Fakultät für Informatik
Technische Universität München

A Spatially Adaptive and Massively Parallel Implementation of the Fault-Tolerant Combination Technique

Michael Johannes Obersteiner

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. Tobias Nipkow, Ph.D.

Prüfende der Dissertation:
1. Prof. Dr. Hans-Joachim Bungartz
2. Prof. Dr. Dirk Pflüger

Die Dissertation wurde am 18.06.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.10.2021 angenommen.

In loving memory of my dad.

Abstract

High-dimensional methods have gained increasing interest in the past decades due to novel machine learning applications and the growing compute power of modern systems. One fundamental problem of high dimensional algorithms is the typically exponential growth of the computing time with the number of dimensions. For grid-based methods this *curse of dimensionality* can be mitigated by the Sparse Grid Combination Technique. Unfortunately, even with the reduced complexity of Sparse Grids, many use cases are still computationally involved. Hence, an HPC implementation of the Combination Technique is required that runs efficiently on modern and future supercomputers. Furthermore, the method should be fault-tolerant due to the higher failure rates in upcoming exascale clusters.

In this work, we address these challenges by presenting novel techniques to increase the scalability and robustness of the Combination Technique. In particular, we introduce an asynchronous variant and enhance the existing fault-tolerant implementation. Furthermore, we discuss how to improve time step sizes to optimize the computational complexity for time-dependent PDEs. To analyze these new concepts, we show results for a real-world application from plasma physics. As a second step, we introduce two novel ways for generalizing the Combination Technique to allow for spatial adaptivity. This represents a fundamental change in the Combination Technique that is constructed based on a combination of regular subgrids. In contrast to this, the novel adaptive methods respectively generate rectilinear grids and block adaptive grids that preserve the most important features of the Combination Technique: the black-box property, their parallel nature, and the error cancellation. We then test these adaptive techniques for typical Sparse Grid applications such as quadrature, interpolation, uncertainty quantification, and machine learning. The results indicate that the new methods can help to tailor the grids towards the application scenarios, thereby reducing the number of needed grid points. With these improvements, it is now possible to scale the Combination Technique to larger core numbers with possibly unreliable hardware and to apply it to a much larger range of applications due to spatially adaptive refinement.

Zusammenfassung

In den letzten Jahrzehnten gewannen hochdimensionale Verfahren immer mehr an Bedeutung. Ein Grund hierfür sind neue Anwendungen im Bereich des maschinellen Lernens sowie die wachsende Rechenleistung moderner Computersysteme. Ein fundamentales Problem von hochdimensionalen Algorithmen ist der üblicherweise exponentielle Zuwachs an Rechenzeit mit der Anzahl der Dimensionen. Dieser *Fluch der Dimension* kann jedoch durch die Dünngitter-Kombinationstechnik abgeschwächt werden. Leider bleibt in vielen Fällen der durch Verwendung von Dünngittern verringerte Rechenaufwand noch immer sehr hoch. Aus diesem Grund benötigt es eine HPC-Implementierung der Kombinationstechnik, welche auch effizient auf aktuellen und zukünftigen Supercomputern verwendet werden kann. Zusätzlich sollte das Verfahren fehltertolerant sein, um trotz der zunehmenden Fehlerraten zukünftiger Exascalecomputer zuverlässig arbeiten zu können.

In dieser Arbeit werden wir uns beiden Aspekten widmen, indem wir neue Techniken vorstellen, welche die Skalierbarkeit und die Robustheit der Kombinationstechnik steigern. Wir präsentieren insbesondere eine asynchrone Variante und verbessern die existierenden Implementierungen zur Fehlertoleranz. Außerdem zeigen wir, wie die Zeitschrittweiten auf den einzelnen Komponentengittern bei der Lösung von zeitabhängigen Differentialgleichungen optimiert werden können. Die Wirkungsweise und Effektivität dieser neuen Konzepte werden dann anhand von realistischen Anwendnungsszenarien in der Plasmaphysik analysiert. In einem zweiten Schritt stellen wir zwei neue Möglichkeiten vor, um räumliche Adaptivität in der Kombinationstechnik zu ermöglichen. Diese Neuerung repräsentiert einen fundamentalen Paradigmenwechsel für die Kombinationstechnik, da die ursprüngliche Methode auf regulären Gittern basiert. Anstelle dessen erzeugen die neuen Methoden rechtwinklige bzw. blockadaptive Gitter. Gleichzeitig erhalten die neuen Methoden die wichtigsten Eigenschaften der Kombinationstechnik: die Blackbox-Eigenschaft, die inherente Parallelität und die Fehlerauslöschung. Als nächsten Schritt demonstrieren wir die Wirkungsweise der neuen adaptiven Methoden anhand zahlreicher Tests aus der klassischen Numerik, wie etwa der Quadratur und Interpolation, aus der Uncertainty Quantification und aus dem maschinellen Lernen. Die Ergebnisse zeigen, dass die neuen Ansätze dabei helfen können, die Gitter an den jeweiligen Problemfall anzupassen. Mithilfe dieser Neuerungen an der Kombinationstechnik ist es jetzt möglich, die Implementierungen bis zu noch größeren Prozessorzahlen mit möglicherweise fehleranfälliger Hardware zu skalieren. Zusätzlich ermöglicht die räumliche Adaptivität ein breiteres Anwendungsspektrum.

Acknowledgments

At this point, I would like to thank all the people that have contributed to this work in one or the other way and who supported me on this journey.

First of all, I like to thank my supervisor Hans-Joachim Bungartz for his continuous support throughout my dissertation project. This includes strategic advice for conferences and journals, late night email conversations, discussions of the latest scientific ideas, and the lively philosophical or political debates at the Halb3 meetings.

Next, I would like to thank my project partners Alfredo Parra, Mario Heene, Theresa Pollinger, Johannes Rentrop, Rafael Lago, Tilman Dannert, Dirk Pflüger, Michael Griebel, and Frank Jenko that created a productive and enjoyable working environment for all of these years. Even in times where we did not succeed with all our ambitions, we always found new and promising ways to expand our framework for the Combination Technique.

My work with Sparse Grids would not have been possible without the help of the Sparse Grid groups in Munich and Stuttgart. They were at all times patient with me and together we unraveled many of the mysteries of Sparse Grids and beyond. I would like to specially mention Paul Sârbu, Ionuţ Farcaş, Kilian Röhner, Alfredo Parra, Christoph Kowitz, and Valeriy Khakhutskyy.

Another major contribution was made by all of my (over 20) students that collaborated with me during their student projects. Without their scientific curiosity and motivation, many of the new methods in this thesis would have never been investigated.

Moreover, I would like to express my deepest gratitude for my colleagues at the Chair of Scientific Computing in Computer Science in Garching who not just represented a never ending source of advice and guidance, but they also managed to created a relaxing and positive atmosphere. Without them many of the working hours at the office would have been far less enjoyable. Although it would have been from time to time more productive to focus solely on my work, I would always prefer to chat in the coffee room or grab an afternoon snack at the bakery with them.

A special thanks goes to my advisor of my Bachelor's and Master's thesis Nikola Tchipev who made me aware of the beauties of scientific computing and helped me to develop my scientific curiosity. Without him I would have maybe never even started this PhD project.

Furthermore, I want to mention Paul Sârbu, Friedrich Menhorn, Johannes Rentrop, Theresa Pollinger, Tilman Dannert, and of course Hans-Joachim Bungartz for proofreading my thesis and contributing to its final state.

Last but not least, I am thankful for my family, my friends, and my girlfriend for supporting me during all this time. You did not just help me with my work but also showed me when it is best to stop and enjoy my life. Especially, I want to mention my

dad who sadly passed away on 01.06.2021. He supported me in every possible way throughout my whole live. I'll be forever grateful for that.

Contents

1 Introduction

Over the past decades, computer systems have undergone a tremendous increase in computing power that changed science fundamentally. Previously unfeasible computations are now in reach and long forgotten methods such as neural networks are gaining enormous popularity. As a consequence, mathematical modeling, simulations, and machine learning approaches are now applied in almost every discipline. For many applications this is due to the new possibility to perform more accurate calculations and to reach higher dimensions. For example, in machine learning tasks it is common that hundreds of features are available which form extremely high-dimensional feature spaces.

However, such high-dimensional problems come with completely new challenges as the cost of discretizing a high-dimensional space grows exponentially with the number of dimensions d. This effect is known as the expression *curse of dimensionality* which was coined by Bellmann in [11] in the context of dynamic programming. This exponential growth is so critical that not even the increasing compute power of modern systems can compensate for it. As a consequence, new methods were invented that deal with such high dimensionalities.

Sparse Grids represent one of these new approaches that can scale to much larger number of dimensions by a cost-efficient discretization scheme. The main advantage of Sparse Grids is that they reduce the growth rate of the point numbers. In particular, the number of points only increase exponentially with $\mathcal{O}(\tilde{N}\log(\tilde{N})^{d-1})$ instead of $\mathcal{O}(\tilde{N}^d)$ where $\tilde{N}$ is the number of points per dimension in a regular grid. This allows to tackle much larger dimensionalities with grid-based techniques.

One especially interesting variation of Sparse Grids is the Sparse Grid Combination Technique. Instead of working on a Sparse Grid explicitly, this method splits the calculation into various independent and cheap subproblems on regular but anisotropic grids. These subproblem can then be solved in parallel with arbitrary full grid solvers. In addition, it is rather simple to implement, which is probably the main reason why it is a frequent choice in Sparse Grid literature.

The application spectrum of the Combination Technique ranges from classical numerical tasks such as quadrature, interpolation, and (time-dependent) partial differential equations (PDEs) to more recent application areas like uncertainty quantification and grid-based machine learning techniques. Especially, for time-dependent PDEs the method offers a high potential for parallelization and reducing the required number of grid points, but it also comes with certain challenges as regular synchronous combination steps are introduced.

This was the starting point for the project EXAHD [97, 72] which was funded by the Germand Research Foundation (DFG) as part of its Priority Programme *Software*

for Exascale Computing (SPPEXA). This collaborative project between the University of Bonn, the University of Stuttgart, the Max-Planck Institute for Plasma Physics, the Max-Planck Computing and Data Facility, and the Technical University of Munich focused on one high-dimensional case study in plasma physics, namely microturbulence simulations in nuclear fusion reactors. These five-dimensional simulations are still highly limited by current computing systems due to the extremely large numbers of grids points. With the help of current and future supercomputers, the Combination Technique could provide more accurate predictions for the stability of the plasma and improve the design and development of new fusion reactors.

For this purpose, a new framework was created to run plasma physics simulations at the extreme scale. This effort revealed completely new challenges from high performance computing for the Combination Technique. First, the recombination step needs to be perform very efficiently and is required to scale up to millions of cores. Second, the long simulation times result in a large risk of failing components. The framework should therefore be fault-tolerant. Last, the numerical convergence of the method needs to be shown for real-world and non-linear test cases.

However, there are also challenges from the Combination Technique itself. Due to the regular structure of the grids that are generated by the method, it is not possible to spatially adapt the grid to the application at hand. This prevents spatial adaptivity and limits the efficiency of the method for localized problem. Hence, a spatially adaptive Combination Technique is required to increase the performance and to apply it to a wider range of applications.

In this work, we tackle both of these problems. We will demonstrate how to improve the implementations of the recombination by using optimal time-steps and an asynchronous communication. Furthermore, we refine the existing implementations for the Fault-Tolerant Combination Technique and apply and analyze it for more realistic scenarios. In addition, we apply the Combination Technique to non-linear plasma physics runs and discuss the current shortcomings and opportunities with these simulations.

In a second step, we outline two novel generalizations of the Combination Technique that allow for spatially adaptivity. One of the core concepts of the standard Combination Technique is that it exploits the regular grid structure of the component grids to guarantee an efficient error cancellation. This concept does not allow for an efficient spatially adaptive refinement. Hence, the novel spatially adaptive methods do not follow this principle. In contrast to this, they create arbitrary rectilinear grids by refining each dimension individually or form block-adaptive grids with an octree-like splitting of the domain, respectively. At the same time, the refinement procedures preserve the important error cancellation property of the Combination Technique, the black-box property, and its inherent parallel nature. We then apply these new approaches to a wide variety of applications to show that these fundamental concepts can be applied to various localized problems. We therefore consider more general applications in this part and not the specialized PDE simulations from plasma physics.

The remainder of this work is structured as follows. In Chapter 2 we summarize the theoretical foundations. This includes an overview of fault tolerance, followed by a description of Sparse Grids, the Combination Technique and its application to time-

dependent PDEs. We will focus on the *high performance computing* (HPC) aspects of fault tolerance and give a brief overview of the different techniques that have been developed in the past decades. We also introduce the different variations of Sparse Grids and the Combination Technique including spatial and dimension adaptivity.

Thereafter, we discuss in Chapter 3 the algorithmic and implementation details of our Combination Technique framework for time-dependent PDEs. Here, we first summarize the previous developments and focus then on the novel aspects in respect to fault tolerance. Next, we cover a new analysis for the selection of appropriate time steps which optimize the computational complexity. Moreover, we show a shared-memory implementation of the combination step and an asynchronous variant of the Combination Technique that reduces the communication overhead. The chapter is concluded by an evaluation of the novel techniques for numerical simulations. This also involves an overview of the main application code `GENE` from plasma physics.

In Chapter 4 we outline the two novel spatially adaptive variants of the Combination Technique. First, we introduce the dimension-wise spatial refinement, which uses rectilinear grids in the Combination Technique. Next, the Split-Extend method is described, which allows for block-adaptive refinement with the Combination Technique. In addition, we describe the implementation aspects of the newly created framework for the spatially adaptive Combination Technique.

This more theoretical chapter will be followed by multiple numerical case studies in Chapter 5 that demonstrate the effectiveness of the novel adaptive schemes. To show their broad applicability, we present results for the basic numerical applications quadrature and interpolation as well as for uncertainty quantification and density estimation.

Finally, we conclude this work in Chapter 6. After a summary of the main results, we highlight the new insights that we obtained. Furthermore, we describe where we see the highest potential for future work on spatial adaptivity and HPC optimizations for the Combination Technique.

2 Foundations

This chapter gives a detailed overview of the background of this work. First, we briefly summarize the main aspects of fault tolerance for *high performance computing* (HPC) in Section 2.1. Here, we start with an explanation of *checkpoint-restart* which is one of the oldest and most popular methods in fault tolerance. This method saves the program state to a persistent memory in order to be able to restore lost information after a fault occurs. However, this method does not scale to arbitrary process numbers due to the slow memory access. We therefore cover common alternatives of the method. One of these approaches is *algorithm-based fault tolerance* where the specific properties of the applied algorithm are used to get a cheap possibility to restore or at least tolerate lost data. An example of such an algorithm class are hierarchical methods that utilize the hierarchical structure for recovering from faults.

In Section 2.2, we summarize the basics of Sparse Grids which represent one promising candidate of these hierarchical methods. In addition, Sparse Grids provide an efficient discretization scheme for high-dimensional problems that usually suffer from the *curse of dimensionality*. We show in detail their advantage compared to classical full grids and the construction of the hierarchical basis. We also discuss variations of the standard approach such as adaptivity and the usage of higher order basis functions. These adaptations can further enhance the efficiency for certain problem classes.

Last, Section 2.3 covers the Sparse Grid Combination Technique which is a commonly used alternative to an explicit Sparse Grid implementation. This section gives a detailed overview of the error cancellation, which is the fundamental concept behind the Combination Technique. Furthermore, the different generalizations are shown that allow for adding adaptivity and fault tolerance. The latter, is achieved by utilizing the hierarchical nature of the method to recover from faults. Finally, we outline the key aspects for the application of the Combination Technique to time-dependent PDEs which represents one of our main use cases. Here, we mainly focus on the recombination approach that combines the solution in regular combination intervals.

2.1 Fault tolerance in HPC

Modern HPC systems focus more and more on parallel computation across millions of computing units. An example from the current top 500 list [1] is the new supercomputer Fugaku with 7,299,072 cores. This development has caused an increasing interest for fault tolerance in the high performance community. A reason for this is that the growing complexity of modern processor architectures and the increasing number of

[1] https://www.top500.org/

components in compute clusters result in more and more complex systems. This complexity comes with an increasing potential for failing components and therefore erroneous results of the computations. In addtion, the *mean time between failures* decreases linearly with the number of components of a system [27, section 1.3.2.1], which is an imminent threat for large-scale computations on future exascale computers. As a consquence, current supercomputers already face several failure events per day [102]. This trend is expected to continue and first estimates suggest that the upcoming exascale clusters could fail as often as every 30 minutes [113]. In the following we will give a brief overview of the fault tolerance approaches. This section mainly follows [27].

A failure event can have different causes. It might arise due to a hardware failure such as a process or network failure or it might be caused by a component of the system that produces erroneous results. In literature one often differentiates between faults that are detected automatically by the system (hard faults) and errors that stay undetected (soft faults or silent data corruption). For the latter, specific routines have to be applied to find the failing component and to recover a correct system state. We will not discuss the different sources or variants of the errors but the means of how to react to general failure events and how to resolve them. In the following, a fault or failure event refers to an erroneous system state due to a failing component or an incorrect computation.

2.1.1 Check-point restart

One of the first and most widely used approaches to add additional fault tolerance is the *coordinated checkpoint-restart* concept [27, section 1.2.2]. This technique saves relevant data of the application state at a synchronization point to a *checkpoint* in persistent memory and *restarts* from this checkpoint if a fault occurs afterwards. This approach is rather easy to implement but it introduces certain problems. First, we have to know which data is necessary to restart an application. Second, writing and reading of data is slow due to the huge gap between the memory bandwidth and the peak performance of the processing units. Third, the coordinated checkpoints introduce a synchronization barrier that might cause idling of processes and a high load on the memory bus. Last, we have to select an appropriate interval between checkpoints. If a checkpoint is written too often, it introduces a severe overhead if no faults occur. However, if the checkpoint interval is too large, it is very costly to recompute the lost computation.

Fortunately, there are techniques to determine the optimal checkpoint interval. In case we know the *mean-time-between-failure* M of a system and the time it takes to write a checkpoint W, Young [120] determined with a rather simplistic first-order model the optimal checkpoint interval of $\sqrt{2WM}$. A more realistic evaluation that also includes a restart time was made in [23]. Although we can often approximate such an interval in practice, it might not be feasible to use checkpoint restart due to the slow memory access.

An alternative to the original *checkpoint-restart* approach is *uncoordinated checkpointing* [27, section 1.2.3]. In this method, the processes write checkpoints independently without coordination. As a consequence, the overhead of synchronization is removed,

but it is more complicated to find a consistent state at which the computation can be restarted due to the absence of a common synchronization point. A solution to this is message logging that can be used to track the sent messages and replay them if necessary. Hence, it is possible to restart single processes and replay the lost messages without restarting the whole application.

We have now discussed *coordinated checkpointing*, which introduces undesired synchronization, and *uncoordinated checkpointing*, which adds additional overhead due to message logging. In *hierarchical checkpointing* [27, section 1.2.4] both methods are combined by grouping processes to get the best of both worlds. We can then perform coordinated checkpointing inside the group and uncoordinated checkpointing between groups. The reasoning is that coordinated checkpointing can be fast for certain process groups, such as groups on the same node. This avoids excessive message logging without adding significant overhead.

Instead of switching the checkpointing algorithm, it is also possible to switch the target memory where the checkpoint is saved. This approach is called *multi-level checkpointing* [80, 9]. Here, we try to avoid as much as possible writing to the parallel file system, but instead use the faster maybe non-persistent memory alternatives. The idea behind this is the assumption that the most frequent errors only affect small parts of the system and it might be still possible to recover data from less reliable memory sources. As a consequence, different checkpointing intervals are defined that are shorter for faster but less reliable memory destinations, such as the local memory of a node, and larger for more resilient but slow memory destinations, such as the parallel file system.

Unfortunately, all of these improvements can mainly mitigate the overhead of *checkpoint-restart* but can not completely remove the necessity to access the slow parallel file system. As a consequence, the method might not be applicable in the future with an ever increasing number of processing units. Thus, a variety of different approaches emerged that try to further improve the robustness of the codes. In the following, we will give an overview of alternative methods for adding fault tolerance.

2.1.2 Alternative methods

Many of the fault-tolerant alternatives to *checkpoint-restart* aim to add fault tolerance in a more or less transparent way for the application programmer. As a consequence, the application code can stay mostly unchanged. We will shortly describe the most important concepts in this area that involve *harware-level fault tolerance*, *replication*, and *proactive fault tolerance*. In the next section, we then discuss *algorithm-based fault tolerance* that targets specific algorithms of applications and provides resilient variants that can tolerate certain failures with typically low overhead.

Hardware-level fault tolerance A common approach to add robustness to a system is to tackle faults directly at the hardware level [41]. Here, *Error Correcting Codes* (ECC), Cyclic Redundancy Checks (CRC) and Raid systems are used to be able to detect and correct faults. These transparent correction techniques correct the system state without the application ever noticing that a fault occurred. Unfortunately, it is in general not

possible to detect and correct all error sources, such as certain hardware faults, with these methods. Hence, further methods are required.

Replication In *replication* [27, section 1.4.2], we just replicate components to be able to tolerate failures in single components. It is possible to either replicate the whole computation on different processors or to replicate each process individually during a parallel execution. As long as not all replicated components fail, we can still obtain the correct result with such an approach. However, it is obvious that such an approach also reduces the efficiency drastically due to the redundant computation.

Proactive fault tolerance *Proactive fault tolerance* [27, section 1.4.1] tries to detect a failure event before it happens. This can be done by monitoring the compute cluster and by looking for irregular behavior of some of the processors. With such an approach it is often possible to detect a fault in advance and to transfer the computation to another healthy node. This avoids restoring the process state after a failure. However, it is not guaranteed that such a system can detect all kinds of error sources. An example for proactive fault-tolerance with MPI can be found in [20].

2.1.3 Algorithm-based fault tolerance

We have seen so far that different approaches require different levels of insight into the application. In general, we can obtain a more efficient method the more it is tailored to the application. We can show this in the context of *checkpoint-restart*. It is possible to tackle this problem at the hardware level. In this case, the whole data from the complete hardware state needs to be stored in a consistent memory in order to be able to restore the current state. If we apply checkpointing on the system level, we might need to store the corresponding processes with all the relevant data needed by the operating system. On the application level, we can instead only checkpoint the relevant data that is necessary to restart the computation. We can see that by moving to the application level, we can reduce the data that needs to be stored. Unfortunately, the implementation is more intrusive the further we go towards the application level. In the worst case, every application needs to be adjusted if we apply checkpointing on the application level, while an implementation on the system level is applicable to all applications without any modification. Hence, we can say that it is usually more efficient to create a specialized fault-tolerant method, but it also increases the implementation effort.

This conclusion inspired the creation of *algorithm-based fault tolerance* (ABFT) [27, section 1.5]. Here, resilient algorithms are created, such as fault tolerant numerical methods. This has the advantage that the algorithms can be used in many different applications, while it is still possible to apply fine-grained optimizations to the algorithm at hand which keeps the overhead low. Hence, such approaches can be very efficient and are useful as a building block for several applications. Algorithm-based fault tolerance covers a wide variety of approaches that can be usually classified in one or several of

the following categories: error detection, error oblivious, and error correction (or error aware) [41].

Error detection algorithms can be based on checks of algorithm-specific invariants, such as positivity, symmetry, or mass-conservation. Another approach is to add checksums to the data. These methods mainly target faults that are not directly detected by the hardware or the operating system, such as silent data corruption. One example for a checksum-based approach is that row and column checksums are still valid after matrix additions. It is even possible to construct them so that they can be used after a matrix multiplication [61]. Another checksum approach for a sparse preconditioned conjugate gradient can be found in [109].

Error-oblivious algorithms converge to the correct solution even if an error occurs and parts of the computations are lost or corrupted. The main idea is that the algorithms do not need to notice that an error occurred as they can inherently tolerate data loss. Hence, these methods do not need specific recovery methods. Common examples are iterative solvers, such as CG, that can usually still converge if the intermediate results are changed or lost[2]. For a guaranteed convergence it might be required that faults happen at a sufficiently low frequency so that the algorithm has the opportunity to converge to the correct result at some point. Usually these methods need more compute steps to converge if faults occur, which might introduce substantial overhead. Consequently, these methods are often combined with an error correction step to reduce the overhead or guarantee convergence. For more details on soft error vulnerability of iterative linear algebra solvers we refer to [14].

Error correction algorithms are able to recover from a fault but they need a specific error recovery routine. This routine is usually called after an error was successfully detected. In this class of algorithms, the lost data is reconstructed in some way. An overview of such algorithms for the conjugate gradient method is presented in [88]. They compare different approaches to reconstruct lost data, which reduces the overhead due to additional iterative steps. Another common approach is to utilize a hierarchical representations of the data such as in multi-grid solvers or in the hierarchical basis representation of Sparse Grids. If a component in the hierarchy is lost, we can often use some redundant information from different levels to reconstruct the lost data to some degree or to find a solution with similar accuracy. An example would be the *Fault-Tolerant Combination Technique* [49, 50, 48] that is explained in Section 2.3.4. In this method, it is possible to solve an optimization problem to reduce the error that is introduced to the numerical scheme due to the lost computation. With such a lossy recovery approach it is still possible to get a good result without the need to recompute in most cases. The algorithmic details of the method are discussed in Section 2.3.4.

For a general overview on current fault-tolerant algorithm design we refer to [41] and to [99] for more information on the hardware aspects.

[2]Of course lost data needs to be replaced by new starting values.

2.2 Sparse Grids

In this section we summarize the core concepts of Sparse Grids. The theory is mainly based on [17] if not stated otherwise. We also recommend [36] and [96] for a more dense introduction to Sparse Grids.

The term Sparse Grid was first coined by Zenger in [121]. Similar ideas were, however, already used in the work of Smolyak in the context of numerical quadrature [112]. The main idea of Sparse Grids is to overcome or at least mitigate the curse of dimensionality by applying an a-priori optimization of the grid structure. This is motivated by experiences from numerical quadrature where it can be shown that quadrature rules typically introduce two error sources: a *grid-dependent* error and a *problem-dependend* error [17, pages 20-21]. The general Sparse Grid construction aims to reduce the *grid-dependent* error.

An overview of the basic ideas is given in Sections 2.2.1 to 2.2.3. Here we first illustrate the curse of dimensionality of standard tensor product grids that have $O(\tilde{N}^d)$ points. By switching to a hierarchical basis, it is possible to group grid points into subspaces. This subspace property is then utilized to formulate an optimization problem that results in the final Sparse Grid construction. Depending on the norm used for the optimization this reduces the number of points to $\mathcal{O}(\tilde{N}log(\tilde{N})^{d-1})$ (L_2 or L_∞ norm) or even $\mathcal{O}(\tilde{N})$ (L_E norm) [17, page 27 and 31]. Thereafter, we show common variations of Sparse Grids in Section 2.2.4 that typically aim at increasing approximation orders. It is however also possible to reduce the *problem-dependent* error with Sparse Grid algorithms. In these cases adaptive refinement is applied which will be discussed in Section 2.2.5.

2.2.1 Nodal Basis

Basis functions are a crucial element when it comes to discretizing a function on a grid. We assume in the following that the grid is constructed by $2^\ell \pm 1$ points depending on if we include boundary points or neglect them by assuming a zero boundary condition. Here, ℓ represents the respective level of the grid. For simplicity we assume the domain $D = [0, 1]$ and the equispaced points, i.e. $x_{\ell,i} = i/2^\ell$ for $i \in \mathcal{I}_\ell$ and $\mathcal{I}_\ell = [1, 2^\ell - 1]$ without boundary points and $\mathcal{I}_\ell = [0, 2^\ell]$ for grids with boundary points. Then, a basis $\Phi_{\ell,i}(x)$ is assigned to each point $x_{\ell,i}$. In the following we will demonstrate the nodal basis using the linear hat function

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

Further basis functions are discussed in Section 2.2.4. By scaling and shifting the basic hat function, we obtain the point centered versions

$$\Phi_{\ell,i}(x) = \Phi(2^\ell x - i). \tag{2.2}$$

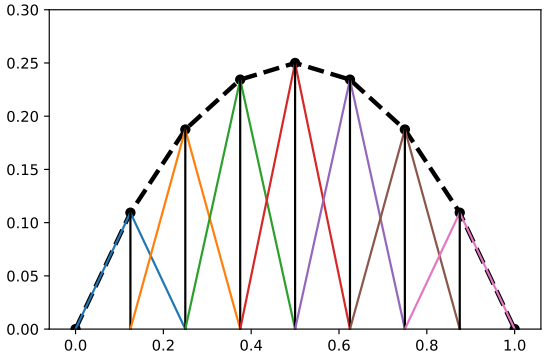


Figure 2.1: Linear interpolation of the function $u(x) = x(1-x)$ (dotted line) and the contribution of the individual hat functions with the nodal basis. The heights $c_{\ell,i}$ are depicted by solid black lines.

In Fig. 2.2 (left) one can see the basis functions for different levels. For discretizing the function u on the grid, we construct

$$u_\ell(x) = \sum_{i \in \mathcal{I}_\ell} \Phi_{\ell,i}(x) \cdot c_{l,i} \tag{2.3}$$

with $c_{\ell,i} = u(x_{\ell,i})$. This is nothing else than the linear interpolation of u via the data points $(x_{\ell,i}, u(x_{\ell,i}))$. Figure 2.1 shows this process for interpolating the function $u(x) = x(1-x)$.

In higher dimensions the d-dimensional grid is defined via a tensor product of one-dimensional grids. Here, a level vector $\boldsymbol{\ell} = (\ell_1, ..., \ell_d)$ is used to specify that the grid has $2^{\ell_k} \pm 1$ points in dimension k and $\boldsymbol{x}_{\boldsymbol{\ell},\boldsymbol{i}} = (x_{\ell_1,i_1}, \ldots, x_{\ell_d,i_d})$, $i_k \in \mathcal{I}_{\ell_k}$, $k \in [d]$. The hat functions can be generalized in a similar way using the tensor product

$$\Phi_{\boldsymbol{\ell},\boldsymbol{i}}(\boldsymbol{x}) = \prod_{k=1}^{d} \Phi_{\ell_k,i_k}(x_k). \tag{2.4}$$

Again discretizing a function u on this grid is just a d-linear interpolation using the data points $(\boldsymbol{x}_{\boldsymbol{\ell},\boldsymbol{i}}, u(\boldsymbol{x}_{\boldsymbol{\ell},\boldsymbol{i}}))$ and the basis functions $\Phi_{\boldsymbol{\ell},\boldsymbol{i}}(\boldsymbol{x})$. The resulting function $u_{\boldsymbol{\ell}}$ can be written as

$$u_{\boldsymbol{\ell}}(\boldsymbol{x}) = \sum_{\boldsymbol{i} \in \mathcal{I}_{\boldsymbol{\ell}}} \Phi_{\boldsymbol{\ell},\boldsymbol{i}}(\boldsymbol{x}) c_{\boldsymbol{\ell},\boldsymbol{i}} \tag{2.5}$$

with $\mathcal{I}_{\boldsymbol{\ell}} = \prod_{k=1}^{d} \mathcal{I}_{\ell_k}$ and $c_{\boldsymbol{\ell},\boldsymbol{i}} = u(x_{\boldsymbol{\ell},\boldsymbol{i}})$.

In this representation we can observe the curse of dimensionality by looking at the number of indices in the index set $\mathcal{I}_{\boldsymbol{\ell}}$, i.e. $N = |\mathcal{I}_{\boldsymbol{\ell}}|$. For a d-dimensional isotropic grid with $\tilde{N}$ points per dimension the grid consists of $N = \tilde{N}^d$ points. At the same time linear basis functions are known to provide an error ϵ of $\mathcal{O}(\tilde{N}^{-2})$ [17, page 18][3].

[3] This corresponds to the L_2 and L_∞ error. For the L_E error derivation, we refer to [17].

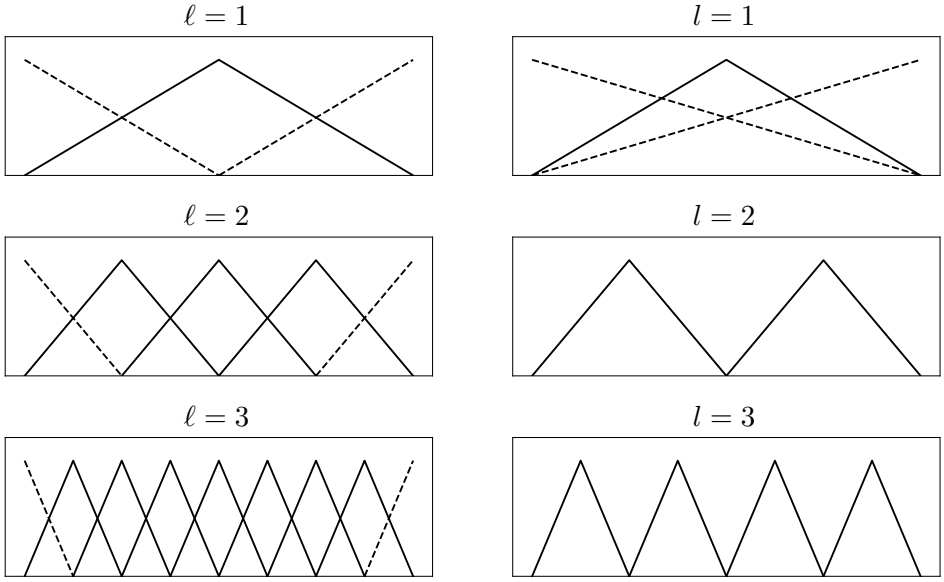


Figure 2.2: Comparison of nodal basis (left) and hierarchical basis (right) for different levels. Basis functions that are assigned to boundary points are depicted with dotted lines.

If we now write the error in terms of N, we result at the relations $\epsilon \in \mathcal{O}(N^{-2/d})$ and $N \in \mathcal{O}(\epsilon^{-d/2})$. This illustrates that we need more and more points in higher dimensions for obtaining the same error. As a result, it is infeasible to use standard tensor grids for high dimensionality.

Another important aspect of the nodal basis is the function space

$$V_{\boldsymbol{\ell}} = \text{span}\{\Phi_{\boldsymbol{\ell},\boldsymbol{i}} | \boldsymbol{i} \in \mathcal{I}_{\boldsymbol{\ell}}\} \tag{2.6}$$

that is formed by the basis functions $\Phi_{\boldsymbol{\ell},\boldsymbol{i}}$. In the next section, we will show that we can create a hierarchical representation of this space.

2.2.2 Hierarchical Basis

In the nodal basis, we constructed a set of basis functions specific to the chosen level. The idea of the hierarchical basis is to consider the full hierarchy of levels and reuse basis functions and coefficients from previous levels. Consequently, the hierarchical increments are defined which represent the change to the previous level. A hierarchical basis of level ℓ always includes all increments of level $l \leq \ell$. Figure 2.2 (right) shows the hierarchical basis functions for the different levels and compares them to the nodal basis (left).

In the hierarchical representation the function u is discretized via

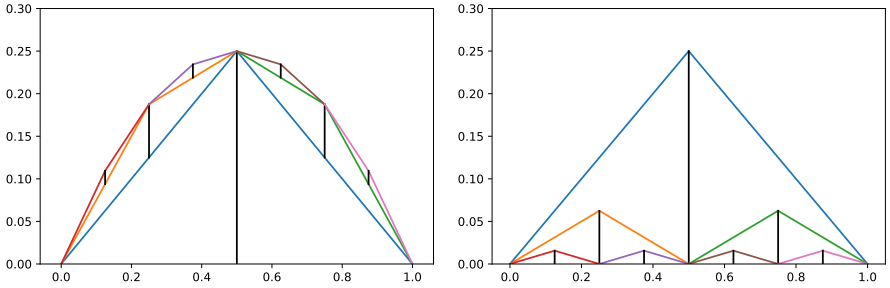


Figure 2.3: Left: Linear interpolation of the function $u(x) = x(1-x)$ (dotted line) and the contribution of the individual hat functions with the hierarchical basis. The surplus values $\alpha_{l,i}$ are depicted by solid black lines. Right: Hat functions of the hierarchical basis scaled by the respective surplus $\alpha_{l,i}$ (black lines).

$$u_\ell(x) = \sum_{l=1}^{\ell} \sum_{i \in \mathcal{I}_l^h} \Phi_{l,i}(x) \cdot \alpha_{l,i}. \tag{2.7}$$

with $\mathcal{I}_l^h = \{i \in \mathcal{I}_\ell | i \text{ odd } \vee \ l = 1\}$. The surplus values $\alpha_{l,i}$ have to be computed by solving the interpolation problem defined by the basis functions $\Phi_{l,i}(x)$ and the interpolation points $(x_{l,i}, u(x_{l,i}))$. In general, this is done by solving the system $A\boldsymbol{\alpha} = \boldsymbol{b}$ with $A \in \mathbb{R}^{N \times N}, A_{mn} = \Phi_{f(n)}(x_{f(m)})$, $\boldsymbol{b} \in \mathbb{R}^N, b_m = u(x_{f(m)})$ and $N = 2^l \pm 1$. Here, f defines a suited bijective mapping between the scalar indices (m or n) and the index pairs (l, i) from Eq. (2.7). Solving this equation requires in general cubic complexity $\mathcal{O}(N^3)$. Fortunately, for the hat function the surplus can be computed in linear complexity by subtraction of parent values (p_l and p_r) via

$$\alpha_{l,i} = u(x_{l,i}) - \frac{1}{2}(p_l + p_r) = u(x_{l,i}) - \frac{1}{2}(u(x_{l,i-1}) + u(x_{l,i+1})). \tag{2.8}$$

It should be noted that $x_{l,i-1}$ and $x_{l,i+1}$ are neighbors of $x_{l,i}$ in the full grid of level l but usually not in the full grid of level ℓ. An example for the interpolation of function $u(x) = x(1-x)$ can be found in Fig. 2.3. This special interpolation problem, where we calculate the surplus values $\alpha_{l,i}$ based on the function values $u(x_{l,i})$, is also referred to as hierarchization.

These concepts are generalized to higher dimensions via the tensor product. Consequently, the d-linear interpolation of a d-dimensional function u is defined by

$$u_\ell(\boldsymbol{x}) = \sum_{\boldsymbol{l} \in \prod_{k=1}^d [l_k]} \sum_{\boldsymbol{i} \in \mathcal{I}_{\boldsymbol{l}}^h} \Phi_{\boldsymbol{l},\boldsymbol{i}}(\boldsymbol{x}) \cdot \alpha_{\boldsymbol{l},\boldsymbol{i}}. \tag{2.9}$$

with $\mathcal{I}_{\boldsymbol{l}}^h = \prod_{k=1}^d \mathcal{I}_{l_k}^h$.

Hierarchization requires, in the general case, again to solve a linear system of equations of the form $A\boldsymbol{\alpha} = \boldsymbol{b}$ with $A \in \mathbb{R}^{N \times N}, A_{mn} = \Phi_{f(n)}(\boldsymbol{x}_{f(m)})$, $\boldsymbol{b} \in \mathbb{R}^N, \boldsymbol{b}_m = u(\boldsymbol{x}_{f(m)})$ and $N = \prod_{k=1}^d 2^{l_k} \pm 1$ with a suited bijective mapping f of the scalar indices (m and n)

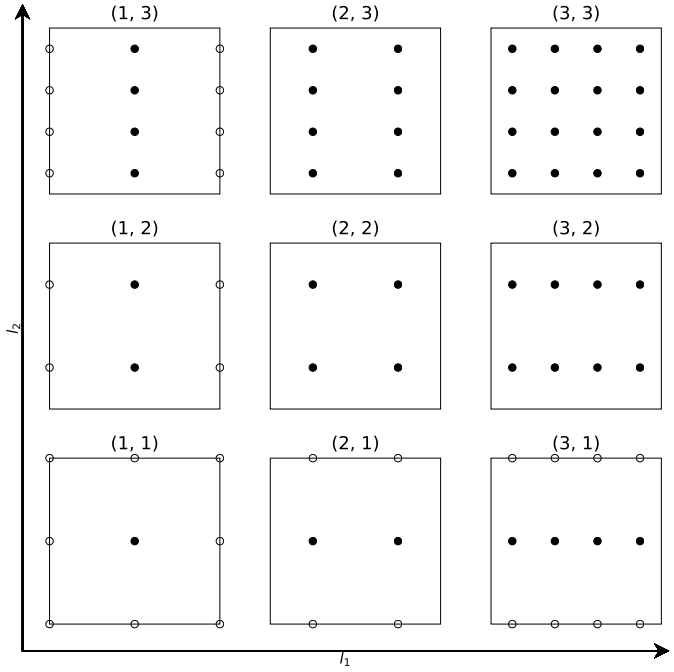


Figure 2.4: Two-dimensional examples of the increment spaces $W_{\boldsymbol{l}}$ that contribute to the nodal space $V_{(3,3)}$. Boundary points are depicted without filling.

to $(\boldsymbol{j}, \boldsymbol{i})$ from Eq. (2.9). For the linear hat basis there is again a possibility to reduce the cubic complexity ($\mathcal{O}(N^3)$) of solving the linear system. A naive approach is to generalize the stencil from Eq. (2.8) to

$$\alpha_{\boldsymbol{l},\boldsymbol{i}} = \sum_{\boldsymbol{i}-\mathbf{1}\leq\tilde{\boldsymbol{i}}\leq\boldsymbol{i}+\mathbf{1}} \left(-\frac{1}{2}\right)^{\|\boldsymbol{i}-\tilde{\boldsymbol{i}}\|_1} u(x_{\boldsymbol{l},\tilde{\boldsymbol{i}}}). \tag{2.10}$$

This involves computation in the order of $\mathcal{O}(3^d N)$. It is, however, possible to further reduce these computations with the *unidirectional principle* [8, 15]. Here, we apply d consecutive one-dimensional hierarchizations to calculate the hierarchization which reduces the cost to $\mathcal{O}(dN)$. A two-dimensional example would be

$$\begin{aligned} \tilde{\alpha}_{\boldsymbol{l},(i_1,i_2)} &= u(x_{\boldsymbol{l},(i_1,i_2)}) - \frac{1}{2}(u(x_{\boldsymbol{l},(i_1-1,i_2)}) + u(x_{\boldsymbol{l},(i_1+1,i_2))}). \\ \alpha_{\boldsymbol{l},(i_1,i_2)} &= \tilde{\alpha}_{\boldsymbol{l},(i_1,i_2)} - \frac{1}{2}(\tilde{\alpha}_{\boldsymbol{l},(i_1,i_2-1)} + \tilde{\alpha}_{\boldsymbol{l},(i_1,i_2+1)}). \end{aligned} \tag{2.11}$$

The hierarchical increments form – similar to the nodal basis – the hierarchical increment space

$$W_{\boldsymbol{l}} = \text{span}\{\Phi_{\boldsymbol{l},\boldsymbol{i}} | \boldsymbol{i} \in \mathcal{I}_{\boldsymbol{l}}^h\}. \tag{2.12}$$

By summing up increment spaces the nodal space is recovered by

$$V_{\boldsymbol{\ell}} = \oplus_{\boldsymbol{l}\in\mathbb{N}^d, \boldsymbol{l}\leq\boldsymbol{\ell}} W_{\boldsymbol{l}}. \tag{2.13}$$

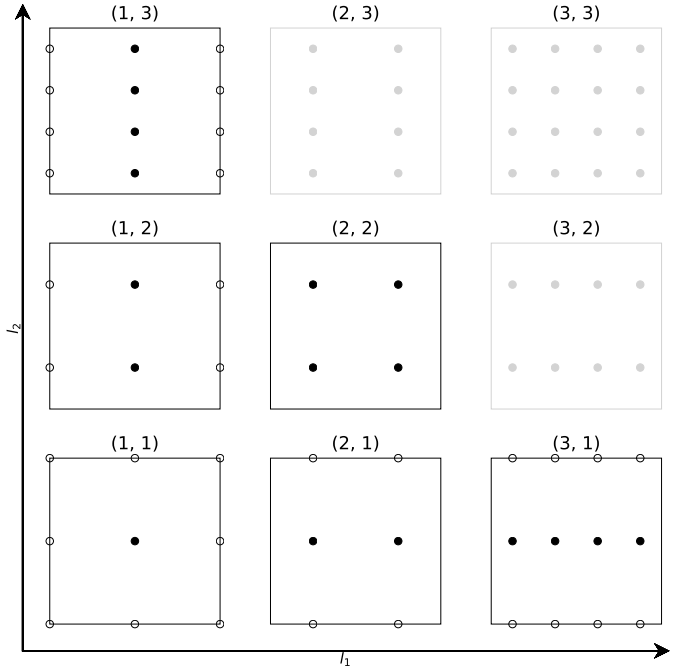


Figure 2.5: Subspace selection for the L_2 and L_∞ optimal Sparse Grid V_3^s. The excluded subspaces from the corresponding full grid are marked in light grey. Boundary points are depicted without filling.

Hence, the hierarchical representation is equivalent to the nodal representation. This is also visualized in Fig. 2.4. In the next section, we will use the hierarchical representation to show the Sparse Grid formalization.

2.2.3 Sparse Grid Construction

In the previous section, we defined the construction of a hierarchical basis with the help of increment spaces. In the Sparse Grid literature these hierarchical increment spaces are often referred to as subspaces. The Sparse Grid construction aims at restricting the grid to the subspaces that result in the best approximation quality for a specified cost budget. It can be shown via continuous or discrete optimization that this is equivalent to optimize a certain cost-benefit ratio [17, pages 21- 25]. The cost is defined via the number of grid points and the benefit is norm-dependent. For the L_2 and L_∞ error norms the optimal subspace choice for a level ℓ Sparse Grid is

$$V_\ell^s = \oplus_{\|\boldsymbol{l}\|_1 \leq \ell+d-1} W_{\boldsymbol{l}}. \tag{2.14}$$

This corresponds to only selecting subspaces of the d-dimensional simplex of level vectors with $\|\boldsymbol{l}\|_1 \leq \ell + d - 1$. In the two-dimensional case this corresponds to a triangle (see Fig. 2.5). The resulting grid can be seen in Fig. 2.6.

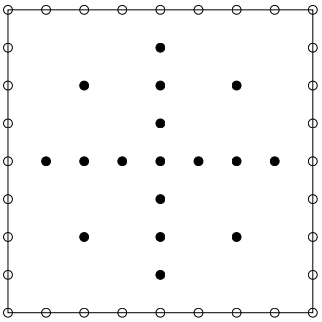


Figure 2.6: Two-dimensional Sparse Grid V_3^s. Boundary points are depicted without filling.

If we want to discretize a d-dimensional function u on a Sparse Grid, we get

$$u_\ell^s = \sum_{\|\boldsymbol{l}\|_1 \leq \ell+d-1} \sum_{\boldsymbol{i} \in \mathcal{I}_{\boldsymbol{l}}^h} \Phi_{\boldsymbol{l},\boldsymbol{i}} \cdot \alpha_{\boldsymbol{l},\boldsymbol{i}}. \tag{2.15}$$

Here, the surplus values can be derived in the same way as in Section 2.2.2 but we only consider the Sparse Grid points.

The number of points in the Sparse Grid V_ℓ^s can be calculated by

$$N = \sum_{\|\boldsymbol{l}\|_1 \leq \ell+d-1} |\mathcal{I}_{\boldsymbol{l}}^h| \in \mathcal{O}(\tilde{N} log_2(\tilde{N})^{d-1}) \tag{2.16}$$

with $\tilde{N} = 2^\ell - 1$. At the same time, the L_2 and L_∞ errors for the linear hat basis only slightly increase to $\mathcal{O}(\tilde{N}^{-2} log_2(\tilde{N})^{d-1})$ [17, page 30]. If we again relate the error to N, we get $N \in \mathcal{O}\left(\epsilon^{-\frac{1}{2}} |log_2 \epsilon|^{\frac{3}{2}(d-1)}\right)$ [17, page 41]. This shows that the curse of dimensionality can be mitigated with Sparse Grids. However, there is still a logarithmic dependency that gets problematic for very high dimensional cases. In addition, there is a dimension-dependent constant which has to be considered.

In case we are interested in the energy norm L_E, it can be shown that an alternative subspace selection with fewer points is optimal. With this selection we obtain $N \in \mathcal{O}(\tilde{N})$ points and an L_E error of $\mathcal{O}(N^{-1})$ [17, pages 30, 41]. This is an interesting observation as this implies that the curse of dimensionality is broken for such cases. However, it should be noted that there is again a dimension-dependent constant involved which does not show up in the complexity.

2.2.4 Variations

There exist many variations of Sparse Grids which optimize the approximation order or reduce the number of points. In this context mainly two questions arise: where to place the points and which basis functions to use.

The first question involves the aspects of whether or not boundary points should be added and in which way the points are distributed throughout the domain. The former

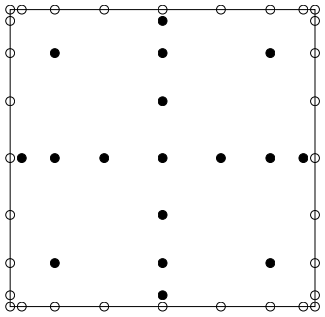


Figure 2.7: Two-dimensional Sparse Grid V_3^s with Chebyshev points. Boundary points are depicted without filling.

is highly problem dependent. Whenever we can assume zero boundary conditions, it is common to just remove boundary points. If this is not possible, one tries to reduce the number of boundary points by assigning the boundary points to a suited level. In our case we usally assign these points to level 1, but there are generalized schemes that can insert them at an arbitrary level[4]. An overview is given in [116, chapter 2.4.1] where also the impact of the insertion level on the overall point numbers is discussed. Another approach is to select the basis functions so that they extrapolate towards the boundaries. In this way boundary points can be omitted. An example is the modified linear basis function [96] that linearly extrapolates towards the domain boundary. Similar approaches can also be applied to higher order basis functions such as b-splines [116].

The second aspect is the point distribution which is especially important if we decide to go for higher order basis functions. It is well known from numerical interpolation and quadrature that equidistant points create instabilities such as the Runge phenomenon. It is therefore common to use Chebyshev (see Fig. 2.7) or Gauss points in these cases [77, 39] which are clustered towards the domain boundaries. Here, an important aspect is whether the level hierarchy provides nested points which can be problematic in case of Gauss-Legendre points. Another prominant point distribution are Leja points [75] that can be used to avoid an exponential increase in point numbers with higher levels and to reduce the interpolation error by minimizing the Lebesgue constant. In addition they are nested.

The choice of the right basis function is again problem-dependent. If the function to approximate with the Sparse Grid shows high differentiability, high order polynomial basis functions or global basis functions are suited. Examples are b-splines [116] or piecewise polynomials [15, 17, 16]. Depending on the problem it might also be appropriate to use a discontinous basis [118] or wavelets [43].

All of the approaches so far aim at minimizing the *grid-dependent* error. Although they might use some information of the problem at hand to choose an appropriate grid,

[4]Only for the spatially adaptive variants we sometimes add the boundary points to level 0.

they do not specifically tailor the grid to the problem function. This can be done with adaptive refinement which is discussed in the next section.

2.2.5 Adaptive Refinement

Sparse Grids mainly tackle the *grid-dependent* error, but it was quickly noticed that this limits the number of target applications significantly. In some scenarios certain subregions of the domain might require a high number of points to obtain reasonable accuracy. For other scenarios the required smoothness (see Section 2.2) might even be violated in certain regions of the domain. For these cases adaptive refinement can help to reduce the number of points and preserve the Sparse Grid properties.

In adaptive refinement [96] the recursive pattern of constructing a Sparse Grid is used. When we go from level ℓ to level $\ell + 1$ we add for each point exactly $2d$ children nodes which are placed in between the point and each of his neighboring points for every dimension[5]. These points are therefore also sometimes referred to as parents of the respective children points. It should be noted that this assignment is not unique. In fact each point has up to $2d$ parents. This recursive property is then used during grid refinement. Whenever we want to add points in a certain region, we just refine points without children in this region. By adding the respective (up to) $2d$ children, we obtain a spatially-refined Sparse Grid (see Fig. 2.8). This process can be repeated iteratively to refine the grid further.

But how can we determine which points or areas we should refine? In many cases automatic grid refinement based on error estimates is applied. Here, one calculates an error estimate for each point in the grid that has no children. Thereafter, points with the highest errors are refined. Common examples for error estimators are the surplus values (surplus refinement) or the volumes of the hierarchical basis functions scaled by their respective surplus (volume refinement).

During the refinement procedure it can happen that not all parents of a point are present in the grid. This might be a problem as the surplus calculation (see Section 2.2.2) is usually based on stencil calculations involving the parents. In this case, one either solves the linear system of equations directly, which might be very costly, or one recursively adds the respective parent points to the grid, which might significantly increase the number of points.

2.3 Sparse Grid Combination Technique

The *Sparse Grid Combination Technique* [44] splits a single Sparse Grid computation into several subproblems solved on anisotropic but cheap full grids. Here, a full grid with level vector $\boldsymbol{\ell}$ refers to a full tensor product grid that contains all subspaces $\tilde{\boldsymbol{\ell}} \leq \boldsymbol{\ell}$. By combining several of these full grid solutions, we obtain a solution equivalent to a Sparse Grid solution (see also Section 2.3.1) which uses exactly the same grid points.

[5] This holds only for inner points. For boundary points fewer children exist as there are no neighbors to the exterior of the domain.

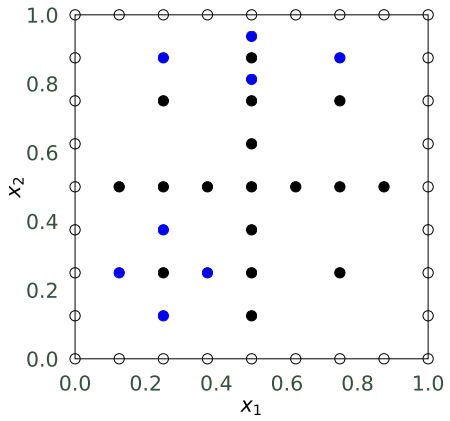


Figure 2.8: Two-dimensional spatially-refined Sparse Grid. The points that were obtained from refining point (0.25, 0.25) and point (0.5, 0.875) are highlighted in blue. Boundary points are depicted without filling.

These full grids are also referred to as component grids in the context of the Combination Technique.

This approach has several advantages compared to a classical Sparse Grid approach. First, one can use existing full grid solvers (black-box property), which reduces the implementation overhead significantly. Second, each component grid can be computed in parallel as they share no dependencies. However, there is no free lunch. Since these grids share some redundant points, the overall computation is typically higher than with pure Sparse Grids. Fortunately, in cases were point evaluations can be reused, this effect can often be neglected. In addition, this redundancy offers the possibility to create a fault tolerant scheme, that is resilient to failing component grids.

In the next sections, we first outline the basic concepts of the standard Combination Technique in Section 2.3.1 which aims to represent a standard Sparse Grid by a combination of full grids. Then, we demonstrate how to generalize this idea in Section 2.3.2 to flexible sets of component grids. This will open the possibility to enable adaptivity to the combination technique which will be discussed in Section 2.3.3. Finally, the Fault-Tolerant Combination Technique is discussed in Section 2.3.4.

2.3.1 Standard Combination Technique

In the standard Combination Technique, we aim to represent a Sparse Grid solution on many anisotropic full grids. The main issue here is to include the same subspaces in the computation. If we look back at Fig. 2.5, we see that a level ℓ Sparse Grid contains exactly the subspaces with level vector $\|\boldsymbol{\ell}\|_1 \leq \ell + (d-1)$. This can be done by including all full grids with $\|\boldsymbol{\ell}\|_1 = \ell + (d-1)$. Since each full grid with level vector $\boldsymbol{\ell}$ contains all subspaces $\boldsymbol{l} \leq \boldsymbol{\ell}$, many subspaces are contained multiple times in such a scheme. Hence, just adding the contributions would result in an incorrect result. We therefore need to subtract these redundant subspaces again. In two dimensions this results in the

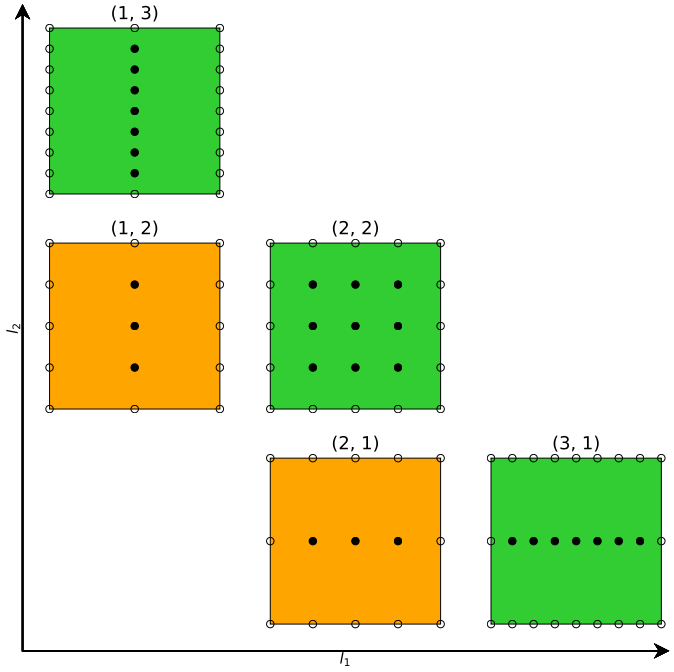


Figure 2.9: Standard Combination technique for the computation of u_3^c. Green grids are added while orange grids are subtracted. As a result each point of the Sparse Grid Fig. 2.6 is accounted for exactly once. Boundary points are depicted without filling.

scheme [44]

$$u_\ell^c = \sum_{\|\boldsymbol{\ell}\|_1=\ell+1} u_{\boldsymbol{\ell}} - \sum_{\|\boldsymbol{\ell}\|_1=\ell} u_{\boldsymbol{\ell}} \tag{2.17}$$

for computing the combination solution u_ℓ^c of a two-dimensional function u. Here, it can be seen that we need to subtract the full grids on the diagonal with $\|\boldsymbol{\ell}\|_1 = \ell$. This is visualized in Fig. 2.9.

For higher dimensions this is generalized – similarly to the inclusion-exclusion principle – by

$$u_\ell^c = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{\boldsymbol{\ell}\in\mathcal{I}_{\ell,q}} u_{\boldsymbol{\ell}} \tag{2.18}$$

with $\mathcal{I}_{\ell,q} = \{\boldsymbol{\ell} \in \mathbb{N}_0^d | \|\boldsymbol{\ell}\|_1 = \ell + d - 1 - q\}$.

The obvious question now is if this combination is identical to the Sparse Grid solution. In case the $u_{\boldsymbol{\ell}}$ are derived from an exact interpolation of u and a suited hierarchical basis is used, it can be shown that they are in fact identical. A rigorous proof can be found in [116, section 4.3.1].

However, sometimes the $u_{\boldsymbol{\ell}}$ are obtained by a discretization that involve mesh width dependent error terms. A typical example is the discretization of a PDE on a grid. In these cases, it can be shown that the errors in the Combination Technique have an

equivalent error complexity as Sparse Grids if the error splitting assumption holds [44]. We will demonstrate this in the following.

The error splitting assumption defines an Anova-like error splitting of the point-wise discretization error:

$$\begin{aligned} u - u_{\boldsymbol{\ell}} = &\sum_{k=1}^{d} C_k(h_{\ell_k}) f_1(h_{\ell_k}) + \sum_{k=1}^{d}\sum_{j=1}^{d} C_{k,j}(h_{\ell_k}, h_{\ell_j}) f_2(h_{\ell_k}, h_{\ell_j}) \\ &+ \ldots + C_{1,\ldots,d}(h_{\ell_1}, \ldots, h_{\ell_d}) f_d(h_{\ell_1}, \ldots, h_{\ell_d}) \end{aligned} \tag{2.19}$$

where the constants $C_{\boldsymbol{i}} < \kappa$ are bounded by a positive constant $\kappa \in \mathbb{R}^+$ and $h_\ell = 2^{-l}$. The functions f_k represent the error decay or convergence rate with respect to the mesh width and should go to zero for small mesh widths. In two dimensions this can be written as

$$u - u_{\boldsymbol{\ell}} = C_1(h_{\ell_1}) f_1(h_{\ell_1}) + C_2(h_{\ell_2}) f_1(h_{\ell_2}) + C_{1,2}(h_{\ell_1}, h_{\ell_2}) f_2(h_{\ell_1}, h_{\ell_2}). \tag{2.20}$$

The error of the combination can then be calculated by

$$\begin{aligned} u - u_\ell^c = &\; u - \sum_{\|\boldsymbol{\ell}\|_1 = \ell+1} u_{\boldsymbol{\ell}} - \sum_{\|\boldsymbol{\ell}\|_1 = \ell} u_{\boldsymbol{\ell}} \\ = &\sum_{\|\boldsymbol{\ell}\|_1 = \ell+1} (u - u_{\boldsymbol{\ell}}) - \sum_{\|\boldsymbol{\ell}\|_1 = \ell} (u - u_{\boldsymbol{\ell}}) \\ = &\sum_{\|\boldsymbol{\ell}\|_1 = \ell+1} (C_1(h_1) f_1(h_{\ell_1}) + C_2(h_{\ell_2}) f_1(h_{l_2}) + C_{1,2}(h_{\ell_1}, h_{\ell_2}) f_2(h_{\ell_1}, h_{\ell_2})) \\ &- \sum_{\|\boldsymbol{\ell}\|_1 = \ell} (C_1(h_{\ell_1}) f_1(h_{\ell_1}) + C_2(h_{\ell_2}) f_1(h_{\ell_2}) + C_{1,2}(h_{\ell_1}, h_{\ell_2}) f_2(h_{\ell_1}, h_{\ell_2})) \\ = &\; C_1(h_\ell) f_1(h_\ell) + C_2(h_\ell) f_1(h_\ell) + \sum_{\|\boldsymbol{\ell}\|_1 = \ell+1} (C_{1,2}(h_{\ell_1}, h_{\ell_2}) f_2(h_{\ell_1}, h_{\ell_2})) \\ &- \sum_{\|\boldsymbol{\ell}\|_1 = \ell} (C_{1,2}(h_{\ell_1}, h_{\ell_2}) f_2(h_{\ell_1}, h_{\ell_2})). \end{aligned} \tag{2.21}$$

Here, we use the fact that the first summation has exactly one term more than the second summation. In addition, one-dimensional error terms with the same mesh width cancel except for the error terms with finest mesh width h_ℓ of level ℓ.

