Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Mixed-Precision Data Mining for Sparse Grids

Florian Klaus

Course of Study: B.Sc. Informatik

Examiner: Prof. Dr. rer. nat. Dirk Pflüger

Supervisor: M. Sc. Malte Brunn

Commenced: July 15, 2018

Completed: January 15, 2019

Abstract

The concept of sparse grids has been introduced to allow the treatment of high-dimensional problems, including data mining tasks like classification and regression. In the process of solving these problems, a discrete approximation of a complex function is required, which is accomplished by locating weighted basis functions on the nodes of a sparse grid. Meanwhile, an implementation of such methods on a computer executes many floating-point operations. These are traditionally performed in double precision to achieve accurate results.

The hierarchical structure of sparse grids often leads to a rapid decline of basis coefficients on higher levels. Hence, the use of double precision throughout all floating-point operations is not strictly necessary. Single and half precision can be incorporated to save computation time and storage space, and a mixed-precision approach has the potential to increase efficiency, while keeping a comparable level of accuracy.

In this work, the alternation of floating-point precision in the classification and regression on sparse grids is investigated. Different mixed-precision strategies are developed with the hierarchical sparse grid structure in mind. They are incorporated into established algorithms for function approximation on sparse grids. The effects of these mixed-precision strategies on the convergence and accuracy are examined, and compared with homogeneous double (FP64), single (FP32) and half (FP16) precision. Tests were conducted on prominent data sets varying in size, dimension and complexity. The test results suggest that the partial or temporary use of a lower precision has the potential to improve efficiency, while ensuring an accuracy which can compete with uniform double precision.

Contents

List of Figures

List of Algorithms

CHAPTER 1

Introduction

In computer science, many problems demand the approximation of a multi-dimensional function in a certain domain. To approximate the function in a discrete way, methods like the *finite element method* are used. Essentially, this means that a function is constructed from a set of simpler basis functions, each of which only covers a portion of the full approximation space. Each basis function then gets assigned an individual weighting coefficient, so that, in combination, a more complex function can be approximated. Therefore, the domain has to be discretized by a multi-dimensional grid, with each grid node housing one basis function.

The common approach is to use a regular grid, which divides the domain into multiple equidistant parts. While regular grids are straightforward and convenient, they have a major disadvantage. In order to improve accuracy and reduce the error of such numerical method, the underlying grid has to be refined, yielding a very dense grid with a high number of grid nodes. Likewise, the computational effort and storage requirement increases exponentially, depending on the problem's dimensionality. That is, a regular d-dimensional grid, which is discretized into N parts in each dimension, has a total of N^d grid nodes. This *curse of dimensionality*, as it was called by [Bel61], is a well-known problem for scientific computations. As a consequence, real-world situations often can only be handled in up to four dimensions [Pfl10].

To break the curse of dimensionality to some extent, the concept of sparse grids was introduced. It was originally developed by [Zen90] in the context of partial differential equations, and has since been adopted to other problems. For instance, [BG04] gives a survey of various applications that can make use of sparse grids, such as solving partial differential equations, quadrature and data mining tasks. The main idea behind the sparse grid approach is, that the grid nodes underlie a hierarchical structure, and thereby the size of the grid is reduced. Due to this hierarchical structure, specific algorithms need to be developed for the use of sparse grid.

In [Pfl10], extensions for the regular sparse grid approach have been developed to allow a spatially adaptive refinement of the grid, with the focus to make data mining problems like classification and regression more practical for real-world problems in higher dimensions. In the process, a toolbox for sparse grids in numerical computations has been developed and published under the name SG++[1] [Pfl10]. The SG++ toolbox has been utilized in the making of this work as a general reference, and as a help to read in data sets and store the sparse grid structure.

As explained above, for each grid node a coefficient and basis function has to be kept and processed, resulting in a large number of floating-point operations. The hierarchical structuring of the sparse grid nodes leads to an interesting behaviour of these coefficients, where their respective values decrease rapidly the higher you go in their hierarchy. Therefore, it might be sufficient to store and process some values in a lower floating-point accuracy than the commonly used double precision, which has the potential to reduce computation time significantly, because floating-point operations in lower precision are often less expensive. Additionally, the required storage size is reduced.

In this work, different floating-point strategies in the context of sparse grids have been developed and implemented. In particular, double (FP64), single (FP32) and half (FP16) precision, as well as various mixtures of those three, are considered here. The influence of these strategies on the performance of data mining tasks, namely classification and regression, are investigated. Tests have been conducted on various well-known data sets. These tests shall uncover, whether anything other than a homogeneous double precision has benefits, therefore justifying further research, and if so how an appropriate strategy could look like.

At the beginning of this work, Chapter 2 gives an introduction to the concept of sparse grids. Chapter 3 goes over the algorithms which have been implemented and tested. Chapter 4 treats the topic of floating-point precision. Relevant data types are explained, floating-point operations in the algorithms are analysed, and mixed-precision strategies in the context of sparse grids are developed. In Chapter 5, the results of the convergence and error tests are presented.

[1]SG++: General Sparse Grid Toolbox: `http://sgpp.sparsegrids.org/`, last visited on 08.01.2019.

CHAPTER 2

Sparse Grids

Section 2.1-2.3 state the mathematical notations used throughout this thesis, as well as a brief introduction to the sparse grid structure and its respective approximation space. These sections are, for the most part, cited from [Bru16]. They shall serve as a quick standalone introduction to sparse grids, which should enable the reader to follow the rest of this thesis. For a more in-depth look into sparse grids and the properties of the hierarchical subspace splitting the reader is directed to [BG04], [Bun98] and [Pfl10]. Other considerations such as boundary treatment and different variants of sparse grids are discussed in greater detail in [Pfl10].

2.1 Notations

To make the later formulation of formulas and algorithms easier (and shorter), some notations have to be made. These notation are commonly used in work on the topic of sparse grid, such as [Bun98], [BG04], [Pfl10] and [Bru16].

For a d-dimensional grid we will deal with d-dimensional multi-values, such as data points $\mathbf{p}$

$$\mathbf{p} := (p_1, ..., p_d) \in \mathbb{R}^d,$$

or multi-indices, such as level $\mathbf{l}$ and index $\mathbf{i}$ of a grid node

$$\mathbf{l} := (l_1, ..., l_d) \in \mathbb{N}^d,$$
$$\mathbf{i} := (i_1, ..., i_d) \in \mathbb{N}_0^d,$$

or special multi-index constants such as

$$\mathbf{0} := (0, ..., 0),$$
$$\mathbf{1} := (1, ..., 1) \text{ or}$$
$$\mathbf{2} := (2, ..., 2).$$

On top of that, we also need some operations on these multi-values. May two exemplary multi-values be given by $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$, we define following operations:

$$\begin{aligned}
\mathbf{a} \cdot \mathbf{b} &:= (a_1 \cdot b_1, ..., a_d \cdot b_d), \\
x^{\mathbf{a}} &:= (x^{a_1}, ..., x^{a_d}),\ x \in \mathbb{R} \\
|\mathbf{a}|_p &:= (\textstyle\sum_{i=1}^d |a_i|^p)^{1/p},\ p \geq 1, \\
\mathbf{a} < \mathbf{b} &:\Leftrightarrow \forall_{i=1,\ldots,d}\ a_i < b_i, \\
\mathbf{a} \leq \mathbf{b} &:\Leftrightarrow \forall_{i=1,\ldots,d}\ a_i \leq b_i, \\
[\mathbf{a}, \mathbf{b}] &:= \textstyle\prod_{i=1}^d [a_i, b_i] \subset \mathbb{R}^d.
\end{aligned}$$

Some other attributes that will constantly be referred to when talking about sparse grids are the mesh size

$$\mathbf{h_l} \in \mathbb{R}^d := (h_{l_1}, ..., h_{l_d}) := 2^{-\mathbf{l}},$$

and grid nodes

$$\mathbf{x_{l,i}} \in \mathbb{R}^d := (x_{l_1,i_1}, ..., x_{l_d,i_d}) := \mathbf{i} \cdot \mathbf{h_l}$$

with neighborhood

$$\mathrm{H}_{\mathbf{l},\mathbf{i}} := [\mathbf{x_{l,i}} - \mathbf{h_l}, \mathbf{x_{l,i}} + \mathbf{h_l}] \cap [0,1]^d \subset \mathbb{R}^d.$$

2.2 Grid Structure

For the sake of simplicity, the domain Ω, on which the grid is going to operate, is assumed to be the d-dimensional unit-hypercube, i.e. $\Omega := [0,1]^d$. We consider a function $u : \Omega \to \mathbb{R}$, which we want to approximate. It is assumed that the function is zero at the boundary of Ω, and there will be no grid nodes on the domain's boundary.

Before we tackle sparse grids, let us first have a look at a regular d-dimensional full grid $\Omega_{0,\mathbf{l}}$ consisting of all grid nodes $\mathbf{x_{l,i}}$ on level $\mathbf{l} \in \mathbb{N}^d$ given by

(2.1) $\Omega_{0,\mathbf{l}} := \{\mathbf{x_{l,i}} = \mathbf{i} \cdot \mathbf{h_l} \ \mid\ \mathbf{i} \in \mathbb{N}_0^d, \mathbf{0} < \mathbf{i} < 2^{\mathbf{l}}\} \subset \Omega.$

Now consider a set of only the odd indices on level $\mathbf{l}$

(2.2) $\mathbb{I}_{\mathbf{l}} \ :=\ \{\mathbf{i} \in \mathbb{N}^d \ \mid\ \mathbf{1} \leq \mathbf{i} \leq 2^{\mathbf{l}} - \mathbf{1},\ i_j \text{ odd } \forall j = 1, \ldots, d\}$

and the resulting subgrid when taking only those odd indices into account

(2.3) $\hat{\Omega}_{\mathbf{l}} \ :=\ \{\mathbf{x_{l,i}} \in \Omega_{0,\mathbf{l}} \ \mid\ i \in \mathbb{I}_{\mathbf{l}}\}.$

Considering the d-dimensional full grid $\Omega_{0,\mathbf{n}}$, then the respective (regular) sparse grid is defined by

$$\Omega_n^{(1)} := \{\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \hat{\Omega}_{\mathbf{l}} \mid |\mathbf{l}|_1 \leq d + n - 1\} \subseteq \Omega_{0,\mathbf{n}}, \quad (2.4)$$

which is constructed by overlaying all subgrids $\hat{\Omega}_{\mathbf{l}}$ with a sum of the level components less or equal then $d + n - 1$. Figure 2.1 shows an exemplary comparison of a full grid (left) and a sparse grid (right), each on level four in the two dimensional case. An overview of all subgrids that make up aforesaid sparse grid is visualized in Figure 2.2.

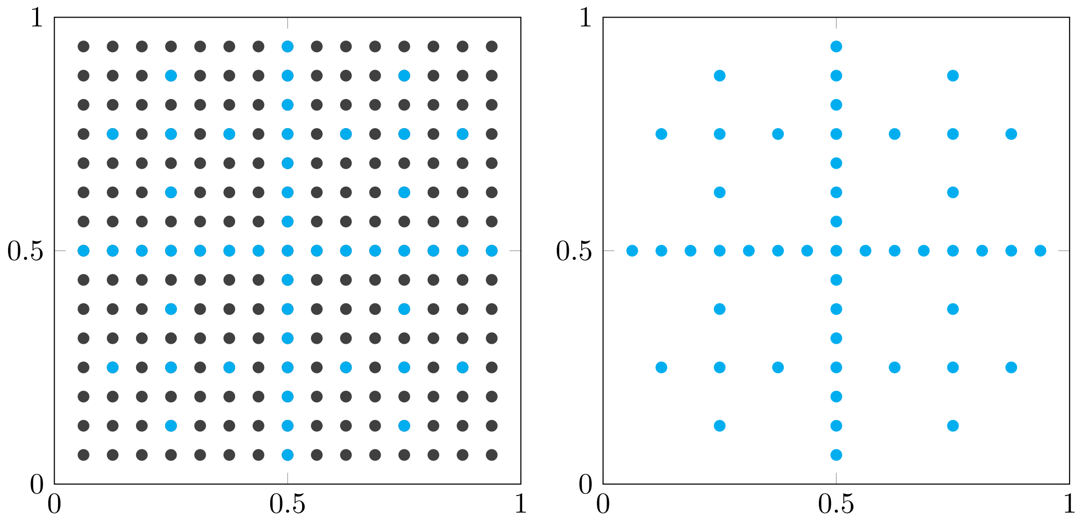


Figure 2.1: Comparison of a two-dimensional full grid (left) and regular sparse grid (right) of level four [Bru16].

