]$ $+1$
        leftIndex $\leftarrow$ $2 \cdot$ gridPoint.index$[j] - 1$
        rightIndex $\leftarrow$ $2 \cdot$ gridPoint.index$[j] + 1$
        leftChild $\leftarrow$ copyOf(gridPoint)
        leftChild.level$[j]$ $\leftarrow$ newLevel
        leftChild.index$[j]$ $\leftarrow$ leftIndex
        if leftChild $\notin \mathcal{G}_{\text{total}}$ then
            PROCESSNEIGHBORS(gridPoint, neighbors$(j,$ left$)$, $j$, "left")
        else if $j \geq$ minDim then
            leftNeighbors $\leftarrow$ ADAPTNEIGHBORS(neighbors, $j$, newLevel, leftIndex)
            leftNeighbors$(j,$ right$))$ $\leftarrow$ gridPoint
            leftNeighbors$(j,$ left$)$ $\leftarrow$ neighbors$(j,$ left$)$
            STEPDOWN($d$, $j$, leftChild, leftNeighbors)
        rightChild $\leftarrow$ copyOf(gridPoint)
        rightChild.level$[j]$ $\leftarrow$ newLevel
        rightChild.index$[j]$ $\leftarrow$ rightIndex
        if rightChild $\notin \mathcal{G}_{\text{total}}$ then
            PROCESSNEIGHBORS(gridPoint, neighbors$(j,$ right$)$, $j$, "right")
        else if $j \geq$ minDim then
            rightNeighbors $\leftarrow$ ADAPTNEIGHBORS(neighbors, $j$, newLevel, rightIndex)
            rightNeighbors$(j,$ left$))$ $\leftarrow$ gridPoint
            rightNeighbors$(j,$ right$)$ $\leftarrow$ neighbors$(j,$ right$)$
            STEPDOWN($d$, $j$, rightChild, rightNeighbors)
```

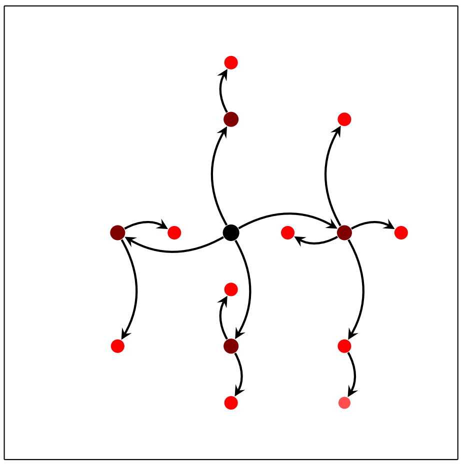


Figure 3.12.: Iteration through a two-dimensional spatially adaptive sparse grid. Every grid point is reached via exactly one path starting with `root`$_2$.

```
function SCOREALLNEIGHBORRELATIONS
    initialNeighbors ← INITIALIZENEIGHBORS(d)
    rootPoint ← $\left(\{1\}^d, \{1\}^d\right)$
    STEPDOWN(d, 1, rootPoint, initialNeighbors)
```

The algorithm is started by calling SCOREALLNEIGHBORRELATIONS(). First, the neighbors of the root point are initialized. Those are all virtual grid points, which represent projections of the root point to the bordering hyperplanes of Ω. We obtain those by calling INITIALIZENEIGHBORS(d). This function sets up a list of left and right virtual grid points in each dimension, identified by said dimension and direction. Essentially, this corresponds to $\texttt{boundary-projections}_{\{\texttt{root}_d\}}$. Then, COLLECTALLNEIGHBORRELATIONS() continues with calling STEPDOWN() on $\texttt{root}_d$.

In STEPDOWN(), we iterate once through the sparse grid by recursively visiting the children of the current grid point. To make sure that we visit every grid point only once, `minDim` is used to specify in which dimensions we are allowed to step down from a specific grid point. For example, in a two-dimensional sparse grid of level 3, the grid point $p_1 = ((2,2),(1,1))$ has two parents, $p_2 = ((1,2),(1,1))$ and $p_3 = ((2,1),(1,1))$. Thus, p_1 could be potentially reached by both p_2 and p_3, but our algorithm ensures that it is reached only through p_3. Through which parents all grid points are recursively reached is depicted in Fig. 3.12 at the example of a spatially adapted sparse grid. For each possible child of the current grid point p, we check in every dimension j (with $j \geq$ `minDim`) if the left and right child exist. If a child exists and $j \geq$ `minDim` (which means that we are allowed to step down into that dimension), the neighbors are adapted and we step down to this child to continue the recursion. Otherwise, p is a leaf in that dimension and direction, so p and its neighbor in the dimension and direction are processed by calling PROCESSNEIGHBORS() on them.

The routine ProcessNeighbors() evaluates the classifier at both the leaf grid point and its geometric neighbor. If the dominant class at those two points is not the same, the score for adding the child (the refinement candidate) in between those two grid points is calculated by calling Score() for the two grid points. Then, the refinement candidate is added to the list of candidates for both the dominant class at the leaf grid point and the dominant class at its neighbor. Because Alg. 5 operates on $\mathcal{G}_{\text{total}}$, it is possible that the leaf grid point and/or its neighbor are not already present in the grid of one or both of the two classes. This is why we need to make certain, that all hierarchical parents are added to the grid when we decide to add the point in between the leaf and its neighbor.

Scoring For the score, we look at the values of the class densities (weighted with the respective priors) $f_y(\boldsymbol{x}) \cdot |\mathcal{M}_y|$. p being the leaf grid point (with level $\boldsymbol{l}$) and n its neighbor, $d_p = c_{\mathcal{M}}(p)$ is the dominant class at p and $d_n = c_{\mathcal{M}}(n)$ is the dominant class at n. The score for the neighbor relation between p and n is then given by

$$\texttt{score}(p, n) = \frac{\left| \left(f_{d_p}(p) \cdot \left|\mathcal{M}_{d_p}\right| - f_{d_n}(p) \cdot |\mathcal{M}_{d_n}| \right) - \left(f_{d_p}(n) \cdot \left|\mathcal{M}_{d_p}\right| - f_{d_n}(n) \cdot |\mathcal{M}_{d_n}| \right) \right|}{2^{\|\boldsymbol{l}\|_1}} . \tag{3.106}$$

The numerator in Eq. 3.106 indicates, how drastic the change of classes between the two neighbors is. First, it determines the difference of the densities of d_p and d_n at both points. Then, it calculates the absolute value of the difference of those two differences. If this quantity is high, it means that the change between the two class densities is steep, thus a new grid point in between the neighbors is desireable. To prevent overfitting, we divide with the L1-norm of the level of p, which is proportional to the size of the support of p.

Refining the Grids After scoring all neighbor relations, we add the point between the two neighbors into the respective classes d_p and d_n. Because we operate on $\mathcal{G}_{\text{total}}$ for all classes, p or even coarser ancestors of the new point might be missing in d_p or d_n. This is why we need to make sure to not only add the new point, but also all of its ancestors to the grids as depicted in Alg. 6. There are two things we do differently in the classification refinement process in contrast to the traditional refinement in other sparse grids settings. Firstly, the grid points we refine do not necessarily have to be in the grid. Technically, p is the point that we refine. p stems from $\mathcal{G}_{\text{total}}$ and might thus originate from another class than the one we currently target. Thus, the grid of the class we refine it in might not contain it prior to that. This is why we need to ensure that p and all of its ancestors are also added to the grid during refinement. Second, when refining p, we take into account the direction to the neighbor. In the traditional setting, all $2d$ children of a refinement candidate (plus all of their respective ancestors) are added to the grid. With growing level, this can sum up to a lot more than just $2d$ new grid points. In our setting, we only add the grid point in-between the refinement

Algorithm 6 Refining all grids of the classifier

Precondition: $\mathcal{G}_y$ is the sparse grid for class y
Precondition: `refinementCandidates`$[y]$ is the list of refinement candidates for class y

```
function RefineAllClasses
    for all classes $y$ do
        refinementCandidates[$y$].sortByScore()
        for all top-scored candidates struct in refinementCandidates[$y$] do
            (score, leafPoint, $j$, $y$) $\leftarrow$ struct
            newPoint $\leftarrow$ copyOf(leafPoint)
            newPoint.level[$j$] $\leftarrow$ newPoint.level[$j$] $+ 1$
            if $y ==$ "left" then
                newPoint.index[$j$] $\leftarrow 2 \cdot$ newPoint.$index[j] - 1$
            else
                newPoint.index[$j$] $\leftarrow 2 \cdot$ newPoint.$index[j] + 1$
            $\mathcal{G}_y \leftarrow \mathcal{G}_y \cup$ ancestors$^*$(newPoint)
```

candidate (the leaf point) and its geometric neighbor, thus reducing the number of new points we add for one refinement candidate from $2d$ to one.

Runtime Complexity Analysis In Alg. 5, every grid point of $\mathcal{G}_{\text{total}}$ is visited once during the recursion of StepDown(). With $N_{\text{total}} := |\mathcal{G}_{\text{total}}|$, the complexity of the pure iteration through all grid points is thus $\mathcal{O}(N_{\text{total}})$. After the collection of all neighboring relations, all class densities are evaluated at every point in $\mathcal{G}_{\text{total}}$ and all of the boundary points that are related to a grid point because of the neighboring relationships. The number of those evaluation points is in $\mathcal{O}(d \cdot N_{\text{total}})$. If l_{max} is the maximum L1-norm of all points' levels in $\mathcal{G}_{\text{total}}$, each evaluation is in $\mathcal{O}(l_{\text{max}})$. With k denoting the number of classes, the total complexity of evaluating all points for all classes is $\mathcal{O}(k \cdot d \cdot N_{\text{total}} \cdot l_{\text{max}})$. After that, determining the dominant class for all points is in $\mathcal{O}(k \cdot N_{\text{total}})$ and scoring all relations is in $\mathcal{O}(d \cdot N_{\text{total}})$. Finally, inserting the new points into the grids is in $\mathcal{O}(d \cdot N_{\text{total}})$, if we choose to add a certain percentage of the refinement candidates to the grid and $\mathcal{O}(1)$ if we choose a fixed number of new points. In total, the whole refinement process is dominated by the evaluation of all class densities at all points, which is in $\mathcal{O}(k \cdot d \cdot N_{\text{total}} \cdot l_{\text{max}})$.

Evaluation We first compare the classification refinement to the surplus-based refinement by learning a classifier on the two-dimensional artificial two-moons dataset. Then, we show how the classification accuracy behaves when employing incremental batch-wise learning on the SDSS DR10 dataset [1] for both refinement indicators.

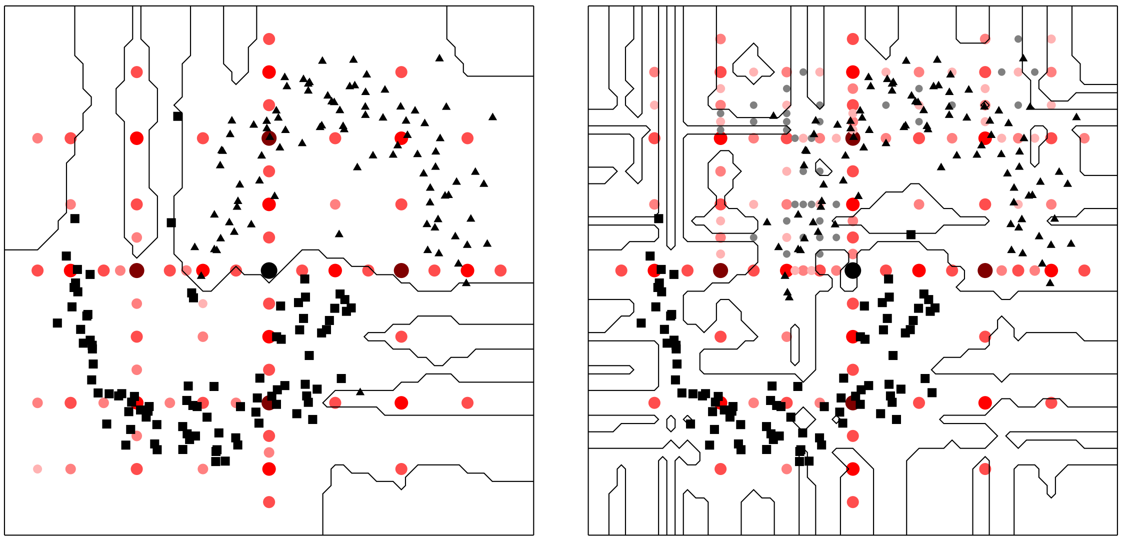

(a) Using the novel classification refinement. (b) Using the legacy surplus-based refinement.

Figure 3.13.: Classification with spatial adaptivity (refinement) of the two-moons dataset. The continuous lines denote the classification boundaries between the two classes. In circles (red), the grid points are visualized. In case of the surplus-based refinement, the grids of the two classes diverge which is why we choose to display the grid of the upper class. The data points are visualized as black squares and triangles for the two classes respectively.

Two-Moons Dataset We evaluate the classification results with the artificial two-moons dataset consisting of 180 samples with refinement enabled. We start with a regular sparse grid of level 4 with kinked linear basis functions, learn the density functions incrementally with batch size 30 and employ spatial adaptivity in-between by refining five points after processing each batch. The final grid together with the classification of the test data is visualized in Fig. 3.13. The classification borders are marked as lines, the grid points in red circles and the data points separated by class as triangles and squares. Fig. 3.13a shows the results for the classification refinement we proposed and Fig. 3.13b shows the results for the surplus-based refinement. For classification refinement, the grid points converge towards the classification borders which is the behaviour we were looking for. Also, the number of additional grid points is low, because we only add the hierarchical children in the directions towards the class boundary. On the other hand, many unnecessary grid points are added with the surplus-based refinement indicator situated in areas that do not contribute to the classification accuracy. Therefore, achieving the same accuracy with both methods is much cheaper with the classification refinement. Not only do we end up with less grid points, we also need less refinement iterations to detect the relevant new grid points and the time spent to update the models is also drastically reduced.

Sloan Digital Sky Survey (SDSS) We use the data release 10 from the Sloan digital sky survey dataset [1] to investigate how the classification accuracy behaves when employing refinement with both the classification refinement we proposed and the

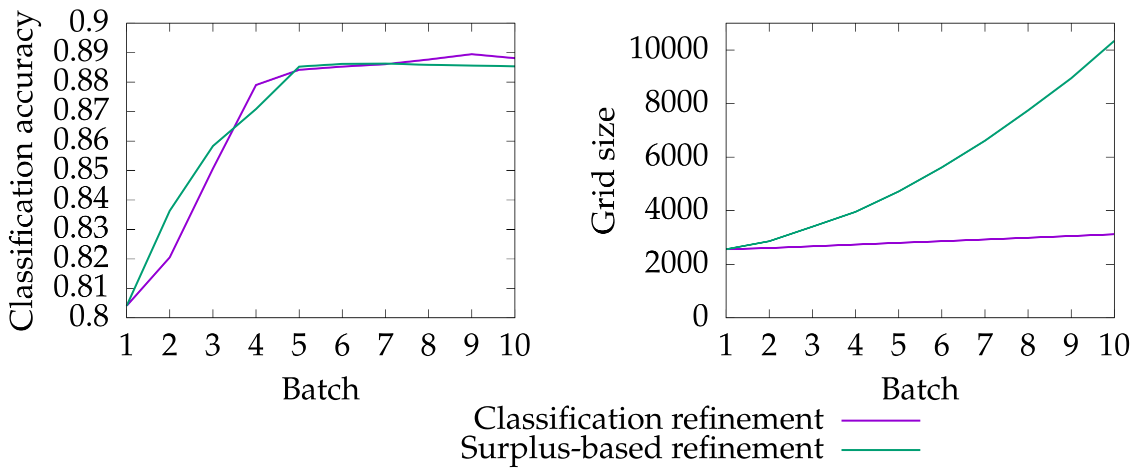


Figure 3.14.: Incremental learning the SDSS DR10 dataset with both classification refinement and surplus-based refinement. On the left, the classification accuracy for both refinement strategies is displayed. There is small difference between the two methods. However, when taking the grid sizes into account, we observe that the surplus-based refinement adds much more grid points than the classification based refinement, resulting in longer runtimes.
Raw data for this figure: Tab. C.11.

surplus-based refinement. The results are shown in Fig. 3.14. This four-dimensional dataset consists of two classes. We learn 10 batches of 50,000 samples each and configure a regularization value of 10^{-3}. With both methods, we choose to refine 50 grid points after each batch. Not counting hierarchical ancestors, this leads to 200 child points per class-grid being added with surplus refinement in general, whereas it only leads to 50 child points per class-grid being added for the classification refinement. While the accuracy of the model is approximately the same after each batch, the resulting grid sizes show that the surplus refinement is much more costly if we want to achieve similar accuracies compared to the classification refinement. Those costs show when updating the matrix decompositions but also when computing the right-hand side of the linear system. In total, we conclude that the classification refinement is to be preferred when performing classification.

4. Parallel Learning

With parallel hardware at hand, we speed up the learning process by distributing the tasks over the node. Parallelization of sparse grid methods for data mining has been investigated in various settings [30–32, 44–46], but the incremental learning scheme together with the offline/online splitting procedures require a targeted analysis of parallelization potentials.

In this chapter, we discuss aspects concerning parallelism during the training process. On cluster-systems with a distributed memory architecture, learning is parallelized using different schemes, which divide the task into independent packages. Both data parallelism and model parallelism are considered. Also, we see how to apply and exploit the sparse grid combination technique.

In essence, the parallelization of the density estimation and the classification based on density estimation is similar. We mainly discuss the classification case, as it is the more general one. For a parallelization on a pure density estimation, the transformation is easily done by assuming a problem with only one class.

4.1. Overview

Several starting points for parallelizing the learning process exist. A schematics of the possibilities we discuss here is depicted in Fig. 4.1. On the highest level, training the classifier is separated into computing the density estimation of batch k for all classes y. Just parallelizing over the classes is generally not a balanced scheme, because the number of training data points differs between the classes. Parallelizing solely over the batches is a valid approach, but needs the models of all classes present at all distributed compute units. The scheme we discuss in Sec. 4.2 is to combine both approaches: parallelizing over the classes and batches at the same time. Thereby, we distribute the computation of $\boldsymbol{\alpha}_{y,k}$, which is the solution of Eq. 3.11 for class y and batch k, over the parallel nodes.

The density estimation for each class relies on a sparse grid. We investigate how the sparse grid combination technique can be applied instead of regular sparse grids. This yields computations speedups in both sequential and parallel executions. As the combination technique is only well-defined for regular sparse grids and dimensional adaptivity, we do not take into account problems that require spatial adaptivity at this level. In Sec. 4.3, we also see that not only the online phase but also the offline phase

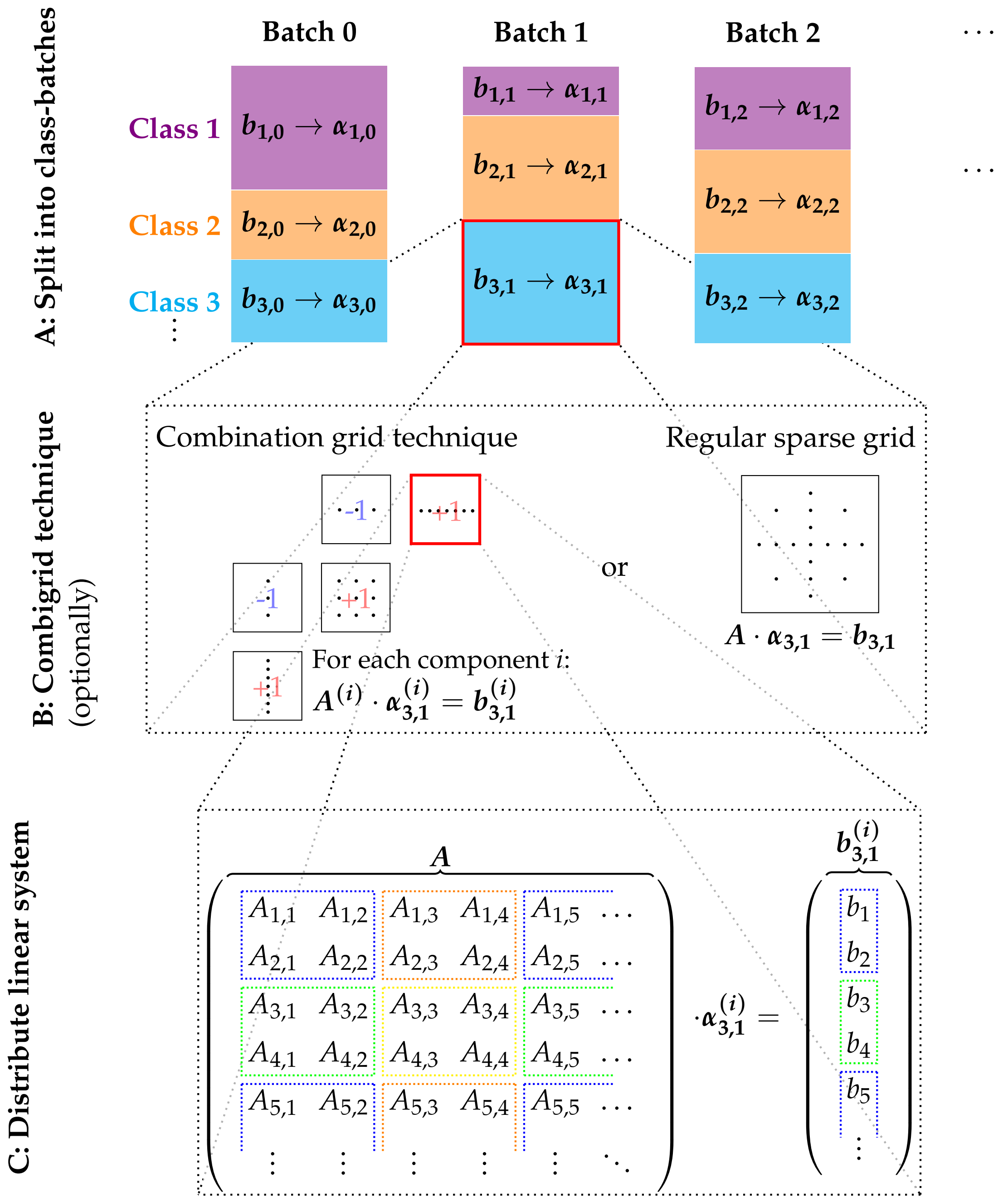

A: Split into class-batches
Batch 0
Batch 1
Batch 2
...
Class 1
Class 2
Class 3
$b_{1,0} \to \alpha_{1,0}$
$b_{2,0} \to \alpha_{2,0}$
$b_{3,0} \to \alpha_{3,0}$
$b_{1,1} \to \alpha_{1,1}$
$b_{2,1} \to \alpha_{2,1}$
$b_{3,1} \to \alpha_{3,1}$
$b_{1,2} \to \alpha_{1,2}$
$b_{2,2} \to \alpha_{2,2}$
$b_{3,2} \to \alpha_{3,2}$
B: Combigrid technique (optionally)
Combination grid technique
Regular sparse grid
-1
+1
-1
+1
+1
or
For each component i: $A^{(i)} \cdot \alpha_{3,1}^{(i)} = b_{3,1}^{(i)}$
$A \cdot \alpha_{3,1} = b_{3,1}$
C: Distribute linear system
A
$A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{1,4}$ $A_{1,5}$ ...
$A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{2,4}$ $A_{2,5}$ ...
$A_{3,1}$ $A_{3,2}$ $A_{3,3}$ $A_{3,4}$ $A_{3,5}$ ...
$A_{4,1}$ $A_{4,2}$ $A_{4,3}$ $A_{4,4}$ $A_{4,5}$ ...
$A_{5,1}$ $A_{5,2}$ $A_{5,3}$ $A_{5,4}$ $A_{5,5}$ ...
$\cdot \alpha_{3,1}^{(i)} =$
$b_{3,1}^{(i)}$
b_1 b_2 b_3 b_4 b_5

Figure 4.1.: Parallel batch learning scheme for density estimation and classification. On the first level, the data is split into class batches. For each such class-batch, the density is estimated. On the second level, this estimation is optionally parallelized with the combination grid technique. Solving the corresponding linear system is then distributed on the third level.

of the algorithm profits a lot from the partial computation of the solution on smaller, regular grids.

Depending on the matrix decomposition method used (ref. Sec. 3.2.3.3), the computation of $\boldsymbol{\alpha}_{y,k}$ breaks down to computing huge matrix vector products. Those multiplications are further parallelized by distributing the matrix over multiple nodes. At this level, we use the ScaLAPACK library [12] to transparently achieve parallel linear algebra operations and discuss the outcomes in Sec. 4.4.

4.2. Batch Parallelization

The number of grid points in class y is denoted by N_y and the total model size amounts to $\sum_y N_y$. The runtime of training our density estimation based classifier with k classes, where M is the dataset size, is then in $\mathcal{O}\left(M \cdot \sum_y N_y^2\right)$. Thus, the runtime grows linearly with the dataset size and quadratically with the model size. To learn as fast as possible, we want to parallelize this training process. The concept we discuss in the following is a *data parallel* scheme. First, we differentiate the roles and tasks of the master node and the worker nodes in Sec. 4.2.1. Then, those tasks are discussed in detail in Sec. 4.2.2 with a special focus on communication. Next, we turn to the scheduling of those tasks in Sec. 4.2.3 before evaluating the batch parallelization in both strong and weak scaling settings in Sec. 4.2.4.

4.2.1. Parallel Setup

In the data parallel scheme, we employ a series of *worker nodes* to solve Eq. 3.11 separately for the class data batches and send their partial solutions to the *master node* who is also responsible of performing model adaptions during the learning process and handling the work scheduling. The schematics are visualized in Fig. 4.2.

Master Node The master node is the central process of the parallel scheme. Its tasks are:

1. Preparing the models
2. Holding available the current solution of the classifier (or the density estimation) for evaluation.
3. Distributing the class batch learning to all available worker nodes.
4. Triggering model adaption concurrent to the learning process.
5. Performing/Distributing model updates based on the model adaptations.

If the master is idling, it assigns work to itself and treats itself as a worker node.

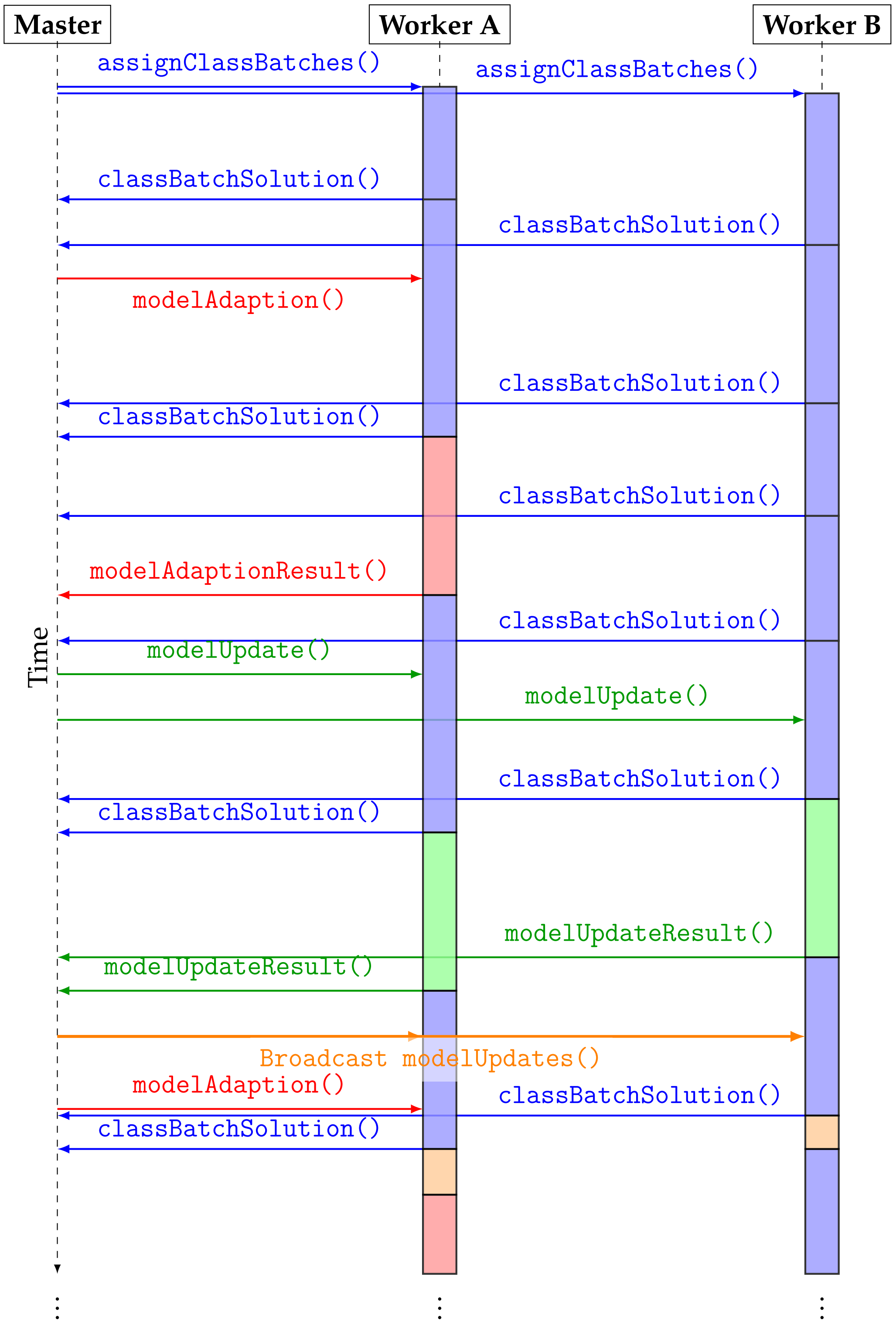


Figure 4.2.: Parallel batch learning sequence diagram with two worker nodes. In blue, the communication and learning of class batches is depicted. The model adaption communication and computations are colored in red, the model update communication and computations in green. The broadcast and subsequent merge of the model updates to the workers is colored in orange.

Worker Node A worker receives work packages from the master and sends back the results. Work packages are:

1. Obtain the solution of class batch (k, y) by computing $\boldsymbol{\alpha}_{y,k}$.
2. Execute model adaptations.
3. Update the model of class y based on model adaptions.

4.2.2. Tasks

The tasks listed in Sec. 4.2.1 are discussed regarding what communication takes place and how the task is executed.

Master Node Workflow The master node is responsible of splitting all data batches into the class batches and distributing those onto the worker nodes. As soon as a worker node finishes the processing of one class batch, the master assigns it another one. Concurrently to the computation of the solution of class batches, the master also triggers the model adaption process as soon as at least one class batch has been processed and its solution is incorporated into the model. As soon as one model adaption step is completed, the next one is triggered immediately afterwards (if in the meantime, more solutions of class batches have been computed). Also, after each model adaption step, the master assigns the model update work packages to worker nodes and subsequently handles the communication of the results to all nodes.

Preparing the Models During the initialization phase, the master process sets up the models. It loads the decomposition of the system matrix of Eq. 3.11 dependent on the dimensionality, the configured initial grid level and the chosen decomposition method. Then, it broadcasts the initial empty classification model (the initial grid and the corresponding system matrix decomposition) to the worker nodes. Note that the initial grids of all classes are identical. Thus, only one grid and decomposition has to be broadcasted and then replicated internally by the workers for each class.

Computing the Solution of a Class Batch The batch learning process introduced in Sec. 3.2.1 provides an inherent way to split the problem into work packages. For each data batch, assembling the right-hand side of the linear system via Eq. 3.18 is independent of other batches. Furthermore, it is possible to adapt the process of how and when to integrate the results of the different batches. In Eq. 3.17, the right-hand side vectors of all batches are agglomerated into an overall right-hand side, which then serves to solve Eq. 3.11 for $\boldsymbol{\alpha}$. We change this process now and first solve Eq. 3.11 for $\boldsymbol{\alpha}$ for each batch separately and then agglomerate the resulting solutions similar to Eq. 3.25.

$\widetilde{\boldsymbol{b}_k}$ still denotes the right-hand side of Eq. 3.11 for batch k. The solution of Eq. 3.11 for batch k is denoted by $\widetilde{\boldsymbol{\alpha}_k}$ and is obtained by solving

$$A\widetilde{\boldsymbol{\alpha}_k} = \widetilde{\boldsymbol{b}_k}\,. \tag{4.1}$$

To obtain the solution of Eq. 3.11 up to batch k, we obtain $\boldsymbol{\alpha}_k$ as

$$\begin{aligned} \alpha_{k_i} := \min\left((1-\beta)^{|m_{k,i}|}, \frac{|\mathcal{M}_{k-1,i}|}{|\mathcal{M}_{k,i}|}\right)\alpha_{k-1_i} \\ + \max\left(1-(1-\beta)^{|m_{k,i}|}, \frac{|m_{k,i}|}{|\mathcal{M}_{k,i}|}\right)\widetilde{\alpha_{k_i}}\,. \end{aligned} \tag{4.2}$$

With this scheme, we can distribute the dataset to an arbitrary number of workers.

Communication All workers hold A or the discussed decompositions. The master sends the data points corresponding of class batch (k, y) to the worker, who then computes $\widetilde{\boldsymbol{b}_k}$ for class y. Upon solving the system Eq. 4.2, the worker sends its partial solution $\widetilde{\boldsymbol{\alpha}_k}$ back to the master process who then computes the overall $\boldsymbol{\alpha}_k$ with Eq. 4.2.

Adapting the Model The master is also responsible for triggering the adaptation of the model. This is done in parallel to the computation of the class batches. In case of a pure density estimation setting, the state of the model is represented by the current grid $\mathcal{G}_r$ and corresponding $\boldsymbol{\alpha}_k$. The adapted grid $\mathcal{G}_{r+1}$ is then obtained via refinement and coarsening as discussed in Sec. 3.2.2. In case of density based classification, the state of the model is represented by the current class grids $\mathcal{G}_{y,r}$ and the corresponding $\boldsymbol{\alpha}_{y,k}$. There, the adapted grids $\mathcal{G}_{y,r+1}$ are obtained via refinement and coarsening as discussed in Sec. 3.3.2.