The critical part is now which error decay f_k we assume. From Eq. (2.21) it becomes obvious that the error cancellation of the one-dimensional terms $C_1(h_{\ell_k}) f_1(h_{\ell_k})$ is the main selling point of the combination technique. An optimal application therefore concentrates most of the error in the one-dimensional terms. Hence, a fast error decay of mixed terms is favourable. Note that for Sparse Grids we assume bounded mixed derivatives which is not the same requirement. In [44] an error decay of $f_k(h_1, \ldots, h_k) = h_1^2 \cdot \ldots \cdot h_k^2$ is assumed. In cases with such an error decay, the equation simplifies to

$$
\begin{aligned}
u - u_\ell^c = & \; C_1(h_\ell)2^{-2\ell} + C_2(h_\ell)2^{-2\ell} + \sum_{\|\boldsymbol{\ell}\|_1=\ell+1} (C_{1,2}(h_{\ell_1}, h_{\ell_2})2^{-2\ell_1} \cdot 2^{-2\ell_2})) \\
& - \sum_{\|\boldsymbol{\ell}\|_1=\ell} (C_{1,2}(h_{\ell_1}, h_{\ell_2})2^{-2\ell_1} \cdot 2^{-2\ell_2}) \\
= & \; C_1(h_\ell)2^{-2\ell} + C_2(h_\ell)2^{-2\ell} + \sum_{\|\boldsymbol{\ell}\|_1=\ell+1} (C_{1,2}(h_{\ell_1}, h_{\ell_2})2^{-2(\ell+1)}) \\
& - \sum_{\|\boldsymbol{\ell}\|_1=\ell} (C_{1,2}(h_{\ell_1}, h_{\ell_2})2^{-2\ell}) \\
\leq & 2^{-2\ell}\left(\kappa + \kappa + \frac{1}{4}\ell\kappa + (\ell-1)\kappa\right) \in \mathcal{O}(2^{-2\ell}\ell) = \mathcal{O}(\tilde{N}^{-2}log_2(\tilde{N}))
\end{aligned}
\tag{2.22}
$$

with $\tilde{N} = 2^\ell$ (see also Section 2.2). We obtained the same error rate for Sparse Grids with linear hat functions (see Section 2.2.3). The solutions are therefore often considered to be equivalent yet not identical. A three-dimensional example of the error cancellation is outlined in [44].

It has to be noted that the error splitting with quadratic error decay per dimension is not generally proven for arbitrary PDEs so far. Hence, the convergence of the Combination Technique is not guaranteed for arbitrary PDEs and as we will see in Section 3.6.4 especially non-linear PDEs still pose problems. Nonetheless, there are already for some special cases convergence proofs. In [18] and [19] Bungartz et al proved the convergence of the Combination Technique for Laplace's equation for sufficiently smooth boundary conditions.

A variation of the standard Combination Technique is the truncated Combination Technique which introduces a minimum level $\boldsymbol{\ell}^{\min}$ that defines, for each dimension, a lower limit for the level vector $\boldsymbol{\ell}$. In this case, the set of level vectors in Eq. (2.18) is changed to $\mathcal{I}_{\ell,q}^{\boldsymbol{\ell}^{\min}} = \{\boldsymbol{\ell} \in \mathbb{N}_0^d | \|\boldsymbol{\ell} - \boldsymbol{\ell}^{\min}\|_1 = \ell + d - 1 - q, \boldsymbol{\ell} \geq \boldsymbol{\ell}^{\min}\}$. Sometimes it is also required to add individual target level ℓ for each dimension. In such cases, a maximum level $\boldsymbol{\ell}^{\max}$ is defined for each dimension. The construction of the index set is a bit more complicated in this case. One possibility is to first define $\ell = \min(\{n \in \boldsymbol{\ell}^{\max} - \boldsymbol{\ell}^{\min}) | n > 0\}$ and $\tilde{\boldsymbol{\ell}}_k^{\min} = \max(\boldsymbol{\ell}_k^{\max} - \ell, \boldsymbol{\ell}_k^{\min})$. This results in the index set $\mathcal{I}_\ell^{\tilde{\boldsymbol{\ell}}^{\min}} = \bigcup_{q=0}^{d-1} \mathcal{I}_{\ell,q}^{\tilde{\boldsymbol{\ell}}^{\min}}$ with $\mathcal{I}_{\ell,q}^{\tilde{\boldsymbol{\ell}}^{\min}} = \{\boldsymbol{\ell} \in \mathbb{N}_0^d | \|\boldsymbol{\ell} - \tilde{\boldsymbol{\ell}}^{\min}\|_1 = \tilde{\ell} + d - 1 - q, \tilde{\boldsymbol{\ell}}^{\min} \leq \boldsymbol{\ell} \leq \tilde{\boldsymbol{\ell}}^{\min} + \ell\mathbf{1}, \boldsymbol{\ell} \leq \boldsymbol{\ell}^{\max}\}$. This index set is a slight modification of the one proposed in [51]. Our definition sets the dimensions with $\boldsymbol{\ell}_k^{\min} = \boldsymbol{\ell}_k^{\max}$ constant in the scheme and only creates a combination scheme with the remaining dimensions.

Setting minimal and maximal levels is often necessary for the solution of PDEs. Here, certain Physical phenomena might only be visible with a specific minimum resolution $\boldsymbol{\ell}^{\min}$ or the solution might be unstable for low resolutions. In addition, a specific maximum level $\boldsymbol{\ell}^{\max}$ might be sufficient for the required accuracy targets.

In cases where we have a constant distance ℓ between $\boldsymbol{\ell}^{\max}$ and $\boldsymbol{\ell}^{\min}$, i.e. $\ell = \min(\boldsymbol{\ell}^{\max} - \boldsymbol{\ell}^{\min}) = \max(\boldsymbol{\ell}^{\max} - \boldsymbol{\ell}^{\min})$, it is common to define the aforementioned scheme by the parameters ℓ and $\boldsymbol{\tau} = \boldsymbol{\ell}^{\min} - \mathbf{1}$. The resulting index set is then $\mathcal{I}_\ell^{\boldsymbol{\tau}} = \bigcup_{q=0}^{d-1} \mathcal{I}_{\ell,q}^{\boldsymbol{\tau}}$ with

$\mathcal{I}^{\boldsymbol{\tau}}_{\ell,q} = \{\boldsymbol{\ell} \in \mathbb{N}^d | \forall k \in [d] : \ell_k > \tau_k, \|\boldsymbol{\ell}\|_1 = \|\boldsymbol{\tau}\|_1 + \ell + d - 1 - q\}$. We will use this notation in Section 4.2.

In [71] they use a similar approach but instead of a minimum level they define a scalar coarsening parameter s that defines how far away the combination is from the full grid with level $\boldsymbol{\ell}$. Here, they do not differentiate between dimensions which makes it a bit less flexible.

The Combination Technique is, however, not restricted to the standard schemes, but can deal with a wide range of possible level sets. In the next section, we will discuss the required properties of the index set and how to derive the combination coefficients.

2.3.2 Generalizations of the Combination Technique

The general Combination Technique is based on the generalized Sparse Grid construction [40]. Here, we do not construct the combination scheme based on the a priori optimized set of subspaces that we derived in the Sparse Grid construction. The general Combination Technique can then be written as

$$u^c_{\mathcal{I}} = \sum_{\boldsymbol{\ell} \in \mathcal{I}} c_{\boldsymbol{\ell}} \cdot u_{\boldsymbol{\ell}} \tag{2.23}$$

where $\mathcal{I}$ is the index set of used level vectors and $c_{\boldsymbol{\ell}} \in \mathbb{Z}$ is the scalar combination coefficient. However, not all index sets result in a valid Sparse Grid construction. For this purpose Gerstner and Griebel formulated the admissibility criterion. It states that each entry in $\mathcal{I}$ needs to satisfy

$$\boldsymbol{\ell} \in \mathcal{I} \wedge l_k > \ell_k^{\min} \implies \boldsymbol{\ell} - \boldsymbol{e}_k \in \mathcal{I} \text{ for } 1 \leq k \leq d, \tag{2.24}$$

where $\boldsymbol{e}_k$ denotes the k-th unit vector. Such an index set is also often called downward closed. Based on this index set, the combination coefficients can be calculated by

$$c_{\boldsymbol{\ell}} = \sum_{\boldsymbol{\ell} \leq \boldsymbol{i} \leq \boldsymbol{\ell}+\mathbf{1}, \boldsymbol{i} \in I} (-1)^{\|\boldsymbol{i}-\boldsymbol{\ell}\|_1}. \tag{2.25}$$

An observation from Eq. (2.25) is that $c_{\boldsymbol{\ell}} = 0$ if all $\tilde{\boldsymbol{\ell}}$ with $\boldsymbol{\ell} \leq \tilde{l} \leq \boldsymbol{\ell} + \mathbf{1}$ are contained in the index set $\mathcal{I}$. These level vectors $\boldsymbol{\ell}$ are therefore often excluded from $\mathcal{I}$ for brevity (see for example the standard Combination Technique in Section 2.3.1).

Another technique to optimize the Combination Technique is the *optimized Combination Technique* [57] which aims at optimizing the combination coefficients $c_{\boldsymbol{\ell}}$. In this technique the restriction of $c_{\boldsymbol{\ell}} \in \mathbb{Z}$ is lifted and instead a continuous optimization problem with $c_{\boldsymbol{\ell}} \in \mathbb{R}$ is solved. This optimization problem has to include specific error norms of the problem. This technique was successfully applied to eigenproblems [33, 70] and regression [32].

The flexibility of the generalized Combination Technique offers possibilities to adapt the index set $\mathcal{I}$ to the problem such as in adaptive Sparse Grids. In the next section, we will discuss such strategies.

2.3.3 Adaptivity with the Combination Technique

The regular grid structure and the fixed index set of the standard Combination Technique make it hard to apply adaptivity. Consequently, adaptive versions mainly concentrate on generalizations of the Combination Technique such as the generalized index set $\mathcal{I}$ from the previous section. This directly leads to dimension adaptivity where we adapt the subspace selection and thereby the index set to the problem at hand. By generalizing the component grids themselves it is even possible to achieve spatial adaptivity. In the following, common adaptive techniques from the literature are discussed.

2.3.3.1 Dimensional adaptivity

Several approaches for dimension-adaptive algorithms were investigated in the past [40, 56]. The dimension-adaptive version of the Combination Technique by Gerstner and Griebel [40] is widely applied throughout Sparse Grid literature (see for example [30, 34, 35]). The main benefit of the method is its rather easy implementation. They use the generalized index set construction from Section 2.3.2 and define a method how to adaptively add subspaces to this index set $\mathcal{I}$. The combination coefficients can then be calculated for example via Eq. (2.25).

Algorithm 1 Dimension adaptive algorithm for the integration of function f with the Combination Technique [40].

procedure DIM_ADAPTIVE_COMBI($\boldsymbol{a}$, $\boldsymbol{b}$, $tolerance$, f) ▷ Output: $integral$
  $\boldsymbol{\ell} = (1, ..., 1)$
  $O = \varnothing$
  $A = \{\boldsymbol{\ell}\}$
  $integral = Q_{\boldsymbol{\ell}}$
  $\epsilon = \epsilon_{\boldsymbol{\ell}}$
  while $\epsilon > tolerance$ **do**
    refine $\boldsymbol{\ell} = \text{argmax}_{\boldsymbol{\ell} \in A}\, \epsilon_{\boldsymbol{\ell}}$
    $A = A \backslash \{\boldsymbol{\ell}\}$
    $O = O \cup \{\boldsymbol{\ell}\}$
    $\epsilon = \epsilon - \epsilon_{\boldsymbol{\ell}}$
    for $k \in [d]$ **do**
      $\tilde{\boldsymbol{\ell}} = \boldsymbol{\ell} + \boldsymbol{e_k}$
      if $\tilde{\boldsymbol{\ell}}$ - $\boldsymbol{e_q} \in O \vee \tilde{\boldsymbol{\ell}}_q = 1$ for all $q = 1...d$ **then**
        $A = A \cup \{\tilde{\boldsymbol{\ell}}\}$
        $\epsilon_{\boldsymbol{\ell}} = \sum_{\boldsymbol{\ell - 1} \leq \boldsymbol{j} \leq \boldsymbol{\ell}} (-1)^{\|\boldsymbol{\ell} - \boldsymbol{j}\|_1} Q_{\boldsymbol{j}}$
        $integral = integral + \epsilon_{\boldsymbol{\ell}}$
        $\epsilon = \epsilon + \epsilon_{\boldsymbol{\ell}}$
      end if
    end for
  end while
end procedure

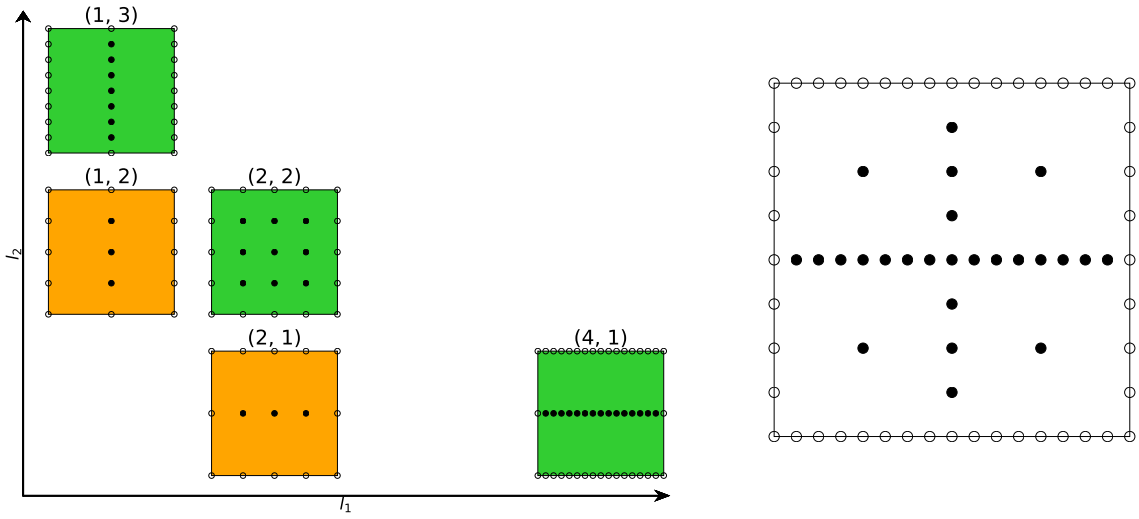


Figure 2.10: Left: Example of a dimension adaptive combination scheme. In this case we started with a combination of level $\ell = 3$ and refined the grid $(3, 1)$. Right: Corresponding Sparse Grid for the combination.

In their method, they differentiate between an old and an active index set. In each step one of the active indices $\boldsymbol{\ell}$ in the the active index set is chosen for refinement. In this refinement procedure, forward neighbors $\tilde{\boldsymbol{\ell}} = \boldsymbol{\ell} + \boldsymbol{e}_k$ might be added to the active index set and $\boldsymbol{\ell}$ is moved to the old index set. Forward neighbors $\tilde{\boldsymbol{\ell}}$ are only added if for all $q \in [d]$ the backward neighbor $\tilde{\boldsymbol{\ell}} - \boldsymbol{e}_q$ is in the old index set or $\tilde{\ell}_q = 1$. This rule guarantees that the index set fulfills the admissibility criterion (see Eq. (2.24)).

The selection process of the subspace that is refined is guided by an error estimator. Here, for each subspace a local error estimate is calculated. Typically the subspace with largest estimated error is then selected for refinement. By summing up all local errors in the active index set of the scheme, they obtain the global error estimate which is used for deciding when to stop the adaptation process. Instead of the pure error estimate, it is also possible to guide the subspace selection by a combination of an error estimate and a cost measure. This is typically more efficient as otherwise the most anisotropic grids are usually preferred by the adaptation.

An example of the algorithm can be seen in Algorithm 1 where the integral of a function f in the domain $D = \prod_{k=1}^{d} [a_k, b_k]$ is calculated. Here, $Q_{\boldsymbol{\ell}}$ denotes the integral approximation of the component grid $\boldsymbol{\ell}$, ϵ the global error estimate, and $\epsilon_{\boldsymbol{\ell}}$ the local error estimate for component grid $\boldsymbol{\ell}$. In Fig. 2.10 a dimension-adaptive combination scheme and the corresponding Sparse Grid are shown.

There exist also other variants of this algorithm for adding new subspaces. For example in [29] the authors do not add all applicable forward-neighbors of the active grid with maximal error estimate but instead look at all grids that could be possible added to the index set and calculate for each of these grids an error based on the existing backward-neighbors.

The cost and error estimates are typically problem-dependent and have to be adjusted to the application to achieve good results. For this reason many modifications have been developed. Examples are sensitivity-based approaches for uncertainty quantification [30] or applications in machine learning that calculate the benefit directly based on the error reduction of the underlying regression problem [34, 35].

2.3.3.2 Spatial adaptivity

Spatial adaptivity with the Combination Technique is rather complicated due to the regular tensor-product nature of the component grids. It is therefore not easily possible to add points at specific regions. A common remedy to this problem is to define a graded grid [46] that concentrates grid points at specific regions. These rectilinear grids define the point hierarchies not in an equidistant fashion but increase the point densities in selected regions. Unfortunately, this grading has to be typically defined a priori and the tensor-product construction limits the applicability of the method. An example of a rectilinear grid can be seen in Fig. 4.1 in Section 4.1.

Due to the current limitations with spatial adaptivity, we have developed two new approaches that are summarized in Chapter 4. They preserve the black-box property of the Combination Technique and are based on rectilinear or block-adaptive grids, respectively. First, in Section 4.1 we show a novel approach that builds on the idea of the graded grids and can adapt itself to the problem at hand by automatically finding efficient one-dimensional grid structures that fit to the problem. Second, we outline in Section 4.2 a refinement algorithm that utilizes the idea of block-adaptivity. In block-adaptive refinement algorithms the domain is first split into different subregions and then regular subgrids of varying resolutions are generated for each of these regions. These subgrids are then combined to form the global grid. This is a technique that is common in PDE simulations. We used this concept and defined a novel block-adaptive Combination Technique in Section 4.2.

Another approach to spatial adaptivity was described in [83] which is similar to a mix of a recursive block-adaptivity and the dimension-adaptive algorithm. In this paper, the authors define a hierarchy of cells that are generated by the Combination Technique. By refining the cells, these grid structures are adaptively refined. The refinement algorithm for the cells is similar to the dimension adaptive algorithm by Gerstner and Griebel [40] but is applied on a cell level and not on a component grid level. As a consequence, the method has no component grid representation as the cell hierarchies differ across the domain. This generalization offers the flexibility to adapt the Combination Technique dimensionally and spatially to the problem. However, the black-box property of the Combination Technique is lost due to the nonexistence of component grids. Instead the cell hierarchies have to be integrated into the application which might require specialized interpolation. Another disadvantage is that the method is only defined for two dimensions. It is therefore necessary to generalize this idea first and to check if the bookkeeping overhead of the individual cell hierarchies pays off. For these reasons this approach provides only limited benefit over a direct Sparse Grid implementation.

2.3.4 Fault-Tolerant Combination Technique

In current HPC systems failure during computations might occur which leads to missing or incorrect results. A fault tolerant numerical scheme should be able to deal with the loss or corruption of certain data. Due to the inherent redundancy across the component grids, the Combination Technique has a high potential to tolerate such faults. The *Fault-Tolerant Combination Technique* (FTCT) was therefore proposed for the ExaHD project and was first implemented and formalized by Harding [49, 50, 48]. These ideas were later applied to a wide range of applications and implemented in an HPC framework [6, 7, 5]. In the ExaHD project a competing HPC implementation was developed which is described in Section 3.2.

The main idea of the FTCT is to modify the combination scheme so that all failed component grids $u_{\boldsymbol{\ell}} \in \mathcal{F}$ are excluded where $\mathcal{F}$ defines the set of failed component grids. This boils down to removing the affected level vectors from the index set $\tilde{\mathcal{I}} = \mathcal{I} \setminus \mathcal{F}$. This, however, might violate the admissibility criterion (see Eq. (2.24)), i.e. the set $\tilde{\mathcal{I}}$ might not be downward closed. Creating valid combination coefficients with Eq. (2.25) is therefore not possible anymore.

Instead, the *general coefficient problem* (GCP) is solved. In the GCP [48] we try to minimize $|u - u^c_{\tilde{\mathcal{I}}}|$ with $u^c_{\tilde{\mathcal{I}}} = \sum_{\boldsymbol{\ell} \in \tilde{\mathcal{I}}} \tilde{c}_{\boldsymbol{\ell}} u_{\boldsymbol{\ell}}$ so that $\sum_{\tilde{\boldsymbol{\ell}} \geq \boldsymbol{\ell}, \tilde{\boldsymbol{\ell}} \in \tilde{\mathcal{I}}} \tilde{c}_{\tilde{\boldsymbol{\ell}}} \in \{0, 1\}, \tilde{c}_{\tilde{\boldsymbol{\ell}}} \in \mathbb{Z}$. This is done by minimizing the upper bound for the combination error. This results in an optimization problem that was shown to be a variant of the 0-1 binary integer programming problem [48]. Hence, the problem is in general NP complete [67].

To reduce the computation cost, Harding simplified the optimization problem by looking at the number of layers or hyperplanes the Combination Technique computes. In the normal case, we compute all component grids with level vector $\|\boldsymbol{\ell}\|_1 = \ell + d - 1 - q$ for $q \in [0, d-1]$. We have hence d layers of component grids that we compute. By extending this range to two additional lower layers we end up with $d+2$ layers ($q \in [0, d+1]$). In addition, the algorithm allows to recompute any component grid with $\|\boldsymbol{\ell}\|_1 \leq \ell + d - 3$ which have at most a quarter of the points compared to points in the highest layer. As a result, only missing component grids of the second layer with $\|\boldsymbol{\ell}\|_1 = \ell + d - 2$ can violate the admissibility criterion. This limits the number of possible solutions for the coefficients in the GCP which makes it feasible to compute the solution. Hence, this approach offers a good trade-off between solving the GCP and the cost of recomputing solutions. In our framework (see Section 3.2), we also followed this approach.

2.3.5 Time-Dependent PDE simulations with the Combination Technique

We have seen in Section 2.3.1 that the solution of the Sparse Grid Combination Technique for PDEs represents additional challenges due to the discretization error introduced in each component grid. This is, however, not the only difference when we want to apply the Combination Technique to PDE simulations. In many cases, the PDEs of interest are time-dependent, i.e. the grid solution changes over time. This problem can be tackled in two ways: treating the time as another dimension in the Sparse Grid

representation or the recombination approach where we combine the grid results at defined times. In our project, we opted for the recombination approach but examples for treating the time and even model parameters as additional dimensions can be found in [104].

If we combine the grid values at defined times, we introduce another parameter to the simulation that needs to be tuned: the combination interval. We could either combine every time step, after a certain amount of time steps, at specific synchronization points, or even only once in the end. We will discuss this issue in the context of Plasma Physics simulations in Chapter 3. In general, the intuition is that longer combination intervals allow the component grids to drift further apart due to an accumulation of different time and spatial errors, whereas small combination intervals typically introduce a large overhead for the combination and decrease the potential for parallelization as the computation needs to be synchronized at the combination points.

Another important issue is how to combine the different grids efficiently. A thorough analysis and efficient implementation of this combination step was one of the main achievements of Mario Heenes PhD which is summarized in [51]. The first consideration was how to combine grids to get the Sparse Grid solution of all component grids. A theoretical analysis [51, section 2.7.1] revealed that the most efficient way to combine the grids is in the hierarchical basis. This means that each component grid gets hierarchized, followed by a summation of surplusses accross the component grids multiplied by the respective combination coefficients. This can be written as

$$u_\ell^c = \sum_{\boldsymbol{\ell} \in \mathcal{I}} c_{\boldsymbol{\ell}} \cdot \sum_{\|\boldsymbol{l}\|_1 \leq \ell+d-1} \sum_{\boldsymbol{i} \in \mathcal{I}_{\boldsymbol{l}}^h} \Phi_{\boldsymbol{l},\boldsymbol{i}} \cdot \alpha_{\boldsymbol{\ell},\boldsymbol{l},\boldsymbol{i}} \tag{2.26}$$

where $\alpha_{\boldsymbol{\ell},\boldsymbol{l},\boldsymbol{i}}$ is the surplus value for the component grid $\boldsymbol{\ell}$ at the hierarchical basis of subspace $\boldsymbol{l}$ and index $\boldsymbol{i}$. Since a component grid only contains subspaces $\boldsymbol{l} \leq \boldsymbol{\ell}$, we set $\alpha_{\boldsymbol{\ell},\boldsymbol{l},\boldsymbol{i}} = 0$ for $\boldsymbol{l} > \boldsymbol{\ell}$. Accordingly, the Sparse Grid representation

$$u_\ell^c = \sum_{\|\boldsymbol{l}\|_1 \leq \ell+d-1} \sum_{\boldsymbol{i} \in \mathcal{I}_{\boldsymbol{l}}^h} \Phi_{\boldsymbol{l},\boldsymbol{i}} \cdot \alpha_{\boldsymbol{l},\boldsymbol{i}} \tag{2.27}$$

is obtained where $\alpha_{\boldsymbol{l},\boldsymbol{i}} = \sum_{\boldsymbol{\ell} \in \mathcal{I}, \boldsymbol{\ell} \geq \boldsymbol{l}} c_{\boldsymbol{\ell}} \cdot \alpha_{\boldsymbol{\ell},\boldsymbol{l},\boldsymbol{i}}$. Note that $u_\ell^c \neq u_\ell^s$ as discussed in Section 2.3.1.

Once the Sparse Grid representation is calculated the information needs to be propagated back to the component grids. This propagation can be easily done by setting all surplus values of a component grid $\boldsymbol{\ell}$ to $\alpha_{\boldsymbol{\ell},\boldsymbol{l},\boldsymbol{i}} = \alpha_{\boldsymbol{l},\boldsymbol{i}}$ for $\boldsymbol{\ell} \geq \boldsymbol{l}$ [6].

As a last step, the new values at the component grids are dehierarchized by converting them back to the nodal basis and are then used as the initial condition for the calculation of the next time slice. This process is repeated until we reach the end of the simulation.

[6] Surplus values of subspaces $\boldsymbol{l}$ with $\boldsymbol{\ell} < \boldsymbol{l}$ are not contained in the component grid $\boldsymbol{\ell}$.

3 DisCoTec: A fault-tolerant HPC framework for time-dependent PDEs

In this chapter, we outline the main characteristics and features of our HPC implementation of the Combination Technique for time-dependent PDEs (see Section 2.3). The Combination Technique is well suited for such an implementation for two reasons: the embarrassing parallel computation of the component grids and the black-box property. The former allows us to distribute different component grids to different computing units without the need for synchronization. The black-box property makes it possible to design a solver-independent framework that can be applied to any high-dimensional PDE.

These considerations led to the development of an HPC framework in the project EXAHD [97, 72], a subproject of the priority program SPPEXA. The code was formally designed as a submodule of SG^{++} and is now a seperate git repositroy named DisCoTec[1]. The basic algorithms and code structure were created in the course of Mario Heenes PhD project [51] and are summatized in Section 3.1. The framework also adds functionality for fault tolerance by implementing the FTCT (Section 2.3.4) which was developped in close collaboration of Mario Heene [51] and Alfredo Parra [89]. The framework is written in C++ and is parallelized using the *Message Passing Interface* (MPI) [31].

As an addition to the already existing code base, we have developed several algorithmic and theoretical improvements for the Combination Technique. First, in Section 3.2 we describe implementation details for the FTCT within DisCoTec. This includes a brief recap of the previously implemented features and a more detailed overview of the novel features such as fault distributions and a sophisticated fault recovery with MPI communicator restructuring with the use of spare ranks [86].

Second, we give in Section 3.3 a theoretical analysis that shows that by using optimal time steps for each component grid, the curse of dimensionality can be overcome in theory. This results in a complexity reduction from $\mathcal{O}(N^2 log(N)^{d-1})$ to $\mathcal{O}(N^2)$ which makes the Combination Technique independent of the problem dimension d for the simulation of PDEs. However, the dimension still influences the constant factors which limits the feasible dimensions in practice. We also implemented this time-stepping technique for our application code GENE.

Thereafter, we describe in Section 3.4 a newly added hybrid parallelization that combines the existing MPI implementation with a shared memory parallelization in DisCoTec. This implementation focuses on the compute intense parts of the combination steps, namely the hierarchization and dehierarchization. We outline the benefits and

[1] https://github.com/SGpp/DisCoTec

the current problems of such an approach in terms of the scaling properties and the resulting runtime. The main advantage is, however, that this implementation allows to couple solvers to DisCoTec which use an OpenMP [87] parallelization.

In Section 3.5, we show an asynchronous version of the combination technique that continues the computation during the combination step. After the combination has finished, the combination result is applied as a correction to the ongoing calculations. With this scheme it is possible to achieve an overlap between communication and computation which makes the method more efficient for large scale simulations and situations where frequent recombination is required.

Finally, we show the numerical results in Section 3.6. Here, we introduce in Section 3.6.1 our main target application from plasma physics, which represent a real-world application scenario for the Combination Technique. In particular, we look at the application code GENE which offers multiple operation modes and can simulate various fusion reactors. We apply the Combination Technique to linear as well as non-linear simulations, which involves several modifications and adjustments to the HPC framework and to GENE. Thereafter, we will show convergence studies and a detailed error analysis of our newly added features. We demonstrate the effectiveness of our implementation with linear simulations and also show recent results with the non-linear plasma simulations in Section 3.6.4 that show existing shortcomings of the method for turbulent scenarios.

3.1 Framework overview

We have already mentioned the two main reasons for creating an HPC framework for the combination technque: the independent computation of the component grids and the black-box property. It is, however, far from trivial to integrate the Combination Technique for time-dependent PDEs into an efficient HPC framework as it is required to recombine the grids at regular intervals (see Section 2.3.5). This recombination introduces synchronization points at which the different component grid solutions need to be exchanged. Hence, two main challenges arise for an HPC implementation: optimizing the load balancing to avoid idling processors at the synchronization points and reducing the communication overhead of the combination step. In Section 3.1.1, we outline the main design of DisCoTec which enables an efficient coordination of the computation including load balancing. The considerations concerning an efficient communication strategy during the combination step are described in Section 3.1.2.

3.1.1 Manager worker scheme

The main programming paradigm used in DisCoTec is the manager worker pattern. Here, a dedicated manager process coordinates the execution of the Combination Technique by distributing work packages to several worker processes. The workers are divided into process groups which work together on a common task. In our case a task is the computation of a component grid for the specified time-dependent PDE. For each

process group the manager controls the workflow and coordinates the synchronization between the groups as the Combination Technique requires frequent recombination.

As a result, the workers act according to a defined finite state machine (see Fig. 3.1). Initially all workers enter the waiting state and wait for the manager to indicate which action needs to be performed. The action is then selected based on a signal that is sent by the manager. Consequently the workers can enter either the RUN FIRST, RUN NEXT, COMBINE, EVAL, RECOVER, and RECOMPUTE state. The RUN FIRST signal specifies a concrete component grid that should be computed by the group. In particular, the grid needs to be initialized, added to the list of owned tasks, and then processed by the solver. In contrast to this, the RUN NEXT command does not specify an individual component grid but initiates the execution of all tasks owned by the process group. RUN FIRST is therefore only used in the initial distribution phase where the tasks are distributed one after the other to the process groups, while RUN NEXT is used after each combination to restart the execution. The COMBINE signal starts a combination step between all process groups that adds up the individual solutions to form a global Sparse Grid and then projects it back to the component grids. A more detailed description of this step is given in Section 3.1.2. The EVAL action triggers an evaluation of the Sparse Grid solution which is then written to a checkpoint. Typically not a Sparse Grid data structure but an interpolation onto a full grid is used for such a checkpoint. The RECOVER and RECOMPUTE signals are used in case a failure occurred during the execution. These routines implement the fault recovery of the FTCT. Section 3.2 explains this process in more detail. After the workers finished the computation of a state, they enter the READY state to inform the manager that they have finished their work. In case a process failure was detected in their process group, they immediately switch from the READY state to the recovery routine. Otherwise, the workers return to the WAIT state. To avoid a one-to-all communication pattern where the manager needs to send signals to all workers, each process group has a dedicated master process that receives the signals and then broadcasts it to the other members of the group. This heavily reduces the number of communications for the manager in practice.

In addition to the coordination of the workflows, the manager process is responsible for the efficient load-balance of the tasks. For this purpose, we use two common load balancing techniques. First, we estimate the runtime of each tasks. Second, we apply a suited task scheduling that accounts for inaccuracies in the runtime estimates. The runtime estimation can be done in various flavours, starting from the pure counting of the degrees of freedom, to a data-driven model based on a polynomial least-squares fit [52] or a machine learning approach such as neural networks [98]. We then sort the tasks according to a decreasing runtime estimate and apply dynamic task scheduling, i.e. we distribute the first g tasks to the g process groups and then each process groups requests a new tasks once it finishes the computation. This approach is common in many parallel programming implementations such as OpenMP and originates from the bin packing problem. For more details we refer to [51, section 3.2].

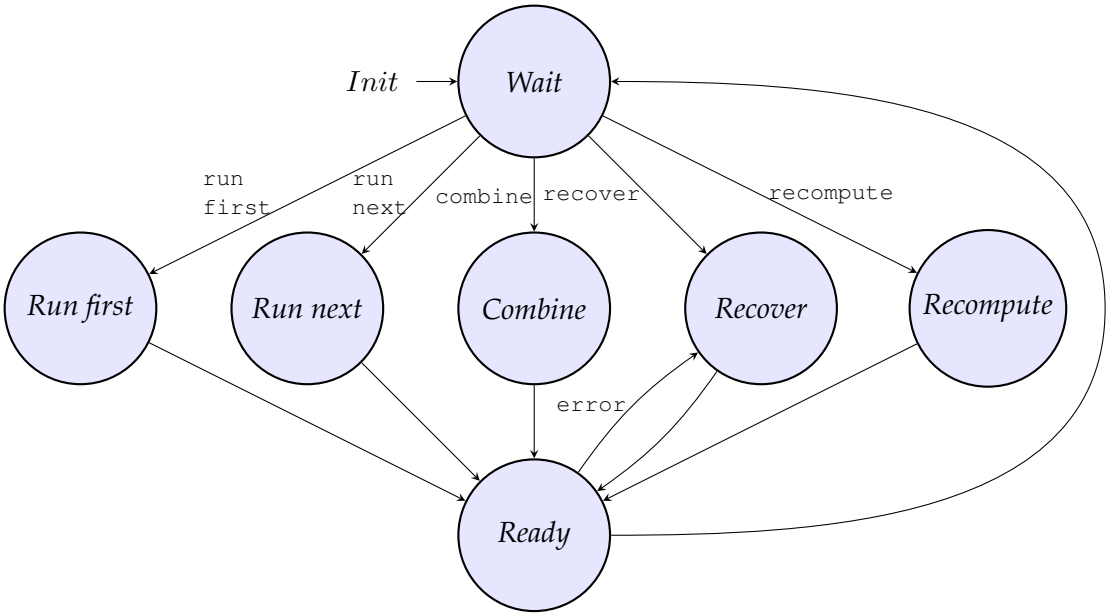


Figure 3.1: Finite state machine that describes the different steps in DisCoTec.

3.1.2 Scalable implementation of the combination step

The combination step represents the most critical part in an HPC implementation of the Combination Technique. Since the method requires frequent recombination for time-dependent PDEs (see Section 2.3.5), synchronization points are introduced. In addition, global communication of all grid values is required during the combination step. The aim of a scalable implementation should therefore reduce the data that needs to be communicated between processes and optimize the communication patterns. In [55] (see also [51]) two key components were investigated: the data exchange pattern between process groups and the domain decomposition of grids within a process group.

For the data exchange pattern two main schemes were considered: the *Sparse Grid Reduce* and the *Subspace Reduce*. In the *Sparse Grid Reduce* every process group builds up a full Sparse Grid representation and sums up the local contributions. Thereafter, these local Sparse Grid representations are combined between the process groups via a global reduce. In this way, only a single global communication is necessary, but the communication volume is large. In the *Subspace Reduce* pattern, all subspaces are reduced independently. Hence, an iterative communication scheme is applied were each process group iterates through all subspaces of the Sparse Grid and then performs a global reduction for each subspaces. Of course, a process group only contributes to the reduction operations of corresponding subspace that are contained in one of the component grids owned by the group. As a result, each process group only communicates the necessary amount of data, but the number of communications is proportional to the number of subspaces which increases the latency. The *Subspace Reduce* can be further improved by optimizing the ordering in which the subspaces are combined (see [51,

section 2.7]). The results of [51, chapter 2] indicate that the *Sparse Grid Reduce* is better suited for the load balancing scheme in `DisCoTec` and it is expected to perform decent while being easy to maintain. Hence, we decided to only consider *Sparse Grid Reduce* in this work.

The domain decomposition of the component grids within the process groups is another crucial aspect. This is not just relevant for the global reduction of the subspaces but also for the hierarchization and subsequent dehierarchization as the black box solver is usually implemented for the nodal basis. Moreover, different decompositions for the nodal and the hierarchical basis are possible. Here, a standard domain decomposition of the PDE solver was assumed which splits the domain into $\boldsymbol{p} \in \mathbf{N}^d$ processors, i.e. a cartesian grid decomposition with p_k subregions in dimension k. The total number of processors of a process group is therefore assumed to be $p = \prod_{k=1}^{d} p_k$. Since the global combination is performed in the hierarchical basis, the most important aspect is the domain decomposition of the hierarchical surplus values for the component grids and the Sparse Grid representation.

Mario Heene et al considered two approaches (see [55] and [51, chapter 3]): a domain decomposition according to subspaces and a geometric decomposition of points equal to the nodal decomposition. In the subspace decomposition, a process gets assigned a set of subspaces and stores them completely, whereas in the geometric decomposition a process stores all surplus values associated to points of the geometric area that falls into his local subarea of the cartesian decomposition. Again the subspace-oriented decomposition introduces a large amount of communication, whereas the geometric approach has a small amount of communication for hierarchization/dehierarchization and is even communication-free for the reduction into the distributed Sparse Grid if it is also geometrically distributed. In addition, the geometric approach fits perfectly to the *Sparse Grid reduce* reduction which was chosen for the `DisCoTec` implementation. Especially, if all process groups apply the same domain decomposition, all processes of a process group only communicate with the processes of the other groups that are responsible for the same domain region. This facilitates the communication pattern and therefore reduces the communication overhead. Unfortunately, this restricts the parallel efficiency of the black box solvers as for some anisotropic grids the domain decomposition might be suboptimal for parallelization. Therefore, a version with varying process group sizes was investigated in collaboration with Martin Molzer in [79]. However, in this work we will restrict ourselves to fixed process group sizes for all component grids as this approach offered sufficient performance for our use cases. Considering all these aspects, we decided to use a geometric decomposition in `DisCoTec`.

The complete combination step is summarized in Algorithm 2. First, all process groups g compute in parallel the hierarchization of the nodal grid values $\mathcal{H}(u_{\boldsymbol{\ell}}^n)$ for all of the component grids $\boldsymbol{\ell} \in \mathcal{I}_g$ they own. This results in the hierarchized grid values $u_{\boldsymbol{\ell}}^h$ (Line 4). The hierarchization (see Section 2.2.2) is implemented via a cache-efficient and parallel version of the unidirectional principle. We refer to [8, 62, 63, 54] or [51, section 3.3.1] for more details on the unidirectional principle and its parallel implementation. Next, the hierarchical surplus values of all local component grids are combined into

Algorithm 2 The Combination Technique in Parallel

1: **procedure** PARALLEL_COMBINATION($G, \mathcal{I}_1, \ldots, \mathcal{I}_G$)
2: **for** $g \in [G]$ **do in parallel**
3: **for** $\boldsymbol{\ell} \in \mathcal{I}_g$ **do**
4: $u^h_{\boldsymbol{\ell}} = \mathcal{H}(u^n_{\boldsymbol{\ell}})$ ▷ local hierarchization
5: **end for**
6: $u^{c,g} = \sum_{\ell \in \mathcal{I}_g} c_\ell \cdot u^h_{\boldsymbol{\ell}}$ ▷ local combination
7: **end for**
8: $u^c = \sum_{g=1}^{G} u^{c,g}$ ▷ global combination
9: **for** $g \in [G]$ **do in parallel**
10: **for** $\ell \in \mathcal{I}_g$ **do**
11: $u^h_{\boldsymbol{\ell}} = \mathcal{P}_{\boldsymbol{\ell}}(u^c)$ ▷ project back onto component grids
12: $u_{\boldsymbol{\ell}} = \mathcal{D}(u^h_{\boldsymbol{\ell}})$ ▷ local dehierarchization
13: **end for**
14: **end for**
15: **end procedure**

a local Sparse Grid representation $u^{c,g}$ (Line 6). After that, the global combination is performed using the *Sparse Grid Reduce* where the local Sparse Grid representations are summed up to get the final Sparse Grid values u^c. Thereafter, all process groups project the surplus values ($\mathcal{P}_{\boldsymbol{\ell}}$) of the Sparse Grid back onto the component grid $\boldsymbol{\ell}$ by copying all surplus values of subspaces $\boldsymbol{l} \leq \boldsymbol{\ell}$ (Line 11). Then the hierarchized values are dehierarchized by applying the inverse function of the hierarchization $\mathcal{D}(u^h_{\boldsymbol{\ell}})$ (Line 12). The dehierarchization is implemented similar to the hierarchization step. In particular, it has the same data access pattern as the hierarchization [2] and for the computation of the nodal values the subtraction of the parent values has to be only switched to an addition.

3.2 Fault tolerance

In Section 2.1, we have described the main concepts of fault tolerance and outlined the imminent problem for future compute clusters that will face system failures on a regular basis. As a result, it will be hard to maintain maximum availability of the system for long simulation runs since subsystems might fail frequently. Thus, we need robust HPC codes that can tolerate failing components if we want to operate at large scales.

In Section 2.3.4, we have seen such a fault-tolerant approach of the Combination Technique. The FTCT can recover from faults that occur during the calculation of component grids by changing the combination scheme so that only fault-free computations are included. The resilience of the FTCT can be further enhanced by adding some means of fault tolerance to the solver that process the component grids to reduce the number of failing grids in practice. An example for such an approach can be seen in [45] where a fault tolerant implementation for eliptic PDEs is presented which is based on

[2] The only difference is that the hierarchical tree is traversed top to bottom instead bottom to top.

space-filling curves and data copies. In addition, a robust MPI implementation called libspina [72] was integrated into GENE in the context of the EXAHD project. With this addition, the GENE code is able to recover the application in cases of failure events. We can therefore expect that in practice component grid failures will be rare and are mainly caused by faults that can not be handled directly by the component grids.

In this section, we want to describe the technical details of our implementation of the FTCT in DisCoTec and will not go into the details on how to avoid component grid failures at the solver level. The core components of the FTCT were already implemented in close collaboration between Mario Heene and Alfredo Parra in the course of their PhD projects [51, 89]. As a first step, they analyzed the convergence of the FTCT in [90] for plasma physics. Here, the FTCT was not coupled to DisCoTec but all calculations were performed off-line, i.e. there was a post-processing step that performed a single combination with and without simulated faults. They then tested a massively parallel implementation within DisCoTec in [53] for hard faults with an advection-diffusion equation. Finally, they added functionality to detect and recover from soft faults to the code in [59].

We extended these implementations in [86] by introducing efficient handling of failed MPI ranks. Here, we added a restructuring of the MPI communicators and introduced fault distributions to simulate realistic scenarios. In addition, the new implementation was tested with a real-world application from plasma physics which was coupled with DisCoTec to allow for arbitrary recombination runs. The results can be found in Section 3.6.2.

In the following we cover the main aspects of the fault-tolerant implementation. First, we introduce in Section 3.2.1 the fault simulator that enables us to simulate failure events that might occur during the execution of the code. Next, Section 3.2.2 shows the possibilities in DisCoTec how to decide which process should fail and at which point. This includes predefined static failure assignments and realistic fault-distribution based on statistical parameters. We will then look at how to detect such a failure in Section 3.2.3 so that the program can react accordingly. Thereafter, Section 3.2.4 covers means of recovering from the faults and restoring a valid system state. This includes the restructuring of MPI communicators and exclusion of failed ranks. Finally, Section 3.2.5 summarizes the whole algorithm and compares it to the normal execution.

3.2.1 Fault simulator

The implementation of a fault-tolerant code makes it necessary to test the functionalities in a realistic fault scenario. It is therefore necessary that we have a simulation environment to create such failures and analyze the reaction of our code. Usually, one would implement such a simulator with the help of the fault-tolerant MPI implementation ULFM (*User Level Failure Mitigation*) [12]. Unfortunately, ULFM was not available on the compute clusters at the point of our studies. For this purpose, a fault simulation layer was implemented in the course of a Master's thesis [117]. This fault layer offers the functionalities of ULFM by implementing the ULFM interfaces and routines for handling faults. It also ensures that MPI communicators can be restored in presence

of faults. It is designed as a wrapper to normal MPI functions and can be used for any operation that should be fault tolerant. This is especially useful for blocking collective operations that cannot complete when an MPI rank is failing. If an ULFM implementation is available, the existing fault layer can be replaced by ULFM with minor code changes.

Another important feature of the fault simulator is the `Kill_me` functionality that "kills" the calling MPI rank. Of course, we cannot completely kill the rank if we want to use classical MPI. Instead, the rank enters a loop where it only contributes to necessary MPI communications and idles otherwise.

See [117] or [51, section 4.3] for further details on the implementation.

3.2.2 Fault distribution

We have now all tools at hand to simulate faults in our framework and to test its fault tolerance capabilities. If we want to analyze the realistic fault tolerance behavior of our function, we have to simulate the faults in a realistic manner. To achieve this, we have to consider the statistical effects of faults and account for the probabilistic nature of the occurrence of faults. For this purpose it is common to insert faults according to a probability distribution that matches closely the real-life observations. In [107] a large scale study showed that a Weibull distribution best models the time-between failures. The Weibull distribution is defined by

$$f(t;\lambda,k) = \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1} e^{-(t/\lambda)^k} \tag{3.1}$$

and $t \geq 0$. k denotes the shape and λ the scale parameter. These parameters can be adjusted to fit the simulation to a real world scenario. Here λ is a measure of the mean time between a failure $\mathrm{E}[f(t;\lambda,k)] = \lambda\Gamma(1+1/k)$. We can therefore adjust this parameter to account for different settings with different *Mean Time Between Failures*. As proposed in [48], we inject faults by drawing values from the Weibull distribution for each of the MPI ranks. In particular, this means that each MPI rank draws a value from the distribution at the beginning of the simulation which marks its *kill time*. After the computation of a task and before the combination step, the rank checks if the elapsed time since the start of the simulation exceeds this *kill time* in the `DecideToKill` function. In case it does, the rank calls the `Kill_me` function.

Of course, we can also statically define which cores should fail at which point of the simulation. This has the advantage that the failure scenarios can be better reproduced and it is possible to simulate specific fault scenarios. However, it is not suited to get accurate predictions for real-world situations.

3.2.3 Fault detection

So far, we have seen how to insert faults into our framework with the fault simulation layer. As a next step, the framework needs to detects such an error. This process is called fault detection. Here, we differentiate between soft faults and hard faults.

A soft fault or silent data corruption is an error that is not detected by the hardware or the operating system. It is therefore very important that the application can actively detect these faults to avoid data corruption and false results. Sources for these errors can be multifarious and range from malfunctioning hardware to cosmic rays that cause bitflips.

For hard faults, the hardware or operating system receives an error signal that can be forwarded to the application. This makes the detection process significantly easier. Typical problems here are to determine which resources failed and how to restore a fault-free MPI communicator. This process is usually supported by a fault-tolerant implementations of MPI such as ULFM. Sources of hard faults are typically failing hardware or software components.

In `DisCoTec` hard faults are detected via the simulated ULFM environment. Each ULFM command can return a specific error flag if an error was detected during its execution. The program can then react to the failure. For soft faults it is a lot harder since we do not get notified by the underlying system. Fortunately, the Combination Technique offers some redundancy as component grids share points with other component grids. The idea for a soft fault detection is now to compare the function values at these points and to find outliers. If a grid is significantly off for at least one grid point, it can be considered to be corrupted. For more information on the available detection algorithms we refer to [59, 90, 89].

In both cases, the manager process detects these errors before the start of the combination step and initiates the fault recovery routine. Here, the manager first informs all process group masters about the affected ranks which broadcast this information to the other processes of the group. Each process can then determine if his group is affected and reacts accordingly. The recovery process will be explained in the next section.

3.2.4 Fault recovery

Once we have discovered a fault, we need to consider countermeasures to restore the correct result of the computation and to guarantee a consistent and valid system state. This procedure does not differ between soft and hard faults and we therefore only use the term fault in the following.

The first thing that needs to be considered after a fault is detected is to find the affected component grids. Those are typically all grids that belong to the process groups in which the faults occurred. For a soft fault we could also only consider the specific erroneous grids, but the chances are high that other grids of the same group are also affected and therefore we are neglecting all computed results of the group. Since soft faults are expected to be very rare, this should not waste resources in practice.

After we have found the faulty component grids, we try to remove all their contributions in the combination scheme as we cannot use partial results in the Combination Technique. Here, we follow the scheme presented in Section 2.3.4 where we solve the adjusted GCP that was proposed by Harding in [48]. This results in a new combination scheme that contains typically no affected grids. In case such a combination scheme is not found, it is allowed to recompute component grids that have a level vector ℓ with

$\|\boldsymbol{\ell}\|_1 \leq \ell + d - 3$. These grids have significantly less points than the large component grids of the first layer. Thus, it only produces little overhead to allow these recomputations. It should be noted that in practice it is rarely necessary to recompute unless we have few process groups with a large number of tasks which then causes a high proportion of the overall grids to fail.

With this combination scheme that contains no affected grid, we can perform the combination and can then continue with the computation of the next time steps of the PDE. For these next time steps, we regenerate the original combination scheme and interpolate the missing grid values of the discarded grids from the combined Sparse Grid representation just as in the regular combination step. The only difference is that the missing hierarchical subspaces of the global Sparse Grid are filled up with zeros which is identical to a linear interpolation of the missing values.

In addition to these numerical considerations, we need to restore a valid system state. As the error detection found some malfunctioning or failed ranks, we want to remove them from the MPI communicators. This can be done via the ULFM interface of our fault simulation layer. Here we just remove the erroneous ranks from the global communicator.

The question now is what to do with the remaining ranks of the process groups that have not failed. Since all process groups have a fixed size that cannot be changed, the easy solution would be to just remove the whole group any time a fault occurs. Such an approach obviously wastes resources and should be avoided. Hence, we define the remaining ranks of the faulty groups as *spare ranks* and the ranks of healthy groups as *active ranks* (see Fig. 3.2). Next, we introduced two MPI communicators: one *spare* communicator containing spares and active ranks and one *world* communicator only containing the active ones. The active processes are assigned to the respective *world* communicator via MPI_Split. In case further errors happen, we can use these spares to replace failed ranks. In such a case, they are exchanged with the faulty rank of the *world* communicator, thereby restoring the process groups (see Fig. 3.3). In particular, we assign the identical rank number to the *spare rank* so that the spare exactly replaces the corresponding failed ranks. As a consequence, this exchange is made transparent for the application. Thus, there is no need to adjust the application if such an exchange takes place. It would also be possible to add some *sparse ranks* right from the beginning to avoid the initial discarding of a process group after the first error. We decided against this option as this would also waste resources in cases where no faults occur.

Another important aspect is the load distribution if a process group gets lost in the process of the recovery. In such a case the manager just redistributes the grids that were assigned to the lost group in a round robin fashion. The initially assigned grids are kept unchanged to avoid excessive transferring of tasks and their data between process groups. If this load balancing shows significant unbalance, one could also decide to redistribute all tasks and restart everything by interpolating from the combined Sparse Grid. In our test cases this was not necessary.

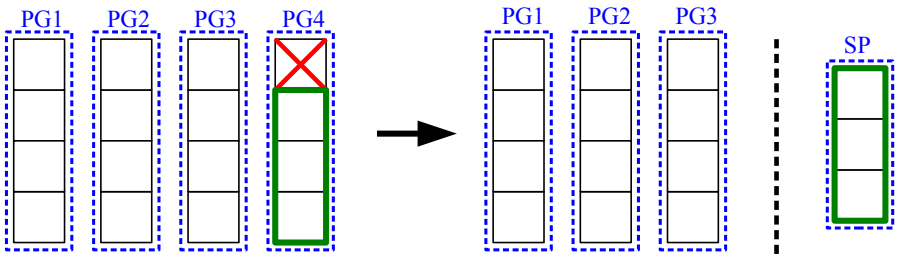


Figure 3.2: In this figure, group 4 fails due to a fault in the first rank and the remaining three ranks are stored as spare processes. The process group can not be recovered as there are no spare ranks to restore a group of size 4.

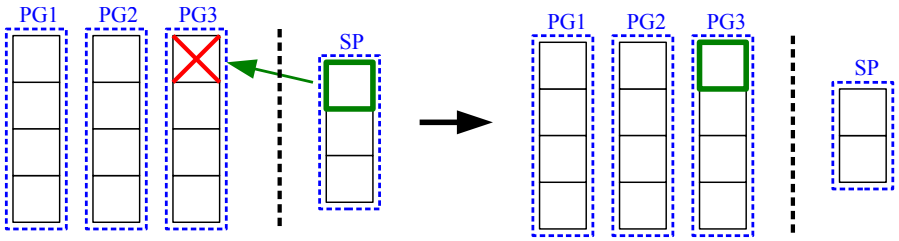


Figure 3.3: A failing rank in process group three is replaced by one of the spare ranks. As a consequence, the process group can be recovered and continues computation.

3.2.5 Fault-tolerant algorithm

In the previous sections we have discussed the individual components of our implementation for the FTCT. We now show in Algorithm 3 how these components are integrated into the overall procedure. The inputs to this algorithm are the dimensionality of the problem d, the minimum and maximum level $\boldsymbol{\ell}^{\min}$ and $\boldsymbol{\ell}^{\max}$, the number of combination steps N_c, the timeframe that should be simulated in between combinations t_c, and the number of process groups G.

In the algorithm, we first start with initializing the combination scheme and distributing the component grids – referred to as tasks – according to a load balancing scheme. Then, we set the initial condition for all tasks and the killing time for each processor in case we want to simulate a realistic fault scenario. After the initialization process, we enter the computation loop. Here, all process groups compute in parallel the solutions for their component grids. Next, each process decides whether he should fail in DECIDE_TO_KILL(). This process measures the elapsed time and checks if it exceeds his *kill time* (see Section 3.2.2) and proceeds accordingly. Once the computations are completed, the manager notifies all groups in case a fault occurred. In such a scenario, the recovery method is started that potentially shrinks the number of process groups and reassigns tasks. With these modifications, the parallel combination step from Section 3.1.2 is started. The only difference to the original procedure is here that the combination coefficients $c_{\boldsymbol{\ell}}$ could be changed due to the updated combination scheme that results from solving the GCP (see Section 2.3.4). The original coefficients are however

Algorithm 3 Parallel implementation of the FTCT

procedure PERFORM_FTCT($d, \boldsymbol{\ell}^{\min}, \boldsymbol{\ell}^{\max}, N_c, N_t, G$)
  $\ell = \min(\{n \in \boldsymbol{\ell}^{\max} - \boldsymbol{\ell}^{\min}) | n > 0\}$ and $\tilde{\boldsymbol{\ell}}_k^{\min} = \max(\boldsymbol{\ell}_k^{\max} - \ell, \boldsymbol{\ell}_k^{\min})$
  $\mathcal{I}_\ell^{\tilde{\boldsymbol{\ell}}^{\min}} = \bigcup_{q=0}^{d+1} \mathcal{I}_{\ell,q}^{\tilde{\boldsymbol{\ell}}^{\min}}$ and compute $c_{\boldsymbol{\ell}}$ ▷ Initialize combination scheme
  $\mathcal{I}_1, \ldots, \mathcal{I}_G$ = LOAD_BALANCING($\mathcal{I}_\ell^{\tilde{\boldsymbol{\ell}}^{\min}}, G$) ▷ Distribute tasks among G groups
  Set initial conditions $u_{\boldsymbol{\ell}} = u(t=0)$ for $\boldsymbol{\ell} \in \mathcal{I}_\ell^{\tilde{\boldsymbol{\ell}}^{\min}}$ and *kill time*
  for $n \in [N_c]$ **do** ▷ Perform N_c combination steps
    for $g \in [G]$ **do in parallel**
      for $\boldsymbol{\ell} \in \mathcal{I}_g$ **do**
        $u_{\boldsymbol{\ell}}$ = SOLVE($u_{\boldsymbol{\ell}}, t_c$) ▷ Compute a simulation time of t_c with the solver
        DECIDE_TO_KILL() ▷ Decide if process should die
      end for
    end for
    if faults detected **then**
      $\tilde{G}, \tilde{\mathcal{I}}_1, \ldots, \tilde{\mathcal{I}}_{\tilde{G}}$ = RECOVER($G, \mathcal{I}_1, \ldots, \mathcal{I}_G$)
    else
      $\tilde{G}, \tilde{\mathcal{I}}_1, \ldots, \tilde{\mathcal{I}}_{\tilde{G}} = G, \mathcal{I}_1, \ldots, \mathcal{I}_G$
    end if
    PARALLEL_COMBINATION($\tilde{G}, \tilde{\mathcal{I}}_1, \ldots, \tilde{\mathcal{I}}_{\tilde{G}}$) ▷ see Algorithm 2
  end for
end procedure

restored immediately after the combination. After this step we continue with the next computation step. The algorithm terminates once the maximum number of combination steps is reached.

3.3 Choosing the time step

For time-dependent PDEs the selection of the right time step is a crucial element. Often the time step is governed by the discretization of the domain and not by accuracy concerns. Here, the mesh width in the different dimensions typically limits the time step size of a grid. Since the mesh-widths in the component grids vary significantly from very isotropic to anisotropic grids, it might be better to apply an individual time step on each component grid instead of the common approach to choose one fixed time step for all component grids.

In this chapter, we summarize the results that were obtained in collaboration with Thomas Bellebaum as part of his Bachelor's thesis [10]. Our analysis matches the results of Lastdrager et al. [73] who showed that for a 2D time-dependent advection problems the complexity of the Combination Technique can be reduced to $\mathcal{O}(\tilde{N}^2)$ by choosing appropriate time steps on each component grid. They also considered the resulting approximation error, while we focus on the computation complexity for a fixed level ℓ

but extend the analysis to arbitrary dimensions. The reason for this is that we assume that the stability is the primary concern for selecting the time step. We summarize the main aspects and refer to [10] for further details. Some formulas differ slightly from the original paper due to a different definition of the standard Combination Technique in [10] but the complexities remain unchanged.

3.3.1 CFL condition

A famous criterion for the stability of a PDE in respect to the mesh width is the Courant-Friedrichs-Lewy (CFL) condition [22] developed by Courant, Friedrich and Lewy back in 1928. They defined the *mathematical/physical domain of dependence* and the *numerical domain of dependence*. Only if the *mathematical/physical domain of dependence* lies within the *numerical domain of dependence* a numerical scheme can converge. To better understand this important finding, we have to explain the terms in more detail. On the one hand the *mathematical/physical domain of dependence* can be seen as the region that impacts a certain point of our PDE solution. On the other hand, the *numerical domain of dependence* is the region that can influence a certain point through the numerical scheme, i.e. from where changes can be propagated to the respective point. An example would be a first order explicit scheme where information propagates from one point to the adjacent points. In this case, the information flow of the PDE (*mathematical/physical domain of dependence)* is not allowed to move faster in each dimension than the mesh size. Therefore, the time step has to be adjusted accordingly.

A mathematical expression for one dimension would be

$$\frac{\Delta t|v|}{\Delta x} \leq C \tag{3.2}$$

where Δt is the time step size, Δx the mesh width, v the speed of the information flow in the PDE and C the CFL number. The CFL number expresses how far information travels in the numerical scheme for one time step. In the example mentioned above, it would be $C = 1$.