2.3 Function Approximation

Such discrete grids are commonly used when dealing with complex functions in a numerical context. It is required to approximate a function $f : \Omega \to \mathbb{R}$ in a discrete way, as well as to evaluate the approximated function at an arbitrary position. Since there is only a finite number of nodes in a grid, we need a method to interpolate the function between those nodes. One common way of achiving this is to simply represent the function approximation u with the sum over a set of weighted basis function, each of which is located at a grid node.

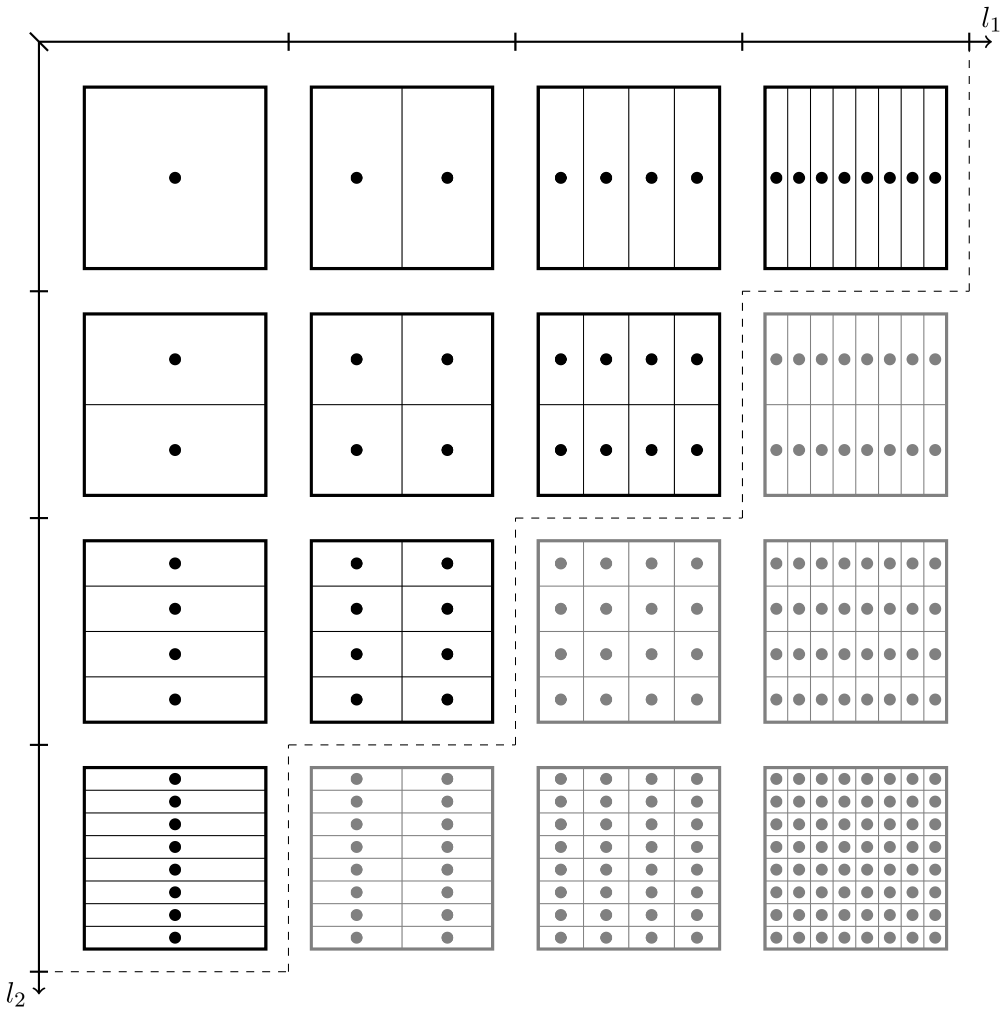


Figure 2.2: Subgrids that compound a two dimensional sparse grid of level $n = 4$ (black), plotted over both level components. The dashed line symbolizes the level sum threshold $d + n - 1$. [Bun98]

A straightforward, yet efficient and therefore popular, choice for the basis functions are piecewise d-linear functions defined by the tensor product approach, also referred to as hat-functions,

$$\begin{aligned} \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) &:= \prod_{j=1}^{d} \phi_{l_j,i_j}(x_j)\ , \\ \text{with}\ \ \phi_{l_j,i_j}(x_j) &:= \begin{cases} 1 - |\frac{x_j}{h_{l_j}} - i_j| & \text{if } x \in \mathrm{H}_{l_j,i_j}, \\ 0 & \text{else.} \end{cases} \end{aligned} \tag{2.5}$$

For the full grid $\Omega_{0,\mathbf{l}}$, this gives us a set of basis function

$$\mathcal{V}_{\mathbf{l}} := \{\phi_{\mathbf{l},\mathbf{i}} \mid \mathbf{0} < \mathbf{i} < 2^{\mathbf{l}}\} \tag{2.6}$$

and the approximation space spanned by said basis functions

$$V_{\mathbf{l}} := \operatorname{span}(\mathcal{V}_{\mathbf{l}}). \tag{2.7}$$

Looking at sparse grids, for each subgrid $\hat{\Omega}_{\mathbf{l}}$ we get a according basis and approximation space

$$\begin{aligned} \mathcal{W}_{\mathbf{l}} &:= \{\phi_{\mathbf{l},\mathbf{i}} \mid \mathbf{i} \in \mathbb{I}_{\mathbf{l}}\}, \\ W_{\mathbf{l}} &:= \operatorname{span}(\mathcal{W}_{\mathbf{l}}). \end{aligned} \tag{2.8}$$

The basis and approximation space for a sparse grid $\Omega_n^{(1)}$ are, as direct consequence of 2.4, given by

$$\begin{aligned} \mathcal{V}_n^{(1)} &:= \bigcup_{|\mathbf{l}|_1 \le d+n-1} \mathcal{W}_{\mathbf{l}}, \\ V_n^{(1)} &:= \bigoplus_{|\mathbf{l}|_1 \le d+n-1} W_{\mathbf{l}}. \end{aligned} \tag{2.9}$$

By utilizing this set of basis function, a function $u(\mathbf{x})$ can be approximated on a sparse grid by the linear combination

$$\tilde{u}(\mathbf{x}) := \sum_{\phi_{\mathbf{l},\mathbf{i}} \in \mathcal{V}_n^{(1)}} \alpha_{\mathbf{l},\mathbf{i}} \cdot \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) \tag{2.10}$$

of all basis functions $\phi_{\mathbf{l},\mathbf{i}}(\mathbf{x})$ with weighting coefficient $\alpha_{\mathbf{l},\mathbf{i}} \in \mathbb{R}$, which is referred to as *hierarchical surplus*, for convenience later just called *surplus*.

CHAPTER 3

Algorithms on Sparse Grids

Sparse grids feature a hierarchical structure and therefore require special methods and algorithms when used for tasks like data mining. In contrast to regular standard grids, where every basis function has a very local support, sparse grids contain basis functions of various support regions. The grid node hierarchy has to be considered in order to achieve adequate efficiency.

As seen in equation (2.10), all hierarchical surplus values need to be known, in order to approximate and interpolate a function. The process of determining the surplus values is referred to as *hierarchisation* [Pfl10]. Considering the hierarchical structure of sparse grids, it becomes clear that a surplus value depends on the surpluses above him in the hierarchy. Every method that approximates a function on sparse grids (or grids in general) needs to perform a hierarchisation beforehand. The specific proceeding depends on the application and available data, and hence will be treated in Section 3.2.

Section 3.1 illustrates how a function evaluation can be performed efficiently on sparse grids. The evaluation forms an essential building block for the data mining tasks presented in Section 3.2, namely regression and classification.

3.1 Evaluation

With given or calculated surpluses, the approximated function can be evaluated at an arbitrary point within the sparse grid domain. In a computational context, all surplus values α_i are stored in an array of length $N \in \mathbb{N}$, which corresponds to the total number of grid nodes (and basis functions). The evaluation at point $\mathbf{x} \in \Omega$ can mathematical be expressed as

$$\tilde{u}(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \cdot \phi_i(\mathbf{x}), \tag{3.1}$$

where ϕ_i is the piecewise d-linear basis function associated to the same grid node as α_i. The surplus array is constructed in the same sequential order as the structure housing the sparse grid itself, i.e. the grid nodes with level and index information. This way, each surplus values has a natural connection to the corresponding grid node and can be easily accessed without further searching. The basis values are calculated on-the-fly according to 2.5.

Strictly following (3.1) yields a naive implementation of the evaluation, by simply iterating over all grid nodes and adding up all surplus values α_i multiplied by the corresponding basis function ϕ_i. By definition, the basis function equals zero if the evaluation point $\mathbf{x}$ does not lie inside of its support, and therefore does not influence the result of an evaluation. Algorithm 3.1 describes such a naive implementation. The value of a d-linear basis at evaluation point $\mathbf{x}$ is computed by Algorithm 3.2.

Algorithm 3.1 Naive implementation of a sparse grid evaluation at point $\mathbf{x}$.

```
function NaiveEvaluation($x$[ ], gridNodes[ ], $\alpha$[ ])
    result $\leftarrow 0$
    for $i = 1, \ldots, N$ do
        result $\leftarrow$ result $+ \alpha[i] \cdot$ D-LinearBasis($x$, gridPoints[$i$])
    end for
    return result
end function
```

Algorithm 3.2 Computation of a piecewise d-linear basis function value at point $\mathbf{x}$.

```
function D-LinearBasis($x$[ ], gridNode)
    basis $\leftarrow 1$
    for $d' = 1, \ldots, d$ do
        level $\leftarrow$ GetLevel(gridNode, $d'$)
        index $\leftarrow$ GetIndex(gridNode, $d'$)
        h $\leftarrow 2^{-\text{level}}$
        basis $\leftarrow$ basis $\cdot$ Max$(1 - |\frac{x[d']}{\text{h}} - index|, 0)$
    end for
    return basis
end function
```

Each basis computation consists of d linear components. The naive evaluation algorithm computes N basis function, where N stands for the number of grid nodes. This N is in $\mathcal{O}(2^n n^{d-1})$ (proof can be found in [Bun98]; Section 4.2 discusses grid node growth in more detail). Hence, Algorithm 3.1 has a time complexity of $\mathcal{O}(Nd) = \mathcal{O}(2^n n^{d-1} d)$.

While this naive implementation, also called *streaming method*, might have its use cases [Bru16], a more sophisticated approach exists. Many basis functions are effectively zero, because the evaluation point does not lie inside their support. The naive evaluation thereby wastes a lot of computation time looking at basis functions that contribute nothing to the result, making it rather inefficient. Figure 3.1 visualizes this behaviour at an exemplary evaluation point in the subgrid scheme. Since neighbouring basis functions in a subgrid do not overlap, each subspace houses at most one basis function that is non-zero in the evaluation.