Communication All grids $\mathcal{G}_{y,r}$ and all surplus vectors $\boldsymbol{\alpha}_{y,k}$ need to be sent from the master to the worker node, which executes the model adaptation before the process is started. After the execution, the worker sends $\mathcal{G}_{y,r+1}$ back to the master for all classes y. Here, depending on the number of grid points added and removed, it might be faster to only communicate the indices of grid points that are added and removed from the respective class grids. Also, the master might choose to execute the model adaption itself, in which case no communication is required at this stage. Note however that after adapting the model, the data structures of the model has to be updated, which again involves communication.

Model Consistency Due to concurrent computation of class batch solutions and model adaptations, it might happen that a worker sends some $\widetilde{\boldsymbol{\alpha}_{y,k'}}$ as a result from computing a class batch to the master that refers to $\mathcal{G}_{y,r'}$, whereas the master already holds $\mathcal{G}_{y,r}$ with $r > r'$. Thus, there might both be points missing from $\mathcal{G}_{y,r}$ that have previously been in $\mathcal{G}_{y,r'}$ and new points in $\mathcal{G}_{y,r}$ that were not included in $\mathcal{G}_{y,r'}$. In the first case, the corresponding surplus values in $\widetilde{\boldsymbol{\alpha}_{y,k'}}$ are silently dropped. In the second case, $\widetilde{\boldsymbol{\alpha}_{y,k'}}$ is extended by as many corresponding entries as needed, which are all set to 0. Then, $\widetilde{\boldsymbol{\alpha}_{y,k'}}$ is safely combined with $\boldsymbol{\alpha}_{y,k-1}$ to form $\boldsymbol{\alpha}_{y,k}$.

Updating the Model after Model Adaption As soon as the new grids $\mathcal{G}_{y,r+1}$ are compiled, the corresponding decompositions are updated (ref. Sec. 3.2.3.3). This is done independently for each class, thus a work package for a worker node is to update the model of a specific class.

Communication The master sends the new class grid $\mathcal{G}_{y,r+1}$ to the worker. After completion of the model update, the worker sends back the whole decomposition back to the master. Afterwards, the master broadcasts this new decomposition to all workers together with $\mathcal{G}_{y,r+1}$, such that all worker nodes compute future solutions of batches for class y.

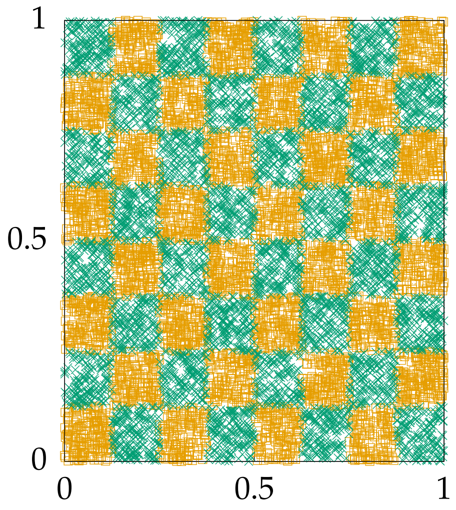
4.2.3. Scheduling

Due to the concurrent procedure of training class batches, adapting the class models, and updating the decompositions, we need to think about the priority of working off tasks if multiple work packages wait to be completed. Thereby, it is important to balance the progression of the training through $\mathcal{M}_{\texttt{train}}$ and the adaption of the class models to the data. To that end, the master schedules the work packages for the workers such that priority packages are sent out first. Also, the workers schedule the work assigned to them by first processing priority packages, even if other packages have been waiting longer.

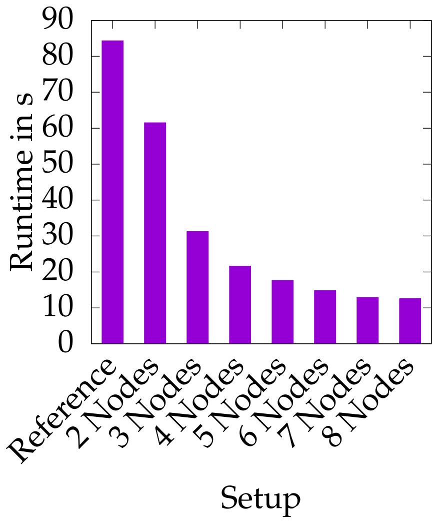
The computation of solutions for class batches is most valuable on models adapted to the data. In order to complete a model adaption phase, the model update packages have to be processed. This is why model update work packages are of highest priority as soon as they are packed and they get scheduled before all other work packages.

Next, the scheduler needs to ensure that a new model adaption process is started as soon as the previous one finished, because out of the tasks presented in Sec. 4.2.2, this is the most compute intensive one. Also, a model adaption work package cannot be distributed to multiple workers as this task is not parallelized in itself so far.

Finally, the scheduler lets all class batches of data batch k be processed before any class batches of data batch $k+1$. This way, the model stays in sync as well as possible over all classes and the model adaption process is able to operate on a model state as consistently as possible.



(a) Two-dimensional checkerboard dataset.



(b) Strong scaling results for the parallel batch learning scheme.

Figure 4.3.: Strong scaling test for the parallel batch learning setting is executed with the checkerboard dataset (Subfig. a). 50 batches of 300,000 data points each are trained on up to eight nodes. The runtimes visualized in Subfig. b show the strong scaling behaviour of the scheme. Adapted from [13].
Raw data for this figure: Tab. C.12.

4.2.4. Evaluation

We are interested in the speedup, the presented class batch parallelization scheme offers. The parallel batch learning scheme was implemented into SG++ in a bachelor project [13], where the strong scaling and the weak scaling was investigated. The tests were executed on the LRZ Linux Cluster (ref. Sec. B.2).

Strong Scaling For the strong scaling test, we choose the artificial checkerboard dataset [56] (visualized in Fig. 4.3a), which allows us to generate as many samples as we like. We generate 50 batches of 300,000 data points each. In the first run, we do not trigger model adaptions. The results are shown in Fig. 4.3b. The figure shows a good strong scaling behaviour. This is to be expected due to the minimal and asynchronous communication required in this case. With the data batches distributed to the workers, all they need to do is send their results back to the master who then computes the total solution.

When refinement is enabled, the performance of the parallel learner depends a lot on the refinement settings. For large grids and many refinements, the time spent for refining grows (ref. Sec. 3.2.3.3.5). Since the refinement procedures themselves are not parallelized, a large dataset or iterating over many epochs is required to balance out the relative long runtime of the model adaption and update procedures. More research into the parallelization of the model adaption and model update packages is necessary

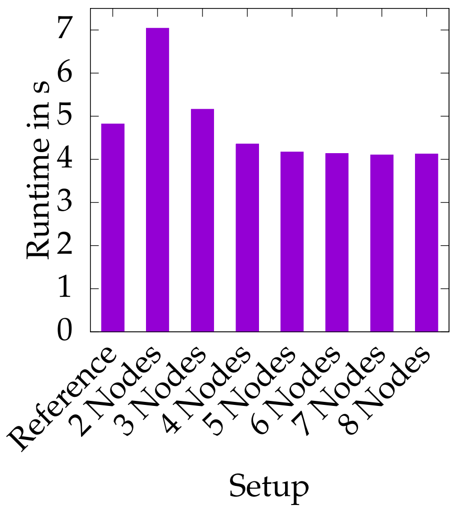


Figure 4.4.: Weak scaling test for the parallel batch learning setting is executed with the checkerboard dataset (Fig. 4.3a). For the 2 node setup, the efficiency is not ideal due to the master not acting as a worker too is this case. Adapted from [13].
Raw data for this figure: Tab. C.12.

to obtain good strong scaling behaviour with refinement enabled too. Therefore, we leave the detailed investigation of this case to the next generation of researchers.

Weak Scaling Keeping the number of data points fixed for each node, we run the test again. Ideally, the time to solution is almost identical independent of the number of nodes involved. The results for the weak scaling test are shown in Fig. 4.4. In this setting, the master node is not working off tasks as well. However, weak scaling requires us to choose the dataset size dependent on the number of nodes involved. Thus, in the test with two nodes, only one worker is tackling the packages meant for two. Extending the setup to even more nodes, the runtime approaches a constant value. This shows that the parallel learning scales well for a sufficient number of computing nodes.

4.3. Employing the Combination Grid Technique

In Sec. 2.3.2, the combination grid technique was presented. It has been applied to data-driven problems before, see [27]. In general, the combination technique solution and the sparse grid solution are not the same, but the accuracy is of the same order.

We want to use this technique to speed up the density estimation in several ways. The first and obvious possibility to exploit the combination technique is to parallelize the computation of the solutions over the involved component grids. This introduces another level of parallelism, additional to the one presented in Sec. 4.2.1. The effort to implement this scheme into the SG++ data mining pipeline (ref. Chap. 6) has been undertaken during a bachelor project [82]. The second possibility to exploit the combination technique, which also enfolds its potential when approximating the probability density estimation in a non-parallel setting, stems out of the cubic complexity of the system matrix factorization and the resulting quadratic complexity of solving Eq. 3.11 for $\boldsymbol{\alpha}$.

The first approach has been investigated many times in similar settings (e.g. [28, 38, 58]). With the computation of the partial solutions of the components being independent of each other, distributing and parallelizing those partial solutions is straightforward. This scheme is also applicable to parallelize the computation of the solution of one class batch. In addition to the parallelization of the components, we investigate the computational gain that results from the mere split of the model from a regular sparse grid to multiple components grids. This computational gain also occurs if the components are processed sequentially, as we see in Sec. 4.3.1. An evaluation of the runtimes of processing a regular sparse grid vs. an ensemble of component grids stemming from the combination grid technique is given in Sec. 4.3.2.

4.3.1. Speeding Up the Offline/Online Scheme

For a regular sparse grid of dimension d and level l (resulting in $N_{d,l}^{\text{regular}}$ grid points), we obtain a system matrix of size $N_{d,l}^{\text{regular}} \times N_{d,l}^{\text{regular}}$. It needs to be factorized in $\mathcal{O}\left(N_{d,l}^{\text{regular}^3}\right)$ during the offline phase so that the online phase runs in $\mathcal{O}\left(N_{d,l}^{\text{regular}^2}\right)$. When employing the combination technique, we obtain a set of smaller system matrices. The number of components $\texttt{numComp}_{d,l}$ involved in the combination technique of dimension d and level l is given by

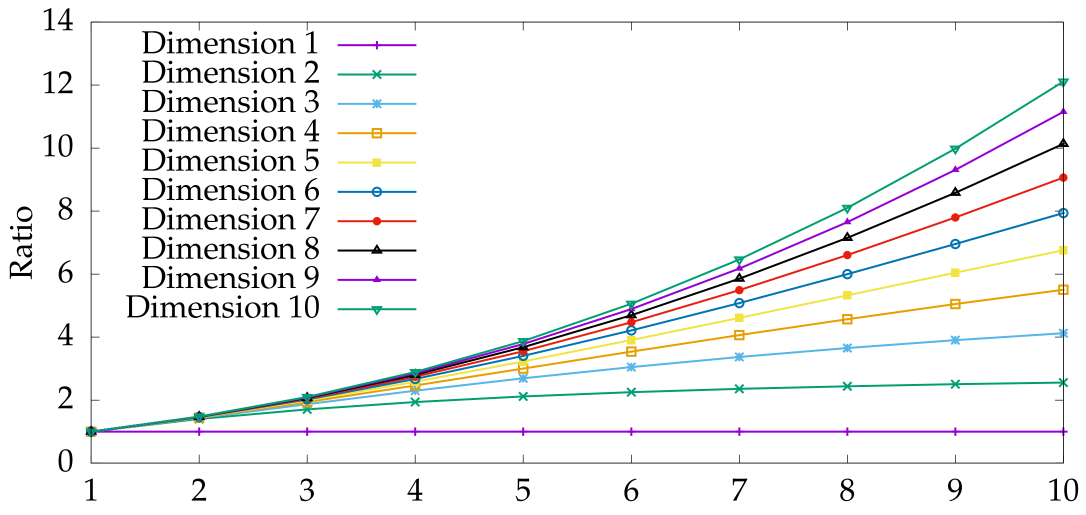
$$\texttt{numComp}_{d,l} = \sum_{q=0}^{d-1} \binom{l-q}{d-1} = \frac{(l+1)\binom{l}{d-1} + (d-l-1)\binom{l-d}{d-1}}{d} . \tag{4.3a}$$

The number of grid points in $\texttt{componentGrid}_{d,l}$ is given by

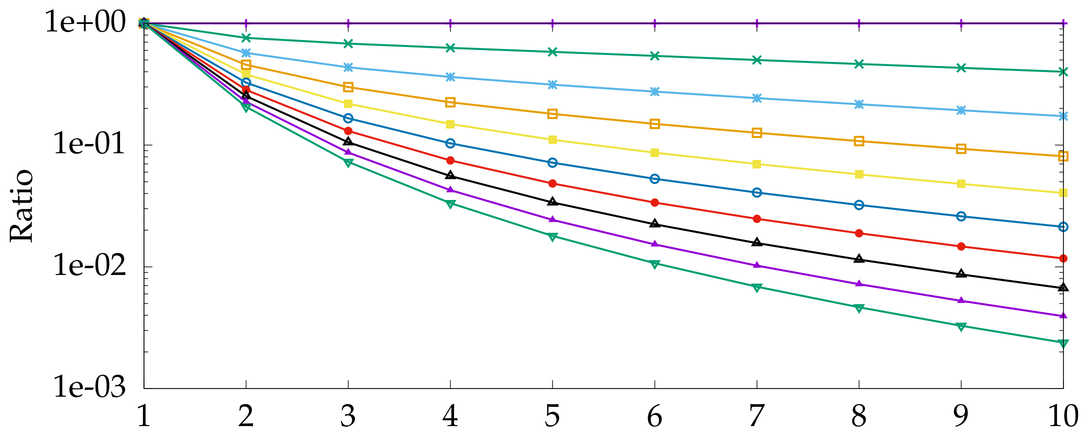
$$\texttt{numCompPoints}_{d,l} = \prod_{i=1}^{d} 2^{l_i} - 1 . \tag{4.3b}$$

We call the sum of the grid points of all involved components $N_{d,l}^{\text{combi}}$. $N_{d,l}^{\text{combi}}$ exceeds $N_{d,l}^{\text{regular}}$, because grid points of level $l' < l$ are contained in multiple components. The ratio of both quantities for dimensions 1 to 10 and level 1 to 10 is visualized in Fig. 4.5a. The ratio grows moderately with growing level and dimension.

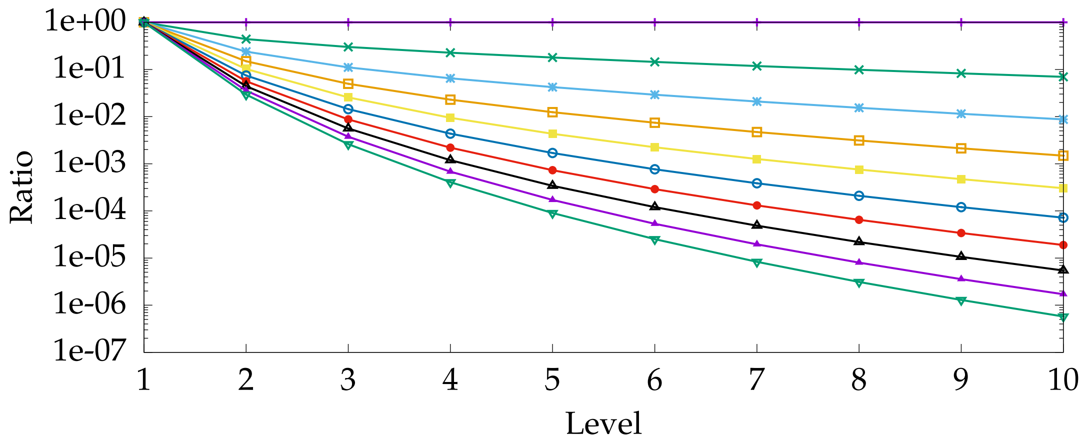
On the other hand, we want to compare the size of the system matrix for the regular sparse grid case (which is $N_{d,l}^{\text{regular}^2}$) with the combined sizes of the system matrices of



(a) Degrees of freedom in a regular sparse grid vs. the combination grid technique for dimension and level from one to ten.



(b) Cost of the online phase in density estimation for a regular sparse grid vs. the combination grid technique.



(c) Cost of the offline phase in density estimation for a regular sparse grid vs. the combination grid technique.

Figure 4.5 *(previous page)*: In Subfig. a, the ratio of $N_{d,l}^{\text{combi}}$ over $N_{d,l}^{\text{regular}}$ for dimensionalities and levels from 1 to 10 is depicted. It can be seen that the total number of grid points in the combination grid technique exceeds the number of grid points for the corresponding regular sparse grid. However, the ratio of the sum of the **squared** number of grid points of each component in the combination grid technique of dimension d and level l over ${N_{d,l}^{\text{regular}}}^2$ as depicted in Subfig. b on a logarithmic scale is in favor of the combination grid technique even for low dimensions or coarse levels. For high dimensions or fine levels, the resource gain in both runtime and memory during the online phase is noticeable even if working off the components sequentially. This becomes even more so for the ratio of the sum of the **cubed** number of grid points of each component in the combination grid technique of dimension d and level l over ${N_{d,l}^{\text{regular}}}^3$ as depicted in Subfig. c on a logarithmic scale. Even for lower dimensions or coarser levels, the resource gain when using the combination grid technique positively impacts runtime and memory consumption. For high dimensions or fine levels, the resource gain in both runtime and memory during the offline phase is several orders of magnitude, making the offline phase feasible to compute directly before the online phase if necessary.
Raw data for this figure: Sec. C.7.

the combination grid technique. The ratio of both quantities for the same dimension-level pairs is shown in Fig. 4.5b. This ratio is also a good approximation for the speedup during the online phase, if the combination grid technique is employed in contrast to the regular sparse grid. To complete the picture, we also show the ratio N^3 to the summed up cubic values for all involved components grids of the combination grid technique in Fig. 4.5c. Note, that both Fig. 4.5b and Fig. 4.5c are logarithmic on the y-axis. It is obvious that the runtimes of both the offline and the online phase as well as the memory consumed by the involved system matrices is in favor of the combination grid technique. This especially holds for the offline phase, where several orders of magnitude in speedup are possible. The sum of all components' runtimes is lower than the runtime for the regular sparse grid case, even if the components are worked off sequentially. Consequently, by employing the combination grid technique, we not only profit from the possibility to distribute the components on the parallel architecture at hand, but also from a sequential processing of the components.

4.3.2. Evaluation

We compare the time spent in the offline phase between employing a regular sparse grid and the combination grid technique for different dimensionalities and grid levels, resulting in the runtimes depicted in Fig. 4.6. The assumption that the time spent in the offline phase for the regular sparse grid is much longer than for the component grids from the combination grid technique combined proves true. However, depending on the dataset, the accuracy might be lower for the combination grid technique. Some datasets such as the chess dataset are well suited to be tackled with the combination

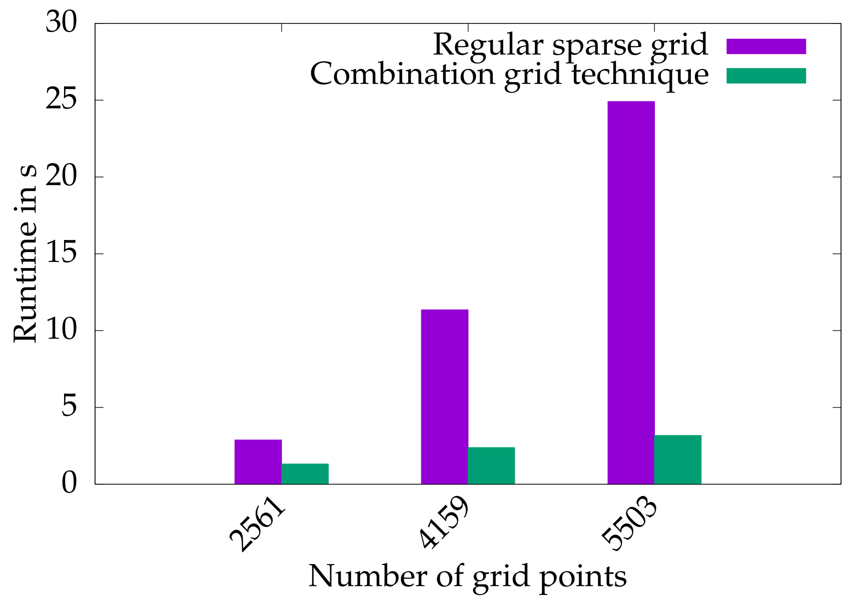


Figure 4.6.: Runtimes regular sparse grid vs. combination grid technique during the offline phase at the example of the Cholesky decomposition. With the combination grid technique, the matrices are factorized much faster than for the regular sparse grid. The tests were executed on the workstation (ref. Sec. B.1).
Raw data for this figure: Tab. C.16.

grid technique, while others don't achieve as good results as with the regular sparse grid technique as investigated in a bachelor project [82]. Also, the combination grid technique might profit from an optimized assembly of $\boldsymbol{b}$ (the right-hand side of Eq. 3.11). In the current setup, $\boldsymbol{b}$ is assembled once for each component. Since many grid points exist in multiple components, assembling $\boldsymbol{b}$ for all possible grid points once before extracting the relevant subset for each component will speed up the training phase further. Due to our focus on the offline/online scheme, we did not implement this concept of globally assembling $\boldsymbol{b}$ in this thesis.

4.4. Parallel Linear Algebra

For this section, we assume that we want to solve Eq. 3.11 for $\boldsymbol{\alpha}$, be it either in a pure density estimation setting or in the classification setting to obtain the solution for a class batch and either for a regular sparse grid or a component grid when employing the combination grid technique. In all cases, the computation of $\boldsymbol{\alpha}$ is parallelized for distributed memory architectures using parallel linear algebra routines. We choose ScaLAPACK [11] for this task, because it already provides a well-tested framework. The effort of implementing this scheme into the SG++ data mining pipeline (ref. Chap. 6) has been tackled during a bachelor project [86].

4.4.1. Matrix Decompositions

From the choices of matrix decompositions discussed in Sec. 3.2.3.3, we limit ourselves to the Cholesky decomposition (ref. Sec. 3.2.3.3.2) and the tridiagonal decomposition (ref. Sec. 3.2.3.3.4), because we want to test and use the parallel scheme in combination with spatial adaptivity.

Cholesky Decomposition In the former case, we target the parallelization of Eq. 3.44. The Cholesky factor $\boldsymbol{L}$ has to be distributed and we employ a parallel version of the forward and backward substitution scheme. This is achieved with ScaLAPACK.

Tridiagonal Decomposition In the latter case, we target the solution of Eq. 3.78. There, we find the classic case of a multiplication of a dense matrix to a vector (BLAS level 2). The matrices $\boldsymbol{Q}$, $\boldsymbol{T}$ and $\boldsymbol{B}$ need to be distributed in this case. Then, the parallel routines to perform dense matrix-vector multiplications (BLAS level 2) are employed.

4.4.2. Distribution of Data Structures in ScaLAPACK

ScaLAPACK forms a rectangular process grid with the available distributed nodes at hand. The process grid consists of p_r rows and p_c columns. We distribute either matrices or vectors over this process grid, whereas a vector behaves as a matrix with just one column. It is, thus, sufficient to illustrate the distribution of a general matrix $\boldsymbol{M} \in \mathbb{R}^{m_r \times m_c}$:

1. **Division into blocks:** $\boldsymbol{M}$ is divided into blocks of size $s_r \times s_c$, forming a block grid. If $\frac{m_r}{s_r} \notin \mathbb{N}$ or $\frac{m_c}{s_c} \notin \mathbb{N}$, this division also yields blocks of smaller size at the right or bottom border of the matrix.

2. **Assignment of blocks to processes:** The block grid is overlayed consecutively with the process grid, thereby assigning the block at position (b_r, b_c) to the process at $(b_r \mod p_r, b_c \mod p_c)$. This might result in the assignment of at most three more blocks to processes with smaller row-index or column-index than to other processes with higher indices. However, the resulting imbalance evens out for large matrices, if the block size is chosen such that each process is assigned a multiplicity of blocks.

3. **Storage of the blocks at the distributed nodes:** Each node stores the blocks assigned to it contiguously in memory. This enables the BLACS operations, which are applied to the data, to take advantage of the hierarchical memory architecture at the node, e.g. by always trying to avoid cache reloads and keeping the data stored in the vector registers and high-speed caches [11].

4.4.3. Tailoring ScaLAPACK

It is important to determine good configurations for the process grid layout as well as the block sizes. We investigate both in the context of the grid-based density estimation.

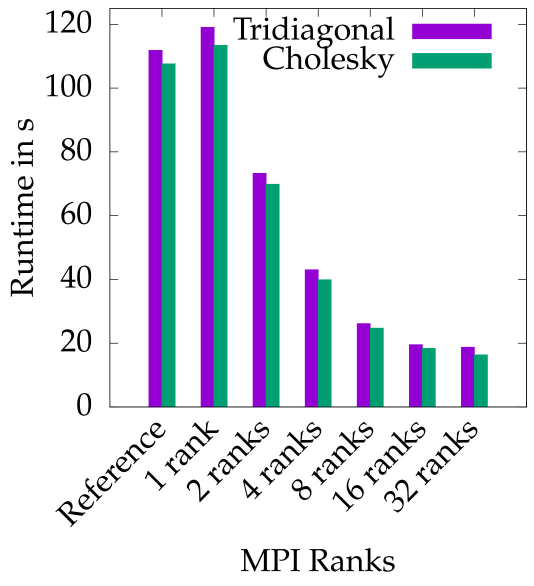
Process Grid Layout We want to distribute both matrices and vectors to the processes. Because a vector is only distributed to processes in the first column, we choose a process grid of size $p \times 1$, where p is the number of available processes. Validating this choice experimentally confirms this choice. Executing the splitting with different process grid layouts ranging from $p_c = 1$ to quadratically to $p_r = 1$ shows that the layout with $p_c = 1$ performs best for both Cholesky and tridiagonal decomposition.

Matrix Block Size The matrix block size should not be too large as to retain a well balanced distribution of the problem over the processes. Also, it should not be too small as this could negatively affect the efficiency of the computations as well as lead to a higher communication overhead. The results of the experiments we conducted for several choices suggest, that a good choice for the block size is 64×64.

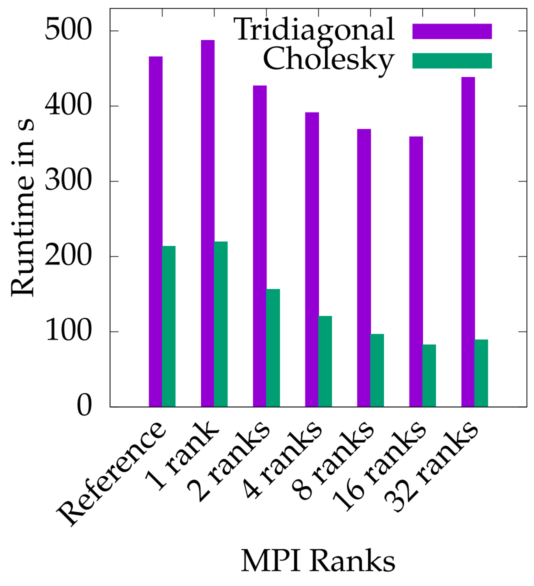
4.4.4. Evaluation

For evaluation, we are using the SDSS dataset [1], which was already used in Sec. 3.3.2 to evaluate the refinement for classification. The LRZ Linux Cluster (ref. Sec. B.2) was used to run the tests.

Strong Scaling With a fixed number of data points and increasing task count, we investigate the strong scaling behaviour. The results shown in Fig. 4.7 indicate that without spatial adaptivity enabled, the methods scale well with increasing number of MPI tasks, when the problem size is fixed. This does not hold if spatial adaptivity is enabled. In this case, the (serial) refinement procedures dominate the runtimes, especially in the case of the tridiagonal decomposition. A decrease of the runtime with increasing number of tasks up to 16 tasks is observed, but the runtime increases again for 32 tasks due to higher communication overhead. If spatial adaptivity is not enabled (regular sparse grid) or not possible (combination grid technique), the parallel scheme is efficient. To obtain similar results for problems with spatial adaptivity, those methods would need to be parallelized too.



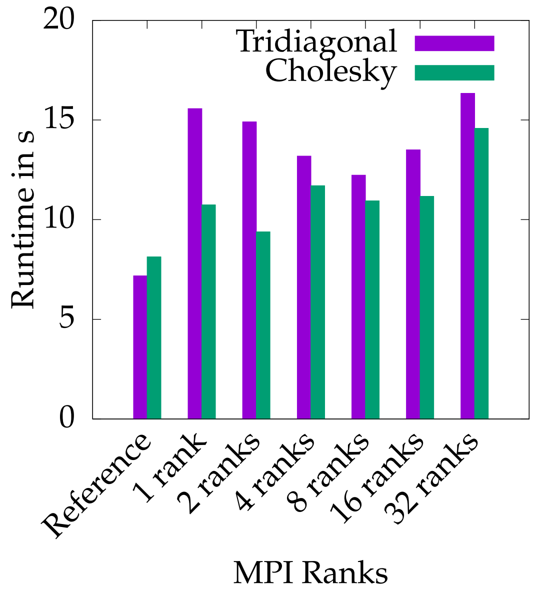
(a) Spatial adaptivity disabled.



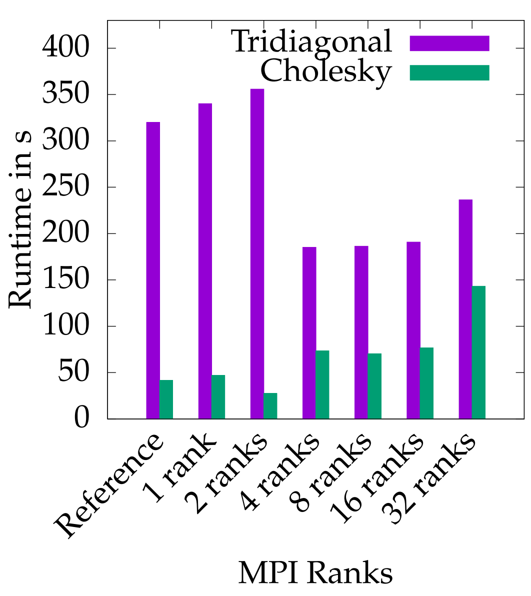
(b) Spatial adaptivity enabled.

Figure 4.7.: Strong scaling results for the parallel linear algebra scheme for both the Cholesky decomposition and the tridiagonal decomposition with (Subfig. b) and without (Subfig. a) spatial adaptivity. With *reference*, we denote the execution without ScaLAPACK enabled. In the case without spatial adaptivity, the Cholesky decomposition performs slightly better. Both techniques have a good scaling with a low number of MPI tasks, but the efficiency decreases with increasing task count due to the smaller problem sizes and the resulting overhead per task. With spatial adaptivity enabled, we observe a drastic increase in runtime due to the serial refinement and coarsening algorithms. For the Cholesky decomposition, this leads to roughly twice the runtime for a small number of tasks and a mitigated scaling behaviour. The tridiagonal decomposition performs even worse, the scheme does not scale well with increasing number of tasks. Eventually, the runtime even increases (from 16 tasks to 32 tasks) due to a larger communication overhead but smaller work packages for each task.
Raw data for this figure: Tab. C.17.

Weak Scaling For the weak scaling test, we choose a dataset size of 20,000 data points per task. The results for increasing number of tasks is shown in Fig. 4.8. If no spatial adaptivity is enabled, the problem setting is well defined. Due to varying communication overhead, we observe a slight variance of the runtimes between the different tasks counts. When spatial adaptivity is employed, the decision of where and how much the model is adapted depends on the local quality of the model. Because this is not a setting for which the workload is per se scalable with the number of cores, the weak scaling test is not well defined for problems with spatial adaptivity enabled. Nevertheless, we test the learning with fixed training dataset size per task and spatial adaptivity enabled as for the strong scaling test. The results shown in Fig. 4.8b show, that the overall runtime is again dominated by the refinement and coarsening procedures. For both the tridiagonal and the Cholesky decomposition, the runtimes vary a lot for different task counts and a qualified statement about the weak scaling behaviour of the scheme with spatial adaptivity enabled is not possible.



(a) Spatial adaptivity disabled.



(b) Spatial adaptivity enabled.

Figure 4.8.: Weak scaling results for the parallel linear algebra scheme for both the Cholesky decomposition and the tridiagonal decomposition with (Subfig. b) and without (Subfig. a) spatial adaptivity. With *reference*, we denote the execution without ScaLAPACK enabled. In the case without spatial adaptivity, the runtimes are not constant independent of the number of tasks due to varying communication overhead depending on the process grid layout. However, the variance is slight. In the case with spatial adaptivity, the runtimes are again dominated by the refinement and coarsening procedures and the runtimes vary a lot between different task counts. Again, the tridiagonal decomposition performs worse than the Cholesky decomposition.
Raw data for this figure: Tab. C.18.

4.5. Summary and Outlook

We presented three layers allowing for distributed computing: Parallelizing over the class-batches, employing the combination grid technique, and distributing the linear algebra routines with ScaLAPACK. With each of those layers, the hardware architecture of a distributed memory system is exploited to speed up the training of both density estimation and classification. Two challenges remain for the future. First, the parallelization of the grid refinement routines and subsequent model updates would lead to better scaling behaviour, if spatial adaptivity is enabled. Second, combining the layers into an holistic parallelization approach might lead to even faster execution times on large distributed systems. Currently, the layers work on their own but do not interact or are even aware of each other. As to work in accordance (e.g. not steal resources from each other), a task scheduler needs to be designed and implemented, which takes into account all parallelization layers. Only then will the parallelized training enfold its full potential.

5. Geometry-aware Sparse Grids

With sparse grids, problems of dimensionalities between 2 and around 100 can be tackled. In extreme cases, they have been applied to even higher dimensional problems (to our knowledge, 166 dimensions is the record so far), but there is a limit even to that. For example, a 200-dimensional sparse grid of level 3 already consists of 80,801 grid points and results in a system matrix of $> 6 \cdot 10^9$ entries for the density estimation method presented in Sec. 3.2. Even with modern hardware, we cannot handle such large matrices, not to mention problems with dimensionalities of $> 1{,}000$ at which we aim. For example, the CIFAR-10 dataset [60] (at which we take a closer look in Sec. 5.5) consists of colored images with 32×32 pixels, thus having 3,072 dimensions.