For higher-dimensional cases the CFL condition can be generalized for example by

$$\sum_{k=1}^{d} \frac{\Delta t|v_k|}{\Delta x_k} \leq C \tag{3.3}$$

where v_k is the information flow in the direction of dimension k and Δx_k the respective mesh width in this dimension. There exist also other generalizations for higher dimensions but we will restrict our analysis to this formalization [10].

It should be noted that the CFL condition is not sufficient for a numerical solution to converge to the exact solution but it is a necessary prerequisite. In the following, we will solely focus on the CFL condition.

3.3.2 Uniform time steps

We have now described a criterion for the selection of the time step size. Next, we need to analyze the complexity for the computation of a time-dependent PDE with the Combination Technique if we choose a uniform time step across all component grids.

Since this time step has to satisfy all the stability criteria for each component grid, we have to select the most restrictive, i.e. the smallest time step. We therefore have

$$\Delta t = \min_{\boldsymbol{\ell}\in\mathcal{I}} \frac{C}{\sum_{k=1}^{d} \frac{|v_k|}{\Delta x_k}} = \min_{\boldsymbol{\ell}\in\mathcal{I}} \frac{C}{\sum_{k=1}^{d} \frac{|v_k|}{2^{-\ell_k}}} = \frac{C}{\max_{\boldsymbol{\ell}\in\mathcal{I}} \sum_{k=1}^{d} |v_k| 2^{\ell_k}} \tag{3.4}$$

where $\mathcal{I}$ denotes the index set of the level vectors used in our scheme. For each level vector $\boldsymbol{\ell}$ the mesh width is defined by $\Delta x_k = 2^{-\ell_k}$. If we consider a standard Combination Technique of level ℓ, it is obvious that the most restrictive time step has to belong to one grid of the hyperplane with $\|\boldsymbol{\ell}\|_1 = \ell + d - 1$. Another observation is that $\max_{\|\boldsymbol{\ell}\|_1=\ell+d-1} \sum 2^{\ell_k} = 2^{\ell} + 2(d-1) \in \mathcal{O}(2^{\ell})$, i.e. the most anisotropic grids have the most restrictive time step. Since the information speed $\boldsymbol{v}$ is assumed to be bounded and C is a constant, we obtain $\Delta t \in \mathcal{O}(2^{-\ell})$. As a result, the number of time steps is $\mathcal{O}(2^{\ell})$ if we assume a fixed simulation length.

The overall complexity is the number of grid points times the number of simulation steps which evaluates to $\mathcal{O}(2^{\ell} \cdot 2^{\ell}\ell) = \mathcal{O}(\tilde{N}^2 log(\tilde{N}))$ (see also Section 2.2 for definition of $\tilde{N}$ and the number of Sparse Grid points). Again, we neglect constant factors that depend on the dimension d in this complexity consideration.

3.3.3 Individual time steps

In the last section, we have seen that the most anisotropic grids require the strictest time steps according to the CFL condition. However, most component grids could use a much larger time step. Therefore, instead of choosing a common time step for each component grid, we can use an individual time step for each component grid. As we assume that the CFL condition is more restrictive than the accuracy requirements, we set the time step of each grid to the maximum allowed time step size

$$\Delta t = \frac{C}{\sum_{k=1}^{d} \frac{|v_k|}{\Delta x_k}} = \frac{C}{\sum_{k=1}^{d} |v_k| 2^{\ell_k}}. \tag{3.5}$$

Now, we approximate the number of grid points of a component grid by $2^{\|\boldsymbol{\ell}\|_1}$ which is not exact but only off by a constant. This is similar to the approach in [36]. Since each grid needs to calculate $\mathcal{O}((\Delta t)^{-1})$ time steps, we can approximate the overall computation cost by

$$\sum_{\boldsymbol{\ell}\in I} \frac{2^{\|\boldsymbol{\ell}\|_1}}{\Delta t} = \sum_{\boldsymbol{\ell}\in I} \frac{2^{\|\boldsymbol{\ell}\|_1} \cdot \sum_{k=1}^{d} |v_k| 2^{\ell_k}}{C}. \tag{3.6}$$

For the standard Combination Technique with $I = \{\boldsymbol{\ell} \in \mathbb{R}^d | \ell \leq \|\boldsymbol{\ell}\|_1 \leq \ell + d - 1\}$[3] this further evaluates to

$$
\begin{aligned}
&\sum_{q=0}^{d-1} \sum_{\|\boldsymbol{\ell}\|_1=\ell+d-1-q} \frac{2^{\|\boldsymbol{\ell}\|_1} \cdot \sum_{k=1}^{d} |v_k| 2^{\ell_k}}{C} \\
=&\frac{1}{C} \sum_{q=0}^{d-1} 2^{\ell+d-1-q} \sum_{\|\boldsymbol{\ell}\|_1=\ell+d-1-q} \sum_{k=1}^{d} |v_k| 2^{\ell_k} \\
=&\frac{1}{C} \sum_{q=0}^{d-1} 2^{\ell+d-1-q} \sum_{k=1}^{d} |v_k| \sum_{\|\boldsymbol{\ell}\|_1=\ell+d-1-q} 2^{\ell_k} \\
=&\frac{1}{C} \sum_{q=0}^{d-1} 2^{\ell+d-1-q} \sum_{k=1}^{d} |v_k| \sum_{m=1}^{\ell-q} 2^m \sum_{\|\boldsymbol{\ell}\|_1=\ell+d-1-q \wedge \ell_k=m} 1 \\
=&\frac{1}{C} \sum_{q=0}^{d-1} 2^{\ell+d-1-q} \sum_{k=1}^{d} |v_k| \sum_{m=1}^{\ell-q} 2^m \binom{\ell+d-1-q-m-1}{d-2} \\
\overset{z:=\ell-q-m}{=}&\frac{1}{C} \sum_{q=0}^{d-1} 2^{\ell+d-1-q} \sum_{k=1}^{d} |v_k| \sum_{z=0}^{\ell-q-1} 2^{\ell-q-z} \binom{z+d-2}{d-2} \\
\overset{x:=0.5}{=}&\frac{\|\boldsymbol{v}\|_1}{C} \sum_{q=0}^{d-1} 2^{2\ell+d-1-2q} \sum_{z=0}^{\ell-q-1} x^z \binom{z+d-2}{d-2}
\end{aligned}
\tag{3.7}
$$

According to [10, 36], we can find an upper bound to the inner part of the equation which we substitute by the function $f_q(x)$. By rewriting this function as the n-th derivative with $n = d - 2$, it is possible to get a simplified formula via

[3] Here we exclude level vectors $\boldsymbol{\ell}$ that have a coefficient $c_{\boldsymbol{\ell}} = 0$

$$
\begin{aligned}
f_q(x) = &\sum_{z=0}^{\ell-q-1} x^z \binom{z+d-2}{d-2} \\
=&\frac{1}{(d-2)!}\left(\sum_{z=0}^{n-q}(x^{z+d-2})^{(d-2)}\right) = \frac{1}{(d-2)!}\left(\sum_{z=0}^{n-q} x^{z+d-2}\right)^{(d-2)} \\
<&\frac{1}{(d-2)!}\left(\sum_{z=0}^{\infty} x^{z+d-2}\right)^{(d-2)} = \frac{1}{(d-2)!}\left(x^{d-2}\frac{1}{1-x}\right)^{(d-2)} \\
=&\frac{1}{(d-2)!}\left(\sum_{k=0}^{d-2}\binom{d-2}{k}\left(x^{d-2}\right)^{(k)}\left(\frac{1}{1-x}\right)^{(d-2-k)}\right) \\
=&\frac{1}{(d-2)!}\left(\sum_{k=0}^{d-2}\binom{d-2}{k}\frac{(d-2)!}{(d-2-k)!}\left(x^{d-2-k}\right)(d-2-k)!\left(\frac{1}{1-x}\right)^{d-2-k}\right) \\
=&\sum_{k=0}^{d-2}\binom{d-2}{k}\left(x^{d-2-k}\right)\left(\frac{1}{1-x}\right)^{d-2-k}
\end{aligned}
\tag{3.8}
$$

Now, we set again $x = 0.5$ which results in

$$
f_q(0.5) = \sum_{k=0}^{d-2}\binom{d-2}{k} = 2^{d-1}. \tag{3.9}
$$

If we subsitute this back into the Eq. (3.7), we get

$$
\begin{aligned}
&\frac{\|\boldsymbol{v}\|_1}{C}\sum_{q=0}^{d-1} 2^{2\ell+d-1-2q}\sum_{m=1}^{\ell-q} x^z\binom{z+d-2}{d-2} \\
<&\frac{\|\boldsymbol{v}\|_1}{C}\sum_{q=0}^{d-1} 2^{2\ell+d-1-2q}\cdot 2^{d-1} \\
=&\frac{\|\boldsymbol{v}\|_1}{C}\sum_{q=0}^{d-1} 2^{2\ell+2d-2-2q} \in \mathcal{O}(2^{2d}2^{2\ell}) = \mathcal{O}(2^{2\ell}) = \mathcal{O}(\tilde{N}^2).
\end{aligned}
\tag{3.10}
$$

As a consequence, the overall complexity for computing the solution of a time-dependent PDE with individual time steps is $\mathcal{O}(\tilde{N}^2)$ which is a clear improvement compared to the $\mathcal{O}(\tilde{N}^2 log(\tilde{N})^{d-1})$ complexity of the uniform time stepping. Even more surprising is that all dimension-dependent terms disappear and hence the solution with the Combination Technique overcomes completely the curse of dimensionality. However, since the dimension still appears in the constant factors, there is still a limit for the feasible dimension in practice. In addition, we do not consider the accuracy of the computation here as we assume that fulfilling the CFL condition already guarantees sufficient

accuracy. In cases where the target accuracy demands a stricter time step, the isotropic grids might require smaller time steps which then reduces the benefit of individual time stepping.

Another important aspect is the synchronization at the combination step. Since every grid has an individual time step, it is not guaranteed that every grid will calculate a solution at the synchronization point. A simple but effective method is to shorten the last time step if we would exceed the global synchronization point. This does not change the overall complexity as long as the global combination interval is long enough. For use cases where the combination interval is larger than the maximum of all individual time steps, the overhead of this approach is less than 100% as every component grid will at most take twice as many steps as usual. Hence, the overall complexity is unaffected. If we, however, choose the global combination interval to be as short as the minimum time step of the most anisotropic grids, we again result in a uniform time stepping scheme.

3.4 Shared-Memory parallelization

The original parallelization concept of `DisCoTec` was based solely on MPI. This parallelization does not differentiate between ranks on the same node or ranks on different nodes. However, many HPC solvers nowadays rely on a hybrid parallelization scheme that combines MPI with a shared-memory parallelization to increase the performance. Hence, we implemented for `DisCoTec` a new hybrid mode that adds OpenMP [87] support to the existing MPI version. This work is part of a collaboration with Sven Hingst for his Bachelor's thesis [58]. In the following, we summarize our main findings.

For the shared-memory parallelization, we focus on the most compute intense parts of `DisCoTec`, namely the hierarchization und the dehierarchization. The parallelization of these methods is quite simple as they are based on the unidirectional principle [8, 15]. The structure of the hierarchization is therefore as follows. In an outer loop we iterate over all dimensions. Here, we have a strong dependency between the iterations and cannot compute them in parallel. The next inner loop iterates over the $d-1$ dimensional slice of the grid that excludes the current (de)hierarchization dimension. At each position of this slice, a one-dimensional (de)hierarchization routine is started which is completely independent of the others. As usually the number of these one-dimensional computations is large, we can easily parallelize them with a PARALLEL FOR in OpenMP. The general principle is the same for hierarchization and dehierarchization. In order to enable a thread-safe implementation, we needed to adjust some of the datastructures and carefully design the in and outputs of the parallel for. For more information on these adjustments and for additional tests we refer to [58]. All tests that we present next were performed on the CoolMUC-2 linux cluster system (see Appendix A.3).

First, we compared the hybrid parallelization mode to the MPI-only version with the hierarchization routine for different process group sizes and different thread numbers.

The results (see Fig. 3.4) show that if we carefully select the right amount of OpenMP threads, we can surpass the MPI-only version. This is mainly due to the efficient calculation of the independent subproblems during the hierarchization steps. Consequently, the shared memory parallelization helps to reduce some of the overhead that the MPI abstraction introduces on a single node. For the dehierarchization similar observations were made.

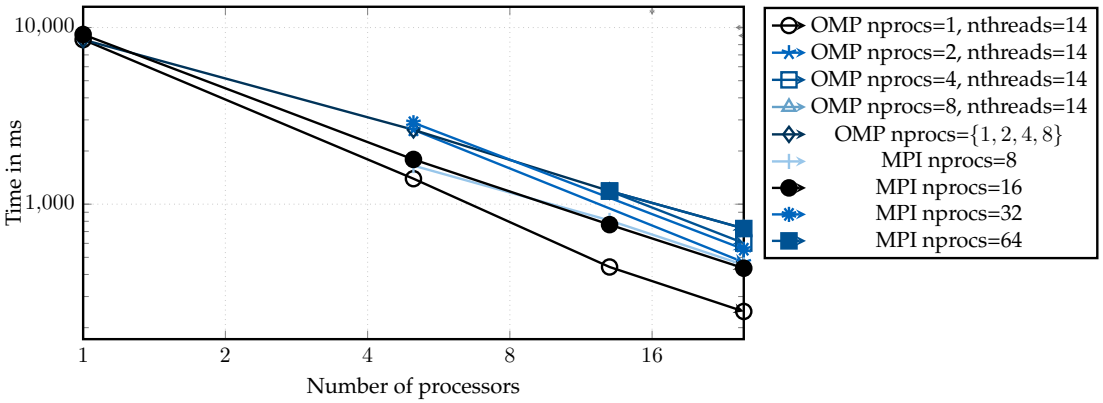


Figure 3.4: Comparision of the performance of the hierarchization with or without shared memory parallelization for different configurations [58].

However, not all steps of the Combination Technique could be sped up with the hybrid scheme. As the communication with MPI is now reduced to a lower amount of MPI ranks per node[4], the communication volume of the global reduction step during the combination increases and the number of communication partners decreases. This is caused by the fact that only MPI ranks communicate and with the hybrid parallelization, we reduce the MPI ranks but keep the same amount of data per node. Hence, the amount of data that is communicated per MPI rank in the global reduction grows. This should not be a problem if the fewer communications can still utilize the bandwith efficiently. However, we observed a severe impact on the CoolMUC-2 system. One possible explanation might be that the bandwidth can only be fully utilized if enough MPI communications take place. We therefore tried to use threaded MPI implementations and asynchronous communications and split the communication volume similar to the MPI-only version. All in all these measures could improve the performance and the best results were achieved with an intel-specific MPI_THREAD_SPLIT mode. Unfortunately, not even this variant could fully reach the MPI-only version for all test cases (see Fig. 3.5) . The reason for this effect seems not to be in the algorithm itself but in the current development state of threaded MPI and cluster-specific properties for the communication. We therefore expect this to be improved for future clusters and upcoming MPI implementations.

It is no surprise that this reduced efficiency of the global reduce also affects the overall performance of the complete combination step in DisCoTec. A comparison to an MPI-

[4]Usually 1 or 1 per NUMA domain.

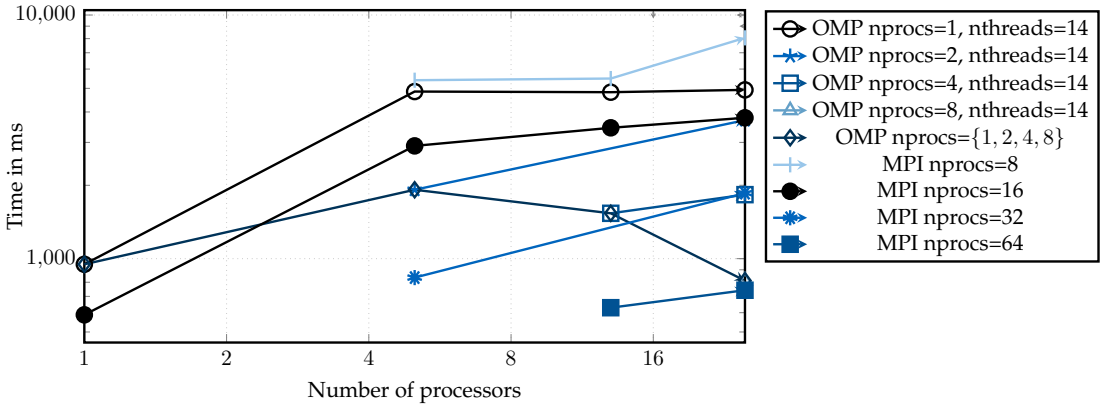


Figure 3.5: Comparision of the performance of the global reduction for an MPI-only implementation and a hybrid approach with multithread MPI_ALLREDUCE and MPI_THREAD_SPLIT enabled [58].

only version for a specific test case can be seen in Fig. 3.6. Again, the MPI-only versions tend to perform better for certain configurations.

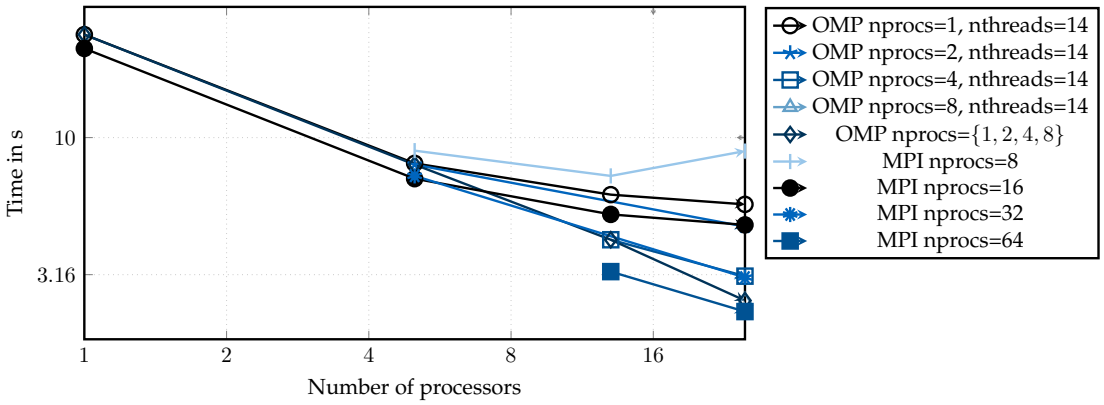


Figure 3.6: Comparision of performance the combination with high minimum level, with or without shared memory parallelization [58].

To put it in a nutshell, we see that an hybrid parallelization with OPENMP can be beneficial for the (de)hierarchization but current limitations with threaded MPI can impact the performance of the global reduce. As a consequence the overall performance is often best with an MPI-only implementation. However, the most compute intense part of a simulation with `DisCoTec` is not the combination step but the coupled PDE solver. Usually the combination takes orders of magnitude less time. Therefore, the possibility to offer hybrid parallelization can be very valuable if a solver supports and benefits from it. In addition, future advances in threaded MPI should mitigate the current shortcomings. Moreover, for other test cases or scenarios the results could be different and

further results in [58] suggest that for lower minimum levels the hybrid variant can already outperform the MPI-only implementations.

3.5 Asynchronous Combination Technique

We have seen that one of the main challenges in implementing the Combination Technique on distributed systems is the overhead that arises due to the global communication in the combination step. Here, we usually have to synchronize all ranks and perform a global reduction. This creates several problems. First, all calculations have to stop and wait until every process group has finished its computation. This effect can be reduced by efficient load balancing but it will never completely solve this issue due to the large amount of computation per component grid which can usually not be distributed perfectly to the process groups. Second, we have to wait until the communication has finished before each rank can proceed with the calculation. This issue is even more relevant for an HPC framework that targets exascale. The reason for that is that with larger process counts the communication time typically increases. Furthermore, the gap between the communication and computation speed has steadily grown over the past decades. A future-proof algorithm should therefore avoid any synchronous global communication.

To solve these issues, we have created a first algorithmic variant of the Combination Technique that offers asynchronous communication. This is achieved by delaying the application of the combination results. More precisely, we continue computing the PDE of our component grids after an asynchronous combination is started. Once we are at the next combination step, we check if the past combination has finished and calculate the changes it would have caused to the original component grid at the time of the last combination. We then apply these changes in a correction step to the current state of the grid. This is of course not exact and will introduce additional errors. This sections extends the preliminary work that was made in a collaboration with Shreyas Shenoy during his Master's thesis [110].

In the following, we will first describe the algorithmic idea in more detail and will then show some mathematical motivation for the approach.

3.5.1 Algorithmic idea

For the description of the algorithmic idea we need some definitions. First, we will consider the solver to be an iterative method $\Psi(u_{\boldsymbol{\ell}}^t) = \tilde{u}^{t+1}$ that updates the grid values u^t at the current combination step t and returns the updated grid values for the next combination step $t+1$. In addition, we perform the combination routine κ that generates the new input for the next solver computation based on the grid values of all component grids, i.e. $\kappa(\{\tilde{u}_{\boldsymbol{\ell}}^t | \boldsymbol{\ell} \in \mathcal{I}\}, \boldsymbol{\ell}) = u_{\boldsymbol{\ell}}^t$ at combination step t. This includes already the projection of the grid values from the Sparse Grid to the respective full grid $\boldsymbol{\ell}$. Consequently, the update that is applied to the grid values due to the combination at combination step t is $\Delta_{u_{\boldsymbol{\ell}}^t} = u_{\boldsymbol{\ell}}^t - \tilde{u}_{\boldsymbol{\ell}}^t$.

Algorithm 4 The Combination Technique in Parallel

procedure ASYNCHRONOUS_COMBI($G, \mathcal{I}_1, \ldots, \mathcal{I}_G, \tilde{u}^1$, combisteps)
  for $t \in$ [combisteps] **do**
    for $g \in [G]$ **do in parallel**
      for $\boldsymbol{\ell} \in \mathcal{I}_g$ **do**
        $\tilde{u}_{\boldsymbol{\ell}}^{h,t} = \mathcal{H}(\tilde{u}_{\boldsymbol{\ell}}^{t})$ ▷ local hierarchization
      end for
      $u^{c,g,t} = \sum_{\ell \in \mathcal{I}_g} c_\ell \cdot u_{\boldsymbol{\ell}}^{h,t}$ ▷ local combination
    end for
    $u^{c,t} = \sum_{g=1}^{G} u^{c,g,t}$ ▷ asynchronous global combination
    if $t \neq$ combisteps **then** ▷ continue without waiting for combination result
      for $g \in [G]$ **do in parallel**
        for $\boldsymbol{\ell} \in \mathcal{I}_g$ **do**
          $\tilde{\boldsymbol{v}}_{\boldsymbol{\ell}}^{t+1} = \Psi(\tilde{u}_{\boldsymbol{\ell}}^{t})$ ▷ apply solver until next combination step
        end for
      end for
    end if
    WAIT() ▷ wait for combination result $u^{c,t}$
    for $g \in [G]$ **do in parallel**
      for $\ell \in \mathcal{I}_g$ **do**
        $u_{\boldsymbol{\ell}}^{h,t} = \mathcal{P}_{\boldsymbol{\ell}}(u^{c,t})$ ▷ project back onto component grids
        $u_{\boldsymbol{\ell}}^{t} = \mathcal{D}(u_{\boldsymbol{\ell}}^{h,t})$ ▷ local dehierarchization
        $\Delta_{u_{\boldsymbol{\ell}}^{t}} = u_{\boldsymbol{\ell}}^{t} - \tilde{u}_{\boldsymbol{\ell}}^{t}$ ▷ compute delta
        $\tilde{u}^{t+1} \approx \tilde{\boldsymbol{v}}_{\boldsymbol{\ell}}^{t+1} + \Delta_{u_{\boldsymbol{\ell}}^{t}}$ ▷ apply delta
      end for
    end for
  end for
end procedure

With these definitions, we can now define the algorithm (see Algorithm 4). Whenever we reach a combination step t, we start asynchronously the combination κ with the solution $\tilde{u}^{t+1}$. Then instead of waiting for the results we continue the computation and compute $\Psi(\tilde{u}_{\boldsymbol{\ell}}^{t}) = \tilde{\boldsymbol{v}}^{t+1}$. We then wait for the combination results $u_{\boldsymbol{\ell}}^{t}$. Now, we can compute the difference value $\Delta_{u_{\boldsymbol{\ell}}^{t}} = u_{\boldsymbol{\ell}}^{t} - \tilde{u}_{\boldsymbol{\ell}}^{t}$ and apply this delta to the solution to approximate the correct solution, i.e. $\tilde{\boldsymbol{v}}^{t+1} + \Delta_{u_{\boldsymbol{\ell}}^{t}} \approx \tilde{u}^{t+1} = \Psi(u_{\boldsymbol{\ell}}^{t})$. This approximated value is then used for the next asynchronous combination.

The key differences of the asynchronous approach is that we can now overlap the communication with the computation of the next parallel solves. In general, we could extend this and compute as long as we need if the communication takes even longer. However, we will see next that the further we proceed the more the resulting approximation quality will suffer. Another difference is that we need to store the grid values $u_{\boldsymbol{\ell}}^{t}$, $\tilde{u}_{\boldsymbol{\ell}}^{t}$, and $\tilde{\boldsymbol{v}}^{t+1}$ instead of only a single copy of the grid. This theoretically triples

the memory consumption for the component grid but compared to the overall Sparse Grid representation $u^{c,t}$ – that each process group stores – this additional memory consumption is usually still small. The reason for that is that with multiple process groups the number of grids that each group owns decreases but the size of the Sparse Grid is similar to the size of all component grids together which stays constant.

3.5.2 Mathematical motivation

In the previous section, we described an asynchronous version of the Combination Technique. In this approach we proceed with the solver step during the communication to overlap communication with computation. After the communication has finished, the combined values are applied to the computation in an correction step that approximates the results of the standard Combination Technique.

The question is now what error this approximation introduces. We can analyze this with the Taylor series. Instead of the correct input to our solver $u_{\boldsymbol{\ell}}^t$ we used $\tilde{u}_{\boldsymbol{\ell}}^t$ with $\Delta_{u_{\boldsymbol{\ell}}^t} = u_{\boldsymbol{\ell}}^t - \tilde{u}_{\boldsymbol{\ell}}^t$. We can now look at the series expansion for the desired value $\Psi(u_{\boldsymbol{\ell}}^t) = \Psi(\tilde{u}_{\boldsymbol{\ell}}^t + \Delta_{u_{\boldsymbol{\ell}}^t})$ which results in the first order approximation

$$\Psi(u_{\boldsymbol{\ell}}^t) = \Psi(\tilde{u}_{\boldsymbol{\ell}}^t + \Delta_{u_{\boldsymbol{\ell}}^t}) \approx \Psi(\tilde{u}_{\boldsymbol{\ell}}^t) + J_{\Psi(u^t)}\Delta_{u_{\boldsymbol{\ell}}^t}. \tag{3.11}$$

The approach shown in the last section uses the rather strong approximation that the Jacobian matrix $J_{\Psi(u^t)}$ of u^t is the identity matrix, i.e. $J_{\Psi(u^t)} = I$. In this case, the equation is simplified to

$$\Psi(u_{\boldsymbol{\ell}}^t) = \Psi(\tilde{u}_{\boldsymbol{\ell}}^t + \Delta_{u_{\boldsymbol{\ell}}^t}) \approx \Psi(\tilde{u}_{\boldsymbol{\ell}}^t) + \Delta_{u_{\boldsymbol{\ell}}^t} = \tilde{v}^{t+1} + \Delta_{u_{\boldsymbol{\ell}}^t}. \tag{3.12}$$

This is exactly the formula that we apply in our algorithm. It should be noted that the remainder term from the truncated Taylor expansion scales quadratically with $\Delta_{u_{\boldsymbol{\ell}}^t}$.

From this mathematical analysis it follows that the approximation is better the smaller $\Delta_{u_{\boldsymbol{\ell}}^t}$ and the better the Jacobian matrix is approximated by the identity. The former is true if the component grids are close together and do not diverge from each other, which is also often a requirement for the recombination approach to ensure convergence. The approximation quality of the identity matrix for the Jacobian is depending on the time step and the PDE itself.

As an example for this approximation quality, we can look at the 1D advection equation

$$\delta_t u + v_x \delta_x u = 0 \tag{3.13}$$

with velocity v_x. This equation can be discretized e.g. via the finite-difference scheme using a first-order backward-difference to

$$\begin{aligned} 0 &= (\tilde{u}^{t+1}(x) - u_i^t)/\Delta_t + v_x(u^t(x) - u^t(x - \Delta_x))/\Delta_x \\ \Psi(u^t)(x) = \tilde{u}^{t+1}(x) &= u^t(x) - \Delta_t v_x(u^t(x) - u^t(x - \Delta_x))/\Delta_x \end{aligned} \tag{3.14}$$

with time step size Δ_t and spatial discretization width Δ_x. This computation is equivalent to one step of the solver $\Psi(u)$ in case we recombine every time step. If we now look at the Jacobian J, we can see that it contains the diagonal values $J_{i,i} = 1 - v_x\Delta_t/\Delta_x, i \in [N]$ and the off-diagonal values $J_{i,i-1} = v_x\Delta_t/\Delta_x, i \in [2, N]$ and is elsewhere zero. Therefore, as long as $v_x\Delta_t/\Delta_x$ is small we can expect the Jacobian to be close to the identity. The concrete formulas will change depending on the PDE and the discretization scheme but we can get a general idea from this analysis. Whenever the changes between the combination steps on a component grid are expected to be small, then the Jacobian will be close to the identity and the approximation quality will be good. Moreover, smaller time steps are in general more favorable for the asynchronous method as they reduce these changes.

In simulation where we do not combine every time step but after a few time steps, the derivation above will change and more grid values will have an influence on a certain point. This will create additional non-zeros in J and will decrease the approximation quality if we use the identity. Depending on the concrete PDE and the combination interval, we could therefore try to find better approximations and use a more suited asynchronous step. For this purpose, we need to adjust the delta of our algorithm accordingly, i.e. adjust our approximation of $\tilde{J}$ in $\Psi(\tilde{u}^t_{\boldsymbol{\ell}} + \Delta_{u^t_{\boldsymbol{\ell}}}) \approx \Psi(\tilde{u}^t_{\boldsymbol{\ell}}) + \tilde{J}_{\Psi(u^t)}\Delta_{u^t_{\boldsymbol{\ell}}}$. One thing that should be considered here is that the more carefully we mimic the original function the more costly it usually gets. In the worst-case, we can just reapply the solver to the combined value and discard $v^{t+1}_{\boldsymbol{\ell}} = \Psi(\boldsymbol{\ell}^t)$ which would not lead to any performance gain. Hence, we should go midway and construct a rather cheap approximation with higher quality if the identity matrix does not deliver reasonable results.

For the special case of linear PDEs, we can always formulate the solver step as $u^{t+1} = Au^t$. For k time steps in between combinations, this would result in $\Psi(u^t) = u^{t+k} = A^k u^t$. Hence, the Jacobian would be $J_{\Psi(u^t)} = A^k$. Here we can directly see that the better we approximate the Jacobian, the closer we move towards the original solution. The real Jacobian in this case would return the exact solution $\Psi(\tilde{u}^t_{\boldsymbol{\ell}} + \Delta_{u^t_{\boldsymbol{\ell}}}) = A^k(\tilde{u}^t_{\boldsymbol{\ell}} + \Delta_{u^t_{\boldsymbol{\ell}}}) = A^k(\tilde{u}^t_{\boldsymbol{\ell}}) + A^k(\Delta_{u^t_{\boldsymbol{\ell}}}) = \Psi(\tilde{u}^t_{\boldsymbol{\ell}}) + J_{\Psi(u^t)}\Delta_{u^t_{\boldsymbol{\ell}}}$.

This approach shows also similarities to the well-known predictor-corrector methods. We can see $\tilde{v}^{t+1}_{\boldsymbol{\ell}}$ as an predictor that is corrected by $\tilde{J}\Delta_{u^t_{\boldsymbol{\ell}}}$. In theory, also more advanced approaches could be used here which compute a cheap correction step.

Instead of approximating J, we could also approximate $\Psi(\Delta_{u^t_{\boldsymbol{\ell}}})$. This idea would give identical results for linear PDEs since here it holds that $J\tilde{u}^t_{\boldsymbol{\ell}} + J\Delta_{u^t_{\boldsymbol{\ell}}} = \Psi(\tilde{u}^t_{\boldsymbol{\ell}}) + \Psi(\Delta_{u^t_{\boldsymbol{\ell}}}) = \tilde{v}^{t+1}_{\boldsymbol{\ell}} + \Psi(\Delta_{u^t_{\boldsymbol{\ell}}}) = \Psi(u^t_{\boldsymbol{\ell}})$. For non-linear PDEs it could however change the approximation quality.

To put it in a nutshell, we have seen that there is some mathematical reasoning behind the asynchronous scheme. In general, it can be shown that for small component grid changes in between the combination steps and with small corrections from the combined result the algorithm should deliver a good approximation. However, the asynchronous method might also increase the errors significantly if this is not fulfilled. In Section 3.6.3 we show results for some practical examples.

3.6 Numerical experiments

So far, we have discussed improvements from the algorithmic and the implementation perspective of the Combination Technique in DisCoTec. In this section, we will demonstrate the effectiveness of these novel techniques with numerical simulations. For this purpose, we will first introduce in Section 3.6.1 our main application code GENE from plasma physics that simulates microinstabilities in a fusion reactor. These five-dimensional simulations are computationally involved and scientifically relevant which makes them a perfect use case for DisCoTec. Therefore, we test all our novel approaches with GENE. In Section 3.6.2, we then show the results for the new fault-tolerant implementation with realistic fault scenarios. Here, we analyze the introduced errors as well as the scaling properties of the FTCT. Next, we discuss in Section 3.6.3 the results for the asynchronous variant of the Combination Technique. This involves test cases with a simplified advection equation and first results with GENE simulations. Thereafter, we present in Section 3.6.4 results for global and non-linear plasma physics simulations that represent the most challenging simulation mode in GENE. Finally, we summarize our findings shortly in Section 3.6.5.

3.6.1 Application to plasma physics

In general, there are three states of matter for materials on earth: solid, liquid or gaseous. The state in which we find any substance is usually determined by its temperature and the pressure that acts on it. With increasing temperatures and decreasing pressure any substance transforms from solid to liquid and then to a gaseous state. If we continue this process, we will reach at some point a state where the electrons are stripped off the atoms resulting in positively charged ions and electrons. This state is called plasma and sometimes referred to as the fourth state of matter.

Plasma physics studies this state in more detail and tries to find methods to utilize the potential of the plasma for generating new and green energy sources. The reasoning behind this is that the sun generates most of its energy from nuclear fusion, which can be performed in a plasma state. Due to the immense energy that is generated with nuclear fusion, we could overcome our energy problems easily. At the same time nuclear fusion offers a possible clean and CO_2-free energy source. It is therefore interesting for achieving the climate goals of the upcoming decades. Unfortunately, nuclear fusion is still far away from a production use case as scientist still face severe problems in maintaining the plasma and generating a positive energy output. We therefore need further research and scientific breakthroughs in the design and operation of fusion reactors to reach this energy goal. The following overview of plasma physics and the application code GENE is mainly based on the overview given in [69].

One of the main challenges of a fusion reactor is to reach the necessary temperatures for a plasma and achieve a confinement for a sufficiently long time to reach nuclear fusion. This can only be reached via strong magnetic fields that force the ions and electrons on gyration trajectories. The ions are deuterium and tritium, two naturally existing hydrogen isotopes, which are heated up to multiple hundred thousands de-

grees Kelvin to create helium and energy. Current fusion reactors are based on either a tokamak or a stellarator design. In tokamak reactors a torus design is used and particles circulate across it. To prevent particle drifts, the particles are not just flying in circles but also perform poloidal turns, i.e. they fly in spirals through the torus shape. This is reached by the combination of a toroidal magnetic field and the magnetic field that is produced by inducing a large toroidal current in the plasma. This induction forces a pulsed operation of a Tokamak. To avoid a pulsed operation, in the Stellarator design the twisted magnetic field is reached only by external magnets.

A problem for current fusion reactors are microinstabilities that transport particles and heat out of the plasma. To study these effects and analyze means to reduce these instabilities, numerical experiments are performed as real experiments are very costly or even impossible with current reactors. Thus, microinstabilities cannot be sufficiently studied through experiments. In this work, we will consider the HPC code GENE[5] [66, 42] to simulate these microinstabilities.

GENE uses a kinetic approach to simulate the plasma in both tokamaks or stellerators. In this approach not just the spatial coordinate $\boldsymbol{x}$ but also the velocity $\boldsymbol{v}$ is explicitly discretized which leads to a 6D full grid for the distribution function which can be viewed after normalization as a probability of a certain particle being at a certain position with a specific velocity. In particular, the 6D collision-free Vlasov equation can be used which is defined by

$$\frac{\delta g_s}{\delta t} + \boldsymbol{v} \cdot \frac{\delta g_s}{\delta \boldsymbol{x}} + q_s \left(\boldsymbol{E}(g) + \frac{\boldsymbol{v}}{c} \times \boldsymbol{B}(g) \right) \cdot \frac{\delta g_s}{\delta \boldsymbol{v}} = 0 \tag{3.15}$$

where s is the species with distribution function g_s and charge q_s. $\boldsymbol{E}(g)$ and $\boldsymbol{B}(g)$ are respectively the electric and magnetic field [69, section 2.2.3]. Since the fields are depending on the distribution function g, the Vlasov equation is non-linear.

Interestingly, microinstabilities happen on a much larger spatial and time scale than other effects simulated by the Vlasov equation such as gyration. To study these microinstabilities with the Vlasov equations directly, would therefore waste valuable resources or would even be infeasible. Therefore, GENE relies on a gyrokinetic approach where an averaging over a gyration of the particles leads to the reduction to a 2D velocity space. As a result, larger time scales can be studied and the simulation grid is reduced to five dimensions $x, y, z, v_{||}$ and μ as the velocity is now only discretized according the velocity parallel to the magnetic field lines $v_{||}$ and the magnetic moment μ. In addition, the spatial dimensions $\boldsymbol{r} = (x, y, z)$ are aligned with the magnetic field lines to enable a fast and efficient computation of the numerical results. The resulting gyrokinetic equation (see [69, section 2.3.1] or [24, section 2.1.3]) is

$$\frac{\delta g_s}{\delta t} + \dot{\boldsymbol{r}} \cdot \frac{\delta g_s}{\delta \boldsymbol{r}} + \dot{v_{||}} \cdot \frac{\delta g_s}{\delta v_{||}} + \dot{\mu} \cdot \frac{\delta g_s}{\delta \mu} = 0. \tag{3.16}$$

Furthermore, GENE splits the magnetic fields and distributions in a background and fluctuation part. This process is called δf splitting and allows to only simulate the fluctuation (see [69, section 2.3.1] or [24, section 2.1.3]).

[5] https://genecode.org/

In a simplified form the equation solved in GENE can be seen as

$$\frac{\delta g}{\delta t} = \mathcal{L}(g) + \mathcal{N}(g) \tag{3.17}$$

with a linear operator $\mathcal{L}(g)$ and non-linear operator $\mathcal{N}(g)$ [69, section 2.4.2]. Due to this separation, it is possible to run GENE in two basic modes: linear and non-linear. In the linear mode the non-linear operator is neglected. Linear simulations can be useful for eigenvalue studies or preliminary simulations. The non-linear simulations represent the core functionality and are the most interesting simulations from a physical point. However, non-linear simulations introduce significantly larger computational costs.

Another important mode in GENE is the *local* and *global* (or *non-local*) mode for the x dimension [69, section 2.4.2]. In the local mode only a small subarea (flux-tube) of the x dimension is calculated and a periodic boundary is introduced. This allows for a Fourier transform along this dimension for *local* simulations. Another Fourier transform is applied to the y dimension in both modes which utilizes the periodic boundaries along the y axis.

In GENE, the 5D full grid with dimensions x,y,z,$v_{||}$, and μ is solved for each species s [24, section 3.1.1]. As the gyrokinetic equation is time-dependent, we need to solve the equation for multiple time steps. In our test cases, only the Runge-Kutta 4 scheme is used, but also other time stepping schemes are available in GENE [69, section 2.4.2]. The code is designed for HPC and offers a grid-based domain decomposition which fits well to the parallelization scheme of DisCoTec. The only issue is that full grids in DisCoTec require $2^{\ell_k} + 1$ points in dimension k, but GENE requires that each processor in the grid has the same number of coordinates in each dimension. Since $2^{\ell_k} + 1$ is typically hard to split into equal parts and is in fact often even a prime number, we instead only compute 2^{ℓ_k} points in each dimension and fill the missing point with a boundary condition. More information on this process can be found in [69, chapter 4].

The coupling of GENE to DisCoTec requires an adapter which was written in the context of the dissertation of Mario Heene [51] and Alfredo Parra [89]. This adapter mainly exchanges grid values between GENE and DisCoTec and adds the missing points according to the boundary conditions. The framework was also extended to handle complex numbers which arise due to the use of Fourier transforms in GENE. See more details in [51] for the treatment of complex numbers in DisCoTec. This GENE adapter was extended in this dissertation to also support non-linear problems. In addition, the fault tolerance mechanisms (see Section 3.2) were added to be able to test the resilience of DisCoTec with a real application scenario.

For a more detailed overview of GENE including further simulation modes, we refer to [66, 42, 25, 24, 69] .

3.6.2 Fault tolerance

In this section, we report the results with our implementation of the FTCT. These results were also published in [86]. For all our test cases with fault tolerance, we used the linear and local configuration of GENE with the parameters file listed in Appendix B.1.

Test case	A	B
$\boldsymbol{\ell}^{\max} = (\ell_x, \ell_y, \ell_z, \ell_{v_\|\|}, \ell_\mu)$	(3, 1, 7, 7, 7)	(3, 1, 8, 8, 8)
$\boldsymbol{\ell}^{\min} = (\ell_x, \ell_y, \ell_z, \ell_{v_\|\|}, \ell_\mu)$	(3, 1, 5, 5, 5)	(3, 1, 6, 6, 6)
steps	6000	6000
timestep Δt (= combination interval)	0.005	0.005
Weibull shape k	0.7	0.7
parallelization $(n_x, n_y, n_z, n_{v_\|\|}, n_\mu)$	(1, 1, 1, 8, 16)	(1, 1, 1, 8, 16)
process groups	4	4

Table 3.1: Parameters for the two test cases A and B.

This means that we do not fully simulate the x domain but only a subarea of it. In addition the non-linear part of the PDE is neglected. As a result, the GENE simulation resembles an iterative solver as the exponential increase due to the dominating eigenvalue converges over time if the distribution is normalized. As part of this thesis, we analyzed for this setup the numerical convergence and the scaling for large scale HPC applications. All tests presented in this section were performed on the Hazel Hen cluster (see Appendix A.1).

3.6.2.1 Numerical error analysis

For the numerical error analysis we will look at two test cases (see Table 3.1). In both test cases we use a 3D combination scheme where only the z, v and w dimension are combined. The x dimension is set constant to 9 points with level 3, the y dimension to 1 point with level 1 and only a single specie is simulated. The reason for the constant dimensions in x and y are that 9 points are necessary but also sufficient in x for local simulations and the y frequencies are independent of each other and are therefore simulated individually for local and linear runs. We combine after every time step and use uniform time steps (see Section 3.3.2). The test cases differ only in their minimum and maximum levels. Both schemes result in 10 component grids that are split up between 4 process groups. In each process group we use a process grid with 8 ranks in $v_\|\|$ and 16 in μ direction resulting in 128 ranks per group. In total this sums up to 512 ranks plus one manager rank. The additional GENE parameters that we used can be found in Appendix B.1.

For these simulations we want to measure the additional error due to the application of the FTCT. We therefore inject faults according to the Weibull distribution (see Section 3.2.2). Here, we fixed the shape parameter k to 0.7 and varied the scale parameter λ between 10^5 and 10^7. For each λ value, 30 test cases were simulated to capture the statistical effects and the distribution of the introduced errors. The manager checks before each combination step if a fault has occurred and enables a recovery if necessary. As a result, we observed up to 117 fault events. Here, we measure the number of iterations

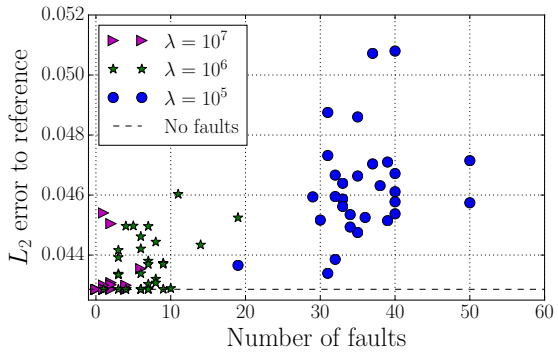


Figure 3.7: L_2 error of the FTCT depending on the number of faults for test case A.

in which a fault occurs. In case multiple ranks fail in the same iteration, this is counted as a single fault event.

For the error analysis, we use the L_2 error ϵ between the absolute values of a fault-free reference full grid of level $\ell^{\text{ref}} = (3, 1, 8, 8, 8)$ and the respective combination solution $u^c_{\boldsymbol{\ell}^{\max},\boldsymbol{\ell}^{\min}}(\boldsymbol{x})$ of the simulation with faults after the 6000 time steps. In particular, both solutions are interpolated at the grid points of the reference grid for the comparison. This can be formalized via

$$\epsilon = \sqrt{\sum_{\boldsymbol{i}\in\mathcal{I}_{\ell^{\text{ref}}}} \left(|u_\ell(x_{\boldsymbol{\ell}^{\text{ref}},\boldsymbol{i}})| - |u^c_{\boldsymbol{\ell}^{\max},\boldsymbol{\ell}^{\min}}(x_{\boldsymbol{\ell}^{\text{ref}},\boldsymbol{i}})|\right)^2}. \tag{3.18}$$

As suggested by [51, chapter 5.2.2], we use the absolute values of the function u since the complex nature of the GENE values would generate large errors for values with phase shift. However, phase shifts are irrelevant for the physical interpretation in this scenario and they are therefore not considered in the error calculation. In addition, the absolute values of the values are not relevant and are frequently rescaled during the simulation. Thus, we normalize each vector output of u before applying the vector norm in Eq. (3.18).

The resulting error values can be seen in Figs. 3.7 and 3.8. It is easy to see that in general there is a trend that the more faults occur the larger the error is compared to a non-faulty result. At the same time, the more faults we observe the more the error values fluctuate. The reason for this is that the final outcome is not just determined by the number of faults but also by the time the faults occur. In Fig. 3.9 we show exemplary for test case B with high λ how the timing of the last fault affects the final error. There is a clear dependency that the later an error occurs the more distorted is the result. An explanation for this is that the whole simulation can be seen as an iterative process that converges towards the solution. If an error occurs at a later point, it affects the final solution more strongly. Early distortions due to faults can be smoothed out by the iterative process. For small λ values with a high number of faults this strong dependency of the last failing iteration was not observed which is probably related to

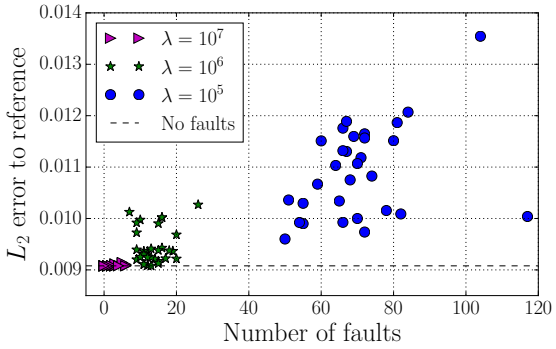


Figure 3.8: L_2 error of the FTCT depending on the number of faults for test case B.

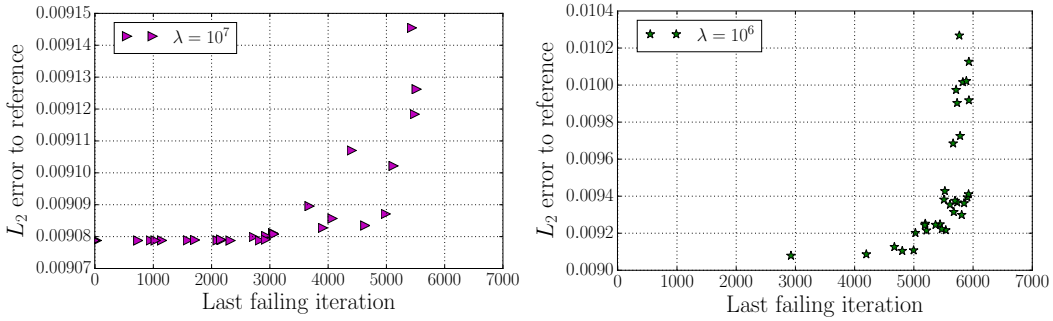


Figure 3.9: L_2-error of the FTCT for different runs of test case B compared to the last iteration with process error for $\lambda = 10^6$ (bottom) and $\lambda = 10^7$ (top) .

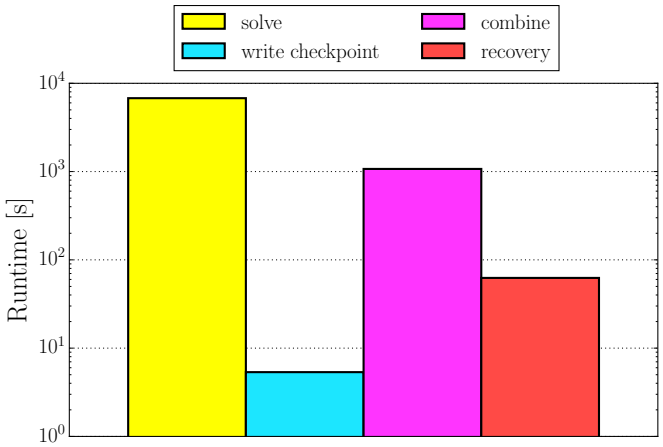


Figure 3.10: Runtimes for the most expensive steps of the FTCT. We plot the maximum time a process spends in total for each step.

the fact that with such a high number of faults there is with high probability a fault

λ	f_{avg}	f_{max}	f_{min}	e_{avg}	e_{max}	e_{min}	Δe (%)
10^7	1.43	6	0	0.0431	0.0454	0.0429	0.480%
10^6	6.75	19	3	0.0437	0.0460	0.0429	1.91%
10^5	35.5	50	19	0.0463	0.0508	0.04339	7.955%

Table 3.2: table
Statistical results of the error of the FTCT for different $\boldsymbol{\lambda}$ in test case A. $\boldsymbol{f}$ represents the number of faults, $\boldsymbol{e}$ the $\boldsymbol{L_2}$ error to the reference and $\boldsymbol{\Delta e}$ the average increase in the error compared to a simulation without faults.

λ	f_{avg}	f_{max}	f_{min}	e_{avg}	e_{max}	e_{min}	Δe (%)
10^7	2.55	6	0	0.00908	0.00915	0.00908	0.0882%
10^6	13.5	26	7	0.00945	0.0103	0.00908	4.09%
10^5	70.3	117	50	0.0109	0.0135	0.00960	20.2%

Table 3.3: table
Statistical results of the error of the FTCT for different $\boldsymbol{\lambda}$ in test case B.

close to the end of the simulation. Hence, most of the simulations have faults at the end.

Another observation is that in general the magnitude of the errors compared to a non-faulty simulation (dashed-line) is rather small even for very large fault numbers. In Tables 3.2 and 3.3, we show a more detailed statistical evaluation for both test cases. Here we see for each variant of λ the average, maximum and minimum number of faults and errors. In addition we added the average increase in the error compared to the base line of the non-faulty simulation. Here, we can clearly see that for both test cases with low fault numbers the error increase is negligible. For large number of faults the errors increase noticeable but considering that here up to, respectively, 10% or 20% of all ranks fail, the average increase is still pretty low with values up to 20.2%. It should be noted that this is the increase in the error and not in the function values and that the errors are in the range of $10^{-2} - 5 \cdot 10^{-2}$. Moreover, a fault has the effect that up to a third of all component grids are incomplete and have to be removed from the respective combination. Considering this, the error increases are very low. The results for test case B are in general more accurate as higher discretization levels are used. It should be noted that in practice the number of faults will be rather small and therefore the error increase will be insignificant for this application.

If we look exemplary at the compute time of the different parts of the algorithm for test case B in Fig. 3.10, we see that the simulation time for this setup is dominated by the computation withing the solver. Here, the combine step takes around a sixth of the time to solve the PDE and the recovery step is even two magnitudes faster. We also plotted the time for the generation of a single checkpoint file which is about five times larger than a single computation step. This means that if we would checkpoint every

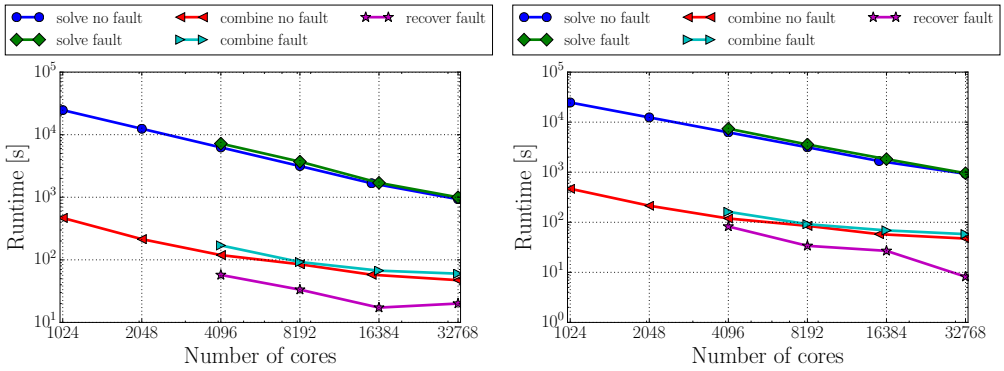


Figure 3.11: Strong scaling results for the different steps of the FTCT. On the left we show the runtime for a single run and on the right the average of three runs.

iteration, we would have a recovery time which whould be around 500 times larger and which would dominate the complete runtime. Our method is therefore much cheaper.

To conclude this section, we can say that our implementation of the FTCT can tolerate massively faulty system where large fractions of the whole system fail. Furthermore, the computational overhead is almost negligible at around 1% and the introduced errors are in practice insignificant.

3.6.2.2 Scaling results

In addition to the convergence tests, we tested our implementation of the FTCT with a large scale setup. Here, we used the same linear GENE test case as before (see Appendix B.1) but increased the problem size. For that we used $\boldsymbol{\ell}^{\min} = (9, 1, 4, 4, 4)$ and $\boldsymbol{\ell}^{\max} = (9, 1, 13, 13, 13)$ for a 3D combination scheme with 185 combination grids. Since we are only interested in the runtimes of the different components, we fixed the number of time steps to 300. The process group size was fixed to 1024 and the number of process groups was doubled until we reached 32k cores. For this setting we simulated both fault-free (as a base-line) and faulty scenarios. For the faulty scenarios only a single fault was injected in one of the process groups. For the fault-tolerant execution we start at 4k cores and the fault-free baseline starts at 1k cores. This difference for the FTCT is a result of a slightly higher memory footprint and the requirement for having multiple process groups as always one group fails.

In Fig. 3.11 we show the scaling results. We compare the times for the solve, the combination and the recovery between the faulty and fault-free version. We can clearly see that the solving time scales very similarly for the faulty execution compared to the fault-free version. We observed only slight increases in the runtime. This increase is caused by the loss of a process group after a failure is detected. Since for larger process counts the loss of a single group is less relevant, the overhead decreases with large core counts. Another source for the overhead is that we need to calculate additional very coarse component grids for the FTCT (see Section 2.3.4). However, this overhead

seems to be very small due to the very similar runtimes for large core counts. Those observations are very promising as the main target of our framework is exascale and therefore the number of process groups is expected to be very large.

Another important part of our algorithm is the combination step. For this routine the observations are similar. We see slight increases in the runtime with the faulty version but similar scaling. In this case, the increase comes from the loss of a process group and in addition from the fact that we have to communicate all subspaces in the FTCT and cannot avoid communicating the subspaces that are specific to a single component grid. The reason for that is that we need to be able to have a valid checkpoint for these subspaces in case we need to recompute a grid. This checkpoint should be stored on all process groups to avoid communication during recovery. Therefore, the complete subspace information needs to be communicated between all groups during the combination step.

For runs with a failing rank, we need to find a fault-free combination scheme, recompute some cheap grids if necessary and restore the communicators. This is done in the recovery step. We can see that the recovery time can fluctuate for specific cases. A reason for that is that the optimization routine for finding a fault-free combination scheme produces different results in different scenarios and the load balancing varies due to its dynamic and non-deterministic nature. Especially the number of grids that need to be recomputed can significantly influence the recovery time. However, if we look at the average runtime of three independent runs, the recovery times seem to scale well, which is mainly caused by the fact that with more process groups less of the 185 grids fail. Hence, recomputation costs decrease and it becomes more easy to find a valid combination scheme. Overall, the recovery time is about two to three orders of magnitude lower than the solving time which makes it negligible for the complete runtime of the simulations. Furthermore, real scenarios would run for far more time steps and have usually only very few failure events.

If we look at concrete numbers, we obtained a parallel efficiency of 93.61% compared to the runtime at 4k processor with faults or 76.97% if we compare it to the fault-free run with 1k processors. The solving overhead at 32k cores is 10.93% and the overhead for the combination time 21.37%. It should be noted again that the impact on the runtime should be far less in realistic scenarios where failure events are rare and usually up to millions of time steps are computed. Therefore, the overhead should further decrease and the parallel efficiency should be even higher.

3.6.3 Asynchronous combination

In this section, we summarize our results with the asynchronous Combination Technique (see Section 3.5). Here, we investigated the convergence and additional errors that are introduced by the method due to the approximation in the correction step. We compare all results to the standard Combination Technique that we will refer to as synchronous Combination Technique. We also discuss different variants how to approximate the Jacobian such as the identity matrix and a diagonal approximation. First, we

will look at an advection problem in Section 3.6.3.1 and will then analyze the method in Section 3.6.3.2 for GENE.

3.6.3.1 Advection equation

As a first test case for the asynchronous Combination Technique, we consider an easy toy problem of a 2D advection problem

$$\delta_t u + v_x \delta_x u + v_y \delta_y u = 0 \tag{3.19}$$

with periodic boundary conditions and initial values

$$u(t=0,x,y) = u^0(x,y) = e^{-100((x-0.5)^2+(y-0.5)^2)} \tag{3.20}$$

on the unit square domain $D = [0,1]^2$. We set $v_x = v_y = 1$ and discretized the simulation via backward differences which results in

$$u^{t+1}(x,y) = u^t(x,y) - \Delta_t \left(v_x \frac{u^t(x,y) - u^t(x-\Delta_x,y)}{\Delta_x} + v_y \frac{u^t(x,y) - u^t(x,y-\Delta_y)}{\Delta_y} \right).$$

For this PDE the diagonal entries of the Jacobian are represented by $J_{i,i} = 1 - \Delta_t v_x/\Delta_x - \Delta_t v_y/\Delta_y$ which are identical for all values of i. For the asynchronous algorithm, we investigated two different approximations for the Jacobian matrix: the identity matrix $\tilde{J} = I$ and $\tilde{J} = \text{diag}(J) = I \cdot J_{0,0}$. In cases where we compute not just one but n time steps in between combinations, we use for the latter case the approximation $\tilde{J} = I \cdot J_{0,0}^n$, i.e. we take the matrix from a single time step and raise it to the power of n.

We compare the results of the synchronous and the asynchronous Combination Technique to the analytic solution

$$u(t=0,x,y) = u^0(x,y) = e^{-100((\tilde{x}-0.5)^2+(\tilde{y}-0.5)^2)} \tag{3.21}$$

with $\tilde{x} = (x - t \cdot \Delta_t) \mod 1$ and $\tilde{y} = (y - t \cdot \Delta_t) \mod 1$. Here, we apply the modulo operation for real numbers to mimic the periodic flow.

With this example we can now easily compare the different convergence of the synchronous and the asynchronous combination with identity matrix (async identity) and the diagonal Jacobian approximation (async diag). We simulated 100 steps with $\Delta_t = 10^{-4}$ and varied the target level. In Fig. 3.12 we show the L2 error of the different runs which is calculated by projecting the Sparse Grid solution to the target full grid of level $\boldsymbol{\ell} = (12,12)$ and computing the L_2 vector norm of the difference between the computed solution and the analytic values [6].

The first observation is that the identity approximation performs significantly better than the diagonal approximation of the Jacobian matrix. This is especially the case for larger values of $\ell^{\max}$. A reason for this is that the values $J_{0,0}$ get smaller the smaller Δ_x

[6] We also tried different norms such as the L_1 and L_∞ which showed similar results with no qualitative difference.