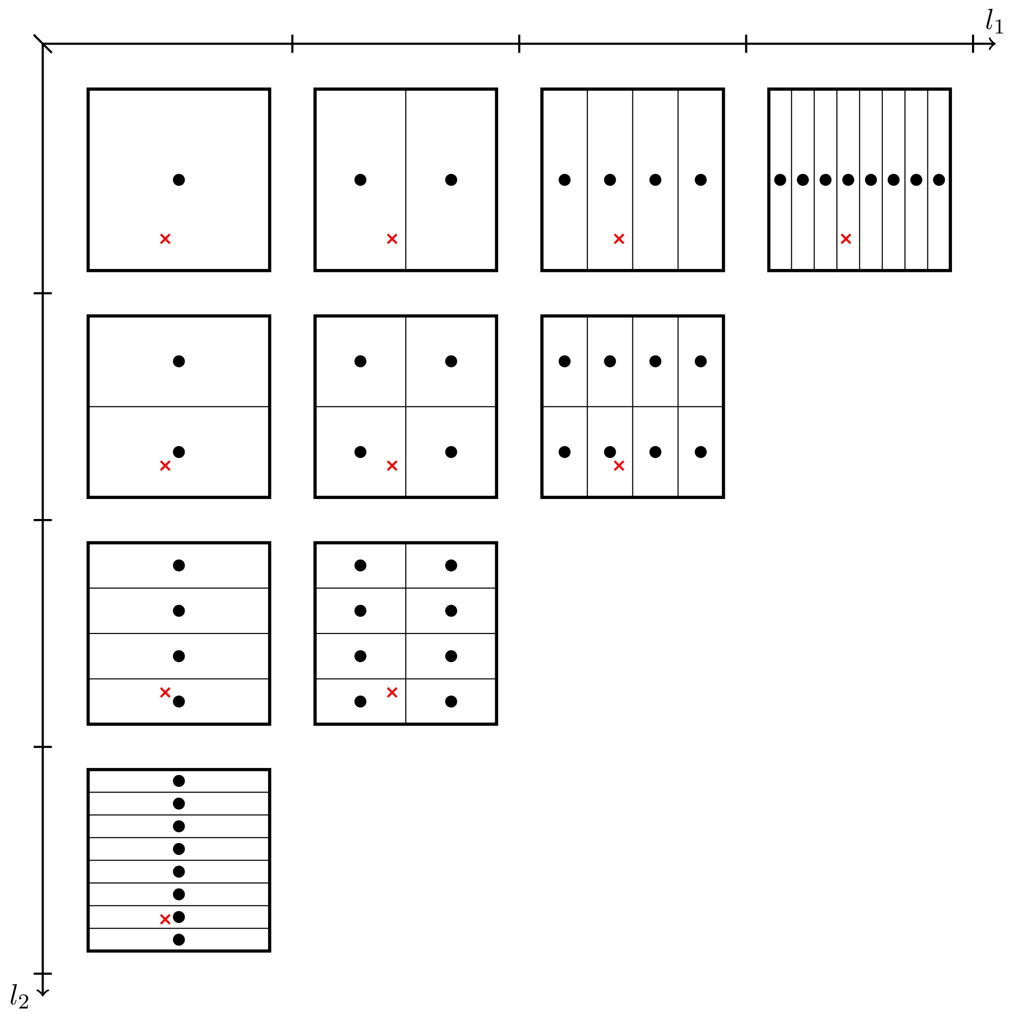


Figure 3.1: Exemplary evaluation point (red) in the subgrids scheme of a two dimensional sparse grid of level $n = 4$, plotted over both level components. [Bru16]

Fortunately, the hierarchical structure of the grid can be utilized to examine only the relevant grid nodes. The subspaces are thereby accessed in the style of a binary search tree in each dimension. That is, each grid node references two child nodes in each dimension, located on the next higher subgrid in the according dimension. Starting at the root node, the relevant grid nodes are found by traversing the subgrids recursively in every dimension, by comparing the position of the current node with the evaluation point. When descending in dimension d', the d'-th component of the evaluation point is compared to the d'-th component of the grid node. In case the evaluation point is lower, the first (left) child node will be looked at next. Else, the second (right) child node is accessed. Hence, many unnecessary calculations of the naive approach are left out. One basis function per subspace is evaluated, and the total number of subspaces is in $\mathcal{O}(n^d)$. In combination with the d-linear basis computation, as presented in algorithm 3.2, that yields a time complexity of $\mathcal{O}(dn^d)$. However, we can do even better.

As can be seen in Figure 3.1, subspaces which partly share the same level entries also share linear parts of their basis functions. In fact, when descending to a child node in dimension d', only the factor $\phi_{l_{d'},i_{d'}}$ differs, while all other parts ϕ_{l_j,i_j} with $j = 1, \dots, d'-1, d'+1, \dots, d$ are the same as before. Therefore, already computed components of the basis function can be propagated when traversing the grid nodes. Algorithm 3.3 realizes a hierarchical evaluation incorporating these improvements.

Algorithm 3.3 Implementation of a hierarchical evaluation on sparse grids at point x.

```
function EvalHierarch($x$[ ], grid, $\alpha$[ ])
    gridnode ← RootNode(grid)
    result ← 0
    RecEvalHierarch($x$, gridNode, $\alpha$, 1, 1, result)
    return result
end function

procedure RecEvalHierarch($x$[ ], gridNode, $\alpha$[ ], $d'$, basis, result)
    level ← 1
    while $true$ do
        seq ← GetSequenceNumber(gridNode)
        index ← GetIndex(gridNode, $d'$)
        h ← $2^{-\text{level}}$
        newbasis ← basis ·Max($1 - |\frac{x[d']}{h} - index|, 0$)
        if $d' = d$ then
            result ← result + $\alpha$[seq] · newbasis
        else
            RecEvalHierarch(x, gridNode, $d' + 1$, newbasis, result)
        end if
        if HasChild(gridNode, $d'$) then
            break
        end if
        if $x[d']$ < index · h then
            gridNode ← LeftChild(gridNode, $d'$)
        else
            gridNode ← RightChild(gridNode, $d'$)
        end if
        level ← level +1
    end while
    gridNode ← ResetToLevelOne(gridNode, $d' + 1$)
end procedure
```

Line 18 of Algorithm 3.3 executes a recursive call into the next dimension, where the value of already calculate shared basis functions is passed down. Only when the last dimension is reached, the basis function value is multiplied with the surplus, and is added to the result (in line 16). In the lines 23 to 27 a step into the next subspace in the respective dimension is performed, depending on the position of the evaluation point. The corresponding surplus value of a grid node is retrieved via its sequence number in the sparse grid structure. The basis function can be calculated with level and index information associated to a grid node.

By sharing already computed linear parts of the basis functions, the computational complexity of the evaluation is reduced from $\mathcal{O}(dn^d)$ to $\mathcal{O}((d-1)n^{d-1} + n^d)$ [Bru16].

3.2 Data Mining on Sparse Grids

Modern applications oftentimes produce a large amount of data, far to big to be interpreted manually by human observation. In order to gain useful information and knowledge, automated methods are required which process and analyse the data at hand. Fields that can benefit from such methods range from geological observation and weather forecasts over medical science to business [Pfl10]. The procedure of extracting relations, structures or models in data is commonly labelled as *data mining*. As many of these methods are computationally intensive and deal with high-dimensional problems, the concept of sparse grid promises potential.

One category of data mining techniques consists of the so-called *predictive models*, which try to predict values on new and untapped data based on information from a given data set. In this context, each data point needs to be associated with a target value that is generally unknown. In order to make accurate predictions, a set of data points with known target values has to be provided. This given data is called the *training set*, and is formally defined by

$$S := \{(\mathbf{x}_i, y_i) \in \Omega \times T\}_{i=1}^{M},$$

where each data point $\mathbf{x}_i$ is located in domain Ω and is associated with target value $y_i \in T$. It contains a total of M data points. As the prediction should work on any point with arbitrary location and target value, the training set has to contain sufficient information on the whole domain and target range. Therefore, larger training sets generally lead to a better accuracy in the prediction.

The prediction process can look different depending on the target values. Whenever a continuous function value is approximated, one speaks of *regression*. In the case of a discrete, finite set of possible target values, each data point is going to be labelled with one of these (class) values, which is referred to as *classification*. Both of these methods are investigated in the following. Basic concepts will be explained and an implementation on sparse grids will be presented.

3.2.1 Regression

The aim of regression is to approximate a continuous, and often complex, function $u : \Omega \to \mathbb{R}$, which is here assumed to be real-valued. Therefore, the training set has to contain real-valued targets as well, and is defined by

$$S := \{(\mathbf{x}_i, y_i) \in \Omega \times \mathbb{R}\}_{i=1}^{M}.$$

As explained earlier, the pre-classified training set S makes up all information at the disposal of the regression algorithm. At the same time, the approximated function should show a certain level of smoothness. Errors might occur due to inaccurate measuring or recording of the training points, leading to outliers in the data set. Approximating such erroneous data exactly can result in high individual spikes in the represented function. To counteract this behaviour, a regularization operator $R : V_n^{(1)} \to [0, \infty)$ is introduced.

The regression problem can then be formulated as the minimization problem given by

$$(3.2)\quad \tilde{u} = \arg \min_{\tilde{u} \in V_n^{(1)}} \left(\frac{1}{M} \sum_{j=1}^{M} (\tilde{u}(\mathbf{x}_j) - y_j)^2 + \lambda \cdot R[\tilde{u}] \right),$$

which is called a *least squares approach*. It minimizes the squared error between approximated function value $\tilde{u}$ and target value y_j over all training points, while incorporating an operator for regularization. The parameter $\lambda \geq 0$ serves as a weighting factor for the regularization operator. High λ values produce a function that is strongly smoothed. Setting $\lambda = 0$ nullifies the regularization.

Applying Equation (3.1) to the function approximation $\tilde{u}(\mathbf{x}_j)$ in (3.2) yields

$$(3.3)\quad \tilde{u} = \arg \min_{\tilde{u} \in V_n^{(1)}} \left(\frac{1}{M} \sum_{j=1}^{M} \left(\left(\sum_{i=1}^{N} \alpha_i \cdot \phi_i(\mathbf{x_j}) \right) - y_j \right)^2 + \lambda \cdot R[\tilde{u}] \right).$$

This least squares approach is presented in [Pfl10], where different choices for the regularization operator are discussed as well. Here, it is chosen as $R[\tilde{u}] = \sum_{i=1}^{N} \alpha_i^2$, which has the advantage that it simplifies the system to be solved in the regression problem, and therefore saves valuable computation time.

The minimization problem defined by equation (3.3) can be reformulated to retrieve a system of linear equations in the form of

$$(3.4)\quad \frac{1}{M} B\mathbf{y} = \left(\frac{1}{M} BB^{\intercal} + \lambda \mathbb{I} \right) \cdot \boldsymbol{\alpha},$$

with matrix $B \in \mathbb{R}^{N \times M}, (B)_{ij} := \phi_i(x_j)$, target values $\mathbf{y} := (y_1, \ldots, y_M)$ and the surplus coefficients $\boldsymbol{\alpha} := (\alpha_1, \ldots, \alpha_N)$ as solution vector. The full derivation is found in [Bru16]. The minimization problem (3.2), and hence the regression problem, is solved by determining $\boldsymbol{\alpha}$. Section 3.2.3 covers how the system can be solved efficiently.

3.2.2 Classification

In contrast to regression, the classification problem deals with a handful of discrete target values, also referred to as classes. The task of a classification is to assign one of these known classes to new, unclassified data points. Again, a training set S of labelled data is given as input to the problem, defined by

$$S := \{(\mathbf{x}_i, y_i) \in \Omega \times \mathbb{K}\}_{i=1}^{M}$$

with a set of class labels $\mathbb{K}$, which obviously has to be known beforehand. The training set should contain enough information on all classes in order to predict unseen data accurately.

Though a classification does not approximate a continuous function, the approach presented in section 3.2.1 for regression can be adapted to the classification problem. For that, the class labels in $\mathbb{K}$ are mapped to the real numbers $\mathbb{R}$, and a regression is performed under the hood. Assuming all data points are classified into only two classes, they can be represented by $\mathbb{K} = \{-1, 1\}$. That gives a natural mapping between $\mathbb{K}$ and $\mathbb{R}$ by the sign of a number, i.e. if the regression produces a positive value it is mapped to 1, in the other case to -1.

3.2.3 Solving the System

The *conjugate gradient* (CG) method is used to approximate the exact solution of the linear system [Pfl10][Bru16]. Starting with an arbitrary initialization, the conjugate gradient method iteratively improves the approximated solution vector, moving closer towards the exact solution in every iteration. Thereby, the approximation shows a convergence towards its optimum, meaning the more iterations have been performed, the smaller the adjustment will be. Therefore, a fixed number of iterations can produce a reasonably accurate outcome. Alternatively, an error bound for the residual can be defined as exit condition of the CG method.

As we have seen, both the regression and classification problem boil down to solving the linear system

$$\frac{1}{M}B\mathbf{y} = \left(\frac{1}{M}BB^{\intercal} + \lambda\mathbb{I}\right) \cdot \boldsymbol{\alpha}.$$

Computing and storing the complete system matrix on the right side of the equation is expensive, since the result of $BB^\intercal$ is a matrix of the form $\mathbb{R}^{N\times N}$. However, efficiency can be improved by considering how the system matrix is compounded. When applying the CG method, $BB^\intercal$ needs to be multiplied with the solution vector α in every iteration. This operation can be decomposed into two separated matrix-vector multiplication in the form of

(3.5) $\tilde{\mathbf{y}} = B^\intercal \boldsymbol{\alpha}$

and

(3.6) $\tilde{\boldsymbol{\alpha}} = B\tilde{\mathbf{y}}$.

By taking a closer look at the composition of $B^\intercal$ it becomes clear that one row in the matrix contains all basis function evaluations of the same point. Therefore, the multiplication with $\boldsymbol{\alpha}$ can be interpreted as a succession of sparse grid evaluations over all training points. As we have seen in Section 3.1, the computational cost of an evaluation benefits from the hierarchical structure of sparse grids. Operation (3.5) comes down to M hierarchical evaluations, performed by iterating over all points in the training set.