In this chapter, we present an approach of how to tackle very high-dimensional data such as images with sparse grids. The problem class we look at is constrained in terms that we need knowledge about the geometric relations of the individual dimensions. How the dimensions need to be related such that the problem can be tackled with geometry-aware sparse grids is discussed in Sec. 5.1. The construction of geometry-aware sparse grids via the application of stencils for different kinds of datasets (grayscale images, color channel images, videos, etc.) is presented in Sec. 5.2. Spatially adapting the resulting grids requires tailored routines, which are proposed in Sec. 5.3. Geometry-aware sparse grids also allow to be integrated into a dimensional adaptive setting such as the combination grid technique. We propose a scheme that combines geometry-aware sparse grids and the combination grid technique and show how it is sped up in Sec. 5.4. Finally, we present and discuss the application of geometry-aware sparse grids at the example of several image classification benchmark datasets in Sec. 5.5.

5.1. Geometric Relations in Datasets

Geometric relations between dimensions are present in various kinds of datasets. In image data, pixels are arranged in a regular grid specified by the resolution of each frame. Going the next step to video data, additional to the arrangement of the pixels, there is the chronology of the frames. Geophysical and meteorological datasets often stem from a multitude of measuring stations for which the distance between them implicates a geometric relation of the resulting dimensions. In all of those cases, we can and should use available knowledge about those geometric relations of the dimensions when learning the data. For sparse grids, this means taking this knowledge into account when discretizing the space spanned by the dimensions. We know that any regular sparse grid with a level higher than 2 is infeasible for most of those problems. Thus, in order to decrease the number of grid points, we only include grid points that encode information about the geometric relations, or the interactions, of the dimensions.

Formally, we introduce the concept of interactions between dimensions as follows.

Definition 5.1.1 (Interactions)
For a d-dimensional problem, we call $I \subseteq [d]$ an interaction between those dimensions. A set of interactions $\mathcal{I}$ with $\mathcal{I} :\subset \mathcal{P}([d])$ is valid, if

$$\forall I \in \mathcal{I} \Rightarrow \left(I' \subset I \Rightarrow I' \in \mathcal{I}\right)$$

holds. The valid hull $\mathcal{I}^*$ of $\mathcal{I}$ is the smallest valid set of interactions containing $\mathcal{I}$ per

$$\mathcal{I}^* = \left\{I' \subseteq I \;\middle|\; I \in \mathcal{I}\right\} . \tag{5.1}$$

Following the analysis of variance (ANOVA), a grid point p of level $\boldsymbol{l}$ encodes information between the group of all dimensions j, for which $l_j > 1$. Thus, we define I_p as the interaction this grid point corresponds to per

$$I_p := \left\{j \in [d] \;\middle|\; l_j > 1\right\} . \tag{5.2}$$

This leads us to the definition of geometry-aware sparse grids:

Definition 5.1.2 (Geometry-aware sparse grid)
Let $\mathcal{G}$ be a sparse grid per Def. 2.1.17 and $\mathcal{I}$ a set of valid interactions per Def. 5.1.1. Then, the geometry-aware sparse grid (GaSG) $\mathcal{G}_{\mathcal{I}}$ is given by

$$\mathcal{G}_{\mathcal{I}} := \left\{p \in \mathcal{G} \;\middle|\; I_p \in \mathcal{I}\right\} . \tag{5.3}$$

So, a grid point p is included in the GaSG $\mathcal{G}_{\mathcal{I}}$, if the interaction I_p corresponding to p is included in $\mathcal{I}$. Thus, which grid points we thereby include (and exclude) is controlled by $\mathcal{I}$. Different systematic approaches how to construct $\mathcal{I}$ from a problem definition are proposed in Sec. 5.2. Note that every geometry-aware sparse grid is also a sparse grid, because for every interaction in $\mathcal{I}$, all of its subsets are also included in $\mathcal{I}$ which in turn pulls all the hierarchical parents into the geometry-aware sparse grid.

Cost per Interaction The concept of the interaction corresponding to a grid point from Eq. 5.2 can be transferred to subspace grids and component grids as well. For such a grid $\mathcal{G}$ with level-vector $\boldsymbol{l}$ we set $I_{\mathcal{G}}$ as the interaction this grid corresponds to per

$$I_{\mathcal{G}} := \left\{j \in [d] \;\middle|\; l_j > 1\right\} . \tag{5.4}$$

The cost of an interaction $c_I^n \in \mathbb{N}$ is the number of grid points that are added to the initial grid of level n because of it. Let $s_I \in \mathbb{N}$ be the number of dimensions in I per

$s_I := |I|$. We want to count the number of subspaces corresponding to $I \in \mathcal{I}$ per Eq. 5.4 that are part of $(\texttt{sparseGrid}_{d,n})_{\mathcal{I}}$ (the geometry-aware regular sparse grid of level n). For $l < n$, the subspaces with level-vector $\boldsymbol{l'}$ such that $\|\boldsymbol{l'}\|_1 - d + 1 = l$ are added to the grid. Firstly, only for $l \geq s_I + 1$, subspaces corresponding to I are added. Further, for $l \geq s_I + 1$ there are $l - s_I - 1$ (not differentiable) levels that are freely distributed onto the s_I (differentiable) dimensions. To do so, we have $\binom{l-I-1+s_I-1}{l-s_I-1} = \binom{l-2}{l-s_I-1} = \binom{l-2}{s_I-1}$ possibilities. Each of those subspaces consists of 2^{l-1} grid points. Thus we obtain:

$$c_I^n = \sum_{l=s_I+1}^{n} 2^{l-1} \binom{l-2}{s_I - 1} . \tag{5.5}$$

In total, the size of $(\texttt{sparseGrid}_{d,n})_{\mathcal{I}}$ is

$$c_{\mathcal{I}}^n := \left|(\texttt{sparseGrid}_{d,n})_{\mathcal{I}}\right| = \sum_{I \in \mathcal{I}} c_I^n . \tag{5.6}$$

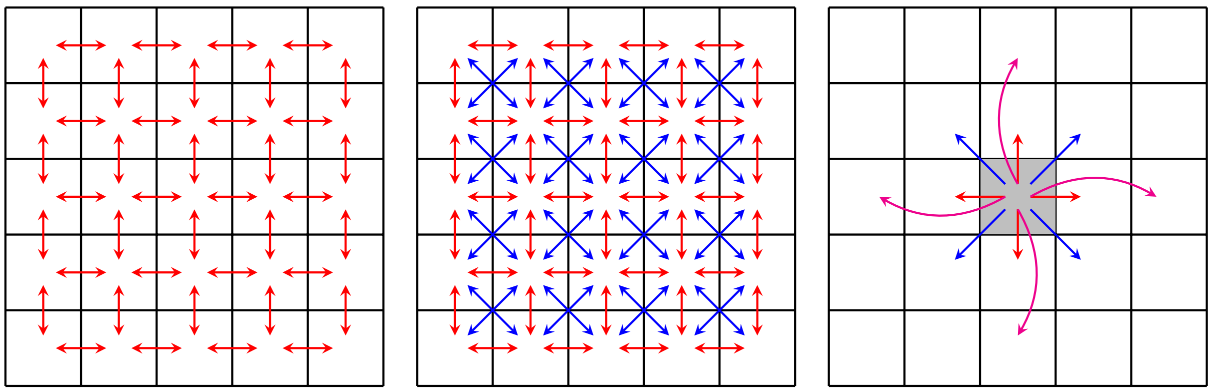
5.2. Image Stencils

In this section, we look at how to build up the interactions for image classification problems via stencils that iterate over the image. Many of the discussed stencils have been applied for a long time in cellular automata theory [97] or in image processing concerning pixel connectivity [84]. We first look at grayscale images in Sec. 5.2.1. Then, we consider how to build up interactions when each data point represents a series of a iteratively coarsened image in Sec. 5.2.2. Next, we turn to colored images in Sec. 5.2.3. Lastly, we look at how to crop the resulting sparse grids by cutting off points that are not mandatory in Sec. 5.2.4.

5.2.1. Grayscale Images

Interpreting an image dataset consisting of images with r pixels as any r-dimensional dataset does not consider the nature of the dataset, namely the images. A better approach is to take the resolution in each spatial dimension $a_x, a_y \in \mathbb{N}$ with $a_x \cdot a_y = d$ into account. Grouping the dimensions as the corresponding rectangle of width a_x and height a_y already reveals further information about the relation of individual dimensions. Obviously, some pixels are neighbors of each other but farther away from other pixels. We want to include the interactions of the pixels that are in the neighborhood of each other into the geometry-aware sparse grid. Different concepts of neighborhood are presented in the following. In essence, they differ in two characteristics:

1. How far away are pixels allowed to be in order to count as neighbors, and
2. how many pixels are grouped together in a single interaction.


(a) Visualization of the DN stencil depicting all interactions of a 5×5 image.

(b) Visualization of the DDN stencil depicting all interactions of a 5×5 image.

(c) Visualization of the DBP-2 stencil depicting all interactions of the gray pixel with other pixels.

Figure 5.1.: Several examples of pairwise stencils in a 5×5 image.

Allowing longer distances or larger groups ultimately leads to larger grids. Thus, we present various options such that a choice can be made according to the problem size and the computational power at hand.

5.2.1.1. Pairwise Stencils

The pairwise stencils we are about to introduce group at most two pixels together. They only vary in how far two pixels are allowed to be from each other in order to be grouped in an interaction. We start with stencils for image datasets and then generalize to stencils applicable to video data or data of even higher number of spatial orders [1].

Direct Neighbor Stencil The *direct neighbor stencil* (abbreviated DN) includes pair-wise interactions of pixels that are direct neighbors (horizontal or vertical) of each other in the image. It is derived from the von Neumann neighborhood [96]. For an image of size $a_x \times a_y$ (thus, $d = a_x \cdot a_y$) and the position of each pixel j given by $\left(j_x, j_y\right)$, the set of interactions $\mathcal{I}^{\mathtt{DN}}$ is then given by

$$\mathcal{I}^{\mathtt{DN}} := \left\{ \{j, j'\} \subseteq [d] \;\middle|\; \left|j_x - j'_x\right| + \left|j_y - j'_y\right| = 1 \right\}^* . \tag{5.7}$$

In Fig. 5.1a, this is visualized at the example of a 5×5 image. The number of the pairwise interactions between the pixels contained in $\mathcal{I}^{\mathtt{DN}}$ is $(a_x - 1)a_y + a_x(a_y - 1) = 2a_x a_y - \left(a_x + a_y\right) \in \mathcal{O}\left(d\right)$. Remember that the number of pairwise interactions alone contained in a (non-geometry-aware) regular sparse grid is $\binom{d}{2} \in \mathcal{O}\left(d^2\right)$. Due to

[1]We intentionally refrain from using the term *dimensions* again in this context, as it is already reserved for the number of pixels.

building the hull of those interactions per Def. 5.1.1, the trivial interactions of $\varnothing$ and $[d]$ are also contained in $\mathcal{I}^{\mathtt{DN}}$. The size $c^n_{\mathcal{I}^{\mathtt{DN}}}$ of the resulting geometry-aware regular sparse grid of level $n \geq 3$ with the direct neighbor stencil then is given by

$$\begin{aligned} c^n_{\mathcal{I}^{\mathtt{DN}}} =& 1 + d\left(2^n - 2\right) + \left(2a_x a_y - \left(a_x + a_y\right)\right) \cdot \sum_{l=3}^{n} 2^{l-1}\left(l-2\right) \\ =& 1 + d\left(2^n - 2\right) + \left(2d - \left(a_x + a_y\right)\right) \cdot \left(2^n n - 3 \cdot 2^n + 4\right) . \end{aligned}$$

While the number of grid points in the regular sparse grid $\left|\mathtt{sparseGrid}_{d,n}\right|$ is in $\mathcal{O}\left(2^n n^{d-1}\right)$ (ref. Eq. 2.24), the number of grid points in the geometry-aware regular sparse grid with the direct neighbor stencil $c^n_{\mathcal{I}^{\mathtt{DN}}}$ is in $\mathcal{O}\left(2^n \cdot d \cdot n\right)$ which is only linear in the dimension d. This is a drastic reduction of the cost and opens a way to employ sparse grid for the mentioned high-dimensional data problems.

Direct and Diagonal Neighbor Stencil The *direct and diagonal neighbor stencil* (abbreviated DDN) contains not only the direct horizontal and vertical neighbors of each pixel, but also its direct diagonal neighbors:

$$\mathcal{I}^{\mathtt{DDN}} := \left\{ \{j, j'\} \subseteq [d] \;\middle|\; \left|j_x - j'_x\right| \leq 1 \text{ and } \left|j_y - j'_y\right| \leq 1 \right\}^* . \tag{5.8}$$

It is derived from the Moore neighborhood of radius 1 [55]. We visualize this stencil again at the example of a 5×5 image in Fig. 5.1b. The number of the pairwise interactions between the pixels contained in $\mathcal{I}^{\mathtt{DDN}}$ is $(a_x - 1)a_y + a_x(a_y - 1) + 2\left(a_x - 1\right)\left(a_y - 1\right) = 4a_x a_y - 3\left(a_x + a_y\right) + 2 \in \mathcal{O}\left(d\right)$. Again, the trivial interactions $\varnothing$ and $[d]$ are also contained in $\mathcal{I}^{\mathtt{DDN}}$ because we construct the stencil via the valid hull. Then, the size $c^n_{\mathcal{I}^{\mathtt{DDN}}}$ of the resulting geometry-aware regular sparse grid of level $n \geq 3$ with the direct and diagonal neighbor stencil then is given by

$$\begin{aligned} c^n_{\mathcal{I}^{\mathtt{DDN}}} =& 1 + d\left(2^n - 2\right) + \left(4a_x a_y - 3\left(a_x + a_y\right) + 2\right) \cdot \sum_{l=3}^{n} 2^{l-1}\left(l-2\right) \\ =& 1 + d\left(2^n - 2\right) + \left(4d - 3\left(a_x + a_y\right) + 2\right) \cdot \left(2^n n - 3 \cdot 2^n + 4\right) . \end{aligned} \tag{5.9}$$

As for the asymptotic growth of $c^n_{\mathcal{I}^{\mathtt{DDN}}}$, we get $c^n_{\mathcal{I}^{\mathtt{DDN}}} \in \mathcal{O}\left(2^n \cdot d \cdot n\right)$, which again is only linear in d.

Generalization to Distance-bounded Pairwise Stencil With the geometric distance between two pixels j and j' given by

$$\begin{aligned} \mathtt{dist}^{\mathrm{2D}} \colon\ & [d]^2 \to \mathbb{R} , \\ & (j, j') \mapsto \sqrt{\left|j_x - j'_x\right|^2 + \left|j_y - j'_y\right|^2} , \end{aligned} \tag{5.10}$$

we look at the direct neighbor stencil as all pairwise interactions with $\mathtt{dist}^{\mathrm{2D}}(j, j') \leq 1$ and at the direct and diagonal neighbor stencil as all pairwise interactions with

$\texttt{dist}^{\text{2D}}(j, j') \leq \sqrt{2}$. This generalizes to the *distance-bounded pairwise stencil* with distance r (abbreviated $\texttt{DBP-}r$):

$$\mathcal{I}^{\texttt{DBP-}r} := \left\{ \{j, j'\} \subseteq [d] \;\middle|\; \texttt{dist}^{\text{2D}}(j, j') \leq r \right\}^* . \tag{5.11}$$

For each pixel j, the number of other pixels j' with $\texttt{dist}^{\text{2D}}(j, j') \leq r$ grows quadratically with r, i.e. $|\mathcal{I}^{\texttt{DBP-}r}| \in \mathcal{O}\left(d \cdot r^2\right)$. This implies that $c^n_{\mathcal{I}^{\texttt{DBP-}r}} \in \mathcal{O}\left(2^n \cdot d \cdot n \cdot r^2\right)$ is still linear in the dimension d. Special cases of the $\texttt{DBP-}r$ are the $\texttt{DN}$ stencil ($r = 1$) and the $\texttt{DDN}$ stencil ($r = \sqrt{2}$). Those two and the $\texttt{DBP-2}$ stencil are visualized in Fig. 5.1.

Generalization to 3D and Beyond The image stencil $\texttt{DBP-}r$ (ref. Eq. 5.11) further generalizes to a stencil for video data or even a stencil for data of arbitrary high number of spatial orders. Let $e \in \mathbb{N}$ be that number of spatial orders, such that $e = 2$ for a (grayscale) image and $e = 3$ for a (grayscale) video. The number of dimensions d is derived from the resolution a_i in the ith spatial order as $d = \prod_{i=1}^{e} a_i$ and the position of each e-xel (in other literature [16, 99] referred to as *hyper-voxel*) j (for $e = 2$ a pixel, for $e = 3$ a voxel) is given by a e-dimensional vector $(j_1, \ldots, j_e)$. Then, with the geometric distance between two e-xels j and j' given by

$$\begin{aligned} \texttt{dist}^{e\text{-D}} \colon\ [d]^2 &\to \mathbb{R}, \\ (j, j') &\mapsto \sqrt{\sum_{i=1}^{e} |j_i - j'_i|^2}, \end{aligned} \tag{5.12}$$

the *e-ordered distance-bounded pairwise stencil* with distance r (abbreviated $e\texttt{-DBP-}r$) is given by

$$\mathcal{I}^{e\texttt{-DBP-}r} := \left\{ \{j, j'\} \subseteq [d] \;\middle|\; \texttt{dist}^{e\text{-D}}(j, j') \leq r \right\}^* . \tag{5.13}$$

For each e-xel j, the number of other e-xels j' that are contained in the hypersphere of radius r around j (which is related to the Delannoy number [6]) grows polynomially with degree e in r, thus: $|\mathcal{I}^{e\texttt{-DBP-}r}| \in \mathcal{O}\left(d \cdot r^e\right)$. It follows, that $c^n_{\mathcal{I}^{e\texttt{-DBP-}r}} \in \mathcal{O}\left(2^n \cdot d \cdot n \cdot r^e\right)$ is also linear in the number of dimensions d.

5.2.1.2. Tupled Stencils

So far, we only looked at stencils that generate interactions of at most two dimensions. The next step is to build up stencils that contain also interactions of more than two dimensions.

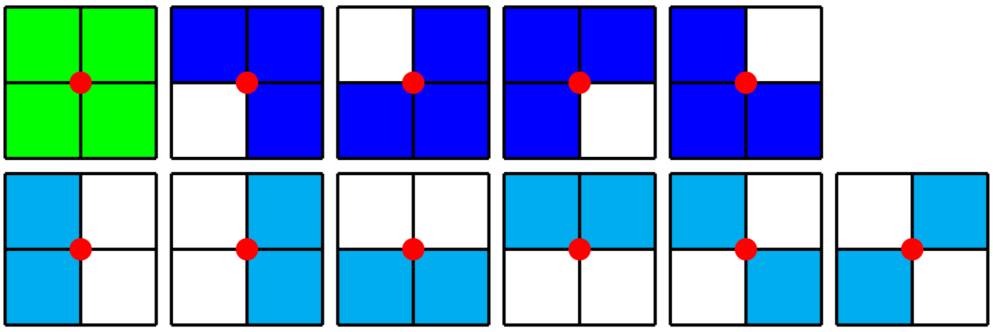


Figure 5.2.: The SQ stencil groups together four pixels sharing a corner into one interaction. The four pixels colored in green (top left) form such an interaction. All subgroups of size three of the original four pixels are also pulled into the $\mathcal{I}^{\texttt{SQ}}$. The four possible combinations are depicted with dark blue-colored pixels at the top. Also, all subgroups of size two are pulled into $\mathcal{I}^{\texttt{SQ}}$, resulting in the six possible combinations depicted at the bottom with light blue-colored pixels.

Square Stencil For image data, taking into account the combination of four pixels forming a square is the most basic structure if we look at interactions with more than two dimensions. We define the *square stencil* (abbreviated SQ) as

$$\begin{aligned}\mathcal{I}^{\texttt{SQ}} := \Big\{ \big\{ j^{(1)}, j^{(2)}, j^{(3)}, j^{(4)} \big\} \subseteq [d] \;\Big|\; & \big(j_x^{(1)}, j_y^{(1)}\big) = \big(j_x^{(2)} - 1, j_y^{(2)}\big) \\ & = \big(j_x^{(3)}, j_y^{(3)} - 1\big) = \big(j_x^{(4)} - 1, j_y^{(4)} - 1\big) \Big\}^* . \end{aligned} \tag{5.14}$$

which includes all quadruples of pixels sharing a corner. The stencil for one corner and the resulting subgroups are shown in Fig. 5.2. Because we define the interactions via the hull (see Def. 5.1.1), all possible subsets of all interactions are also contained in $\mathcal{I}^{\texttt{SQ}}$. In contrast to the pairwise stencils defined in Sec. 5.2.1.1, those are not only the trivial interactions $\varnothing$ and $[d]$ but also all tree-tuples of pixels sharing a corner and all pairs of pixels sharing a corner (which correspond to the DDN stencil). Those non-trivial subsets are visualized in Fig. 5.2 at the example of one interaction of four pixels.

To calculate the size $c^n_{\mathcal{I}^{\texttt{SQ}}}$ of the geometry-aware regular sparse grid with the square stencil, we first count the interactions of size four. This quantity corresponds to the number of inner corners in the image, which is $(a_x - 1) * (a_y - 1) = a_x a_y - (a_x + a_y) + 1 = d - (a_x + a_y) + 1$. For each of those interactions of size four, there are four subsets of size three which are added. Thus, the number of interactions of size three is $4d - 4(a_x + a_y) + 4$. For the remaining interactions of size smaller than three, we refer to $c^n_{\mathcal{I}^{\texttt{DDN}}}$ from Eq. 5.9. For an interaction of size four to be present in the geometry-aware sparse grid, the level has to be at least 5. In total, with Eq. 5.5 we get

$$\begin{aligned} c^n_{\mathcal{I}^{\texttt{SQ}}} =& c^n_{\mathcal{I}^{\texttt{DDN}}} + \big(4d - 4(a_x + a_y) + 4\big) \cdot \sum_{l=4}^{n} 2^{l-1} \binom{l-2}{2} \\ & + \big(d - (a_x + a_y) + 1\big) \cdot \sum_{l=5}^{n} 2^{l-1} \binom{l-2}{3} \\ =& 1 + d\left(2^n - 2\right) + \big(4d - 3\left(a_x + a_y\right) + 2\big) \cdot \left(2^n n - 3 \cdot 2^n + 4\right) \end{aligned}$$

$$
\begin{aligned}
&+ \left(4d - 4(a_x + a_y) + 4\right) \cdot \left(2^{n-1}n^2 - 7 \cdot 2^{n-1}n + 7 \cdot 2^n - 8\right) \\
&+ \left(d - (a_x + a_y) + 1\right) \cdot \frac{1}{6} \left(2^n n^3 - 3 \cdot 2^{n+2} n^2 + 53 \cdot 2^n n - 45 \cdot 2^{n+1} + 96\right) \\
=& 1 + d\left(2^n - 2\right) + \left(4d - 3\left(a_x + a_y\right) + 2\right) \cdot \left(2^n n - 3 \cdot 2^n + 4\right) \\
&+ \left(d - (a_x + a_y) + 1\right) \cdot \frac{1}{6} \left(2^n n^3 - 31 \cdot 2^n n + 39 \cdot 2^{n+1} - 96\right) . \qquad (5.15)
\end{aligned}
$$

In total, $c^n_{\mathcal{I}^{\mathtt{SQ}}} \in \mathcal{O}\left(2^n \cdot d \cdot n^3\right)$.

Generalization to Corner-centered Stencil for *e*-xels The square stencil includes all four-tuples of pixels sharing the same corner. We generalize this in two ways:

1. Group even more pixels in one interaction by specifying in which radius r pixels around the pixel corner are allowed to be.
2. Generalize from image data, which has two spatial orders, to data of arbitrary high number of spatial orders e containing e-xels.

The corners of a d-dimensional dataset containing e-xels with $d = \prod_{i=1}^{e} a_i$ are given by $K = \{(x_1 + 0.5, x_2 + 0.5, \cdots, x_e + 0.5) \mid x_i \in [a_i - 1]\}$. With the distance from an e-xel j to a corner $k \in K$ obtained via

$$
\begin{aligned}
\mathtt{dist}^{e\text{-D}}_{\text{corner}} \colon K \times [d] &\to \mathbb{R}, \\
(k, j) &\mapsto \sqrt{\sum_{i=1}^{e} |k_i - j_i|^2},
\end{aligned}
\qquad (5.16)
$$

the *e-ordered corner-centered distance-bounded stencil* with distance r (abbreviated e-CCDB-r) is then given by

$$
\mathcal{I}^{e\text{-}\mathtt{CCDB}\text{-}r} := \left\{ E \subseteq [d] \;\middle|\; \exists k \in K : \forall j \in E : \mathtt{dist}^{e\text{-D}}_{\text{corner}}(k, j) \le r \right\}^* . \qquad (5.17)
$$

The square stencil is the special case 2-CCDB-$\frac{\sqrt{2}}{2}$, another example with $e = 2$ and $r = 2$ is visualized in Fig. 5.3. To estimate the number of grid points in the resulting geometry-aware sparse grid, we need to know the maximum size of an interaction, i.e. the number of e-xels which are at most contained in the hypersphere of radius r around any corner. We call this number $s_{e,r} \in \mathbb{N}$, whereas we know that $s_{e,r} \in \mathcal{O}\left(r^e\right)$. A geometry-aware sparse grid using this stencil has to be at least of level $s_{e,r} + 1$ to take into account all interactions obtained by the stencil. Further, the number of corners is in $\mathcal{O}\left(d\right)$ and in particular independent of r. With the sum over the binomial coefficient from Eq. 5.5, we know that in total, $c^n_{\mathcal{I}^{e\text{-}\mathtt{CCDB}\text{-}r}} \in \mathcal{O}\left(2^n \cdot d \cdot n^{s_{e,r}-1}\right)$. We see that with growing $s_{e,r}$, the size of the geometry-aware sparse grid resulting from this stencil easily grows out of hand.

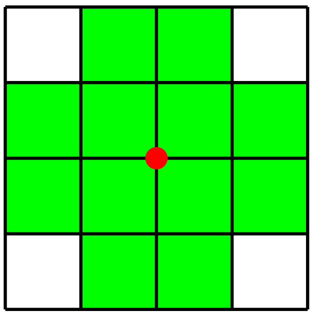


Figure 5.3.: The 2-CCDB-2 stencil groups together 10 pixels around one corner into one interaction. This also results in 4,082 possible subgroups of size > 1 of which most are not redundant with subgroups generated from other corners.

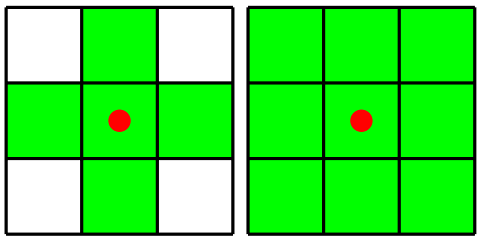


Figure 5.4.: On the left, the 2-XCDB-r stencil is visualized grouping together five pixels. On the right, 2-XCDB-$\sqrt{2}$ is grouping together nine pixels.

***e*-xel-centered Stencil** Instead of centering the stencil at a corner, we can also center it at an e-xel. This yields a slightly different structure than the e-CCDB-r stencil and gives us the *e-ordered e-xel-centered distance-bounded stencil* with distance r (abbreviated e-XCDB-r):

$$\mathcal{I}^{e\text{-}\mathtt{XCDB}\text{-}r} := \left\{ E \subseteq [d] \;\middle|\; \exists j' \in [d] : \forall j \in E : \mathtt{dist}^{e\text{-D}}(j, j') \leq r \right\}^{*}. \tag{5.18}$$

A visualization for $e = 2$ and different values of r is shown in Fig. 5.4. With this stencil, for each e-xel j, we group together all e-xels j' that are contained in the hypersphere of radius r around j. The maximum size of the generated interactions is denoted by $s'_{e,r} \in \mathbb{N}$. We draw the same conclusions as with the e-CCDB-r stencil: $s'_{e,r} \in \mathcal{O}\left(r^e\right)$ and $c^n_{\mathcal{I}^{e\text{-}\mathtt{XCDB}\text{-}r}} \in \mathcal{O}\left(2^n \cdot d \cdot n^{s'_{e,r}-1}\right)$.

5.2.1.3. Comparison of Grayscale Stencils

In Fig. 5.5, the number of points in the regular sparse grid is compared to the number of points of the geometry-aware sparse grid resulting from different stencils for a dataset consisting of 28×28 grayscale images. The number of points in the regular sparse grid surpasses 10^6 already at level 3, rendering it infeasible for such problems. In contrast, the number of points in the geometry-aware sparse grid is smaller by a factor in the order of 10^3 for level 3, containing between 10,753 and 22,409 grid points for the chosen stencils. However, larger stencils such as the 2-XCDB-1 are also difficult to apply to

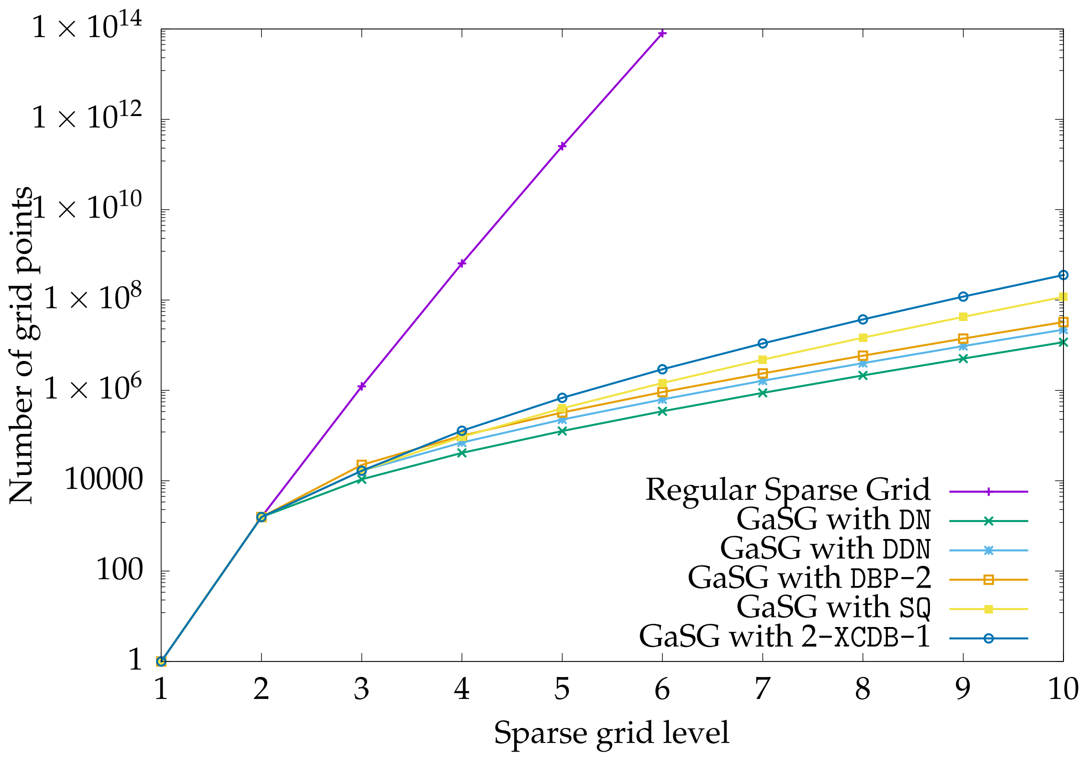


Figure 5.5.: Comparison of the number of grid points in a regular sparse grid vs. geometry-aware sparse grids for a 28×28 grayscale image dataset with different stencils. After level 2, the number of points in the regular sparse grid is not feasible to handle anymore while the number of points in the geometry-aware sparse grids is growing much slower. Among the geometry-aware sparse grids, those with smaller stencils grow slowest.
Raw data for this figure: Tab. C.19.

such high-dimensional problems, as the number of grid points in level 6 (which is the first level for which this stencil unfolds its full potential) is already at $2.93 \cdot 10^6$ grid points. Due to those numbers, treating the geometry-aware sparse grids in a monolithic fashion is not suitable. We discuss how to apply the combination grid technique in Sec. 5.4

5.2.2. Multilayer Stencils

A lesson to take away from deep learning [4] is that when learning an image dataset, it is best to project the hierarchy of the data to a hierarchy in the model. For example, to recognize a human face, it is good to know that a face consists of two eyes, a nose and a mouth. Further, an eye can be partitioned to eyelashes, the eyeball, the pupil and so on. Important is that we model the coarse structures as well as the details. With increasing resolution of an image (or the spatially ordered dataset of order e), two neighboring pixels (or e-xels) differ less and less. Thus, just looking at the relations of pixels in the vicinity of each other as we have proposed so far does not suffice to model the coarser structures of the dataset. In order to incorporate those structures, we artificially coarsen the images in the dataset recursively and append the resulting dimensions to the original dataset as discussed in Sec. 5.2.2.1. Then, we investigate how to connect the layers with stencils heeding the layer hierarchy in Sec. 5.2.2.2 before comparing the stencils in Sec. 5.2.2.3. Some of the stencils presented in this section have been implemented in SG++ during a bachelor project [104].

5.2.2.1. Data Preprocessing

We start again with a dataset with e spatial orders and resolution a_i in order i and are aiming to construct a set of $k+1$ layers

$$L = \left\{ l^{(0)}, l^{(1)}, \dots, l^{(k)} \right\} \tag{5.19}$$

by iteratively coarsening the image to the next layer. Thereby, layer 0 corresponds to the original image and thus has the finest resolution whereas layer $h+1$ is coarser than layer h. The resolution at layer h in the ith spatial order is then given by $a_i^{(h)}$, thus the number of dimensions in layer h is $d^{(h)} = \prod_{i=1}^{e} a_i^{(h)}$. Concatenating those dimensions together, we extend each original data point of dimension $d^{(0)}$ to a data point of dimension $d^{(\text{total})} = \sum_{h=0}^{k} d^{(h)}$. Each layer contains the indices of the dimensions associated to it, i.e. $l^{(0)} = \left[d^{(0)} \right]$ and $l^{(h)} = \left\{ d^{(h-1)} + 1, \dots, d^{(h)} \right\} \ \forall h \in [k]$.

An example of iteratively coarsening an image of original size 28×28 is shown in Fig. 5.6. The images are downscaled using bicubic interpolation [51].