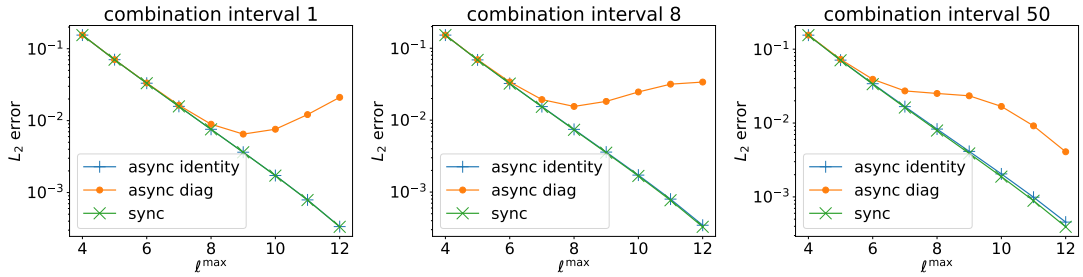


Figure 3.12: Asynchronous recombination errors for 100 steps with $\Delta_t = 10^{-4}$ of a 2D advection example with $\boldsymbol{\ell}_{min} = (3,3)$ and varying $\boldsymbol{\ell}^{\max} = (\ell^{\max}, \ell^{\max})$ for combination interval 1(left), 8 (middle) and 50 (right).

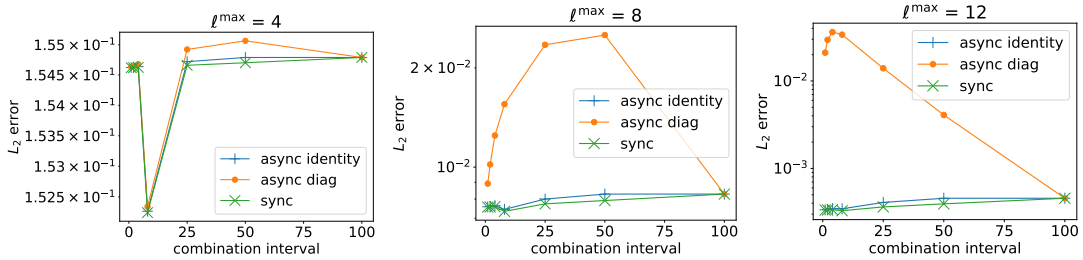


Figure 3.13: Asynchronous recombination errors for 100 steps with $\Delta_t = 10^{-4}$ of a 2D advection example for varying combination interval with $\boldsymbol{\ell}_{min} = (3,3)$ and different $\boldsymbol{\ell}^{\max} = (\ell^{\max}, \ell^{\max})$. We show the results for $\ell^{\max} = 4$ (left), $\ell^{\max} = 8$ (middle) and $\ell^{\max} = 12$ (right).

which is directly coupled to the level. As a consequence, the test cases with high levels damp the correction due to the combination. This apparently results in large error increases. This result is surprising as an diagonal approximation of the Jacobian seems to be better suited than the identity matrix. An explanation for this phenomenon could be that the sum of all entries of a row of the Jacobian sums up to 1, i.e. $\sum_{j=1}^{n} J_{i,j} = 1$. In addition, the nonzero entries for a combination interval of 1 are located at the positions of the backward neighbours of a grid point and if we compute several steps, the largest contributions come from backward neighbours and the close vicinity of the surrounding points. That means if points within a close range have similar grid values the identity approximation is rather good. Since the combination mainly affects lower subspaces – the largest subspaces are exclusive to single component grids and are therefore unchanged –, it is clear that the changes to $\Delta_{u_\ell^t}$ are of low frequency and should therefore not change too fast. At higher levels such changes should however be larger as the bigger number of different discretized grids tends to cause larger corrections. A reason for this is that the single grids diverge further from each other in such cases. This is also an explanation why the errors increase for larger $\ell^{\max}$.

Another observation is that for all combination intervals the errors of the asynchronous combination with the identity matrix are almost identical to the synchronous com-

bination with only slight error increases. Furthermore, this difference increases only slowly with the maximum level and the combination interval. Therefore, most of the time the asynchronous error and the synchronous error curve are not distinguishable. Only at a combination interval 50 and $\ell_{\max} = 12$ we can see a slight difference. This can also be seen if we directly compare different combination intervals with constant $\ell^{\max}$. This is shown in Fig. 3.13 for different maximum levels. Here, mainly for higher levels the errors increase for larger combination intervals. For $\ell^{\max} = 4$ the error mainly stays constant except for a small fluctuation at a combination interval of 8. This very small drop is unexpected and might just be related to a random error cancellation of the spatial and temporal error of the combination with the asynchronous approximation error. For combination interval 100 there is no difference as here only a single combination is performed and therefore the synchronous and asynchronous version behave exactly the same.

Our observations for the identity matrix match exactly the mathematical analysis of Section 3.5.2 where we have seen that the approximation quality of the Jacobian matrix is mainly influenced by the term $v_x \Delta_t / \Delta_x$ for a 1D advection equation. With increasing level $\ell^{\max}$, Δ_x gets smaller and therefore the error rises. Also for larger combination intervals the Jacobian further deviates from the identity matrix which further increases the error. As the advection example is a linear PDE, the approximation quality of the Jacobian is the only error source.

3.6.3.2 GENE

In addition to the very simple advection equation, we also performed several test cases with GENE. Similar to the fault tolerant test case, we used linear runs with the parameter file from Appendix B.1 and simulated 6000 time steps until the solution converged to the dominating eigenvalue. We again performed several runs with varying combination intervals – measured in number of timesteps – and used two different minimum and maximum levels. More information is listed in Table 3.4. Due to the superiority of the approximation quality with the identity matrix (see Section 3.6.3.1), we focused on this version of the asynchronous algorithm for the GENE runs. All tests in this section were performed on the SuperMUC-NG cluster (see Appendix A.2).

The results are shown in Fig. 3.14 where we calculated the errors in the same way as in Section 3.6.2. Here, we see that in general the errors rise with the combination interval. Furthermore, for some combination intervals the errors rise drastically which is a sign that the combination does not work and seems to become unstable. These situations usually happen in the mid-range of the combination intervals. We also observed that this range is larger for the asynchronous combination technique, i.e. it can be applied in fewer cases than the synchronous version. However, for stable simulations the errors of the synchronous and asynchronous combination are comparable.

From our numerical results we can follow that in cases where the Combination Technique is stable the asynchronous Combination Technique with an identity matrix approximation for the Jacobian performs very good with very little additional error. However, if the Combination Technique tends to become unstable for certain combination

Test case	A	B
$\boldsymbol{\ell}^{\max} = (\ell_x, \ell_y, \ell_z, \ell_{v_{\|\|}}, \ell_\mu)$	(3, 1, 8, 8, 8)	(3, 1, 8, 8, 8)
$\boldsymbol{\ell}^{\min} = (\ell_x, \ell_y, \ell_z, \ell_{v_{\|\|}}, \ell_\mu)$	(3, 1, 5, 5, 5)	(3, 1, 6, 6, 6)
steps	6000	6000
timestep Δt	0.005	0.005
parallelization $(n_x, n_y, n_z, n_{v_{\|\|}}, n_\mu)$	(1, 1, 1, 8, 16)	(1, 1, 1, 8, 16)
process groups	4	4

Table 3.4: Parameters for the two GENE test cases A and B with the asynchronous Combination Technique.

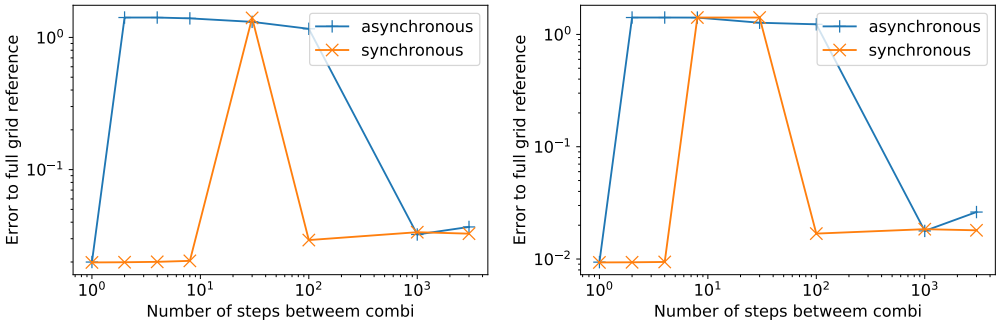


Figure 3.14: Asynchronous recombination errors with GENE for the test cases (see Table 3.4) A (left) and B (right). Here we vary the combination intervals and compare the results to the full grid reference with $\boldsymbol{\ell} = (3, 1, 8, 8, 8)$.

intervals, the asynchronous mode can reduce this stability region due to the additional errors that are introduced. A thorough analysis of the PDE at hand is therefore required to check whether the asynchronous Combination Technique remains stable.

3.6.4 Non-linear Plasma runs

In the previous sections, we only used linear GENE simulations which neglect the non-linear part $\mathcal{N}(g)$ of the PDE

$$\frac{\delta g}{\delta t} = \mathcal{L}(g) + \mathcal{N}(g). \tag{3.22}$$

As a second step of the EXAHD project, we investigated the non-linear and global runs of GENE to test our implementation of the Combination Technique. For these cases we used the parameters that are listed in Appendix B.2. In all of the runs we make use of individual time steps which means that there is not a fixed number of steps but a fixed

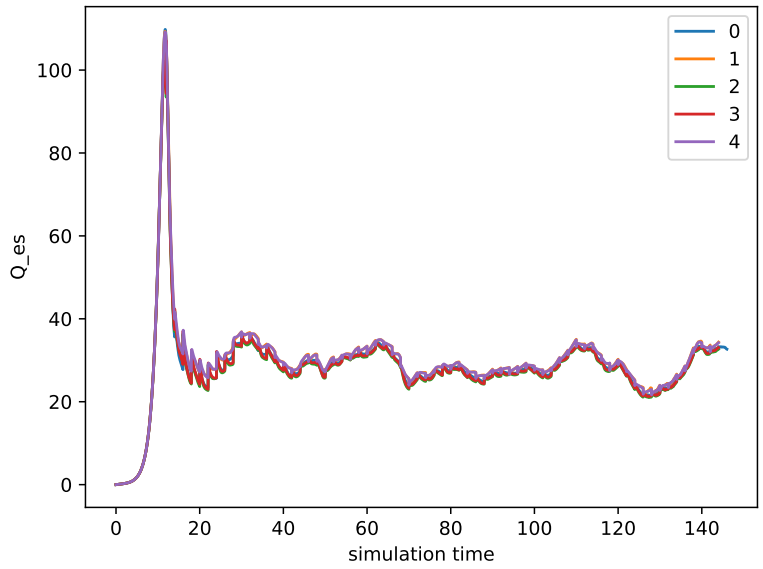


Figure 3.15: Non-linear recombination run with $\ell_{min} = (8, 5, 3, 4, 3)$ and $\ell_{max} = (10, 5, 5, 4, 3)$ resulting in 5 component grids. The selected recombination interval is 2.

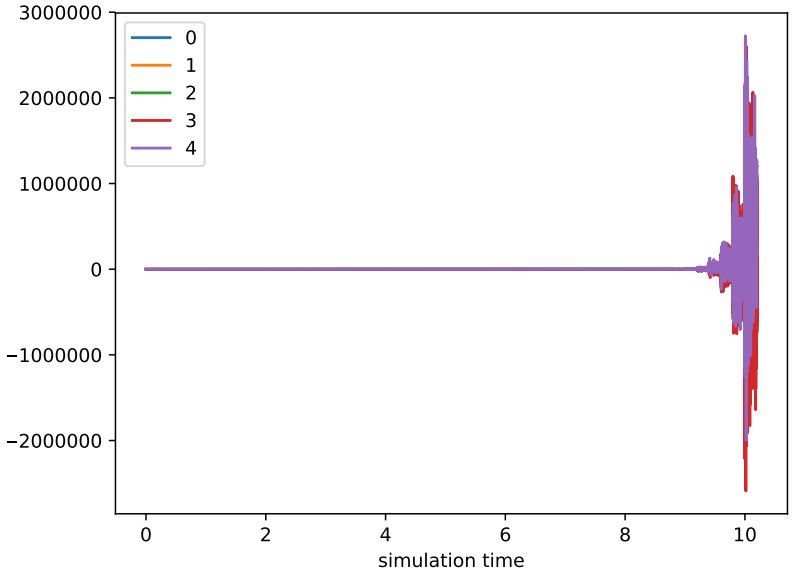


Figure 3.16: An unstable non-linear recombination run with $\ell_{min} = (8, 5, 3, 4, 3)$ and $\ell_{max} = (10, 5, 5, 4, 3)$ resulting in 5 component grids. The selected recombination interval is 0.2.

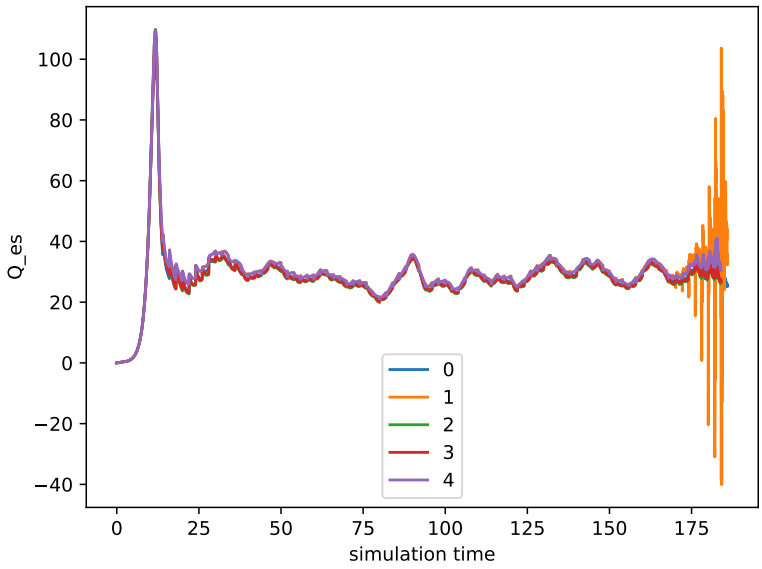


Figure 3.17: A very long non-linear recombination run that is unstable. Here we use $\ell_{min} = (8, 5, 3, 4, 3)$ and $\ell_{max} = (10, 5, 5, 4, 3)$ resulting in 5 component grids and a recombination interval of 2.

simulation time[7] between combinations for each component grid. As a consequence, the number of time steps is set very high so that a GENE run always simulates until the next combination step. If a time step would exceed this checkpoint time, it is adjusted accordingly and the simulation is stopped afterwards. In the following, we will summarize our findings. Additional information on the non-linear tests can be found in the final project report [72].

In our analysis, we will mainly consider the main quantity of interest Q_{es} and its time-average in the non-linear phase. Our evaluation of the time-average and the statistical error bounds follow the methods described in [114]. All tests in this section were performed on the SuperMUC-NG cluster (see Appendix A.2).

In general, we observed two different scenarios for the time-evolution of Q_{es}. On the one hand, for large combination intervals[8] of around 2 the simulation was stable and the time evolution looks as expected (Fig. 3.15). Here, we first observe the exponential phase where the value of Q_{es} grows exponentially. At a simulation time of around 20, we can see that the simulation enters a stationary phase with seemingly random fluctuations which are caused by the non-linearities.

Unfortunately, we faced severe instabilities for short combination intervals ≤ 0.2 (Fig. 3.16) and also for very long simulation periods with larger combination intervals

[7] This should not be confused with the runtime. The simulation time is the simulated time that passes in the GENE simulation.

[8] This interval results in approximately 7000 steps per component grid between combinations.

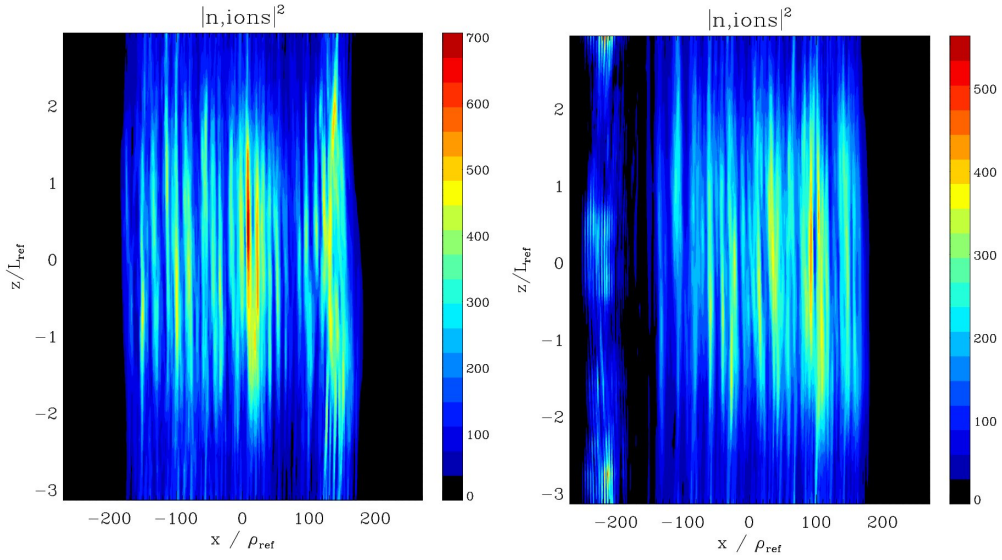


Figure 3.18: An x-z slice of an unstable non-linear recombination run with $\ell_{min} = (8, 5, 3, 4, 3)$ and $\ell_{max} = (10, 5, 5, 4, 3)$ and combination interval 2. On the left we show the initial situation in an stable state and on the right the situation when the instability kicks in.

(Fig. 3.17). Also simulations with more component grids showed these instabilities. In general, the more frequent we combine and the more grids we combine the less stable the simulation gets. This is a clear sign that the combination introduces significant errors or even violates the physical models. Up to now, it is not clear what exactly is the cause of this behavior. By investigating the instabilities for very long runs, we noticed that sometimes close to the boundary of the x-z slices large values arise which are unphysical. An example can be seen in Fig. 3.18. Here, we plot the state at the beginning and a snapshot at the time when the instability kicks in. We can clearly see that the values at around x=-200 start to increase and explode over time which causes the instability. Unfortunately, we do not know if this is a problem of the implementation in `GENE` or `DisCoTec` or if this is a generic problem with the Combination Technique.

By looking at other Combination Technique literature with non-linear PDEs [71], we could find similar problems with the Navier-Stokes equations. Here, similarly to our case instabilities appeared after long intervals. In this case, it could be solved by adapting the Combination Technique to be divergence-free. Sadly, this is not directly applicable to `GENE` as the code with δf splitting is not divergence-free. However, this indicates that our observations could be an inherent problem of the way we apply the Combination Technique and not an implementation problem.

Nevertheless, we can analyze the performance for the stable runs by combining the averaged Q_{es} value in the non-linear phase. This is one of the main quantities of interests for global, non-linear `GENE` runs. We therefore compared the combination results to the reference values of a fully resolved full grid with the target resolution $\boldsymbol{\ell}^{\max}$. In

description	$\boldsymbol{\ell}$	$c_{\boldsymbol{\ell}}$	avg. Q_{es} recombination	avg. Q_{es} independent
component grid 0	(9, 5, 4, 4, 3)	1	28.69 (+- 0.11)	23.75 (+- 0.59)
component grid 1	(8, 5, 5, 4, 3)	1	29.53 (+- 0.12)	30.89 (+-0.60)
component grid 2	(10, 5, 3, 4, 3)	1	28.18 (+- 1.00)	28.79 (+- 0.74)
component grid 3	(9, 5, 3, 4, 3)	-1	28.41 (+- 0.80)	29.96 (+- 0.57)
component grid 4	(8, 5, 4, 4, 3)	-1	29.60 (+- 0.66)	29.53 (+- 0.64)
combination	-	-	28.38	23.854990282
target run	(10,5,5,4,3)	-	-	27.32 (+- 0.61)

Table 3.5: Recombination in x and z with $\ell_{min} = (8, 5, 3, 4, 3)$ and $\ell_{max} = (10, 5, 5, 4, 3)$. We show results for the averaged Q_{es} values with recombination and for an independent simulation. In addition the statistical errors are shown. This is compared to the combined values and the result of an target level run with a full grid of $\boldsymbol{\ell} = \ell_{max}$.

description	$\boldsymbol{\ell}$	$c_{\boldsymbol{\ell}}$	avg. Q_{es} recombination	avg. Q_{es} independent
component grid 0	(9, 5, 5, 4, 3)	1	24.87 (+- 0.35)	26.35 (+- 0.81)
component grid 1	(10, 5, 4, 4, 3)	1	24.95 (+- 0.65)	27.35 (+- 0.82)
component grid 2	(11, 5, 3, 4, 3)	1	24.85(+- 0.89)	30.14 (+- 0.95)
component grid 3	(9, 5, 4, 4, 3)	-1	24.86(+- 0.30)	23.75 (+- 0.59)
component grid 4	(10, 5, 3, 4, 3)	-1	24.91 (+- 0.32)	28.79 (+- 0.74)
combination	-	-	24.90	31.30
target run	(11,5,5,4,3)	-	-	28.26 (+- 0.85)

Table 3.6: Recombination in x and z with $\ell_{min} = (9, 5, 3, 4, 3)$ and $\ell_{max} = (11, 5, 5, 4, 3)$. We show results for the averaged Q_{es} values with recombination and for an independent simulation. In addition the statistical errors are shown. This is compared to the combined values and the result of an target level run with a full grid of $\boldsymbol{\ell} = \ell_{max}$.

description	$\boldsymbol{\ell}$	$c_{\boldsymbol{\ell}}$	avg. Q_{es} recombination	avg. Q_{es} independent
component grid 0	(8, 5, 3, 6, 3)	1	28.77(+- 0.97)	32.51 (+- 0.78)
component grid 1	(10, 5, 3, 4, 3)	1	27.75 (+- 0.94)	28.79(+- 0.74)
component grid 2	(9, 5, 3, 5, 3)	1	28.15 (+- 0.82)	29.04 (+- 0.53)
component grid 3	(8, 5, 3, 5, 3)	-1	28.78 (+- 1.00)	31.74 (+- 0.55)
component grid 4	(9, 5, 3, 4, 3)	-1	27.96 (+- 0.82)	29.96 (+- 0.57)
combination	-	-	27.93	28.63
target run	(10,5,3,6,3)	-	-	27.67 (+- 0.94)

Table 3.7: Recombination in x and v with $\ell_{min} = (8, 5, 3, 4, 3)$ and $\ell_{max} = (10, 5, 4, 6, 3)$. We show results for the averaged Q_{es} values with recombination and for an independent simulation. In addition the statistical errors are shown. This is compared to the combined values and the result of an target level run with a full grid of $\boldsymbol{\ell} = \ell_{max}$.

Tables 3.5 to 3.7 the results for three different experiments are listed. We varied the combined dimensions and the minimum and maximum levels but stayed at 5 component grids to remain stable. In all of these experiments, the combined Q_{es} value is closer to the target value if we apply recombination. This is a first indication that recombination does also work for non-linear GENE simulation as long as they remain stable. Due to the large statistical errors that arise in the averaging process, we cannot be completely sure if these results are just lucky shots. However, since all results point towards this conclusion, we can follow that it is at least likely that the recombination works for these situations and that it is superior to a single combination of independent component grids without frequent recombination steps. Hence, recombination with DisCoTec provides an error reduction for such cases while having similar cost.

To conclude this section, we can say that non-linear simulations introduce still problems for the Combination Technique. We observed instabilities especially for small combination intervals and large number of component grids. These instabilities need to be further examined in future research to enable a robust simulation of non-linear PDEs with the Combination Technique. Fortunately, the stable test cases seem to work well with our implementation which indicates that there is a potential to solve such problems with the Combination Technique.

3.6.5 Summary

In the last sections, we have presented our numerical results with the novel additions to DisCoTec. We have shown tests with realistic plasma physics simulations in GENE that demonstrate the applicability of our implementation. We demonstrated that our version of the FTCT can handle large numbers of faults with negligible cost and results close to the fault-free simulation. This indicates that our implementation is suited even in faulty environments with frequent failure events. In addition, the asynchronous version of the Combination Technique can perform similarly to the synchronous Combination Technique for many cases. Moreover, the asynchronous mode of the Combination Technique allows for a much more efficient combination step which is currently the bottleneck of the complete algorithm. Thus, we have set the ground work for future exascale computations where massively parallel and robust frameworks are required to handle millions of potentially unreliable processing units.

Furthermore, we tested non-linear plasma simulation. Here, we observed mixed results. On the one hand, the simulations showed sometimes instabilities, especially for large simulation times, frequent recombinations or large index sets. On the other hand, we achieved good combination results for the most important quantities of interest with the stable simulations. This indicates that there are still unsolved problems that arise with non-linear simulations for the Combination Technique. More research has to be conducted to analyse these effects and to improve the robustness of the Combination Technique. If these problems can be solved, it would open up completely new possibilities with a much broader spectrum of possible PDE applications.

4 `sparseSpACE`: Spatial adaptivity for the Combination Technique

We have seen in Section 2.3 that the standard Combination Technique uses regular full grids. As a consequence, spatial adaptivity is not possible as this would violate the regular structure of the component grids. Hence, it is not possible to carefully tailor the grids to the application for problems that show for example complex localized behavior. An adaptive variant needs to lift this restriction to regular grids and should at the same time preserve the main properties of the Combination Technique: the error cancellation (see Section 2.3.1), the parallel execution of the component grids, and the black-box property. The latter means that we can separate the grid generation from the computation on the respective grid. As a result, black-box solvers can be applied to a given grid and the results can be combined externally. This allows for a broad applicability of the Combination Technique and is in fact the main selling-point in comparison to an explicit Sparse Grid discretization. This property was also the main reason for the creation of the HPC framework in Chapter 3.

There were already first attempts (see Section 2.3.3.2) to generalize and modify the Combination Technique to achieve better performance for problems that need spatial adaptivity. These attempts either define a priori *graded grids* that cluster points in regions of interest or define a cell hierarchy that abolishes the notion of component grids. The former is in general infeasible as the important regions might not be known a priori or might even change during the simulation. It is therefore necessary to find automatically an appropriate adaptation. The latter methods violate the black-box property as no component grids are available that can be distributed to black-box solvers. As a consequence, many applications would therefore need to be completely reimplemented for such an adaptive Combination Technique within these cell hierarchies. Such a Combination Technique offers therefore no real benefit over classical Sparse Grid approaches which are already quite mature.

In this chapter we describe in detail our newly developed spatially adaptive generalizations for the Combination Technique: the dimension-wise refinement [85] in Section 4.1 and the Split-Extend method [84] in Section 4.2. In this work, we extends the already published versions for both algorithms. After introducing the theoretical and algorithmic details of the two approaches, we present in Section 4.3 the implementation details. All algorithms presented in this chapter were implemented in the Python-based framework `sparseSpACE`[1]. We will discuss the general software architecture as well as the application-specific adaptations. In Chapter 5 we then show numerical results for the adaptive methods.

[1] https://github.com/obersteiner/sparseSpACE

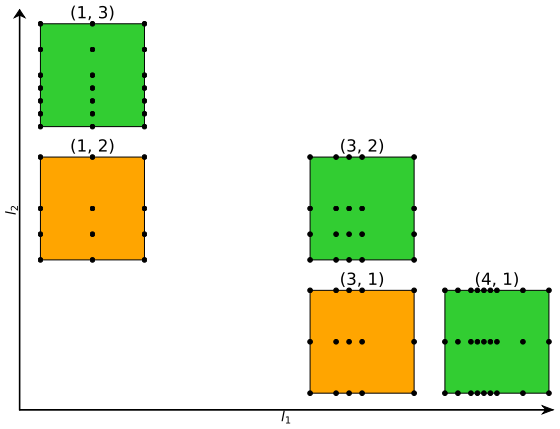


Figure 4.1: Spatially adaptive Combination Technique with rectilinear grids.

4.1 Dimension-wise refinement

The dimension-wise refinement [85] is an easy and comfortable option to get a spatially adaptive Combination Technique. In contrast to the Split-Extend method (Section 4.2), the dimension-wise refinement does not apply operations on d-dimensional subregions of the domain. Instead, we are applying spatial refinement on each dimension individually. For this we define a 1D point set for each dimension and refine it individually. Based on these refined point sets, we generate a combination scheme that contains rectilinear grids (see Fig. 4.1), i.e. grids that are constructed by tensor product construction from irregular spaced one-dimensional grid structures. Since all refinements that are used in this scheme are one-dimensional, common approaches for 1D problems can be utilized. This restriction to one-dimensional refinements is less flexible than the Split-Extend scheme but it can be quite effective especially if only few dimensions and/or only subregions of each dimensions need extensive refinement. In addition, it is usually very easy to integrate rectilinear grids in many applications and often such grids are already supported by the solver.

In the next sections, we will see how to construct the combination scheme that is shown in Fig. 4.1. We first summarize the concept of the 1D point sets in Section 4.1.1. Here, we focus on the initial construction of the sets and their refinement. After that we demonstrate in Section 4.1.2 how to use these point sets to obtain a valid combination scheme. This involves creating an appropriate index set for the combination, i.e. which component grids with corresponding level vectors ℓ should be used in the combination scheme, as well as the actual grid structure of the individual component grids. We are then looking at a technique to improve the 1d point sets via tree balancing in Section 4.1.3. This technique restructures the levels in the Sparse Grid which can help to target the important regions of the domain more effectively. Thereafter, we describe in

Section 4.1.4 the surplus-based error estimate that guides the adaptive process. Finally, we summarize the complete algorithm in Section 4.1.5.

4.1.1 1D point sets

The dimension-wise refinement is based on 1D point sets $\boldsymbol{P}^k$ and their levels $\boldsymbol{L}^k$ in the point hierarchy for each dimension $k \in [d]$. We assume both sets to be identically sorted so that point P_j^k has level L_j^k. By refining these point sets individually we can adapt the algorithm to the specific use case. Hence, the algorithm incorporates both ideas from spatial adaptivity and dimension adaptivity as we spatially refine single dimensions.

In our algorithm, we first initialize these sets with the regular point distribution of the starting level $\boldsymbol{\ell}$, i.e. $P_j^k = a_k + j \cdot h_k$ with $h_k = 2^{-\ell_k} \cdot (b_k - a_k)$ where $\boldsymbol{a}$ and $\boldsymbol{b}$ define the rectangular domain $D = [\boldsymbol{a}, \boldsymbol{b}]$. As an example, we would have for a starting level of $\ell = 2$, $\boldsymbol{a} = \boldsymbol{0}$, and $\boldsymbol{b} = \boldsymbol{1}$, for each dimension k the point distributions $\boldsymbol{P}^k = \{0, 0.25, 0.5, 0.75, 1\}$ and $\boldsymbol{L}^k = \{0, 2, 1, 2, 0\}$ or $\boldsymbol{P}^k = \{0.25, 0.5, 0.75\}$ and $\boldsymbol{L}^k = \{2, 1, 2\}$ depending if we add or omit boundary points. A visualization with boundary points can be seen in Fig. 4.2 (top left). In general, arbitrary starting configurations and level constructions could be used as long as one can define a tree structure of the points.

With these point sets we can now form point hierarchies by constructing binary trees that always connect points to their direct hierarchical parent, i.e. the hierarchical parent from the last level (see also Section 2.2.2). This is visualized in Fig. 4.2 (bottom left) for the previous example.

We can now use these point hierarchies to refine leaves. For this we calculate for each leaf the contribution to the solution and add their respective children if this contribution is high enough. This contribution is estimated by an error estimator that approximates for each leaf the potential error reduction that can be expected by refining it. If the chosen error estimate differentiates between the two possible children, only the most promising one of the children could be added. Otherwise both children are added. In Fig. 4.2 an example is shown were we refine a single point and add both of its children. A more detailed explanation of the error estimator is discussed in Section 4.1.4.

We have seen now how to initialize and how to refine the 1D point sets. The next section describes how these point hierarchies are used to create a consistent combination scheme that incorporates the adaptive idea efficiently.

4.1.2 Generating the combination scheme

The normal combination technique is based on an index set $\mathcal{I}$ that defines regular grids which are constructed for each level vectors $\boldsymbol{\ell} \in \mathcal{I}$. The index set is typically defined by an initial level ℓ and can be adaptively refined with the dimension adaptive algorithm. For the dimension-wise spatially adaptive scheme, we have so far only the 1D point sets. As a next step, we need to derive a corresponding index set and show how to map the points to the corresponding grids.

In this section, we want to generalize the construction of a combination scheme according to the one-dimensional point sets $\boldsymbol{P}^k$ and $\boldsymbol{L}^k$. This is done in a two-step ap-

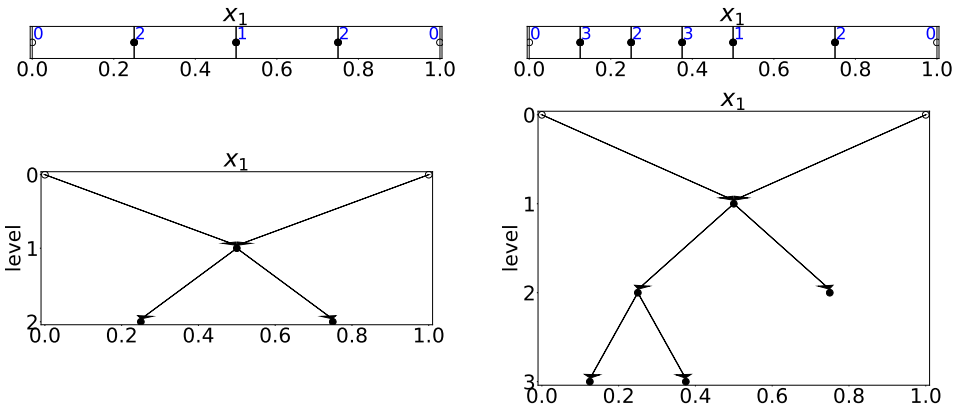


Figure 4.2: Visualization of the 1D refinement process. We show exemplary the starting configuration and a refinement step for dimension 1. **Left:** At the top we illustrate the starting configuration for the dimensions $\boldsymbol{P}^1$ (black dots) and $\boldsymbol{L}^1$ (blue) and at the bottom the respective tree hierarchy. **Right:** $\boldsymbol{P}^1$ (black dots) and $\boldsymbol{L}^1$ (blue) at the top and the tree hierarchy at the bottom after a single refinement of the leaf $x_1 = 0.25$.

proach: we first construct an appropriate index set $\mathcal{I}$ and then construct for each component grid $\boldsymbol{\ell}$ the specific one-dimensional point sets $\boldsymbol{P}^{k,\boldsymbol{\ell}}$ that define its 1D point sets. The actual grid structure is then build via tensor product from these 1D point sets $\boldsymbol{P}^{k,\boldsymbol{\ell}}$.

For the index set we use an approach that is very close to the classical index set of the Combination Technique. For this we will look at the maximum level $\boldsymbol{\ell}^{\max}$ that we reach with our point hierarchy in each dimension, i.e. $\ell_k^{\max} = \max(\boldsymbol{L}^k)$ for dimension k. We then define the index set according to the maximum level across all dimensions $\max(\boldsymbol{\ell}^{\max})$ but restrict it in the way that the maximum level in a dimension can only be reached when all other level vector entries are 1. In this way, we basically cut of the standard index set for dimensions that have a lower maximum level. The resulting index set is

$$\mathcal{I} = \{\boldsymbol{l} \in \mathbb{N}^d | \|\boldsymbol{l}\|_1 \leq \max(\boldsymbol{l}^{\max}) + d - 1, l_i < l_i^{\max} \vee (l_i = l_i^{\max} \wedge \forall k \in [d]/i : l_k = 1)\}. \quad (4.1)$$

It should be noted that the following construction is not limited to this definition of $\mathcal{I}$. In fact, our approach is compatible with any downward-closed index set and can therefore also be combined with any dimension-adaptive approach.

Based on this index set, we can now look at possibilities to construct $\boldsymbol{P}^{k,\boldsymbol{\ell}}$ for each $\boldsymbol{\ell} \in \mathcal{I}$. Here, it is important to first think about what such a mapping should guarantee so that we get a valid Combination Technique. An adaptive combination should ensure the error cancellation and represent an actual adaptive Sparse Grid. A valid combination scheme should therefore fulfill at least

$$\boldsymbol{P}^{k,\boldsymbol{i}} \subseteq \boldsymbol{P}^{k,\boldsymbol{j}} \text{ for } \boldsymbol{i}, \boldsymbol{j} \in \mathcal{I}, k \in [d], \boldsymbol{j} \geq \boldsymbol{i} \quad (4.2)$$

and

$$\boldsymbol{P}^{k,\boldsymbol{i}} = \boldsymbol{P}^{k,\boldsymbol{j}} \text{ for } \boldsymbol{i},\boldsymbol{j} \in \mathcal{I}, k \in [d], i_k = j_k. \tag{4.3}$$

In this definition, Eq. (4.2) enforces that by increasing the level vector of a component grid, only new points can be added to a point set but no points can be lost. The second condition in Eq. (4.3) requires that a point set $\boldsymbol{P}^k$ is only dependent on the value of ℓ_k of the level vector $\boldsymbol{\ell}$ and is not influenced by any other entry from another dimension. This second condition is required for the error cancellation as otherwise the grids that cancel the 1D errors in the combination would not have the same 1D point distributions. It should be noted that the first equation is only meaningful for nested grids. For combinations with non-nested grids a similar condition could be formulated that guarantees a growing point set. Such a growing point set should always be designed so that the approximation quality increases with higher levels [2]. Another useful but not necessary requirement is that a point is never added before any of his ancestors. If Eqs. (4.2) and (4.3) hold, this requirement can always be fulfilled by adjusting the point levels and therefore the point hierarchy. The standard Combination Technique of course fulfills these criteria.

We will now discuss three possible strategies for a valid combination scheme with our dimension-wise refinement.

Strategy 1 The easiest and straight-forward solution would be to add all points P_j^k when $L_j^k \leq \ell_k$, i.e. cut the point hierarchy at level ℓ_k. This results in

$$\boldsymbol{P}^{k,\boldsymbol{l}} = \{P_j^k \in \boldsymbol{P}^k | L_j^k \leq \ell_k\}. \tag{4.4}$$

One problem of this strategy (see Fig. 4.3) is that we introduce full grid-like structures in areas where the point hierarchy is low. As a result the leaves in this region are included in almost all component grids. An example would be if we have a tree were the maximum level is 4 but there exists a leaf at level 2 in dimension k. This leaf would be included in all component grids with $l_k \geq 2$ and therefore there is little variation in this region between different component grids for dimension k. This causes an almost full grid structure of level 2 in this region. We therefore have to delay the addition of new descendants in regions with low maximum depth in the tree. We will see such an approach in the next strategy.

Strategy 2 In this strategy we build on the experience from the last strategy and define a point-specific delay c_j^k. This delay has the consequence that a point is added at c_j^k levels after its own level, i.e. only if $L_j^k \leq l_k - c_j^k$. We define this delay based on how far a point is "behind the maximum level". Of course a point has a fixed level but the subtree rooted at the point has a maximum level that can be compared to the maximum level in the respective dimension. We therefore define the maximum level of all descendants of a point P_j^k by D_j^k and for the delay $c_j^k = \ell_k^{\max} - D_j^k$. For points that do not have any descendants, we set $D_j^k = L_j^k$. As a result, we get

[2] An example would be an increasing number of Gaussian quadrature points.

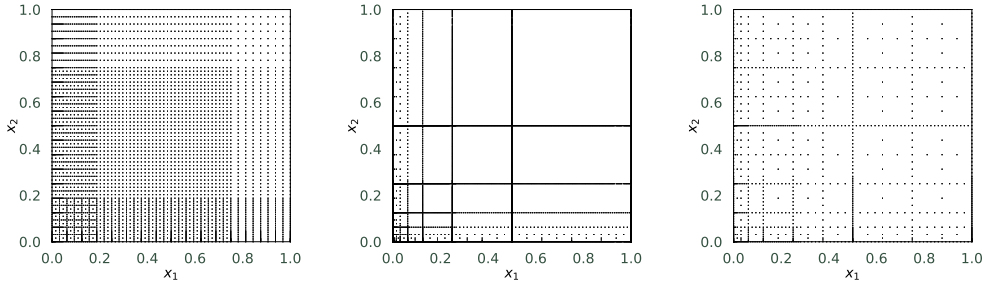


Figure 4.3: Comparison of the three different approaches (Eqs. (4.4) to (4.6)) for generating a valid combination scheme to reach a relative error tolerance of 10^{-4} for f_{expvar} (see Chapter 5), $\gamma = 0.9$ (see Section 4.2.5), and $d = 2$. **Left**: *Strategy 1* with 4349 points. **Middle:***Strategy 2* with 6145 points. **Right:** *Strategy 3* with 1075 points which was selected for all further test cases.

$$\boldsymbol{P}^{k,\boldsymbol{l}} = \{P_j^k \in \boldsymbol{P}^k | L_j^k \leq l_k - c_j^k\}. \tag{4.5}$$

Another interpretation of this delay is that we rise artificially the level of certain points so that they are only added to the component grids with a higher level.

Let us consider an example. If we look at the midpoint of the first dimension in the refinement step in Fig. 4.4 (left), we would have $D_7^1 = 4$ (for the grid point at $x_1 = 0.5$) as it has an descendant of level four (e.g. at $x_1 = 0.3125$). Another example in the same figure would be $D_8^1 = 2$ (at $x_1 = 0.75$ in the first dimension) as this point does not have any further descendants and as a consequence his own level is used. The delays for the points would be therefore $c_7^1 = 0$ and $c_8^1 = 2$ as the maximum level is 4.

This method efficiently eliminates full grid structure but it also removes too many points. This can be seen in Fig. 4.3 where interactions that are usually expected between the dimensions are missing in a large area of the domain. The reasons for this is that the delays c_j^k between different dimensions add up. So if we have a delay of $c_j^1 = 2$ for some j in dimension 1 and $c_{\tilde{j}}^2 = 1$ for some $\tilde{j}$ in dimension 2, then the interaction, i.e. grids where a point $\boldsymbol{x}$ with both $x_1 = P_j^1$ and $x_2 = P_{\tilde{j}}^2$ would exist, is actually delayed by 3. If this delay is too big, then this interaction is removed completely.

It should be noted that this method can construct grids with very low numbers of points which might be useful in situation where the interactions between the dimensions are not so essential for the accuracy. In our experiments with quadrature and interpolation, we observed fast convergence at low point numbers with this strategy. However, with increasing unbalance of the refinement trees the convergence is usually stagnating at some point and no further progress can be made. We therefore conclude that the interactions are indeed essential for many applications and need to be considered for constructing an efficient mapping. We therefore address this issue in the next strategy.

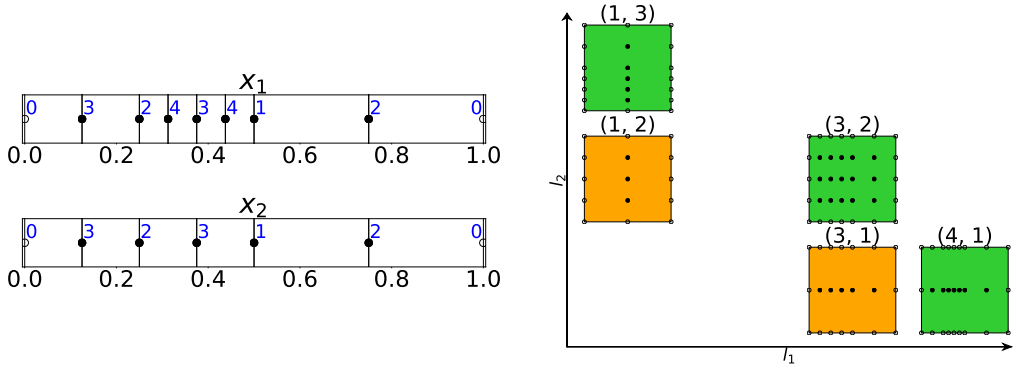


Figure 4.4: Example for *Strategy 1* Eq. (4.4) applied to the final refinement (left) of Fig. 4.5. We show $\boldsymbol{P^k}$ and $\boldsymbol{L^k}$ (blue) for each dimension x_k and the combination scheme (right). *Strategy 2* would be the same as *Strategy 3* from Fig. 4.5 for this refinement. Green component grids are added and orange ones are subtracted.

Strategy 3 In the previous strategies, we have seen that is essential to use a delay for the points from the tree refinement if we want to obtain Sparse Grid structures. At the same time, we have seen that such a delay has to be carefully chosen so that important interactions between dimensions are preserved. We therefore constructed a scheme were the previous delay c_j^k is reduced to $\tilde{c}_j^k \in [0.c_j^k]$. As we have outlined before, the delays between dimensions add up. Hence, the new delay has to consider also the other dimensions and distribute the delays in a balanced fashion. The approach we chose is

$$\boldsymbol{P}^{k,\boldsymbol{l}} = \{P_j^k \in \boldsymbol{P}^k | L_j^k \leq l_k - \tilde{c}_j^{k,\boldsymbol{l}} \wedge (L_j^k < D_j^k \vee l_k = l_k^{\max})\} \tag{4.6}$$

with

$$\begin{aligned}
\tilde{c}_j^k &= \max(\{m \in \mathbb{N} | g(c_j^k - (m-1), k) + \sum_{i=0}^{m-2} g(c_j^k - i, d) \leq c_j^k\} \cup \{0\}), \\
g(x, n) &= \sum_{s=1}^{n} h(x, s), \text{and} \\
h(x, s) &= \begin{cases} 1 \text{ if } \max(\boldsymbol{c}^s) \geq x \\ 0 \text{ otherwise} \end{cases} .
\end{aligned} \tag{4.7}$$

Here, the first clause of Eq. (4.6) $L_j^k \leq l_k - \tilde{c}_j^{k,\boldsymbol{l}}$ uses the modified delay $\tilde{c}_j^k$ while the second clause $(L_j^k < D_j^k \vee l_k = l_k^{\max})$ makes sure that leaves from the refinement trees are only added in component grids with maximum level in the respective dimension, i.e. for our case where all other levels are 1. This is reasonable as leaves, which represent the highest resolution in an area, are not meant to interact with other dimensions in a Sparse Grid. The calculation of $\tilde{c}_j^k$ is outlined in Eq. (4.7). Here we try to find a delay m that distributes all delays across all dimensions in a balanced fashion. In our case, we distribute the delays c_j^k in a round-robin approach. If we want to use a delay of

m, we therefore have to distribute first $m-1$ times delays to all *relevant* dimensions and in the last round it suffices if we only distribute delays until dimension k. Here, a dimension s counts as *relevant* if it has a sufficiently high maximum delay $\max(\boldsymbol{c}^s)$. This idea is incorporated in the two helper functions g and h. $g(x,n)$ sums up all relevant dimensions $s \in [n]$, i.e. all dimensions with $h(x,s)=1$. $h(x,s)$ checks if the dimensions s is *relevant* by comparing its maximum delay to x. This threshold x is reduced by 1 for each of the m rounds in which we distribute delays. Out of all possible m values, we then chose the maximum value for $\tilde{c}_j^k$ that guarantees that the total summation does not exceed c_j^k.

Let us now look again at an example. In the refinement shown in Fig. 4.4 (left), we have $\max(\boldsymbol{c}^1)=2$ and $\max(\boldsymbol{c}^2)=1$. For the point P_8^1 (at $x_1=0.75$), we have the delay $c_8^1=2$. Consequently, two "delays" are available that can be distributed across the dimensions. In this case, this results in $\tilde{c}_8^1=2$ since it is the only dimension with a c_j^k value ≥ 2 and the second delay is first distributed to dimension one. Similarly, the point P_6^2 (at $x_2=0.75$) with $c_6^2=1$, gets assigned $\tilde{c}_6^2=0$ since the only available delay is first assigned to dimension one.

In Fig. 4.3 we compare the refinements for all strategies with the same function that achieve a relative integral error of 10^{-4}. One can clearly see that *Strategy 3* performs best since it produces common Sparse Grid structures. *Stategy 1* creates a full grid in the upper right corner while *Strategy 2* puts almost no points in that regions as it removes too many points. The benefit of *Strategy 3* is also reflected in the point numbers that are needed to reach the tolerance which is by far the lowest for *Strategy 3*. This difference further increases with higher refinement levels.

A detailed example for the refinement process of *Strategy 3* can be found in Fig. 4.5. Here we show for different refinement steps the point sets $\boldsymbol{P}^k$ with their levels $\boldsymbol{L}^k$ and the resulting combination schemes and Sparse Grids. The corresponding final refinement step with *Strategy 1* can be seen in Fig. 4.4. At this stage *Strategy 2* and *Strategy 3* are still identical due to the fact that only for deep refinement trees, points are affected by the reduction of the delay to $\tilde{c}_j^k$. At the beginning only for leaves the delay is reduced but for leaves the second clause of Eq. (4.6) ensures a maximum delay.

4.1.3 Tree rebalancing

We have described in the last section how to map a point set $\boldsymbol{P}^k$ with its corresponding point hierarchy – represented as trees – to the points of a component grid $\boldsymbol{P}^{k,\boldsymbol{\ell}}$.

By adaptively refining this tree structure, we often observed a strong unbalance in the tree. This is of course related to the fact that if certain regions require excessive refinement the tree will grow more in those regions than in others. We have seen before that such an unbalance has to be treated carefully in order to preserve the Sparse Grid structure and obtain good results. An optimal mapping, however, requires a completely balanced tree as here the full potential of the Sparse Grid construction can be utilized.

Instead of only optimizing the mapping of $\boldsymbol{P}^k$ to $\boldsymbol{P}^{k,\boldsymbol{\ell}}$, we therefore introduced tree rebalancing to minimize the unbalance in the tree. Here, we make use of the existing

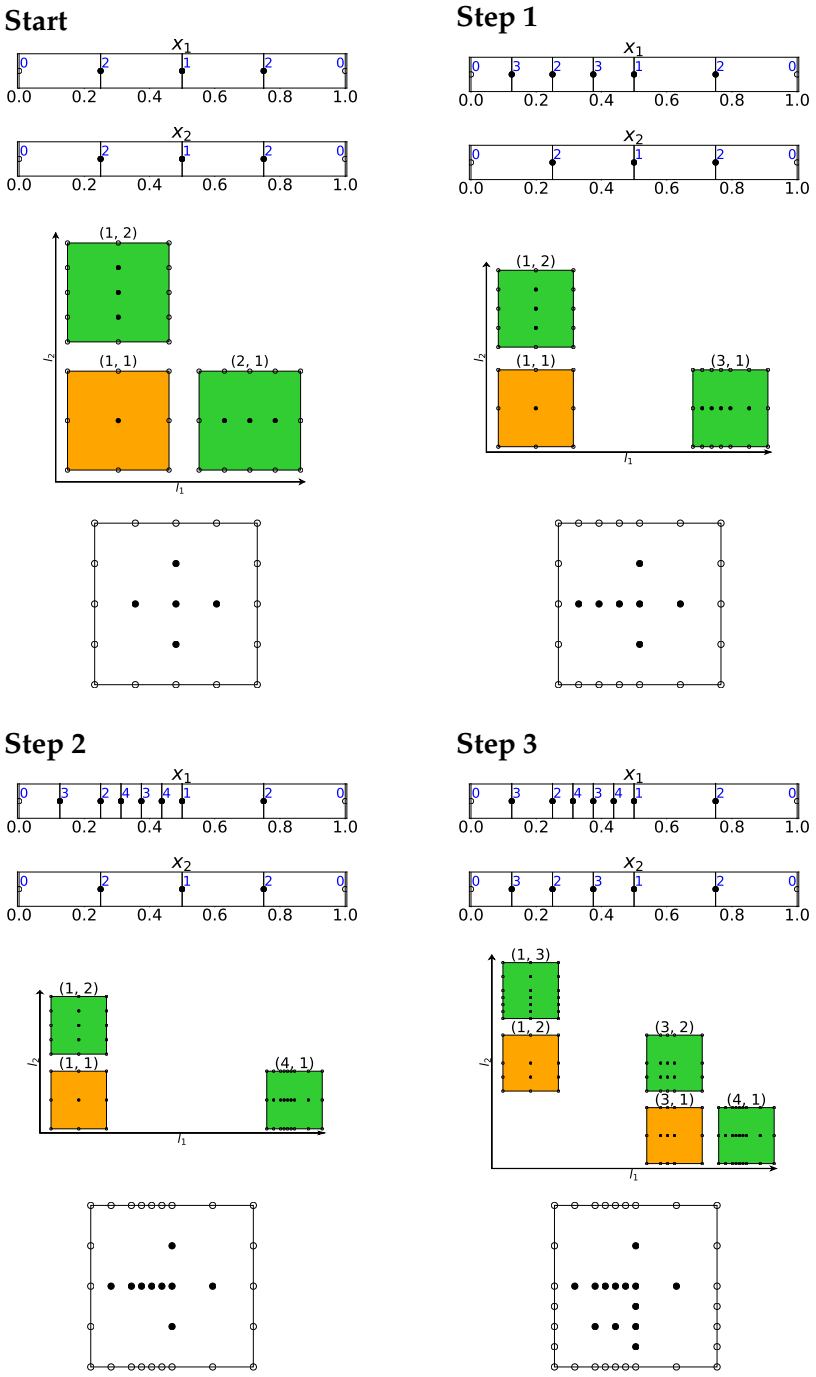


Figure 4.5: Three refinement steps of the dimension-wise spatial refinement. At each step we show $\boldsymbol{P^k}$ and $\boldsymbol{L^k}$ (blue) for each dimension x_k, the combination scheme, and the resulting sparse grid. Green component grids are added and orange ones are subtracted.

tree balancing strategies know from binary search trees. Algorithm 5 shows our approach, where we recursively restructure the tree if a too strong unbalance is observed. We assume that the tree is linked via pointers and for each node n we can address its left and right children via n.left and n.right. In the algorithm, we look at each recursive call at the subtree rooted in r and compare the number of points in the hierarchy starting from the right child |right_desc(r)| and left child |left_desc(r)|. Ideally, both children would have an equal number of points as ancestors. Unfortunately, this is usually not the case. Consequently, we test whether switching the right child or left child to the root improves the balancing and apply the restructuring if needed. Of course, this restructuring also requires some changes in the pointer structure of the binary tree which are outlined in Algorithm 5. To avoid rebalancing back and forth in consecutive refinements, we introduced a safety factor s that slightly delays a rebalancing operation by requiring a larger unbalance. In our test cases we used $s = 0.1$.

In Fig. 4.6 we show an example of this rebalancing strategy. Here, the unbalanced refinement tree can be completely balanced by applying a single rebalancing operation at the root. In the figure we also indicate in red how the point location associated with subspace $\boldsymbol{l} = (1, 1)$ is moved.

The complexity of the tree rebalancing procedure is $O(\sum_{k=1}^{d} |\boldsymbol{P}^k|)$ as we call the procedure recursively for every point in the 1D point sets and each call can be performed in constant time.

4.1.4 Error estimation

In the last sections we have seen how to generate from 1D point sets $\boldsymbol{P}^k$ component grids and how to optimize this mapping with tree rebalancing. The last part that is missing for an efficient adaptive algorithm is the calculation of error estimates that guide the adaptive process. These error estimates need to be localized to help us find points in the 1D point sets that need to be refined. We denote the error estimate for point P_j^k by ϵ_j^k. This error estimate is based on multiple estimates coming from each of the component grids. These component grid specific error estimates $\epsilon_j^{k,\boldsymbol{\ell}}$ are then combined according to

$$\epsilon_j^k = \left| \sum_{\boldsymbol{l} \in \mathcal{I}} c_{\boldsymbol{l}} \cdot \epsilon_j^{k,\boldsymbol{l}} \right|. \tag{4.8}$$

In this formula, we use the combination of component grid specific error estimates, which are all positive, to get to the final error estimate. This combination might be counterintuitive as we do not know the actual sign of the errors. In reality the combination could amplify or cancel the errors in various ways. However, moving the absolute value inside the summation to get an upper bound would result in ever growing errors due to the growing number of component grids with higher refinement levels. Therefore, the error estimates would not converge. The combination of error is therefore a way to avoid this problem. Another motivation for this error combination is the er-

Algorithm 5 Pseudocode for the recursive tree rebalancing starting from root r

```
procedure REBALANCE(r)                          ▷ Output: Rebalanced tree with root r_new
    total_desc = |right_desc(r)| + |left_desc(r)|
    if total_desc = 0 then
        return r
    end if
    ratio_left = |left_desc(r)| / total_desc
    ratio_right = 1 - ratio_left
    if ratio_left > ratio_right then
        ratio_left_child = |left_desc(r.left)| / total_desc
        if |ratio_left_child - 0.5| + s < |ratio_left - 0.5| then
            r_new = r.left                                        ▷ Rebalance to Left
            r.left = r_new.right
            r_new.right = r
        else
            r_new = r
        end if
    else
        ratio_right_child = |right_desc(r.right)| / total_desc
        if |ratio_right_child - 0.5| + s < |ratio_right - 0.5| then
            r_new = r.right                                       ▷ Rebalance to Right
            r.left = r_new.left
            r_new.left = r
        else
            r_new = r
        end if
    end if
    r_new.left = REBALANCE(r_new.left)                ▷ Recursive call on left child
    r_new.right = REBALANCE(r_new.right)              ▷ Recursive call on right child
    return r_new
end procedure
```

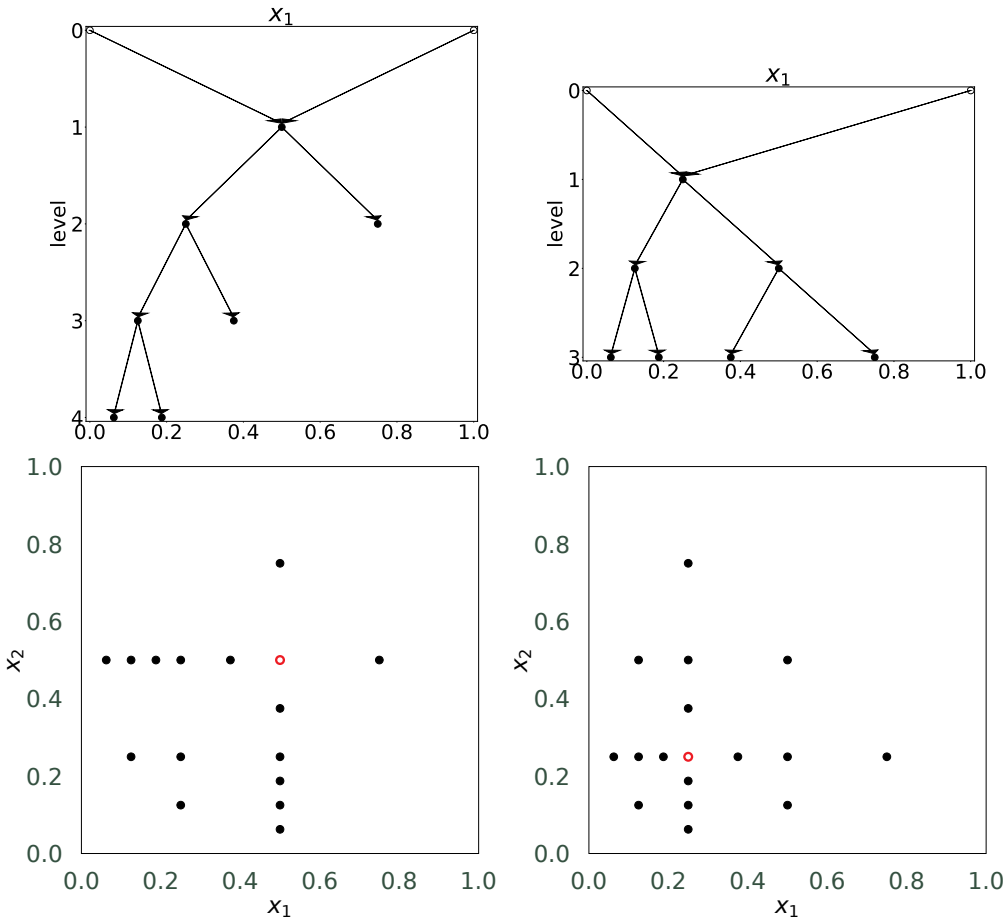


Figure 4.6: Rebalancing of the refinement trees and resulting Sparse Grid. **Left**: Initial refinement tree (equal for both dimensions) with clear unbalance and corresponding Sparse Grid below. **Right:** Refinement tree after rebalancing (for both dimensions) and corresponding Sparse Grid below. We have marked the point in the subspace with level vector $\boldsymbol{l} = (1, 1)$ in red and without filling for both situations.

ror cancellation of the Combination Technique (Section 2.3.1) which is similar to our procedure.

The question is now how we calculate the component grid specificc errors ϵ_j^k. For this we make use of the well-known surplus error estimate which uses the hierarchical surplus (see Section 2.2.2). The surplus value is suited as an error estimate as it approximates the recent changes in the function value and therefore a point's contribution. In addition, the surplus value is a measure for the second derivative [36]. Since we need only error estimates for 1D point sets, we use the 1D counterpart of the surplus, i.e. we only execute the first hierarchization step with the unidirectional principle for the respective dimension. By summing up these 1D surplus values along all other dimensions we get a projection into 1D.