The following multiplication in (3.6) deals with a transposed version of the matrix, and is thus referred to as *transposed evaluation*. Matrix B still has the same amount of zeros, but the implementation is not as straightforward, since a row now contains one basis function evaluated at all training points. However, all improvements on the computational complexity of the regular evaluation can be applied to (3.6) as well, as discussed in [Bru16]. The hierarchical sparse grid structure is utilized to additively build up the solution vector over multiple transposed evaluations.

Some additional computations are required to solve Equation (3.4), such as the addition of the unit matrix multiplied by λ, the transposed evaluation on the left side, and extra operation performed in the CG method. Algorithm 3.4 depicts a schematic representation of the implementation.

Algorithm 3.4 Implementation of the conjugate gradient method which solves the linear system of the regression/classification problem, cited from [Bru16].

```
1: procedure CGMETHOD(MAXITERATIONS, ERRORBOUND)
2:     $\mathbf{t} \leftarrow \mathrm{B}^{\mathsf{T}}\boldsymbol{\alpha}$
3:     $\mathbf{p} \leftarrow \frac{1}{M}\mathrm{B}\mathbf{t} + \lambda\boldsymbol{\alpha}$
4:     $\mathbf{r} \leftarrow \frac{1}{M}\mathrm{B}\mathbf{y} - \mathbf{p}$
5:     $\mathbf{p} \leftarrow \mathbf{r}$
6:     $\rho \leftarrow \mathrm{REDUCE}(\mathbf{r}, \mathbf{r})$        ▷ $\rho = \sum_i r_i \cdot r_i$
7:     for $i = 1, \dots,$ MAXITERATIONS do
8:         $\mathbf{t} \leftarrow \mathrm{B}^{\mathsf{T}}\mathbf{p}$
9:         $\hat{\mathbf{p}} \leftarrow \frac{1}{M}\mathrm{B}\mathbf{t} + \lambda\mathbf{p}$
10:        $\sigma \leftarrow \mathrm{REDUCE}(\mathbf{p}, \hat{\mathbf{p}})$
11:        $\mathrm{FMA}(\boldsymbol{\alpha}, \boldsymbol{\alpha}, \rho/\sigma, \mathbf{p})$        ▷ $\boldsymbol{\alpha} = \boldsymbol{\alpha} + (\rho/\sigma)\mathbf{p}$
12:        $\mathrm{FMA}(\mathbf{r}, \mathbf{r}, -\rho/\sigma, \hat{\mathbf{p}})$
13:        $\hat{\rho} \leftarrow \mathrm{REDUCE}(\mathbf{r}, \mathbf{r})$
14:        if $\hat{\rho} <$ ERRORBOUND then
15:            break
16:        end if
17:        $\mathrm{FMA}(\mathbf{p}, \mathbf{r}, \hat{\rho}/\rho, \mathbf{p})$
18:        $\rho \leftarrow \hat{\rho}$
19:    end for
20: end procedure
```

Chapter 4

Considering Floating-Point Precision

The rapid growth of nodes in a grid with increasing level, especially in higher dimensions, has always been a hurdle for the applicability of grids in complex, real-world data mining problems. With a large number of grid nodes comes a high computational cost and storage requirement. In fact, breaking the *curse of dimensionality* was the main reason why the concept of sparse grids was originally developed [Zen90].

As discussed in chapter 2, to approximate a function via a sparse grid, each grid node (or rather the corresponding basis function) get assigned a weighting coefficient, also called the *hierarchical surplus*. All surplus values are determined in the process of hierarchisation, and are usually stored and processed in double precision. However, since floating-point operations are rather expensive to compute, it might be worth considering if a homogeneous double precision is actually needed. Using lower floating-point precisions (single/float and half) has the potential to improve the computation time significantly, and besides can lower the required memory space of the surplus array.

The main motivation for an alternative precision of the surplus values lies in the hierarchical structure of the sparse grid. When evaluating an approximated function, basis functions related to grid nodes in subspaces of higher level are stacked on top of all basis functions of their ancestors, corresponding to grid nodes on lower levels. Therefore, with increasing level the surplus values often tend to decline rapidly, and many values on high levels, and hence their influence on the evaluation result, become vanishingly small. Figure 4.1 visualizes this behaviour at the one-dimensional example of the quadratic function $f(x) = 4x * (1 - x)$, that is approximated with the help of a sparse grid of level 3. Every basis function is drawn in a colour indicating the level of the underlying grid node (blue stands for level 1, orange for level 2, grey for level 3). As can be seen, the size of a basis function, and therefore the corresponding surplus value, decreases on higher levels. While the surplus on level one has a value of 1 (since $f(0.5) = 1$), for grid nodes on level two it equals 0.25, and on level three it is 0.0625, indicating a decline with factor 4 per level. The difference between true function and approximation (space between black and grey lines), while still being significant for the performance in a realistic scenario, has shrunken to a comparatively small volume after three levels already. As this trend continues, surpluses on the highest levels get marginal, so their influence to the approximation, or at least part of their value starting at some position after binary point, might be neglectable.

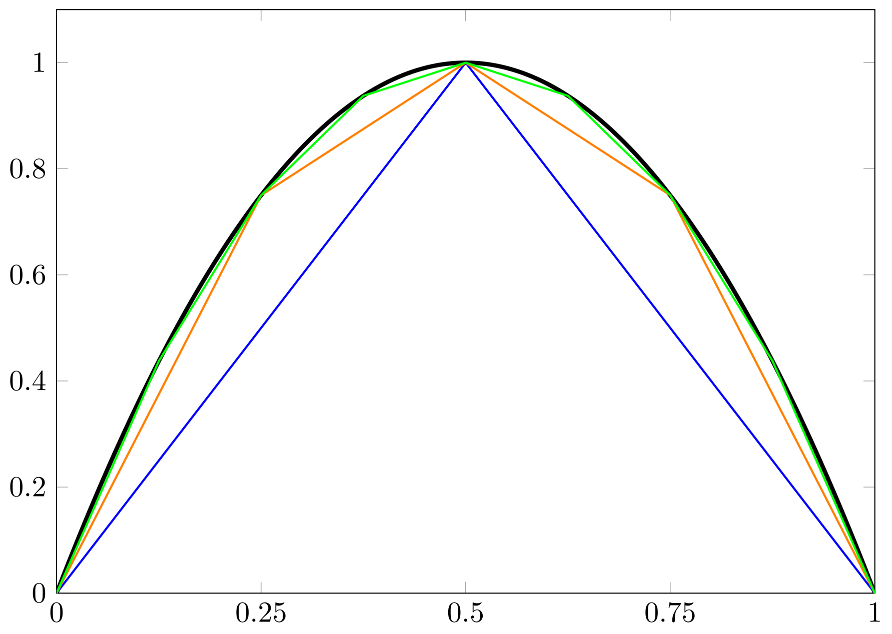


Figure 4.1: Stacked basis functions of a one-dimensional sparse grid of level 3, approximating the quadratic function $f(x) = 4x * (1 - x)$ (black). Basis function on level 1 is drawn in blue, functions on level 2 in orange, functions on level 3 in green.

While the storage in a traditional homogeneous single or half precision array is possible, and therefore will be investigated in this work (if only to serve as a reference), we are more interested in a combination of different precisions to achieve an optimal proportion of accuracy and computational cost. For a practical implementation of sparse grid algorithms this requires us to use some kind of heterogeneous array-like structure in mixed precision when dealing with the surplus values, which introduces new challenges. The rest of this chapter will firstly have a closer look at floating-point datatypes and operations in general. Secondly, we will analyse the algorithms discussed in Chapter 3 to get an idea of where relevant floating-point operations happen. Lastly, strategies for the use of mixed-precision will be presented.

4.1 Data Types and Calculations

To be able to represent and perform calculations with real numbers, a computer needs some kind of discretization. For accurate results, representable values should have a high relative resolution, but large numbers must also be possible. Aside from that, the real number representation should be rather storage efficient. Typically, for an implementation on machines the *floating-point representation* is utilized [Gol91]. There, a floating-point number is encoded by a *mantissa* m (often referred to as *significand*), a *base* b and an *exponent* e, so the value of a floating-point number x is calculated by

$$x = m \times b^e.$$

Common choices for base b are 10 (as in the scientific notation) or 2; the latter of which is especially relevant for a binary representation on a computer. Since we only have a certain amount of bits available to store a variable, the mantissa m contains a finite number of places p, and has the form

$$m = \pm m_0.m_1m_2...m_{p-1}$$

with $0 \leq m_i < b$ for $1 \leq i < p$, and $0 < m_0 < b$. Here, m_0 is assumed to be non-zero, i.e. the mantissa m is said to be normalized, to guarantee a unique representation of each number. The same holds for the exponent. Therefore, a largest exponent e_{max} and a smallest exponent e_{min} exist. Accordingly, a maximum of b^p different mantissas and $e_{max} - e_{min} + 1$ different exponents is possible.

Obviously, not every real number can be represented exactly. Some values, or the result of an operation, might have to be rounded, which in turn might introduce an error compared to the actual value. An important parameter when talking about rounding errors is the *machine epsilon* ϵ, defined by

$$\epsilon = \left(\frac{b}{2}\right) b^{-p} = \frac{b^{1-p}}{2}.$$

When rounding a value to any one of the nearest two floating-point number, the relative error, i.e. the absolute error divided by the absolute value of the number, is smaller than 2ϵ, and when rounding to the nearest floating-point number, the relative error does not exceed ϵ [Gol91]. For further information regarding floating-point representations and rounding the reader is directed to [Gol91].

In the need of a universal standard that satisfies the requirements on a real number representation, the Institute of Electrical and Electronics Engineers (IEEE) developed the *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), which was originally published in 1985 and later revised and expanded in 2008 [ZCA$^+$08]. It initially contained binary formats for floating-point numbers in single and double precision, giving the exact bit layout, as well as methods for arithmetic operations, rounding and exception handling. The revision also includes formats for half and quad precision, among other changes. After its introduction, the standard was quickly adopted by many hardware manufacturers [Gol91].

Data Type	Mantissa Bits	Exponent Bits	e_{min}	e_{max}	e_{bias}	Total Bits	ϵ
Half (FP16)	10	5	-14	15	15	16	2^{-11}
Single/Float (FP32)	23	8	-126	127	127	32	2^{-24}
Double (FP64)	52	11	-1022	1023	1023	64	2^{-53}

Table 4.1: Bit arrangement, minimum exponent, maximum exponent, bias value and machine epsilon of the floating-point data types in half, single and double precision proposed by IEEE 754.

Some important implementation details recorded in the IEEE 754 standard are depicted in the following:

- The mantissa is stored without its sign, meaning that the absolute value of m is encoded in the bit string plus an individual bit holding the sign.
- An advantage of the binary representation, i.e. base $b = 2$, is that one bit of the mantissa can be saved. Since the mantissa m is declared to be normalized, the leading place m_0 before the point has to be 1. With this information known, wasting one bit to store this place would be superfluous. Hence, only the places after the point need to be stored (the skipped bit is also called a *hidden bit*), so the required number of bits for the mantissa is reduced to $p - 1$.
- The exponent also has a signed value, however its sign is handled differently. A constant *bias* e_{bias} gets introduced, so that the result of $e + e_{bias}$ is always positive. That way, the exponent can be encoded as an unsigned integer in the form of $e + e_{bias}$. The bias obviously has to be factored in when processing a floating-point number later on.
- The process of normalization does not allow a natural representation of exactly zero. As a workaround, the binary encoding of $e_{min} - 1$ is reserved to signal a zero value. Consequently, the range of possible exponents is reduced by one.

In this work, binary (i.e. $b = 2$) formats in double, single (also referred to as float) and half precision for floating-point numbers following the IEEE standard will be considered. Table 4.1 depicts the exact bit layout and various exponent values of these data types conform to IEEE 754. One has to remember that each data type contains one fixed sign bit. Additionally, every type makes use of the hidden bit, thus the actual number of binary digits in the mantissa increases by one.

All methods and tests presented in this work were implemented in C++, which provides native support for single and double precision variables. For the use of a floating-point representation in half precision an external library had to be utilized[1]. According to the Author, it is aimed to provide a half data type conform to IEEE 754, that, in its use, is similar to comparable primitive data types (like float and double) at the best possible performance. Type conversions, standard arithmetic operation and some common mathematical functions are supported. Regarding performance however, the author writes the following: "...many of the mathematical functions provided by the library as well as all arithmetic operations are actually carried out in single-precision under the hood, calling to the C++ standard library implementations of those functions whenever appropriate, meaning the arguments are converted to floats and the result back to half. [...] Even with its efforts in reducing conversion overhead as much as possible, the software half-precision implementation can most probably not beat the direct use of single-precision computations. Usually using actual float values for all computations and temproraries and using halfs only for storage is the recommended way."[1] As a matter of fact, this renders an experimental analysis of the runtime in this work pretty much pointless. Such testing would not be properly representative anyway, since processors might be optimized for use of a certain precision, or are not equipped with a hardware implementation of half precision operations at all. Especially the use of the latter would most likely happen on specialized machines. Nevertheless, one can draw theoretical implications on a possible saving in runtime based on an estimated number of floating-point operations in various precisions.