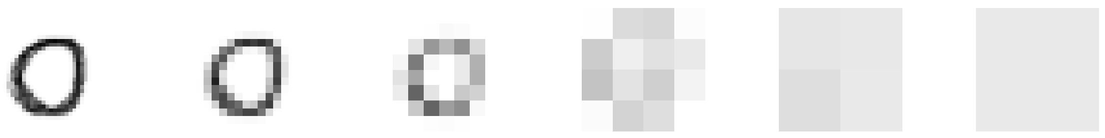

(a) 28×28 (b) 14×14 (c) 7×7 (d) 4×4 (e) 2×2 (f) 1×1

Figure 5.6.: Iteratively coarsening an image of original size 28×28.

For the coarsening factor, the most natural choice is to divide the resolution by 2 in each spatial order from layer $h+1$ to layer h. If for the ith spatial order, the corresponding $a_i^{(h+1)}$ is not divisible by 2, we round the result up: $a_i^{(h)} = \left\lceil \frac{a_i^{(h+1)}}{2} \right\rceil$. In this case, the original resolution in every spatial order is 2^r with $r \in \mathbb{N}$ and the resulting dimension is

$$d^{(\text{total})} = \sum_{h=0}^{r} \frac{1}{2^{h \cdot e}} d^{(0)} = d^{(0)} \frac{2^e - 2^{-e \cdot r}}{2^e - 1} . \tag{5.20}$$

In general, the number of total dimensions $d^{(\text{total})}$ is in $\mathcal{O}\left(d^{(0)} \cdot \max_{i \in [e]} a_i^{(0)}\right)$.

5.2.2.2. Stencils for the Layer Hierarchy

We now look at stencils that connect the dimensions of different layers by grouping them in interactions of various size. Apart from connecting the layers, we also apply a stencil s to each layer individually, whereas s is any of the previously presented stencils.

No Interactions between Layers The first possibility so treat the layers is to not connect them at all. This stencil is called the *no layer interaction* stencil (abbreviated `NoLay`). With $c^n_{\mathcal{I}^s_{l^h}}$ denoting the number of grid points resulting from applying stencil s to layer h, the number of grid points for the `NoLay` stencil is then given by

$$c^n_{\mathcal{I}^{s+\texttt{NoLay}}} = -k + \sum_{h=0}^{k} c^n_{\mathcal{I}^s_{l^h}} . \tag{5.21}$$

This corresponds (except for the summand $-k$) to the sum when applying s to a dataset containing only layer h for each h.

Neighboring Layers Pairwise Connecting Stencil We want to group an e-xel j from layer h together with an e-xel j' from layer $h+1$ if j' contains information from j. To this end, we define the relation $P \subseteq \left[d^{(\text{total})}\right]^2$, which specifies if two e-xels of two different

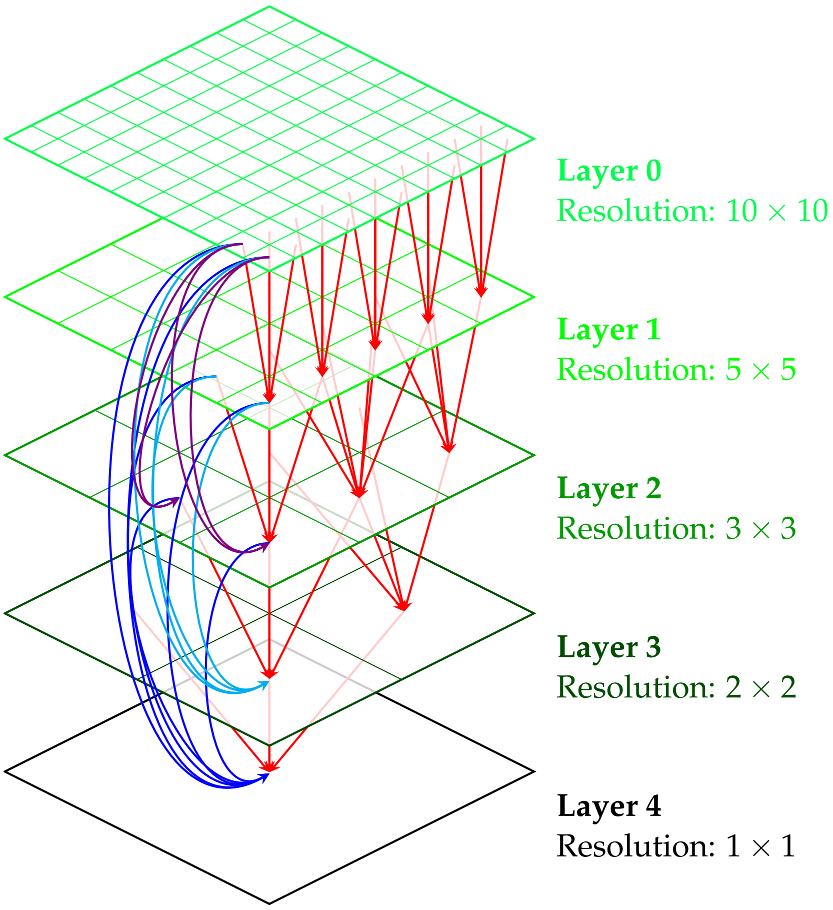


Figure 5.7.: Coarsening an image with initial resolution 10×10 leads to the four additional layers 5×5, 3×3, 2×2 and 1×1. The arrows denote possible pairwise interactions between pixels of different layers. If the `NoLay` stencil is chosen, none of those interactions are taken. Some example arrows in red depict interactions that are added via the `NPLay` stencil. Other example arrows in shades of blue depict interactions that are added via the `APLay` stencil. Adapted from [104].

layers are overlapping as

$$P_L = \left\{ (j, j') \in \left[d^{(\text{total})}\right]^2 \;\middle|\; \exists h, h' \in [k] : h < h' \wedge j \in l^{(h)} \wedge j' \in l^{(h')} \wedge \exists i \in [e] : \left(\left(\frac{j'_i - 1}{a_i^{(h')}} \leq \frac{j_i - 1}{a_i^{(h)}} < \frac{j'_i}{a_i^{(h')}} \right) \vee \left(\frac{j'_i - 1}{a_i^{(h')}} < \frac{j_i}{a_i^{(h)}} \leq \frac{j'_i}{a_i^{(h')}} \right) \right) \right\} . \tag{5.22}$$

Then, we obtain the *neighboring layers pairwise connecting* stencil (abbreviated `NPLay`) as

$$\mathcal{I}^{\texttt{NPLay}} := \left\{ \{j, j'\} \subset \left[d^{(\text{total})}\right] \;\middle|\; \exists h \in [k] : j \in l^{(h)} \wedge j' \in l^{(h+1)} \wedge (j, j') \in P_L \right\}^* . \tag{5.23}$$

In Fig. 5.7, the red arrows depict some of the interactions pulled in via the `NPLay` stencil at the example of a five-layered problem. Each e-xel j of layer h is part of exactly one such pair from h to $h+1$ only if for each spatial order i, the resolution $a_i^{(h)}$ is divisible by $a_i^{(h+1)}$. In this case, the number of total pairs is $d^{(\text{total})} - 1$. In general, if there are spatial orders i for which $a_i^{(h)}$ is not divisible by $a_i^{(h+1)}$, the number of pairs $p^{\texttt{NPLay}}$ is given by

$$p^{\texttt{NPLay}} = \sum_{h=0}^{k-1} \prod_{i=1}^{e} \left(a_i^{(h)} + \left(\frac{a_i^{(h+1)}}{\gcd\left\{a_i^{(h)}, a_i^{(h+1)}\right\}} - 1 \right) \cdot \frac{a_i^{(h)}}{\gcd\left\{a_i^{(h)}, a_i^{(h+1)}\right\}} \right) . \tag{5.24}$$

In the worst case, p is only slightly less than $2^e d^{(\text{total})}$. However, this case only occurs for a very slow image coarsening rate and awkward image resolutions. With appropriate data preprocessing and a reasonable coarsening rate (of e.g. 2), the best case of $p = d^{(\text{total})}$ is well achievable. However, for the general case, the resulting number of grid points $c^n_{\mathcal{I}^{\texttt{NPLay}}}$, which are added to the geometry aware sparse grid of level n with the `NPLay` stencil, is in $\mathcal{O}\left(2^n \cdot 2^e \cdot d^{(\text{total})} \cdot n\right)$.

All Layers Pairwise Connecting Stencil Instead of only connecting neighboring layers, we can also connect all layers with each other with the same idea behind of grouping together e-xels that contain information from each other. The *all layers pairwise connecting* stencil (abbreviated `APLay`) is given by

$$\mathcal{I}^{\texttt{APLay}} := \left\{ \{j, j'\} \subset \left[d^{(\text{total})}\right] \;\middle|\; (j, j') \in P_L \right\}^* . \tag{5.25}$$

Analogous to the `NPLay` stencil, the number of pairs $p^{\texttt{APLay}}$ is

$$p^{\texttt{APLay}} = \sum_{h=0}^{k-1} \sum_{h'=h+1}^{k} \prod_{i=1}^{e} \left(a_i^{(h)} + \left(\frac{a_i^{(h')}}{\gcd\left\{a_i^{(h)}, a_i^{(h')}\right\}} - 1 \right) \cdot \frac{a_i^{(h)}}{\gcd\left\{a_i^{(h)}, a_i^{(h')}\right\}} \right) \tag{5.26}$$

and $c^n_{\mathcal{I}^{\texttt{APLay}}} \in \mathcal{O}\left(2^n \cdot 2^e \cdot k \cdot d^{(\text{total})} \cdot n\right)$. In Fig. 5.7, the blue shaded arrows depict some of the interactions pulled in via the APLay stencil at the example of a five-layered problem.

All Layers Full Connecting Stencil We generalize the APLay stencil to not only include pairwise interactions of overlapping e-xels but to also include groups of those if they all share a common information through the coarsening process. This leads us to the *all layers full connecting* stencil (abbreviated AFLay) as

$$\mathcal{I}^{\texttt{AFLay}} := \left\{E \subset \left[d^{(\text{total})}\right] \;\middle|\; \forall j, j' \in E : (j, j') \in P_L \vee (j', j) \in P_L\right\}^* . \tag{5.27}$$

The number of interactions of size $k+1$ is given by $d^{(0)}$ only if for each spatial order i, the resolution $a_i^{(0)}$ is a power of 2. Otherwise, the number of those interactions is in $\mathcal{O}\left(d^{(0)}k^e\right)$ in the worst case. For the number of points $c^n_{\mathcal{I}^{\texttt{AFLay}}}$, which are added to the geometry-aware sparse grid with the AFLay stencil, holds: $c^n_{\mathcal{I}^{\texttt{AFLay}}} \in \mathcal{O}\left(2^n \cdot k^e \cdot k \cdot d^{(\text{total})} \cdot n^k\right)$. Even for lower spatial orders e and a moderate number of additional layers k, this is not feasible to handle on current hardware. Not only do we have lots of interaction groups of size $k+1$ but also do they pull in an even higher number of interaction groups of slightly lower size resulting a huge number of grid points. In Fig. 5.8, the AFLay is visualized at the example with $e = 1$ and $d^{(0)} = 10$. A coarsening factor of 2 results in four additional layers. The e-xels overlap such that 12 unique interaction groups of five e-xels (one from each layer) are generated with the AFLay stencil.

5.2.2.3. Comparison of Layer Stencils

In Fig. 5.9, the number of points in the regular sparse grid of both the original dimensionality (784) and the extended dimensionality (1,050) is compared to the number of points of the geometry-aware sparse grid resulting from DN together with different multilayer stencils for a dataset consisting of 28×28 grayscale images. Choosing a coarsening factor of 2, the resulting additional layers have a resolution of 14×14, 7×7, 4×4, 2×2 and 1×1. Thus, the interactions added via the AFLay stencil have a size of six, which are taken fully into account starting at a grid of level 7. The number of grid points of the corresponding geometry-aware sparse grid (together with the DN) is $3.21 \cdot 10^7$, which renders this stencil infeasible for such problems on the hardware we have at hand. However, the other multilayer stencil offer a viable option to incorporate the relations between coarse and fine structures into the model.

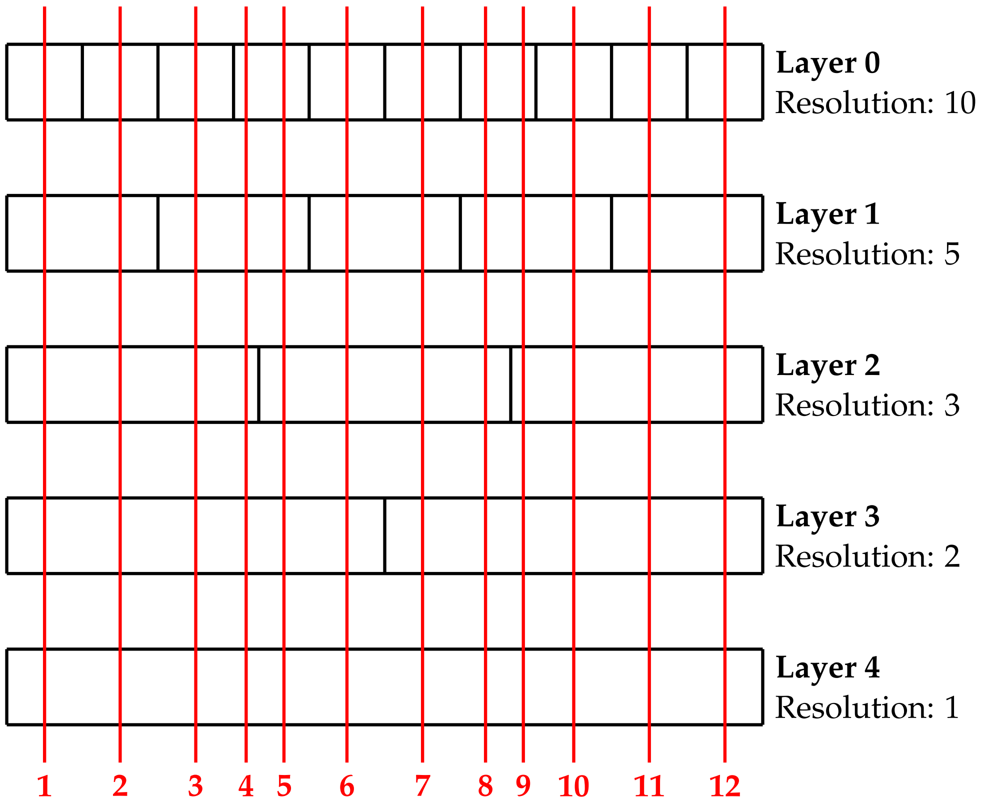


Figure 5.8.: The initial 10 *e*-xels of the only spatial order are coarsened to four additional layers with resolutions of 5, 3, 2 and 1. This results in 12 unique interactions of size five, each containing an *e*-xel from each of the five layers.

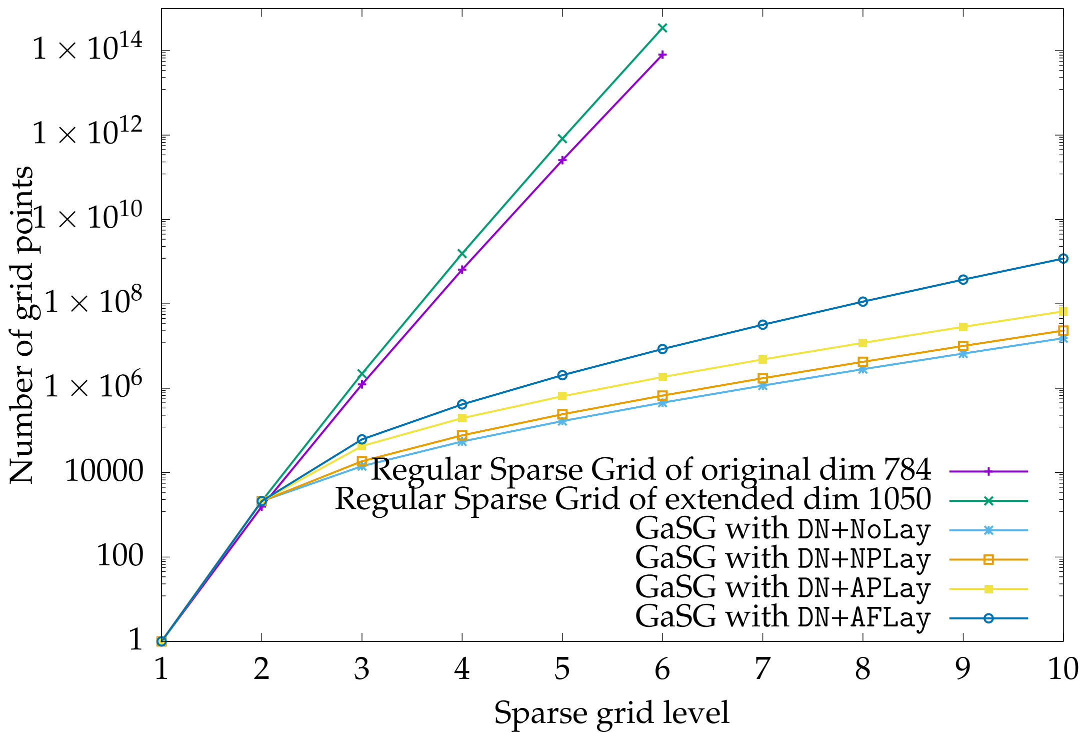


Figure 5.9.: Comparison of the number of grid points in a regular sparse grid vs. geometry-aware sparse grids for a 28×28 image dataset extended to a total of five layers resulting in 1,050 dimensions with different multilayer stencils. After level 2, the number of points in the regular sparse grid is not feasible to handle anymore while the number of points in the geometry-aware sparse grids is growing much slower.
Raw data for this figure: Tab. C.20.

5.2.3. Colored Datasets

A colored image or video consists of the three channels red (R), green (G) and blue (B). Between those channels, there exists no order, so we should not treat those colors just as another spatial order. Otherwise, we would need to define a ″left″ channel, a ″center″ channel and a ″right″ channel. Then, when employing a `DBP-`r stencil with $r < 2$ (e.g. the `DN` stencil or the `DDN` stencil), the center channel would be connected to the other two but the left and the right channel would not be connected. Obviously, this would be inconsistent behaviour, which is why we don't treat the colors as another spatial order but in a different fashion.

With e spatial orders, the resolution in the ith spatial order is given by a_i. With the three color channels, the dimensionality d is thus given by

$$d = 3 \cdot \prod_{i=1}^{e} a_i \,, \tag{5.28}$$

whereas for an e-xel j, the color channel values are given by R_j, G_j and B_j respectively. Whilst feasible, we define the color stencils as upgrades to the grayscale (multilayer) stencils. When referring to such a stencil s that we apply to each channel, the interactions in each channel are then denoted as $c^n_{\mathcal{I}^s_{\text{gray}}}$, whereas the internal dimensionality of this channel is of course only $\frac{d}{3} = \prod\limits_{i=1}^{e} a_i$.

This section is structured as follows. First, we look at a stencil that does not take into account any interactions between different color channels in the next paragraph. Then, we define stencils that take into account the interactions between different color channels of an e-xel in Sec. 5.2.3.1 before looking at stencils that also model interactions between different color channels of different e-xels in Sec. 5.2.3.2. In the end, the sizes of different color stencils are compared in Sec. 5.2.3.3

No Interactions between Color Channels The first possibility is to apply the desired stencil from Sec. 5.2.1 or Sec. 5.2.2 to each channel individually without modelling interactions between the different channels. Treating the color channels this way is called the *no color interactions* stencil (abbreviated `NoCol`). Let s be the stencil we want to apply, then the number of grid points for the combined stencil s+`NoCol`is given by

$$c^n_{\mathcal{I}^{s+\texttt{NoCol}}} = 3\left(c^n_{\mathcal{I}^s_{\text{gray}}} - 1\right) + 1 = 3c^n_{\mathcal{I}^s_{\text{gray}}} - 2\,. \tag{5.29}$$

Thus, comparing a dataset with colored images to a dataset with grayscale images but both of the same resolution, this effectively increases the number of grid points by a factor of three independent of the stencil used. None of the magenta and orange connections, but only the black connections between the same channels of the e-xels depicted in Fig. 5.10 are allowed to be included with the `NoCol` stencil.

5.2.3.1. Color Interactions Limited to each e-xel

Taking into account relations between different color channels, the first possibility is to only group color channels of an individual e-xel together. We present two such options in the following paragraphs.

Pairwise Interactions between Color Channels of each e-xel To include interactions between the channels of an e-xel j, the first approach is to include pairwise interactions $\{R_j, G_j\}$, $\{R_j, B_j\}$, $\{G_j, B_j\}$. We call this the *pairwise color interactions per e-xel* stencil (abbreviated `PairCol`). The number of those pairwise interactions is d. Thus, for the grayscale stencil s we want to apply, the number of grid points in the resulting geometry-aware sparse grid is

$$c^n_{\mathcal{I}^{s+\texttt{PairCol}}} = c^n_{\mathcal{I}^{s+\texttt{NoCol}}} + d \cdot (2^n n - 3 \cdot 2^n + 4) = 3c^n_{\mathcal{I}^s_{\text{gray}}} - 2 + d \cdot (2^n n - 3 \cdot 2^n + 4) \ . \tag{5.30}$$

As with the grayscale distance-bounded pairwise stencil, the additional number of grid points is in $\mathcal{O}\left(2^n \cdot d \cdot n\right)$, which renders this a feasible option to take into account the relation between the color channels of one e-xel. The magenta connections in Fig. 5.10 depict the interactions of the `PairCol` stencil.

Full Interactions between Color Channels of each e-xel Additional to the pairwise interactions of the channels of each e-xel j, we can also include the interaction containing all three channels $\{R_j, G_j, B_j\}$. This is called the *full color interactions per e-xel* stencil (abbreviated `FullCol`). The number of those triple interactions is $\frac{d}{3}$, so for a grayscale stencil s we apply to each color channel, the number of grid points in the resulting geometry-aware sparse grid is

$$\begin{aligned} c^n_{\mathcal{I}^{s+\texttt{FullCol}}} =& c^n_{\mathcal{I}^{s+\texttt{PairCol}}} + \frac{d}{3} \cdot \left(2^{n-1}n^2 - 7 \cdot 2^{n-1}n + 7 \cdot 2^n - 8\right) \\ =& 3c^n_{\mathcal{I}^s_{\text{gray}}} - 2 + d \cdot (2^n n - 3 \cdot 2^n + 4) + \frac{d}{3} \cdot \left(2^{n-1}n^2 - 7 \cdot 2^{n-1}n + 7 \cdot 2^n - 8\right) \\ =& 3c^n_{\mathcal{I}^s_{\text{gray}}} - 2 + \frac{d}{3} \cdot \left(2^{n-1}n^2 - 2^{n-1}n - 2^{n+1} + 4\right) \ . \end{aligned} \tag{5.31}$$

In contrast to `PairCol`, the rate of growth of the number of grid points for `FullCol` is in $\mathcal{O}\left(2^n \cdot d \cdot n^2\right)$, which is also an acceptable approach to deal with color images. Grouping together each set fully connected with magenta lines in Fig. 5.10 would result in the `FullCol` stencil.

5.2.3.2. Color-Interactions between Different e-xels

Instead of limiting the grouped color channels to the same e-xels, we can also group together different color channels of different e-xels. We present several such stencils in the following.

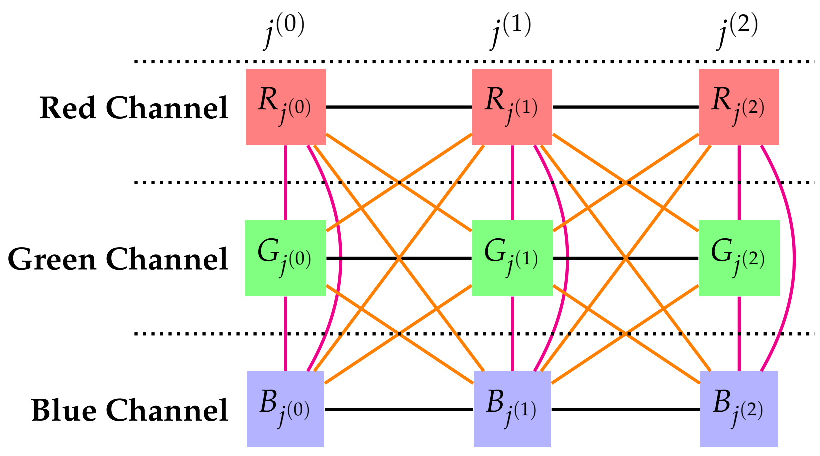


Figure 5.10.: For three e-xels $j^{(0)}$, $j^{(1)}$, and $j^{(2)}$ with three color channels each, different color channel stencils are shown. The lines in magenta denote the pairwise interactions from the `PairCol` stencil. With the `FullCol` stencil, the groups connected in magenta would form interactions of size three. If `NoCol` is chosen, the black lines connect the e-xels inside of the channels via some pairwise stencil s. The orange lines are included within the `PCPSof(`s`)Col` stencil.

Pairwise Cross-interactions between Color Channels for Pairwise Stencils So far, we only considered to include the interactions between different color channels if they belong to the same e-xel. Now, we also want to include interactions between different color channels of related e-xels. To this end, we extend pairwise stencils s such as the e-`DBP`-r stencil or the `APLay` stencil to colors by defining the *pairwise color cross-interactions for pairwise stencil* s stencil (abbreviated `PCPSof(`s`)Col`) as

$$\mathcal{I}^{\texttt{PCPSof(}s\texttt{)Col}} := \left\{ \left\{ C_j^{(1)}, C_{j'}^{(2)} \right\} \;\middle|\; (j, j) \in [d]^2, \right. \\ \left. \left(C^{(1)}, C^{(2)} \right) \in \{R, G, B\}^2, \{j, j'\} \in \mathcal{I}^s \right\}^* . \tag{5.32}$$

For each pair of e-xels j and j' that are grouped together in a grayscale image with the stencil s, there are now nine possible pairs of color channels, which are grouped together in the `PCPSof(`s`)Col` stencil. Additionally, the pairs from the `PairCol` stencil are also included in the `PCPSof(`s`)Col` stencil. The number of grid points $c^n_{\mathcal{I}^{\texttt{PCPSof(}s\texttt{)Col}}}$ is thus estimated as

$$c^n_{\mathcal{I}^{\texttt{PCPSof(}s\texttt{)Col}}} \approx 9 \cdot c^n_{\mathcal{I}^{s_{\text{gray}}}} + c^n_{\mathcal{I}^{\texttt{None+PairCol}}} , \tag{5.33}$$

which implies $c^n_{\mathcal{I}^{\texttt{PCPSof(}s\texttt{)Col}}} \in \mathcal{O}\left(c^n_{\mathcal{I}^s}\right)$. Whether it is feasible to employ this stencil in practice of course depends on the problem. $s = e$-`DBP`-r with moderate e and r (such as in the special cases $(e, r) = (2, 1)$ which corresponds to `DN` for grayscale images or $(e, r) = (2, \sqrt{2})$ which corresponds to `DDN` for grayscale images) is a viable option to treat a color image. In Fig. 5.10, the orange lines denote the interactions taken in, if the e-xels are connected via the pairwise stencil s resulting in the connections drawn in black.

Full Cross-interactions between Color Channels for Pairwise Stencils As a last step, we group together all three color channels of a pair of related e-xels in a pairwise stencil s. Although we only consider pairs of related e-xels, this already implies an initial level of at least 7 to initially have the effect of the resulting interaction present in the model. The number of grid points at this level is not only large because of the interactions of size six. It also grows due to all the partial interactions generated by the subsets of all larger interactions, which are pulled in to form a valid interaction set. However, for the sake of completeness, we define the *full color cross-interactions for pairwise stencil s* stencil (abbreviated `FCPSof(s)Col`) as

$$\mathcal{I}^{\texttt{FCPSof}(s)\texttt{Col}} := \left\{ \left\{ R_j, G_j, B_j, R_{j'}, G_{j'}, B_{j'} \right\} \;\middle|\; (j, j) \in [d]^2, \{j, j'\} \in \mathcal{I}^s \right\}^* . \tag{5.34}$$

and make note, that $c^n_{\mathcal{I}^{\texttt{FCPSof}(s)\texttt{Col}}} \in \mathcal{O}\left(c^n_{\mathcal{I}^s} \cdot n^4\right)$.

Further Stencils for Color Images Going even further than the `PCPSof(s)Col` stencil would mean to group together all color channels from a set of related e-xels e.g. given by the e-`CCDB`-r stencil, the e-`XCDB`-r stencil, or the `AFLay` stencil. However, grouping together all three color channels of k e-xels results in an interaction of size $3k$. In order for this interaction to be present in the geometry-aware sparse grid, the initial level needs to be at least $3k + 1$. Each subspace added to such an interaction is of size 2^{3k}. Those numbers illustrate, that we need not bother with interactions of this size because the resulting geometry-aware sparse grids are too large to handle.

Instead, if one wishes to apply a tupled grayscale stencil on color image data, it is of course possible to still model the relations between different channels of different e-xels additionally with one of the color stencils presented above. The resulting geometry-aware sparse grid is then given by the union of the applied stencils.

5.2.3.3. Comparison of Color Stencils

In Fig. 5.11, the number of points in the regular sparse grid is compared to the number of points of the geometry-aware sparse grid resulting from `DN` together with different color stencils for a dataset consisting of 32×32 color channel images. With a total number of $1.89 \cdot 10^7$ grid points at level 3, the regular sparse grid is obviously not feasible to employ. The stencils including interactions of at most two e-xels are of 42,241 grid points (`NoCol`), 54,529 grid points (`PairCol`) and 102,145 grid points (`PCPSof(DN)Col`) at level 3, which can be handled with enough computational power on modern machines. However, we also notice that `FCPSof(DN)Col` consists already of $7.15 \cdot 10^7$ grid points at level 7, which is the minimum level where the interactions of size six are fully taken into account. We are not able to employ this stencil with the computational power we have at hand.

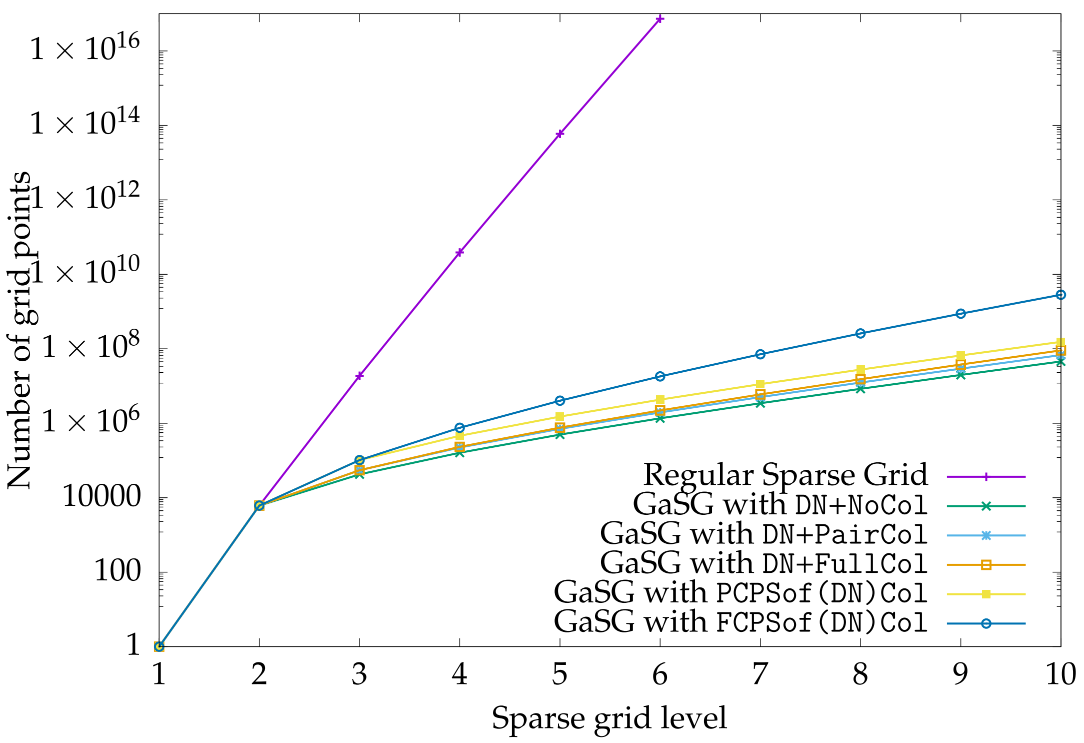
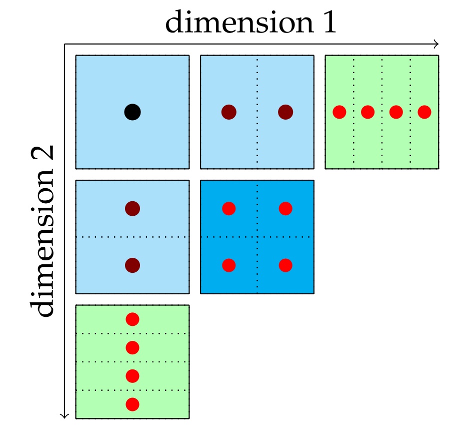
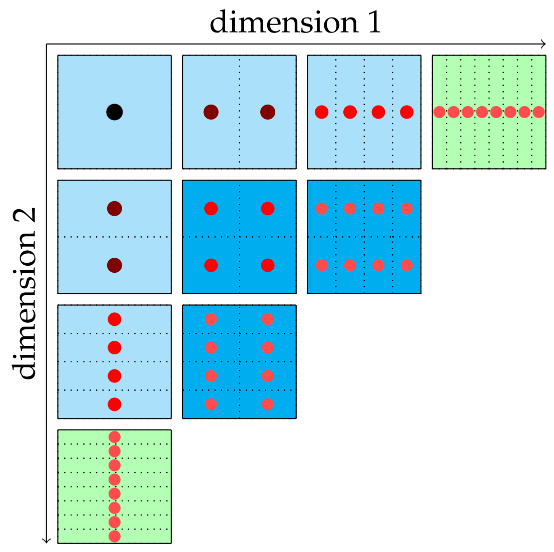


Figure 5.11.: Comparison of the number of grid points in a regular sparse grid vs. geometry-aware sparse grids for a 32×32 color image dataset with different color stencils. After level 2, the number of points in the regular sparse grid is not feasible to handle anymore while the number of points in the geometry-aware sparse grids is growing much slower.
Raw data for this figure: Tab. C.21.