In Algorithm 6, we oultine the complete error calculation for dimension k. We first iterate over all 1D slices in dimension k, i.e. we fix all other dimensions and iterate through dimension k and repeat this process for every combination in the other dimensions. For each of the slices the 1D hierarchization is performed by solving $A\boldsymbol{\alpha} = \boldsymbol{y}$ with the matrix $A_{ij} = \phi_i^{hier}(P_j^{k,\boldsymbol{\ell}})$. Here we evaluate the 1D hierarchical basis functions ϕ_i^{hier} (see Section 2.2.2) associated to point $P_i^{k,\boldsymbol{\ell}}$ with level $L_i^{k,\boldsymbol{\ell}}$ at the points $P_j^{k,\boldsymbol{\ell}}$ and calculate the function values $y_k = f(\boldsymbol{x})$ with $x_k = P_j^{k,\boldsymbol{\ell}}$ and the other components of $\boldsymbol{x}$ set to the fixed coordinates of the 1D slice from the other dimensions. The resulting surplus values $\boldsymbol{\alpha}$ are then used as error estimates. It should be noted that the matrix A is not inverted but instead we solve a linear system of equations via Gaussian elimination. As the matrix stays constant we can also decrease the complexity of the repeated solve by computing the LU decomposition (or a similar decomposition) of A. For the linear hat basis this system of equations can even be solved in linear time by using the hierarchization shown in Section 2.2.2.

One important observation is that the leaves of $\boldsymbol{P}^k$ are only contained in grids with highest resolution in dimension k. Since we only need error estimates for these leaves, we need to map the surplus values to the leaves. This is done by using the surplus α_j for point $P_j^{k,\boldsymbol{\ell}}$ if it is a local leave in $\boldsymbol{P}^{k,\boldsymbol{\ell}}$ and split it equally between all of his ancestors in the point hierarchy $\boldsymbol{P}^k$ that are leaves. We also multiply the surplus by the volume of the partially hierarchized basis function associated to point $\boldsymbol{x}$. This ensures that we do not refine too excessively towards less and less relevant basis function with a small support.

The overall complexity of the error estimation for a component grid $\boldsymbol{\ell}$ is therefore $\mathcal{O}(N^{\boldsymbol{\ell}} \cdot d + \sum_{k=1}^{d} |\boldsymbol{P}^k|)$ as we need to compute the error estimates for all d dimensions and each calculation is done via the 1D hierarchization of the linear hat basis ($\mathcal{O}(N^{\boldsymbol{\ell}})$) and thereafter these error estimates are applied to each point set $\boldsymbol{P}^k$. Here, $N^{\boldsymbol{\ell}} = \prod_{k=1}^{d} n_k$ denotes the number of points in the component grid $\boldsymbol{\ell}$ with $n_k = |\boldsymbol{P}^{k,\boldsymbol{l}}|$.

4.1.5 Overall Algorithm

We have now seen all the components of the dimension-wise refinement. The only thing that is left is to combine these components to get the overall refinement procedure. This procedure is shown in Algorithm 7. First, we initialize the combination scheme and then we enter the while loop that calculates the combination result. In this loop, we first iterate over all component grids, generate for the level vector $\boldsymbol{\ell}$ the corresponding 1D point sets, apply the application-specific operation on the resulting grid, and combine it to get the result. We also calculate for each of the component grids the dimension specific errors (see also Algorithm 6). In the next step we calculate the global error ϵ by either comparing to a reference solution or using the error estimates from Section 4.1.4. If this error is below a pre-defined tolerance or if the maximum number of points is exceeded, we terminate the procedure. Otherwise the refinement process is started.

Algorithm 6 Calculation of the error estimates for the dimension k of the component grid $\boldsymbol{\ell}$ with $n_s = |\boldsymbol{P}^{s,\boldsymbol{\ell}}|$.

procedure CALC_ERRORS($\boldsymbol{P}^{1,\boldsymbol{\ell}}$, ..., $\boldsymbol{P}^{d,\boldsymbol{\ell}}$, $\boldsymbol{L}^{1,\boldsymbol{\ell}}$, ..., $\boldsymbol{L}^{d,\boldsymbol{\ell}}$, k, $\boldsymbol{\ell}$) ▷ Output: $\epsilon_1^{k,\boldsymbol{\ell}}, ..., \epsilon_{n_k}^{k,\boldsymbol{\ell}}$
  build Matrix A
  initialize $\boldsymbol{\alpha}$ and $\epsilon^{k,\boldsymbol{\ell}}$ with **0**
  for $s = 1$ to d **do**
    if $s \neq k$ **then**
      $\tilde{n}_s = n_s$
    else
      $\tilde{n}_s = 1$
    end if
  end for
  for $\boldsymbol{i} \in \prod_{s=1}^{d}[\tilde{n}_s]$ **do** ▷ iterate over all dimensions except for k
    for $s = 1$ to d and $s \neq k$ **do**
      $x_s^{\boldsymbol{i}} = P_{i_s}^{s,\boldsymbol{\ell}}$
    end for
    for $j = 1$ to n_k **do** ▷ iterate through dimension k and fill vector $\boldsymbol{b}$
      $x_k^{\boldsymbol{i}} = P_j^{k,\boldsymbol{\ell}}$
      $b_j = f(\boldsymbol{x}^{\boldsymbol{i}})$
    end for
    $\boldsymbol{\alpha}$ += $A^{-1}\boldsymbol{b}$ ▷ solve system for hierarchization
  end for
  for $j = 1$ to n_k **do** ▷ sum up error values
    if $P_j^{k,\boldsymbol{\ell}}$ is local leaf **then**
      volume = $2^{-L_j^{k,\boldsymbol{\ell}}}$
      for $s = 1$ to d and $s \neq k$ **do**
        volume *= $2^{-\ell_s}$
      end for
      for $leaf$ in $leaves(P_j^{k,\boldsymbol{\ell}}, k)$ **do**
        $\epsilon_{leaf}^{k,\boldsymbol{\ell}}$ += volume * $|\alpha_{i_k}|/|leaves(P_j^{k,\boldsymbol{\ell}}, k)|^m$
      end for
    end if
  end for
end procedure

Algorithm 7 Pseudocode for the spatially adaptive algorithm for integrating f in the domain defined by a and b. We pass a reference solution to calculate exact errors.

procedure DIMENSION_WISE_COMBI($\boldsymbol{a}$, $\boldsymbol{b}$, tolerance, reference, f, max_points)
▷ Output: integral
    initialize combiScheme of level $\ell = 2$
    while true **do**
        result = 0
        for combiGrid in combiScheme **do**
            $\boldsymbol{l}$ = combiGrid.levelvector
            $\boldsymbol{P}^{1,\boldsymbol{\ell}}, ..., \boldsymbol{P}^{d,\boldsymbol{\ell}}, \boldsymbol{L}^{1,\boldsymbol{\ell}}, ..., \boldsymbol{L}^{d,\boldsymbol{\ell}}$ = GET_COMBI_GRID($\boldsymbol{P}^1, ..., \boldsymbol{P}^d, \boldsymbol{L}^1, ..., \boldsymbol{L}^d, \boldsymbol{l}$)
                ▷ generate tensor product grid and calculate integral
            result += grid.OPERATION(f, $\boldsymbol{P}^{1,\boldsymbol{\ell}}, ..., \boldsymbol{P}^{d,\boldsymbol{\ell}}$) · combiGrid.coefficient
            for $k = 1$ to d **do** ▷ calculate errors estimates for every dimension
                CALC_ERRORS($\boldsymbol{P}^{1,\boldsymbol{\ell}}, ..., \boldsymbol{P}^{d,\boldsymbol{\ell}}, \boldsymbol{L}^{1,\boldsymbol{\ell}}, ..., \boldsymbol{L}^{d,\boldsymbol{\ell}}, k, \boldsymbol{\ell}$)
            end for
        end for
        error = |(integral-reference)/reference|
        if error $<$ tolerance or number_of_points ¿ max_evaluations **then**
            break
        end if
        γ = 0.5 ▷ refining
        max_error = refinement.GET_MAX_ERROR()
        for $k = 1$ to d **do**
            $\boldsymbol{P}^k, \boldsymbol{L}^k$ = REFINE_LEAVES($\boldsymbol{P}^k, \boldsymbol{L}^k$, max_error ·$\gamma$)
        end for
        for $k = 1$ to d **do** ▷ tree rebalancing
            root$_k$ = GET_ROOT($\boldsymbol{P}^k, \boldsymbol{L}^k$)
            root$_k$ = REBALANCE(root$_k$)
            $\boldsymbol{P}^k, \boldsymbol{L}^k$ = UPDATE_LEVELS(root$_k$)
        end for
    end while
end procedure

In the refinement procedure, we first refine the leaves in the 1D point sets that have a local error $\epsilon_j^k \geq \gamma \cdot \epsilon^{\max}$ with the maximum local error $\epsilon^{\max} = \max_{k\in[d]} \left(\max_{j\in[n_k]} \epsilon_j^k\right)$. Afterwards the rebalancing strategy (see Section 4.1.3) rebalances the tree if necessary.

The overall complexity of the whole algorithm for the n-th iteration is

$$O\left(\left(\sum_{\boldsymbol{\ell}\in\mathcal{I}^{(n)}, c_{\boldsymbol{\ell}}!=0} N^{\boldsymbol{\ell},(n)} \cdot d + N^{\boldsymbol{\ell},(n)} \cdot d + \sum_{k=1}^{d} \left|\boldsymbol{P}^{k,(n)}\right|\right) + \sum_{k=1}^{d} \left|\boldsymbol{P}^{k,(n)}\right| + \sum_{k=1}^{d} \left|\boldsymbol{P}^{k,(n)}\right|\right),$$

where $N^{\boldsymbol{\ell},(n)}$ and $\boldsymbol{P}^{k,(n)}$ are the number of grid point in grid l and the 1D point sets in this iteration, respectively. The first summation covers the complexity of the grid operation[3] and the error calculation with the hierarchization using linear hat basis functions. These operations are applied to every component grid. The last two terms are from the rebalancing and refinement procedures which are only called once every refinement step. Since usually $d \cdot N^{\boldsymbol{\ell},(n)} > \sum_{k=1}^{d} \left|\boldsymbol{P}^{k,(n)}\right|$, this can be further simplified to

$$O(d \cdot \sum_{\boldsymbol{\ell}\in I^{(n)}, c_{\boldsymbol{l}}!=0} N^{\boldsymbol{\ell},(n)}).$$

This is optimal as it is the same complexity that is already needed for iterating through all points for every component grid. Of course we might need several refinement steps until we reach the maximum number of points or fall below the tolerance. In theory, we can reuse computation to reduce the cost for such a use case but the solver would also need to support such an adaptive refinement. This is the reason why we usually reevaluate everything for every refinement step.

To conclude this section, we can say that we have presented a novel dimension-wise spatially adaptive refinement procedure that uses adaptively refined 1D point sets to construct a valid combination scheme. The resulting rectilinear grids can be processed in a black-box manner with suited solvers. In addition, the computational complexity per time step is linear in the number of grid points which is an optimal complexity for such an approach.

4.2 Split-Extend scheme

In this section we will give a detailed description of the Split-Extend scheme [84]. The approach is characterized by a combination of two refinement operations that can be applied on d-dimensional – usually rectangular – subregions of the domain: the *Split* and the *Extend* operation. After each refinement step, a consistent combination scheme is formed that creates component grids covering the whole domain. These grids are block-adaptive, i.e. grids that are formed of blocks of regular grids (see Fig. 4.7). Suited black-box solvers can then be used for the computation on these grids. Block-adaptive

[3] We assume here $\mathcal{O}(N_{\boldsymbol{l}}^{(n)} \cdot d)$ which holds for many operations such as integration

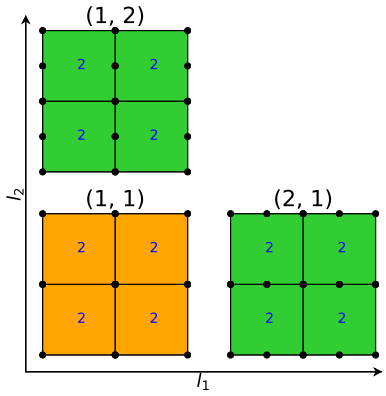


Figure 4.7: Initial *Split* for standard Combination Technique with $\boldsymbol{\ell} = \mathbf{2}$ and $\boldsymbol{\tau} = \mathbf{0}$. Each of the subareas resemble a Combination Technique with $\boldsymbol{\ell} = \mathbf{2}$ and $\boldsymbol{\tau} = -\mathbf{1}$. We show the local level ℓ_i for each subregion in blue.

grids are for example common in PDE calculations and were also implemented for GENE [65].

The *Split* operation can split a subregion in an octree-like fashion, i.e. the domain is split in one or multiple dimensions, which results in multiple smaller disjoint subregions. This operation allows for more fine-grained refinement as it allows the algorithm to focus on specific subregions tailored to the application. Depending on the configuration and the needs of the application, each call to the operation can split the subregion in single or multiple dimensions.

The *Extend* operation does not modify the boundaries of a subregion but it increases the Sparse Grid level of the subregion. This is implemented with the use of local levels which are assigned to each subregion. The *Extend* operation simply increases the local level of a specified subregion by one. As a result, the number of Sparse Grid points inside the subregion will grow.

In the next sections, we give a detailed explanation and motivation for both operations. To formalize the mathematical description, we assign each subregion a unique identifier $\boldsymbol{i}$ which could be for example the boundaries of its subdomain. In addition, we outline how to generate a consistent combination scheme after applying these operations. Thereafter, we look at how to estimate errors that guide the adaptation process. At this point, we mainly focus on numerical quadrature, but a possible adaptation for other applications is straight-forward. Finally, we describe the structure of the overall algorithm that is used in our implementation.

4.2.1 Initial setup

For the initial configuration of the algorithm, we will start with a standard Combination Technique of level $\ell = 2$. If we look closely at Fig. 4.7, we can see that the scheme can also be seen as a union of 2^d subregions where each subregion forms a truncated

combination scheme (see Section 2.3.1) with $\ell = 2$ and $\boldsymbol{\tau} = -\mathbf{1}$. This observation gives an important insight: there is no difference between applying the individual point pattern to 2^d subregions separately or to increase the minimum level by $\mathbf{1}$ and apply it to the complete domain. In our case, the point pattern within the subregions is that of a Combination Technique with $\ell = 2$ and $\boldsymbol{\tau} = -\mathbf{1}$, but the overall structure originates from the Combination Technique with $\ell = 2$ and $\boldsymbol{\tau} = \mathbf{0}$, which is the standard Combination Technique of level 2. This equality of splitting the domain and increasing the minimum level is actually a general pattern. Adding $\mathbf{1}$ to the minimal level is equal to using the current point pattern in a region $\boldsymbol{i}$ and applying it to each subregion of an octree-like split of the region $\boldsymbol{i}$. This concept is important for the *Split* operation. The initial *Split* into 2^d subregions is the starting point for the Split-Extend algorithm.

In addition to this *Split*, we define for each subregion $\boldsymbol{i}$ a local level $\ell_{\boldsymbol{i}}$. This local level describes the local size of the combination scheme and is initialized to $\ell_{\boldsymbol{i}} = 2$ for the initial subregions as we start with a regular Combination Technique with level $\ell = 2$.

4.2.2 Split

In the previous section, we outlined the initial setup for the Split-Extend method. We create initially 2^d subregions $\boldsymbol{i}$ with local level $\ell_{\boldsymbol{i}} = 2$ and $\boldsymbol{\tau} = -\mathbf{1}$. We have also seen that increasing the parameter $\boldsymbol{\tau}$ by $\mathbf{1}$ is equal to splitting a region $\boldsymbol{i}$ equally into 2^d regions and replicating the previous point pattern of region $\boldsymbol{i}$ in each of the new subregions.

This concept is used in the *Split* operation. We, however, never actually increase the $\boldsymbol{\tau}$ value but keep it at $-\mathbf{1}$. Instead we create a separate combination for each of the newly created subregions $\boldsymbol{j} \in \text{children}(\boldsymbol{i})$ by using the replication idea. We address the parent cell $\boldsymbol{i}$ later by $\text{parent}(\boldsymbol{j}) = \boldsymbol{i}$. In Fig. 4.8, we show two refinement steps. Here, we select always one subregion for refinement and then split it in an octree-like fashion into its $2^2 = 4$ children. Then we apply the combination scheme from the parent on each of these children. By repeating this process, we can target more and more fine-grained where we want to add points. It is important to see that all local levels $\ell_{\boldsymbol{j}} = \ell_{\boldsymbol{i}}$ for $\boldsymbol{j} \in \text{children}(\boldsymbol{i})$ stay unchanged and are inherited from the parent subarea $\boldsymbol{i}$.

Instead of splitting all dimensions at the same time, it is of course also possible to perform the split only in a single dimensions which creates 2 children. This resembles a Combination Technique where we increase $\boldsymbol{\tau}$ only in the respective dimension k by 1, but again we are instead replicating the existing combination pattern in both children. An example is shown in Fig. 4.9 where only dimension x_1 is split. Especially for high dimensions this is interesting as adding 2^d children is not feasible for such cases. In addition, this strategy benefits from use cases where some dimensions need less refinement. However, the more flexible the refinement gets, the more dependent we are on a good error estimate as otherwise suboptimal decisions can lead to inefficient refinements.

We have seen that the *Split* Operation can efficiently refine towards certain regions of the domain by performing octree-like splitting in singe dimensions or multiple. However, since the local levels $\ell_{\boldsymbol{i}}$ stay constant the combination scheme does not change.

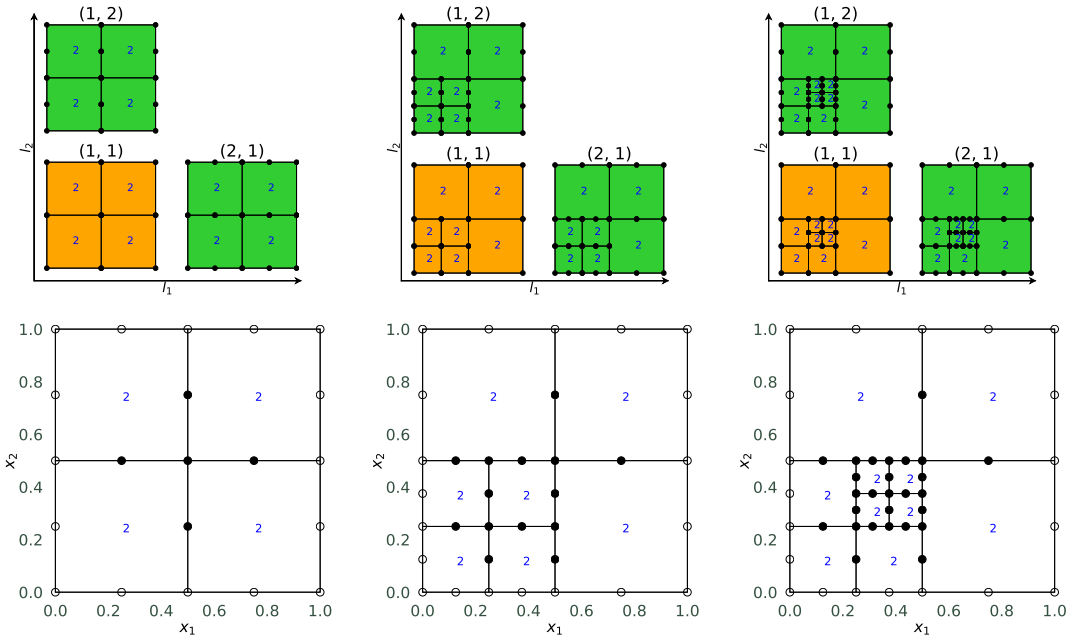


Figure 4.8: Two refinement steps with the *Split* operation: Starting from the initial setup (left), we perform two *Split* operations (middle and right). In the first step we refine the lower left area. Then out of the 4 new subareas the upper right one is selected for the next *Split* operation. We show the respective combination scheme on the top and the resulting Sparse Grid on the bottom. Local levels are inherited from the parent area and are shown in blue.

This has the effect that we are moving further towards a full grid structure since we are effectively increasing $\boldsymbol{\tau}$ but not ℓ. This process just replicates patches of the original Sparse Grid. The *Split* method is therefore only interesting to narrow down the location where we want to refine but not as a full refinement procedure. Additionally, we have to modify the local level $\ell_{\boldsymbol{i}}$ which will be done by the *Extend* operation.

4.2.3 Extend

We have seen now how to refine subareas to get the refinement more and more localized with the *Split* operation. In this section we will introduce the second operation *Extend*. With the *Extend* operation we do not change the boundaries of subareas but increase the local levels $\ell_{\boldsymbol{i}}$ by one. This can be seen in Fig. 4.10 where we refine twice the lower left subregion.

By increasing local levels, the different subregions do not fit anymore to one global combination scheme with fixed level ℓ. We therefore have to find the maximum local level and use this for the global combination scheme, i.e. $\ell = \max_{\boldsymbol{i} \in \text{refinement}} \ell_{\boldsymbol{i}}$. With this combination scheme, we are obtaining level vectors $\boldsymbol{\ell}$ according to the new index set $\mathcal{I}_\ell$.

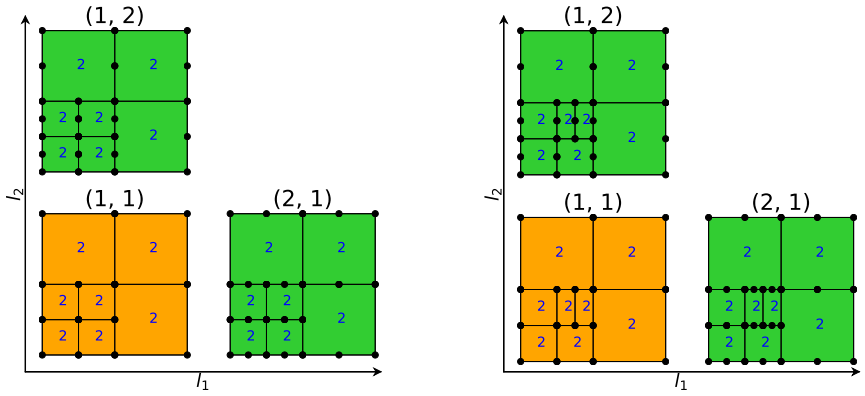


Figure 4.9: The *Split* operation in single dimensions: We show one split in dimension x_1. We show the respective combination scheme and the local levels in blue.

However, since not all subareas match this level, we cannot just apply the component grid level vector $\boldsymbol{\ell}$ to every region. Instead each of the subareas get its own coarsened level vector $\boldsymbol{\ell_i} \leq \boldsymbol{\ell}$ to adjust it to the potentially lower local level ℓ_i. This process of reducing the level vector is therefore referred to as coarsening which is defined by a mapping function $m_\ell : \mathbb{N}^d \times \mathbb{N} \to \mathbb{N}^d$ with $f_\ell(\boldsymbol{\ell}, \ell_i) = \boldsymbol{\ell_i}$. With this local level vector $\boldsymbol{\ell_i}$, we then create the respective local grid for the calculation in this subarea. By fusing all grids from all subareas, we then get the final component grids.

The question is now how to define such a mapping f. One of these procedures m_ℓ^1 is shown in Fig. 4.10 where $f_\ell(\boldsymbol{\ell}, \ell_i)$ recursively coarsen the level vector $n = \ell - \ell_i$ times by reducing in each step the maximum entry of the level vector by 1. In case there are multiple possible maximum entries, we reduce first the one with the smallest dimension number. If we end up with an invalid level vector that contains negative entries, we just map it to the level vector $\mathbf{0}$ and will ignore the respective result later (see next paragraph). The set of all not ignored level vectors will be denoted by $\tilde{\mathcal{I}}_\ell \subseteq \mathcal{I}_\ell$. This in fact specifies a surjective mapping of level vectors $\boldsymbol{\ell} \in \tilde{\mathcal{I}}_\ell$ to level vectors $\tilde{\boldsymbol{\ell}} \in \mathcal{I}_{\ell_i}$, i.e. $m_\ell^1 : \tilde{\mathcal{I}}_\ell \to \mathcal{I}_{\ell_i}$ and m_ℓ^1 surjective. This mapping is surjective as every level vector $\tilde{\boldsymbol{\ell}} \in \mathcal{I}_{\ell_i}$ has a respective level vector $\boldsymbol{\ell} \in \tilde{\mathcal{I}}_\ell$ where the maximum entry of $\tilde{\boldsymbol{\ell}}$ was increased $n = \ell - \ell_i$ times. In other words: if we consider a local level $\ell_i < l$, we can add to any level vector in the index set $\mathcal{I}_{\ell_i}$ the value $n = \ell - \ell_i$ to an arbitrary component – in our case the maximal component – and result in a level vector which is part of the index set I_ℓ. Hence, the mapping via coarsening is surjective. This mapping also guarantees that the combination coefficient is correct, i.e. $c_{\boldsymbol{\ell}} = c_{\boldsymbol{\ell_i}}$. This is important as this allows us to fuse the subarea to one big combination grid that still supports the black box property.

One problem of this approach is that since the mapping is surjective, we can hit some level vectors multiple times. This would lead to situations were points are accounted for too often or points are subtracted too often. We therefore only sum up the results

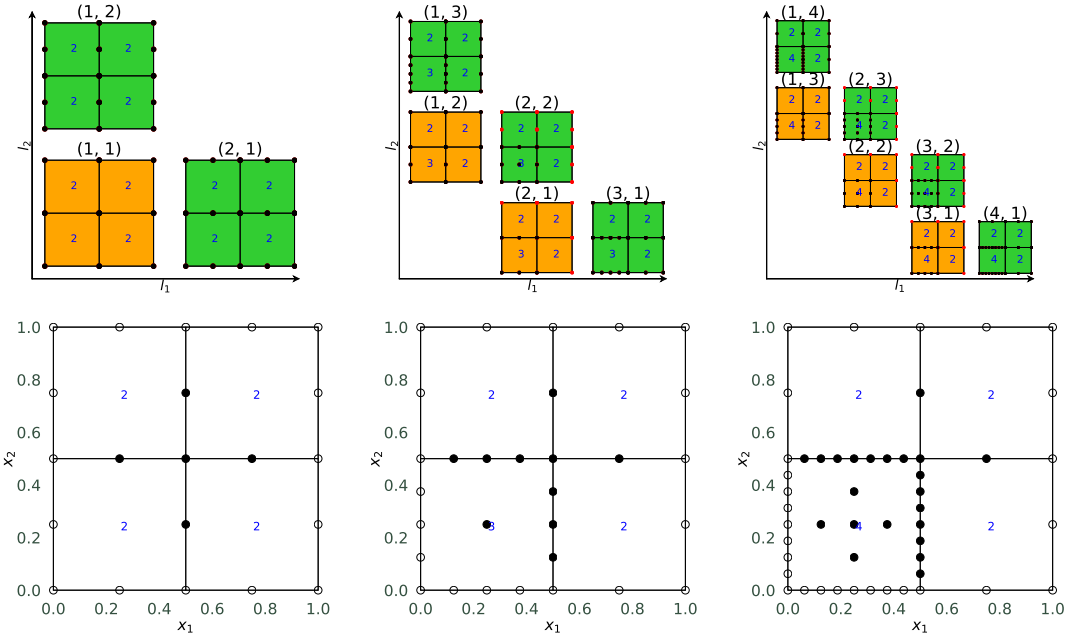


Figure 4.10: Two refinement steps with the *Extend* operation: Starting from the initial setup (left), we perform two *Extend* operations (middle and right). In both steps we increase the level in the lower left subarea. We show the respective combination scheme on the top and the resulting Sparse Grid on the bottom. Local levels are shown in blue. The areas with points marked in red are not considered for refinement.

for these regions at the first occurance of a local level vector $\boldsymbol{\ell_i}$. Otherwise, we ignore the contribution of this subarea in the final combination. This is visualized in Fig. 4.10 by the red marked points which define the affected subareas for which the local result is discarded. Areas for which the level vector was mapped to an invalid value are also always discarded and are filled up with points of level vector $\mathbf{0}$. This allows us to perform operations on the global grid but only combine the necessary information. We do not even need to compute the respective values for the red points if the respective operation on the grid does not require a grid that spans the complete domain. In such a case, we can just generate the component grids without the respective points.

It should be noted that the equations Eqs. (4.2) and (4.3) are not satisfied globally between subregions. The reason for this is that we are more or less building a separate Sparse Grid for each subregion. But even in each subregion it is not necessarily satisfied with the shown mapping. This does not create any problems as the level vectors that violate the condition are excluded from the final combination. If there is an application where it would be necessary to fulfil this condition or where we always want to include all grid values of every grid in the final combi, we have created another mapping $m_\ell^2 : \mathcal{I}_\ell \to \mathcal{I}_{\ell_i} \cup \mathcal{I}_\ell$. Here, we do not decrease the maximum element of the level vector but use $m_\ell^2(\boldsymbol{\ell}, \ell_{\boldsymbol{i}}) = \tilde{\boldsymbol{\ell}}$ with $\tilde{\ell}_k = max(0, \ell_k - \left\lfloor \frac{\ell - \ell_{\boldsymbol{i}} + (d-k)}{d} \right\rfloor)$. This is basically a round-robin

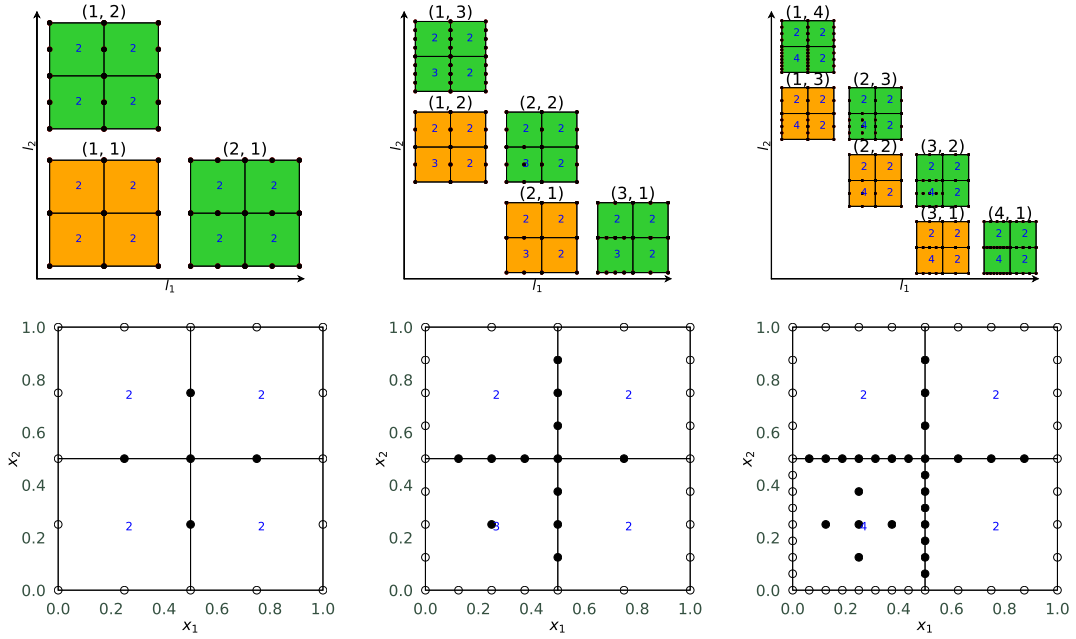


Figure 4.11: Two refinement steps of an *Extend* operation that avoids removing of points: Starting from the initial setup (left), we perform two *Extend* operations (middle and right). In both steps we increase the level in the lower left subarea. We show the respective combination scheme on the top and the resulting Sparse Grid on the bottom. Local levels are shown in blue. No points have to be removed but point numbers can grow in regions that are not refined.

distribution of the $n = \ell - \ell_i$ coarsenings where we discard coarsenings that would decrease the level vector below 0. This has the downside that we are not guaranteed to achieve a maximum coarsening which can increase point numbers in regions that are not affected by an *Extend* operation. However, we now satisfy locally Eqs. (4.2) and (4.3) as the coarsening of a the k-th component of a level vector is independent of the other dimensions and by increasing ℓ_k we cannot decrease $\tilde{\ell}_k$. An example of this mapping is shown in Fig. 4.11. We can see that in contrast to Fig. 4.10 new points are introduced in regions with unchanged local levels. However, the benefit is that we do not need to exclude points from the results any more.

We have now described how to create a consistent combination scheme in the presence of varying local levels. At this point, the only missing part for an adaptive algorithm is the estimation of the benefits that we get by refining a certain area and which operation we should chose. This is explained in the next section.

4.2.4 Error estimation

In the last sections, we have seen which refinement operations we can perform on a subarea. This section discusses the logical next step which is the topic of how to find

the subarea that we like to refine and which operation we would like to apply to it. We will only discuss error estimates for integration but similar definitions for other applications can be obtained if the integral value is for example swapped with the function approximation in a subarea. In these cases, we can make use of surplus values instead of integrals or use a suited norm on the subtraction of such function approximations from different refinement states. But now to the quadrature-specific errors.

First we will define a subarea-specific error estimate $\epsilon_{\boldsymbol{i}}$ that will show the potential benefit for refining the subarea. Here, we will always use the comparison of the current integral approximation $I_{\boldsymbol{i}}^{\text{new}}$ to the previous one $I_{\boldsymbol{i}}^{\text{old}}$ for subarea $\boldsymbol{i}$ which is defined by

$$\epsilon_{\boldsymbol{i}} = |I_{\boldsymbol{i}}^{\text{old}} - I_{\boldsymbol{i}}^{\text{new}}|. \tag{4.9}$$

This estimate shows the most recent change which is a good heuristic for the future trend of the convergence. In other words, we try to estimate the future changes by looking at the previous change of the integral.

In the following, we will analyze how this integral approximation I is calculated. We will use the notation $q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}})$ which calculates the quadrature of the function f with the local Sparse Grid level $\ell_{\boldsymbol{i}}$ in the subarea $\boldsymbol{i}$. This quadrature includes already the whole combination of component grid integrals. It can be defined as

$$q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}}) = \sum_{\boldsymbol{\ell} \in \mathcal{I}_\ell} c_{\boldsymbol{\ell}} \sum_{\boldsymbol{x} \in \text{points}(m(\boldsymbol{\ell}, \ell_{\boldsymbol{i}}), \boldsymbol{i})} f(\boldsymbol{x}) \cdot w_{\boldsymbol{x}} \tag{4.10}$$

where $m(\boldsymbol{\ell}, \ell_{\boldsymbol{i}})$ denotes the mapping function that coarsens the level vector $\boldsymbol{\ell}$ (see Section 4.2.3), points$(\tilde{\boldsymbol{\ell}}, \boldsymbol{i})$ the points generated with level vector $\tilde{\boldsymbol{\ell}}$ for region $\boldsymbol{i}$, $f(\boldsymbol{x})$ the function value at point $\boldsymbol{x}$, and $w_{\boldsymbol{x}}$ the quadrature weight for point $\boldsymbol{x}$. For a subarea $\boldsymbol{i}$ with local level $\ell_{\boldsymbol{i}}$, we therefore define $I_{\boldsymbol{i}}^{\text{new}} = q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}})$ as the current integral value. The number of function evaluations used in this quadrature is defined as $n_{\ell_{\boldsymbol{i}}, \boldsymbol{i}}$.

The old integral approximation $I_{\boldsymbol{i}}^{\text{old}}$ is a bit more tricky. Here, we need to consider which operation happened last that generated the current state of the subarea. For treating this problem we differentiate between the different basis functions that we use. We will first outline the main ideas for the linear basis. This includes how to choose the next operation once we have found the subarea we want to refine. Thereafter, we will describe the changes for higher order basis functions. Last, we will also describe how to adapt the error estimates so that we can perform *Split* operations in single dimensions.

4.2.4.1 Linear Basis

For the linear basis, we have various options to calculate the old integral approximation. In case the last operation was an *Extend*, we simple define $I_{\boldsymbol{i}}^{\text{old, ex}} = q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}} - 1)$ as only the local level increased. If the last operation was a *Split*, this is more complicated since we do not have a corresponding previous quadrature for the specified region but only for the parent area parent$(\boldsymbol{i})$. Hence, we have to map the approximation of the parent region to the specific subarea. We will describe three different variants for such

a mapping in the following. The first approach is to filter the integral contributions of the parent region that contribute to the subarea $\boldsymbol{i}$. For the linear basis this translates to

$$I_{\boldsymbol{i}}^{\text{old,sp,1}} = \sum_{\boldsymbol{\ell} \in \mathcal{I}_\ell} c_{\boldsymbol{\ell}} \sum_{\substack{\boldsymbol{x} \in \text{points}(m(\boldsymbol{\ell},\ell_{\boldsymbol{i}}),\text{parent}(i)) \\ \boldsymbol{x} \in \text{area}(\boldsymbol{i})}} w_{\boldsymbol{x}}^* f(\boldsymbol{x}) = \Pi(q(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}}), \boldsymbol{i}) \tag{4.11}$$

where we introduce $\Pi(q(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}}), \boldsymbol{i})$ which filters out the integral contributions of $q(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}})$ (see Eq. (4.10)) that fall into the subregion $\boldsymbol{i}$. area($\boldsymbol{i}$) returns the subdomain of the subregion $\boldsymbol{i}$, i.e. $[\boldsymbol{a_i}, \boldsymbol{b_i}]$ for the rectangular subarea with lower boundaries $\boldsymbol{a_i}$ and upper boundaries $\boldsymbol{b_i}$. We also use the modified weights $w_{\boldsymbol{x}}^*$ that adjust the weight for point x if x lies on the border between subareas. In this case the weight is split equally between those subareas, i.e. $w_{\boldsymbol{x}}^* = w_{\boldsymbol{x}} / |\{\boldsymbol{j} \in \text{children}(\text{parent}(i)) | \boldsymbol{x} \in \text{area}(\boldsymbol{j})\}|$.

Another option for the calculation of the old integral approximation after a split is to evaluate a surrogate $\tilde{f}$ of f based on the previous state before the *Split* and evaluated the integral based on this surrogate. To make this process more explicit, we use $\tilde{f} = \Gamma(f, \boldsymbol{i}, \ell_i)$. As a result we get

$$I_{\boldsymbol{i}}^{\text{old,sp,2}} = q(\tilde{f}, \boldsymbol{i}, \ell_{\boldsymbol{i}}) = q(\Gamma(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}}), \boldsymbol{i}, \ell_{\boldsymbol{i}}). \tag{4.12}$$

For the linear basis such a surrogate would be a d-linear interpolation of f based on the previous grid points of the parent. This surrogate $\tilde{f}$ can then be evaluated at the new grid points of the child just as the regular function. Hence, the quadrature approximation for the previous contribution is then $q(\tilde{f}, \boldsymbol{i}, \ell_{\boldsymbol{i}})$.

A less efficient alternative would be

$$I_{\boldsymbol{i}}^{\text{old,sp,3}} = q(\text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}}) / |\text{children}(\text{parent}(\boldsymbol{i}))|. \tag{4.13}$$

In this variant, all children have the same parent integral approximation value which could hide different contributions of the different subareas to the parent integral. If one of the subareas for example contributes to 90% of the parent integral, it is not optimal to just split the integral equally between all children. In this case the error estimate will not correctly indicate that one of the subareas has a higher impact.

Our results showed that the combination of both $I_{\boldsymbol{i}}^{\text{old,sp,1}}$ and $I_{\boldsymbol{i}}^{\text{old,sp,2}}$ resulted in the best refinement. In particular we used $\min(|I_{\boldsymbol{i}}^{\text{old,sp,1}} - I_{\boldsymbol{i}}^{\text{new}}|, |I_{\boldsymbol{i}}^{\text{old,sp,2}} - I_{\boldsymbol{i}}^{\text{new}}|)$ which is just the minimum of both associated errors. The reason for that is that both terms tend to overestimate the errors in different cases. The first term can for example better estimate the error for constant subregions of the parent region, while the second term performs better when the function is close to the d-linear surrogate.

With these error estimates, we have now a tool to detect the most promising subregion by selecting the regions with highest error estimate. But the next question is which operation we should perform for the selected subregion. We therefore have to estimate the potential gain that we can expect from a *Split* and an *Extend* operation. One intuitive approach is to select a splitting depth δ that defines how often we recursively split the

subareas before applying an *Extend* operation. If we for example select $\delta = 2$, then we can split an initial subarea and its (potential) children, but children of the children can only be extended. A value of $\delta = 0$ means that we are not allowing any *Split* operations. The reasoning behind this splitting depth is that we first want to narrow down the refinement area and then increase the level where needed. This approach can be quite effective but it also has the problem that the optimal depth δ might vary throughout the domain and it is another parameter that needs to be optimized first. Trying various parameters might not be feasible for some applications. In our test cases, we found that typically the best depth is in the region $\delta \in [0, 4]$ depending on how localized the function is.

Due to the aforementioned problems, we also created an automated system that chooses the operation by estimating the benefit of the different operations. To get a fair comparison, we need to start for a subregion $\boldsymbol{i}$ at a common *reference state*. From this reference state, we have to be able to get to the current state of $\boldsymbol{i}$ by applying an *Extend* and a *Split* operation. We therefore use the reference region parent($\boldsymbol{i}$) with local level $\ell_{\boldsymbol{i}} - 1$. By performing a *Split* and an *Extend* in arbitrary order, we get to the current state $\boldsymbol{i}$ and $\ell_{\boldsymbol{i}}$. This idea can be seen in Fig. 4.12. It should be noted that such a *reference state* always exists as we initially split the domain and we start with a level of 2.

We can now define the benefit of doing each operation. For this we measure how close we come to the final state with the *Extend* or the *Split* operation starting from the *reference state*. We use relative errors as the higher degree variants later have different target integrals for the *Split* and *Extend* operation [4].

If we apply a *Split* operation to the *reference state*, we result in the current subarea but with a decreased local level. We therefore define the integral after the *Split* by $q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}} - 1)$. This gives us the error estimate

$$\epsilon_{\boldsymbol{i}}^{\text{Sp}} = \left| \frac{q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}}) - q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}} - 1)}{q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}})} \right|. \tag{4.14}$$

In case we apply the *Extend* operation to the *reference state*, we obtain the same level as the current subarea but are still at the parent region. Hence, we can use for the error of the *Extend* all previous approaches to we compare the region $\boldsymbol{i}$ to its parent region. Here it is possible to use the approaches with $I_{\boldsymbol{i}}^{\text{old,sp},1/2/3}$ from one of the three described approaches before or alternatively the scheme for high degree methods that we will describe later. A key aspect is here that we do not just want to take the integral at the parent region with local level $\ell_{\boldsymbol{i}}$ which comes from refining the *reference state* once with an *Extend*, but we want to look at refining it $k \in \mathbb{N}$ times. The reason for that is that the increase in the number of points is typically more drastically with an *Split* than with an *Extend*. Even if we adjust the estimate by dividing through the number of points, the first refined points will always achieve a bigger absolute benefit than later ones. We therefore have to do multiple *Extend* operations to match the number of points as closely as possible to get a fair comparison. Consequently, we have the options

[4] For the linear basis one can also neglect the division.

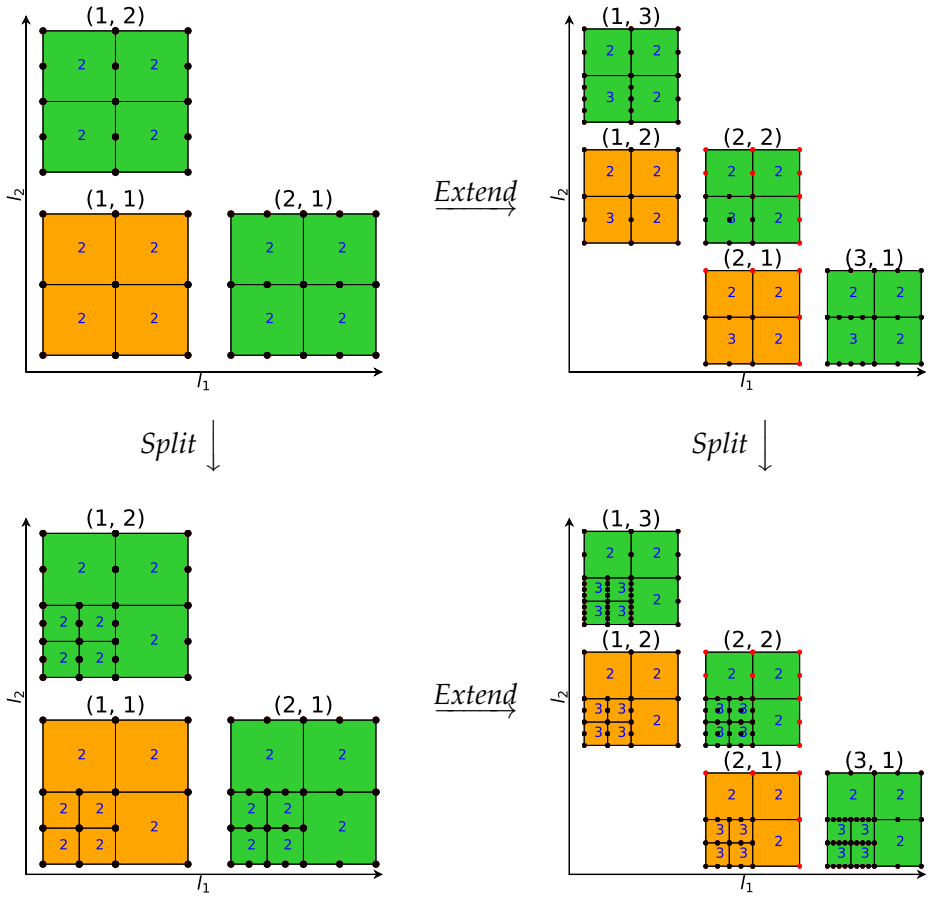


Figure 4.12: Visualization of the *reference state* for the automatic refinement (top left) that can be transformed to current state (bottom right) by applying an *Extend* and *Split* operation to the respective region. Any of the lower left final subareas could be the currently considered region $\boldsymbol{i}$. We therefore also show the *Extend* operation for all of these regions.

$$\epsilon_{\boldsymbol{i}}^{\text{Ex}} = \left| \frac{q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}}) - \tilde{I}_{\boldsymbol{i}}^{\text{old,sp}}}{q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}})} \right| \tag{4.15}$$

with

$$\tilde{I}_{\boldsymbol{i}}^{\text{old,sp,k,1}} = \Pi(q(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}} - 1 + k), \boldsymbol{i}),$$
$$\tilde{I}_{\boldsymbol{i}}^{\text{old,sp,k,2}} = q(\Gamma(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}} - 1 + k), \boldsymbol{i}, \ell_{\boldsymbol{i}}), \text{or}$$
$$\tilde{I}_{\boldsymbol{i}}^{\text{old,sp,k, 3}} = q(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}} - 1 + k)/|\text{children}(\text{parent}(\boldsymbol{i}))|.$$

Again, we use a combination of $\tilde{I}_{\boldsymbol{i}}^{\text{old,sp,k,1}}$ and $\tilde{I}_{\boldsymbol{i}}^{\text{old,sp,k,2}}$ where we select the integral approximation that produces the lower error. For k we choose the smallest k value that fulfills

$$3 \cdot \#\text{points}(\Pi(q(f, l_{\boldsymbol{i}} - 1 + k, \text{parent}(\boldsymbol{i})), \boldsymbol{i})) \geq n_{\ell_{\boldsymbol{i}}-1,\boldsymbol{i}}$$

with #points counting the used point values. This has shown to be a good heuristic which generates usually the largest number k with

$$\#\text{points}(\Pi_{\boldsymbol{i}}(l_{\boldsymbol{i}} - 1 + k, \text{parent}(\boldsymbol{i}))) = n_{\boldsymbol{i}}^{\text{ex}} < n_{\boldsymbol{i}}^{\text{sp}} = n_{\ell_{\boldsymbol{i}}-1,\boldsymbol{i}}.$$

Finally, we choose the optimal operation that has a lower adjusted error α. These operation-specific errors are

$$\alpha_{\boldsymbol{i}}^{\text{ex}} = \epsilon_{\boldsymbol{i}}^{\text{ex}} \cdot (n_{\boldsymbol{i}}^{\text{ex}} - n_{\boldsymbol{i}}^{\text{ref}}) \tag{4.16}$$

for the *Extend* operation and

$$\alpha_{\boldsymbol{i}}^{\text{sp}} = \epsilon_{\boldsymbol{i}}^{\text{sp}} \cdot (n_{\boldsymbol{i}}^{split} - n_{\boldsymbol{i}}^{\text{ref}}) \tag{4.17}$$

for the *Split* operation, where $n_{\boldsymbol{i}}^{\text{ref}} = n_{\ell_{\boldsymbol{i}}-1,\text{parent}(\boldsymbol{i})}$.

All of these error estimates stay constant and are only recomputed whenever the global combination scheme changes due to an increase in the global level ℓ (see Section 4.2.3) or whenever the local level $\ell_{\boldsymbol{i}}$ of a subarea $\boldsymbol{i}$ is changed.

4.2.4.2 Higher order methods

So far all estimates were designed with a linear basis in mind. The interpolation option for the surrogate $\tilde{f}$ could be adjusted to higher orders but can not match the quadrature degree in all cases, such as for Gaussian quadrature. We therefore have to adjust the error estimators.

We will first consider the general error estimate for each subarea. If the last operation has been a *Split* operation, we sum up all integral contributions from all children of the parent and compare this sum to the parent value. This error estimate is then equally split among all children. The error estimate in these cases is therefore

$$\epsilon_{\boldsymbol{i}} = \frac{|q(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}}) - \sum_{\boldsymbol{j} \in \text{children}(\text{parent}(\boldsymbol{i}))} q(f, \boldsymbol{j}, \ell_{\boldsymbol{i}})|}{|\text{children}(\text{parent}(\boldsymbol{i}))|}. \tag{4.18}$$

In case the last operation has been an extend, we proceed as before with

$$\epsilon_{\boldsymbol{i}} = |q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}} - 1) - q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}})|. \tag{4.19}$$

In addition, we have to change the error estimates for the automated selection of the operations. For the benefit of the *Extend* operation, we use

$$\begin{aligned}\epsilon_{\boldsymbol{i}}^{\text{Ex}} = &\left| \frac{q(f, \text{parent}(\boldsymbol{i}), \ell_{\text{parent}(\boldsymbol{i})} - 1 + k) - \sum_{\boldsymbol{j} \in \text{children}(\text{parent}(\boldsymbol{i}))} q(f, \boldsymbol{j}, \ell_{\text{parent}(\boldsymbol{i})} - 1 + k)}{\#\text{children}(\text{parent}(\boldsymbol{i})) \cdot \sum_{\boldsymbol{j} \in \text{children}(\text{parent}(\boldsymbol{i}))} q(f, \boldsymbol{j}, \ell_{\text{parent}(\boldsymbol{i})} - 1 + k)} \right| \\ &+ \left| q(f, \boldsymbol{i}, \ell_{\text{parent}(\boldsymbol{i})} - 1 + k) - q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}}) \right|.\end{aligned} \tag{4.20}$$

Here, we select the local levels from the parent node that were used before the last *Split* was performed as the current state could have gone through many *Extend* operations already. Since high degree methods, such as Gaussian quadrature, are often not nested, we would waste valuable function evaluations by using the local level $\ell_{\boldsymbol{i}}$. This is problematic as the error estimation should not add new function evaluations. We therefore have to avoid such evaluations by all means. The first term stays constant if *Extend* operations are applied to the region $\boldsymbol{i}$ while the last term measures how far away we are with the reference from the current state with current local level $\ell_{\boldsymbol{i}}$. k is this time selected slightly more conservative as the smallest k that fulfills

$$2 \cdot \#points(\ell_{\text{parent}(\boldsymbol{i})} - 1 + k, \text{parent}(\boldsymbol{i})) \geq n_{\ell_{\boldsymbol{i}}-1,\boldsymbol{i}}.$$

For the *Split* operation we also change the error estimate by using the parent level. This results in the error estimate

$$\epsilon_{\boldsymbol{i}}^{\text{Sp}} = \left| \frac{q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}}) - q(f, \boldsymbol{i}, \ell_{\text{parent}(\boldsymbol{i})} - 1)}{q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}})} \right| \tag{4.21}$$

with $n_{\boldsymbol{i}}^{\text{Sp}} = n_{\ell_{\text{parent}(\boldsymbol{i})}-1,\boldsymbol{i}}$. Another change is that we do not subtract $n_{\boldsymbol{i}}^{\text{ref}}$ for the calculation of α if the high degree methods do not reuse the points from the reference. An example of such a method would be again the Gaussian quadrature.

4.2.4.3 Splits in Single Dimensions

So far we have seen how to choose a subarea for refinement and what operation to apply. If we select the *Split* operation, there is, however, the possibility to only split in certain dimensions. For this we need another error estimate that can be activated if necessary. This work is based on a collaboration with Vivian Haller as part of his Bachelor's thesis [47].

Before we can define the error estimate, we need some definitions. In addition to the relation *parent* and *children*, we add the term *twin*. A *twin* in dimension k is the cell which is the direct neighbour in dimension k that originates from the same parent, i.e.

$\boldsymbol{j} = \text{twin}(i) \Rightarrow \text{parent}(\boldsymbol{i}) = \text{parent}(\boldsymbol{j})$. If we only Split in single dimensions, a parent only has 2 children. We can then define the twin error in dimension k as

$$\epsilon_{\boldsymbol{i}}^{\text{twin},k} = \left| q(f, \text{parent}(\boldsymbol{i}), \ell_{\boldsymbol{i}}) - (q(f, \boldsymbol{i}, \ell_{\boldsymbol{i}}) + q(f, \text{twin}(\boldsymbol{i}), \ell_{\boldsymbol{i}})) \right|. \tag{4.22}$$

These error estimates stay constant and are only recalculated for the new twins in dimension k once a single-dimensional *Split* in dimension k is performed. This is also the reason why all local levels in the equation are equal. All twin errors from other dimensions $\tilde{k} \neq k$ are inherited from the parent if a single-dimensional *Split* is performed in dimension k. The only change is that the inherited error estimates are equally *Split* between the children, i.e. $\forall \tilde{k} \in [d], \tilde{k} \neq k : \epsilon_{\boldsymbol{i}}^{twin,\tilde{k}} = 0.5 \cdot \epsilon_{\text{parent}(\boldsymbol{i})}^{twin,\tilde{k}}$. For efficiency reasons, it is also possible to select multiple dimensions at once for this *Split* if the error estimates are close. Initially, we calculate for all 2^d subareas $\boldsymbol{i}$ all twin errors by calculating all 2^{d-1} parent integrals explicitly. This gives us right from the start the possibility to decide which dimension we should split first.

With these error estimates, we can now decide which subareas need to be refined and which operation should be performed on this subarea. The next section will present the whole workflow of the Split-Extend method.

4.2.5 Overall algorithm

In Algorithm 8, we show the complete procedure for computing a d-dimensional numerical integral of a function f in the domain $[\boldsymbol{a}, \boldsymbol{b}]$ with the Split-Extend method. After initializing the combination scheme (see Section 4.2.1), we iterate over all component grids. For each of the component grids, we go through all subareas and compute the integral contribution of the subarea. Here, m (see Section 4.2.3) is mapping the level vector $\boldsymbol{\ell}$ to $\tilde{\boldsymbol{\ell}}$ according to the local level of the subarea. The resulting integral values are combined together and summed up for all subareas. We then stop the refinement if the final integral is close enough to the reference solution or if a maximum number of points is exceeded. *Close enough* is defined in respect to the analytic integral error or via the global error estimate $\epsilon = \sum_{\boldsymbol{i} \in \text{refinement}} \epsilon_{\boldsymbol{i}}$ that sums up all local errors. This global error ϵ is then be compared against a pre-defined tolerance. If none of these stopping criteria are fulfilled, we continue the computation by refining new subareas.

The refinement procedure starts by selecting the areas with maximum local error estimate $\epsilon_{\boldsymbol{i}}$ (see Section 4.2.4). In our case, we first determin the maximum error $\epsilon^{\max} = \max_{\boldsymbol{i} \in \text{refinement}} \epsilon_{\boldsymbol{i}}$ and refine all subareas with $\epsilon_{\boldsymbol{i}} \geq \gamma \cdot \epsilon^{\max}$. In our experiments with the Split-Extend method, we set $\gamma = 0.9$. The chosen subareas are then refined. If we select the depth based refinement, the number of previous *Split* operations determines if a subregion is *Split* further or if we perform an *Extend*. Otherwise the automated process is used as described in Section 4.2.4 and we choose the operation that has the lower α value. For a *Split* operation we can first select the dimensions that should be split (see Section 4.2.4.3) or refine every dimension equally. All children inherit the local level from their parent. In contrast to this, the *Extend* increases the local level and adjusts the global combination scheme if the new local level $\ell_{\boldsymbol{j}}$ exceeds the global level

Algorithm 8 Pseudocode for the Split-Extend method

```
1: procedure SPLIT_EXTEND_COMBI(a, b, tol, ref, f, max_points)        ▷ Output: integral
                                                                      ▷ Initialization
2:     ℓ = 2, τ_global = −1
3:     generate combination_scheme using l_global and τ_global
4:     create 2^d children i with l_i = 2 and τ_i = −1 as initial subareas
5:     while true do
6:         integral = 0
7:         for combiGrid in combination_scheme do
8:             for i in refinement.NEW_AREAS() do
9:                 ℓ = combiGrid.level_vector
10:                ℓ̃ = m(ℓ, ℓ_i)
11:                if ℓ̃ not negative and not duplicate then
12:                    integral += grid.INTEGRATE(f, i, ℓ̃, a_i, b_i) ·c_ℓ
13:                end if
14:            end for
15:        end for
16:        ϵ = |(integral-reference)/ref|
17:        if ϵ < tol or number_of_points > max_points then
18:            break
19:        end if
20:        γ = 0.9
21:        subareas = refinement.GET_AREAS_WITH_MAX_BENEFIT(γ)
22:        for i in subareas do                                       ▷ Refinement procedure
23:            α_i^ex, α_i^sp = GET_OPERATION_ERRORS(i)
24:            if α_i^ex > α^sp then                                  ▷ Split: split i in selected dimensions
25:                dimensions = GET_DIMENSIONS_FOR_SPLITTING(i)
26:                children = SPLIT(i, dimensions)
27:                for j in children do
28:                    ℓ_j = ℓ_i
29:                end for
30:            else                                                   ▷ Extend: increase local level
31:                j = EXTEND(i)
32:                l_j = l_i + 1
33:                if ℓ_j > ℓ then
34:                    combi_scheme = UPDATE_SCHEME(l_j, combi_scheme)
35:                    ℓ = ℓ_j
36:                end if
37:            end if
38:        end for
39:    end while
40: end procedure
```

ℓ. This process is repeated until either the maximum number of points is exceeded or the tolerance is reached.

The complexity of the algorithm is dominated by the grid integration as all error estimations are based on cached integrals or are computed by integrating a constant number of integrals with a less or equal number of points compared to the subarea integral. All other operations only iterate through the subareas which are far less than the number of total points. The resulting complexity at refinement step n is therefore

$$\mathcal{O}\left(\sum_{\boldsymbol{\ell}\in\mathcal{I}_{\ell^{(n)}}}\sum_{\boldsymbol{i}\in\text{refinement}}\#\text{points}(\ell_{\boldsymbol{i}},\boldsymbol{i})\right)$$

where $\ell^{(n)}$ is the global level at iteration n. This linear complexity per step is optimal. In addition, it is only necessary to recalculate the integrals for new subareas that have been added by the refinement procedure. In this case, we do not set the integral to 0 at the beginning of the while loop and subtract the integral contributions of the removed subareas after the refinement procedure. Only if the global level ℓ changes it might be necessary to recalculate all integral values for all subareas. Hence, the refinement steps are usually less costly than with the dimension-wise scheme as most parts of the calculation can be reused. However, reusing old values might depend on the chosen application. For application with global dependencies between points, it is usually not possible to reuse values in regions that have not changed.

4.3 Implementation overview

In this section, we present the main aspects of our implementation for the previously explained spatially adaptive methods. As there were no suited frameworks available, a new Python-based framework `sparseSpACE`[5] was created which allows for fast prototyping. `sparseSpACE` is designed to test new application areas for the spatially adaptive combination schemes presented in Sections 4.1 and 4.2 and allows to compare them to the standard Combination Technique. The framework supports arbitrary operations on the generated grids that can be implemented via black box solvers which operate directly on the component grids. In addition, various strategies to map the grid points to the domain can be selected such as equidistant grids or Gaussian quadrature points. These features guarantee a high flexibility of the framework for prototyping new applications and testing them with the different variants of the Combination Technique. The framework offers multiple tutorials for the different application scenarios and uses continuous integration to ensure code quality. In the following we will describe in more detail the design of `sparseSpACE`.

[5] https://github.com/obersteiner/sparseSpACE

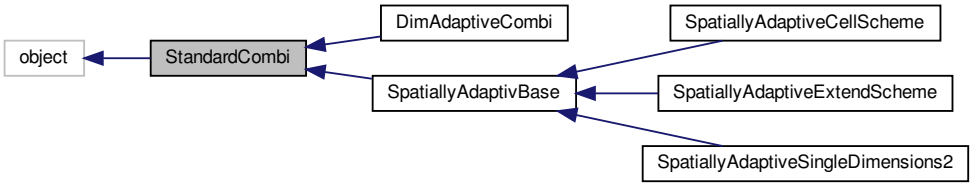


Figure 4.13: Class hierarchy of the different Combination Technique approaches generated with doxygen [3].