4.2 Efficiency Considerations

4.2.1 Floating-Point Operations

At the beginning of this chapter we have seen that the hierarchical surpluses can become very small on higher levels. That directs the focus on computations which involve the hierarchical surplus. These serve as weighting coefficient for the basis functions, hence surplus and basis value are going to be multiplied at some point in time. The product will afterwards be added to the overall result of the evaluation. Naturally, summands with a smaller (absolute) value contribute less towards the result. When operating with floating-point variables, a certain number of the least-significant mantissa bits can even get lost in the addition process, depending on the difference in the exponent. Therefore, it can suffice to compute and store some summands in lower precision, without a notable drop in accuracy.

As discussed in Chapter 3, sparse grid evaluations make up a large part of the CG runtime. After the learning algorithm of the regression or classification problem has been finished, the

[1]Half-precision floating point library by Christian Rau: `http://half.sourceforge.net/`, last visited on 28.12.2018.

labelling of new data happens through individual evaluations. Consequently, the evaluation is the main focus here. The CG algorithm performs additional vector operations, but these were left untouched, i.e. they are computed in double precision. The resulting surplus values are also stored in mixed-precision.

Looking back at the hierarchical evaluation discussed in Section 3.1, the surplus-basis multiplication happens once on every subgrid. It is found in Algorithms 3.3 on line 16. In the successive addition (on the same line), the old result value obviously has to be computed in double precision, or else the higher accuracy of earlier calculations is lost. The construction of the linear basis in line 14 can be considered as well. During the level and dimension recursion, the operation level (meaning the sum over all level entries) on the sparse grid is never going to be decreased. Once a point of precision reduction is reached, the intermediate basis value can be converted and passed down in lower precision, because the floating-point precision is not going to be increased in recursive function calls.

4.2.2 Grid Nodes and Subgrids

To get an idea of how much computation time and memory can potentially be saved, we need to take a closer look at how the size of a sparse grid grows with an increasing level. Looking back at the subgrids which in combination make up the sparse grid, as formally defined by (2.2), (2.3) and (2.4) and exemplary illustrated in Figure 4.2, it can be seen that by increasing the level n of a d-dimensional sparse grid $\Omega_n^{(1)}$ by one, a new d-dimensional layer of subgrids

$$\hat{L}_n := \{\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \hat{\Omega}_\mathbf{l} \mid |\mathbf{l}|_1 = d + n - 1\}, \tag{4.1}$$

is appended to the grid. This layer simply contains all subgrids with the new maximum level sum. Each of these subgrids contains exactly 2^{n-1} nodes, as directly implied by Equation (2.2). To calculate the number of distinct subgrids on layer $\hat{L}_n$ one can think of reaching a subspace by following a path of $n-1$ hierarchical steps, each of which can be performed in one of the d dimensions (see right side of Figure 4.2). However, the order of successive steps should not be taken into account, since that would lead to multiple paths ending at the same subgrid (indicated by dashed lines). In other words, we are searching for a combination of $n-1$ elements from a set of d basic steps, where an unique element can be repeated arbitrarily, and the order of elements is disregarded. In the area of combinatorics, this concept is referred to as a *k-combination with repetitions* (or *k-multicombination*). The number of possible combinations, and therefore the quantity of distinct subgrids on layer $\hat{L}_n$ is given by

$$\left(\!\!\binom{d}{n-1}\!\!\right) := \binom{d+n-2}{n-1} = \frac{(d+n-2)!}{(d-1)!(n-1)!}. \tag{4.2}$$

Hence, the highest layer $\hat{L}_n$ comprises

(4.3) $|\hat{L}_n| = \dfrac{(d+n-2)!}{(d-1)!(n-1)!}\, 2^{n-1}.$

grid nodes, and in total, the sparse grid contains

(4.4) $|\Omega_n^{(1)}| = \sum_{i=1}^{n} |\hat{L}_i| = \sum_{i=1}^{n} \dfrac{(d+i-2)!}{(d-1)!(i-1)!}\, 2^{i-1}.$

nodes.

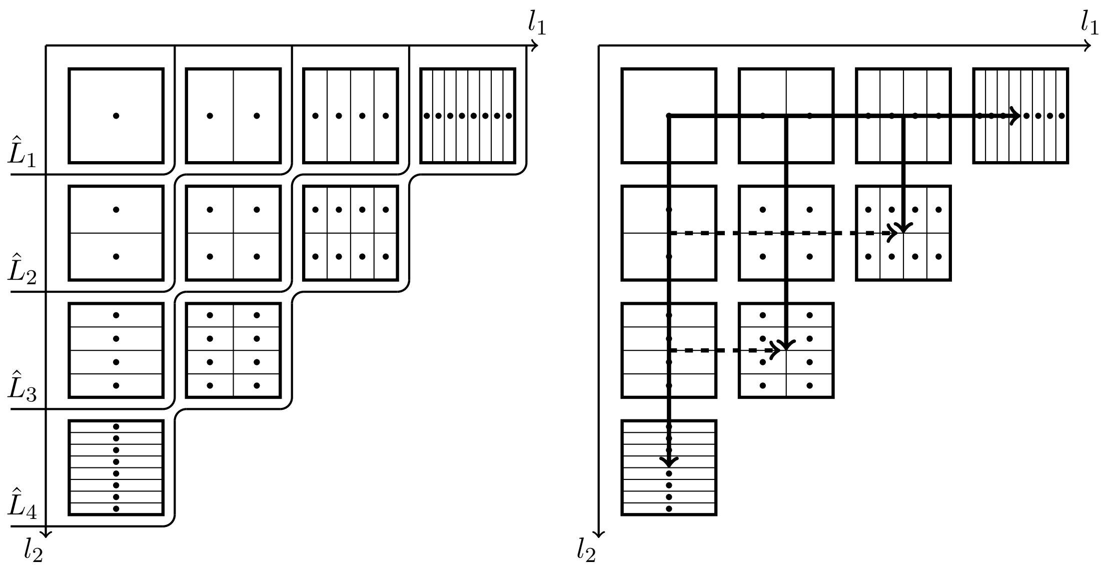


Figure 4.2: Subgrid scheme of a two-dimensional sparse grid up to level $n = 4$, highlighting the subgrid layers $\hat{L}_1$ - $\hat{L}_4$ (left) and construction paths (right).

4.3 Mixed-Precision Strategies

When we want to save storage space, and possibly computation time, by using a different precisions, the question of how to choose the fitting precision to store and compute a certain value remains. Obviously, we want to store important surplus values, i.e. big values that have a large impact on the result of a computation, in high precision, while for smaller values it might be sufficient to store them in lower precision. Hence, the simultaneous use of different precision might be an interesting idea. As said before, double (FP64), single (FP32) and half (FP16) are considered.

The approaches are classified into two categories:

1. **Homogeneous**: a uniform precision is chosen for the complete array
2. **Level dependent**: the precision choice is based on the level sum of the corresponding grid node

All mixed-precision strategies investigated in this work will be presented and explained in more detail below.

4.3.1 Homogeneous

The straightforward approach is to store all surplus values in a simple array, where one fixed precision is set throughout. The traditional choice is a homogeneous double array. A huge runtime and storage save can be achieved by using only single or half precision. Nevertheless, these alternatives are most likely unfavourable, because the big loss in precision makes it impractical for a real life situation. Still these three methods were tested to have a general reference for the other strategies regarding accuracy and efficiency. In the later test chapter, the homogeneous strategies are often referred to as simply *double*, *single* and *half* for the sake of convenience.

4.3.2 Level Dependent

As earlier discussed, surplus values decline rapidly on higher levels. Therefore, a more sophisticated approach would be to make use of the hierarchical sparse grid structure. Each grid node $\mathbf{x}_{\mathbf{l},\mathbf{i}}$ and hierarchical surplus $\alpha_{\mathbf{l},\mathbf{i}}$ exists on a multi-valued level $\mathbf{l}$. To get an idea of how deep the node is positioned in the hierarchy, one can look at the sum over all of its individual level entries $|\mathbf{l}|_1$, also called the *level sum*. We then introduce two threshold values t_{single} and t_{half} at which the precision gets lowered. That is, $\alpha_{\mathbf{l},\mathbf{i}}$ with $|\mathbf{l}|_1 < t_{single}$ are stored in double, $\alpha_{\mathbf{l},\mathbf{i}}$ with $t_{single} \leq |\mathbf{l}|_1 < t_{half}$ in single, and $\alpha_{\mathbf{l},\mathbf{i}}$ with $|\mathbf{l}|_1 \geq t_{half}$ in half precision. Of course, not all available precisions necessarily need to be utilized.

Three specific implementations following the level dependent approach are examined in the tests presented in Chapter 5. Each has been assigned with a short, representative name:

1. **Level_DSH**: A mixture of all precision (double, single and half) is used. Surplus values belonging to grid nodes with the highest possible level sum are stored in half precision, values on the second highest level are stored in single precision, and all other values are stored in double precision. For a given sparse grid $\Omega_n^{(1)}$, it is $t_{single} = d + n - 2$ and $t_{half} = d + n - 1$. Looking back at Equation (4.1), that means half precision is applied to all $\alpha_{\mathbf{l},\mathbf{i}}$ associated to $\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \hat{L}_n$, single precision to $\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \hat{L}_{n-1}$, and double to $\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \{\hat{L}_m \mid 1 < m \leq n - 2\}$ (see Figure 4.3).

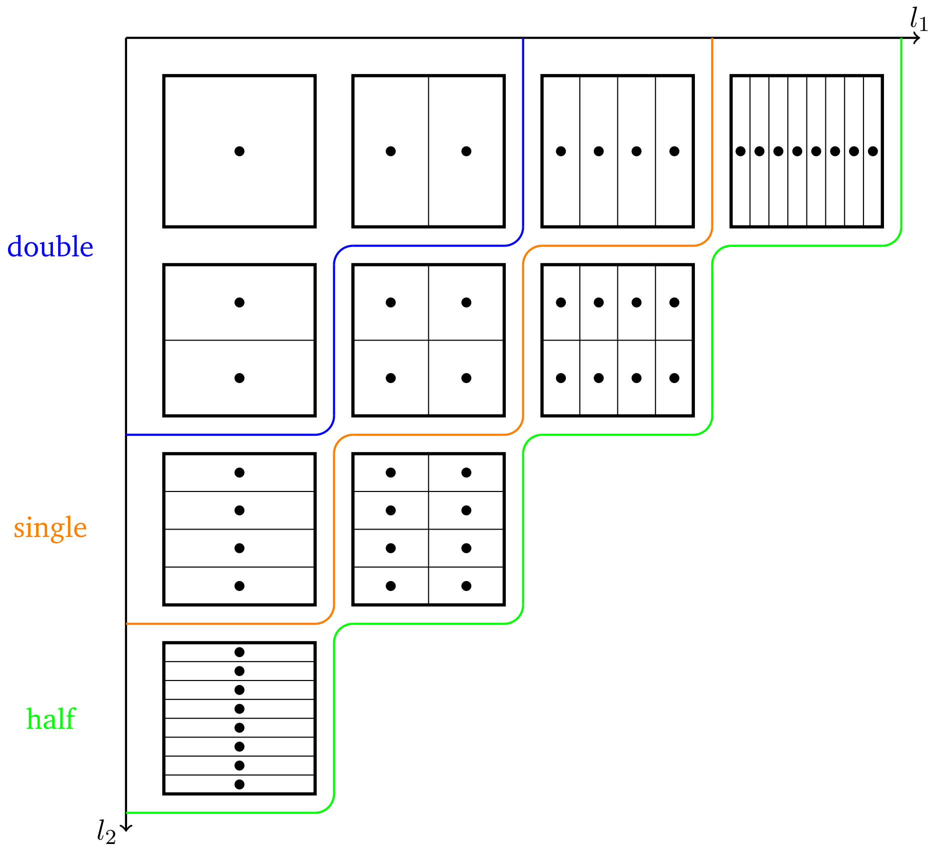


Figure 4.3: Visualization of *Level_DSH* in the subspace scheme of a two-dimensional sparse grid of level four. The strategy utilizes three different floating-point precisions: half on the highest (green), single on the second highest (orange), and double on all lower levels (blue).