(a) Two-dimensional CGaSG of level 3



(b) Two-dimensional CGaSG of level 4

Figure 5.12.: In the two-dimensional case and a target interaction size of two, the subspace grids colored in green are omitted in the cropped version of the geometry-aware sparse grid because they are not required as ancestors for any of the points in the target interaction subspace grids (colored in dark blue). Above, this is depicted for both level 3 and level 4.

5.2.4. Cropping the Grid

We already discussed that to initially have the effect of an interaction of size t in the model, the sparse grid requires at least a level of $t+1$. Only then is the subspace grid present that has a level of 2 at the dimensions from said interaction (and 1 at the rest of the dimensions). However, such a level also pulls in lots of subspace grids of the same level-vector sum that do not necessarily contain ancestral grid points for the subspaces we really want to be part of the grid. For example, a geometry-aware sparse grid with the `FCPSof(s)Col` stencil requires a level of at least 7. This also results of subspaces with level-vectors such as $(7, 1, \ldots, 1)$, $(1, 7, \ldots 1)$ and so forth to be pulled in the grid. None of those subspaces contains hierarchical ancestors for the larger interactions, which is why they are not mandatory to be present in the final grid. Truly, the subspaces where only one level-entry is higher than 1 are not even the major problem, but more so those where the level is higher than 1 at multiple level-entries because there exist a huge number of such subspaces. To further coarsen the grid, we omit those subspaces, and only pull them in as soon as they are required to serve as ancestors for grid points pulled in by the target interactions of size t. This concept is visualized in Fig. 5.12 for a two-dimensional problem with a grid of level 3 and 4.

Formally, we define the cropped geometry-aware sparse grid as follows:

Definition 5.2.1
Let $\texttt{sparseGrid}_{d,n}$ be the regular sparse grid of dimension d and level n and $\mathcal{I}$ be a set of valid interactions. The maximum size of an interaction $I \in \mathcal{I}$ is denoted as $t_{\mathcal{I}}$ via

$$t_{\mathcal{I}} := \max\left\{|I| \quad \middle| \quad I \in \mathcal{I}\right\} . \tag{5.35}$$

Then, the cropped geometry-aware sparse (CGaSG) grid of level n denoted as $\left(\texttt{sparseGrid}_{d,n}\right)_{I}^{\text{crop}}$ is given by

$$\begin{aligned}\left(\texttt{sparseGrid}_{d,n}\right)_{\mathcal{I}}^{\text{crop}} := \Big\{(\boldsymbol{l}, \boldsymbol{i}) \in \left(\texttt{sparseGrid}_{d,n}\right)_{\mathcal{I}} \quad \Big| \\ \forall j \in [d] : l_j \leq \max\left\{1, 1 + n - t_{\mathcal{I}}\right\}\Big\} .\end{aligned} \tag{5.36}$$

This implies that until level $t_{\mathcal{I}}$, $\left(\texttt{sparseGrid}_{d,n}\right)_{I}^{\text{crop}}$ only consists of $\texttt{root}_d$. Starting with level $t_{\mathcal{I}} + 1$, it then contains the subspaces encoding the target interactions and the necessary ancestors. In Tab. 5.1, the number of grid points in the GaSG vs. the number of grid points in the CGaSG for several grayscale stencils at the respective level $t_{\mathcal{I}} + 1$ is compared. With the cropping technique, especially stencils containing interactions of size higher than 2 become worthwhile again due to a drastic reduction of the grid points.

Table 5.1.: Exemplary numbers comparing the grid sizes for level $t_{\mathcal{I}} + 1$ of an geometry-aware sparse grid with and without cropping for several stencils at hand of an 28×28 grayscale image dataset. For the pairwise stencils `DN`, `DDN` and `DBP-2`, the gain is marginal. For `SQ` at the first level $t_{\mathcal{I}} + 1 = 5$, we omit about 88% of the grid points and for 2-`XCDB`-1 at level $t_{\mathcal{I}} + 1 = 6$, the omission is at about 95% making this stencil a feasible option again.

Stencil	Initial level	GaSG size	CGaSG size
`DN`	3	10,753	7,617
`DDN`	3	16,585	13,449
`DBP-2`	3	22,409	19,273
`SQ`	5	400,441	48,441
`2-XCDB-1`	6	2,934,697	148,265

5.3. Spatial Adaptivity

When spatially refining a sparse grid, the most common strategy is to add all $2 \cdot d$ hierarchical children of a refinement candidate and all of their hierarchical ancestors to the grid. In the context of geometry-aware sparse grids, we propose a strategy, which decreases the number of grid points that are added for each refinement candidate. Namely, for a geometry-aware sparse grid $\mathcal{G}_{\mathcal{I}}$ and a refinement candidate $p \in \mathcal{G}_{\mathcal{I}}$, we only add the children $q \in \texttt{children}(p)$ and their hierarchical ancestors, for which holds: $I_q \in \mathcal{I}$. As a consequence, the number of children that are added via refinement is

$$2 \cdot |I_p| + 2 \cdot \left|\{I \in \mathcal{I} \quad | \quad I_p \subset I \wedge |I| = |I_p| + 1\}\right| \tag{5.37}$$

(not counting the hierarchical ancestors if some of those are still missing). In the special case where I_p is already a target interaction in $\mathcal{I}$, this simplifies to $2 \cdot |I_p|$. The number of dimensions j, in which hierarchical ancestors of the child $q = (\boldsymbol{l}, \boldsymbol{i})$ may need to be added to the grid as well is limited by $|I_q| - 1$, namely all dimensions in I_q except for the one in which q is the child of p. The number of missing ancestors then depends on the levels l_j in said dimensions j of q (and p) and is at most $-1 + \prod\limits_j l_j$.

For example, in case of the pairwise stencil `DN`, if I_p is already a pairwise interaction ($|I_p| = 2$), the number of children per Eq. 5.37 is four. If $|I_p| = 1$, it is at most 10, due to a maximum of four other pixels connected to I_p. We want to emphasize that those numbers are constant for all stencils (with potential parameters fixed as in parameter dependent stencils such as e-`DBP`-r and others), independent of the dimensionality of the problem. Thus, with this proposed refinement strategy, the growth of the grid size upon refining one grid point is no longer $2 \cdot d$ but constant.

For grid coarsening, we employ the same strategies for GaSG as for regular sparse grids. Only grid points with none of their children present in the current grid are allowed to be removed from it. As long as this is the case, there is no further limitation to the coarsening process.

5.4. Dimensional Adaptive Combigrid Scheme

The grid points we omit in a GaSG (or a CGaSG) compared to a regular sparse grid are all related to entire subspace grids that are omitted. It cannot be the case, that some grid points from a specific subspace grid are contained in a GaSG but other grid points from the same subspace grid are not contained. Thus, we can construct every GaSG as the union of certain subspace grids. This leads us to the combination grid technique, because we relate the subspace grid $\texttt{subspaceGrid}_{d,\boldsymbol{l}}$ to the component grid of the same level-vector $\texttt{componentGrid}_{d,\boldsymbol{l}}$. The combination grid technique solution, which involves all component grids (possibly with coefficient 0) that are thus related to the subspace grids contained in the GaSG, is what interests us here.

This section is structured as follows: First, we investigate the components and their coefficients that are contained in the combination grid technique solution related to the GaSG in Sec. 5.4.1. Then, we investigate how dimensional adaptivity is employed specifically for GaSG in Sec. 5.4.2. Finally, we discuss how the similarity between many of the involved components of a GaSG is exploited to speed up the offline/online scheme in Sec. 5.4.3.

5.4.1. Components and their Coefficients

As formalized in Eq. 5.4, each component grid $\texttt{componentGrid}_{d,l}$ is related to the interaction $I_{\texttt{componentGrid}_{d,l}}$. If for an GaSG $\left(\texttt{sparseGrid}_{d,n}\right)_{\mathcal{I}}$, it holds that $I_{\texttt{componentGrid}_{d,l}} \in \mathcal{I}$ and $n > \|\boldsymbol{l}\|_1 - d$, we include $\texttt{componentGrid}_{d,l}$ in the combination grid technique scheme. The set of component grids $\mathcal{C}_{\mathcal{I}}^{n}$ that are included in the combination grid technique related to the GaSG $\left(\texttt{sparseGrid}_{d,n}\right)_{\mathcal{I}}$ is thus formalized as

$$\mathcal{C}_{\mathcal{I}}^{d,n} := \left\{ \texttt{componentGrid}_{d,l} \;\middle|\; I_{\texttt{componentGrid}_{d,l}} \in \mathcal{I} \wedge n > \|\boldsymbol{l}\|_1 - d \right\} . \tag{5.38}$$

With the component grids related to the regular sparse grid of same level and dimensionality given by $\mathcal{C}_{\text{reg}}^{d,n} := \left\{ \texttt{componentGrid}_{d,l} \;\middle|\; n > \|\boldsymbol{l}\|_1 - d \right\}$, the set $\mathcal{C}_{\mathcal{I}}^{n}$ is just a small subset of $\mathcal{C}_{\text{reg}}^{d,n}$ for the problems we are interested in (high dimensionality and stencils presented in Sec. 5.2).

This has also an impact on the coefficients for each component grid. Because $\mathcal{C}_{\mathcal{I}}^{n}$ is not regular, the coefficients provided in Eq. 2.51 are not applicable anymore. Instead, we obtain the coefficients via the method presented in [69]. For example, in the case of a pairwise stencil, the coefficient for $\texttt{componentGrid}_{d,l}$ with $\left|\left\{ l_j \;\middle|\; l_j > 1 \right\}\right| = 2$, is either 1, -1 or 0. Only for components with $\left|\left\{ l_j \;\middle|\; l_j > 1 \right\}\right| \leq 1$ are there coefficients of higher absolute value than 1.

5.4.2. Dimensional Adaptivity

With the combination grid technique at hand, we have a green light to employ dimensional adaptivity as well. Depending on the score assigned to each (refinable) component grid in the set of components currently indexed, we choose the components we want to refine. This score is obtained via refinement indicators, which inherently depend on the problem class we are tackling. Then, we add the components that have higher resolutions in certain dimensions. Which dimensions we want to resolve higher depends on the refinement strategy. Starting with $\mathcal{C}_{\mathcal{I}}^{n}$, we want to extend this set with more components. Since the result is not regular anymore, we call it $\mathcal{C}_{\mathcal{I}}$, whereas $\mathcal{C}_{\mathcal{I}}^{n} \subseteq \mathcal{C}_{\mathcal{I}}$ holds.

Refinement Indicators Refinement indicators for density estimation estimate the error of each singular component grid solution by computing the l2-norm of all surpluses of grid points at the finest level in that component grid. Higher values indicate, that the solution at this component has not converged yet.

For classification based on density estimation, we instead look at the Bayes classifier built up by the same components from all classes. Then, we compute the classification accuracy for each component and refine those that perform worst.

Refinement Strategy The refinement strategy consists of determining the refinable components, choosing the components we want to refine based on the employed refinement indicator and then adding the higher-resolved components in those dimensions we are interested in.

For $\texttt{componentGrid}_{d,l}$ already included in $\mathcal{C}_{\mathcal{I}}$, the higher-resolved components we are interested in are given by

$$\begin{aligned}\texttt{candidates}_{\mathcal{I}}\left(\texttt{componentGrid}_{d,l}\right) := \Big\{\texttt{componentGrid}_{d,l'} \ \Big| \\ \left(\exists j \in [d] : l'_j = l_j + 1 \wedge \left(\forall j' \in [d] : j = j' \vee l'_j = l_j\right)\right) \wedge I_{\texttt{componentGrid}_{d,l'}} \in \mathcal{I}\Big\} \ .\end{aligned} \tag{5.39}$$

$\texttt{componentGrid}_{d,l}$ is refinable, if $\texttt{candidates}_{\mathcal{I}}\left(\texttt{componentGrid}_{d,l}\right) \not\subset \mathcal{C}_{\mathcal{I}}$. For all refinable components, the score is calculated via the refinement indicator. To determine which components to actually refine, we take a fixed number of components or a percentage of the included components based on the components' ranking by the score from the refinement indicator. Alternatively, we take the components exceeding a certain threshold of the score. For a component, we want to refine, we add $\texttt{candidates}_{\mathcal{I}}\left(\texttt{componentGrid}_{d,l}\right) \setminus \mathcal{C}_{\mathcal{I}}$ to $\mathcal{C}_{\mathcal{I}}$. Also, we need to ensure that all of their coarser-resolved predecessors are also indexed in $\mathcal{C}_{\mathcal{I}}$. For $\texttt{componentGrid}_{d,l'} \in \texttt{candidates}_{\mathcal{I}}\left(\texttt{componentGrid}_{d,l}\right)$, those predecessors are given by

$$\left\{\texttt{componentGrid}_{d,\boldsymbol{l}^*} \ \middle| \ \boldsymbol{l}^* \neq \boldsymbol{l}' \wedge \forall j \in [d] : l^*_j \in \left[l'_j\right]\right\} \ . \tag{5.40}$$

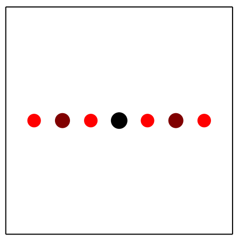
Similar to Eq. 5.37, the maximum number of component grids that are added when refining $\texttt{componentGrid}_{d,l}$ (not counting missing predecessors) is given by

$$\begin{aligned}\left|\texttt{candidates}_{\mathcal{I}}\left(\texttt{componentGrid}_{d,l}\right)\right| = \\ \left|I_{\texttt{componentGrid}_{d,l}}\right| + \left|\left\{I \in \mathcal{I} \ \middle| \ I_{\texttt{componentGrid}_{d,l}} \subset I \wedge |I| = \left|I_p\right| + 1\right\}\right|\end{aligned} \tag{5.41}$$

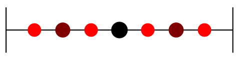
and for each $\texttt{componentGrid}_{d,l'}$ that is added to increase the resolution of $\texttt{componentGrid}_{d,l}$ in dimension j, the number of missing predecessors is at most $-1 + \prod\limits_{j' \neq j} l'_{j'}$.

5.4.3. Fast Offline Phase

When employing the combination grid technique with GaSG for the problems we have in mind, still lots of different components (differentiable by their level-vectors) are involved. For all of those, we want to obtain the matrix decomposition related to Eq. 3.11 as discussed in Sec. 3.2.3.3. However, the number of possibilities of component-grid vectors formed by different dimensionalities and distribution of the levels over the dimensions is quickly getting out of hand for the high dimensionalities we are looking at. This is why, in this section, we develop a method to derive the matrix decompositions of component grids with very large level-vectors from other component grids with much smaller level-vectors. The implementation of these concepts into SG++ have been the topic of a bachelor project [33].



(a) The two-dimensional grid componentGrid$_{2,(3,1)}$



(b) The one-dimensional grid componentGrid$_{1,(3)}$

Figure 5.13.: Embedding componentGrid$_{2,(3,1)}$ into a one-dimensional space by removing the 1 from its level-vector yields componentGrid$_{1,(3)}$.

5.4.3.1. Embedding Grids into Lower Dimensionality

The level-vectors of the components in $\mathcal{C}_{\mathcal{I}}$ are of length d. Lots of their entries are ones. Adding a 1 to a level-vector $\boldsymbol{l} = (l_1, \ldots l_d)$ as $\boldsymbol{l'} = (l_1, \ldots, l_d, 1)$ embeds componentGrid$_{d,\boldsymbol{l}}$ into the $d+1$-dimensional space. Then, in this new dimension $d+1$, the level of all grid points of componentGrid$_{d+1,\boldsymbol{l'}}$ is 1, and there are no grid points with a different level in this dimension. In reverse, we can also reduce the dimensionality by deleting a 1 from a level-vector. This is shown at the example of embedding componentGrid$_{1,(3)}$ into a second dimension by extending it to componentGrid$_{2,(3,1)}$ in Fig. 5.13. We now investigate what happens to $\boldsymbol{A}$ and the matrix decompositions of $\boldsymbol{A}$ presented in Sec. 3.2.3.3, when we do this.

In this section, we consider $\boldsymbol{A}$ without the regularization, thus $\boldsymbol{A} = \boldsymbol{R}$. We remember from Eq. 3.7, that $R_{ij} = \int_\Omega \varphi_i(\boldsymbol{x})\varphi_j(\boldsymbol{x})\mathrm{d}\boldsymbol{x}$. For the linear, modified linear and kinked linear basis functions (ref. Sec. 2.2), it holds:

$$R_{ij} = \prod_{k=1}^{d} \int_0^1 \varphi^{1\mathrm{d}}_{\mathtt{proj}_{i,k}}(x) \cdot \varphi^{1\mathrm{d}}_{\mathtt{proj}_{j,k}}(x)\mathrm{d}x\,, \tag{5.42}$$

whereas $\varphi^{1\mathrm{d}}$ stands for the one-dimensional basis function of the respective type (linear, modlinear or kinklinear). Now, when adding a 1 to the level-vector and thereby extending the dimensionality from d to $d+1$, Eq. 5.42 becomes

$$\begin{aligned} R_{ij} &= \prod_{k=1}^{d+1} \int_0^1 \varphi^{1\mathrm{d}}_{\mathtt{proj}_{i,k}}(x) \cdot \varphi^{1\mathrm{d}}_{\mathtt{proj}_{j,k}}(x)\mathrm{d}x \\ &= \left(\int_0^1 \varphi^{1\mathrm{d}}_{\mathtt{proj}_{i,d+1}}(x) \cdot \varphi^{1\mathrm{d}}_{\mathtt{proj}_{j,d+1}}(x)\mathrm{d}x \right) \cdot \prod_{k=1}^{d} \int_0^1 \varphi^{1\mathrm{d}}_{\mathtt{proj}_{i,k}}(x) \cdot \varphi^{1\mathrm{d}}_{\mathtt{proj}_{j,k}}(x)\mathrm{d}x\,. \end{aligned}$$

Per definition of dimension $d+1$, the projection of any point p via $\mathtt{proj}_{p,d+1}$ to dimension 1 is the one-dimensional root $\mathtt{root}^1$. The value of this specific inner product is known per Eq. 2.35, Eq. 2.40 and Eq. 2.46:

$$r := r^{\text{type}}_{\texttt{root}^1,\texttt{root}^1} = \begin{cases} \frac{1}{3}, & \text{type} = \text{linear}, \\ 1, & \text{type} = \text{modlinear}, \\ 1, & \text{type} = \text{kinklinear}. \end{cases} \tag{5.43}$$

This means, when extending the dimensionality from d to $d+1$ by adding a 1 to the level-vector of the grid, each entry in $\boldsymbol{A}$ has to be multiplied with r to obtain the system matrix of the new grid $\boldsymbol{A'}$. Going one step further and extending the dimensionality from d to $d+d'$ by adding d' 1s to the level-vector of the grid, each entry in $\boldsymbol{A}$ has to be multiplied with $r^{d'}$. We write this as

$$\boldsymbol{A'} = r^{d'} \cdot \boldsymbol{A}. \tag{5.44}$$

Note, that for the modified linear and the kinked linear basis functions, it holds that $\boldsymbol{A'} = \boldsymbol{A}$ and thus, no computation is required. For the linear case, we now look at how the factor $r^{d'} = 3^{-d'}$ translates to the Cholesky decomposition and the tridiagonal decomposition.

Cholesky Decomposition With $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\mathsf{T}$, we write:

$$\boldsymbol{A'} = r^{d'} \cdot \boldsymbol{A} = 3^{-d'} \cdot \boldsymbol{L}\boldsymbol{L}^\mathsf{T} = \left(3^{-\frac{d'}{2}}\boldsymbol{L}\right) \cdot \left(3^{-\frac{d'}{2}}\boldsymbol{L}\right)^\mathsf{T}. \tag{5.45}$$

The scalar multiplication retains the triangular form. Thus, the Cholesky factor $\boldsymbol{L'}$ of $\boldsymbol{A'}$ is obtained from $\boldsymbol{L}$ via $\boldsymbol{L'} = 3^{-\frac{d'}{2}}\boldsymbol{L}$ which means that we multiply every element of the Cholesky factor $\boldsymbol{L}$ with $3^{-\frac{d'}{2}}$ to obtain $\boldsymbol{L'}$. However, as discussed in Sec. 3.2.3.3.2, it is not computationally feasible to update the regularization parameter λ once $\boldsymbol{A}$ has been factorized to the Cholesky factor. Since we are still operating on $\boldsymbol{A} = \boldsymbol{R}$, this technique is only applicable with the Cholesky decomposition as long as $\lambda = 0$.

Tridiagonal Decomposition With $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^\mathsf{T}$, we write:

$$\boldsymbol{A'} = r^{d'} \cdot \boldsymbol{A} = 3^{-d'} \cdot \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^\mathsf{T} = \boldsymbol{Q}\left(3^{-d'} \cdot \boldsymbol{T}\right)\boldsymbol{Q}^\mathsf{T}. \tag{5.46}$$

Thus, the tridiagonal decomposition $\boldsymbol{Q'}$ and $\boldsymbol{T'}$ is obtained from $\boldsymbol{Q}$ and $\boldsymbol{T}$ via $\boldsymbol{Q'} = \boldsymbol{Q}$ and $\boldsymbol{T'} = 3^{-d'} \cdot \boldsymbol{T}$. To obtain $\boldsymbol{T'}$, we multiply every element of the tridiagonal matrix $\boldsymbol{T}$ with $3^{-d'}$. This is valid, because the scalar multiplication retains the tridiagonal form. Other possibilities would be feasible too, we could for example also incorporate the factor into $\boldsymbol{Q}$ as well or split it over both factors. All of those possibilities are valid, but we choose to only incorporate it into $\boldsymbol{T}$ because accessing its elements is cheaper than accessing the elements from $\boldsymbol{Q}$. In contrast to the Cholesky decomposition, it is of no consequence that $\lambda = 0$ in $\boldsymbol{A}$ so far, because at this stage, we are still able to add λ to $\boldsymbol{T}$ and follow the procedure described in Sec. 3.2.3.3.4 to optimize the regularization term. Even if $\boldsymbol{A}$ would have already contained a $\lambda \neq 0$, we can subtract it from $\boldsymbol{T}$ before applying the factor of $3^{-d'}$ to each element and subsequently adding λ again.

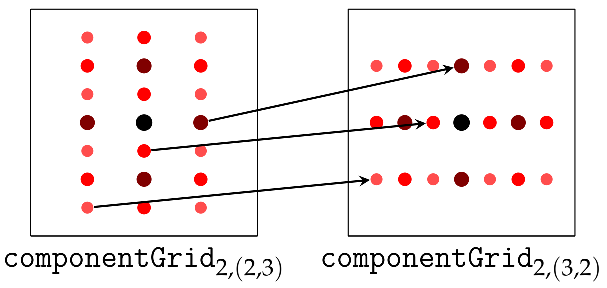


Figure 5.14.: Symmetry between the components `componentGrid`$_{2,(2,3)}$ and `componentGrid`$_{2,(3,2)}$. Exemplary, some source points and their permuted targets are connected via arrows.

5.4.3.2. Exploiting Component Grid Symmetries

Apart from embedding grids into lower dimensional spaces to derive them from one another, we also exploit symmetries of the component grids, which transfer from their level-vectors to the points in the grid. For example, the symmetry between `componentGrid`$_{2,(2,3)}$ and `componentGrid`$_{2,(3,2)}$ is visualized in Fig. 5.14. In general, if $\pi\colon [d] \to [d]$ is a permutation on the level-vector $\boldsymbol{l}^{\text{comp1}}$ of a component grid such that $\forall j \in [d] : l_j^{\text{comp2}} = \pi\left(l_j^{\text{comp1}}\right)$, then there is a symmetry between the grid points of `componentGrid`$_{d,\boldsymbol{l}^{\text{comp1}}}$ and `componentGrid`$_{d,\boldsymbol{l}^{\text{comp2}}}$ as well. To simplify the notation, we define a function that permutes the level and index vectors of a grid point directly, as

$$\begin{aligned} \pi^{\text{point}}\colon \mathbb{G}_d &\to \mathbb{G}_d\,, \\ (\boldsymbol{l},\boldsymbol{i}) &\mapsto \left(\left(\pi\left(l_1\right),\dots \pi\left(l_d\right)\right),\left(\pi\left(i_1\right),\dots \pi\left(i_d\right)\right)\right)\,, \end{aligned} \tag{5.47}$$

so the point $p \in$ `componentGrid`$_{d,\boldsymbol{l}^{\text{comp1}}}$ corresponds to $\pi^{\text{point}}\left(p\right) \in$ `componentGrid`$_{d,\boldsymbol{l}^{\text{comp2}}}$.

We are now aiming at transferring this permutation to the system matrices $\boldsymbol{A}^{(1)}$ of `componentGrid`$_{d,\boldsymbol{l}^{\text{comp1}}}$ and $\boldsymbol{A}^{(2)}$ of `componentGrid`$_{d,\boldsymbol{l}^{\text{comp2}}}$. For that, we need to know how the N grid points of a specific d-dimensional component grid with given level-vector $\boldsymbol{l}$ are serialized to $[N]$. We denote this serialization function as $s_{d,\boldsymbol{l}}\colon \mathbb{G}_d \to [N]$, which maps a grid point to its serialized index. Obviously, $s_{d,\boldsymbol{l}}$ is implementation specific and can be chosen arbitrarily (as long as it's deterministic). This is the case for SG++ where we implemented this technique. With $s_{d,\boldsymbol{l}}$ at hand (and thus also its inverse $s_{d,\boldsymbol{l}}^{-1}$), we construct a permutation matrix $\boldsymbol{P} \in \{0,1\}^{N\times N}$ such that

$$\boldsymbol{A}^{(2)} = \boldsymbol{P}\boldsymbol{A}^{(1)}\boldsymbol{P}^{\mathsf{T}} \tag{5.48}$$

with

$$P_{ij} = \begin{cases} 1\,, & s_{d,\boldsymbol{l}}^{-1}\left(j\right) = \pi^{\text{point}}\left(s_{d,\boldsymbol{l}}^{-1}\left(i\right)\right)\,, \\ 0\,, & \text{else}\,. \end{cases} \tag{5.49}$$

We now investigate how this translates to the Cholesky decomposition and to the tridiagonal decomposition.

Cholesky Decomposition For $\boldsymbol{A}^{(1)} = \boldsymbol{L}^{(1)}\boldsymbol{L}^{(1)^\mathsf{T}}$, we obtain with Eq. 5.48:

$$\boldsymbol{A}^{(2)} = \boldsymbol{P}\boldsymbol{L}^{(1)}\boldsymbol{L}^{(1)^\mathsf{T}}\boldsymbol{P}^\mathsf{T} = \left(\boldsymbol{P}\boldsymbol{L}^{(1)}\right) \cdot \left(\boldsymbol{P}\boldsymbol{L}^{(1)}\right)^\mathsf{T} . \tag{5.50}$$

Unfortunately, $\boldsymbol{P}\boldsymbol{L}^{(1)}$ is not of lower triangular form anymore. Instead, the applied permutation results in $\mathcal{O}\left(N^2\right)$ non-zeros above the main diagonal in the worst case. Transforming this result back to lower triangular form with givens rotations costs $\mathcal{O}\left(N^3\right)$ operations, which is why we cannot benefit from the symmetric grid property when employing the Cholesky decomposition.

Tridiagonal Decomposition For $\boldsymbol{A}^{(1)} = \boldsymbol{Q}^{(1)}\boldsymbol{T}^{(1)}\boldsymbol{Q}^{(1)^\mathsf{T}}$, we obtain with Eq. 5.48:

$$\boldsymbol{A}^{(2)} = \boldsymbol{P}\boldsymbol{Q}^{(1)}\boldsymbol{T}^{(1)}\boldsymbol{Q}^{(1)^\mathsf{T}}\boldsymbol{P}^\mathsf{T} = \left(\boldsymbol{P}\boldsymbol{Q}^{(1)}\right) \cdot \boldsymbol{T}^{(1)} \cdot \left(\boldsymbol{P}\boldsymbol{Q}^{(1)}\right)^\mathsf{T} . \tag{5.51}$$

Being a permutation matrix, $\boldsymbol{P}$ is orthogonal and the product $\boldsymbol{P}\boldsymbol{Q}^{(1)}$ is orthogonal too. Thus, we obtain the factors $\boldsymbol{Q}^{(2)}$ and $\boldsymbol{T}^{(2)}$ of $\boldsymbol{A}^{(2)}$ via $\boldsymbol{Q}^{(2)} = \boldsymbol{P}\boldsymbol{Q}^{(1)}$ and $\boldsymbol{T}^{(2)} = \boldsymbol{T}^{(1)}$. Note, that in contrast to Sec. 5.4.3.1, the regularization parameter λ is allowed to be incorporated in $\boldsymbol{A}$ via $\boldsymbol{A}^{(1)} = \boldsymbol{R}^{(1)} + \lambda\boldsymbol{I}$, because permuting both the rows and the columns in $\boldsymbol{A}^{(1)}$ keeps all diagonal entries of $\boldsymbol{A}^{(1)}$ also on the diagonal of $\boldsymbol{A}^{(2)}$. However, since we employ this technique only with the tridiagonal decomposition, it doesn't hurt to factorize $\boldsymbol{A}^{(1)}$ with $\lambda = 0$ in both the lower dimensional embedding and the grid symmetry case, because the desired λ can still be incorporated efficiently into the decomposition at the early stages of training as discussed in Sec. 3.2.3.3.4.

5.4.3.3. Equivalence Classes of Component Grids

With the techniques presented in Sec. 5.4.3.1 and Sec. 5.4.3.2, we now define equivalence classes of component grids, whose decompositions are efficiently derived from each other. Let the set of valid level-vectors $\mathcal{L}$ be given by

$$\mathcal{L} := \left\{(l_1, \ldots, l_d) \;\middle|\; d \in \mathbb{N} \wedge \forall j \in [d] : l_j \in \mathbb{N}\right\} . \tag{5.52}$$

Two operations on a level-vector retain the equivalence class:

1. Adding and removing levels of 1 at arbitrary positions in the level-vector.
2. Permuting the elements of the level-vector.

This leads us to the set of equivalence class representatives $\mathcal{L}^{\text{rep}}$ as

$$\begin{aligned}\mathcal{L}^{\text{rep}} :=& \{(1)\} \\ & \cup \left\{(l_1, \ldots, l_d) \in \mathcal{L} \;\middle|\; (\forall i \in [d] : l_i \neq 1) \wedge \left(\forall i, j \in [d] : i < j \Rightarrow l_i \leq l_j\right)\right\} .\end{aligned} \tag{5.53}$$

The special equivalence class $[(1)]$ is then given by

$$[(1)] := \{(l_1, \dots, l_d) \in \mathcal{L} \quad | \quad \forall i \in [d] : l_i = 1\} \; . \tag{5.54}$$

For all other elements of $\mathcal{L}$, let $\texttt{stripOnes}\colon \mathcal{L} \setminus [(1)] \to \mathcal{L} \setminus [(1)]$ be the function that removes all 1s from a level-vector. Then, the equivalence class for $(l_1, \dots l_d) \in \mathcal{L}^{\text{rep}} \setminus \{(1)\}$ is given by

$$\begin{aligned}[(l_1, \dots l_d)] := \Big\{ \hat{\boldsymbol{l}} \in \mathcal{L} \setminus [(1)] \quad \Big| \quad & (l'_1, \dots l'_d) = \texttt{stripOnes}\left(\hat{\boldsymbol{l}}\right) \\ & \wedge \exists \text{ permutation } \pi\colon [d] \to [d] : \forall j \in [d] : l_j = l'_j \Big\} \; . \end{aligned} \tag{5.55}$$

The equivalence class representatives are chosen so that they don't contain 1s and their entries are sorted in ascending order. With the techniques presented in this section, it is now sufficient to decompose $\boldsymbol{A}$ only for the component grids in $\mathcal{L}^{\text{rep}}$ (but separately for each basis function type). This drastically reduces the number of matrix decompositions we have to perform and save in order to profit from the offline/online scheme in the context of the combination grid technique for geometry-aware sparse grids.

5.5. Applications in Image Classification

To evaluate the performance of the geometry-aware sparse grids, we employ them for several image classification benchmark datasets: MNIST [62], CIFAR-10 [60] and Fashion-MNIST [105]. The choice of those datasets allows us to cover all the variants of stencils we discussed: grayscale, multilayer and color. An overview of the datasets is given in Tab. 5.2. MNIST and Fashion-MNIST are quite similar in their characteristics.

Table 5.2.: Characteristics of the image classification benchmark datasets under investigation.

Dataset	Channels	Resolution	Dim.	#classes	$\lvert\mathcal{M}_{\texttt{train}}\rvert$	$\lvert\mathcal{M}_{\texttt{test}}\rvert$
MNIST	grayscale	28×28	784	10	60,000	10,000
Fashion-MNIST	grayscale	28×28	784	10	60,000	10,000
CIFAR-10	RBG	32×32	3,072	10	50,000	10,000

While MNIST consists of images of handwritten digits, Fashion-MNIST consists of images of clothes such as shirts or pants. With MNIST being *the* standard image classification benchmark dataset, Fashion-MNIST is somewhat less known but addresses some shortcomings of MNIST [20]. CIFAR-10 consists of colored images of different objects such as birds or ships. Due to it's high dimensionality, it represents a greater challenge to most image classification techniques than MNIST.

We now discuss the results of training MNIST (Sec. 5.5.1), Fashion-MNIST (Sec. 5.5.2) and CIFAR-10 (Sec. 5.5.3).

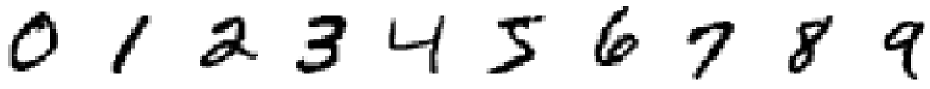


Figure 5.15.: One example data point from MNIST per class.

5.5.1. MNIST

One example data point from MNIST [62] per class is shown in Fig. 5.15. Before we show the numerical results for training MNIST with GaSG, we first investigate several aspects of how the geometry-aware sparse grids should generally be configured for image classification problems.

5.5.1.1. Parameter Tuning

The parameters we take a closer look at are the choice of the basis functions, the training dataset size and the regularization parameter λ.

Choice of the Basis Function Linear basis function do not perform well for problems with dimensionalities as we encounter them in image classification. The "pagoda"-shape of the linear basis function degenerates with increasing dimensionality. Its value is close to 0 for the majority of its support and only adopts the values close to 1 for evaluation points that are close to the supports' center in most of the dimensions. This is problematic, because the weights of the basis functions are less hierarchical increments than they already represent the function value at those points. The effect is, the weights alternate faster, leading faster to overfitting.