4.3.1 Combination scheme

One of the key elements of the Combination Technique is the combination scheme. This combination scheme represents the used index set $\mathcal{I}$ that includes the level vectors, and the combination coefficients $c_{\boldsymbol{\ell}}$ for $\boldsymbol{\ell} \in \mathcal{I}$. Therefore, the class `CombiScheme` was implemented that offers the creation of a combination scheme for a fixed $\boldsymbol{\ell}^{\max}$ and $\boldsymbol{\ell}^{min}$. For a default Combination Technique of level ℓ, we have $\boldsymbol{\ell}^{\min} = \mathbf{1}$ and $\boldsymbol{\ell}^{\max} = \ell \cdot \mathbf{1}$. Another feature of the class is that it offers the adaptation of the index set according to the dimension-adaptive algorithm (see Section 2.3.3.1). Here, active and old index sets are generated and updated during the adaptation. A getter function returns the index set $\mathcal{I}$ and the combination coefficients $c_{\boldsymbol{\ell}}$ of the current combination scheme.

4.3.2 Implementation of the Combination Technique approaches

In `sparseSpACE` we want to offer a wide variety of versions for the Combination Technique to compare and test their applicability for several applications. To facilitate the implementation, we implemented a base class `StandardCombi` from which all adaptive approaches inherit (see Fig. 4.13). The base class implements common interfaces such as a getter function for the number of points used in the combination or their position. It also allows to compute the standard Combination Technique with a selected operation such as numerical quadrature. The standard Combination Technique uses a fixed level and generates via the `CombiScheme` the corresponding combination scheme. The `StandardCombi` class also implements most of the plotting routines that are used in this thesis. By overwriting specific getters the plotting can be customized for child classes.

The child classes are split into the dimension-adaptive version `DimAdaptiveCombi` and `SpatiallyAdaptiveBase`. The `DimAdaptiveCombi` class mainly uses the functionalities of the `StandardCombi` but it updates the combination scheme according to the scheme described in Section 2.3.3.1. For stopping the adaptive process a tolerance or a maximum number of evaluations is passed. `SpatiallyAdaptiveBase` serves as a base class for all the implemented spatially adaptive strategies. It incorporates the

general workflow of adapting the refinement structures and offers a new method to start an spatially adaptive execution of the Combination Technique. This class is designed quite flexible so that different spatially adaptive schemes can be implemented as child classes using the *Strategy pattern*.

For the different spatially adaptive implementations, we offer the `SpatiallyAdaptiveCellScheme`, the `SpatiallyAdaptiveExtendScheme`, and the `SpatiallyAdaptiveSingleDimension2`. The `SpatiallyAdaptiveCellScheme` class implements our own version of the previously reported spatially adaptive Combination Technique from [83] (see Section 2.3.3.2). Here we split the domain into cells and apply on each cell a separate quadrature or interpolation. As this scheme does not have any component grids, each cell has to manage its own combination scheme including its ancestor cells. The adaptation process is slightly different than in [83] as it incorporates for each cell the scheme from Gerstner and Griebel [40] outlined in Section 2.3.3.1. Since these combination schemes are local, the approach achieves spatial adaptivity. During the adaptive process, we interpolate parent cells to get an error estimate for quadrature and interpolation. Since we have no component grids, this aproach does not support any other operations apart from interpolation and quadrature. For more details see [83] and the implementation in `sparseSpACE`.

`SpatiallyAdaptiveSingleDimension2` implements the dimension-wise refinement (see Section 4.1) that performs the Combination Technique on rectilinear grids. It generates according to the 1D point sets for each level vector the respective points of the rectilinear grid. These grids are then passed to the operation. The refinement is guided by the errors estimates which are calculated with the hierarchical surplus values. Here the operation transfers the calculated point values to start the error estimation. Due to the flexibility of the surplus calculation and the easy interface with rectilinear grids, this is the approach which can be extended most easily to new applications.

`SpatiallyAdaptiveExtendScheme` represents the Split-Extend scheme from Section 4.2. Here we offer the splitting of the domain, and the increase of the local levels via the *Split* and *Extend* operations. The implementation generates for each subarea its own level vector and applies the specified operation to it. In case the black box solver needs a grid that covers the complete domain, it is possible to aggregate all points from all subareas and to apply the operations once on the complete grid. The operation is also used to calculate the error estimates. These estimates a currently fixed for the quadrature operation but it is possible to extend the scheme to add further operations.

4.3.3 Grid operations

So far, we have seen the modules that are responsible for the creation of the combination scheme and the (adaptive) component grids. The `GridOperation` represents the next logical step as it defines the operation that is performed on the grids. Here we have already implemented a variety of operations such as `Integration`, `Interpolation`, `Uncertainty Quantification`, `DensityEstimation` and `PDESolve`.

Again, we utilize the *Strategy pattern* with the base class `GridOperation` that specifies the interfaces that need to be implemented to allow the use of the different combina-

tion types. These operations need to support the respective grids that are constructed by the used Combination Technique approach. Thus, the operations needs to be adjusted for every (adaptive) combination variant that we want to use. In particular, the operation needs to implement the evaluation of a regular grid for the standard and dimension-adaptive Combination Technique. For the dimension-wise refinement we need to be able to process rectilinear grids and for the Split-Extend scheme we require either an operation that operates on regular subareas or that is able to process block-adaptive grids. Additionally, each operation needs to specify means of calculating local error estimates if default ones are not sufficient or not available. These error estimates can be based on surplus values or application-related such as misclassification rates in an Machine Learning scenario.

In this section we have only focused on the main implementation ideas. For more details on the different operations and the interfaces to implement them, we refer to the description of the numerical results in Chapter 5 and the code base[6].

4.3.4 Functions

The `Function` class is used for integrating and interpolating functions. Here, a base class implements the general interface and functionalities such as caching function values to avoid recalculation. Via *Strategy pattern* arbitrary functions can be implemented that only need to implement an evaluation method that returns the function value for a given input. An analytic integral solution can be added to allow the framework to exactly calculate the quadrature errors. If this analytic solution is not implemented, the error estimates are used to measure the global error.

In general all function evaluations are calculated single-threaded but it is possible to start any parallel job inside the evaluation method. In this case it is necessary to either wait for the job to finish or pre-calculate all necessary function values at the beginning of a computation step. The caching will then avoid recalculation. It is also possible to enhance the evaluation speed by implementing a vectorized evaluation that calculates function values for multiple evaluation points at once.

The framework supports already various functions such as the Gentz test functions [38] and was successfully coupled to complex solvers such as Vadere[7] [68] and Larsim[8] [78] for Uncertainty Quantification calculations.

4.3.5 Different grid types

In general the Combination Technique only specifies level vectors for each component grid but not a specific mapping of points or even the point numbers that correspond to a certain level. We therefore need a class that defines how to map levels to actual point numbers and positions. The `Grid` class offers this functionality.

[6] https://github.com/obersteiner/sparseSpACE
[7] http://www.vadere.org/
[8] https://www.larsim.info

Here, we use once again a *Strategy pattern*. The base class represents the interface that is used by the adaptive algorithms. The grid returns the number of points, the positions and quadrature weights that are associated with a level vector. Additionally, the grids can be configured to include or exclude boundary points or use a modified basis. It therefore provides several helper functions for the adaptive algorithms. Examples for implemented grids are equidistant grids, such as the `TrapezoidalGrid` or `SimpsonGrid`, and other nested or non-nested grids, such as `LejaGrid`, `ClenshawCurtisGrid`, `BSplineGrid`, `LagrangeGrid` and `GaussLegendreGrid`. These grids are typically named according to the quadrature rule they are associated with. However, they can also be used in conjunction with other operations.

It is also possible to define point positions from outside by passing point sets and levels. This is used in the `GlobalGrids` which are only relevant for the dimension-wise refinement. Examples are the `GlobalTrapezoidalGrid` or the `GlobalSimpsonGrid` that we will see later in Chapter 5.

4.3.6 Refinement Container

The refinement information of the spatially adaptive methods is encapsulated inside of a `RefinementContainer` to allow for implementing different data structures for accessing this data. This refinement information is specific to the used algorithm and stores a collection of `RefinementObject` instances. Currently we store the objects in a list but different data structures such as trees or hash maps are possible. The container can therefore easily be exchanged with new data structures that give faster access times or more efficient data layouts if the algorithm requires this.

The `RefinementObject` stores the essential information that is needed for each individual refinement step of the spatially adaptive algorithms. For `SpatiallyAdaptiveCellScheme` it saves for each cell the boundaries, parent infos and further information to generate the cell hierarchies. In the `SpatiallyAdaptiveExtendScheme` a refinement object saves the local level and the subarea domain. Finally, for `SpatiallySingleDimension2` the refinement container stores all point sets with their levels. In addition to the custom information each `RefinementObject` stores the local error estimate that is used for deciding which objects to refine. It also offers a refinement method to automatically generate refined objects. This can be a split into further objects or an update of a information such as the local level.

4.3.7 Wrapper for Machine Learning

In addition to the base functionalities, we added in the course of [82] a wrapper for the framework that offers a clear Machine Learning interface to the framework. This wrapper uses the `DensityEstimation` operation to calculate the density of a given data set. Based on this density either classification or clustering can be performed.

For classification a dataset is split according to their class assignments and a separate density is created for each class. As a consequence, multiple instances of the framework

are created. A new data point is classified by first evaluating all densities of the different classes and then it is assigned to the class with maximal density.

For clustering the density estimation is used to cut edges in a k-nearest neighbor graph. Edges who cross areas of low densities are removed from the graph. The remaining connected components represent the found clusters. This results in a good reconstruction of clusters.

In this section, we only gave a brief overview of the implementation of the Machine Learning wrapper. The algorithmic and mathematical details will be explained in Section 5.3.1. Further details on the implementation can be found in [82].

5 Numerical case studies with the Spatially Adaptive Combination Technique

In the previous chapter, we have seen how to generate and refine a spatially adaptive Combination Technique. As a next step, we need to test these methods for various settings to analyze the benefit of the new approaches. For this purpose, we performed several case studies in different application scenarios that are commonly used with the Combination Technique. We introduce each application area briefly and present results for the methods that are most suited for the specific use case.

In particular, we will look at classical numerical tasks such as quadrature and interpolation in Section 5.1. We also investigate the influence of different basis functions and quadrature rules. Thereafter, we will quickly summarize our results for uncertainty quantification which makes extensive use of quadrature operations to calculate statistical properties of a model function. Then, we will look at machine learning applications. Here, we will focus on Sparse Grid density estimation which can be used for both supervised and unsupervised learning. We introduce different modes how to perform this density estimation and analyze the results for different classification scenarios.

5.1 Numerical quadrature and interpolation

In numerical quadrature and interpolation we use input-output pairs $((\tilde{\boldsymbol{x}}, f(\tilde{\boldsymbol{x}}))$ for $\tilde{\boldsymbol{x}} \in \mathcal{X} \subset \mathcal{D})$ of an (unknown) function to either compute a numerical integral or to estimate function values at arbitrary points within the domain $\mathcal{D}$.

In general the quadrature problem $q(f)$ is defined by

$$q(f) = \sum_{\tilde{\boldsymbol{x}} \in \mathcal{X}} w_{\tilde{\boldsymbol{x}}} \cdot f(\tilde{\boldsymbol{x}}) \approx \int_{\boldsymbol{x} \in \mathcal{D}} f(\boldsymbol{x}) d\boldsymbol{x} \tag{5.1}$$

with quadrature weights $w_{\boldsymbol{x}}$ for point $\boldsymbol{x}$. The result of the quadrature is the integral approximation for f.

The interpolation problem is defined by

$$s(\boldsymbol{x}) = \sum_{\tilde{\boldsymbol{x}} \in \mathcal{X}} a_{\tilde{\boldsymbol{x}}} \cdot \Phi_{\tilde{\boldsymbol{x}}}(\boldsymbol{x}) \approx f(\boldsymbol{x}), \forall \tilde{\boldsymbol{x}} \in \mathcal{X} : s(\tilde{\boldsymbol{x}}) = f(\tilde{\boldsymbol{x}}), \tag{5.2}$$

with basis functions $\Phi_{\tilde{\boldsymbol{x}}}(\boldsymbol{x})$ associated with points $\tilde{\boldsymbol{x}}$ and their scaling $a_{\tilde{\boldsymbol{x}}}$. The result is the surrogate $s(\boldsymbol{x})$ that can be evaluated at arbitrary points $\boldsymbol{x} \in \mathbb{R}^d$.

For both methods we require the function evaluations $f(\tilde{\boldsymbol{x}})$ at the points $\tilde{\boldsymbol{x}} \in \mathcal{X}$. For high dimensions these methods quickly run into the *curse of dimensionality* as they

usually sample these points on regular grids. Here, high dimensional grids are often constructed via a tensor product structure of one-dimensional grids.

We will look at rectangular domains $\mathcal{D} = [\boldsymbol{a}, \boldsymbol{b}]$ with the domain boundaries $\boldsymbol{a}$ and $\boldsymbol{b}$. Since we apply the combination technique, both problems are slightly transformed to

$$q(f) = \sum_{\boldsymbol{\ell} \in \mathcal{I}} c_{\boldsymbol{\ell}} \cdot \sum_{\tilde{\boldsymbol{x}} \in \mathcal{X}_{\boldsymbol{\ell}}} w_{\tilde{\boldsymbol{x}}} \cdot f(\tilde{\boldsymbol{x}}) \approx \int_{\boldsymbol{x} \in \mathcal{D}} f(\boldsymbol{x}) d\boldsymbol{x} \tag{5.3}$$

and

$$s(\boldsymbol{x}) = \sum_{\boldsymbol{\ell} \in \mathcal{I}} c_{\boldsymbol{\ell}} \cdot \sum_{\tilde{\boldsymbol{x}} \in \mathcal{X}_{\boldsymbol{\ell}}} a_{\tilde{\boldsymbol{x}}} \cdot \Phi_{\tilde{\boldsymbol{x}}}(\boldsymbol{x}) \approx f(\boldsymbol{x}), \forall \boldsymbol{\ell} \in \mathcal{I} \, \forall \tilde{\boldsymbol{x}} \in \mathcal{X}_{\boldsymbol{\ell}} : s(\tilde{\boldsymbol{x}}) = f(\tilde{\boldsymbol{x}}), \tag{5.4}$$

where $\mathcal{X}_{\boldsymbol{\ell}}$ represents the set of grid points from component grid $\boldsymbol{\ell}$. This could be a regular grid for the standard and dimension-adaptive combination, a rectilinear grid for the dimension-wise refinement, or a block-adaptive grid for the Split-Extend scheme.

We will consider the following test functions:

$$f_{corner}(\boldsymbol{x}) = \left(1 + \sum_{i=1}^{d} k_i \cdot x_i\right)^{-d-1}$$

$$f_{prod}(\boldsymbol{x}) = \frac{10^{-d}}{\prod_{i=1}^{d} (k_i^{-2} + (x_i - p_i)^2)}$$

$$f_{discont}(\boldsymbol{x}) = \begin{cases} 0, & \text{for } \boldsymbol{x} \geq \boldsymbol{p} \\ e^{-\sum_{i=1}^{d} k_i \cdot x_i}, & \text{otherwise} \end{cases}$$

$$f_{cont}(\boldsymbol{x}) = e^{-\sum_{i=1}^{d} k_i \cdot |x_i - p_i|}$$

$$f_{gauss}(\boldsymbol{x}) = e^{-\sum_{i=1}^{d} k_i \cdot (x_i - p_i)^2}$$

$$f_{expvar}(\boldsymbol{x}) = (1 + 1/d)^d \cdot (\prod_{i=1}^{d} x_i^{1/d})$$

The first five functions are common test funtions in high dimensional numerics that were published in the Gentz test functions package [38] and the last one is taken from [39, 81] where the authors used it to analyze a function with exponential variation. For all of these methods an analytic solution is available which makes it easy to investigate the convergence properties for our novel approaches and compare them to existing methods. Parts of the results shown in this section were already published in [84] and [85].

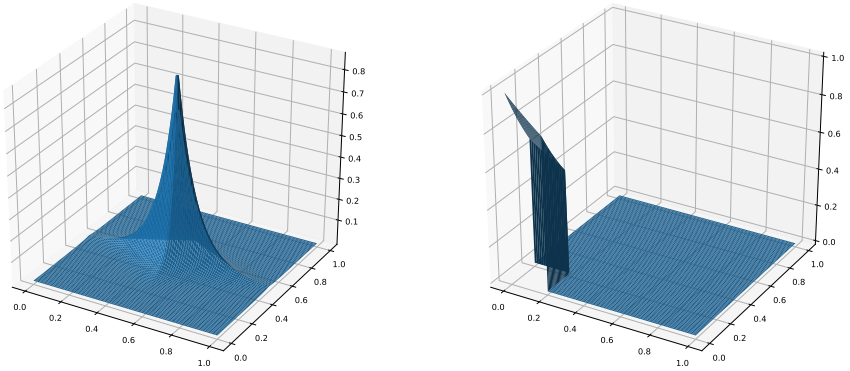


Figure 5.1: Plot of the continuous and discontinuous Gentz function.

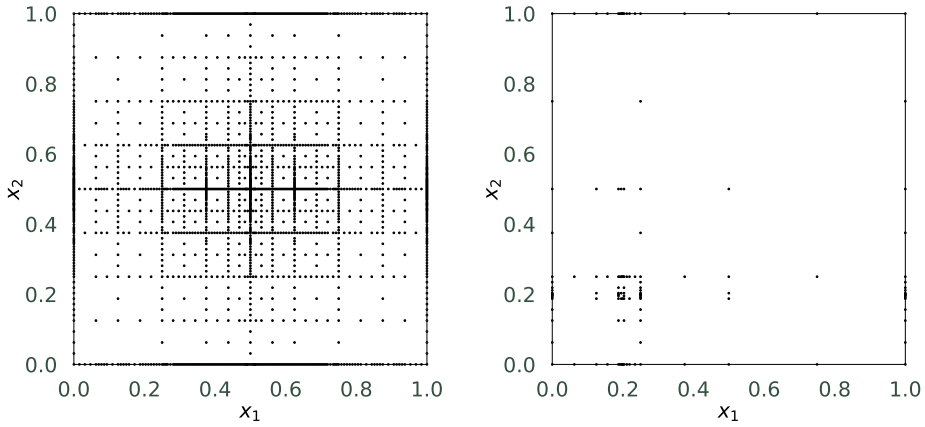


Figure 5.2: Resulting Sparse Grid of the Dimension-wise Spatially Adaptive Combination Technique for f_{cont} (left) with $k_i = 10i$, $p_i = 0.5$, and 1873 points, and f_{discont} (right) with $k_i = i$, $p_i = 0.2$, and 139 points. We refined up to a relative error tolerance of 10^{-2} with the linear hat basis and enabled tree rebalancing.

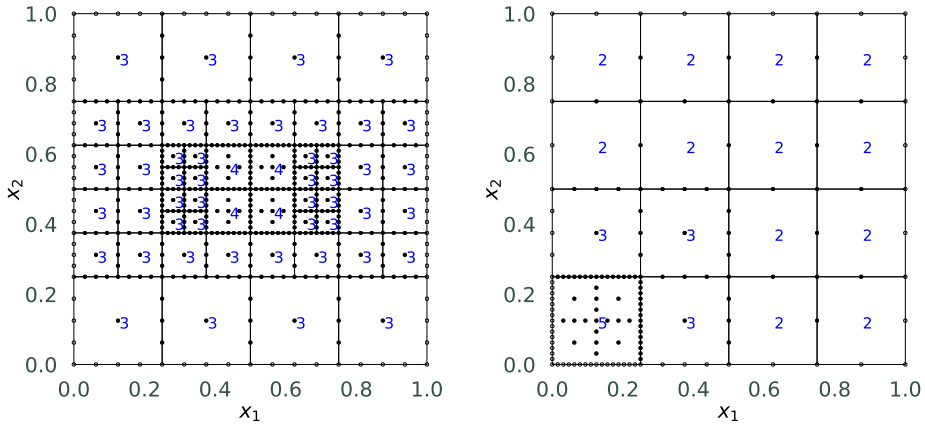


Figure 5.3: Resulting Sparse Grid of the Split-Extend method for f_{cont} (left) with $k_i = 10i$, $p_i = 0.5$, and 545 points, and $f_{discont}$ (right) with $k_i = i$, $p_i = 0.2$, and 157 points. In both cases we refine up to a tolerance of 10^{-2} and use a linear basis with Newton-Cotes grid and the automated selection of *Extend* and *Split* operations. The local level of each subarea is shown in blue.

5.1.1 Visual inspection

Before we go into a more detailed convergence analysis, we will look visually at the refinement of the different adaptive approaches and discuss the point sets they use for two exemplary test functions (see Fig. 5.1). We can see that the dimension-wise as well as the Split-Extend method can adapt themselves to the function at hand. It is clearly visible that the points are concentrated at the most important areas.

We have chosen these two test functions to show the key differences for the two methods. For the continuous function a large part of the x and y axis are relevant for the correct representation of the function while for the discontinuous case only the are between 0 and 0.2 for both dimensions is most relevant. As a consequence the dimension-wise method performs better for the discontinuous case as the relevant 1D regions for each dimension are directly influencing the performance. In the worst case a dimension would need equal refinement along the whole dimension which would eliminate any benefit for spatial refinement in this dimension. For the continuous function this can be partly seen as here large areas within each dimensions are relevant. For a correct representation only the 1D intervals close to the boundaries are irrelevant. Therefore, the points concentrate towards the center but are still covering a large area.

In contrast to that the Split-Extend methods performance is not depending on the important 1D intervals but on the actual d-dimensional subarea of the domain that needs extensive refinement. Since both functions are close to 0 on most parts of the domain, it performs well for both scenarios. It should be noted that the Split-Extend scheme in this case needs to perform a high number of *Split* operation to get to a sufficiently small region that can be used to efficiently represent the function. This is not a problem but actually shows that the important regions are very localized. There is only one slight problem with the continuous function. Since the important regions are on the boundaries between different subareas (at 0.5 in each dimension), we have to perform more *Split* operations to get a sufficient fine-grained refinement. This effect is less predominant if the important regions are for example in the corner of the domain.

As both methods were refined until a relative error threshold of 0.01 was reached, we can directly compare point numbers to investigate the effectiveness of the two methods. Here, we see that for the continuous case the Split-Extend method performs better as it can refine in the 2D space and not just via 1D refinements. In contrast to this, for the discontinuous case the 1D refinements are more efficient as the 1D splits perform very well here. This is probably caused by the fact that the dimension-wise method does not need to increase the truncation parameter which results in a more efficient Sparse Grid approximation. This effect is, however, usually more predominant for rather smooth functions as we will see in the next section.

5.1.2 Convergence analysis

We have seen that at least visually the refinement seems to work as expected. The next step is to look at the actual convergence plots and verify our visual analysis. Here, we consider different basis functions starting with linear approximations. We will then

consider higher order approximations such as Simpson and Gaussian quadrature. In addition we will discuss the modified linear basis with its benefits for higher dimensional cases and the single-dimensional *Split* operation of the Split-Extend scheme. We show results for numerical integration and also discuss interpolation for the linear basis. We restrict ourselves to five-dimensional test cases since they represent a sufficiently complicated problem class that cannot be efficiently tackled with full grid methods.

5.1.2.1 Linear basis

In the first test case, we look at the linear hat basis that is frequently used in classical Sparse Grid literature. We present results for numerical quadrature as well as interpolation.

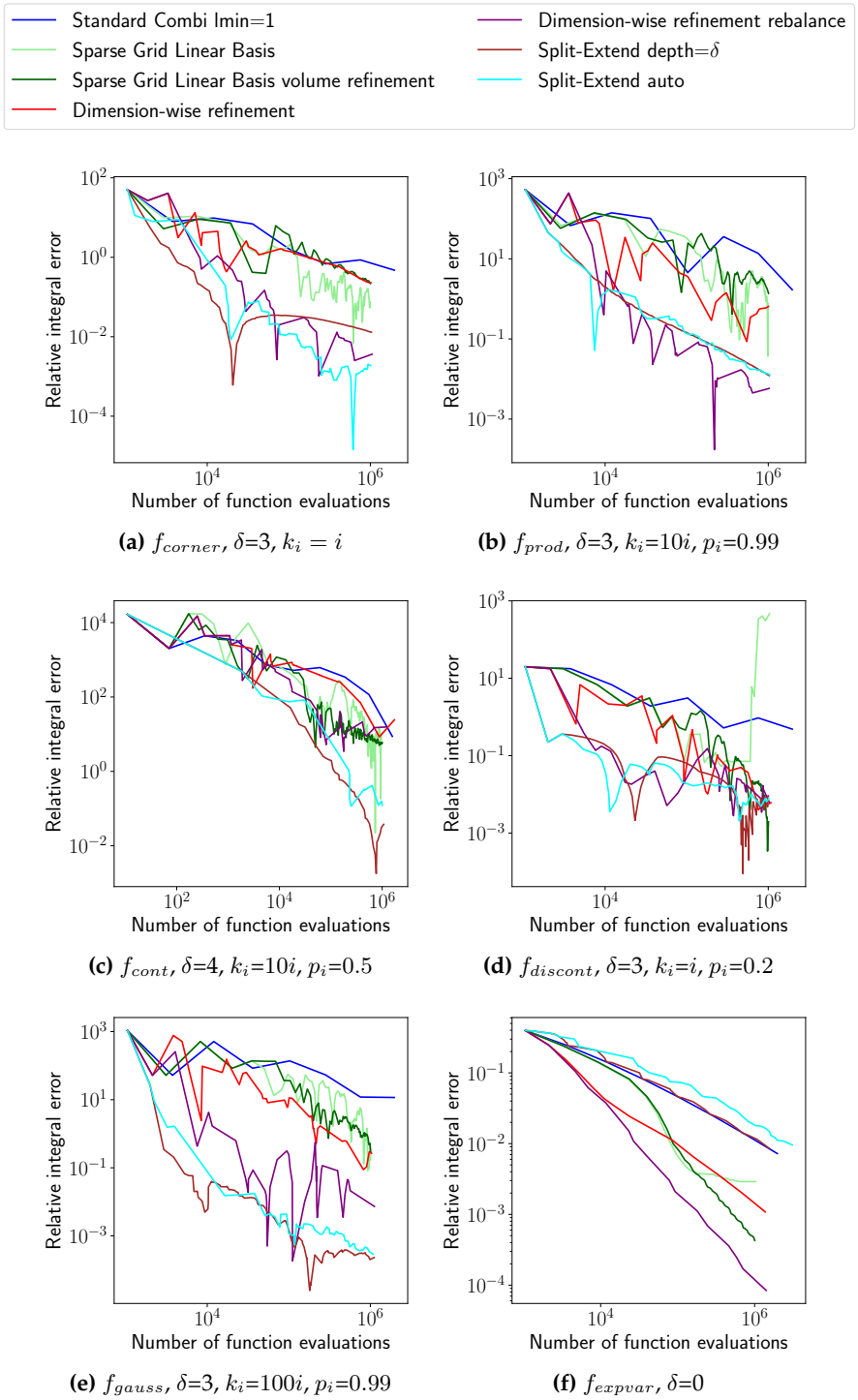
Quadrature First, we will look at the quadrature results. In figure Fig. 5.4 we show the evolution of the relative quadrature error with increasing function evaluations for the six different test functions in 5D. One can see the results for the standard Combination Technique (blue) with $\ell_{\min} = 1$, two variants of the classical spatially adaptive Sparse Grid quadrature with surplus or volume-based refinement that are implemented with an explicitly Sparse Grid discretization[1] in SG^{++} (light and dark green), the dimension-wise spatially adaptive Combination Technique with and without rebalancing (red and purple), and the Split-Extend method with fixed depth and automatic adaptation (brown and cyan). We use boundary points for all test cases except for the function f_{cont} as here the function values at the boundary are negligible small.

In general, the standard Combination Technique performs the worst due to its inability to adapt to the concrete test functions which benefit from an adaptive increase of points in specific areas of the domain. Only for f_{expvar} the Split-Extend method achieves worse results than the standard Combination Technique.

The dimension-wise refinement seems to perform especially well with the rebalance variant. This variant persistently outperforms the method without rebalancing. Moreover, the method seems to perform especially well for rather smooth functions such as f_{expvar}. Unfortunately the rebalancing method shows a more erratic convergence which is probably caused by the rebalancing operations. A rebalancing operation offers the possibility to refine more efficiently in future refinements but also disturbs the current refinement.

For the Split-Extend method, we observed that the automatic selection of *Split* and *Extend* operations can usually perform similarly as an explicit selection of a splitting depth δ. Furthermore, the method seems to perform especially well for functions that are very localized and are therefore not so smooth. The reason for this observation is that the *Split* operation restricts the negative effect of non-smooth regions to a small subarea. Hence, large portions of the domain can converge rather quickly and require consequently less grid points.

[1] These variants therefore do not use the Combination Technique.



(a) f_{corner}, δ=3, $k_i = i$

(b) f_{prod}, δ=3, k_i=10i, p_i=0.99

(c) f_{cont}, δ=4, k_i=10i, p_i=0.5

(d) $f_{discont}$, δ=3, k_i=i, p_i=0.2

(e) f_{gauss}, δ=3, k_i=100i, p_i=0.99

(f) f_{expvar}, δ=0

Figure 5.4: Relative quadrature error for the six test functions with $d = 5$ for the linear basis with trapezoidal quadrature. We compare the new dimension-wise refinement versions to the standard Combination Technique, explicit Sparse Grid implementations, and the Split-Extend scheme.

These observations show that the dimension-wise scheme and the Split-Extend method complement each other since they perform well in opposite use cases. This can also be detected via the splitting depth δ. For functions with an optimal splitting depth ≥ 3, the Split-Extend method easily outperforms the dimension-wise scheme. In cases were splitting does not improve the performance the dimension-wise scheme has the upper hand. This is not surprising as the spatial adaptivity of the Split-Extend scheme is a result of the *Split* operations. With few *Splits* the method provides only rather coarse refinement.

If we look at the classical spatial adaptivity with $\texttt{SG}^{++}$, we can see that it performs better than the standard Combination Technique. Surprisingly, in all of our test cases either the Split-Extend or the dimension-wise method can significantly outperform the pure Sparse Grid approach. This is a surprise as the spatially adaptive schemes with the Combination Technique offer less flexibility compared to the pure Sparse Grid implementation. It seems that either this flexibility can mislead the refinement or that the rebalancing and splitting can improve the Sparse Grid representation in our test cases.

Overall, we can see that with a suited spatially adaptive refinement, we can gain around two orders of accuracy for the integral calculation with the same amount of function evaluations in comparison to a standard Combination Technique.

Interpolation Next, we will look at the interpolation results for the same test cases. We interpolated the functions on the regular grid $[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]^5$. The interpolation error is then calculated for the L_2 and L_∞ norm via

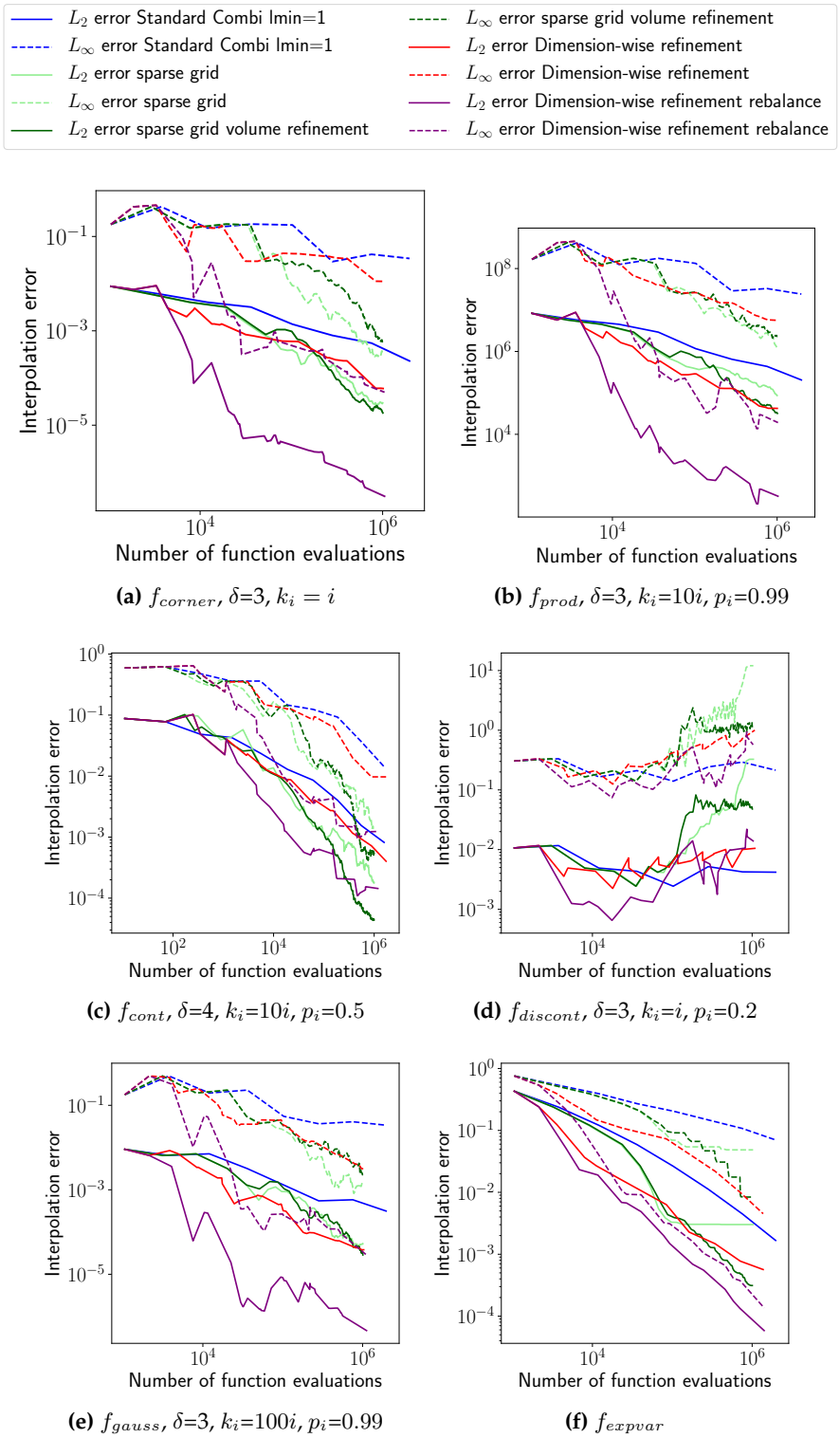
$$\epsilon_{L_2} = \frac{1}{\sqrt{|S|}} \cdot \sqrt{\sum_{\boldsymbol{x} \in S} (f(\boldsymbol{x}) - \tilde{f}(\boldsymbol{x}))^2} \text{ and } \epsilon_{L_\infty} = \max_{\boldsymbol{x} \in S} |f(\boldsymbol{x}) - \tilde{f}(\boldsymbol{x})|,$$

where $\tilde{f}(\boldsymbol{x})$ is the evaluation of the generated adaptive Combination Technique model at the point $\boldsymbol{x}$ that is constructed by the linear interpolation of the function evaluations with the linear hat basis.

In Fig. 5.5 we show the interpolation errors for the standard Combination Technique (with $\boldsymbol{\ell}_{\min} = \mathbf{1}$), the classical adaptive Sparse Grid approach with $\texttt{SG}^{++}$, and the dimension-wise scheme with and without rebalancing.

First of all, we can see that the results are qualitatively the same for both norms. This indicates that not just the maximum error but also the accumulated errors across the domain behave similarly with all schemes. This is beneficial as for different applications different norms might be relevant.

If we look at the convergence of the single approaches, we can see that – similarly to the quadrature results – the adaptive approaches perform significantly better than the standard Combination Technique. Furthermore, the rebalancing variant again outperforms the standard dimension-wise refinement significantly. Interestingly, the dimension-wise scheme with rebalancing clearly surpasses in all test cases the adaptive Sparse Grid approach. This holds even for f_{cont} where the quadrature results performed similarly. This indicates that the dimension-wise approach delivers very good interpolation



(a) f_{corner}, δ=3, $k_i = i$

(b) f_{prod}, δ=3, k_i=10i, p_i=0.99

(c) f_{cont}, δ=4, k_i=10i, p_i=0.5

(d) $f_{discont}$, δ=3, k_i=i, p_i=0.2

(e) f_{gauss}, δ=3, k_i=100i, p_i=0.99

(f) f_{expvar}

Figure 5.5: Interpolation errors for the six test functions with $d = 5$ and the linear basis. The L_∞ errors are represented by dotted lines and the L_2 errors by straight lines. We compare the new dimension-wise refinement versions to the standard Combination Technique and pure Sparse Grid implementations.

results and seems to refine efficiently in the correct regions. Again, we see a more erratic convergence for the rebalancing approach.

An outlier to these observations is f_{discont}. Here, the errors do not seem to converge at all. This turned out to be a result of the chosen interpolation points, since 0.2 is exactly at the position of the discontinuous step. The linear model is therefore heavily disturbed at this point as the grids do not have an evaluation point here. This results in an almost constant error term that can not be reduced due to the refinement. Consequently, the errors do not decrease which can be observed in the error plot. Similarly, the classical adaptive Sparse Grid approaches do not converge and might even get unstable which can be seen with the classical surplus refinement (light green).

Our tests show that the conclusions from the interpolation results are in general similar to the quadrature results. We will therefore consider only quadrature in the following part.

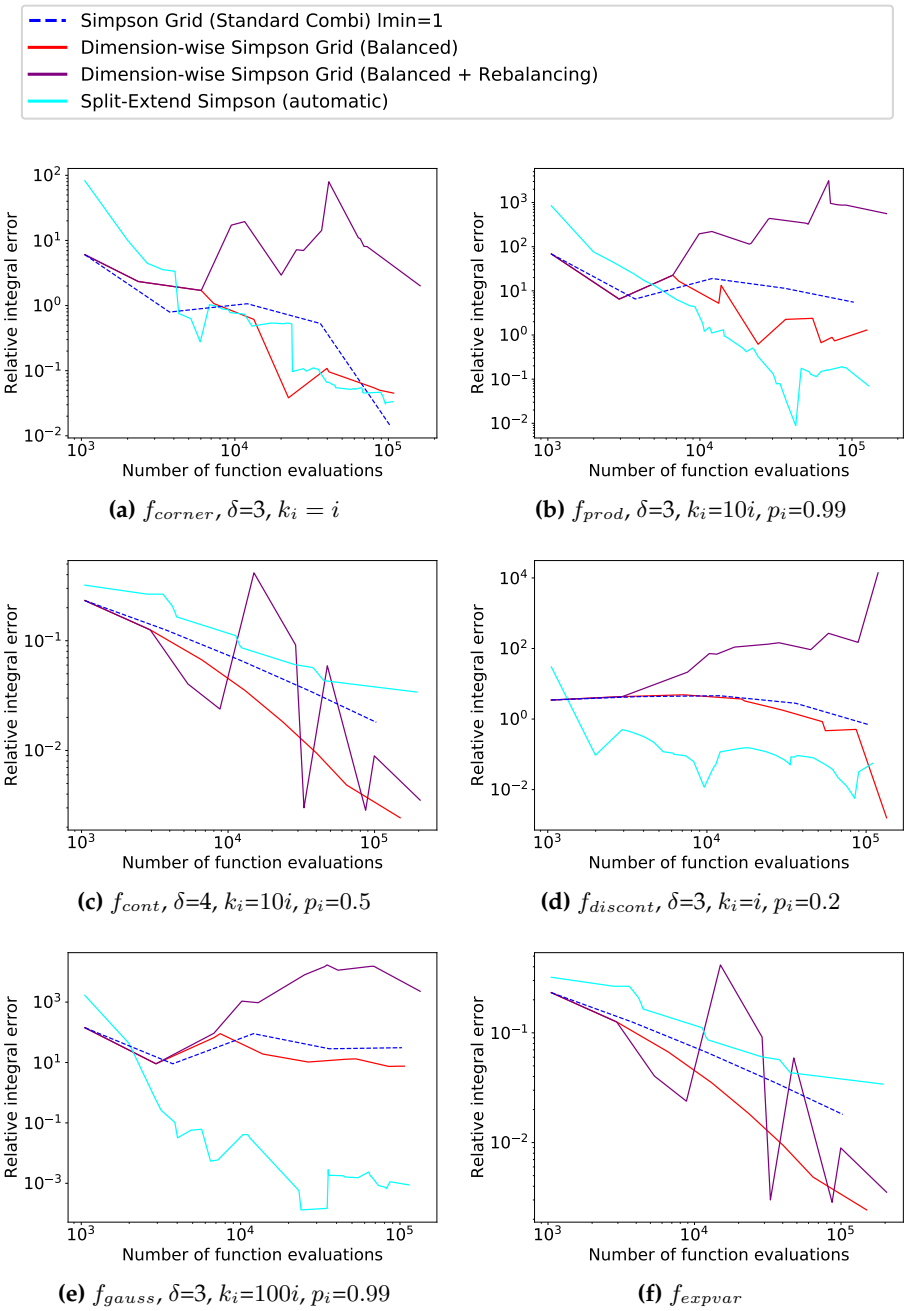
5.1.2.2 Quadratic approximation

So far, we have seen the convergence results for the linear hat basis. In many cases we can profit from higher order basis functions to get higher accuracies with lower function evaluations. We will first look at second order approximations. In this case, Simpson's rule is used to calculate the integrals. Again, we compare the standard Combination Technique, the dimension-wise scheme and the *Split-Extend* method (only with automatic refinement). It should be noted that here we use boundary points for f_{cont}.

In addition, we use a balanced version of the dimension-wise refinement that always adds the second child to a node in the 1D point hierarchy of a component grid if it is missing. The resulting binary tree has therefore the specific structure that every node has either two or zero children. The only exception to this are the nodes at level 0 that have always only one child. If a component grid lacks one of these children, the balancing just adds the respective point. The reason why we have to add these points with the balanced approach is that the resulting 1D point sets have a guaranteed point number that is odd, which is a requirement for Simpson's quadrature. Furthermore, the quadrature results are often better with the balanced approach as normally only positive quadrature weights appear[2]. This balancing routine is different to the rebalancing procedure (see Section 4.1.3) which tries to balance the depth in the global refinement hierarchies and not to fix the number of ancestors in the individual component grids.

In general, for the smooth functions the standard Combination Technique performs better with the quadratic approximation compared to the linear basis from before. Also the dimension-wise scheme seems to profit from the higher degree. This can be seen especially with f_{expvar}. In contrast to this, the Split-Extend scheme seems not to achieve better results compared to the linear basis. This indicates that the Split-Extend method mainly benefits from the *Split* operations, which are independent of the basis functions, and not from the increased approximation order of the basis functions.

[2]Only the rebalancing approach can cause negative weights as the support points of the parabolas might become non-equidistant.



(a) f_{corner}, δ=3, $k_i = i$

(b) f_{prod}, δ=3, k_i=10i, p_i=0.99

(c) f_{cont}, δ=4, k_i=10i, p_i=0.5

(d) $f_{discont}$, δ=3, k_i=i, p_i=0.2

(e) f_{gauss}, δ=3, k_i=100i, p_i=0.99

(f) f_{expvar}

Figure 5.6: Relative quadrature error for the six test functions with $d = 5$ for quadratic basis with simpson quadrature. We compare the new dimension-wise refinement versions to the standard Combination Technique, pure Sparse Grid implementations and the Split-Extend scheme from [84].

Surprisingly, the rebalancing method seems to be rather unstable for higher order basis functions and either fluctuates a lot or even diverges. A reason for this might be that without rebalancing the spacing between the points of each parabola in Simpson's rule is always equidistant, i.e. the three support points lie one an equidistant subpart of the grid. In case we perform a rebalancing of the point levels, this property is lost. We therefore assume that the resulting quadrature weights tend to decrease the performance which can be due to an increased condition number caused by negative weights.

The general observations are however similar to the linear basis. Again we can benefit from spatial adaptivity and the dimension-wise scheme performs best for smoother functions while the Split-Extend method performs better for the non-smooth functions.

Another observation is that especially for the dimension-wise method the functions that are not twice continuously differentiable perform bad with the Simpson rule. This was expected as higher order can usually not be reached if the function is not sufficiently differentiable. An example for this phenomenon is f_{discont}. f_{cont} represents an exception to this rule. The reason for the good convergence here is that the function is only not differentiable whenever $x_i = 0.5$ for any dimension i. Since we however always split the parabolas at 0.5 this is not a problem as the subintervals for which we use the quadratic approximation are always twice differentiable[3]. This does not hold necessarily for the rebalancing approach which might cause the fluctuations.

To put it in a nutshell, we can see that especially the dimension-wise scheme benefits from a second order approximation, while the Split-Extend scheme mostly achieves the high accuracy through the *Split* operations and not because of the higher order approximations.

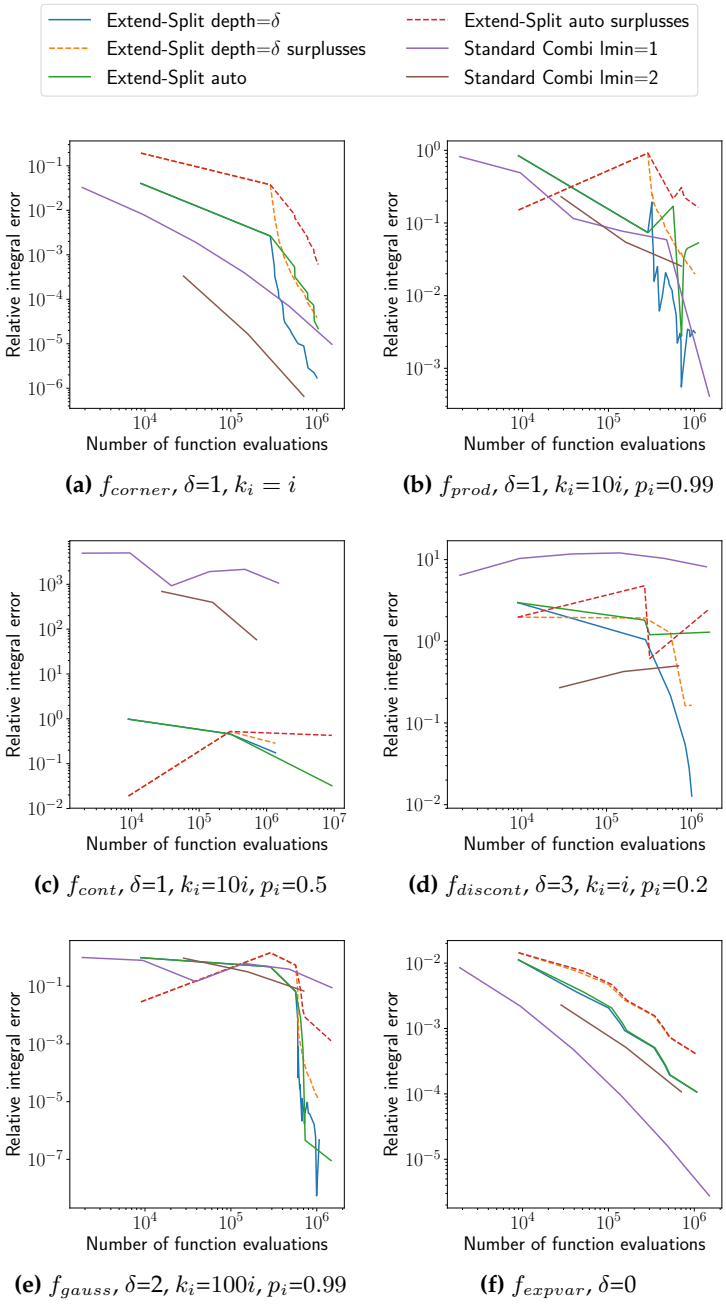
5.1.2.3 Gaussian Quadrature

In addition to the trapezoidal and simpson quadrature, we tested the Gaussian quadrature for our adaptive approaches. Here, we can only use the Split-Extend scheme as the adaptive refinement of the 1D point hierarchies would not be possible with a classical Gaussian quadrature scheme. In each subarea of the Split-Extend scheme, a separate Gauss quadrature with corresponding local level is performed. Again we look at the same six test functions.

In Fig. 5.7 we show our results. We compare the Split-Extend scheme with predefined depth (blue) and automatic refinement (green) and the standard Combination Technique with $\boldsymbol{\ell}^{\min} \in \{1, 2\}$ (purple and brown). In addition, we show the sum of all error estimates that act as a measurement for the global error. This sum can be compared to the actual error to assess the quality of our approximation.

For the smooth function f_{expvar} the adaptive process performs worse than the standard Combination Technique, while for the other functions, which are less smooth and more localized, the adaptive process usually works better. The former is caused by the (initial) *Split* operations that reduce the number of grid points per subarea and therefore reduce the reachable polynomial degree. This limits the efficiency significantly.

[3] A parabola always covers the interval of three adjacent points in our one-dimensional point sets. Since the point with position 0.5 has always an odd index, it is always at the boundary of these intervals.



(a) f_{corner}, δ=1, $k_i = i$ **(b)** f_{prod}, δ=1, k_i=10i, p_i=0.99

(c) f_{cont}, δ=1, k_i=10i, p_i=0.5 **(d)** $f_{discont}$, δ=3, k_i=i, p_i=0.2

(e) f_{gauss}, δ=2, k_i=100i, p_i=0.99 **(f)** f_{expvar}, δ=0

Figure 5.7: Convergence tests for the six test functions with $d = 5$ for Gauss-Legendre quadrature. The dotted lines represent the error estimate that is given by the summation of all surplus estimates ϵ_i for the respective adaptive algorithm.

The latter observation is especially dominant for the non-differential functions (f_{cont} and f_{discont}) and the very localized function f_{gauss}. This indicates that the high quadrature degree cannot be reached due to the missing differentiability or that we are still in a preconvergence state. Here, the *Split* operation can improve the results by narrowing down the refinement areas and by refining separately in the differentiable subareas. Of course the refinement can not exactly find the non-differentiable steps but it can at least reduce their influence by limiting their impact to small subregions. This is interesting if the non-differentiable regions and discontinuities are very localized. We can therefore reach fast convergence in the remaining subareas.

If we compare the automatic refinement with the pre-defined depth, we can see that in some cases they are very similar while in others the automatic refinement clearly performs worse. Consequently, the automatic refinement needs to be further optimized to be a real alternative to finding the optimal splitting depth δ.

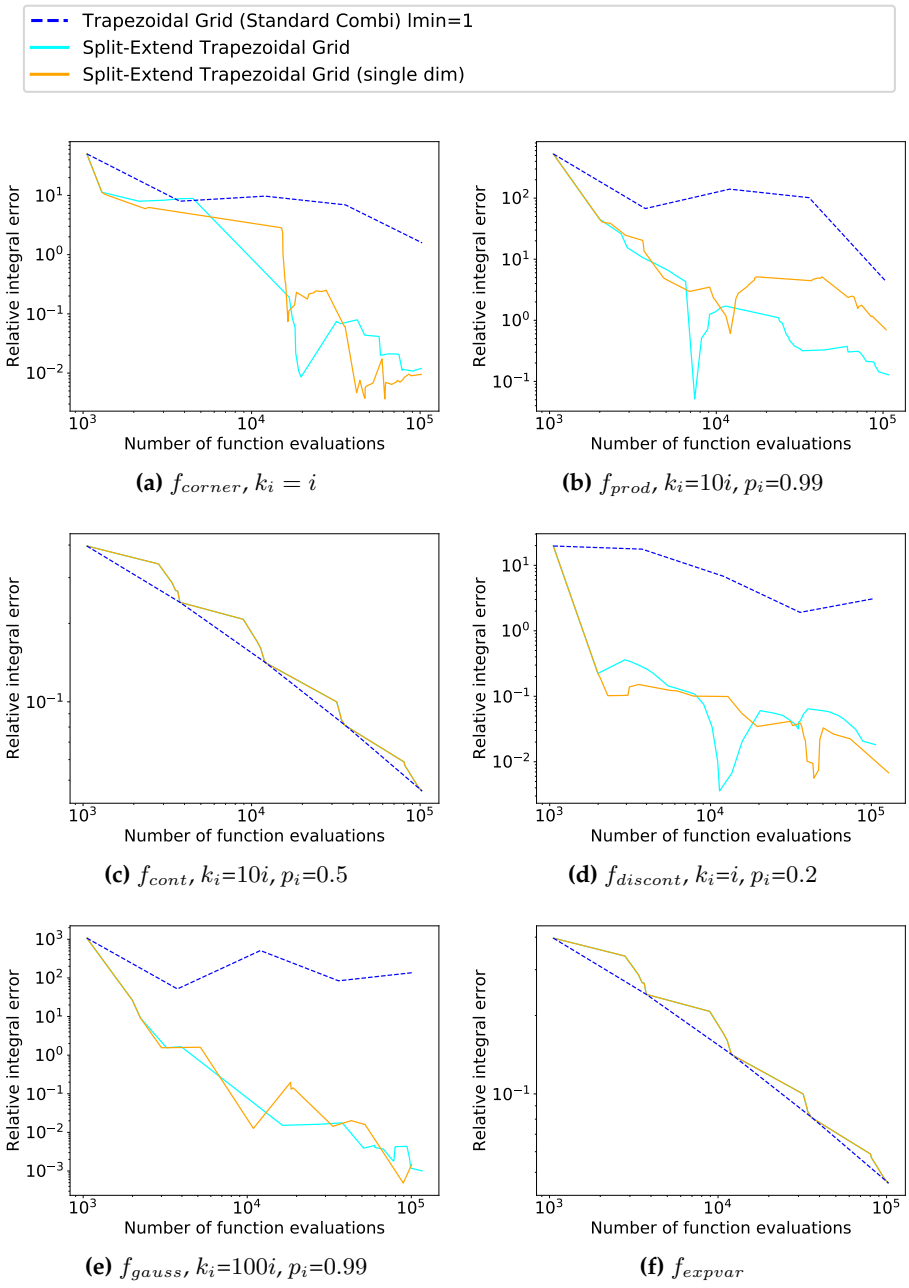
The error estimates for the global error seem to be in some cases very inaccurate at low evaluation numbers but follow the trend of the real error for higher numbers. Unfortunately, the estimates usually significantly overestimate the error and can therefore not directly approximate the real error for the general use case. This is not a problem for the refinement procedure itself which only needs a relative error estimate, but this behavior makes it more difficult to decide when to stop the refinement. It will require further research to improve this global error estimate in order to use it as a robust error approximation.

To conclude this analysis, we can say that especially for functions that usually perform bad with standard gaussian quadrature, we can significantly increase the performance with the Split-Extend scheme. Here, the *Split* operation helps to limit the influence of non-differentiable subareas and can help to find regions faster that need intensive refinement. In cases where the standard Gauss quadrature performs well the *Split* operation reduces the overall degree which decreases the performance. We should therefore only use the adaptive method for cases that cannot be tackled with the standard approach.

5.1.2.4 Single-dimensional splits

We have also implemented the single-dimensional *Split* operation that we described in Section 4.2. In this variant, the main difference is that an area is not divided into 2^d equally sized subareas but that it is only split in any number k of dimensions resulting in 2^k children. In the extreme case we only split in one dimension resulting in 2 children areas.

In Fig. 5.8 we show for the six test functions the results with and without this single-dimensional split and compare it to the standard Combination Technique. We can see that in general both variants for the *Split* perform similar. Sometimes the single-dimensional *Split* performs better and sometimes slightly worse. This indicates that either we do not profit from single-dimensional *Split* operations or that our test functions do not show enough variations between the dimensions so that we do not profit from a separate treatment of the dimensions.



(a) f_{corner}, $k_i = i$ **(b)** f_{prod}, k_i=$10i$, p_i=0.99

(c) f_{cont}, k_i=$10i$, p_i=0.5 **(d)** $f_{discont}$, k_i=i, p_i=0.2

(e) f_{gauss}, k_i=$100i$, p_i=0.99 **(f)** f_{expvar}

Figure 5.8: Relative quadrature error for the six test functions with $d = 5$ for linear basis with trapezoidal quadrature. We compare the standard *Split* of the Split-Extend method to single-dimensional *Split* operations. The standard Combination Technique is given as a reference.

.

It should be noted that the single-dimensional *Split* performs usually more refinement steps in total which increases the runtime when function evaluations are rather cheap. We should therefore avoid them if it does not provide a better approximation with fewer point evaluations.

Hence, we conclude that single-dimensional *Split* operations should only be used if the function shows significantly different characteristics between the dimensions. In case the dimensions are too similar, we usually do not improve the approximation enough to compensate for the increased number of refinement steps and could even end up with a worse representation. An improved error estimator that results in better single-dimensional *Split* operations might change this situation and could increase the performance and robustness of the method.

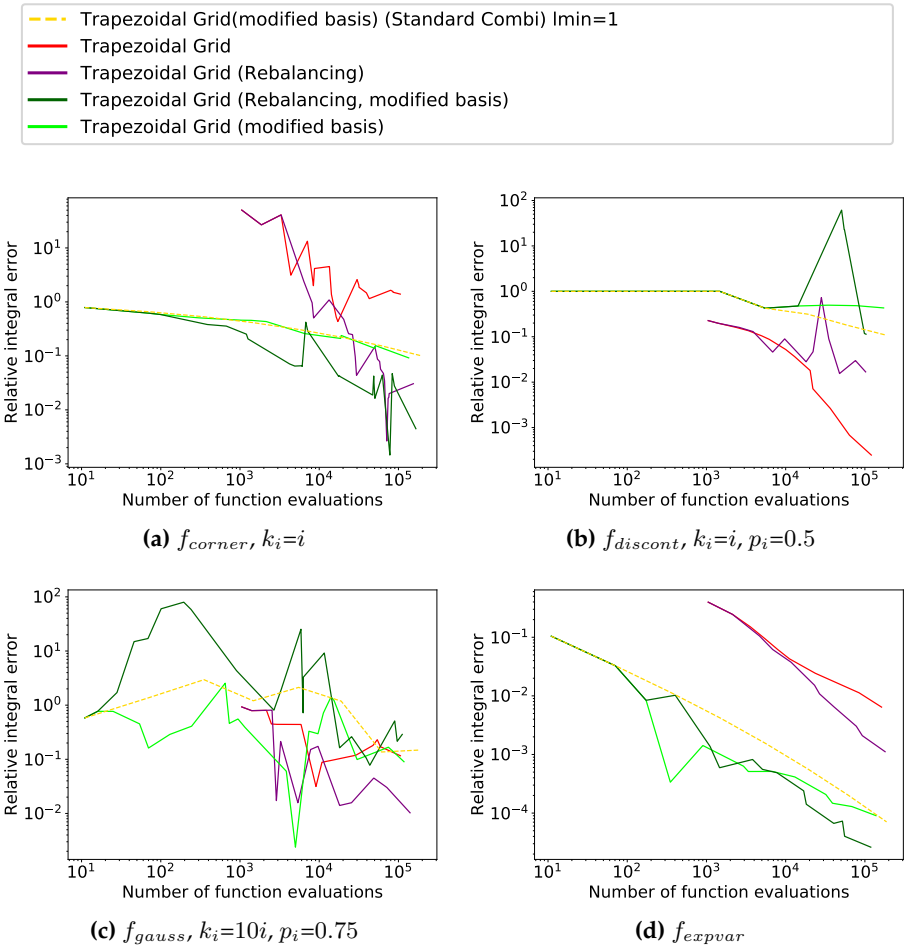
5.1.2.5 Modified basis

So far, we have mainly seen basis functions that are used when we either have points on the boundary or the boundary has a predefined value (usually 0). In cases where the boundary values are unknown and where we do not want to spent points on the boundary, a special type of basis functions is often used: the modified basis [96]. We will look at the linear modified basis which linearly extrapolates towards the boundary. For this purpose, we build the linear interpolation function between each of the two outermost points in each direction and their corresponding inner neighbour. This function is then extrapolated towards the respective boundary. If only one point is present in level 1, we use the constant function $\Phi(x) = 1$ as basis. This modification only changes the basis functions of the two outermost points and their direct neighbours in 1D. All other basis functions in the nodal basis remain unchanged. Via tensor product this basis definition then generalizes to higher dimensions. The surplus calculation is adjusted to the modified basis. More details on the surplus calculation and the concept of the modified basis can be found in [96].

With the modified basis, we adapted the original six test functions so that the local phenomena are not localized in the corners. The reason for that is that only for very large levels points are placed close to the corners. Hence, the results are bad for these cases and the adaptive process cannot find a suitable refinement due to the missing corner points. In addition, we removed the results for f_{cont} as it does not require extrapolation due to a boundary value very close to 0, and we also removed f_{prod} as it delivered results similar to f_{gauss}. Furthermore, some slight adjustments to the shape of the functions were made to increase their smoothness.

In Fig. 5.9 we show our results for the four test functions. We look at the (rebalanced) dimension-wise scheme with the modified basis (dark and light green) and compare it to the regular linear basis with boundary points (red and purple). We also compare these results to the standard Combination Technique with the modified basis (yellow).

We can see that for smooth function representations, such as f_{corner} and f_{expvar}, we can improve our results with the modified basis. Especially for f_{expvar} we achieve a significant boost of up to two orders for the standard Combination Technique in comparison to the normal linear basis with rebalancing.



(a) f_{corner}, k_i=i **(b)** $f_{discont}$, k_i=i, p_i=0.5

(c) f_{gauss}, k_i=$10i$, p_i=0.75 **(d)** f_{expvar}

Figure 5.9: Relative quadrature error for four test functions with $d = 5$ for the modified linear basis. We compare the new dimension-wise refinement versions to the standard Combination Technique, and the standard linear hat basis with boundary points.