2. **Level_DS1**: Only double and single precision are considered. Surplus values with the highest possible level sum are stored in single precision, while all other values are stored in double precision. That is, $t_{single} = d+n-1$. Single precision is associated to $\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \hat{L}_n$, and double to $\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \{\hat{L}_m \mid 1 < m \leq n-1\}$ (see Figure 4.4).

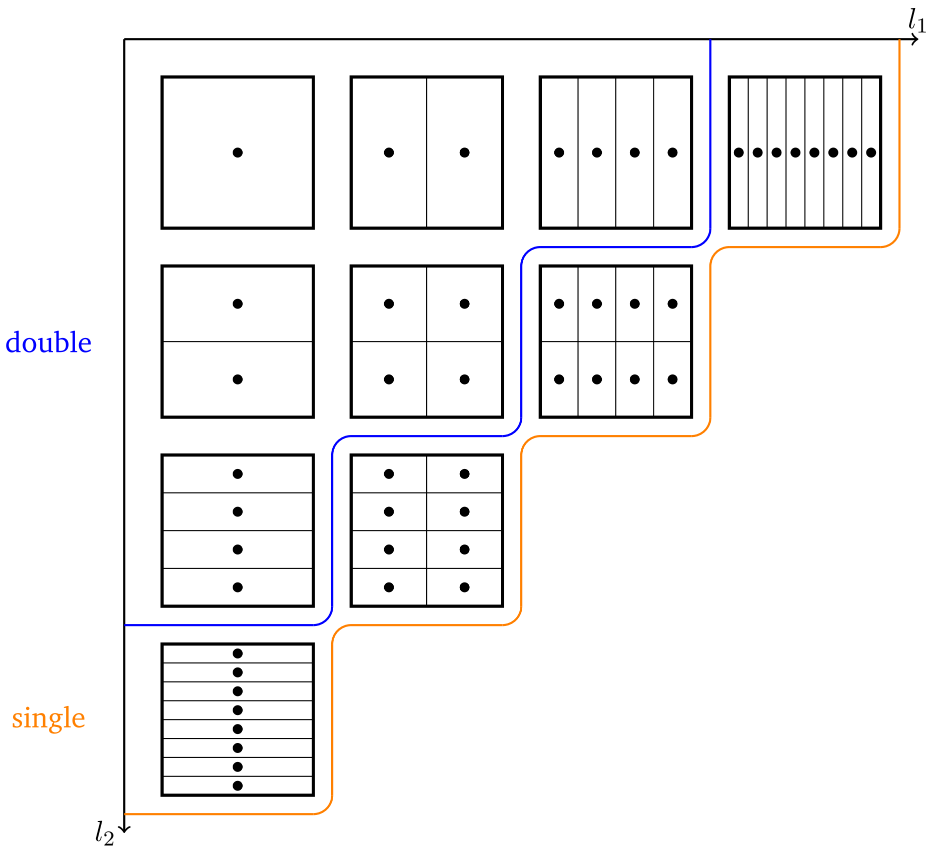


Figure 4.4: Visualization of *Level_DS1* in the subspace scheme of a two-dimensional sparse grid of level four. The strategy utilizes two different floating-point precisions: single on the highest (orange), and double on all lower levels (blue).

3. **Level_DS2**: A similar strategy as *Level_DS1*, with the difference that grid nodes on the two highest levels, i.e. $t_{single} = d + n - 2$, and $\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \hat{L}_n$ and $\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \hat{L}_{n-1}$ (see Figure 4.5), were implemented in single precision.

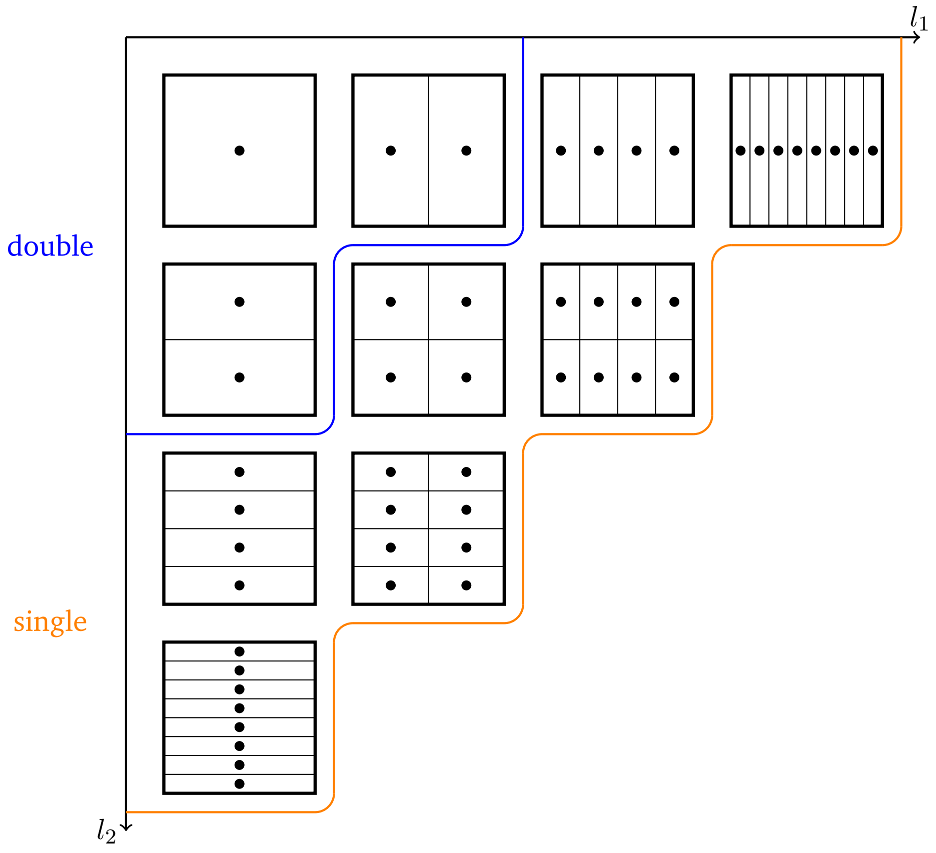


Figure 4.5: Visualization of *Level_DS2* in the subspace scheme of a two-dimensional sparse grid of level four. The strategy utilizes two different floating-point precisions: single on the two highest (orange), and double on all lower levels (blue).

Figure 4.6 depicts the storage size required for the surplus array over an increasing grid size in the different precision strategies. They are exemplary plotted in two, four and five dimensions, as these are the dimensions of the data set used in the tests. Obviously, single requires twice the size of half, and double likewise needs twice the size of single. The size of strategies incorporation only double and single precision, Level_DS1 and Level_DS2, logically are bounded by the respective homogeneous strategies. Analogously, Level_DSH is set to lie between double and half. It can clearly be seen that the mixed strategies come closer to their lower bounds in higher dimension, relatively speaking, which follows from the findings of Section 4.2.

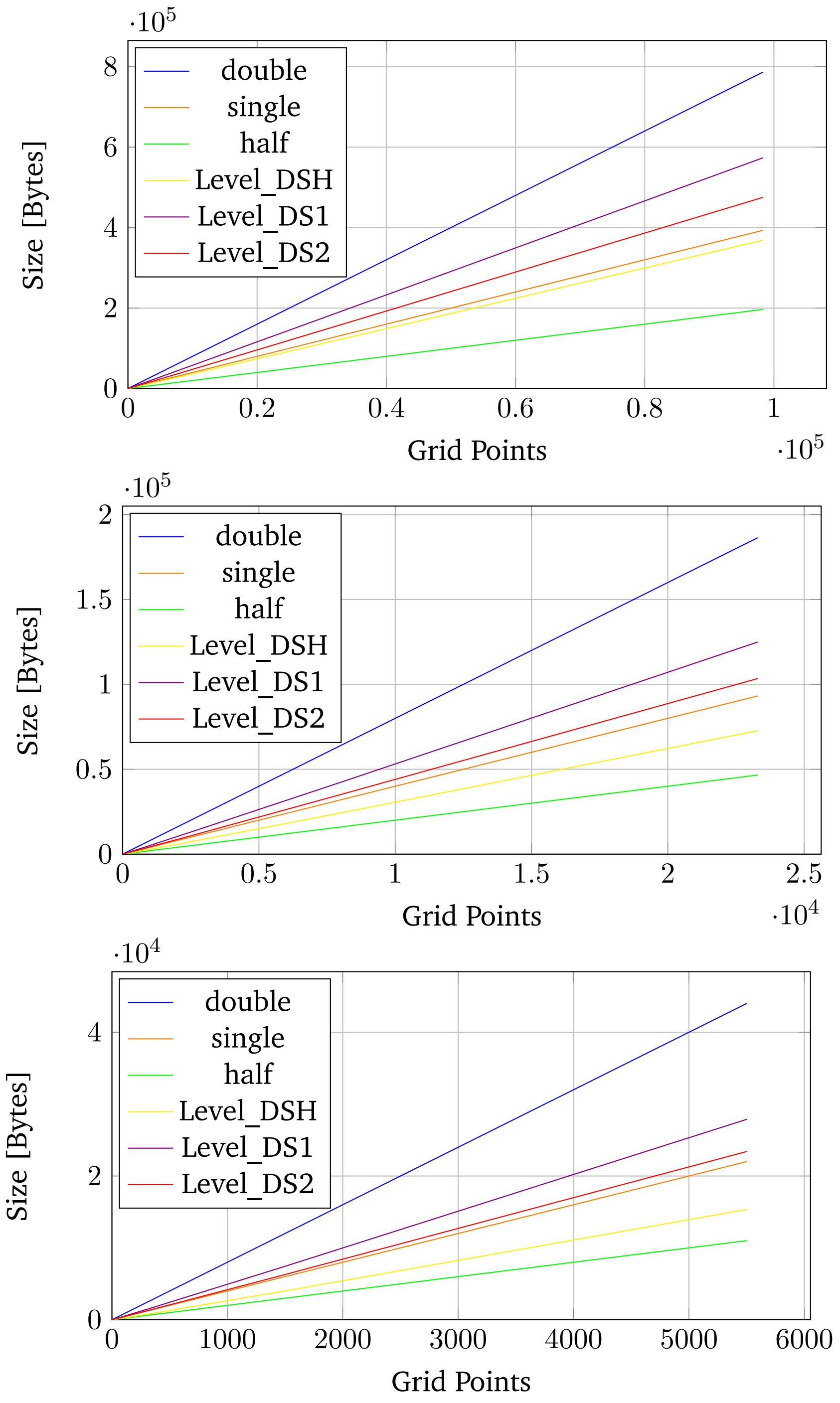


Figure 4.6: Storage size in bytes required to store the surplus array. It is plotted over an increasing number of grid nodes on a regular sparse grid in 2D (top), 4D (middle) and 5D (bottom). Different strategies for floating-point precision are shown.

Chapter 5

Test Results

The following chapter presents test results regarding the classification and regression methods on sparse grids as described above. Different strategies for choosing the floating-point precision of the surplus values, described in Section 4.3, were considered and will be compared to each other. For the tests, three different data sets were consulted: a self-generated classification set in the form of a checkerboard, presented in Section 5.1, the artificial Friedman set [Fri91], presented in Section 5.2, and the real-world DR5 set [AAA$^+$07], presented in Section 5.3; all of which have been investigated in the context of data mining on sparse grids before (see [Pfl10] and [Bru16]).

The *mean-squared error* (MSE) serves as a tool to measure the accuracy of a regression. As suggested by its name, for an approximated function u, MSE calculates the average of the squared error e_{MSE} compared to a given set of data points, in this case the testing set, containing M points y_j:

$$e_{MSE} := \frac{1}{M}\sum_{j=1}^{M}(u(x_j) - y_j)^2.$$

Additionally, whenever a classification with discrete classes is performed, the *classification accuracy* η will be observed. In our checkerboard data set each point is mapped to one of two classes, i.e. $y_j \in \{-1, 1\}$, therefore the classification accuracy is given by

$$\text{(5.1)}\quad \eta[u] = \frac{1}{M}\sum_{j=1}^{M}\left|sgn(u(x_j)) + y_j\right|, \text{ where } \ sgn(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ -1 & \text{else,} \end{cases}$$

which essentially states the fraction of correctly classified point by checking only the sign of the approximated value. Since the classification algorithm implicitly performs a regression, the MSE can be regarded as well.