An equally problematic attribute of linear basis functions is the fact that they are not trivial in every dimension, independent of their level. The modified linear and the kinked linear basis functions are always constant on level 1 at the value of 1. This simplifies the evaluation drastically compared to the linear basis, as we only need to consider the evaluation in dimensions, where the level of the basis is higher than 1.

Between the modified linear basis functions and the kinked linear basis functions, there is no such different from a computational point of view, as both basis types are constant on level 1. However, although the modified linear basis functions do not degenerate with growing dimensionality per se, they are problematic for the grid points closest to the boundary, if those points are at least at level 2 in multiple dimensions. Then, the extrapolating branch of the modified linear basis adapts high values, leading to extreme function approximations near the boundaries.

When looking at the first numerical results for the classification of MNIST, we immediately see how this effects the classification accuracy.

Training Dataset Size vs. Grid Size MNIST consists of approximately the same number of training samples as the other datasets we investigate. We have already seen that the geometry-aware sparse grids easily grow to more than 10,000 points. Estimating a density with $\approx$ 6,000 data points on a grid of size $>$ 10,000 quickly leads to overfitting. This is why we employ data augmentation techniques to increase the size of $\mathcal{M}_{\mathtt{train}}$. By slightly shifting, rotating, scaling, and sometimes even mirroring an image, the object on it is still recognizable. However for the algorithm, such a transformation results in a completely different data point. Thus, we can increase the training dataset size at will by applying those transformation to samples from the original training datasets.

Tuning the Regularization Parameter λ Even when employing data augmentation, the ratio between the number of grid points and the number of data points generally indicates that regularization is important. A study of the parameter λ, which was done during a bachelor project [102], shows, that the value of $\lambda = 1$ is generally a good choice. Due to computational limitations of the hardware we have at hand, we did not investigate whether the need for regularization might be mitigated by generating an augmented dataset with a size of a high multiple of the original dataset.

5.5.1.2. Training

Naively training the dataset with geometry-aware sparse grids (but also with regular sparse grids, for that matter) yields unsatisfactory results as the classification accuracy with the `DN` stencil and a regularization value of $\lambda = 1$ is merely at 45.16% when employing kinked linear basis functions (30.28% with modified linear basis functions). An investigation done in a bachelor project [102] revealed that the class of digit 1 is largely dominating the domain, so that many test samples are wrongly classified as 1. Consecutively, the digits with the lowermost variance always dominate the classifier in the order of 1, 9, 7, 4, 6, 8, 3, 5, 2, and 0. This insight was easily obtained because of the explainable structure of the data model, generated via grid-based density estimations building up a Bayes classifier. We emphasize that the deep insight into the domination effect and its explanation via the surpluses at the basis functions would not have been easily possible with most other methods competing in image classification.

Taking into account only digits with similar variance leads to better results. If, for example, we limit the classes to the digits 6 and 8, we obtain a classification accuracy of 95.44% when employing the `DN` stencil with kinked linear basis functions (94.82% with the modified linear basis). Interestingly, this accuracy decreases when a larger stencil is chosen. For example, the accuracy with the `DDN` stencil (which increases the grid size to 16,585 grid points) is only at 93.98%. Similar results are observed, when the dataset is coarsened to an image resolution of 16×16 or 8×8, where we can employ even larger stencils such as the `SQ` stencil. While the classification accuracy generally decreases with decreasing image resolution, it also decreases with a larger grid.

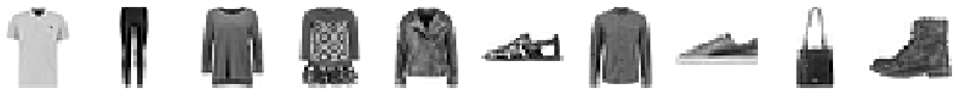


Figure 5.16.: One exemplary data point from Fashion-MNIST for each of the classes t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot.

Although limiting the classes like that does not give us a classifier for the whole dataset, it still shows the general applicability of GaSG for image classification.

5.5.2. Fashion-MNIST

At the example of Fashion-MNIST [105], we want to take the multilayer stencils presented in Sec. 5.2.2 to the test. One example data point from Fashion-MNIST per class is shown in Fig. 5.16. In order to be able to apply a multilayer stencil, we iteratively coarsen the images and append the resulting coarser data points to the original ones, in order of decreasing resolution. Starting with images of resolution 28×28, we obtain the resolutions 14×14, 7×7, 4×4, 2×2, and 1×1. Appending all those pixels yields a total dimensionality of 1,050. The results are shown in Tab. 5.3. The classification

Table 5.3.: Classification results of Fashion-MNIST with different configurations

Stencil	Resolution	Dim.	Classes	Classification accuracy
`DN`	28×28	784	all	47.73%
`APLay`	$28 \times 28, 14 \times 14, 7 \times 7, 4 \times 4, 2\times, 1 \times 1$	1050	all	50.01%
`NPLay`	$28 \times 28, 14 \times 14, 7 \times 7, 4 \times 4, 2\times, 1 \times 1$	1050	all	48.99%

accuracy is a better than with MNIST, but still far away from the rates that other techniques such as convolutional neural networks achieve on this dataset. Employing the data hierarchy with the multilayer approach increases the classification accuracy a bit, but only in the range of some percent. At least, there is not one class that dominates the problem as it was the case with MNIST. Falsely classified samples from the test dataset are present in all classes. This indicates that GaSG could model the dataset better, if the training dataset size is larger. Whether data augmentation is a solution for this dataset remains to be investigated.

As a concluding remark for the Fashion-MNIST dataset, we want to point out that to our knowledge, this is the first time that sparse grids have been successfully applied to problems with more than 1,000 dimensions. While the classification accuracy is far from away from being satisfying, it still shows that we are to some extent able to model this data and that sparse grids are generally applicable to problems of such high dimensionality.

Figure 5.17.: One example data point from CIFAR-10 for each of the classes airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

5.5.3. CIFAR-10

With the CIFAR- dataset [60], we test the color stencils. One example data point from CIFAR-10 per class is shown in Fig. 5.17. The results are shown in Tab. 5.4. First, we

Table 5.4.: Classification results of CIFAR-10 with different configurations

Stencil	Classification accuracy
`NoCol+DN`	41.71%
`PairCol+DN`	44.51%
`FullCol+DN`	45.95%

want to mention that for CIFAR-10, as for Fashion-MNIST, there is not one class that dominates the classifier. The falsely classified samples are distributed evenly over the confusion matrix. Next, we observe that including the interactions between the channels increases the classification accuracy. Whereas 45.95% is not a good result compared to other classification methods, it also shows that we are able to handle sparse grids in such high dimensionalities.

6. Data Mining Pipeline

The SG++ project [79] offers state-of-the-art sparse grid methods as a software library. Started more than 10 years ago by Dirk Pflüger at Technical University of Munich, it is nowadays developed by contributors from multiple universities and institutions. It offers several numerical methods based on sparse grids, such as function interpolation, quadrature, and the approximation of partial differential equations. It also consists of a data-driven module targeting all data related problems. In this module, density estimation, classification, regression, and clustering are implemented.

As a SG++ spin-off, the offline/online scheme discussed in Sec. 3.2.3.3 has been implemented in a library called *libtool* (not to be confused with GNU Libtool) by Benjamin Peherstorfer, together with the front-end *clustc*. From a software engineering perspective, one of the first steps for this thesis was to port the offline/online scheme algorithms from *libtool* to the SG++ data-driven module to have all relevant code under one roof. In this course, the ported algorithms also profit from the algorithmic and architectural improvements of SG++. The structure of the data-driven module was then as follows:

- **Algorithms:** The mathematical data structures and algorithms such as matrix factorizations, computation of the right-hand side and solving of Eq. 3.11 are located in the `algorithms/` folder. This is where most of the code from the former *libtool* library was ported to.
- **Applications:** The implementation of learners in many different variants for density estimation, classification, regression, and clustering was located in the `applications/` folder. Due to a wild growth of methods without a vision from a software engineering perspective, many tasks such as data preprocessing, setting up the model, cross-validation and visualization were implemented in many of the applications independently. This led to long and obfuscated code as well as lots of code duplication. Application developers spent long times (re-)implementing concepts that were already present elsewhere but could not be used in other applications. One major drawback of this structure was, that it made the applications and data mining concepts unflexible to use in similar settings without again performing lots of code duplication.
- **Examples:** The interfaces for users of the applications were implemented as executables in the `examples/` folder. Usually, there existed one example for each application, which was setting up the application model by instantiating the corresponding class from the `applications/` folder. Then, the learning process

would be started with subsequent evaluation of the learner quality based on the configured test dataset.

Our next step was to encapsulate the software contributions resulting from this thesis as well as to address the problems of the structure in the data-driven module. Therefore, the *data mining pipeline* was designed as a component in the data-driven module of SG++. Over the years, the implementation of various components and models into the data mining pipeline has been the topic of multiple student projects [2, 14, 25, 33, 57, 64, 82, 86, 104]. As a final step, this pipeline is now in transition to a standalone library in the context of SG++.

In the following, the paradigms and concepts of the data mining pipeline are presented in Sec. 6.1. The components of the pipeline are the *data source*, the *scorer*, the *fitter*, the *visualizer*, and the *hyperparameter optimizer*. Those are discussed in detail in Sec. 6.2. Finally, the models implemented in the *fitter* component are subject of Sec. 6.3.

6.1. Paradigms

The pipeline is designed to be easy to use for users and developers. Users directly start to use the methods without having to write lots of configurations or even code. However, if they choose to, they have full control over the methods by specifying all desired configuration options. Developers focus on their task by encapsulated and well defined scopes for the different code parts. Optional vertical features can be included if desired.

6.1.1. User View

Our publicly funded research implies that the results of this dissertation project are made accessible for the public in a barrier-free manner. From that, we derived the main goal for the user perspective: To provide an interface to sparse grid-based learning without the users depending on domain knowledge of sparse grids and without the users having to program before executing the methods. Therefore, the data mining pipeline is run by specifying the problem in JavaScript Object Notation (JSON) format. The only mandatory arguments are the filename of the dataset from which to learn and the data mining problem class that should be applied (regression, classification, etc.) A short but functional example for such a JSON configuration is shown in Fig. 6.1. All arguments that are not specified are set to reasonable default values. That way, the user doesn't need to configure a whole lot of options unfamiliar with. However, if he chooses so, he can configure all components of the data mining pipeline in detail to fine-grainedly control its behaviour. As a result, the interface is both simple and flexible, depending on the expertise of the user.

```
{
  "dataSource": {
      "filePath": "../datasets/exampleTrain.arff",
      "batchSize": 500,
      "epochs": 10
  },
  "fitter": {
      "type": "classification",
      "gridConfig": {
          "gridType": "kinklinear",
          "level": "4"
      },
      "regularizationConfig": {
          "lambda": "1e-03"
      },
      "densityEstimationConfig": {
          "densityEstimationType": "decomposition",
          "matrixDecompositionType": "chol"
      }
  }
}
```

Figure 6.1.: Example of a data mining pipeline JSON configuration file. This configuration trains the specified dataset over 10 epochs, splitting the data into batches of 500 samples each. Classification specified as the target method, the underlying sparse grid is of level 4 with kinked linear basis functions. As regularization parameter, 10^{-3} is chosen and the density estimation for each class is tackled with the offline/online scheme involving the Cholesky decomposition.

Alternatively to executing the data mining pipeline through JSON configurations, its components are also accessible as a library in C++, Python, Java or Matlab. The function signatures are exported automatically with swig from C++ definitions to library wrappers in the other listed languages. Since those signatures are available for the whole range from high-level calls to the pipeline down to the atomic methods and classes, the user can plug the parts together at need and also use them in foreign software projects.

6.1.2. Developer View

From a developer perspective, the data mining pipeline is designed for the ability to focus on the specific task at hand. The developer doesn't have to bother with components currently not of interest. For writing fitters, the developer can focus on the model specifics without worrying about data input, parameter optimization, visualization etc. To this end, global tasks such as preprocessing and hyperparameter optimization are implemented generically so that they are applicable to all use cases of the data mining pipeline.

6.2. Components

Control of the data mining process is handled by the `SparseGridMiner` class, which is usually initiated through a `MinerFactory`. The miner initializes the required components and then runs the main program loop. This loop feeds the data batch-wise to the fitter and controls when to refine the model (possibly based on the output of the scorer). At user-defined intervals, evaluations or visualizations of the model are triggered by calls to the visualizer.

A dataset can be looped over several times by specifying multiple epochs in the configuration. During each epoch, $\mathcal{M}_{\texttt{train}}$ is read in batch-wise. After fitting a batch of data points to the model, the refinement process is triggered via the `refinement monitor`. It takes into account the scores of the model on $\mathcal{M}_{\texttt{val}}$ and the current progress of the mining process. At the end of the learning phase, the final score of $\mathcal{M}_{\texttt{test}}$ can be evaluated.

The individual components of the data mining pipeline are sketched in Fig. 6.2 and explained in the following. Those are the *data source* (Sec. 6.2.1), the *fitter* (Sec. 6.2.2), the *scorer* (Sec. 6.2.3), the *visualizer* (Sec. 6.2.4) and the *hyperparameter optimizer* (Sec. 6.2.5).

6.2.1. Data Source

The data source component handles the data input and preprocessing. It consists of methods to provide data, shuffle it, split it and transform it.

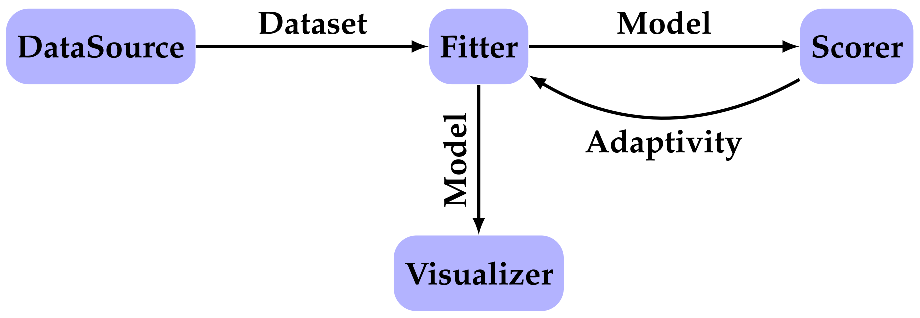


Figure 6.2.: Data mining pipeline components and the interactions between them.

Sample Provider To provide the data on a basic level, the `Sample Provider` abstracts reading in data from a specific file format or another data source. Whether the data originates from a file, a database or a stream does not matter after this point. Also the support for compressed data sources is implemented.

Data Shuffling Feeding the sorted data to the fitter might lead to problems such as lack of generalization. To prevent that, the order how the data is read in can be configured to random shuffling.

Data Splitting Before fitting the model to the data, we usually split off a validation dataset $\mathcal{M}_{\texttt{val}}$ from the training dataset $\mathcal{M}_{\texttt{train}}$. This is done by splitting off a fixed (relative) portion, such that the validation score is always calculated with the same validation dataset. Alternatively, one can configure a rotating validation dataset, which is for example useful for k-fold cross-validation. There, the training dataset is split into k parts and we take k turns with training the model. In each turn, a different part is used to validate the model whereas the data is fit to the remaining $k-1$ parts. Variables such as the regularization parameter λ can be varied across the turns to optimize them.

Data Transformation Before feeding the data to the fitter, the user can choose to apply certain transformations. The sparse grid methods require all data to reside in the unit hypercube $[0,1]^d$. For each dimension, in which the data exceeds this interval, a mapping to $[0,1]$ should be configured.

Optionally, the data is scaled to optimally cover $[0,1]^d$ using the Rosenblatt transformation [83]. The advantage of such scaled data is the better initial distribution of the grid points in the area covered by the data points.

6.2.2. Fitter

The fitter component is the heart of the data mining pipeline. Here, the data is fitted to the model. On a basic level, the fitters are divided into single-grid models and many-grid models. A single-grid model is for example a least squares regression and a many-grid model is a density estimation based classification.

Before the actual fitting is started, the grid is initialized by the fitter. To support multiple grid types concerning the basis function type and the grid layout, the `Grid Factory` creates the grid points based on the configuration provided by the user. For some models the combination grid technique is implemented, introducing an additional level in the fitter class hierarchy.

A model implementation mainly provides the methods `fit()` and `refine()`. A call to `fit()` fits a batch of data points to the model. Calling `refine()` performs a model adaption using the refinement method configured by the user.

6.2.3. Scorer

In the scorer component, the methods to score models and compute the metrics for the validation and test datasets are implemented. Available measures are the mean squared error, negative log likelihood, accuracy (for classification problems) and the residual score (for evaluating density estimation). The component definition demands that lower scores indicate better model quality than higher scores.

Based on these scores, model adaptivity is controlled. As long as the model quality is still improving when learning more batches, no action needs to be taken. As soon as the score converges, a new iteration of refinement and coarsening is triggered.

6.2.4. Visualizer

Producing graphical output for the results is out of the scope of the data mining pipeline. The visualizer component rather generates output that allows to be fed into standalone visualization tools such as Matplotlib [49] or PGF/Ti*k*Z.

On a basic level, the techniques are divided into two-dimensional visualization and higher-dimensional visualization. For the two-dimensional case, available options are to generate continuous heat maps (e.g. useful for density estimation) or discrete heat maps (e.g. useful for classification). The corresponding grid points and data points can be overlapped with the heat maps to provider further insight into the results of the mining. For higher-dimensional cases, two fundamental approaches exist: Generating two-dimensional cuts and visualizing them with the previously discussed two-dimensional techniques or performing dimensionality reduction for the output data and visualizing it in the reduced space.

For the latter, the tSNE algorithm has been implemented in a student project [2] and we refer to [65] for details. For the former, out of a total of d dimensions, two can be chosen to be free, whereas values of the remaining $d - 2$ need to be fixed. Per default, a value of 0.5 (the middle of Ω^1) is chosen for the fixed dimensions. However, other positions for cuts can be chosen. If k different values are configured, a total of $k^{d-2} \times \binom{d}{2}$ two-dimensional cuts are generated. Thus, for high dimensions, $k = 1$ is the only reasonable choice as to not encounter the curse of dimensionality.

6.2.5. Hyperparameter Optimizer

The hyperparameter optimization [8] (HPO) component is responsible to automatically find the best set of values for a given set of parameters. It was implemented into the SG++ data mining pipeline during a bachelor project [57].

Parameter Classes Several classes of parameters exist:

- **Continuous parameters** ranging on a real-valued scale. Examples for such a parameter is λ or the threshold t for the value of the score of a model that indicates whether to keep learning and refining or if convergence has been reached.
- **Discrete parameters** usually ranging on the natural numbers. Examples are the initial sparse grid level or the number of grid points to refine in each model adaption step.
- **Categorical parameters**. Examples are the basis function type or the refinement indicator.

The difference between discrete and categorical parameters is, that only for the former, an ordering is defined.

Optimization Strategies The parameter space that encompasses all possible combinations of parameters. In it, we search for an optimal parameter configuration in relation to a specific score, which is our target function. Two optimization strategies are implemented: Bayesian optimization and Harmonica [41].

Bayesian Optimization With Bayesian optimization [23], we treat the target function as a black box. Interpreting the function evaluations as data, we apply Bayesian statistics to construct a posterior probability, which we use as a predictor for unknown function values. Modelling the target function as a Gaussian process, we sequentially choose the next best guess in the parameter space, which in turn sharpens our knowledge about the unknown model score. For a detailed explanation about Gaussian process, we refer to [81]. Let it be known that we use the squared exponential kernel in the Gaussian process and use Automatic Relevance Determination to deal with the internal hyperparameters of this kernel. For the acquisition function, expected improvement [17] is used.

Harmonica Harmonica is a relatively new approach to hyperparameter optimization introduced in [41]. It operates on a Boolean representation of the hyperparameter space. The continuous hyperparameters are discretized and represented as a set of Boolean variables whereas the number of Boolean variables determines the resolution of the hyperparameter in question. The value ranges of the discrete and categorical parameters are mapped onto a set of Boolean variables as well. Starting with a high number of possible Boolean assignments, a set of randomly chosen initial hyperparameter configurations is evaluated with the chosen target score. Then, the search space is iteratively reduced. Therefore, in each round, one or multiple Boolean variables are set to a constant value or made dependent on one or multiple other Boolean variables. Then, the next round of randomly chosen configurations is drawn, only varying Boolean variables that are still free.

Discussion and Evaluation Harmonica requires lots of samples from the hyperparameter search space to draw good conclusions about correlations between hyperparameters and reducing the search space. However, multiple parameter configurations can be drawn and processed independent of each other. Thus, Harmonica is easily parallelized using any parallel architecture. In contrast, Bayesian optimization computes the score for one parameter configuration at a time. Ideas to parallelize it exist, but are not straightforward to implement and don't offer abstractions as with Harmonica. Still, our experiments show that Bayesian optimization convergences fast. To summarize, there are learning problems that highlight the strengths of both approaches and we cannot conclude that one is superior to the other.

6.3. Models

In the fitter component, single-grid models and many-grid models are implemented. We give an overview over the implementations for density estimation, classification, regression, and clustering.

Density Estimation The density estimation method discussed in Sec. 3.2 is implemented in the data mining pipeline. The linear system Eq. 3.11 is solved either using the CG method [47] or offline/online scheme using the matrix factorizations. To adapt the model to the data, surplus volume refinement is the recommended strategy. For the offline/online scheme, a parallel fitter has been implemented following Chap. 4.

Classification Classification building up on grid-based density estimation as presented in Sec. 3.3 is implemented as a many-grid model using the functionalities of the density estimation models. Thus, the classification routines profit from all features that are also available in the density estimation implementation such as choice of solver, parallel execution or exploitation of the combi-grid technique. For model adaption, the recommended refinement technique is the classification refinement strategy presented in Sec. 3.3.2.

Regression The supervised learning task of regression is implemented, following the sparse grid-based approach by [29]. The least-squares method is used to fit the model to the data. Iteratively, the sparse grid approximation is obtained via the CG method, similar to the CG method implemented for density estimation. To adapt the model to the data, the surplus refinement strategy is employed.

Clustering Similar to classification, the unsupervised learning task of clustering can be built up on grid-based density estimation as well [77]. This approach is implemented in the data mining pipeline, with surplus volume refinement as the recommended strategy to perform model adaptions. Also, hierarchical clustering using sliding thresholds is supported, as well as treating the result of the clustering as a classification model.

6.4. Summary and Outlook for the Data Mining Pipeline

The data mining pipeline paves the way for the utilization of sparse grid-based data mining methods for a broad range of users. Due to the flexible yet powerful JSON configuration interface, it is suited for both sparse grid laymen and experts. Also, the extension with more features, components and models is made easy by the well-thought software design. The next step for the pipeline is to become a related, yet independent project of SG++. Also, an integration to modern data mining frameworks such as scikit-learn or R is desirable. Development continues and currently heads into this direction with a growing user base hopefully to establish soon.

7. Conclusion

The offline/online splitting combined with spatial adaptivity offers a flexible tool to tackle problems, where the model needs to be adapted to the data. We showed how matrix decompositions are updated according to grid points being added or removed from the grid. This process allows us now to combine the strengths of both spatial adaptivity and the offline/online scheme: Reaching higher accuracies by adapting the model to the data and being fast doing so. Of course, meaningful refinement criteria are indispensable which is why we proposed a refinement strategy tailored to many-grid-based classification. With this strategy, we target specifically the areas where the result of the classifier fluctuates and we exploit the sparse grid structure doing so. This allows us to reach higher classification accuracies investing far less points than with naive refinement strategies such as surplus-based refinement. Furthermore, we discussed how both the grid-based density estimation technique as well as the employment of the Bayes classifier yield explainable data models. By investigating their surpluses, the models become transparent and traceable.

We discussed several possibilities to parallelize the learning of grid-based density estimation and classification. Those parallelization schemes scale well in both strong and weak scaling tests when working with regular sparse grids. However, spatial adaptivity procedures tend to dominate the problems and they are not subject to parallelization (yet) internally. Breaking up the spatial adaptivity and model update procedures in order to split them into parallelizable tasks would result in better scaling behaviour if we mean to refine and coarsen the grids. To tackle this, further research is necessary on how to design and implement such a scheme.

With geometry-aware sparse grids, we introduced an methodical approach on how to exploit the interactions between dimensions in problems where dimensions are varyingly related to one another, e.g. image classification. Several methods on how to incorporate those relations into a sparse grid-based model depending on the problem size and the computational power at hand have been proposed for datasets with an arbitrary number of channels. Our experiments with benchmark image classification datasets show that we are generally able to grasp the structure of the data in the model even in more than 1,000 dimensions. However, the model accuracies are still far away from what other methods in this field achieve. By employing techniques such as data augmentation and representing the hierarchy of the data in the model, we are able to gain a bit, but not enough to be competitive.

We made the results of this (publicly-funded) research are available to the public by designing and implementing the data mining pipeline based on SG++, which allows users to employ sparse grids for data driven problems. With an easy-to-use but highly configurable interface, this product is accessible for both sparse grids laymen and experts. In order to establish sparse grids-based methods in the data science community, further work should be undertaken to make the routines directly available in frameworks such as scikit-learn or R.

Acknowledgments

Throughout this thesis project, I enjoyed support from many individuals. First of all, I thank you, Hans, for the opportunity of working and researching at the chair of scientific computing and for all the guidance, trust, independence, and freedom you invested in me during the last years. For your hospitality and an amazing stay abroad at the Australian National University, I thank you, Markus.

For their companionship and for making the chair more than just a working place, I thank all my colleagues at IN5. I especially thank my sparse grid sparring partners Benjamin, Dirk, Michael, Paul, and Valeriy with whom I had the pleasure to work with over the years for many fruitful discussions and contributions that ultimately led to the shape of this thesis. Also, I'm grateful to Benjamin and Friedrich for proof-reading this work. Shout-out to my fellow SG++ developers in both Munich and Stuttgart with whom I had many fruitful scientific discussions. Also, the final state of this project would not have been reached without the contribution of many students from several TUM departments of which I thank in particular Adrian, Dmitrij, Dominik, Eric, Jan, Kilian, Michael, Nico, Samuel, Sebastian, Tim, Vincent and Vincent.

The acquisition of new skills, development of software, and the writing of this thesis has built up on free content contributed by countless people all over the world. Therefore, I want to thank all open-access paper authors, public domain contributors, free- and open-source software developers and all other people who altruistically share their work.

I wholeheartedly thank all my friends for the time we had together and your invaluable friendship. My love goes out to my parents and my sisters for raising and always supporting me, no matter what. And last but not least, thank you Lena for your love and for carrying me across the finish line.

Appendix A.

Proofs

A.1. Inner Product of Basis Functions

A.1.1. Linear Basis Functions

Let $q_1 = (l_1, i_1)$ and $q_2 = (l_2, i_2)$ be two one-dimensional grid points. We proof that Eq. 2.35 is correct. For each case, we start with

$$r^{\text{linear}}_{q_1,q_2} = \int_0^1 \varphi^{\text{1d-linear}}_{q_1}(x) \cdot \varphi^{\text{1d-linear}}_{q_2}(x)\mathrm{d}x\,.$$

Note that in case of $l_1 = l_2$, we also use $l := l_1$ as level variable and in case of $i_1 = i_2$, we also use $i := i_1$ as index variable.

- $q_1 = q_2$:

  We show that in this case, $r^{\text{linear}}_{q_1,q_2} = \frac{2^{1-l}}{3}$. The value of i is irrelevant, because φ^{linear} is translation invariant. Thus, we look at the case of $i = 1$. The support of q_1 is $\left[0, 2^{-l+1}\right]$, whereas the left half and the right half of the basis function are identical. Therefore, we compute the solution for the left part only and multiply it by 2. The integral we calculate is:

$$\begin{aligned}
r^{\text{linear}}_{q_1,q_2} =& 2 \cdot \int_0^{2^{-l}} \varphi_r(2^l x - 1)^2 \mathrm{d}x \\
=& 2 \cdot \int_0^{2^{-l}} \left(2^l x\right)^2 \mathrm{d}x \\
=& 2^{2l+1} \int_0^{2^{l-1}} x^2 \mathrm{d}x \\
=& 2^{2l+1} \left[\frac{1}{3}x^3\right]_0^{2^{-l}} \\
=& 2^{2l+1}\frac{1}{3}2^{-3l}
\end{aligned}$$

$$= \frac{2^{1-l}}{3}.$$

- $l_1 < l_2$ and $\max\left\{\frac{i_1-1}{2^{l_1}}, \frac{i_2-1}{2^{l_2}}\right\} < \min\left\{\frac{i_1+1}{2^{l_1}}, \frac{i_2+1}{2^{l_2}}\right\}$:

  We show that in this case, $r^{\text{linear}}_{q_1,q_2} = \frac{1-2^{l_1}\left|\frac{i_2}{2^{l_2}}-\frac{i_1}{2^{l_1}}\right|}{2^{l_2}}$. Since $l_1 < l_2$, the supports of q_1 and q_2 only overlap, if q_1 is an ancestor of q_2. With Eq. 2.2d, this is the case, if $\frac{i_1+1}{2^{l_1}} \leq \frac{i_2-1}{2^{l_2}}$ or if $\frac{i_2+1}{2^{l_2}} \leq \frac{i_1-1}{2^{l_1}}$, which transforms to

$$\max\left\{\frac{i_1-1}{2^{l_1}}, \frac{i_2-1}{2^{l_2}}\right\} < \min\left\{\frac{i_1+1}{2^{l_1}}, \frac{i_2+1}{2^{l_2}}\right\}. \tag{A.1}$$

  If q_2 is one of the left descendants (the left child or one of the left child's descendants), the support of q_2 overlaps only with the left half of the support of q_1 (equivalently for the right side). In this case, we need to calculate

$$\begin{aligned}
r^{\text{linear}}_{q_1,q_2} &= \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \varphi^{\text{1d-linear}}_{p_1}(x) \cdot \varphi^{\text{1d-linear}}_{p_2}(x)\mathrm{d}x \\
&= \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \varphi_l(2^{l_1}x - i_1) \cdot \varphi^{\text{1d-linear}}_{p_2}(x)\mathrm{d}x \\
&= \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(2^{l_1}x - i_1 + 1\right) \cdot \varphi(2^{l_2}x - i_2)\mathrm{d}x \\
&= \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(2^{l_1}x - i_1 + 1\right) \cdot \left(\varphi_l(2^{l_2}x - i_2) + \varphi_r(2^{l_2}x - i_2)\right)\mathrm{d}x \\
&= \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(2^{l_1}x - i_1 + 1\right) \cdot \varphi_l(2^{l_2}x - i_2)\mathrm{d}x \\
&\quad + \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(2^{l_1}x - i_1 + 1\right) \cdot \varphi_r(2^{l_2}x - i_2)\mathrm{d}x
\end{aligned}$$

$$
\begin{aligned}
&= \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} \left(2^{l_1}x - i_1 + 1\right) \cdot \left(2^{l_2}x - i_2 + 1\right) \mathrm{d}x \\
&\quad + \int_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(2^{l_1}x - i_1 + 1\right) \cdot \left(-2^{l_2}x + i_2 + 1\right) \mathrm{d}x \\
&= \int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} \left(2^{l_1+l_2}x^2 + \left(-2^{l_2}i_1 + 2^{l_2} - 2^{l_1}i_2 + 2^{l_1}\right)x + (1-i_2)(1-i_1)\right) \mathrm{d}x \\
&\quad + \int_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(-2^{l_1+l_2}x^2 + \left(2^{l_2}i_1 - 2^{l_2} + 2^{l_1}i_2 + 2^{l_1}\right)x + (1+i_2)(1-i_1)\right) \mathrm{d}x \\
&= \left[\frac{2^{l_1+l_2}}{3}x^3 + \frac{-2^{l_2}i_1 + 2^{l_2} - 2^{l_1}i_2 + 2^{l_1}}{2}x^2 + (1-i_2)(1-i_1)x\right]_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} \\
&\quad + \left[\frac{-2^{l_1+l_2}}{3}x^3 + \frac{2^{l_2}i_1 - 2^{l_2} + 2^{l_1}i_2 + 2^{l_1}}{2}x^2 + (1+i_2)(1-i_1)x\right]_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \\
&= \frac{2^{l_1+l_2}}{3}\left(\frac{i_2}{2^{l_2}}\right)^3 + \frac{-2^{l_2}i_1 + 2^{l_2} - 2^{l_1}i_2 + 2^{l_1}}{2}\left(\frac{i_2}{2^{l_2}}\right)^2 \\
&\quad + (1-i_2)(1-i_1)\left(\frac{i_2}{2^{l_2}}\right) \\
&\quad - \frac{2^{l_1+l_2}}{3}\left(\frac{i_2-1}{2^{l_2}}\right)^3 - \frac{-2^{l_2}i_1 + 2^{l_2} - 2^{l_1}i_2 + 2^{l_1}}{2}\left(\frac{i_2-1}{2^{l_2}}\right)^2 \\
&\quad - (1-i_2)(1-i_1)\left(\frac{i_2-1}{2^{l_2}}\right) \\
&\quad + \frac{-2^{l_1+l_2}}{3}\left(\frac{i_2+1}{2^{l_2}}\right)^3 + \frac{2^{l_2}i_1 - 2^{l_2} + 2^{l_1}i_2 + 2^{l_1}}{2}\left(\frac{i_2+1}{2^{l_2}}\right)^2 \\
&\quad + (1+i_2)(1-i_1)\left(\frac{i_2+1}{2^{l_2}}\right) \\
&\quad - \frac{-2^{l_1+l_2}}{3}\left(\frac{i_2}{2^{l_2}}\right)^3 - \frac{2^{l_2}i_1 - 2^{l_2} + 2^{l_1}i_2 + 2^{l_1}}{2}\left(\frac{i_2}{2^{l_2}}\right)^2 \\
&\quad - (1+i_2)(1-i_1)\left(\frac{i_2}{2^{l_2}}\right) \\
&= -2^{l_1-2l_2+1}i_2 + 2^{-l_2}i_1 - 2^{-l_2} + 2^{l_1-2l_2}i_2 + 2^{l_1-2l_2+1}i_2 + 2^{-l_2+1} - 2^{-l_2+1}i_1
\end{aligned}
$$

$$
\begin{aligned}
&= 2^{-l_2} - 2^{-l_2} i_1 + 2^{l_1 - 2l_2} i_2 \\
&= \frac{1 - i_1 + 2^{l_1 - l_2} i_2}{2^{l_2}} = \frac{1 - 2^{l_1}\left(\frac{i_1}{2^{l_1}} - \frac{i_2}{2^{l_2}}\right)}{2^{l_2}} \,.