The adaptive refinement also seems to work well for the rather smooth functions and fails for f_{discont} and f_{gauss}. In these cases the normal linear basis performs usually better and less erratic. If we look at the rebalancing variant, we see a mixed picture. For the test cases where the modified basis works well, we usually have a similar or better performance with the rebalancing option. However, in the other cases rebalancing might increase the errors further.

In conclusion, we can say that a higher smoothness is required with the modified basis compared to a regular basis with boundary points. However, the absence of boundary points has the potential to significantly decreases the number of function evaluations if these requirements are met. Hence, the modified basis should be preferred for all functions with sufficient smoothness. This holds especially for higher dimensions where boundary points become increasingly costly (see for example [116, chapter 2.4.1]).

5.1.3 Summary

We have seen that the spatially adaptive approaches for the Combination Technique can perform significantly better than the standard Combination Technique and can even outperform pure Sparse Grid implementations for the linear basis. In general, the dimension-wise refinement performs better for smoother functions and functions for which only a small spatial area per dimension needs extensive refinement. The Split-Extend method works best for functions that are not smooth or that are even not continuously differentiable. Here the *Split* operation helps to restrict the influence of such problematic regions which leads to a faster convergence in the remaining parts of the domain. Both methods therefore complement each other and could be used in conjunction. In fact, one could *Split* the domain and then apply a single dimension-wise refinement in each subarea. Such a method could combine both benefits but would also increase the need for a robust and efficient error estimate that guides the refinement and decides which operations to apply.

We have also seen that our adaptive approaches do not just work with the linear hat basis but can also be successfully adapted to higher order basis functions. Here, the regular structure with the Split-Extend method even allows for Gaussian quadrature in subregions. Again, we can outperform the standard Combination Technique for these higher order basis functions for sufficiently localized functions. This also holds for the modified basis which is a widely used variant that allows to remove boundary points which significantly decreases the grid point numbers in high dimensions.

In our analysis, we have only looked at the number of function evaluations and not at the time to compute the result. It is clear that for the used test functions the evaluations were not the most time consuming part of the computation but the actual adaptive refinement procedure. We therefore can not assume to always be faster if the number of function evaluations is lower. This especially holds if many refinement steps are necessary. However, in many real-world applications these function evaluations are extremely costly. Thus, they represent the most time-consuming part for such a quadrature or interpolation problem. Consequently, we can save a lot of computing time or

increase the accuracy significantly with the adaptive approaches due to the reduction in point evaluations.

It should be noted that we have mainly looked at very localized functions that have for example peaks at the corner or at the center of the domain. Hence, the results are not representative for all functions. This was intentional as we wanted to show the possibility to use the Combination Technique in such complicated cases for which the standard Combination Technique clearly fails. For smooth functions the schemes usually perform similarly to the classical Combination Technique[4] or can even be mislead in certain cases which can sometimes result in a lower efficiency. This is, however, a problem that is common to spatial adaptivity. However, for larger dimensionalities it might also be useful to apply the adaptive approach to smooth functions as the approaches are able to generate intermediate grids between two regular Sparse Grids of different levels. This might be beneficial if the next level already contains too many points.

The presented results are not all tests that we have been conducted so far. In [101] we have analysed more advanced extrapolation approaches – similar to Romberg's method [103] – for the dimension-wise refinement. Here, we could show that we can outperform the linear basis in multiple cases. We have also tried to increase the order of the dimension-wise scheme by using the method from [64] that can generate stable high order quadrature rules for arbitrary point distributions by reducing the maximal possible degree until the quadrature weights are positive. First results with this method showed for many test cases erratic behavior which might be caused by the fact that adding points in suboptimal regions might even decrease the quadrature degree. As a consequence, this often leads to stagnating or even increasing errors at high point numbers. Another option to increase the order of the dimension-wise scheme would be to utilize the piecewise Gauss quadrature from [16]. Such an approach would however prohibit the use of rebalancing operations as this conflicts with the optimized grid point placement.

Outside of the Sparse Grid community mainly randomized approaches, such as (quasi) Monte Carlo methods, are used to solve high-dimensional integrals. These approaches usually guarantee a convergence rate that is independent of the dimensionality of the problem. This is especially interesting for very high dimensional cases as the Combination Technique can not reach arbitrary high dimensions[5]. Another alternative is Bayesian quadrature that uses Gaussian processes for integrating the target function. A review of Bayesian quadrature methods can be found in [26]. For high-dimensional interpolation mainly irregular grids are used due to the *curse of dimensionality*. Examples are distance-based methods, such as nearest neighbor interpolation, the use of specialized basis function that are centered at arbitrary data points, such as radial-basis functions [21], or probabilistic approaches, such as Gaussian process regression [100].

[4] Of course some additional overhead is introduced due to the refinement procedure.

[5] The Combination Technique works usually best for dimensions smaller than 10-20 unless an efficient dimension reduction, e.g. with the dimension adaptive scheme, can be achieved.

For both use cases Gaussian processes represent an alternative to find an adaptive refinement by estimating the regions with highest uncertainty.

To put it in a nutshell, the spatially-adaptive variants of the Combination Technique represent an efficient alternative to existing methods in quadrature and interpolation. In particular, they can improve on existing Sparse Grid techniques for localized problems.

5.2 Uncertainty quantification

In this section, we give a short overview of *uncertainty quantification* (UQ) and summarize our results with `sparseSpACE`. For a more detailed description of the mathematical foundations and algorithms in this field we refer to [111] and [74].

In UQ the main goal is to determine the uncertainty of a model with respect to a set of uncertain or stochastic parameters. Here, we usually differentiate between deterministic parameters $\boldsymbol{x}$ and stochastic parameters $\boldsymbol{y}$. An exemplary model evaluation would be $u(\boldsymbol{x}, \boldsymbol{y})$. Sources for uncertainty in the parameter choice arise for example in input parameters that are real-world measurements with certain measurement errors, or model parameters that are fitted to match real world experiments.

The target of UQ is now to get the statistical properties of u for specific values of $\boldsymbol{x}$ with respect to the stochastic parameters $\boldsymbol{y}$. This could be the expectation, the variance, or the sensitivity with respect to changes in certain stochastic parameters. In our case, we assume a certain probability density distribution ϕ that defines the random distribution of the stochastic parameters $\boldsymbol{y}$. Popular examples are the normal distribution or the uniform distribution.

There are different methods to calculate these statistical quantities. One approach is to approximate the function $u(\boldsymbol{x}, \boldsymbol{y})$ via an analytic function and calculate the statistical properties exactly for this approximation. One specific incarnation of this idea is the *polynomial chaos expansion* [119] that uses the approximation

$$u(\boldsymbol{x}, \boldsymbol{y}) = \sum_{m=0}^{\infty} \hat{u}_m(\boldsymbol{x}) \Phi_m(\boldsymbol{y}) \approx \sum_{m=0}^{M} \hat{u}_m(\boldsymbol{x}) \Phi_m(\boldsymbol{y}) = \tilde{u}(\boldsymbol{x}, \boldsymbol{y}) \tag{5.5}$$

with scalar-valued function $\hat{u}_m(\boldsymbol{x})$, orthogonal and normalized polynomials $\Phi_m(\boldsymbol{y})$, i.e. $\langle \Phi_i(\boldsymbol{y}), \Phi_j(\boldsymbol{y}) \rangle_\phi = \delta_{ij}$, and a fixed truncation M. This orthogonal property allows us to calculate

$$\hat{u}_m(\boldsymbol{x}) = \int_{\boldsymbol{y} \in D} \tilde{u}(\boldsymbol{x}, \boldsymbol{y}) \Phi_m(\boldsymbol{y}) \phi(\boldsymbol{y}) d\boldsymbol{y} = \langle \tilde{u}, \Phi_m \rangle_\phi \tag{5.6}$$

in the domain of the stochastic parameters D. Once all of these integrals have been calculated the expectation is just $\hat{u}_0(\boldsymbol{x})$ and the variance $\sum_{m=1}^{M} \hat{u}_m(\boldsymbol{x})^2$.

Hence, the main task of uncertainty quantification with the chaos expansion is to solve a quadrature formula to determine the coefficients $\hat{u}_m(\boldsymbol{x})$. We can therefore directly use the methods from the last section and apply them to uncertainty quantification. Furthermore, it follows that the findings from the last section also automatically apply to uncertainty quantification.

We have conducted several experiments in close collaboration with students in [60, 76, 115] which showed that the Sparse Grids and the spatially adaptive Combination Technique can be applied for these use cases. However, for many of the tested UQ experiments the quadrature problem is not localized enough so that the state of the art methods such as Gaussian quadrature techniques or the (quasi) Monte Carlo methods perform usually better. This is in accordance with the conclusion of the last section that showed that only for localized functions a spatially adaptive Combination Technique outperforms the standard approaches. The main use case of our spatially adaptive methods with uncertainty quantification are therefore high-dimensional localized problems that cannot be tackled easily with current approaches.

However, the adaptive process and the adaptively generated grids can give valuable insights in the structure of the problem even without a superior quadrature accuracy. For example, by looking at the refinements of the dimension-wise scheme, the important dimensions and subregions of a dimension can be determined. This can be valuable beyond the mere quadrature calculation.

This overview is only a summary of the results from our test cases with UQ that we presented for the sake of completeness. We will however not go into further details in this dissertation as the majority of the work was done by the respective students. For further information on the test cases and the results, we refer to the respective student works that show results for classical toy problems including a predator-prey model [60], a pedestrian and crowd dynamics simulator [76] and an application in hydrology [115].

5.3 Machine Learning with Sparse Grid density estimation

A typical objective of machine learning is to approximate a function based on given data points and certain objectives. It is common to differentiate the methods between supervised and unsupervised learning techniques. For supervised learning we assume that the function values are known at the given data points, while for unsupervised learning no function values are provided.

A use case for supervised learning is classification where each data point $\boldsymbol{x}_i$ is assigned to a specific class $\boldsymbol{c}_i$ – sometimes also referred to as a label. The classification task is then to find a function expression that closely fits the data points to the given labels and which generalizes well to unseen data points. The latter part is the most important aspect as the main use case for classification is to apply the learned model to classify new data points for which usually no labels are available. Famous examples to define a classifier are support vector machines – potentially using kernel functions –, k-nearest neighbors, Bayes classifiers, decision trees, and neural networks.

For unsupervised learning a classical use case is clustering. Here, we try to cluster the data points according to how "similar" they are. An optimal definition of similarity is however rather complicated and usually application-dependent. Data points that are similar form a cluster. An example would be to find customers with similar interests based on previous purchases. Well-known classes of clustering algorithms are centroid-

based methods, agglomerative clustering, hierarchical clustering, and density-based methods.

In this section, we will consider Sparse Grid Density Estimation [95] which is a Sparse Grid variant of density estimation. Based on the computed density, we will then show how to perform classification and clustering. But first we will outline the idea of density estimation and show how to compute the density function. This section and the results are based on joint works with several students. In particular, Lukas Schulte [108] implemented the density estimation with the standard Combination Technique, Cora Moser [82] extended this implementation and applied it to clustering and classification, and Markus Fabry [28] added spatial adaptivity to the density estimation. Further density estimation experiments with the Combination Technique in SG^{++} can be found in [105, 106]. Moreover, in [105, section 4.3] the possible performance gain is described that can be achieved when switching from an explicit Sparse Grid structure to the Combination Technique.

For information on how to apply the Combination Technique to regression tasks we refer to [37, 34, 35].

5.3.1 Algorithm overview

In density estimation, we want to find a density function that represents the given data well. One can think of it as a probability distribution from which the data points were sampled. However, the integral of a density is not necessarily normalized to 1. A common method in Sparse Grid Density Estimation is to start from a maximal overfitted estimate $f_\epsilon(\boldsymbol{x})$ that puts delta functions on the respective data points and sets the rest to 0. Then, spline smoothing is applied to generalize the density and achieve better results at unseen points. This results in the density representation [95]

$$\hat{f} = \operatorname*{argmin}_{u \in V} \int_D (u(\boldsymbol{x}) - f_\epsilon(x))^2 d\boldsymbol{x} + \lambda \|Lu\|_{L^2}^2. \tag{5.7}$$

The first term of the equation fits the density $u(\boldsymbol{x})$ as close as possible to the data points and the second term $\|Lu\|_{L^2}^2$ is used for regularization and uses the Laplace operator L for smoothing. The parameter $\lambda > 0$ defines the trade-off between both terms of the equation. This equation can be transformed (see [95]) to the linear equation

$$(R + \lambda C)\boldsymbol{\alpha} = \boldsymbol{b} \tag{5.8}$$

with matrices $R, C \in \mathbf{R}^{N \times N}$ where N is the number of Sparse Grid points. Here, we use $R_{ij} = \int_D \phi_i(\boldsymbol{x})\phi_j(\boldsymbol{x})d\boldsymbol{x}$ which is the scalar product between basis function i and j, $C_{ij} = \int_D L\phi_i(x)L\phi_j(\boldsymbol{x})d\boldsymbol{x}$ as regularization term, and the right hand side $b_i = \sum_{j \in [N]} \phi_i(\boldsymbol{x}_j)$ which is the only data dependent term. In case we want to scale different data points differently, we can also include coefficients $\tilde{\boldsymbol{c}}$ to adjust the right hand side $b_i = \sum_{j \in [N]} \phi_i(\boldsymbol{x}_j) \cdot \tilde{c}_j$. This factor is usually tied to the label c_j of a data point and is applied if we want to have a binary classification where one class has negative factors and one has positive factors [28]. The resulting density will then change

the sign according to the dominant class in a particular region. It should be noted that the integral computations for R_{ij} can be calculated very fast analytically [105, appendix A.1].

Since the computation of C is usually quite involved, we will instead use the identity matrix I. The reasoning behind this and the comparisons between different regularization matrices can be found in [13, 96, 93]. As a result we get

$$(R + \lambda I)\boldsymbol{\alpha} = \boldsymbol{b} \tag{5.9}$$

which is the final formula that we will use in this chapter for the calculation of the density $\hat{f} = \sum_{i \in [N]} \alpha_i \phi_i(x)$. Here, the $\boldsymbol{\alpha}$ vector is used as weight for the basis functions, which describes the final density function on the grid. Algorithm 9 summarizes this approach.

Algorithm 9 Pseudocode for the density estimation on a component grid

procedure DENSITY_ESTIMATION(evaluation_points, data, label_coefficients, δ, $\boldsymbol{\ell}$)
  ▷ Output: evaluations $\boldsymbol{y}$ ($\{\hat{f}(\boldsymbol{x}) | \boldsymbol{x} \in$ evluation_points$\}$)
  N = GET_NUMBER_OF_POINTS($\boldsymbol{\ell}$)
  initialization of basis functions $\phi_i, i \in [N]$
  M = SIZE_OF(data)
  for $i \in [N]$ **do** ▷ Compute Matrix R and right hand side $\boldsymbol{b}$
    for $j \in [N]$ **do**
      $R_{ij} = \int_D \phi_i(\boldsymbol{x})\phi_j(\boldsymbol{x})d\boldsymbol{x}$
    end for
    $b_i = 0$
    for $\boldsymbol{j} \in [M]$ **do**
      x = data$_j$
      b_i += $\phi_i(\boldsymbol{x})$·label_coefficients$_j$
    end for
  end for
  $\boldsymbol{\alpha} = (R + \lambda I)^{-1} b$ ▷ Calculate density function $\hat{f}$
  $\tilde{\boldsymbol{\alpha}}$ = NORMALIZE($\boldsymbol{\alpha}$)
  for $p \in$ evaluation_points **do** ▷ Evaluate $\hat{f}$ at evaluation_points
    $y_p = 0$
    for $i \in [N]$ **do** ▷ Evaluate basis functions at point p
      y_p += $\phi_i(p) \cdot \tilde{\alpha}_i$
    end for
  end for
end procedure

To avoid calculating a linear system which costs up to $\mathcal{O}(N^3)$ operations, we can also just use the diagonal part of R, i.e. $\tilde{R} = \text{diag}(R)$, which effectively reduces the complexity to linear. This novel approach that we will discuss in this section is similar to the mass-lumping approaches in PDE calculations where the mass matrix is diagonalized

to reduce the computational costs. We will therefore name this approach *mass-lumping*. It should be noted that in the combination technique N is typically not too large and therefore it is often still affordable to solve the full system with the standard approach. Since the matrix is guaranteed to be symmetric and positive definite, it can also be solved with the conjugate gradient method which can further boost the performance.

We can then evaluate the density at arbitrary evaluation points. This is often needed for machine learning algorithms where we want to gain new information on previously unseen points. We therefore evaluate for each evaluation point p all hat functions Φ_i weighted by their respective α_i component to get the density value. We can utilize the limited support of the basis functions to avoid iterating through all basis functions and instead only evaluate those that are non-zero at point p. This can reduce the complexity significantly.

If we look closely at Algorithm 9, we can see that we do not directly use the $\boldsymbol{\alpha}$ values. The reason for this is that we later want to combine different densities on different grids. As the magnitude of the densities is grid dependent, we first have to normalize the densities to be able to combine the results. This *normalization* routine scales $\boldsymbol{\alpha}$ by the integral of the density. For this scaling we use two different approaches depending if we use negative label coefficients or not. If all label coefficients are positive[6], we use the scaling factor $s = \frac{\sum_{i\in[N]} \max(0,\alpha_i)\cdot w_i}{\sum_{i\in[N]} w_i}$ with the quadrature weights w_i of the basis functions Φ_i. We clip negative values at 0 as negative values in this case are only artifacts of the calculation as the density should be positive. We then compute the normalized value by $\tilde{\boldsymbol{\alpha}} = \frac{1}{s} \cdot \boldsymbol{\alpha}$. In case we use a binary classification with label coefficients < 0, we also first adjust the $\boldsymbol{\alpha}$ value so that the overall integral is 0. The reason for that is that sometimes the positive and sometimes the negative values can slightly dominate the density which will lead to chaotic behavior if we combine different densities. We therefore compute the integral $I = \frac{\sum_{i\in[N]} \alpha_i\cdot w_i}{\sum_{i\in[N]} w_i}$. Here, we do not clip as negative values arise due to the binary density. This integral is then subtracted from $\boldsymbol{\alpha}$ resulting in $\boldsymbol{\alpha}^* = \boldsymbol{\alpha} - I$. We then compute the scaling factor and $\tilde{\boldsymbol{\alpha}}$ as before with this modified $\boldsymbol{\alpha}^*$, i.e. $s = \frac{\sum_{i\in[N]} \max(0,\alpha_i^*)\cdot w_i}{\sum_{i\in[N]} w_i}$ and $\tilde{\boldsymbol{\alpha}} = \frac{1}{s} \cdot \boldsymbol{\alpha}^*$. This normalized $\tilde{\boldsymbol{\alpha}}$ is then used for the evaluation of the density.

Of course, we save the computed $\tilde{\boldsymbol{\alpha}}$ vector in case we need to evaluate the density again at a later point. Hence, the linear system is only evaluated once for a specific grid. Unfortunately, if we adaptively refine the grids, the density has to be recalculated whenever the component grids change. The reason for this is that R and b change whenever the basis functions change. If we work in the hierarchical basis, one can typically formulate these adaptive refinements as low-rank updates to the matrix R. As a result it is possible to perform an adaptive step more efficiently. This low-rank update will not be considered in this work but for more information on this we refer to [105].

Since we use the Combination Technique, we have to evaluate the density for each of the component grids. We then use these approximations on the different grids and

[6] In our case this means that they are all 1.

combine them according to the combination coefficients $c_{\boldsymbol{\ell}}$ (see Section 2.3.2) to get the final density representation. The process of evaluating the combined density is outlined in Algorithm 10. It should be noted that for this combination the *normalization* routine explained before, which scales and potentially shifts the $\boldsymbol{\alpha}$ vector, is needed to allow the combination of the different densities from the different component grids. Without these normalizations single grids can dominate which can potentially flip the density to the negative side if these grids are subtracted. Of course, such effects completely falsify the results and should therefore be avoided.

Algorithm 10 Pseudocode for the density estimation with the Combination Technique

```
1: procedure DENSITY_ESTIMATION_COMBI(evaluation_points, data, coefficients, δ, I)
                                                    ▷ Output: evaluations of density y
2:     y = 0
3:     for ℓ ∈ I do
4:         y^ℓ = DENSITY_ESTIMATION(evaluation_points, data, coefficients, δ, ℓ)
5:         y += c_ℓ · y^ℓ
6:     end for
7: end procedure
```

We can now use this definition to create a Sparse Grid classifier according to the scheme presented in [93, 92]. First, we split the dataset according to the class labels. Then we independently learn the density of the data points related to each label. In Fig. 5.10, we show an example of the individual densities for each class of a classification dataset including the combination that leads to these densities. If we want to classify a new point, the density for each class is evaluated and the maximum value determines the label of the data point. This process is summarized in Algorithm 11 where we use training data with known class labels and test data for which we want to predict the label. We also pass the index set $\mathcal{I}$ for the Combination Technique and the list of classes. T denotes the number of training samples. In this approach, the SPLIT_DATASET method just returns the training samples of the respective class and label $\mathbf{1}$ as we only do a single-class density estimation.

For a binary classification, it is also common to modify the approach by creating only one density for which the sign of the density determines the class label. For multi-class classification this can be utilized in a *one-vs-others* approach [28] where we create a binary density for each class. Here, we aggregate all samples of other classes in one artificial class that is compared to the respective class. As a result, the density is positive in regions where the corresponding class is dominant and negative in regions with more data points from other classes. In our implementation this is realized with a modified behavior of the SPLIT_DATASET method. This method returns all training samples but it adjusts the label coefficients array so that it is 1 for samples from the respective class k and $-\frac{M_k}{\sum i=1;i\neq k^k M_i}$ for all other classes. We do not chose -1 for the other classes as the number of points between classes changes. As a consequence the density would have an unbalance towards the positive or negative side. Adjusting the label factors accord-

ingly helps to reduce this effect. In addition, we normalize the $\boldsymbol{\alpha}$ vector as explained before in this *one-vs-others* approach by subtracting the integral $I_{\boldsymbol{\ell}}$ of the computed density on each component grid $\boldsymbol{\ell}$ and normalizing the integral of the positive part of the density. Hence, the integral has a perfect balance between positive and negative regions and is comparable to all other component grids. We can see an example of the *one-vs-others* approach in Fig. 5.11 where we show the densities for each class.

In addition, this approach allows us to define a novel application-specific error estimator [28] for the adaptive combination technique. Since we can just evaluate the density at each training point after the density estimation is finished, we can check which points result in a positive and which in a negative density. If we compare the signs of these results to the actual label of the data points, we can define which data points are misclassified. We can then just check for each leaf in our 1D point arrays P^k and for all dimensions $k \in [d]$ how many sample points are misclassified in their support. In this case, the support of a point is determined by the one-dimensional hat basis functions that are defined by our 1D point arrays $\boldsymbol{P}^k$ with level $\boldsymbol{L}^k$. This means that for example a misclassified data point $p = (0.1, 0.7)$ counts once for the leaf in dimension 1 with a support that contains 0.1 and once for the respective leaf in dimension 2 with a support that contains 0.7. The final error estimate for each leaf is then the sum of all such misclassified data points in its support times the width of the support. Multiplying by the width avoids too strong refinement in certain dimensions as with increasing level the width decreases exponentially. As a consequence, this procedure creates for a broader and more balanced refinement across all dimensions.

Algorithm 11 Classification with Sparse Grid Density Estimation

procedure CLASSIFICATION(trainingdata, testdata, δ, $\mathcal{I}$, $classes$)
    ▷ Output: class_assignments
    for $c \in classes$ **do** ▷ Perform training and evaluate densities for each class
        trainingdata_c, coefficients = SPLIT_DATASET(trainingdata, c)
        $\boldsymbol{y}^c$ = DENSITY_ESTIMATION_COMBI(testdata, trainingdata_c, coefficients, δ, $\mathcal{I}$)
    end for
    for $i \in [T]$ **do** ▷ Assign classes based on maximum density value
        class_assignments$_i$ = $\operatorname{argmax}_{c \in classes} y_i^c$
    end for
end procedure

For the clustering application (see Algorithm 12), we rely on the k-nearest neighbors algorithm to build a graph between every data point and its k closest neighbors [94]. Here, we use the already existing implementation in scikit-learn[7][91]. Based on this graph, we use the density estimation to cut edges that cross or lie in low density regions. For this purpose, we define a threshold relative to the maximum density and compare it to the density at the midpoint of an edge. If the density is below the threshold, we remove the edge. This is applied to all edges. On the remaining graph a connected

[7] https://scikit-learn.org/

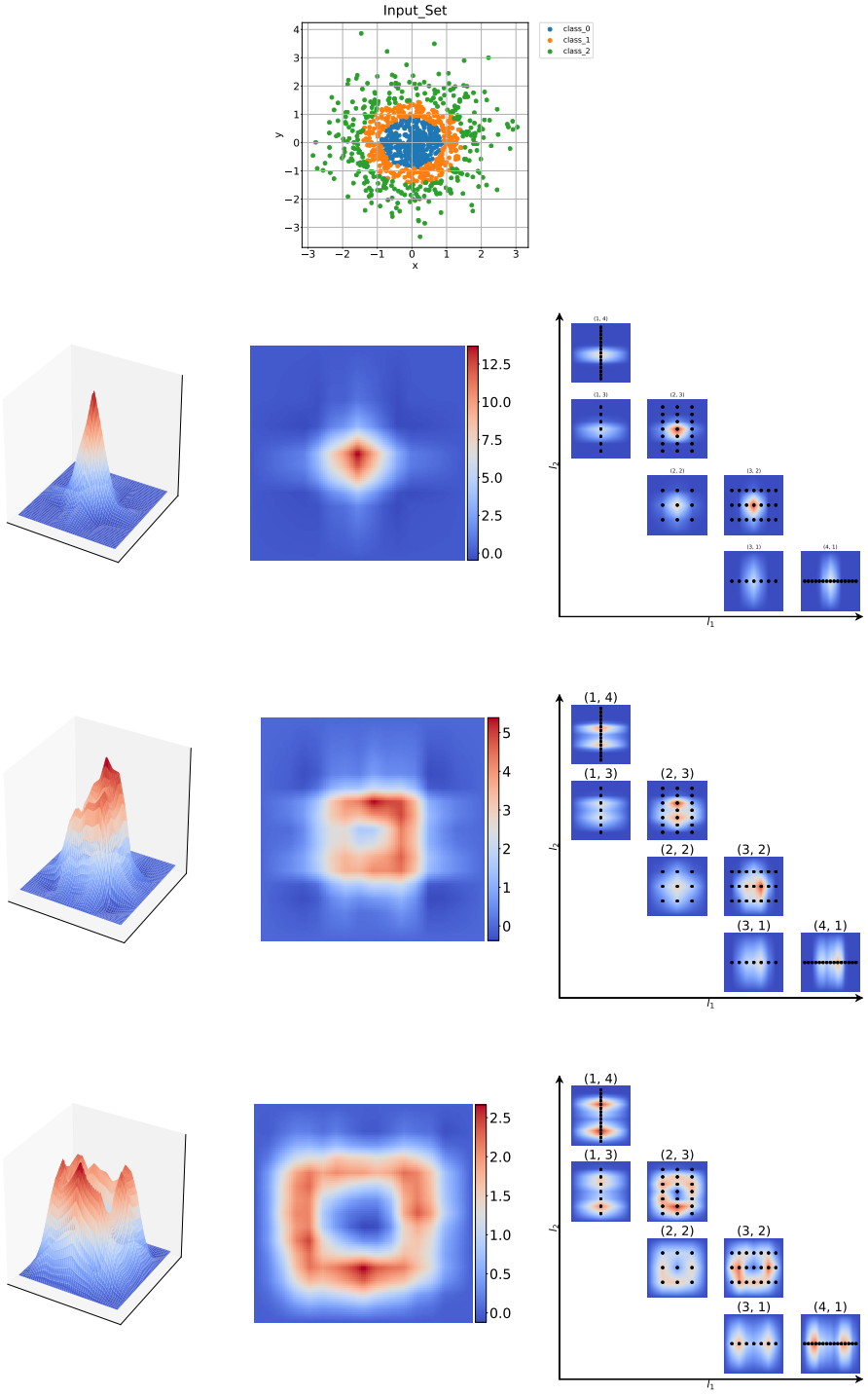


Figure 5.10: The plots show an exemplary density estimation for the *Gaussian Quantiles* dataset (top) using an independent density for each class as a 3D (left) and a contour plot (middle). We show the three densities that are calculated for the three different classes. In addition, the individual results on the component grids are shown on the right for each class.

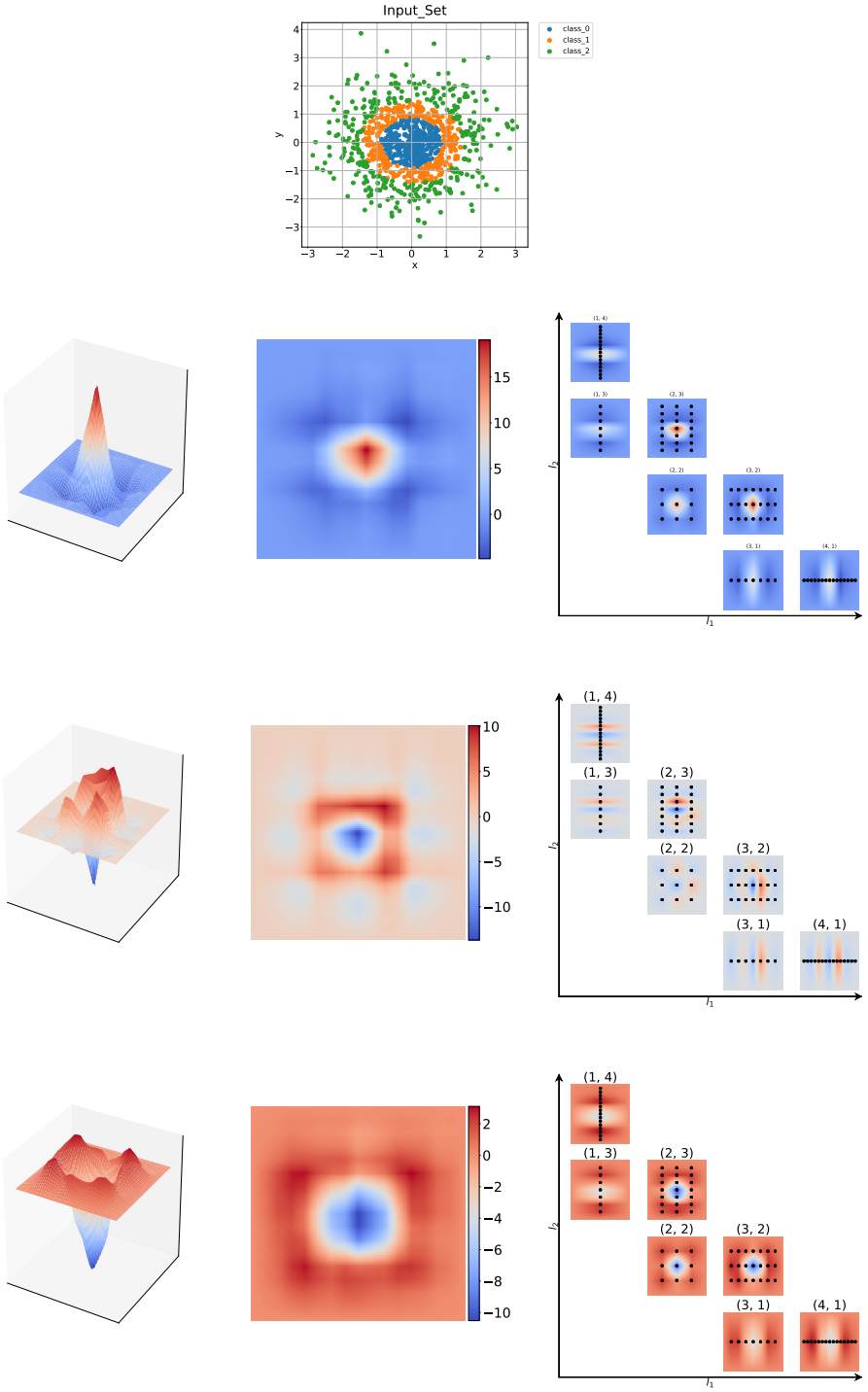


Figure 5.11: The plots show an exemplary density estimation using the *one-vs-others* approach for the *Gaussian Quantiles* dataset (top). We show the three densities that are calculated for the three different classes as a 3D plot (left), a contour plot (middle), and the individual densities on the component grids (right). For each class the density is negative in regions where the samples from other classes dominate.

component search is performed to detect clusters. This procedure is based on the work in [94, 122].

Algorithm 12 Clustering with Sparse Grid Density Estimation

1: **procedure** CLASSIFICATION(data, δ, $\mathcal{I}$,k, threshold)
▷ Output: cluster_assignments
2: edges = K_NEAREST_NEIGHBOURS(data) ▷ build k-nearest neighbour graph
3: midpoints = GET_MID_POINTS(edges) ▷ get midpoints for edges
4: $\boldsymbol{y}$ = DENSITY_ESTIMATION_COMBI(midpoints, data, δ, $\mathcal{I}$) ▷ evaluate density
5: edges_filt = $\{\text{edges}_i | y_i \geq \text{threshold}\}$ ▷ filter edges
6: clusters = CONNECTED_COMPONENT_SEARCH(data, edges_filt) ▷ find clusters
7: **end procedure**

5.3.2 Classification results

For the evaluation of the spatially adaptive Combination Technique, we will focus on the results with classification, as they are easier to quantify by the rate of correct classified samples. However, the results should also apply to clustering applications as both applications use density estimation as a core component. For people interested in first clustering results, we refer to [82].

We will look at the *Classification* and *Gaussian Quantiles* datasets which we generate using scikit-learn [91]. An example of these datasets with 1000 samples can be found in Fig. 5.12. In all our test cases, we generate 10000 samples split into 4 classes. For the *Classification* dataset each of these classes contains 1 cluster. The exact calls to the scikit-learn library are

```
make_classification(n_samples=10000, n_features=dim, n_redundant=0,
    n_clusters_per_class=1, n_information=2, n_classes=4)
make_gaussian_quantiles(n_samples=10000, n_features=dim, n_classes=4)
```

These datasets are perfectly suited for the comparison of our implementation as we can generate arbitrary numbers of samples with arbitrary dimensions *dim* and the problems are usually complicated enough so that a low Sparse Grid level does not suffice to achieve the maximum classification rate. We can therefore study how the classification rate converges if we apply the density estimation with the different variants of our implementation. In Table 5.1, we give an overview of all the different test cases that we will discuss in the following.

5.3.2.1 Standard Combination Technique

We will first compare the different algorithmic modes for the standard Combination Technique, namely, constructing a single density function for each class independently

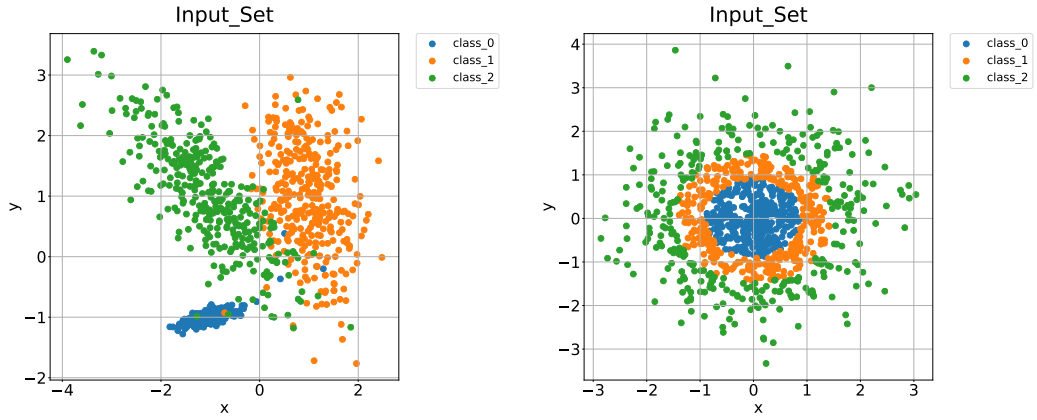


Figure 5.12: A 2D example of the *Classification* (left) and the *Gaussian Quantiles* (right) datasets with 4 classes. The different classes are highlighted in different colors.

Type	(Fixed) parameters	Dataset	Figures
Standard Combination Technique	dim, *ml*, *ovo*	*Classification*	Fig. 5.13
		Gaussian Quantiles	Fig. 5.14
	dim, *ml*, *ovo*, λ	*Classification*	Fig. 5.15
		Gaussian Quantiles	Fig. 5.16
Spatially Adaptive Combination Technique	ml=0, k=-1, dim, *ovo*, y	*Classification*	Fig. 5.17
		Gaussian Quantiles	Fig. 5.18
	ml=1, k=-1, dim, *ovo*, y	*Classification*	Fig. 5.19
		Gaussian Quantiles	Fig. 5.20
	ml=0, k=1, dim, *ovo*, y	*Classification*	Fig. 5.21
		Gaussian Quantiles	Fig. 5.22

Table 5.1: Overview of the different results for the standard Combination Technique and the Spatially Adaptive Combination Technique. We list the investigated parameters (ovo = *one-vs-others*, ml = *mass-lumping*) and indicate whether they are fixed to a specific value or varied in the respective figure. A value $k = -1$ indicates that the refinement starts at level 2 and is continuously refined from there, while $k = 1$ indicates that we reset the adaptive grid to a regular Sparse Grid once it diverges too much from the standard Sparse Grid. For more information on γ we refer to Section 4.1.5.

or the *one-vs-others* approach and the mass-lumping approach or a full matrix construction of R. In the following test cases, we first set $\lambda = 0$ and consider later the influence of the regularization parameter.

Different modes First we show in Fig. 5.13 results with varying dimensions for different *Classification* datasets. We can see that in general all approaches improve the classifi-

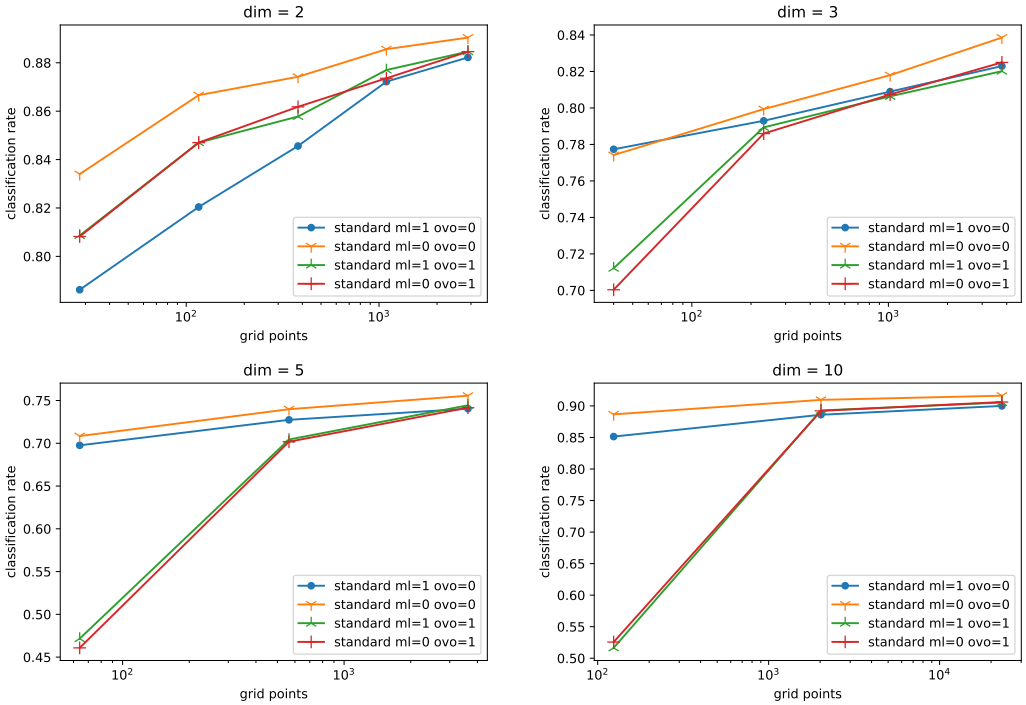


Figure 5.13: The plots show our results with the *Classification* dataset for varying dimensions. It displays how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different algorithmic approaches with *mass-lumping*(ml) or not and with *one-vs-others*(ovo) or not.

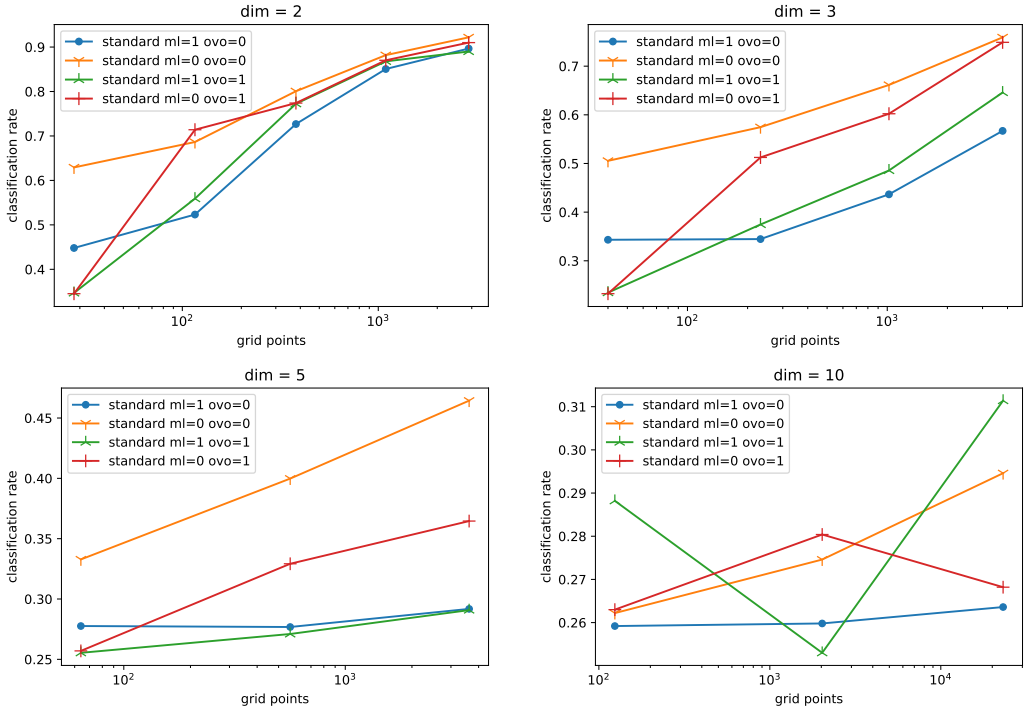


Figure 5.14: The plots show our results with the *Gaussian Quantiles* dataset for varying dimensions. It displays how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different algorithmic approaches with *mass-lumping*(ml) or not and with *one-vs-others*(ovo) or not.

cation rate with increasing grid point numbers [8]. It can also be noticed that sometimes the *mass-lumping* approaches are better and sometimes worse. Interestingly, this correlates with whether *one-vs-others* is used or not. On the one hand, for cases without *one-vs-others*, *mass-lumping* usuallly decreases the classification rate. On the other hand, for cases with *one-vs-others*, the *mass-lumping* approaches sometimes slightly improve the performance. Overall the approach without *mass-lumping* and without *one-vs-others* seems to be the best choice across all dimensions.

Next, we will look at the same configuration but with the *Gaussian Quantile* datasets in Fig. 5.14. Here, the situation is similar. In general, *mass-lumping* performs worse (unless for 10 dimensions) and the best performance is reached without *one-vs-others* and with the full matrix generation, i.e. without *mass-lumping*. This is rather consistent through all dimensions. One explanation might be the circular shape of each class which is in general complicated for the Sparse Grid representation. If we now also remove the entries in the matrix which are not on the diagonal, we also remove the

[8] It should be noted that the test data was chosen so that the accuracy at the starting level was not too high so that an increasing classification rate was observed. In some cases the generated test cases was so easy to classify that already the starting level was sufficient. In such cases the data was generated anew.

interactions between the basis functions. However, for this circular structure these interactions are vital as they show large mixed derivatives. Another observation is that the approaches are usually closer together for low dimensional problems and diverge more from each other for higher dimensions. It is therefore more crucial to chose the appropriate variant in high-dimensional scenarios. For 10 dimensions the achieved classification rates are rather low and perform close to 25% which is for the 4 class classification the baseline for a random choice. This might explain the outlier where the version with *one-vs-others* and with *mass-lumping* performs for some configurations best.

These observations show that the general configuration of the density estimation depends on the dataset. Datasets that require large interactions between the dimensions, due to for example circular shapes, perform better without *mass-lumping* while for other datasets *mass-lumping* might even boost the performance or at least give similar results. The latter case is surprising as the *mass-lumping* version is an approximation with significantly smaller cost due to the linear complexity for each component grid. The *one-vs-others* version usually performs worse for low grid point numbers but can achieve good results at high grid point numbers. The *one-vs-others* approach can therefore be especially suited for complicated cases that require high discretization levels. It should be noted that the *one-vs-others* approach significantly increases the computational cost as the full dataset is involved in the computation of the density and not just the samples of the respective class. This is especially relevant for classification scenarios with a high number of classes. We should therefore only use it if it reaches a higher classification rate.

Regularization parameter So far we have only looked at test cases without the regularization term, i.e. $\lambda = 0$. We have also analyzed the effect of the regularization parameter λ on the classification result. First of all, if mass lumping is activated the λ value is irrelevant as the matrix is already a diagonal matrix with identical entries as we use the nodal basis and consequently all hats in the component grid have an identical value for the inner product. Thus, the lambda value only scales the density up or down and therefore does not affect the classification result. For this reason, we only show different λ values if *mass-lumping* is deactivated.

For the *Classification* dataset the results (see Fig. 5.15) without *one-vs-others* indicate that usually the lower the λ values the better the results. Interestingly, the higher the λ value the more the results converge towards the version with *mass-lumping*. This is not surprising if we remember what the *mass-lumping* approach does. In this version we set every entry in the R matrix to zero except for the diagonals that stay unmodified. The λ value is always added to the diagonal. As a consequence, if the diagonal becomes very large due to this addition, the influence of the off-diagonal entries decreases. We can therefore see the *mass-lumping* as a version with a maximal λ value. For higher dimensions the same λ values have a larger impact which is due to the fact that the matrix entries in R are getting exponentially smaller with the dimension. Similarly,

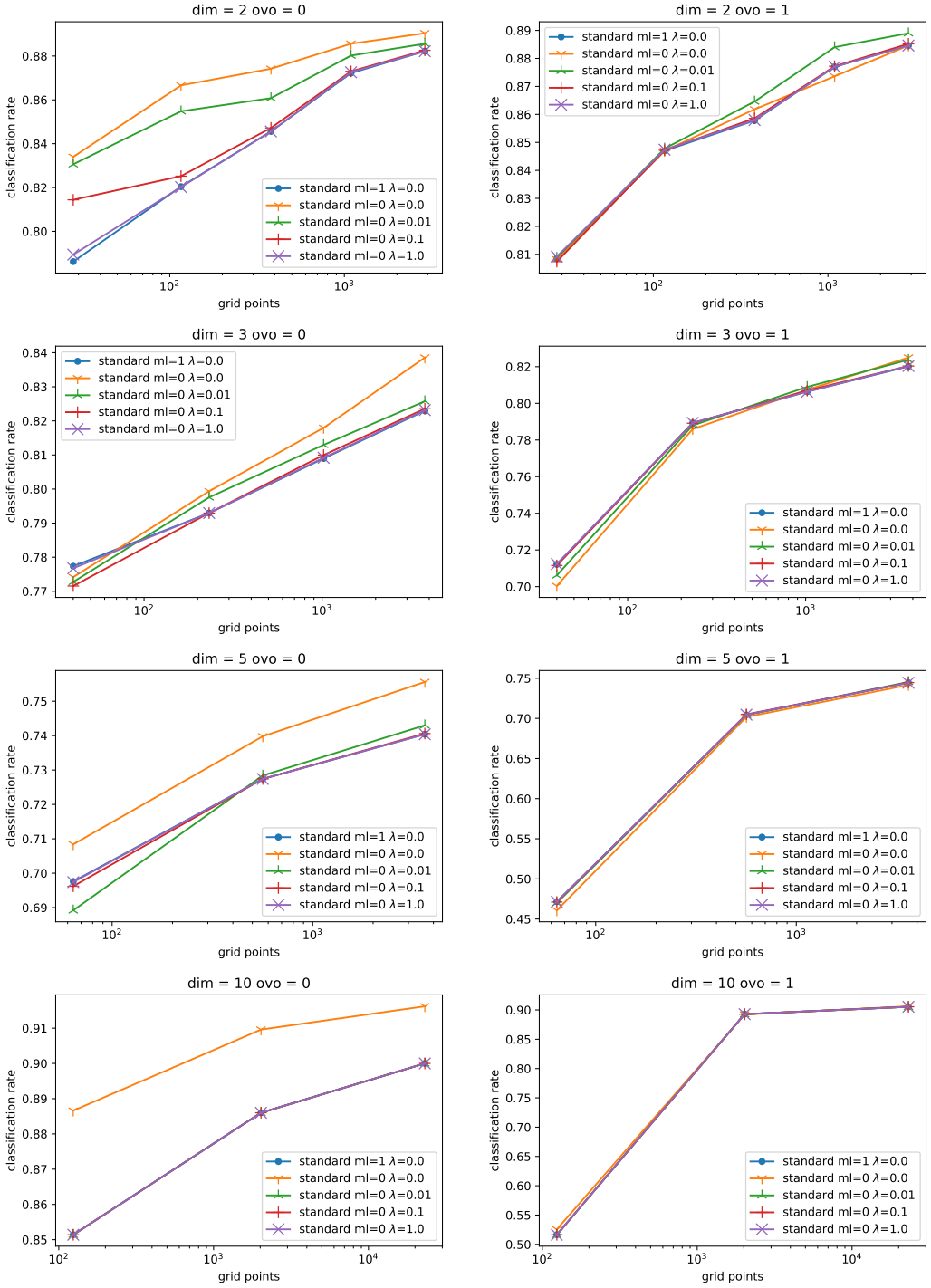


Figure 5.15: The plot shows the results with the *Classification* dataset for varying dimensions. Here we analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different λ values with *mass-lumping*(ml) or not and with activated or deactivated *one-vs-others*(ovo). For the test case with dimension 10 and ovo=0, all curves with a non-zero λ value and the *mass-lumping* version are below the purple curve.

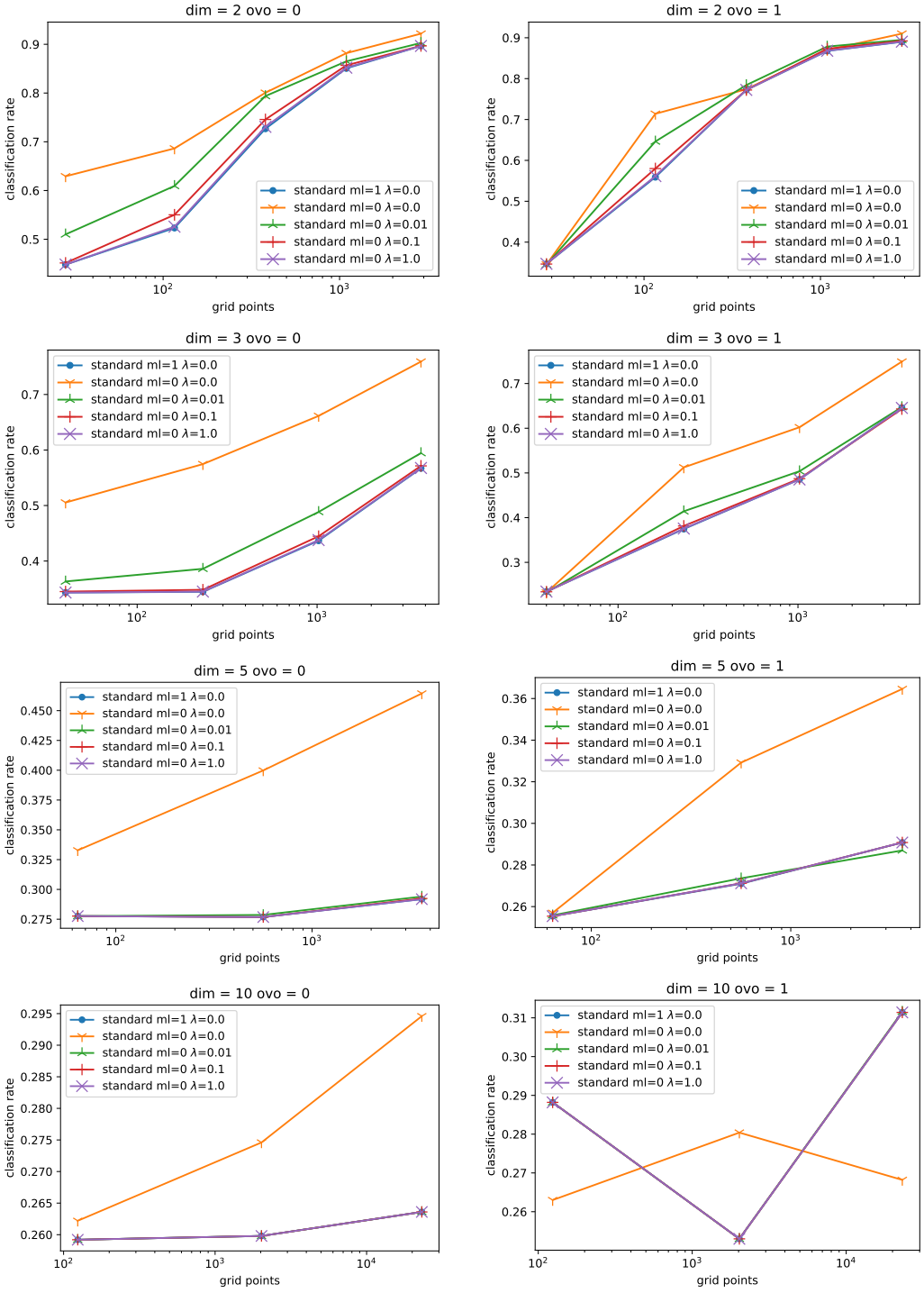


Figure 5.16: The plot shows the results with the *Gaussian Quantiles* dataset for varying dimensions. Here we analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different λ values with *mass-lumping*(ml) or not and with activated or deactivated *one-vs-others*(ovo). For the test case with dimension 10 and ovo=0, all curves with a non-zero λ value and the *mass-lumping* version are below the purple curve.

for decreasing λ values the curves converge towards the combination results without *mass-lumping* and with $\lambda = 0$.

If we use the *one-vs-others* variant, the lambda value seems to not make a big difference as all variants are pretty close together (see Fig. 5.15 on the right).

The *Gaussian Quantiles* dataset shows a similar behavior. In Fig. 5.16 one can see that usually the variant without *mass-lumping* performs best. This is very similar with and without *one-vs-others*. The only exception is the case with dimension 10 and with *one-vs-others* which could be an outlier as explained before.

5.3.2.2 Spatially adaptive Combination Technique

In the last section we have analyzed our implementation of the classification with the Sparse Grid density estimation using the standard Combination Technique. The key component of our framework is, however, the spatially adaptive Combination Technique. For this reason, we also implemented the density estimation for the dimension-wise scheme. As the density on a grid is grid-dependent and changes with the basis functions, we can not assume anymore that the results of the Combination Technique are the same as a classical Sparse Grid approximation. It is therefore an interesting test case to analyze the Combination Technique. This is especially interesting for adaptive processes as the adaptive process also transforms the density and not just increases the resolution. Furthermore, classical effects of machine learning such as overfitting can appear. Consequently, the refinement is critical and a good error estimate is necessary. In addition, the $\gamma \in [0, 1]$ parameter (see also Section 4.1.5) that decides how broad we refine is more relevant here. A γ value of 1 only refines the points with highest error estimate, while a low γ value allows for a broader refinement that includes regions with a lower error estimate. We will again consider the *Classification* and *Gaussian Quantiles* datasets at different dimensions. We will only look at test cases with $\lambda = 0$ as those cases resulted in the best classification rates for the standard Combination Technique. For the *one-vs-others* approach we use the novel error estimator based on the misclassification rate, while we use the standard surplus-based error measure if *one-vs-others* is deactivated.

In Figs. 5.17 to 5.20, we show the results for the two datasets with and without *one-vs-others* and with and without *mass-lumping*. In all cases the adaptive refinement starts at a standard grid of level 2. We can see that in comparison to the standard Combination Technique, we get an increased performance especially for test cases with *mass-lumping* (see Figs. 5.19 and 5.20). This indicates that the adaptivity can decrease the gap between the performance of the *mass-lumping* approach and the full matrix approximation that we observed for the standard Combination Technique. Also for *one-vs-others* we usually achieve an improvement (see right columns of Figs. 5.17 to 5.20). This is an indication that the novel error estimator based on the misclassification rate in the *one-vs-others* approach works nicely. If we use neither *mass-lumping* nor *one-vs-others*, we, however, only achieve for some configurations improvements over the standard Combination Technique.

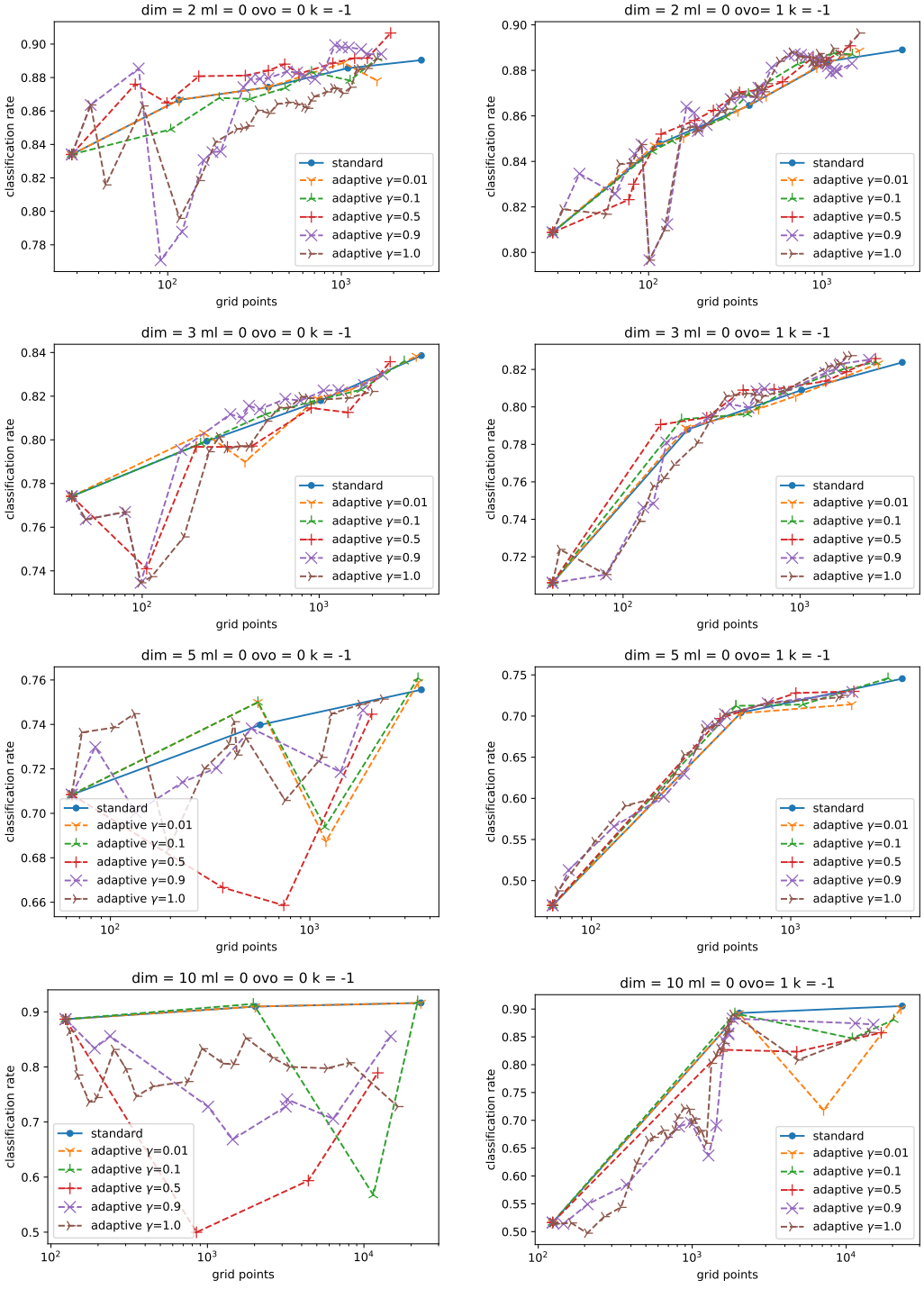


Figure 5.17: The plot shows the adaptive combination results with the *Classification* dataset for varying dimensions. We analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different γ values without *mass-lumping*(ml) and with activated or deactivated *one-vs-others*(ovo). $\lambda = 0$ for all cases. As a reference the respective result for the standard Combination Technique is given. All adaptive refinement start with a starting level of 2.