For each data set, it will be investigated how the floating-point precision influences the convergence of the CG solver. Therefore, the number of iterations to reach a fixed termination condition was recorded. The convergence tests were run over an increasing level of the sparse grid, i.e. number of grid nodes. Alternatively, the quality of a regression or classification can be judged by looking at the error and accuracy when applied to a testing set. For the accuracy test, an appropriate level considering the size and complexity of each data set was chosen, and the results will be examined over the iterations of our training algorithm. Both experiments were conducted on each data set.

As stated in Chapter 2, all experiments were performed on regular sparse grid on various levels, without any adaptive refinement, and devoid of nodes on the domain's boundary. To ensure a sophisticated testing configuration, parameters were, whenever possible, chosen as in tests from comparable literature ([Pfl10] and [Bru16]).

5.1 Checkerboard Data Set

Firstly, classification tests were conducted on a artificially generated, two-dimensional data set in the shape of a checkerboard. Therefore, the domain $[0, 1]^2$ was subdivided into three equal parts in each dimension with alternating classes. That gives a total of nine square sections belonging to one of two classes, conveniently labeled with class value $y_j \in \{-1, 1\}$. The data set is depicted in Figure 5.1. The set used to train the classification algorithm contains 100000 data point, while the testing set comprises 50000 data point, all which are randomly distributed over the hole domain. A regularization parameter of $\lambda = 0.001$ was chosen.

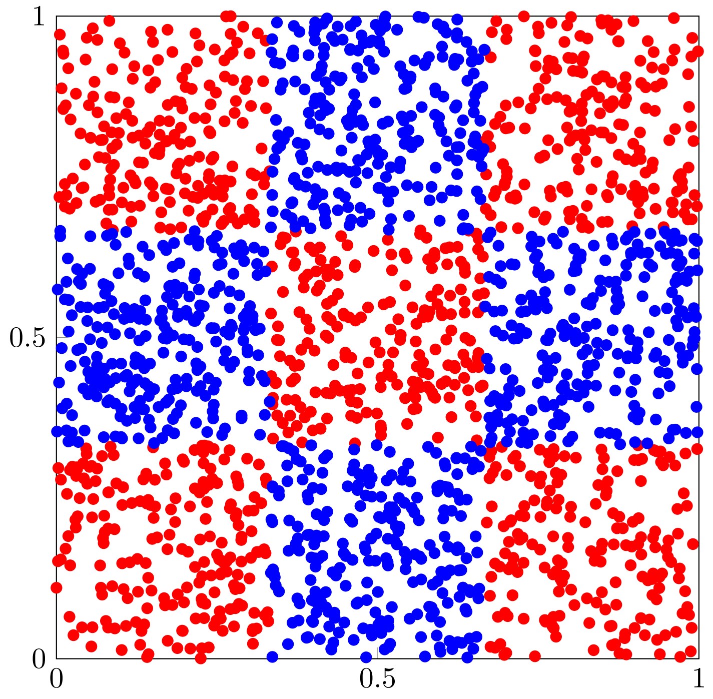


Figure 5.1: Visualization of the two-dimensional checkerboard data set. A portion of 2000 randomly distributed points in the domain $[0, 1]^2$ are displayed. Points with class value 1 are drawn in red, points with -1 in blue.

5.1.1 Convergence Tests

The plots in Figure 5.2 show the number of iterations needed to reach an error bound of 10^{-7} in the CG method used to train the classification algorithm. In the upper plot, all strategies are compared to each other, while the plots below each only show strategies of an individual approach (homogeneous, level dependent) in comparison to double. The results are plotted over the number of node in the sparse grid. All curves rise monotonously up to a grid size of 769 nodes, which corresponds to a regular sparse grid of level seven. After that point, the curves settle at a more or less constant number of iterations.

As expected, whenever single precision is utilized, the CG convergence takes slightly, yet notably, more time. The use of half precision, however, raises the number of needed iteration by a large margin. At the aforesaid point of 769 grid nodes, Level_DS1 and Level_DS2 need one additional iteration, single needs two, Level_DSH needs five and half eight, when compared to 23 double iterations. Looking at the whole plot, though, leads to the conclusion that Level_DS1 and Level_DS2 serve as somewhat of a middle ground between complete double and single precision. At the same time, they apply single precision to more than half of all surplus values.

In general, it can be seen that the curves incorporating half precision start to separate early from the others, and keep drifting further away. This implies that the use of half precision might be to much of a loss in accuracy, especially when considering that the difference between single and double precision is much smaller than between half and single, while the difference in byte size is actually greater.

Figure 5.3 illustrates the MSE and accuracy of the classification applied to the testing set. They have been tracked during the convergence test, however, all strategies achieve (almost) exactly the same result, which makes sense considering that the CG method has a constant error bound as exit condition. Parallel to the observations above, the accuracy approaches its optimum at around 750 grid nodes. The MSE, however, does not drop below a value of 0.04, which is most likely due to error on data points close to the domain's boundary.

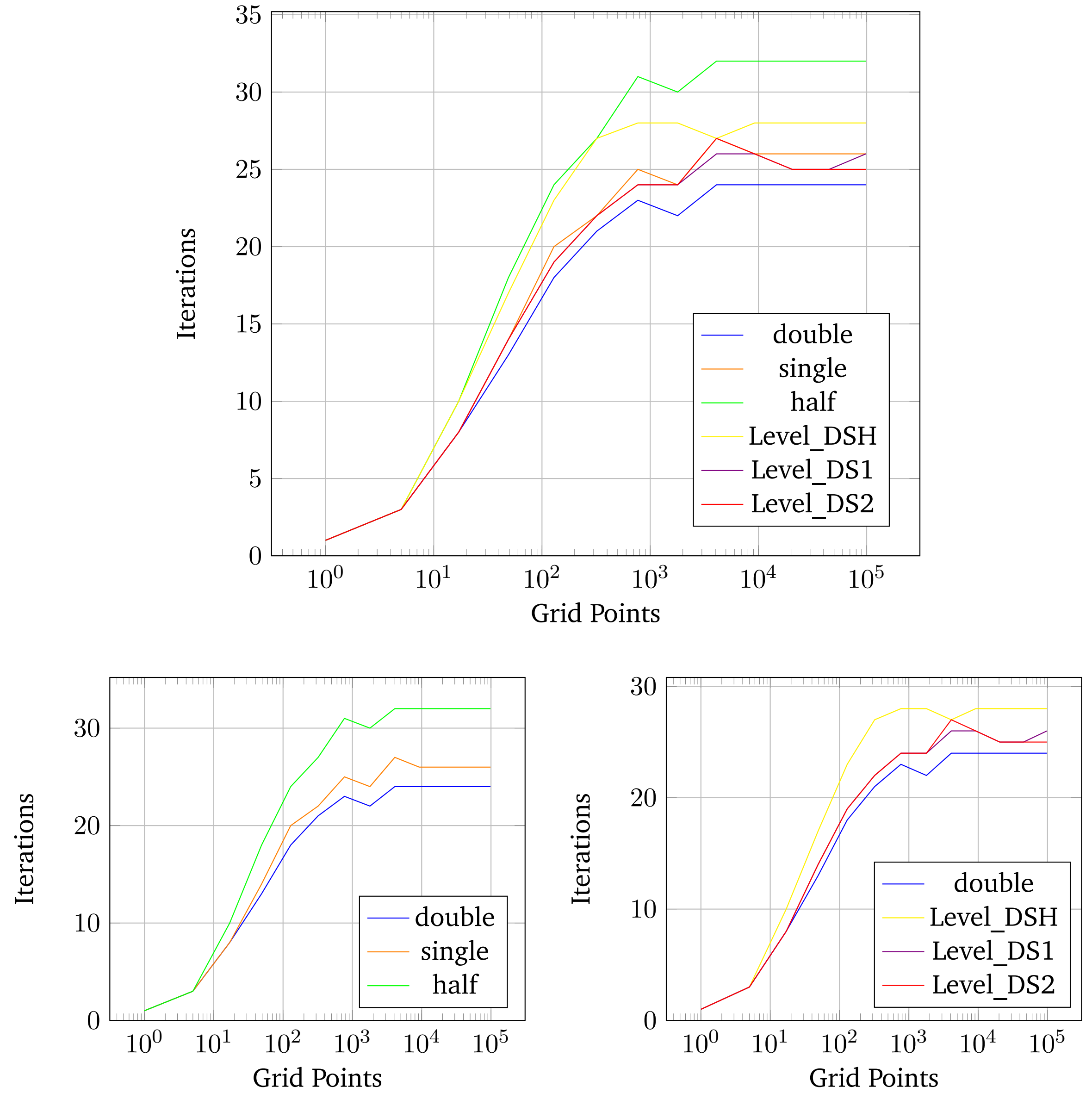


Figure 5.2: CG iterations needed to meet an error bound of 10^{-7}. Tests were conducted on a regular, two-dimensional sparse grid without boundary nodes or adaptive refinement. The checkerboard set containing 100000 data points was used to train the classification algorithm. A regularization parameter of $\lambda = 0.001$ was chosen. Different floating-point precision strategies were tested over an increasing size of the grid. The upper plot contains all different strategies. Below, the results are split into multiple plots, each comparing strategies of different approaches, homogeneous (left) and level-dependent precision (right).

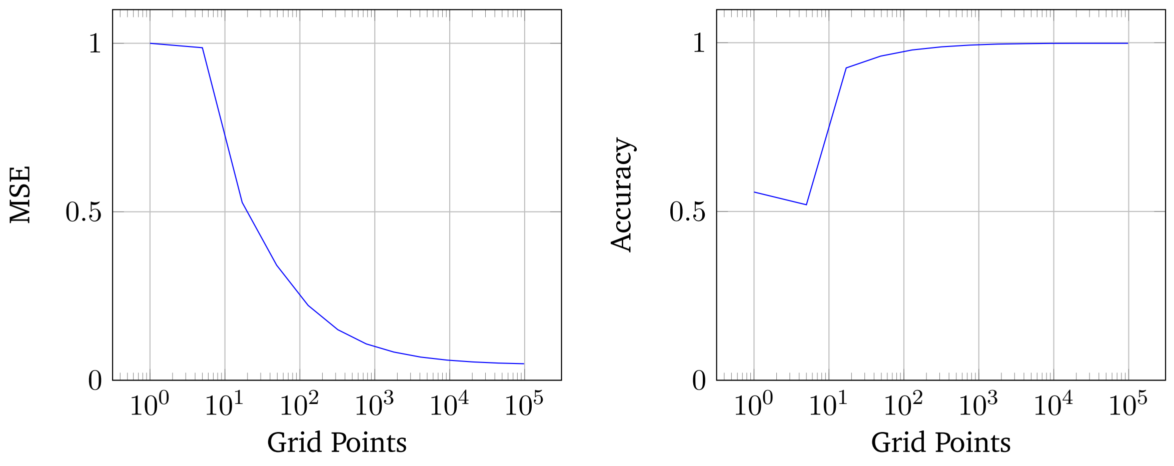


Figure 5.3: Mean-Square-Error (MSE) (left) and classification accuracy (right) in the convergence test (see Figure 5.2), plotted over the number of grid nodes. From the checkerboard data set, the training set with 100000 data points was used to train the classification algorithm, and the testing set with 50000 data points was used to measure the MSE and accuracy. All precision strategies perform practically identical, therefore only one curve is drawn.

5.1.2 Error Tests

Investigating the classification accuracy and MSE over the number of iterations provides information on the performance when a fixed, uniform number of iterations was set in the learning algorithm. Thereby, it is guaranteed that alternative strategies do not take more computation time than double precision, even on platforms where floating-point operations in lower precisions yield no timely benefits. In the tests on the checkerboard data set a regular sparse grid on level seven, therefore containing 769 nodes, was used, based on the observations of the convergence test.

Figure 5.4 shows the test result in the form of classification accuracy, MSE and CG error norm, plotted over the number of fixed iteration. Again, all strategies introduced in Section 4.3 have been tested, though the result have been spread onto multiple subplots for better visibility. Additionally, the display area of the y-axes were scaled down so the curves can be better distinguished from each other. As the MSE and accuracy plots indicate, whenever half precision is incorporated, the classification algorithm needs to execute notably more iterations in order to achieve the same quality. Surprisingly, single, Level_DS1 and Level_DS2 can, for the most part, not be distinguished from homogeneous double precision. When looking at the numbers however, it can be seen that these strategies still produce a slightly greater error.