\end{aligned}
\tag{A.2}
$$

If the support of q_2 is located only under the right part of the support of q_1, we obtain:

$$
\begin{aligned}
r^{\text{linear}}_{q_1,q_2} &= \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \varphi^{\text{1d-linear}}_{p_1}(x) \cdot \varphi^{\text{1d-linear}}_{p_2}(x) \mathrm{d}x \\
&= \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \varphi_r(2^{l_1}x - i_1) \cdot \varphi^{\text{1d-linear}}_{p_2}(x) \mathrm{d}x \\
&= \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(-2^{l_1}x + i_1 + 1\right) \cdot \varphi(2^{l_2}x - i_2) \mathrm{d}x \\
&= \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(-2^{l_1}x + i_1 + 1\right) \cdot \left(\varphi_l(2^{l_2}x - i_2) + \varphi_r(2^{l_2}x - i_2)\right) \mathrm{d}x \\
&= \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(-2^{l_1}x + i_1 + 1\right) \cdot \varphi_l(2^{l_2}x - i_2) \mathrm{d}x \\
&\quad + \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(-2^{l_1}x + i_1 + 1\right) \cdot \varphi_r(2^{l_2}x - i_2) \mathrm{d}x \\
&= \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} \left(-2^{l_1}x + i_1 + 1\right) \cdot \left(2^{l_2}x - i_2 + 1\right) \mathrm{d}x \\
&\quad + \int\limits_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(-2^{l_1}x + i_1 + 1\right) \cdot \left(-2^{l_2}x + i_2 + 1\right) \mathrm{d}x
\end{aligned}
$$

$$
\begin{aligned}
=&\int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} \left(-2^{l_1+l_2}x^2 + \left(2^{l_2}i_1 + 2^{l_2} + 2^{l_1}i_2 - 2^{l_1}\right)x + (1-i_2)(1+i_1)\right)\mathrm{d}x \\
&+ \int\limits_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \left(2^{l_1+l_2}x^2 + \left(-2^{l_2}i_1 - 2^{l_2} - 2^{l_1}i_2 - 2^{l_1}\right)x + (1+i_2)(1+i_1)\right)\mathrm{d}x \\
=&\left[\frac{-2^{l_1+l_2}}{3}x^3 + \frac{2^{l_2}i_1 + 2^{l_2} + 2^{l_1}i_2 - 2^{l_1}}{2}x^2 + (1-i_2)(1+i_1)x\right]_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} \\
&+ \left[\frac{2^{l_1+l_2}}{3}x^3 + \frac{-2^{l_2}i_1 - 2^{l_2} - 2^{l_1}i_2 - 2^{l_1}}{2}x^2 + (1+i_2)(1+i_1)x\right]_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} \\
=&\frac{-2^{l_1+l_2}}{3}\left(\frac{i_2}{2^{l_2}}\right)^3 + \frac{2^{l_2}i_1 + 2^{l_2} + 2^{l_1}i_2 - 2^{l_1}}{2}\left(\frac{i_2}{2^{l_2}}\right)^2 \\
&+ (1-i_2)(1+i_1)\left(\frac{i_2}{2^{l_2}}\right) \\
&- \frac{-2^{l_1+l_2}}{3}\left(\frac{i_2-1}{2^{l_2}}\right)^3 - \frac{2^{l_2}i_1 + 2^{l_2} + 2^{l_1}i_2 - 2^{l_1}}{2}\left(\frac{i_2-1}{2^{l_2}}\right)^2 \\
&- (1-i_2)(1+i_1)\left(\frac{i_2-1}{2^{l_2}}\right) \\
&+ \frac{2^{l_1+l_2}}{3}\left(\frac{i_2+1}{2^{l_2}}\right)^3 + \frac{-2^{l_2}i_1 - 2^{l_2} - 2^{l_1}i_2 - 2^{l_1}}{2}\left(\frac{i_2+1}{2^{l_2}}\right)^2 \\
&+ (1+i_2)(1+i_1)\left(\frac{i_2+1}{2^{l_2}}\right) \\
&- \frac{2^{l_1+l_2}}{3}\left(\frac{i_2}{2^{l_2}}\right)^3 - \frac{-2^{l_2}i_1 - 2^{l_2} - 2^{l_1}i_2 - 2^{l_1}}{2}\left(\frac{i_2}{2^{l_2}}\right)^2 \\
&- (1+i_2)(1+i_1)\left(\frac{i_2}{2^{l_2}}\right) \\
=&2^{l_1-2l_2+1}i_2 - 2^{-l_2}i_1 - 2^{-l_2} - 2^{l_1-2l_2}i_2 - 2^{l_1-2l_2+1}i_2 + 2^{-l_2+1} + 2^{-l_2+1}i_1 \\
=&2^{-l_2}i_1 + 2^{-l_2} - 2^{l_1-2l_2}i_2 \\
=&2^{-l_2} + 2^{-l_2}i_1 - 2^{l_1-2l_2}i_2 \\
=&\frac{1 + i_1 - 2^{l_1-l_2}i_2}{2^{l_2}} = \frac{1 - 2^{l_1}\left(\frac{i_2}{2^{l_2}} - \frac{i_1}{2^{l_1}}\right)}{2^{l_2}}\,. \qquad \text{(A.3)}
\end{aligned}
$$

Both Eq. A.2 and Eq. A.3 are expressed as

$$r_{q_1,q_2}^{\text{linear}} = \frac{1 - 2^{l_1}\left|\frac{i_2}{2^{l_2}} - \frac{i_1}{2^{l_1}}\right|}{2^{l_2}}\,.$$

- $l_2 < l_1$ and $\max\left\{\frac{i_1-1}{2^{l_1}}, \frac{i_2-1}{2^{l_2}}\right\} < \min\left\{\frac{i_1+1}{2^{l_1}}, \frac{i_2+1}{2^{l_2}}\right\}$:

  We show that in this case, $r_{q_1,q_2}^{\text{linear}} = \frac{1-2^{l_2}\left|\frac{i_1}{2^{l_1}} - \frac{i_2}{2^{l_2}}\right|}{2^{l_1}}$. This case is similar to the previous one ($l_1 < l_2$ and $\max\left\{\frac{i_1-1}{2^{l_1}}, \frac{i_2-1}{2^{l_2}}\right\} < \min\left\{\frac{i_1+1}{2^{l_1}}, \frac{i_2+1}{2^{l_2}}\right\}$) only with exchanged q_1 and q_2. Thus, by also exchanging l_1 and l_2 as well as i_1 and i_2, the result follows as

$$r_{q_1,q_2}^{\text{linear}} = \frac{1 - 2^{l_2}\left|\frac{i_1}{2^{l_1}} - \frac{i_2}{2^{l_2}}\right|}{2^{l_1}}\,.$$

- All other cases:

  If $l_1 = l_2$, but $i_1 \neq i_2$, the supports of q_1 and q_2 do not overlap and the integral simplifies to zero. The same happens when $l_1 \neq l_2$ and Eq. A.1 does not hold. Thus, in all remaining cases we obtain:

$$r_{q_1,q_2}^{\text{linear}} = 0\,.$$

A.1.2. Modified Linear Basis Functions

Let $q_1 = (l_1, i_1)$ and $q_2 = (l_2, i_2)$ be two one-dimensional grid points. We proof that Eq. 2.40 is correct. For each case, we start with

$$r_{q_1,q_2}^{\text{modlinear}} = \int_0^1 \varphi_{q_1}^{\text{1d-modlinear}}(x) \cdot \varphi_{q_2}^{\text{1d-modlinear}}(x)\mathrm{d}x\,.$$

Note that in case of $l_1 = l_2$, we also use $l := l_1$ as level variable and in case of $i_1 = i_2$, we also use $i := i_1$ as index variable.

- $l_1 = l_2 = 1$:

  We show that in this case, $r_{q_1,q_2}^{\text{modlinear}} = 1$:

$$r_{q_1,q_2}^{\text{modlinear}} = \int_0^1 1 \cdot 1\mathrm{d}x = 1\,.$$

- $q_1 = q_2$ and $l > 1$ and $(i = 1 \text{ or } i = 2^l - 1)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = \frac{2^{3-l}}{3}$. The two cases $i = 1$ and $i = 2^l - 1$ are symmetric and identical. Thus, we only need to show the result for $i = 1$.

  The support of q_1 is $\left[0, 2^{-l+1}\right]$, thus the integral we calculate is:

$$\begin{aligned}
r^{\text{modlinear}}_{q_1,q_2} =& \int_0^{2^{-l+1}} \left(2 \cdot \varphi_r(2^{l-1}x)\right)^2 \mathrm{d}x \\
=& 4 \int_0^{2^{-l+1}} \left(-2^{l-1}x + 1\right)^2 \mathrm{d}x \\
=& 4 \int_0^{2^{-l+1}} \left(2^{2l-2}x^2 - 2^l x + 1\right) \mathrm{d}x \\
=& 4 \left[\frac{1}{3} 2^{2l-2} x^3 - 2^{l-1} x^2 + x\right]_0^{2^{-l+1}} \\
=& 4 \left(\frac{1}{3} 2^{2l-2} \cdot 2^{-3l+3} - 2^{l-1} \cdot 2^{-2l+2} + 2^{-l+1}\right) \\
=& \frac{2^{-l+3}}{3} - 2^{-l+3} + 2^{-l+3} \\
=& \frac{2^{-l+3}}{3} .
\end{aligned}$$

- $l_1 = 1$ and $(i_2 = 1 \text{ or } i_2 = 2^{l_2} - 1)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = 2^{1-l_2}$. The two cases $i = 1$ and $i = 2^l - 1$ are symmetric and identical. Thus, we only need to show the result for $i = 1$.

  The support of q_2 is $\left[0, 2^{-l_2+1}\right]$, thus the integral we calculate is:

$$\begin{aligned}
r^{\text{modlinear}}_{q_1,q_2} =& \int_0^{2^{-l_2+1}} 1 \cdot 2 \cdot \varphi_r(2^{l_2-1}x) \mathrm{d}x \\
=& 2 \cdot \int_0^{2^{-l_2+1}} \left(-2^{l_2-1}x + 1\right) \mathrm{d}x \\
=& 2 \left[-2^{l_2-2}x^2 + x\right]_0^{2^{-l_2+1}} \\
=& 2 \left(-2^{l_2-2} \cdot 2^{-2l_2+2} + 2^{-l_2+1}\right) \\
=& -2^{-l_2+1} + 2 \cdot 2^{-l_2+1} \\
=& 2^{1-l_2} .
\end{aligned}$$

- $l_2 = 1$ and $(i_1 = 1 \text{ or } i_1 = 2^{l_1} - 1)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = 2^{1-l_1}$. This case is similar to the previous one ($l_1 = 1$ and $(i_2 = 1 \text{ or } i_2 = 2^{l_2} - 1)$), only with exchanged q_1 and q_2. Thus, by also exchanging l_1 and l_2, the result follows as

$$r^{\text{modlinear}}_{q_1,q_2} = 2^{1-l_1}\,.$$

- $l_1 = 1, 1 < i_2 < 2^{l_2} - 1$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = 2^{-l_2}$. For $1 < i_2 < 2^{l_2} - 1$, all basis functions are of the same shape and the concrete value of i_2 does not matter, because $\varphi^{\text{1d-modlinear}}_{q_1} = 1$, which is constant. We choose to calculate it for $i_2 = 3$. Also, we know that $\varphi^{\text{1d-modlinear}}_{q_2}$ is zero outside of the support of q_2. Furthermore, we use the fact that the integral value under the first half of $\varphi^{\text{1d-modlinear}}_{q_2}$ is the same as under the second half. In total, we calculate:

$$\begin{aligned}
r^{\text{modlinear}}_{q_1,q_2} =& 2 \cdot \int_{2^{-l_2+1}}^{3\cdot 2^{-l_2}} 1 \cdot \varphi_l(2^{l_2}x - 3)\mathrm{d}x \\
=& 2 \cdot \int_{2^{-l_2+1}}^{3\cdot 2^{-l_2}} \left(2^{l_2}x - 2\right) \mathrm{d}x \\
=& 2 \cdot \left[2^{l_2-1}x^2 - 2x\right]_{2^{-l_2+1}}^{3\cdot 2^{-l_2}} \\
=& 9 \cdot 2^{-l_2} - 3 \cdot 2^{-l_2+2} - 2^{-l_2+2} + 2^{-l_2+3} \\
=& 9 \cdot 2^{-l_2} - 12 \cdot 2^{-l_2} - 4 \cdot 2^{-l_2} + 8 \cdot 2^{-l_2} \\
=& 2^{-l_2}\,.
\end{aligned}$$

- $l_2 = 1, 1 < i_1 < 2^{l_1} - 1$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = 2^{-l_1}$. This case is similar to the previous one ($l_1 = 1, 1 < i_2 < 2^{l_2} - 1$), only with exchanged q_1 and q_2. Thus, by also exchanging l_1 and l_2, the result follows as

$$r^{\text{modlinear}}_{q_1,q_2} = 2^{-l_1}\,.$$

- $l_1 < l_2$ and $\left(i_1 = i_2 = 1 \text{ or } \left(i_1 = 2^{l_1} - 1 \text{ and } i_2 = 2^{l_2} - 1\right)\right)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = \frac{4}{2^{l_2}}\left(1 - \frac{1}{3}2^{l_1-l_2}\right)$. The two cases $i_1 = 1, i_2 = 1$ and $i_1 = 2^{l_1} - 1, i_2 = 2^{l_2} - 1$ are symmetric and identical. Thus, we only need to show the result for $i = 1$.

The support of q_2 is $\left[0, 2^{-l_2+1}\right]$, thus the integral we calculate is:

$$\begin{aligned}
r^{\text{modlinear}}_{q_1,q_2} =& \int\limits_0^{2^{-l_2+1}} 2\varphi_r(2^{l_1-1}x) \cdot 2\varphi_r(2^{l_2-1}x)\mathrm{d}x \\
=&4 \cdot \int\limits_0^{2^{-l_2+1}} \left(-2^{l_1-1}x+1\right) \cdot \left(-2^{l_2-1}x+1\right) \mathrm{d}x \\
=&4 \cdot \int\limits_0^{2^{-l_2+1}} \left(2^{l_1+l_2-2}x^2 - \left(2^{l_1-1}+2^{l_2-1}\right)x+1\right) \mathrm{d}x \\
=&4 \cdot \left[\frac{1}{3}2^{l_1+l_2-2}x^3 - \left(2^{l_1-2}+2^{l_2-2}\right)x^2+x\right]_0^{2^{-l_2+1}} \\
=&4 \cdot \left(\frac{1}{3}2^{l_1-2l_2+1} - 2^{l_1-2l_2} - 2^{-l_2} + 2^{-l_2+1}\right) \\
=&4 \cdot \left(-\frac{1}{3}2^{l_1-2l_2} + 2^{-l_2}\right) \\
=&\frac{4}{2^{l_2}}\left(1-\frac{1}{3}2^{l_1-l_2}\right) .
\end{aligned}$$

- $l_2 < l_1$ and $\left(i_1 = i_2 = 1 \text{ or } \left(i_1 = 2^{l_1}-1 \text{ and } i_2 = 2^{l_2}-1\right)\right)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = \frac{4}{2^{l_1}}\left(1-\frac{1}{3}2^{l_2-l_1}\right)$. This case is similar to the previous one ($l_1 < l_2$ and $\left(i_1 = i_2 = 1 \text{ or } \left(i_1 = 2^{l_1}-1 \text{ and } i_2 = 2^{l_2}-1\right)\right)$), only with exchanged q_1 and q_2. Thus, by also exchanging l_1 and l_2, the result follows as

  $$r^{\text{modlinear}}_{q_1,q_2} = \frac{4}{2^{l_1}}\left(1-\frac{1}{3}2^{l_2-l_1}\right) .$$

- $l_1 < l_2$ and $\left(i_1 = 1 \text{ and } 1 < i_2 < \frac{2^{l_2}}{2^{l_1}}\right)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = \frac{2-2^{l_1-l_2}i_2}{2^{l_2}}$. With $i_1 = 1$ and $i_2 < \frac{2^{l_2}}{2^{l_1}}$, we know that $\texttt{1d-support}_{q_2} \subset \texttt{1d-support}_{q_1}$. Thus, the integral we calculate is:

$$\begin{aligned}
r^{\text{modlinear}}_{q_1,q_2} =& \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} 2\varphi_r(2^{l_1-1}x) \cdot \varphi_l(2^{l_2}x-i_2)\mathrm{d}x + \int\limits_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} 2\varphi_r(2^{l_1-1}x) \cdot \varphi_r(2^{l_2}x-i_2)\mathrm{d}x \\
=& \int\limits_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}} 2\left(-2^{l_1-1}x+1\right) \cdot \left(2^{l_2}x-i_2+1\right) \mathrm{d}x
\end{aligned}$$

$$
\begin{aligned}
&+\int_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}} 2\left(-2^{l_1-1}x+1\right)\cdot\left(-2^{l_2}x+i_2+1\right)\mathrm{d}x\\
=&\int_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}}\left(-2^{l_1+l_2}x^2+\left(2^{l_1}i_2-2^{l_1}+2^{l_2+1}\right)x+(2-2i_2)\right)\mathrm{d}x\\
&+\int_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}}\left(2^{l_1+l_2}x^2+\left(-2^{l_1}i_2-2^{l_1}-2^{l_2+1}\right)x+(2+2i_2)\right)\mathrm{d}x\\
=&\left[-\frac{1}{3}2^{l_1+l_2}x^3+\left(2^{l_1-1}i_2-2^{l_1-1}+2^{l_2}\right)x^2+(2-2i_2)x\right]_{\frac{i_2-1}{2^{l_2}}}^{\frac{i_2}{2^{l_2}}}\\
&+\left[\frac{1}{3}2^{l_1+l_2}x^3+\left(-2^{l_1-1}i_2-2^{l_1-1}-2^{l_2}\right)x^2+(2+2i_2)x\right]_{\frac{i_2}{2^{l_2}}}^{\frac{i_2+1}{2^{l_2}}}\\
=&-\frac{1}{3}2^{l_1+l_2}\left(\frac{i_2}{2^{l_2}}\right)^3+\left(2^{l_1-1}i_2-2^{l_1-1}+2^{l_2}\right)\left(\frac{i_2}{2^{l_2}}\right)^2\\
&+(2-2i_2)\left(\frac{i_2}{2^{l_2}}\right)\\
&+\frac{1}{3}2^{l_1+l_2}\left(\frac{i_2-1}{2^{l_2}}\right)^3-\left(2^{l_1-1}i_2-2^{l_1-1}+2^{l_2}\right)\left(\frac{i_2-1}{2^{l_2}}\right)^2\\
&-(2-2i_2)\left(\frac{i_2-1}{2^{l_2}}\right)\\
&+\frac{1}{3}2^{l_1+l_2}\left(\frac{i_2+1}{2^{l_2}}\right)^3+\left(-2^{l_1-1}i_2-2^{l_1-1}-2^{l_2}\right)\left(\frac{i_2+1}{2^{l_2}}\right)^2\\
&+(2+2i_2)\left(\frac{i_2+1}{2^{l_2}}\right)\\
&-\frac{1}{3}2^{l_1+l_2}\left(\frac{i_2}{2^{l_2}}\right)^3-\left(-2^{l_1-1}i_2-2^{l_1-1}-2^{l_2}\right)\left(\frac{i_2}{2^{l_2}}\right)^2\\
&-(2+2i_2)\left(\frac{i_2}{2^{l_2}}\right)\\
=&2^{l_1-2l_2+1}i_2-2^{l_1-2l_2}i_2-2^{l_1-2l_2+1}i_2-2^{-l_2+1}+2^{-l_2+1}\\
=&-2^{l_1-2l_2}i_2+2^{-l_2+1}\\
=&\frac{2-2^{l_1-l_2}i_2}{2^{l_2}}\,.
\end{aligned}
$$

- $l_2 < l_1$ and $\left(i_2 = 1 \text{ and } 1 < i_1 < \frac{2^{l_1}}{2^{l_2}}\right)$:

We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = \frac{2-2^{l_2-l_1}i_1}{2^{l_1}}$. This case is similar to the previous one ($l_1 < l_2$ and $\left(i_1 = 1 \text{ and } 1 < i_2 < \frac{2^{l_2}}{2^{l_1}}\right)$) only with exchanged q_1 and q_2. Thus, by also exchanging l_1 and l_2 as well as i_1 and i_2, the result follows as

$$r^{\text{modlinear}}_{q_1,q_2} = \frac{2-2^{l_2-l_1}i_1}{2^{l_1}}\,.$$

- $l_1 < l_2$ and $\left(i_1 = 2^{l_1} - 1 \text{ and } \frac{2^{l_1+l_2}-2^{l_2}}{2^{l_1}} < i_2 < 2^{l_2} - 1\right)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = \frac{2-2^{l_1}+2^{l_1-l_2}i_2}{2^{l_2}}$. This case is similar to the penultimate one ($l_1 < l_2$ and $\left(i_1 = 1 \text{ and } 1 < i_2 < \frac{2^{l_2}}{2^{l_1}}\right)$) only that q_1 and q_2 are mirrored at $\frac{1}{2}$. Thus, by substituting i_2 with $2^{l_2} - i_2$, the result follows as

$$r^{\text{modlinear}}_{q_1,q_2} = \frac{2-2^{l_1-l_2}\left(2^{l_2}-i_2\right)}{2^{l_2}} = \frac{2-2^{l_1}+2^{l_1-l_2}i_2}{2^{l_2}}\,.$$

- $l_2 < l_1$ and $\left(i_2 = 2^{l_2} - 1 \text{ and } \frac{2^{l_2+l_1}-2^{l_1}}{2^{l_2}} < i_1 < 2^{l_1} - 1\right)$:

  We show that in this case, $r^{\text{modlinear}}_{q_1,q_2} = \frac{2-2^{l_2}+2^{l_2-l_1}i_1}{2^{l_1}}$. This case is similar to the previous one ($l_1 < l_2$ and $\left(i_1 = 2^{l_1} - 1 \text{ and } \frac{2^{l_1+l_2}-2^{l_2}}{2^{l_1}} < i_2 < 2^{l_2} - 1\right)$) only with exchanged q_1 and q_2. Thus, by also exchanging l_1 and l_2 as well as i_1 and i_2, the result follows as

$$r^{\text{modlinear}}_{q_1,q_2} = \frac{2-2^{l_2}+2^{l_2-l_1}i_1}{2^{l_1}}\,.$$

- All other cases:

  In all other cases, either the supports do not overlap (thus, the result is zero) or we are in the case of the linear basis function. Since the first case (the supports do not overlap) is also covered in $r^{\text{linear}}_{q_1,q_2}$, we write for all remaining cases, that

$$r^{\text{modlinear}}_{q_1,q_2} = r^{\text{linear}}_{q_1,q_2}\,.$$

A.1.3. Kinked Linear Basis Functions

Let $q_1 = (l_1, i_1)$ and $q_2 = (l_2, i_2)$ be two one-dimensional grid points. We proof that Eq. 2.46 is correct. For each case, we start with

$$r^{\text{kinklinear}}_{q_1,q_2} = \int_0^1 \varphi^{\text{1d-kinklinear}}_{q_1}(x) \cdot \varphi^{\text{1d-kinklinear}}_{q_2}(x)\mathrm{d}x\,.$$

Note that in case of $l_1 = l_2$, we also use $l := l_1$ as level variable and in case of $i_1 = i_2$, we also use $i := i_1$ as index variable.

- $l_1 = l_2 = 1$:

  We show that in this case, $r^{\text{kinklinear}}_{q_1,q_2} = 1$:

$$r^{\text{kinklinear}}_{q_1,q_2} = \int_0^1 1 \cdot 1 \mathrm{d}x = 1\,.$$

- $q_1 = q_2$ and $l > 1$ and $(i = 1$ or $i = 2^l - 1)$:

  We show that in this case, $r^{\text{kinklinear}}_{q_1,q_2} = \frac{2^{2-l}}{3}$. The two cases $i = 1$ and $i = 2^l - 1$ are symmetric and identical. Thus, we only need to show the result for $i = 1$.

  The support of q_1 is $\left[0, 2^{-l+1}\right]$, thus the integral we calculate is:

$$\begin{aligned}
r^{\text{kinklinear}}_{q_1,q_2} &= \int_0^{2^{-l}} 1^2 \mathrm{d}x + \int_{2^{-l}}^{2^{-l+1}} \left(\varphi_r(2^l x - 1)\right)^2 \mathrm{d}x \\
&= [x]_0^{2^{-l}} + \int_{2^{-l}}^{2^{-l+1}} \left(2 - 2^l x\right)^2 \mathrm{d}x \\
&= 2^{-l} + \int_{2^{-l}}^{2^{-l+1}} \left(4 - 2^{l+2}x + 2^{2l}x^2\right) \mathrm{d}x \\
&= 2^{-l} + \left[4x - 2^{l+1}x^2 + \frac{1}{3}2^{2l}x^3\right]_{2^{-l}}^{2^{-l+1}} \\
&= 2^{-l} + 2^{3-l} - 2^{3-l} + \frac{1}{3}2^{3-l} - 2^{2-l} + 2^{1-l} - \frac{1}{3}2^{-l} \\
&= 2^{-l}\left(1 + \frac{8}{3} - 4 + 2 - \frac{1}{3}\right) \\
&= 2^{-l}\frac{4}{3} \\
&= \frac{2^{2-l}}{3}\,.
\end{aligned}$$

- $l_1 > l_2$ and $\left(\left(i_2 = 1 \text{ and } i_1 < \frac{2^{l_1}}{2^{l_2}}\right) \text{ or } \left(i_2 = 2^{l_2} - 1 \text{ and } i_1 > \frac{2^{l_1}\left(2^{l_2}-1\right)}{2^{l_2}}\right)\right)$:

  We show that in this case, $r^{\text{kinklinear}}_{q_1,q_2} = 3 \cdot 2^{-l_1-1}$. The two cases $\left(i_2 = 1 \text{ and } i_1 < \frac{2^{l_1}}{2^{l_2}}\right)$ and $\left(i_2 = 2^{l_2} - 1 \text{ and } i_1 > \frac{2^{l_1}\left(2^{l_2}-1\right)}{2^{l_2}}\right)$ are symmetric and identical. Thus, we only need to show the result for $\left(i_2 = 1 \text{ and } i_1 < \frac{2^{l_1}}{2^{l_2}}\right)$. In this case, in `support`$^1(q_1)$, it holds that $\varphi_{q_2}^{\text{1d-kinklinear}}(x) = 1$. Without loss of generality, we assume that $i_1 = 1$, because $r_{q_1,q_2}^{\text{kinklinear}}$ is translation invariant due to the constant value of $\varphi_{q_2}^{\text{1d-kinklinear}}$.

The support of q_1 is $\left[0, 2^{-l+1}\right]$, thus the integral we calculate is:

$$\begin{aligned}
r_{q_1,q_2}^{\text{kinklinear}} &= \int_0^{2^{-l}} 1 \cdot 1 \mathrm{d}x + \int_{2^{-l}}^{2^{-l+1}} 1 \cdot \varphi_r(2^l x - 1) \mathrm{d}x \\
&= [x]_0^{2^{-l}} + \int_{2^{-l}}^{2^{-l+1}} \left(2 - 2^l x\right) \mathrm{d}x \\
&= 2^{-l} + \left[2x - 2^{l-1}x^2\right]_{2^{-l}}^{2^{-l+1}} \\
&= 2^{-l} + 2^{2-l} - 2^{1-l} - 2^{1-l} + 2^{-l-1} \\
&= 2^{-l}\left(1 + 4 - 2 - 2 + \frac{1}{2}\right) \\
&= 3 \cdot 2^{-l-1} .
\end{aligned}$$

- $l_2 > l_1$ and $\left(\left(i_1 = 1 \text{ and } i_2 < \frac{2^{l_2}}{2^{l_1}}\right) \text{ or } \left(i_1 = 2^{l_1} - 1 \text{ and } i_2 > \frac{2^{l_2}(2^{l_1}-1)}{2^{l_1}}\right)\right)$:

  We show that in this case, $r_{q_1,q_2}^{\text{kinklinear}} = 3 \cdot 2^{-l_2-1}$. This case is similar to the previous one ($l_1 > l_2$ and $\left(\left(i_2 = 1 \text{ and } i_1 < \frac{2^{l_1}}{2^{l_2}}\right) \text{ or } \left(i_2 = 2^{l_2} - 1 \text{ and } i_1 > \frac{2^{l_1}(2^{l_2}-1)}{2^{l_2}}\right)\right)$), only with exchanged q_1 and q_2. Thus, by also exchanging l_1 and l_2, the result follows as

$$r_{q_1,q_2}^{\text{kinklinear}} = 3 \cdot 2^{-l_2-1} .$$

- All other cases:

  In all other cases, either the supports do not overlap (thus, the result is zero) or we are in the case of the linear basis function. Since the first case (the supports do not overlap) is also covered in $r_{q_1,q_2}^{\text{linear}}$, we write for all remaining cases, that

$$r_{q_1,q_2}^{\text{kinklinear}} = r_{q_1,q_2}^{\text{linear}} .$$

Appendix B.

Testing Environment

B.1. Workstation

Most serial tests were run on the workstation provided by the TUM Department of Informatics:

Type	Personal Computer
CPU Type	Intel® Core™ i7-4770
Hardware threads	8
Core Frequency	3.40GHz
Memory	16GB
Compiler	GCC (version 8.3.0)

Table B.1.: Workstation specification

B.2. Linux Cluster

The parallel tests were executed on the LRZ Linux Cluster CoolMUC-2 [63]:

Type	Cluster Computer
Max. number of nodes	812
CPU Type	Intel® Xeon® E5-2690 v3
Hardware threads per node	56
Core Frequency	2.60GHz
Memory per node	64GB
Compiler	ICC (version TODO)

Table B.2.: Linux Cluster specification

Appendix C.

Raw Data

C.1. Optimizing λ (Sec. 3.2.2.3)

Table C.1.: Raw data for Fig. 3.3a.

Level	λ						
	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}
1	0.00399381	0.00399379	0.00399319	0.00399319	0.00405334	0.00473121	0.00573947
2	0.00287775	0.00287781	0.00287836	0.00288535	0.00302031	0.00412208	0.00559591
3	0.00170443	0.00170451	0.00170589	0.00175670	0.00241823	0.00390102	0.00552704
4	0.00114618	0.00114368	0.00112223	0.00113156	0.00220765	0.00382974	0.00546687
5	0.00184365	0.00183308	0.00174408	0.00148587	0.00214403	0.00365791	0.00542166
6	0.00285691	0.00282868	0.00262104	0.00199680	0.00200525	0.00357847	0.00539224
7	0.00448266	0.00437229	0.00375927	0.00248373	0.00196103	0.00352228	0.00537290
8	0.00690990	0.00661896	0.00528091	0.00305691	0.00198765	0.00348490	0.00536062

Table C.2.: Raw data for Fig. 3.3b.

Level	λ						
	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}
1	0.00441082	0.00441082	0.00441088	0.00441148	0.00441930	0.00452496	0.00499410
2	0.00387174	0.00387174	0.00387174	0.00387267	0.00392797	0.00438048	0.00500445
3	0.00277828	0.00277830	0.00277853	0.00278499	0.00292542	0.00369954	0.00494938
4	0.00158018	0.00158108	0.00159137	0.00173282	0.00258600	0.00357825	0.00477395
5	0.00186800	0.00186529	0.00185041	0.00197700	0.00254740	0.00341661	0.00469452
6	0.00265659	0.00263286	0.00249864	0.00231304	0.00239156	0.00331838	0.00465299
7	0.00410038	0.00397985	0.00354415	0.00282178	0.00237743	0.00324434	0.00463021
8	0.00631679	0.00605244	0.00513247	0.00351472	0.00243565	0.00320289	0.00461700

Table C.3.: Raw data for Fig. 3.3c.

Level	λ						
	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}
1	0.00431819	0.00431819	0.00431822	0.00431857	0.00432672	0.00445286	0.00503899
2	0.00420697	0.00420700	0.00420729	0.00421125	0.00426244	0.00446390	0.00496074
3	0.00411630	0.00411627	0.00411602	0.00411582	0.00414701	0.00433460	0.00486337
4	0.00173159	0.00173277	0.00174614	0.00187192	0.00243554	0.00361145	0.00479485
5	0.00202501	0.00201884	0.00198060	0.00210346	0.00244883	0.00342035	0.00473851
6	0.00279683	0.00276282	0.00258057	0.00225875	0.00234419	0.00331228	0.00470482
7	0.00425843	0.00413131	0.00361882	0.00280294	0.00231503	0.00324562	0.00468400
8	0.00649266	0.00623455	0.00519046	0.00345118	0.00236068	0.00320680	0.00467133

C.2. Sparsity of R (Fig. 3.4)

Table C.4.: Raw data for Fig. 3.4.

Dim.	**Level**									
	1	2	3	4	5	6	7	8	9	10
1	1	0.77778	0.55102	0.36889	0.23621	0.14588	0.08748	0.05123	0.02942	0.01664
2	1	0.84	0.64014	0.47689	0.35437	0.26530	0.20131	0.15536	0.12213	0.09782
3	1	0.87755	0.69407	0.53332	0.40671	0.31069	0.23891	0.18542	0.14546	0.11541
4	1	0.90124	0.73344	0.57583	0.44602	0.34428	0.26621	0.20682	0.16173	0.12742
5	1	0.91736	0.76394	0.61080	0.47947	0.37348	0.29035	0.22604	0.17659	0.13862
6	1	0.92899	0.78829	0.64046	0.50901	0.40006	0.31287	0.24438	0.19109	0.14981
7	1	0.93778	0.80817	0.66603	0.53550	0.42464	0.33422	0.26215		
8	1	0.94464	0.82470	0.68831	0.55943	0.44749	0.35456			
9	1	0.95014	0.83864	0.70789	0.58116	0.46879	0.37396			
10	1	0.95465	0.85055	0.72523	0.60096	0.48870	0.39246			

C.3. Permutation (front vs end) for Cholesky coarsening Fig. 3.6

Table C.5.: Selected raw data points for Fig. 3.6. Runtimes in microseconds.