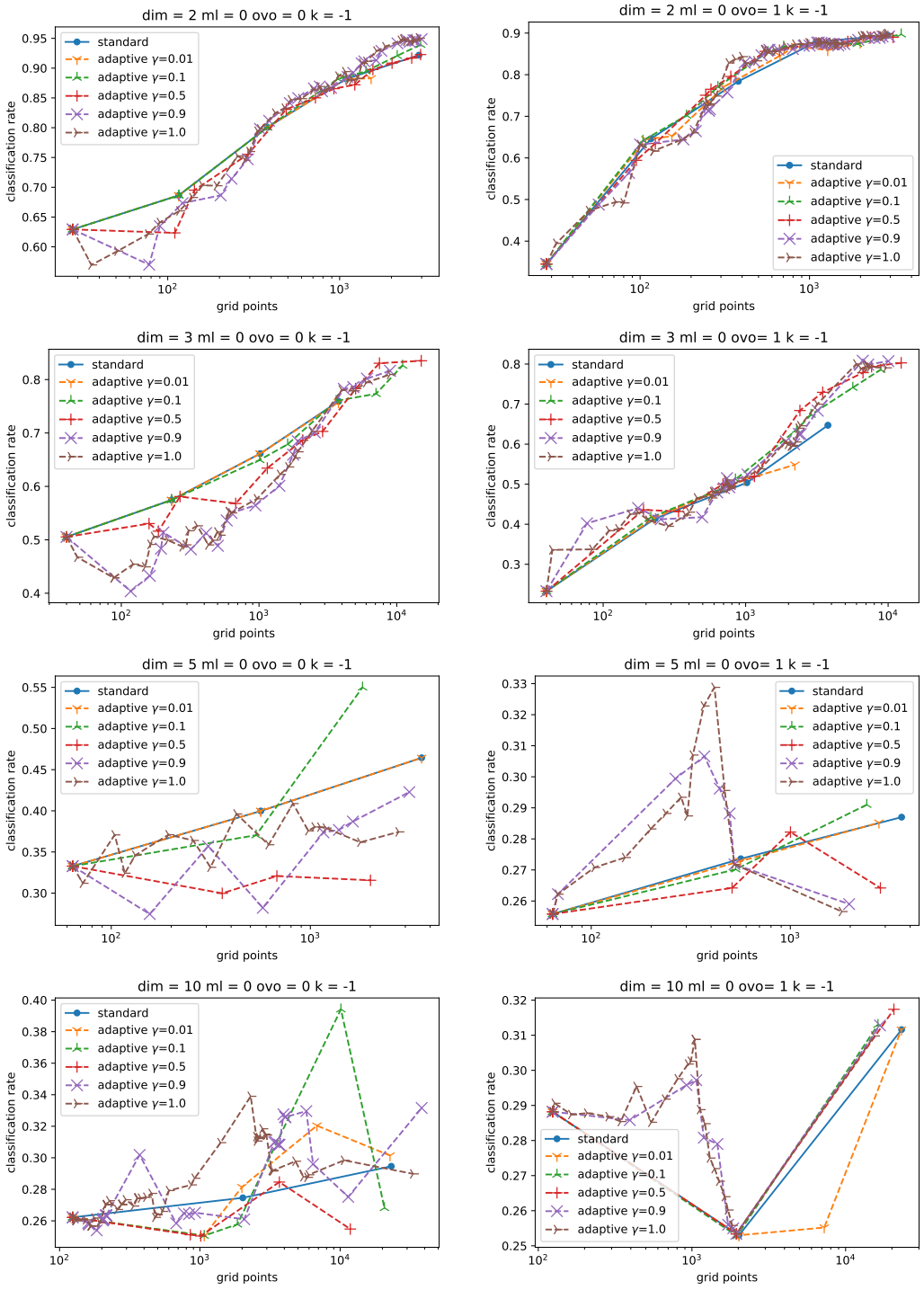


Figure 5.18: The plot shows the adaptive combination results with the *Gaussian Quantiles* dataset for varying dimensions. We analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different γ values without *mass-lumping*(ml) and with activated or deactivated *one-vs-others*(ovo). $\lambda = 0$ for all cases. As a reference the respective result for the standard Combination Technique is given. All adaptive refinement start with a starting level of 2.

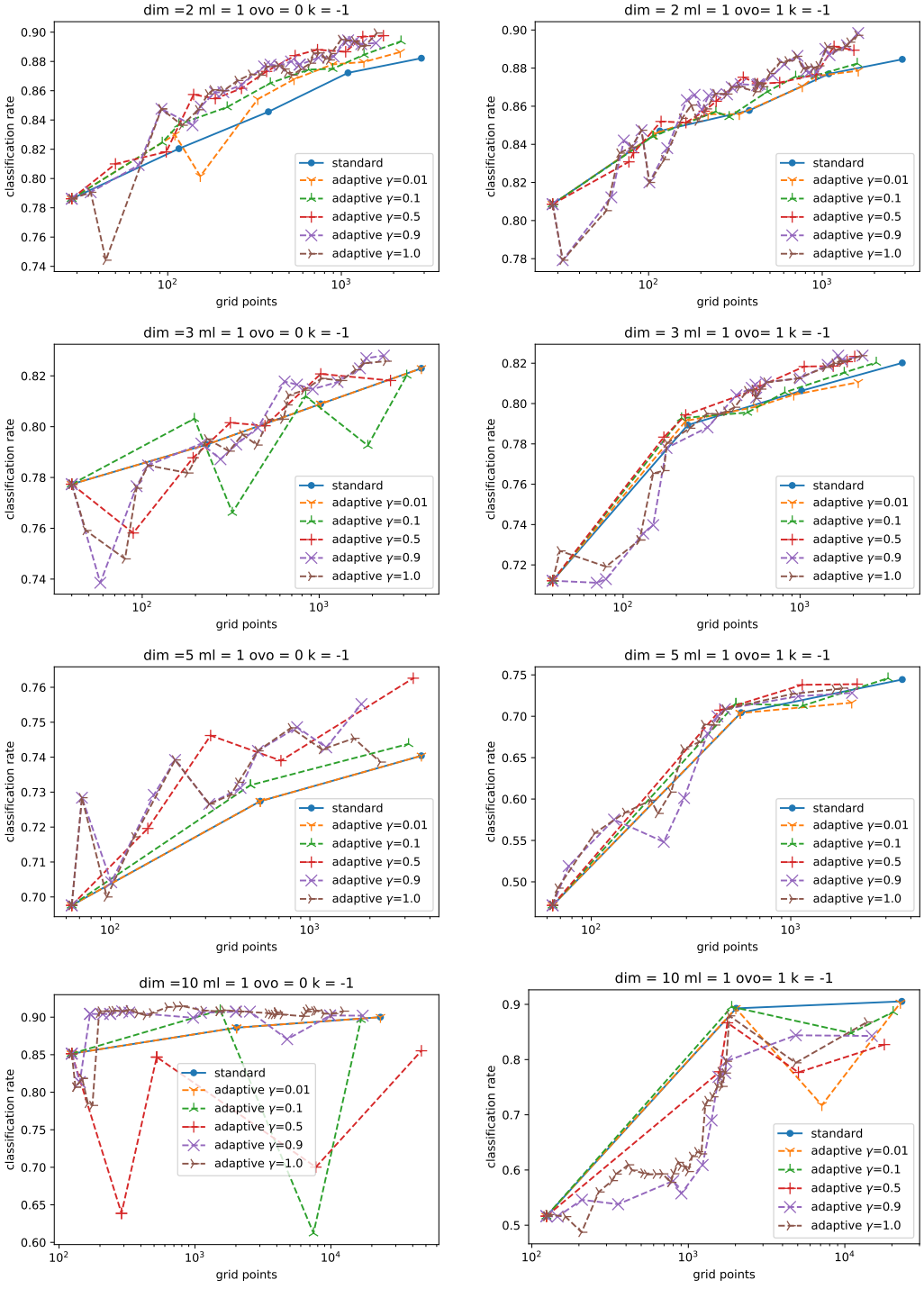


Figure 5.19: The plot shows the adaptive combination results with the *Classification* dataset for varying dimensions. We analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different γ values with *mass-lumping*(ml) and with activated or deactivated *one-vs-others*(ovo). $\lambda = 0$ for all cases. As a reference the respective result for the standard Combination Technique is given. All adaptive refinement start with a starting level of 2.

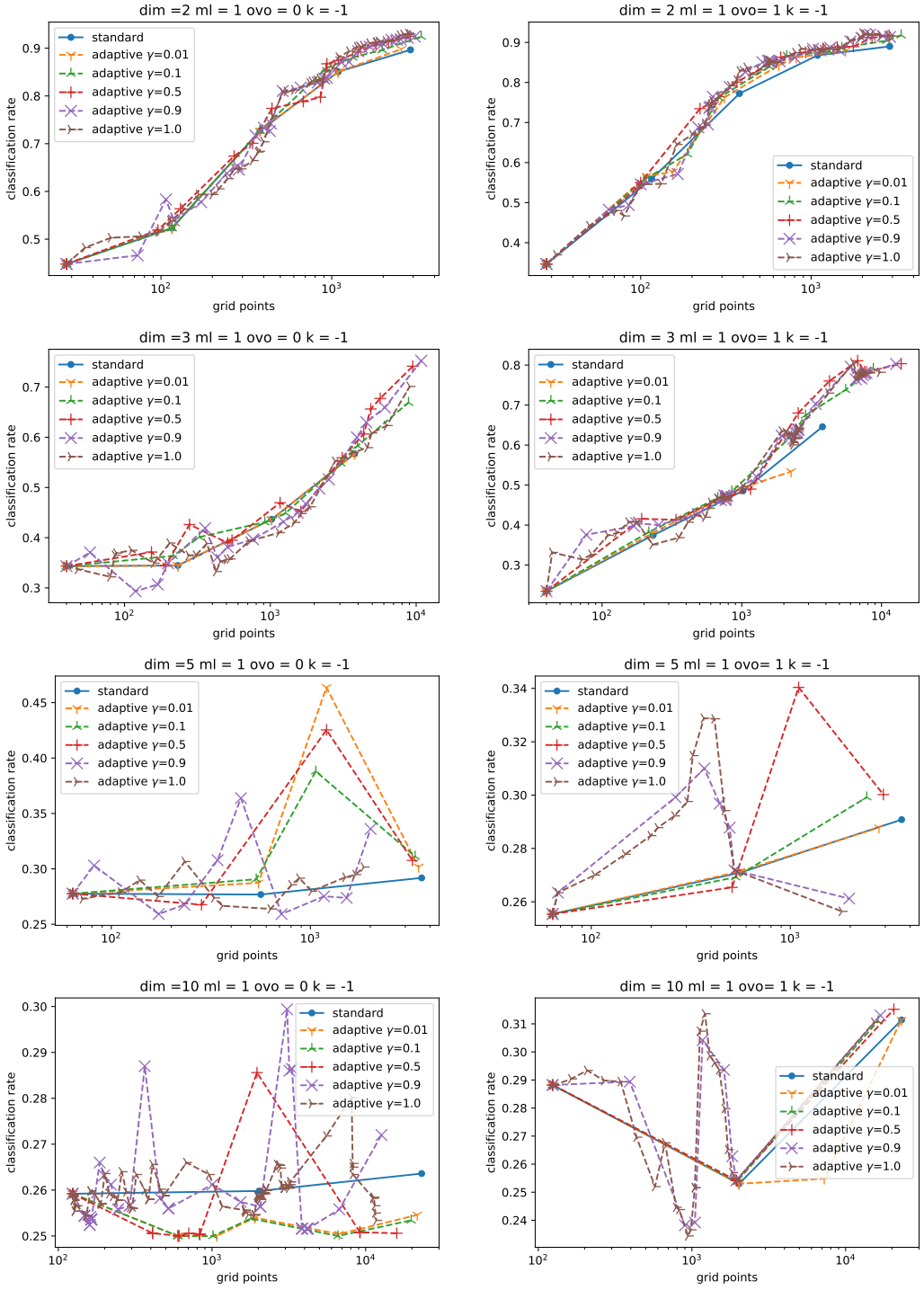


Figure 5.20: The plot shows the adaptive combination results with the *Gaussian Quantiles* dataset for varying dimensions. We analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different γ values with *mass-lumping*(ml) and with activated or deactivated *one-vs-others*(ovo). $\lambda = 0$ for all cases. As a reference the respective result for the standard Combination Technique is given. All adaptive refinement start with a starting level of 2.

If we look at the different γ values, we can see that for very broad refinements with $\gamma \leq 0.1$ we usually achieve classification rates that are close to the standard Combination Technique. Sometimes we observe slight improvements and in other cases slight decreases. The reason for that is that a large portion of the points in our 1D point lists get refined. As a consequence, the resulting grids are close to the component grids in the standard Combination Technique. The higher the γ value the more local we refine. This has the potential of an increased performance, but it can also lead to a very erratic behavior that can also cause severe decreases in the classification rate for example with premature convergence. It seems that by starting from a very coarse grid of level 2, we often refine not optimally. This can be especially seen for cases without *one-vs-others* which use the standard surplus-based error estimator. For cases with the misclassification error estimator in the *one-vs-others* approach, the curves are usually less erratic and we achieve more consistent improvements through all γ ranges. This indicates that the error estimator is more robust and can also correct the refinement if a non-optimal decision was made.

These observations with the erratic behavior and non-optimal refinement decision lead to another refinement strategy. Instead of starting from a fixed starting level we refine the grid until we exceed the number of grid points of the following Sparse Grid level l. Then we reset the grid to a standard grid of level $l-1$ and restart the refinement procedure until we exceed the number of points of a Sparse Grid with level $l+1$ and restart with a standard Sparse Grid of level l and so on. This avoids that the refinement is mislead due to wrong decisions in the beginning since the grids are not allowed to diverge too far from the grids in the standard Combination Technique.

If we look at the results in Figs. 5.21 and 5.22 , we can see that the results support this idea. For this scenario, we only show the results for the more erratic and less efficient cases without *mass-lumping*. We obtained usually better results for the configuration without *one-vs-others*. Moreover, the results are less erratic for higher grid point numbers. Thus, the method is in general more robust. Unfortunately, this also limits the possible gain of the refinement as we can only adapt the grids to a smaller degree.

5.3.3 Summary

Our results have shown that both the standard and the spatially adaptive Combination Technique can be used for the classification with the density estimation. This was not clear from the start as the density estimation differs significantly from the basic numerical operations such as interpolation and integration. We have achieved these results by carefully normalizing the different grids to obtain grid densities that can be combined. Without these normalizations, single grids can dominate, which can deteriorate the results.

We have shown different ways to tackle the classification by using single densities or pair-wise binary classifications with the *one-vs-others* approach. We also introduced a novel *mass-lumping* approach that can significantly reduce the complexity per component grid from cubic to linear while still resulting in only slightly worse classification results in many cases.

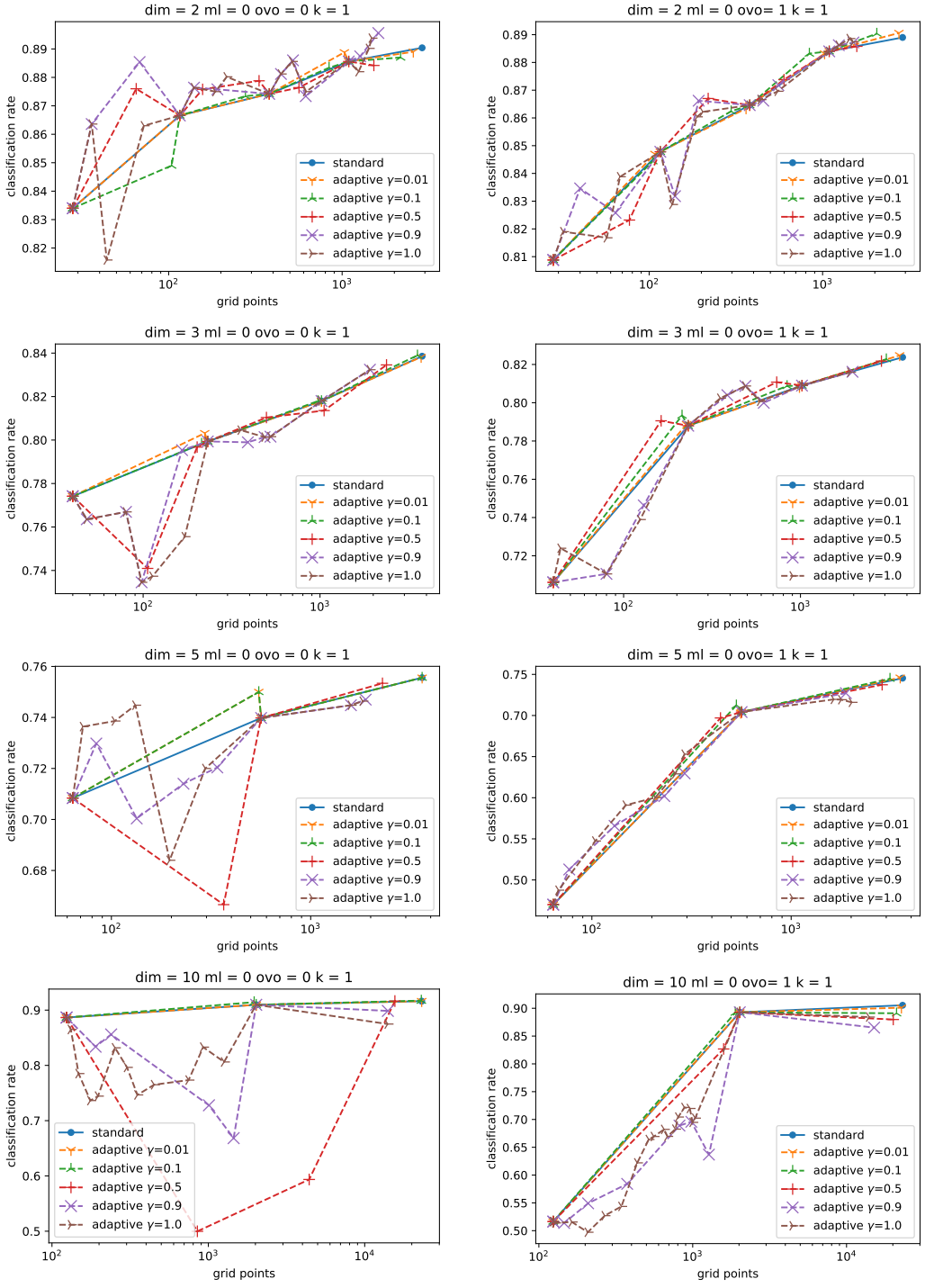


Figure 5.21: The plot shows the adaptive combination results with the *Classification* dataset for varying dimensions. We analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different γ values without *mass-lumping*(ml) and with activated or deactivated *one-vs-others*(ovo). $\lambda = 0$ for all cases. As a reference the respective result for the standard Combination Technique is given. The starting level varies throughout the adaptive process and is always set to the last level $l - 1$ once the grid points exceed the point numbers of the standard Combination Technique with level l.

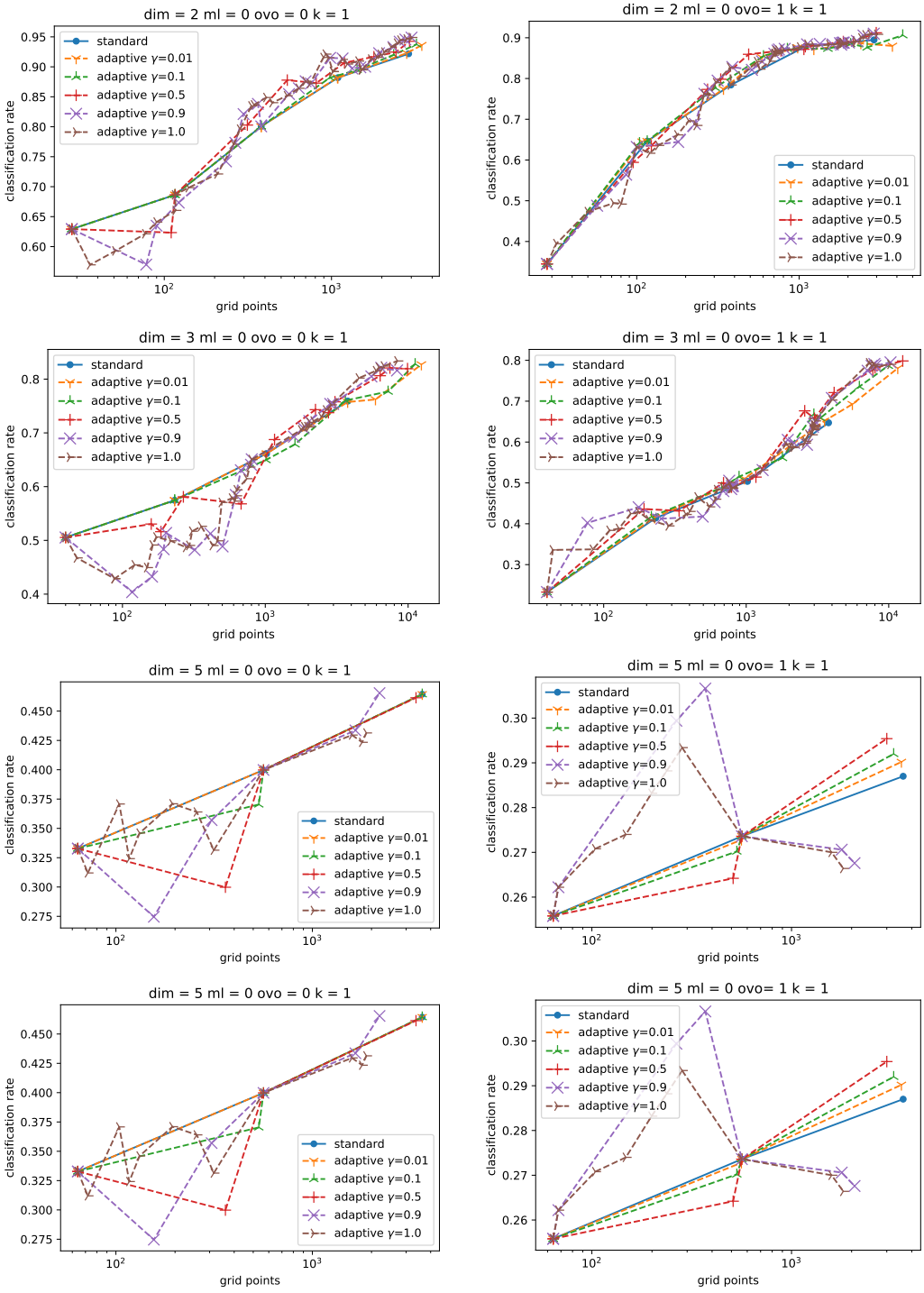


Figure 5.22: The plot shows the adaptive combination results with the *Gaussian Quantiles* dataset for varying dimensions. We analyze how the classification rate depends on the number of grid points used in the standard Combination Technique. We compare different γ values without *mass-lumping*(ml) and with activated or deactivated *one-vs-others*(ovo). $\lambda = 0$ for all cases. As a reference the respective result for the standard Combination Technique is given. The starting level varies throughout the adaptive process and is always set to the last level $l - 1$ once the grid points exceed the point numbers of the standard Combination Technique with level l.

The best configuration was usually the one without *mass-lumping* and without *one-vs-others*. Also the usage of the regularization with $\lambda > 0$ did not increase the performance. This indicates that the regularization does not help to generalize the learned model for the Combination Technique. One reason could be that the identity matrix does not represent a viable alternative for the Laplacian with the Combination Technique. In particular, we use the nodal basis for the construction of the matrix entries of R, which is different to the usage in regular Sparse Grids.

Regarding spatial adaptivity, we have seen that the adaptive implementation can in some cases improve the classification rates or reduce the number of grid points. This holds especially for configurations with *mass-lumping* or *one-vs-others*. By resetting the refinement to regular sparse grids during the refinement procedure, we can further increase the robustness of our implementation. However, this also limits the potential benefit of the refinement since the grids are then closer to the standard grids.

In addition, the selection of the error estimator is crucial and the use of a simple surplus-based error estimator for the densities seems not to be an optimal choice. This is obvious as for the classification accuracy it is not necessary to perfectly represent the density, but to find the borders between classes. This corresponds to resolving the intersection between the densities accurately. In our implementation the densities are computed and refined independently. As a consequence, only the *one-vs-others* approach can consider misclassifications in the error estimate. Here, we observed more consistent improvements and a more robust behavior which is desirable. In the future, it would be beneficial to use a refinement that considers the interplay of the different densities and which measures the misclassification errors across the densities. This might significantly increase the performance. Another possibility would be to look at intersections between the densities and refine in those regions.

It should be noted that in general the adaptive process takes significantly longer to reach the same grid point numbers. This is caused by a more compute intense implementation of the non-standard grids and the numerous refinement steps to reach the final grid. In cases where we significantly reduce the number of points, this can still pay off, but in most scenarios the *training time* will increase for constructing the densities. However, in practice often not the *training time* but the time to later evaluate the model is relevant. At this point, the reduced grid point numbers come into play and can cause a significant performance increase. The shown adaptive strategies therefore do have a relevant use case even though training might take longer.

From an implementation aspect, we showed that we can utilize the Combination Technique to get fully decoupled component grid calculations that can easily be parallelized. We can even fully decouple the density calculation of the different classes which adds an additional third parallelization layer to the two-layers of the Combination Technique itself (see also Chapter 3). This can in the future allow for the calculation of massive grids with large datasets. The density estimation is especially suited for large data sets since the most time-consuming part – the solving of the linear equation

$(R - \lambda I)\boldsymbol{\alpha} = \boldsymbol{b}$ – mainly depends on the number of grid points and not on the number of samples[9].

We have only shown results for two datasets in this section but additional (adaptive) density estimation, classification, and clustering results with various datasets can be found in [108, 82, 28]. For more localized datasets that are difficult to resolve with the standard Combination Technique, the benefit of spatial refinement is also expected to be higher.

These findings indicate that grid-based approaches for machine learning are a viable alternative for higher dimensionalities. Of course, we do not achieve everywhere equal or superior results in comparison to the industry standard. This is also related to a fundamental difference in the methods. With Sparse Grids we explicitly discretize the d-dimensional input space on a grid. This grid is optimized to generate an efficient representation for the general case and it can be customized via spatial adaptivity. Classical methods for classification, such as neural networks or support vector machines, often adaptively generate an approximative function representation and not an explicit discretization of the space. Here, the adaptive process is usually a result of solving iteratively an optimization problem. This can be more efficient for many cases but the methods often require expert knowledge to set up an efficient architecture or kernel function. In contrary, the Sparse Grid and the Combination Technique approaches are universal and can be theoretically applied to all classification problems. It is therefore not expected that they always achieve similar or superior performance. Furthermore, Sparse Grids can not scale to extremely high-dimensional problems. In general, we would recommend them in cases up to approximately 20 dimensions[10] unless an efficient dimension reduction, such as a dimension adaptive approach, is applied. One possibility for the future is to combine classical methods and Sparse Grid methods. A suited architecture could reduce the input dimensions for Sparse Grids to extend their application spectrum and performance. An example would be a Sparse Grid layer within neural networks. First experiments with such a scheme show already promising results and they could further increase the impact of Sparse Grids for machine learning in the future.

[9] Only the right hand side is depending linearly on the number of data samples.

[10] There exist specialized Sparse Grid variants that can scale up to thousands of dimensions. These application-specific *geometry-aware Sparse Grids* are discussed in [105].

6 Conclusion and Outlook

The Combination Technique has shown to be a promising and efficient method for high-dimensional problems across many application areas. Its inherent parallel nature and the efficient discretization of high-dimensional spaces with regular but anisotropic grids have made it one of the most widely used techniques in Sparse Grid literature. In this work, we have mainly focused on two aspects of the Combination Technique: an HPC implementation for time-dependent PDEs and two novel generalizations that support spatial adaptivity. For both topics we have presented new algorithmic approaches and showed their benefit with various numerical studies.

In the first part of the dissertation, we have continued the work on an HPC implementation of the Combination Technique and created the framework `DisCoTec` that aims to run massively parallel time-dependent PDE simulations on exascale computers. For this purpose, we have analyzed some of the main weaknesses and outlined possible algorithmic approaches to overcome them. First, we created a more resilient implementation of the FTCT by adding flexible communicator restructering with spare ranks that allow for simulating realistic scenarios with massively failing environments for real-world simulations. Results with the plasma physics code `GENE` have shown that the error increase is very low even in highly unreliable settings. In addition, the overhead is negligible. Second, we have shown how to improve the computational complexity by choosing optimal time steps for each component grid. Third, we have discussed the novel shared-memory parallelization that increases the performance of the (de)hierarchization step and that offers the possibility for hybrid parallelization of the black-box PDE solver. Last, we have developed a novel asynchronous Combination Technique that uses a non-blocking combination step with a delayed correction step. This scheme allows to overlap computation with the communication during the combination, which represents the most critical part of the Combination Technique for HPC as it normally does not scale to large process numbers. All of these additions have been tested by numerical experiments that showed the effectiveness of the different methods. Moreover, we investigated as a case study non-linear plasma physics simulations, which pose the biggest challenge to the Combination Technique due to their turbulent nature. Here, we observed mixed results with instabilities for frequent recombination but accurate results for long recombination intervals.

With these additions to `DisCoTec`, we have moved another step forward towards an exascale-ready framework. The presented methods offer a higher reliability in environments with frequent failures. Furthermore, the novel time stepping scheme and the shared-memory parallelization allow for more efficient execution of the PDE solver. The asynchronous combination technique offers completely new opportunities due to the possibility to hide the combination overhead. This allows to scale to much higher

process counts, and it can enable simulations in completely new settings with larger communication delays between the process groups. In addition, the presented idea for generating the asynchronous scheme might be transferable to other techniques as well.

In the second part of the dissertation, we have described two novel generalizations of the Combination Technique that add the possibility for spatial adaptivity and outlined their implementation in the framework `sparseSpACE`. This spatially adaptive idea represents a fundamental paradigm change as the standard Combination Technique was constructed to exploit the regular structure of the component grids. The new adaptive approaches lift this restriction to regular grids while keeping the main benefits of the Combination Technique. On the one hand, the dimension-wise refinement utilizes rectilinear grids that result from spatially refined one-dimensional grid structures. We have defined a novel approach to map such refined structures to the different component grids and guarantee that the combination stays valid. In addition, we presented a rebalancing structure that adaptively adjusts the level hierarchy to the problem at hand. On the other hand, the Split-Extend scheme is based on a block-adaptive refinement with the Combination Technique. Here, the *Split* operation recursively refines the grid in an octree-like fashion and the *Extend* method increases the local combination level. To result in a valid combination scheme, we presented an approach that maps the individual local levels carefully to component grids. Both refinement strategies were then tested extensively with different test cases from numerical quadrature, interpolation, uncertainty quantification, and machine learning. In these tests we could show that spatial adaptivity can improve on the standard Combination Technique in scenarios with non-smooth functions.

With these two spatially adaptive methods, we have managed to add spatial adaptivity to the Combination Technique without sacrificing its most important features: the black-box property, the embarrassingly-parallel evaluation of the component grids, and the error cancellation. The first property is guaranteed as both methods generate component grids that can be passed to black-box solvers. The only requirement is that the black-box solvers can handle, respectively, graded grids and block-adaptive grids. In addition, the computations of the individual component grids stay independent of each other which allows for a maximum of parallelism. Only the refinement steps represent synchronous points in our algorithms.

In the future, we could use these new achievements to apply to Combination Technique to huge PDE simulations on exascale computers. In such scenarios, we could test the Combination Technique with real failure events for different applications. Moreover, the asynchronous combination allows for completely new settings. One example would be to distribute the computation along different compute clusters with asynchronous recombination. There were already first attempts for such a setting with synchronous combination that however suffer from the big latency between compute clusters. This could be overcome to some extent with the non-blocking combination. Of course, it is necessary to monitor the introduced errors for further PDEs to guarantee that the asynchronous combination does not affect convergence. Similarly, we have to study how the combination of non-linear PDEs can be improved to create a Combination Technique that is more robust against instabilities.

Additionally, the spatial adaptivity can offer completely new application areas that were previously not feasible due to complicated local phenomena that need extensive refinement. It is important to test more use cases and to improve on the existing methods. An important part will be the creation of suited application-specific error estimates that guide the refinement. Currently, we are already investigation PDEs and further machine learning applications such as regression. In addition, an HPC implementation of our methods could test its capabilities for time-dependent PDEs in `DisCoTec`.

To put it in a nutshell, we have presented novel enhancements to the Combination Technique that enable a more efficient computation of time-dependent PDEs at the exascale and that increase the spectrum of possible application scenarios by introducing spatial adaptivity.

List of Figures

List of Tables

Acronyms

ABFT Algorithm-Based Fault Tolerance.

FTCT Fault-Tolerant Combination Technique.

GCP General Coefficient Problem.

HPC High Performance Computing.

MPI Message Passing Interface.

NUMA Non-Uniform Memory Access.

OpenMP Open Multi-Processing.

PDE Partial Differential Equation.

ULFM User Level Failure Mitigation.

UQ Uncertainty Quantification.

Notation

Notation	Explanation
d	Dimensionality of the problem
N	Number of points in a grid
$\tilde{N}$	Number of points per dimension in a full grid, i.e. $\tilde{N} = 2^\ell \pm 1$
$\boldsymbol{v}$	Boldface indicates vectors of corresponding dimension d, i.e. $\boldsymbol{v} = (v_1, ..., v_d)$
$\|\boldsymbol{v}\|_p$	p-Norm for vector $\boldsymbol{v}$, i.e. $\|\boldsymbol{v}\|_p = (\sum_k \|v_k^p\|)^{1/p}$
ℓ	Usually the level of a Sparse Grid or a Combination Technique
$\boldsymbol{\ell}$	Usually the level vector of a component grid
$\boldsymbol{l}$	Usually the level vector of an increment space
$V_{\boldsymbol{\ell}}$	Function space of a full grid with level vector $\boldsymbol{\ell}$
$W_{\boldsymbol{l}}$	Hierarchical increment space for level $\boldsymbol{l}$
$[n]$ ($n \in \mathbb{N}$)	Set of all numbers from 1 to n, i.e. $\{1, ..., n\}$
$[n_1, n_2]$ ($n_1, n_2 \in \mathbb{Z}$)	Set of all numbers from 1 to n, i.e. $\{n_1, ..., n_2\}$
$[a, b]$ ($a, b \in \mathbb{R}$)	Set of all real numbers in the interval between a and b
$[a, b]^d$ ($a, b \in \mathbb{R}$)	Tensor product of interval $[a, b]$, i.e. $\prod_1^d [a, b]$

Bibliography

[1] Coolmuc-2. https://doku.lrz.de/display/PUBLIC/CoolMUC-2.

[2] Cray xc40 (hazel hen). https://www.hlrs.de/systems/cray-xc40-hazel-hen/.

[3] Doxygen. https://www.doxygen.nl/index.html.

[4] Supermuc-ng. https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG.

[5] Md Mohsin Ali, James Southern, Peter E. Strazdins, and Brendan Harding. Application level fault recovery: Using Fault-Tolerant Open MPI in a PDE solver. In *Proceedings of the IEEE 28th International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2014)*, pages 1169–1178, Phoenix, USA, May 2014.

[6] Md Mohsin Ali, Peter E. Strazdins, Brendan Harding, and Markus Hegland. Complex scientific applications made fault-tolerant with the sparse grid combination technique. *International Journal of High Performance Computing Applications*, 30(3):335–359, 2016.

[7] Md Mohsin Ali, Peter E. Strazdins, Brendan Harding, Markus Hegland, and Jay W Larson. A fault-tolerant gyrokinetic plasma application using the sparse grid combination technique. In *Proceedings of the 2015 International Conference on High Performance Computing & Simulation (HPCS 2015)*, pages 499–507, Amsterdam, The Netherlands, July 2015.

[8] Robert Balder. *Adaptive Verfahren für elliptische und parabolische Differentialgleichungen auf dünnen Gittern*. Dissertation, Technische Universität München, München, 1994.

[9] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Thomas Bellebaum. Evaluation of different time-synchronization schemes for the combination technique. Bachelorarbeit, Technical University of Munich, Sep 2018.

[11] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[12] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[13] J. Blank and R. Röttger. Data mining mit dünnen gittern. Idp, Fakulty of Informatics, Technical University of Munich, January 2008.

[14] Greg Bronevetsky and Bronis Supinski. Soft error vulnerability of iterative linear algebra methods. pages 155–164, 01 2008.

[15] Hans-Joachim Bungartz. *Finite Elements of Higher Order on Sparse Grids*. Habilitationsschrift, Fakultät für Informatik, Technische Universität München, Aachen, November 1998.

[16] Hans-Joachim Bungartz and Stefan Dirnstorfer. Multivariate quadrature on adaptive sparse grids. *Computing*, 71(1):89–114, Aug 2003.

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

[18] Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, and Christoph Zenger. Pointwise convergence of the combination technique for laplace's equation. *East-West J. Numer. Math*, 2:21–45, 1994.

[19] Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, and Christoph Zenger. A proof of convergence for the combination technique for the laplace equation using tools of symbolic computation. *Mathematics and Computers in Simulation*, 42(4):595–605, 1996. Sybolic Computation, New Trends and Developments.

[20] Sayantan Chakravorty, Celso L. Mendes, and Laxmikant V. Kalé. Proactive fault tolerance in mpi applications via task migration. In Yves Robert, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing - HiPC 2006*, pages 485–496, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[21] CS Chen, YC Hon, and RA Schaback. Scientific computing with radial basis functions. *Department of Mathematics, University of Southern Mississippi, Hattiesburg, MS*, 39406, 2005.

[22] Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische annalen*, 100(1):32–74, 1928.

[23] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 2006.

[24] Tilman Dannert. *Gyrokinetische Simulation von Plasmaturbulenz mit gefangenen Teilchen und elektromagnetischen Efekten*. PhD thesis, Fakultät für Physik, January 2005.

[25] Tilman Dannert and Frank Jenko. Vlasov simulation of kinetic shear alfvén waves. *Computer Physics Communications*, 163(2):67 – 78, 2004.

[26] Persi Diaconis. Bayesian numerical analysis. *Statistical decision theory and related topics IV*, 1:163–175, 1988.

[27] Jack Dongarra, Thomas Herault, and Yves Robert. *Fault Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer International Publishing, Cham, 2015.

[28] Markus Fabry. Spatially adaptive density estimation with the sparse grid combination technique. Masterarbeit, Technical University of Munich, Sep 2020.

[29] IonuŢ-Gabriel Farcaş, Paul Cristian Sârbu, Hans-Joachim Bungartz, Tobias Neckel, and Benjamin Uekermann. Multilevel adaptive stochastic collocation with dimensionality reduction. In Jochen Garcke, Dirk Pflüger, Clayton G. Webster, and Guannan Zhang, editors, *Sparse Grids and Applications - Miami 2016*, pages 43–68, Cham, 2018. Springer International Publishing.

[30] Ionuţ-Gabriel Farcaş, Tobias Görler, Hans-Joachim Bungartz, Frank Jenko, and Tobias Neckel. Sensitivity-driven adaptive sparse stochastic approximations in plasma microinstability analysis. *Journal of Computational Physics*, 410:109394, 2020.

[31] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[32] Jochen Garcke. Regression with the optimised combination technique. In *Proceedings of the 23rd international conference on Machine learning*, pages 321–328. ACM Press, 2006.

[33] Jochen Garcke. An optimised sparse grid combination technique for eigenproblems. *PAMM*, 7(1):1022301–1022302, 2007.

[34] Jochen Garcke. A dimension adaptive sparse grid combination technique for machine learning. In Wayne Read, Jay W. Larson, and A. J. Roberts, editors, *Proceedings of the 13th Biennial Computational Techniques and Applications Conference, CTAC-2006*, volume 48 of *ANZIAM J.*, pages C725–C740, December 2007. `http://anziamj.austms.org.au/ojs/index.php/ANZIAMJ/article/view/70` [December 27, 2007].

[35] Jochen Garcke. A dimension adaptive combination technique using localised adaptation criteria. In Hans Georg Bock, Xuan Phu Hoang, Rolf Rannacher, and Johannes P. Schlöder, editors, *Modeling, Simulation and Optimization of Complex Processes*, pages 115–125, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[36] Jochen Garcke. Sparse grids in a nutshell. In *Sparse grids and applications*, pages 57–80. Springer, 2013.

[37] Jochen Garcke, M. Griebel, and Michael Thess. Data mining with sparse grids. *Computing*, 67:225–253, 10 2001.

[38] Alan Genz. A package for testing multiple integration subroutines. Jan 1987.

[39] Thomas Gerstner and Michael Griebel. Numerical integration using sparse grids. *Numerical Algorithms*, 18(3):209, Jan 1998.

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

[41] Luc Giraud, Ulrich Rüde, and Linda Stals. Resiliency in Numerical Algorithm Design for Extreme Scale Simulations (Dagstuhl Seminar 20101). *Dagstuhl Reports*, 10(3):1–57, 2020.

[42] T. Goerler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told. The global version of the gyrokinetic turbulence code GENE. *J. Comput. Phys.*, 230:7053–7071, 2011.

[43] Michael Griebel and Jan Hamaekers. A wavelet based sparse grid method for the electronic schrödinger equation. *Proceedings oh the International Congress of Mathematicians, Vol. 3, 2006-01-01, ISBN 978-3-03719-022-7, pags. 1473-1506*, 3, 01 2006.

[44] Michael Griebel, Michael Schneider, and Christoph Zenger. A combination technique for the solution of sparse grid problems. In *Iterative Methods in Lin. Alg.*, pages 263–281, 1992.

[45] Michael Griebel, Marc-Alexander Schweitzer, and Lukas Troska. A fault-tolerant domain decomposition method based on space-filling curves, 2021.

[46] Michael Griebel and Veronika Thurner. The efficient solution of fluid dynamics problems by the combination technique. *International Journal of Numerical Methods for Heat & Fluid Flow*, 5(3):251–269, March 1995.

[47] Vivian Haller. Evaluation of dimension-wise error estimates using the spatially adaptive combination technique. Bachelorarbeit, Technical University of Munich, May 2019.

[48] Brendan Harding et al. Fault tolerant computation with the sparse grid combination technique. *SIAM Journal on Scient. Comp.*, 37(3):C331–C353, 2015.

[49] Brendan Harding and Markus Hegland. A robust combination technique. *ANZIAM Journal*, 54:C394–C411, 2013.

[50] Brendan Harding and Markus Hegland. Robust solutions to PDEs with multiple grids. In Jochen Garcke and Dirk Pflüger, editors, *Sparse Grids and Applications - Munich 2012 SE*, volume 97 of *Lecture Notes in Computational Science and Engineering*, pages 171–193. Springer International Publishing, 2014.

[51] Mario Heene. *A massively parallel combination technique for the solution of high-dimensional PDEs*. PhD thesis, University of Stuttgart, 2017.

[52] Mario Heene, Christoph Kowitz, and Dirk Pflüger. Load balancing for massively parallel computations with the sparse grid combination technique. In *Parallel Computing: Accelerating Comp. Science and Eng.*, pages 574–583, 2013.

[53] Mario Heene, Alfredo Parra Hinojosa, Hans-Joachim Bungartz, and Dirk Pflüger. A massively-parallel, fault-tolerant solver for high-dimensional pdes. Euro-Par, 2016. Accepted.

[54] Mario Heene and Dirk Pflüger. Efficient and scalable distributed-memory hierarchization algorithms for the sparse grid combination technique. In *Parallel Computing: On the Road to Exascale*, 2016.

[55] Mario Heene and Dirk Pflüger. Scalable algorithms for the solution of higher-dimensional pdes. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 165–186. Springer, 2016.

[56] Markus Hegland. Adaptive sparse grids. In *Proc. of 10th Computational Techniques and Applications Conference CTAC-2001*, volume 44, pages C335–C353, apr 2003.

[57] Markus Hegland, Jochen Garcke, and Vivien Challis. The combination technique and some generalisations. *Linear Algebra Appl.*, 420(2–3):249–275, 2007.

[58] Sven Hingst. Shared-memory parallelization of a parallel combination technique framework. Bachelorarbeit, Technical University of Munich, Nov 2019.

[59] Alfredo Parra Hinojosa, Brendan Harding, Markus Hegland, and Hans-Joachim Bungartz. Handling silent data corruption with the sparse grid combination technique. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 187–208. Springer, 2016.

[60] Fritz Hofmeier. Applying the spatially adaptive combination technique to uncertainty quantification. Bachelorarbeit, Technical University of Munich, Sep 2019.

[61] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

[62] Philipp Hupp. Performance of unidirectional hierarchization for component grids virtually maximized. In *ICCS 2014*, Procedia Computer Science. Elsevier, June 2014.

[63] Philipp Hupp and Riko Jacob. A cache-optimal alternative to the unidirectional hierarchization algorithm. In Jochen Garcke and Dirk Pflüger, editors, *Sparse Grids and Applications - Stuttgart 2014*, pages 103–132, Cham, 2016. Springer International Publishing.

[64] Daan Huybrechs. Stable high-order quadrature rules with equidistant points. *Journal of Computational and Applied Mathematics*, 231(2):933–947, 2009.

[65] Denis Jarema. *Efficient Eulerian Gyrokinetic Simulations with Block-Structured Grids*. Dissertation, Technische Universität München, München, 2017.

[66] Frank Jenko et al. Electron temperature gradient driven turbulence. *Physics of Plasmas (1994-present)*, 7(5):1904–1910, 2000.

[67] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[68] Gerta Köster, Michael Seitz, Franz Treml, Dirk Hartmann, and Wolfram Klein. On modelling the influence of group formations in a crowd. *Contemporary Social Science*, 6(3):397–414, 2011.

[69] Christoph Kowitz. *Applying the Sparse Grid Combination Technique in Linear Gyrokinetics*. Dissertation, Technische Universität München, München, 2016.

[70] Christoph Kowitz and Markus Hegland. An Opticom Method for Computing Eigenpairs. In Jochen Garcke and Dirk Pflüger, editors, *Sparse Grids and Applications - Munich 2012 SE*, volume 97 of *Lecture Notes in Computational Science and Engineering*, pages 239–253. Springer International Publishing, 2014.

[71] Christoph Kranz. *Untersuchungen zur Kombinationstechnik bei der numerischen Strömungssimulation auf versetzten Gittern*. Dissertation, TU München, 2002.

[72] Rafael Lago, Michael Obersteiner, Theresa Pollinger, Johannes Rentrop, Hans-Joachim Bungartz, Tilman Dannert, Michael Griebel, Frank Jenko, and Dirk Pflüger. Exahd: A massively parallel faulttolerant sparse grid approach for high-dimensional turbulent plasma simulations. In Severin Reiz Hans-Joachim Bungartz, Benjamin Uekermann, Philipp Neumann, and Wolfgang E. Nagel, editors, *Software for Exascale Computing - SPPEXA 2016-2019*, number 136 in Lecture Notes in Computational Science and Engineering, pages 301–329. Springer, Gewerbestrasse 11, 6330 Cham, Switzerland, Jul 2020.

[73] Boris Lastdrager, Barry Koren, and Jan Verwer. The sparse-grid combination technique applied to time-dependent advection problems. *Applied Numerical Mathematics*, 38(4):377–401, 2001.

[74] Olivier Le Maître and Omar M Knio. *Spectral methods for uncertainty quantification: with applications to computational fluid dynamics*. Springer Science & Business Media, 2010.

[75] Franciszek Leja. Sur certaines suites liées aux ensembles plans et leur application à la représentation conforme. *Annales Polonici Mathematici*, 4(1):8–13, 1957.

[76] Anastasiya Liatsetskaya. Adaptive quadrature with the combination technique for uq applications. Bachelorarbeit, Technical University of Munich, Jun 2020.

[77] Meilin Liu, Zhen Gao, and Jan S. Hesthaven. Adaptive sparse grid algorithms with applications to electromagnetic scattering under uncertainty. *Applied Numerical Mathematics*, 61(1):24–37, 2011.

[78] K. Ludwig and M. Bremicker. The Water Balance Model LARSIM: Design, Content and Applications. Technical report, 2006.

[79] Martin Molzer. Implementation of a parallel sparse grid combination technique for variable process group sizes. Bachelor's thesis, Jan 2018.

[80] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[81] William J. Morokoff and Russel E. Caflisch. Quasi-monte carlo integration. *Journal of Computational Physics*, 122(2):218–230, 1995.

[82] Cora Charlotte Moser. Machine learning with the sparse grid density estimation using the combination technique. Bachelorarbeit, Technical University of Munich, Sep 2020.

[83] J. Noordmans and P. W. Hemker. Application of an adaptive sparse-grid technique to a model singular perturbation problem. *Computing*, 65(4):357–378, Dec 2000.

[84] Michael Obersteiner and Hans-Joachim Bungartz. A spatially adaptive sparse grid combination technique for numerical quadrature. In *Sparse Grids and Applications - Munich 2018*, 2019. Accepted for publication.

[85] Michael Obersteiner and Hans-Joachim Bungartz. A generalized spatially adaptive sparse grid combination technique with dimension-wise refinement. *SIAM Journal on Scientific Computing*, 2020. Accepted for publication.

[86] Michael Obersteiner, Alfredo Parra Hinojosa, Mario Heene, Hans-Joachim Bungartz, and Dirk Pflüger. A highly scalable, algorithm-based fault-tolerant solver for gyrokinetic plasma simulations. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, New York, NY, USA, 2017. Association for Computing Machinery.

[87] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, 2018.

[88] Carlos Pachajoa, Markus Levonyak, and Wilfried N. Gansterer. Extending and evaluating fault-tolerant preconditioned conjugate gradient methods. In *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pages 49–58, 2018.

[89] Alfredo Parra Hinojosa. *Toward Resilient Exascale PDE Solvers Using the Combination Technique*. Dissertation, Technische Universität München, München, 2017.

[90] Alfredo Parra Hinojosa, Christoph Kowitz, Mario Heene, Dirk Pflüger, and H-J Bungartz. Towards a fault-tolerant, scalable implementation of gene. In *Recent Trends in Computational Engineering-CE2014*, pages 47–65. Springer, 2015.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[92] Benjamin Peherstorfer. *Model Order Reduction of Parametrized Systems with Sparse Grid Learning Techniques*. Dissertation, Technische Universität München, München, 2013.

[93] Benjamin Peherstorfer, Fabian Franzelin, Dirk Pflüger, and Hans-Joachim Bungartz. Classification with probability density estimation on sparse grids. In Jochen Garcke and Dirk Pflüger, editors, *Sparse Grids and Applications - Munich 2012*, pages 255–270, Cham, 2014. Springer International Publishing.

[94] Benjamin Peherstorfer, Dirk Pflüger, and Hans-Joachim Bungartz. Clustering based on density estimation with sparse grids. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence*, pages 131–142, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[95] Benjamin Peherstorfer, Dirk Pflüge, and Hans-Joachim Bungartz. *Density Estimation with Adaptive Sparse Grids for Large Data Sets*, pages 443–451.

[96] Dirk Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, Munich, Aug 2010.

[97] Dirk Pflüger, Hans-Joachim Bungartz, Michael Griebel, Frank Jenko, Tilman Dannert, Mario Heene, Christoph Kowitz, Alfredo Parra Hinojosa, and Peter Zaspel. Exahd: an exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *European Conference on Parallel Processing*, pages 565–576. Springer, 2014.

[98] Theresa Pollinger and Dirk Pflüger. Learning-based load balancing for massively parallel simulations of hot fusion plasmas. Advances in Parallel Computing 36, pages 137–146, 2020.

[99] Petar Radojkovic, Manolis Marazakis, Paul Carpenter, Reiley Jeyapaul, Dimitris Gizopoulos, Martin Schulz, Adria Armejach, Eduard A Ayguade, François Bodin, Ramon Canal, Franck Cappello, Fabien Chaix, Guillaume Colin De Verdiere, Said Derradji, Stefano Di Carlo, Christian Engelmann, Ignacio Laguna, Miquel Moreto, Onur Mutlu, Lazaros Papadopoulos, Olly Perks, Manolis Ploumidis, Bezhad Salami, Yanos Sazeides, Dimitrios Soudris, Yiannis Sourdis, Per Stenstrom, Samuel Thibault, Will Toms, and Osman Unsal. Towards Resilient EU HPC Systems: A Blueprint. Research report, European HPC resilience initiative, April 2020.

[100] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 11 2005.

[101] Pascal Resch. Adaptive romberg-quadrature for the sparse grid combination technique. Masterarbeit, Technical University of Munich, Feb 2021.

[102] Yves Robert. An overview of fault-tolerant techniques for HPC. Euro-Par, 2016.

[103] Werner Romberg. Vereinfachte numerische integration. *Det Kongelige Norske Videnskabers Selskabs*, 28:30–36, 1955.

[104] A. Rüttgers and M. Griebel. Multiscale simulation of polymeric fluids using the sparse grid combination technique. *Applied Mathematics and Computation*, 319:425–443, 2018. also available as INS Preprint No. 1623.

[105] Kilian Michael Röhner. *Learning from Data with Geometry-Aware Sparse Grids*. Dissertation, Technische Universität München, München, 2020.

[106] Nico Rösel. Combigrid based dimensional adaptivity for sparse grid density estimation and classification. Bachelorarbeit, Technical University of Munich, Apr 2019.

[107] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct 2010.

[108] Lukas Schulte. Sparse grid density estimation with the combination technique. Bachelorarbeit, Technical University of Munich, Mar 2020.

[109] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 69–78, New York, NY, USA, 2012. Association for Computing Machinery.

[110] Shreyas Shenoy. Towards non-blocking combination schemes in the sparse grid combination technique. Masterarbeit, Technical University of Munich, Feb 2019.

[111] Ralph C Smith. *Uncertainty quantification: theory, implementation, and applications*, volume 12. Siam, 2013.

[112] S. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics, Doklady*, 4:240–243, 1963.

[113] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28:129–173, 2014.

[114] Daniel Told, Jonathan Cookmeyer, Florian Muller, Patrick Astfalk, and Frank Jenko. Comparative study of gyrokinetic, hybrid-kinetic and fully kinetic wave physics for space plasmas. *New Journal of Physics*, 18, 05 2016.

[115] Jonas Treplin. Parallel evaluation of adaptive sparse grids with application to uncertainty quantification of hydrology simulations. Projektarbeit, Techinische Universität München, Dec 2020.

[116] Julian Valentin. *B-Splines for Sparse Grids: Algorithms and Application to Higher-Dimensional Optimization*. PhD thesis, 2019.

[117] Johannes Walter. Design and implementation of a fault simulation layer for the combination technique on hpc systems. Master's thesis, University of Stuttgart, 2016.

[118] Zixuan Wang, Qi Tang, Wei Guo, and Yingda Cheng. Sparse grid discontinuous galerkin methods for high-dimensional elliptic equations. *Journal of Computational Physics*, 314:244–263, 2016.

[119] Norbert Wiener. The homogeneous chaos. *American Journal of Mathematics*, 1938.

[120] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.

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

[122] Florian Zipperle. Density-based clustering with periodic adaptive sparse grids. Bachelor's thesis, Technical University of Munich, Sep 2014.

A Technical Specifications of compute clusters

A.1 Hazel Hen

Peak performance	7420 TFlop/s
Cabinets	41
Number of nodes	7712
Number of cores per socket	$2 \cdot 12 = 24$
Total number of cores	185,088
Processor type	Intel® Xeon® CPU E5-2680 v3 (30M Cache, 2.50 GHz)
Memory per node	128 GB
Node-Node interconnect	Aries
Power consumption	3200 KW

Table A.1: Technical description of Hazel Hen [2].

A.2 SuperMUC-NG

Peak performance	26.3 PFlop/s
Islands	8
Number of nodes	6336
Number of cores per node	$2 \cdot 24 = 48$
Total number of cores	304,128
Processor type	Intel Skylake Xeon Platinum 8174
Memory per node	96 GB
Node-Node interconnect	OmniPath

Table A.2: Technical description of the thin nodes of SuperMUC-NG [4].

A.3 CoolMUC-2 Linux Cluster

Peak performance	1400 TFlop/s
Number of nodes	812
Number of cores per node	$2 \cdot 14 = 28$
Total number of cores	22,736
Processor type	Intel Haswell (2.6 GHz)
Memory per node	64 GB
Node-Node interconnect	FDR14 Infiniband

Table A.3: Technical description of the CoolMUC-2 linux cluster [1].

B Parameter Files

B.1 Linear and local GENE runs

For linear and local GENE runs we used the following parameter file. The entries with $ sign are set according to the specific parallelization and discretization of the component grid.

```
&parallelization
n_procs_s =    $ps
n_procs_v =    $pv
n_procs_w =    $pw
n_procs_x =    $px
n_procs_y =    $py
n_procs_z =    $pz
n_procs_sim = $nprocs
n_parallel_sims = $ngroup
/

&box
n_spec =    1
nx0     =    $nx0
nky0    =    $nky0
nz0     =    $nz0
nv0     =    $nv0
nw0     =    $nw0

kymin =    0.3000
lv     =      3.00
lw     =      9.00
adapt_lx = T
ky0_ind =    1
mu_grid_type = 'equidist'
/

&in_out
diagdir = './'
chptdir = './'

read_checkpoint  = F
write_checkpoint = T

istep_field =      1
istep_mom   =     -100
istep_nrg   =       10
```

```
istep_omega =      $istep_omega
istep_vsp   =      -500
istep_schpt =     -500
istep_energy =     -500

write_std = T
write_h5 = F
chpt_h5 = F
momentum_flux = F
/

&general
nonlinear =    F
comp_type = 'IV'
perf_vec  =  2 2 1 1 1 1 2 1 2
!nblocks   =       16
arakawa_zv = T
arakawa_zv_order = 2
hypz_opt = F

timescheme = 'RK4'
dt_max     =   0.005

timelim = 10000
ntimesteps = $ntimesteps_combi
calc_dt = F
omega_prec = 1e-12
underflow_limit = 1e-12

beta        =     0.0000000
debye2      =     0.0000000
collision_op = 'none'

init_cond = 'alm'

hyp_z =    -1
hyp_v =    0.2000

perf_tsteps =  20

/

&geometry
magn_geometry = 's_alpha'
q0         =     1.4000000
shat       =    0.7960
trpeps     =    0.18000000
major_R    =     1.0000000
norm_flux_projection   = F
/

&species
name    = 'ions'
omn     =     2.2200000
```

```
omt     =     6.9200000

mass    =     1.0000000
temp    =     1.0000000
dens    =     1.0000000
charge  =  1
/
```

B.2 Non-linear and global GENE runs

```
1
2 &parallelization
3 n_procs_s =  $ps
4 n_procs_v = $pv
5 n_procs_w = $pw
6 n_procs_x = $px
7 n_procs_y = $py
8 n_procs_z = $pz
9 n_procs_sim = $nprocs
10 n_parallel_sims = $ngroup
11 /
12
13 &box
14 n_spec =  $nspec
15 nx0    =  $nx0
16 nky0   =  $nky0
17 nz0    =  $nz0
18 nv0    =  $nv0
19 nw0    =  $nw0
20
21 kymin =  0.40670792E-01
22 lv    =     4.00
23 lw    =     16.0
24 lx    =  540.000
25 x0    =   0.5000
26 n0_global =   8
27 mu_grid_type = 'gau_lag'
28 /
29
30 &in_out
31 diagdir = './'
32 chptdir = './'
33
34 read_checkpoint  = F
35 write_checkpoint = T
36
37 istep_field =     100
38 istep_mom   =     100
39 istep_nrg   =      10
40 istep_vsp   =     100
```

```
41 istep_schpt =      000
42 /
43
44 &general
45 nonlinear =  T
46 x_local    =  F
47 comp_type = 'IV'
48 calc_dt = T
49
50 timelim     = 27000
51 ntimesteps =   $ntimesteps_combi  !number of steps between combinations
52 simtimelim =   $combitime !combination interval
53
54 beta        =     0.000
55 debye2      =     0.000
56 collision_op = 'none'
57
58 init_cond = 'db'
59
60 !hyperdiffusion parameters = damping of high modes
61 hyp_x =     0.1
62 hyp_y = 0.05
63 hyp_z =     1.000
64 hyp_v =    0.2000
65
66 arakawa_zv = F
67 /
68
69 &nonlocal_x
70 l_buffer_size =    0.1000
71 lcoef_krook    =     1.000
72 u_buffer_size =    0.1000
73 ucoef_krook    =     1.000
74 ck_heat =    0.3500E-01
75 !rad_bc_type =  2
76 /
77
78 &geometry
79 magn_geometry = 'circular'
80 q0         =     1.423
81 trpeps     =     0.000
82 minor_r    =    0.3600
83 major_R    =     1.000
84 mag_prof   = T
85 q_coeffs   =    0.8680,    0.000,    2.221
86 rhostar    =    0.1786E-02
87 /
88
89
90 &species
91 name    = 'ions'
92 prof_type =  3
93 kappa_T    =     6.900
94 LT_center =    0.5000
```

```
95 LT_width   =    0.5000E-01
96
97 kappa_n    =     2.232
98 Ln_center  =    0.5000
99 Ln_width   =    0.5000E-01
100
101 delta_x_T  =    0.3250
102 delta_x_n  =    0.3250
103
104 mass    =     1.000
105 temp    =     1.000
106 dens    =     1.000
107 charge  =  1
108 /
```