Grid point index	Permutation Direction	
	Front	End
0	16983.4	12350
1	16838	12077.2
2	16983.8	12494
4	16874	12248.8
8	17114.2	12911.2
16	17824	12260.6
32	17160	11909.6
64	17573.4	12141.2
128	18173.2	10931.8
256	19174.6	10018.8
512	20789.6	7905
1024	23634.2	6013.4
1470	26548.2	5353.4

C.4. Comparison of the Matrix Decomposition Methods (Sec. 3.2.3.3.5)

Table C.6.: Raw data for Fig. 3.7. Runtimes in microseconds.

# grid points	Decomposition Method				
	Cholesky	LU	Eigen	Tridiagonal	Incomplete Cholesky
5	0.000000e+00	0.000000e+00	1.250000e+00	1.600000e+00	1.949000e+02
7	1.000000e−01	0.000000e+00	2.200000e+00	3.850000e+00	4.370000e+01
9	2.000000e−01	0.000000e+00	3.500000e+00	4.150000e+00	6.380000e+01
11	2.000000e−01	0.000000e+00	3.600000e+00	4.200000e+00	1.763000e+02
17	1.750000e+00	1.600000e+00	2.440000e+01	1.430000e+01	1.090000e+02
31	7.900000e+00	1.115000e+01	9.790000e+01	6.250000e+01	2.379000e+02
49	1.872500e+01	3.460000e+01	2.597500e+02	1.528250e+02	5.787000e+02
71	5.080000e+01	9.200000e+01	6.107000e+02	3.997000e+02	1.376900e+03
111	1.733000e+02	3.181000e+02	2.498600e+03	1.379550e+03	1.807200e+03
129	2.639000e+02	4.827000e+02	4.265700e+03	2.161200e+03	1.833300e+03
209	1.160250e+03	1.923550e+03	1.567910e+04	8.787900e+03	2.834700e+03
351	5.823400e+03	8.574800e+03	8.602485e+04	5.077895e+04	5.691950e+03
769	6.098630e+04	8.461840e+04	1.159442e+06	6.459414e+05	2.349560e+04
1471	4.233507e+05	8.081081e+05	4.361430e+07	3.125739e+07	1.285340e+05

Table C.7.: Raw data for Fig. 3.8. Runtimes in microseconds.

# grid points	Decomposition Method				
	Cholesky	LU	Eigen	Tridiagonal	Incomplete Cholesky
5	0.2500000	0.1000000	0.4500000	0.1000000	11.1111111
7	0.0500000	0.0500000	0.0500000	0.0000000	8.4444444
9	0.0500000	0.0500000	0.1000000	0.0500000	9.1111111
11	0.1000000	0.0500000	0.1000000	0.0500000	9.8888889
17	1.0500000	1.0000000	1.0500000	1.0500000	16.4000000
31	2.0000000	2.0000000	1.1500000	2.0000000	14.2000000
49	6.7500000	6.6250000	3.6500000	4.6750000	20.3333333
71	11.3000000	11.3000000	7.0500000	11.0500000	22.5555556
111	28.2000000	28.1000000	16.8500000	25.7000000	33.0000000
129	38.2777778	38.5500000	23.4000000	33.9500000	38.5555556
209	102.3000000	101.5000000	57.9500000	88.9500000	73.8000000
351	288.8750000	295.5750000	163.5750000	255.9250000	130.3888889
769	1400.2000000	1404.3500000	804.8000000	1382.0500000	452.0000000
1471	7339.5500000	5341.9500000	3253.2500000	6485.5500000	1559.4285714

Table C.8.: Selected raw data points for Fig. 3.9a. Runtimes in microseconds.

# initial grid points	Decomposition Method		
	Cholesky	Tridiagonal	Incomplete Cholesky
8	6.835000e+03	2.994530e+04	4.142600e+03
16	9.375900e+03	5.897330e+04	5.228800e+03
32	1.387410e+04	1.173523e+05	1.064620e+04
64	2.313920e+04	2.441599e+05	1.405460e+04
134	4.509580e+04	2.441599e+05	1.727680e+04
260	9.417770e+04	1.141447e+06	2.212710e+04
513	2.420015e+05	2.802620e+06	5.118250e+04
682	3.924279e+05	4.194174e+06	7.817160e+04

Table C.9.: Raw data for Fig. 3.9b. Runtimes in microseconds.

# new grid points	Decomposition Method		
	Cholesky	Tridiagonal	Incomplete Cholesky
129	3.074400e-03	1.599350e-02	1.998200e-03
351	1.020330e-02	8.083340e-02	2.430300e-03
769	3.494880e-02	3.945665e-01	1.651010e-02

Table C.10.: Selected raw data points for Fig. 3.10. Runtimes in microseconds.

# removed grid points	Decomposition Method		
	Cholesky	Tridiagonal	Incomplete Cholesky
2	2.449000e+03	7.961000e+03	4.000000e+00
4	2.557000e+03	1.306100e+04	4.000000e+00
8	2.539000e+03	2.356400e+04	6.000000e+00
20	5.782000e+03	5.491500e+04	1.400000e+01
32	7.825000e+03	8.982750e+04	2.350000e+01
63	1.126900e+04	1.686353e+05	3.850000e+01
88	1.523700e+04	2.360090e+05	5.250000e+01

C.5. Classification Refinement (Sec. 3.3.2)

Table C.11.: Raw data for Fig. 3.14.

Batch	Classification Refinement		Surplus-based Refinement	
	Accuracy	# grid points	Accuracy	# grid points
1	0.804094	2561	0.804094	2561
2	0.820504	2609	0.836358	2863
3	0.850649	2673	0.858323	3405
4	0.879035	2736	0.870861	3959
5	0.884182	2799	0.885275	4719
6	0.885257	2861	0.886183	5617
7	0.886079	2926	0.886338	6613
8	0.887683	2990	0.885843	7746
9	0.889473	3054	0.885616	8948
10	0.888117	3118	0.885348	10344

C.6. Batch Parallelization (Sec. 4.2)

Table C.12.: Raw data for Fig. 4.3b and Fig. 4.4. Runtimes in seconds.

Setup	Runtime for strong scaling	Runtime for weak scaling
Reference	84.3	4.82
2 nodes	61.5	7.04
3 nodes	31.2	5.16
4 nodes	21.6	4.35
5 nodes	17.6	4.17
6 nodes	14.8	4.13
7 nodes	12.9	4.10
8 nodes	12.5	4.12

C.7. Employing the Combination Grid Technique (Sec. 4.3)

Table C.13.: Raw data for Fig. 4.5a.

Dim.	Level									
	1	2	3	4	5	6	7	8	9	10
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	1.40	1.71	1.94	2.12	2.25	2.36	2.44	2.51	2.56
3	1.00	1.43	1.87	2.30	2.69	3.04	3.37	3.65	3.90	4.12
4	1.00	1.44	1.94	2.46	3.00	3.54	4.06	4.57	5.05	5.50
5	1.00	1.45	1.99	2.58	3.23	3.91	4.61	5.33	6.05	6.76
6	1.00	1.46	2.02	2.67	3.41	4.21	5.08	6.00	6.95	7.94
7	1.00	1.47	2.05	2.74	3.55	4.47	5.50	6.60	7.80	9.06
8	1.00	1.47	2.07	2.80	3.68	4.69	5.85	7.15	8.58	10.14
9	1.00	1.47	2.09	2.85	3.78	4.89	6.18	7.65	9.31	11.15
10	1.00	1.48	2.10	2.89	3.87	5.06	6.46	8.10	9.98	12.12

Table C.14.: Raw data for Fig. 4.5b.

Dim.	Level									
	1	2	3	4	5	6	7	8	9	10
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	7.60e−1	6.81e−1	6.29e−1	5.83e−1	5.40e−1	5.00e−1	4.64e−1	4.30e−1	4.01e−1
3	1.00	5.71e−1	4.35e−1	3.63e−1	3.13e−1	2.75e−1	2.43e−1	2.16e−1	1.93e−1	1.73e−1
4	1.00	4.57e−1	2.99e−1	2.25e−1	1.80e−1	1.49e−1	1.26e−1	1.08e−1	9.32e−2	8.09e−2
5	1.00	3.80e−1	2.18e−1	1.49e−1	1.11e−1	8.66e−2	6.98e−2	5.75e−2	4.80e−2	4.05e−2
6	1.00	3.25e−1	1.66e−1	1.03e−1	7.16e−2	5.29e−2	4.07e−2	3.22e−2	2.60e−2	2.13e−2
7	1.00	2.84e−1	1.31e−1	7.47e−2	4.83e−2	3.38e−2	2.48e−2	1.89e−2	1.47e−2	1.17e−2
8	1.00	2.53e−1	1.05e−1	5.57e−2	3.38e−2	2.23e−2	1.57e−2	1.15e−2	8.65e−3	6.68e−3
9	1.00	2.27e−1	8.68e−2	4.27e−2	2.43e−2	1.53e−2	1.02e−2	7.20e−3	5.25e−3	3.94e−3
10	1.00	2.06e−1	7.28e−2	3.34e−2	1.79e−2	1.07e−2	6.87e−3	4.65e−3	3.28e−3	2.39e−3

Table C.15.: Raw data for Fig. 4.5c.

Dim.	Level									
	1	2	3	4	5	6	7	8	9	10
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	4.40e−1	2.99e−1	2.27e−1	1.79e−1	1.45e−1	1.18e−1	9.82e−2	8.23e−2	6.97e−2
3	1.00	2.39e−1	1.11e−1	6.48e−2	4.23e−2	2.92e−2	2.09e−2	1.54e−2	1.15e−2	8.76e−3
4	1.00	1.50e−1	4.98e−2	2.29e−2	1.24e−2	7.43e−3	4.72e−3	3.12e−3	2.13e−3	1.48e−3
5	1.00	1.02e−1	2.55e−2	9.44e−3	4.31e−3	2.24e−3	1.26e−3	7.55e−4	4.71e−4	3.04e−4
6	1.00	7.42e−2	1.44e−2	4.35e−3	1.69e−3	7.65e−4	3.86e−4	2.09e−4	1.20e−4	7.17e−5
7	1.00	5.63e−2	8.74e−3	2.19e−3	7.30e−4	2.91e−4	1.31e−4	6.46e−5	3.40e−5	1.89e−5
8	1.00	4.42e−2	5.60e−3	1.19e−3	3.41e−4	1.20e−4	4.87e−5	2.18e−5	1.06e−5	5.46e−6
9	1.00	3.56e−2	3.75e−3	6.79e−4	1.71e−4	5.33e−5	1.95e−5	7.98e−6	3.57e−6	1.71e−6
10	1.00	2.93e−2	2.61e−3	4.08e−4	9.01e−5	2.52e−5	8.32e−6	3.12e−6	1.29e−6	5.75e−7

Table C.16.: Raw data for Fig. 4.6. Runtimes in seconds.

# grid points	Regular sparse grid	Combination grid technique
2561	2.88	1.32
4159	11.35	2.39
5503	24.91	3.18

C.8. Parallel Linear Algebra (Sec. 4.4)

Table C.17.: Raw data for Fig. 4.7. Runtimes in seconds.

Setup	Spatial adaptivity disabled		Spatial adaptivity enabled	
	Tridiagonal	**Cholesky**	**Tridiagonal**	**Cholesky**
Reference	111.81	107.5	465.66	213.72
1 rank	119.01	113.34	487.19	219.60
2 ranks	73.17	69.83	426.81	156.32
4 ranks	43.02	39.90	391.36	120.55
8 ranks	26.15	24.70	369.18	96.66
16 ranks	19.54	18.41	359.22	82.69
32 ranks	18.72	16.27	437.96	89.16

Table C.18.: Raw data for Fig. 4.8. Runtimes in seconds.

Setup	Spatial adaptivity disabled		Spatial adaptivity enabled	
	Tridiagonal	**Cholesky**	**Tridiagonal**	**Cholesky**
Reference	7.18	8.13	319.86	41.49
1 rank	15.57	10.74	340.13	46.91
2 ranks	14.90	9.39	355.80	27.53
4 ranks	13.19	11.70	185.17	73.45
8 ranks	12.23	10.94	186.31	70.29
16 ranks	13.50	11.16	190.71	76.63
32 ranks	16.33	14.58	236.40	143.11

C.9. Geometry-aware Sparse Grids (Chap. 5)

Table C.19.: Raw data for Fig. 5.5.

Level	**Grid type**					
	Reg. SG	GaSG w. `DN`	GaSG w. `DDN`	GaSG w. `DBP-2`	GaSG w. `SQ`	GaSG w. `2-XCDB-1`
1	1	1	1	1	1	1
2	1.569e+03	1.569e+03	1.569e+03	1.569e+03	1.569e+03	1.569e+03
3	1.232e+06	1.075e+04	1.659e+04	2.241e+04	1.659e+04	1.659e+04
4	6.462e+08	4.122e+04	7.038e+04	9.950e+04	9.371e+04	1.278e+05
5	2.544e+11	1.263e+05	2.255e+05	3.245e+05	4.004e+05	6.833e+05
6	2.544e+11	3.450e+05	6.307e+05	9.161e+05	1.459e+06	2.935e+06
7	2.112e+16	8.790e+05	1.631e+06	2.383e+06	4.792e+06	1.098e+07
8	4.770e+18	2.141e+06	4.013e+06	5.882e+06	1.464e+07	3.736e+07
9	9.439e+20	5.051e+06	9.536e+06	1.401e+07	4.237e+07	1.188e+08
10	1.662e+23	1.165e+07	2.210e+07	3.254e+07	1.176e+08	3.584e+08

Table C.20.: Raw data for Fig. 5.9.

Level	**Grid type**					
	Reg. SG a	Reg. SG b	GaSG a	GaSG b	GaSG c	GaSG d
1	1	1	1	1	1	1
2	1.569e+03	2.101e+03	2.101e+03	2.101e+03	2.101e+03	2.101e+03
3	1.232e+06	2.209e+06	1.425e+04	1.865e+04	4.251e+04	6.129e+04
4	6.462e+08	1.550e+09	5.446e+04	7.646e+04	1.957e+05	4.151e+05
5	2.544e+11	8.165e+11	1.667e+05	2.415e+05	6.470e+05	2.033e+06
6	2.544e+11	3.444e+14	4.547e+05	6.703e+05	1.839e+06	8.492e+06
7	2.112e+16	1.212e+17	1.158e+06	1.726e+06	4.802e+06	3.208e+07
8	4.770e+18	3.658e+19	2.819e+06	4.232e+06	1.189e+07	1.126e+08
9	9.439e+20	9.670e+21	6.651e+06	1.003e+07	2.838e+07	3.735e+08
10	1.662e+23	2.275e+24	1.533e+07	2.322e+07	6.599e+07	1.183e+09

- **Reg. SG a:** Regular Sparse Grid of original dim 784
- **Reg. SG b:** Regular Sparse Grid of extended dim 1,050
- **GaSG a:** GaSG with `DN+NoLay`
- **GaSG b:** GaSG with `DN+NPLay`
- **GaSG c:** GaSG with `DN+APLay`
- **GaSG d:** GaSG with `DN+AFLay`

Table C.21.: Raw data for Fig. 5.11.

Level	**Grid type**					
	Reg. SG	GaSG a	GaSG b	GaSG c	GaSG d	GaSG e
1	1	1	1	1	1	1
2	6.145e+03	6.145e+03	6.145e+03	6.145e+03	6.145e+03	6.145e+03
3	1.889e+07	4.224e+04	5.453e+04	5.453e+04	1.021e+05	1.021e+05
4	3.871e+10	1.620e+05	2.235e+05	2.317e+05	4.616e+05	7.555e+05
5	5.953e+13	4.969e+05	7.058e+05	7.631e+05	1.515e+06	4.049e+06
6	7.326e+16	1.357e+06	1.959e+06	2.213e+06	4.292e+06	1.807e+07
7	7.515e+19	3.458e+06	5.043e+06	5.953e+06	1.119e+07	7.146e+07
8	6.610e+22	8.423e+06	1.237e+07	1.524e+07	2.765e+07	2.590e+08
9	5.089e+25	1.988e+07	2.932e+07	3.770e+07	6.594e+07	8.788e+08
10	3.484e+28	4.583e+07	6.786e+07	9.092e+07	1.532e+08	2.830e+09

- **GaSG a:** GaSG with `DN+NoCol`
- **GaSG b:** GaSG with `DN+PairCol`
- **GaSG c:** GaSG with `DN+FullCol`
- **GaSG d:** GaSG with `PCPSof(DN)Col`
- **GaSG e:** GaSG with `FCPSof(DN)Col`

List of Figures

Bibliography

[1] Ahn, C. P. et al. "The Tenth Data Release of the Sloan Digital Sky Survey: First Spectroscopic Data from the SDSS-III Apache Point Observatory Galactic Evolution Experiment". In: *The Astrophysical Journal Supplement Series* 211.2 (Mar. 2014), p. 17.

[2] Anguiano, V. B. B. "Visualization of High Dimensional Models within the SG++ Data Mining Pipeline". Student project. Technical University of Munich, Oct. 2019.

[3] Anzt, H., Chow, E., Saak, J., and Dongarra, J. "Updating Incomplete Factorization Preconditioners for Model Order Reduction". In: *Numerical Algorithms* 73.3 (Feb. 2016), 611–630.

[4] Arel, I., Rose, D. C., and Karnowski, T. P. "Deep machine learning-a new frontier in artificial intelligence research [research frontier]". In: *IEEE computational intelligence magazine* 5.4 (2010), pp. 13–18.

[5] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. "Models and issues in data stream systems". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2002, pp. 1–16.

[6] Banderier, C. and Schwer, S. "Why Delannoy numbers?" In: *Journal of statistical planning and inference* 135.1 (2005), pp. 40–54.

[7] Bellman, R. E. "Adaptive control processes: a guided tour". In: Princeton university press, 1961.

[8] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems*. 2011, pp. 2546–2554.

[9] Beutelspacher, A. *Lineare Algebra: Eine Einführung in die Wissenschaft der Vektoren, Abbildungen und Matrizen*. Gießen: Springer, 2014.

[10] Bisseling, R. H. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. New York, NY, USA: Oxford University Press, Inc., 2004.

[11] Blackford, L. S. et al. *ScaLAPACK User's Guide*. Ed. by Dongarra, J. J. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[12] Blackford, L. S. et al. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[13] Bode, V. "Parallelization of a Sparse Grids Batch Classifier". Bachelor's thesis. Technical University of Munich, Sept. 2017.

[14] Boschko, D. “Orthogonal Matrix Decomposition for Adaptive Sparse Grid Density Estimation Methods”. Bachelor’s thesis. Technical University of Munich, Sept. 2017.

[15] Boser, B. E., Guyon, I. M., and Vapnik, V. N. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT ’92. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1992, pp. 144–152.

[16] Brimkov, V. E., Moroni, D., and Barneva, R. “Combinatorial relations for digital pictures”. In: *International Conference on Discrete Geometry for Computer Imagery*. Springer. 2006, pp. 189–198.

[17] Brochu, E., Cora, V. M., and Freitas, N. de. “A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning”. In: *CoRR* abs/1012.2599 (2010). arXiv: `1012.2599`.

[18] Bungartz, H.-J. and Griebel, M. “Sparse Grids”. In: *Acta Numerica* 13 (May 2004), pp. 147–269.

[19] Chang, X.-W. and Stehlé, D. “Rigorous perturbation bounds of some matrix factorizations”. In: *SIAM journal on matrix analysis and applications* 31.5 (2010), pp. 2841–2859.

[20] Chollet, F. *About the shortcomings of MNIST*. `https://twitter.com/fchollet/status/852594987527045120`. Apr. 2017.

[21] Dierig, C. *Matthias Hartmann: “Daten sind das Gold des 21. Jahrhunderts”*. `https://www.welt.de/wirtschaft/article127418980/Daten-sind-das-Gold-des-21-Jahrhunderts.html`. Accessed: 2020-01-28. Apr. 2014.

[22] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern classification and scene analysis*. Vol. 3. Wiley New York, 1973.

[23] Feurer, M., Springenberg, J. T., and Hutter, F. “Initializing bayesian hyperparameter optimization via meta-learning”. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[24] Fryer, M. “A review of some non-parametric methods of density estimation”. In: *IMA Journal of Applied Mathematics* 20.3 (1977), pp. 335–354.

[25] Fuchsgruber, D. “Integration of SGDE-based Classification into the SG++ Datamining Pipeline”. Bachelor’s thesis. Technical University of Munich, Aug. 2018.

[26] Galerkin, B. G. *Rods and plates: series in some questions of elastic equilibrium of rods and plates*. National Technical Information Service, 1968.

[27] Garcke, J., Griebel, M., and Thess, M. “Data mining with sparse grids”. In: *Computing* 67.3 (2001), pp. 225–253.

[28] Garcke, J. “A dimension adaptive sparse grid combination technique for machine learning”. In: *Anziam Journal* 48 (2007), pp. 725–740.

[29] Garcke, J. et al. “Maschinelles Lernen durch Funktionsrekonstruktion mit verallgemeinerten dünnen Gittern”. PhD thesis. University of Bonn, Germany, 2004.

[30] Garcke, J. and Griebel, M. “On the parallelization of the sparse grid approach for data mining”. In: *International Conference on Large-Scale Scientific Computing*. Springer. 2001, pp. 22–32.

[31] Garcke, J., Hegland, M., and Nielsen, O. “Parallelisation of sparse grids for large scale data analysis”. In: *International Conference on Computational Science*. Springer. 2003, pp. 683–692.

[32] Garcke, J., Hegland, M., and Nielsen, O. “Parallelisation of sparse grids for large scale data analysis”. In: *The ANZIAM Journal* 48.1 (2006), pp. 11–22.

[33] Glas, K. “Exploitation of Component Grid Symmetries for Sparse Grid Density Estimation with the Combination Method”. Bachelor’s thesis. Technical University of Munich, Oct. 2019.

[34] Golub, G. H. and Van Loan, C. F. *Matrix Computations (3rd Ed.)* Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[35] Gram, J. P. *Undersøgelser angående Mængden af Primtal under en given Grænse*. Lunos, 1884.

[36] Gramacki, A. *Nonparametric kernel density estimation and its computational aspects*. Springer, 2018.

[37] Griebel, M. “Adaptive Sparse Grid Multilevel Methods for Elliptic PDEs Based on Finite Differences”. In: *Computing* 61.2 (Oct. 1998), pp. 151–179.

[38] Griebel, M. “The Combination Technique for the Sparse Grid Solution of PDE’s on Multiprocessor Machines”. In: *Parallel Processing Letters* 02 (Sept. 2000).

[39] Griebel, M., Schneider, M., and Zenger, C. *A Combination Technique for the Solution of Sparse Grid Problems*. Tech. rep. 1990.

[40] Hand, D. and Yu, K. “Idiot’s Bayes: Not So Stupid after All?” In: *International Statistical Review* 69 (May 2007), pp. 385–398.

[41] Hazan, E., Klivans, A. R., and Yuan, Y. “Hyperparameter Optimization: A Spectral Approach”. In: *CoRR* abs/1706.00764 (2017). arXiv: `1706.00764`.

[42] Hegland, M. “Adaptive sparse grids”. In: *Anziam Journal* 44 (2002), pp. 335–353.

[43] Hegland, M., Hooker, G., and Roberts, S. “Finite element thin plate splines in density estimation”. In: *ANZIAM Journal* 42 (July 2009), p. 712.

[44] Heinecke, A. “Boosting Scientific Computing Applications through Leveraging Data Parallel Architectures”. Dissertation. München: Technische Universität München, 2014.

[45] Heinecke, A., Peherstorfer, B., Pflüger, D., and Song, Z. “Sparse grid classifiers as base learners for AdaBoost”. In: *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2012, pp. 161–166.

[46] Heinecke, A. and Pflüger, D. “Multi-and many-core data mining with adaptive sparse grids”. In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. 2011, pp. 1–10.

[47] Hestenes, M. R., Stiefel, E., et al. “Methods of conjugate gradients for solving linear systems”. In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436.

[48] Householder, A. S. “Unitary Triangularization of a Nonsymmetric Matrix”. In: *J. ACM* 5.4 (Oct. 1958), pp. 339–342.

[49] Hunter, J. D. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95.

[50] Johansson, L. “Einsatz der Dünngitterklassifikation für Benchmark-Probleme”. Diploma thesis. Apr. 2008.

[51] Keys, R. “Cubic convolution interpolation for digital image processing”. In: *IEEE transactions on acoustics, speech, and signal processing* 29.6 (1981), pp. 1153–1160.

[52] Khakhutskyy, V. “Sparse Grids for Big Data: Exploiting Parsimony for Large-Scale Learning”. Dissertation. Technical University of Munich, 2016.

[53] Khakhutskyy, V. and Hegland, M. “Sparse Grids and Applications - Stuttgart 2014”. In: ed. by Pflüger, D. and Garcke, J. Vol. 109. LNCSE. Springer International Publishing, Mar. 2016. Chap. Spatially-Dimension-Adaptive Sparse Grids for Online Learning, pp. 133–162.

[54] Kiefer, J. “Sequential minimax search for a maximum”. In: *Proceedings of the American mathematical society* 4.3 (1953), pp. 502–506.

[55] Kier, L. B., Seybold, P. G., and Cheng, C.-K. *Modeling chemical systems using cellular automata*. Springer Science & Business Media, 2005.

[56] Kluger, Y., Basri, R., Chang, J. T., and Gerstein, M. “Spectral biclustering of microarray data: coclustering genes and conditions”. In: *Genome research* 13.4 (2003), pp. 703–716.

[57] Koepke, E. “Optimizing Hyperparameters in the SG++ Datamining Pipeline”. Bachelor’s thesis. Technical University of Munich, Apr. 2018.

[58] Kowitz, C. and Hegland, M. “The sparse grid combination technique for computing eigenvalues in linear gyrokinetics”. In: *International Conference on Computational Science 2013*. Procedia Computer Science. Elsevier, June 2013, pp. 449–458.

[59] Kristan, M., Skočaj, D., and Leonardis, A. “Online kernel density estimation for interactive learning”. In: *Image and Vision Computing* 28.7 (2010), pp. 1106–1116.

[60] Krizhevsky, A., Hinton, G., et al. “Learning multiple layers of features from tiny images”. In: (2009).

[61] Laney, D. *3D Data Management: Controlling Data Volume, Velocity, and Variety*. `https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf`. Feb. 2001.

[62] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[63] Leibniz Supercomputing Centre. *Linux Cluster CoolMUC-2 specifications*. `https://doku.lrz.de/display/PUBLIC/CoolMUC-2`.

[64] Lettrich, M. "Iterative Incomplete Cholesky Decomposition for Datamining using Sparse Grids". Student project. Technical University of Munich, May 2017.

[65] Maaten, L. v. d. and Hinton, G. "Visualizing data using t-SNE". In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605.

[66] McCulloch, W. S. and Pitts, W. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[67] Montavon, G., Samek, W., and Müller, K.-R. "Methods for interpreting and understanding deep neural networks". In: *Digital Signal Processing* 73 (2018), pp. 1–15.

[68] Murphy, K. P. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[69] Obersteiner, M. and Bungartz, H.-J. "A Spatially Adaptive Sparse Grid Combination Technique for Numerical Quadrature". In: *Sparse Grids and Applications - Munich 2018*. Preprint. 2018.

[70] Palmer, M. *Clive Humby: "Data is the New Oil"*. `https://ana.blogs.com/maestros/2006/11/data_is_the_new.html`. Accessed: 2020-01-28. Nov. 2006.

[71] Parliament, T. E. *Automated decision-making processes: Ensuring consumer protection, and free movement of goods and services*. `https://www.europarl.europa.eu/doceo/document/TA-9-2020-0032_EN.pdf`. Accessed: 2020-03-15. Feb. 2020.

[72] Parzen, E. "On Estimation of a Probability Density Function and Mode". In: *Ann. Math. Statist.* 33.3 (Sept. 1962), pp. 1065–1076.

[73] Pearson, K. "X. Contributions to the mathematical theory of evolution.—II. Skew variation in homogeneous material". In: *Philosophical Transactions of the Royal Society of London.(A.)* 186 (1895), pp. 343–414.

[74] Pearson, K. "Mathematical Contributions to the Theory of Evolution. XIX. Second Supplement to a Memoir on Skew Variation". In: *Philosophical Transactions of the Royal Society of London Series A* 216 (Jan. 1916), pp. 429–457.

[75] Peherstorfer, B. "Model Order Reduction of Parametrized Systems with Sparse Grid Learning Techniques". Dissertation. München: Technische Universität München, 2013.

[76] Peherstorfer, B., Franzelin, F., Pflüger, D., and Bungartz, H.-J. “Classification with Probability Density Estimation on Sparse Grids”. Deutsch. In: *Sparse Grids and Applications - Munich 2012*. Ed. by Garcke, J. and Pflüger, D. Vol. 97. Lecture Notes in Computational Science and Engineering. Springer International Publishing, Jan. 2014, pp. 255–270.

[77] Peherstorfer, B., Pflüger, D., and Bungartz, H.-J. “Clustering Based on Density Estimation with Sparse Grids”. In: *KI 2012: Advances in Artificial Intelligence*. Lecture Notes in Computer Science 7526. Springer, Oct. 2012.

[78] Peherstorfer, B., Pflüger, D., and Bungartz, H.-J. “Density Estimation with Adaptive Sparse Grids for Large Data Sets”. In: *Proceedings of the 2014 SIAM International Conference on Data Mining*. Ed. by Zaki, M., Obradovic, Z., Tan, P. N., Banerjee, A., Kamath, C., and Parthasarathy, S. Philadelphia, PA: Society for Industrial and Applied Mathematics, Apr. 2014. Chap. 50, pp. 443–451.

[79] Pflüger, D. “Spatially Adaptive Sparse Grids for High-Dimensional Problems”. Dissertation. München: Institut für Informatik, Technische Universität München, Feb. 2010.

[80] Pflüger, D. “Spatially Adaptive Refinement”. In: *Sparse Grids and Applications*. Ed. by Garcke, J. and Griebel, M. Lecture Notes in Computational Science and Engineering. Berlin Heidelberg: Springer, Oct. 2012, pp. 243–262.

[81] Rasmussen, C. and Williams, C. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, Jan. 2006, p. 248.

[82] Rösel, N. “Combigrid Based Dimensional Adaptivity for Sparse Grid Density Estimation and Classification”. Bachelor’s thesis. Technical University of Munich, Apr. 2019.

[83] Rosenblatt, M. “Remarks on a multivariate transformation”. In: *The annals of mathematical statistics* 23.3 (1952), pp. 470–472.

[84] Rosenfeld, A. *Digital picture processing*. Academic press, 1976.

[85] Samek, W., Wiegand, T., and Müller, K.-R. “Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models”. In: *arXiv preprint arXiv:1708.08296* (2017).

[86] Schopohl, J. “Domain Parallelization of SGDE based Classification”. Bachelor’s thesis. Technical University of Munich, Apr. 2019.

[87] Scott, D. W. and Sheather, S. J. “Kernel density estimation with binned data”. In: *Communications in Statistics-Theory and Methods* 14.6 (1985), pp. 1353–1359.

[88] Sherman, J. and Morrison, W. J. “Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix”. In: *Ann. Math. Statist.* 21.1 (Mar. 1950), pp. 124–127.

[89] Sieler, A. “Refinement and Coarsening of Online-Offline Data Mining Methods with Sparse Grids”. Bachelor’s thesis. Technical University of Munich, Mar. 2016.

[90] Silverman, B. W. *Density estimation for statistics and data analysis*. Vol. 26. CRC press, 1986.

[91] Smolyak, S. A. “Quadrature and interpolation formulas for tensor products of certain classes of functions”. In: *Doklady Akademii Nauk*. Vol. 148. 5. Russian Academy of Sciences. 1963, pp. 1042–1045.

[92] Stewart, G. W. *Matrix Algorithms: Volume 1: Basic Decompositions*. Vol. 1. Siam, 1998.

[93] Stone, C. J. “Optimal global rates of convergence for nonparametric regression”. In: *The annals of statistics* (1982), pp. 1040–1053.

[94] Sultana, F., Sufian, A., and Dutta, P. “Advancements in Image Classification using Convolutional Neural Network”. In: *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)* (Nov. 2018).

[95] Sylvester, J. J. “XIX. A demonstration of the theorem that every homogeneous quadratic polynomial is reducible by real orthogonal substitutions to the form of a sum of positive and negative squares”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 4.23 (1852), pp. 138–142.

[96] Toffoli, T. and Margolus, N. *Cellular Automata Machines: A New Environment for Modeling*. Cambridge, MA, USA: MIT Press, 1987.

[97] Toffoli, T. and Margolus, N. *Cellular automata machines: a new environment for modeling*. MIT press, 1987.

[98] Uddin, M. S., Kuh, A., Weng, Y., and Ilić, M. “Online bad data detection using kernel density estimation”. In: *2015 IEEE Power & Energy Society General Meeting*. IEEE. 2015, pp. 1–5.

[99] Udupa, J. K., Srihari, S. N., and Herman, G. T. “Boundary detection in multidimensions”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1 (1982), pp. 41–50.

[100] Vapnik, V. *The nature of statistical learning theory*. Springer science & business media, 2013.

[101] Verhulst, P.-F. “Notice sur la loi que la population suit dans son accroissement”. In: *Corresp. Math. Phys.* 10 (1838), pp. 113–126.

[102] Waegemans, T. “Image Classification with Geometrically Aware Sparse Grids”. Bachelor’s thesis. Technical University of Munich, Oct. 2017.

[103] Wang, Z. and Scott, D. W. “Nonparametric density estimation for high-dimensional data—Algorithms and applications”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* (2019), e1461.

[104] Weber, S. “Exploiting the Data Hierarchy with Geometry Aware Sparse Grids for Image Classification”. Bachelor’s thesis. Technical University of Munich, Aug. 2019.

[105] Xiao, H., Rasul, K., and Vollgraf, R. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: `cs.LG/1708.07747 [cs.LG]`.

[106] Zenger, C. “Sparse Grids”. In: *Parallel Algorithms for Partial Differential Equations*. Ed. by Hackbusch, W. Notes on Numerical Fluid Mechanics 31. Vieweg, 1991, pp. 241–251.