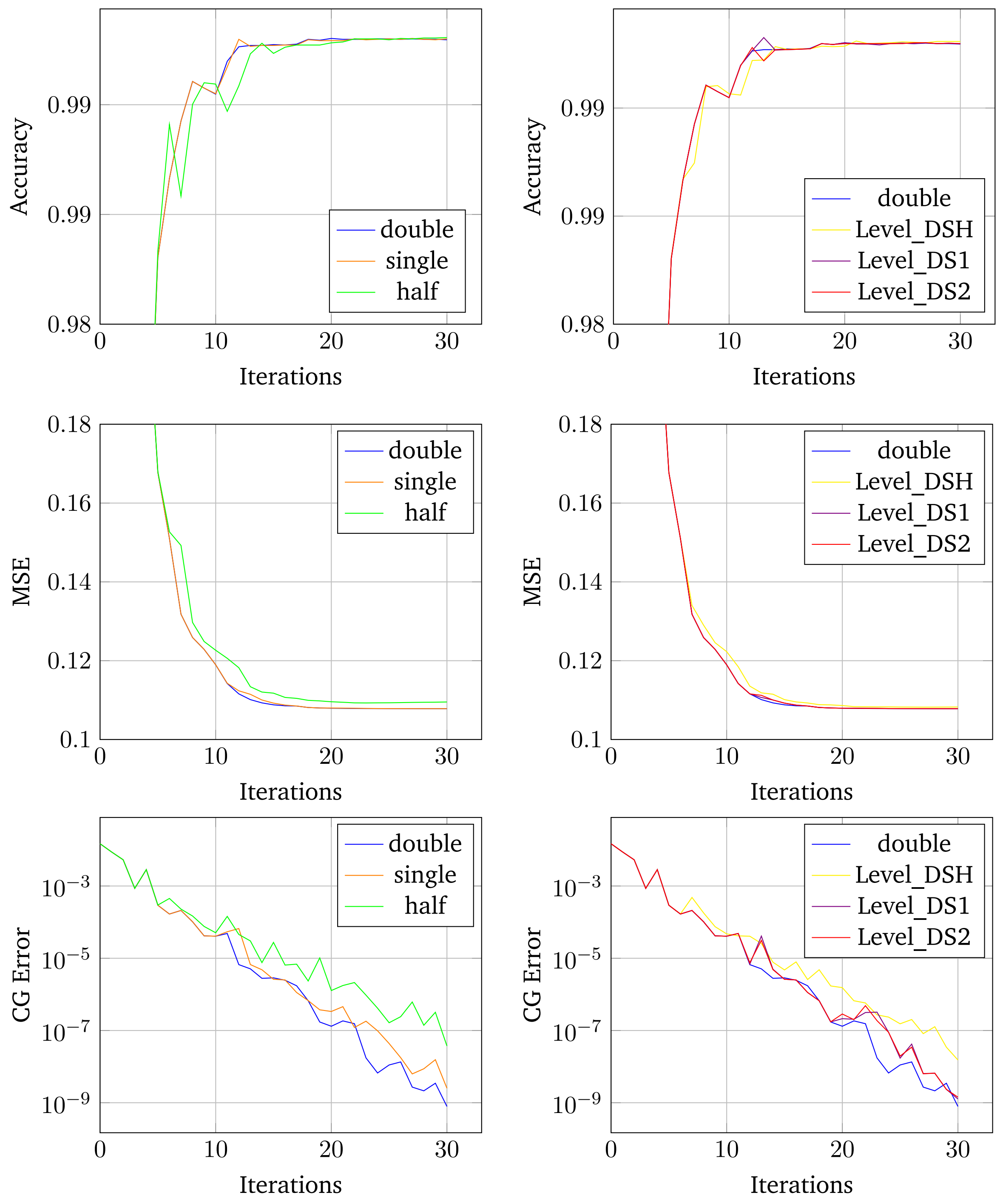


Figure 5.4: Classification accuracy (top), MSE (middle) and error norm of the CG method (bottom) of the classification are shown over the number of executed CG iteration. Tests were conducted on a regular, two-dimensional sparse grid, containing 769 grid nodes, without boundary nodes or adaptive refinement. The checkerboard set containing 100000 training and 50000 testing point was used. A regularization parameter of $\lambda = 0.001$ was chosen. Different floating-point precision strategies were tested. The result have been split into homogeneous (left) and level-dependent strategies (right).

While lower precisions never quite reach the minimal MSE of uniform double precision, in the end all strategies converge to an equal level of classification accuracy. This indicates that the use of single, and partially even half, precision might actually be sufficient to classify simple data such as the checkerboard set.

5.2 Friedman Data Set

As the classification algorithm implicitly performs a regression, one could also approximate real-valued function directly, without grouping all data points into discrete classes. Additionally, random noise can be introduced to simulate real-world conditions. One data set retrieved by following this approach is the Friedman data set, named after its originator Friedman [Fri91]. It is retrieved by evaluating the function

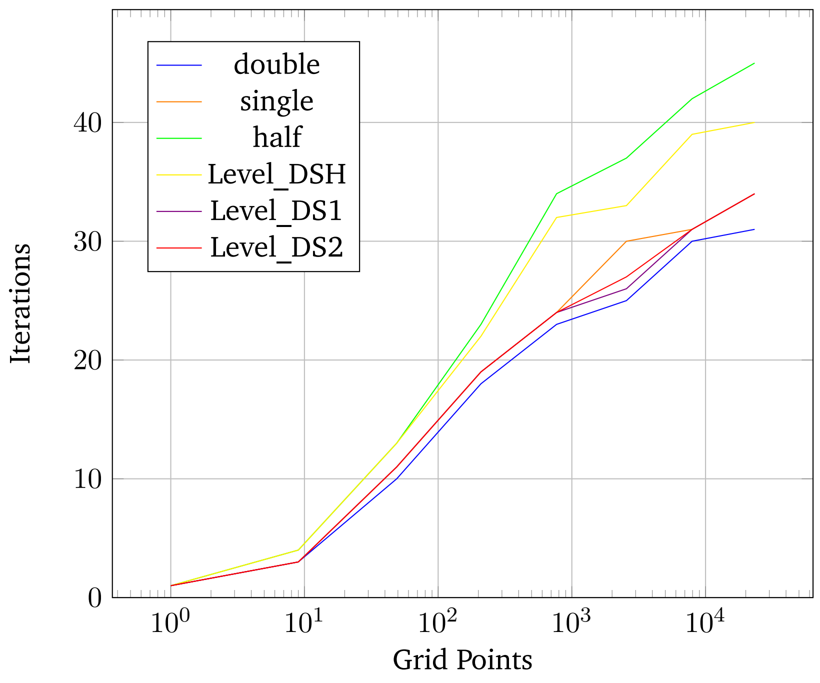
$$\arctan\left(\frac{x_2x_3 - (x_2x_4)^{-1}}{x_1}\right) + \epsilon, \tag{5.2}$$

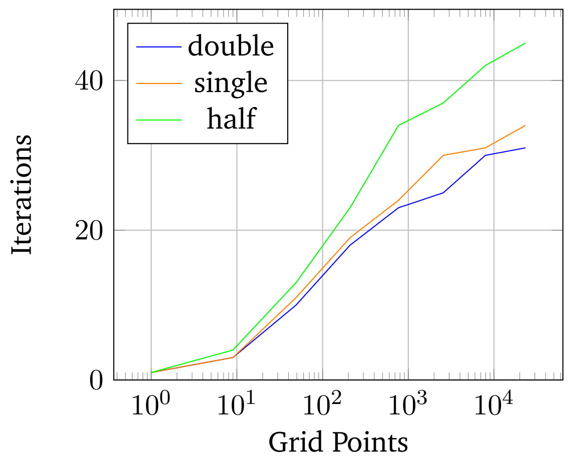
with random noise $\epsilon \in \mathcal{N}(0, 0.1)$. Since the noise has a standard deviation of 0.1, it is expected that the MSE has an optimum of 0.01. Similar to tests in [Bru16], a training set of 90000 and a testing set of 10000 data points have been generated by evaluating (5.2) at random positions in $[0, 1]^4$. The regularization parameter used was $\lambda = 10^{-4}$.

5.2.1 Convergence Tests

Figure 5.5 illustrates the results of the convergence test, in a presentation similar to the results of the checkerboard data set. Again, an error threshold of 10^{-7} was set as exit condition of the CG solver. As can be seen, the number of iterations increases rather steadily over a growing grid size. At around 800 grid nodes a point is reached after which the incline slightly decreases.

Parallel to the earlier results of the checkerboard data set, half precision performs worse than single precision, which itself performs below double precision. This time however, the discrepancy between strategies utilizing half precision and the rest is bigger. The curve of Level_DSH resides close to the curve of homogeneous half precision. Again, Level_DS1 lies between single and double, while Level_DS2 performs similar. These facts further support the observations made in Section 5.1, as here a more complex function was approximated. Analogously, the MSE, depicted in Figure 5.6 behaves as expected.





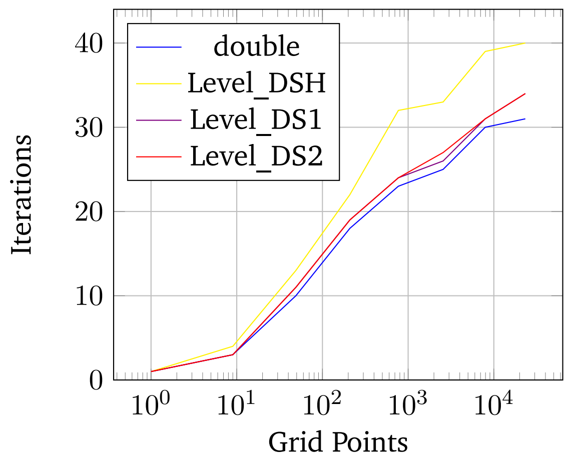


Figure 5.5: CG iterations needed to meet an error bound of 10^{-7}. Tests were conducted on a regular, four-dimensional sparse grid without boundary nodes or adaptive refinement. The Friedman set containing 90000 data points, with normally distributed noise of standard deviation 0.1, was used to train the regression algorithm. A regularization parameter of $\lambda = 10^{-4}$ was chosen. Different floating-point precision strategies were tested over an increasing size of the grid. The upper plot contains all different strategies. Below, the results are split into multiple plots, each comparing strategies of different approaches, homogeneous (left) and level-dependent precision (right).

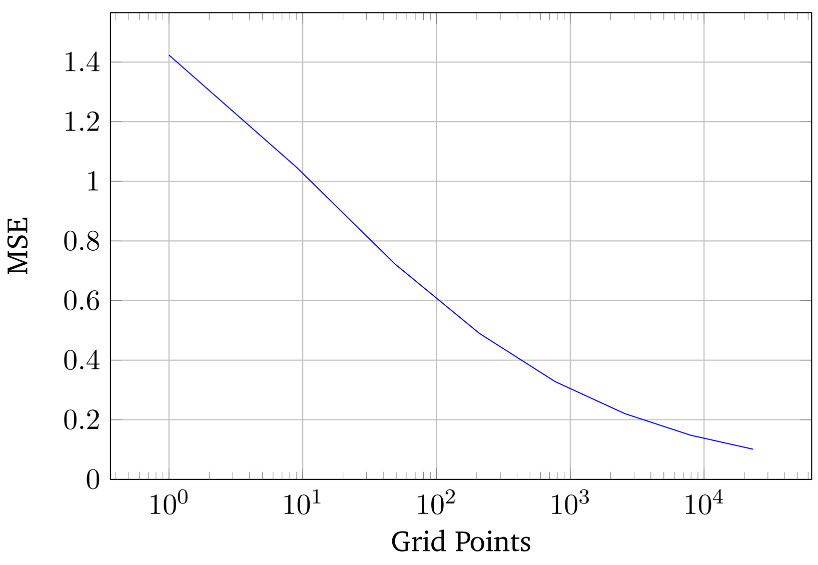


Figure 5.6: Mean-Square-Error (MSE) in the convergence test (see Figure 5.5), plotted over the number of grid nodes. From the Friedman data set, the training set with 90000 data points was used to train the regression algorithm, and the testing set with 10000 data points was used to measure the MSE. All precision strategies perform practically identical, therefore only one curve is drawn.

5.2.2 Error Tests

Since the Friedman data set is real-valued, the classification accuracy as defined by (5.1) has no meaningful interpretation here, and will therefore not be considered. Tests were conducted on a regular sparse grid on level five, yielding 769 nodes in four dimensions. Figure 5.7 depicts the result as MSE and CG error norm over the executed iterations. Again, the observations of the checkerboard section can adopted to the test results on the more complex Friedman set. The deviation of alternative strategies compared to double precision becomes slightly more apparent. Level_DS1 and Level_DS2 perform practically identical.

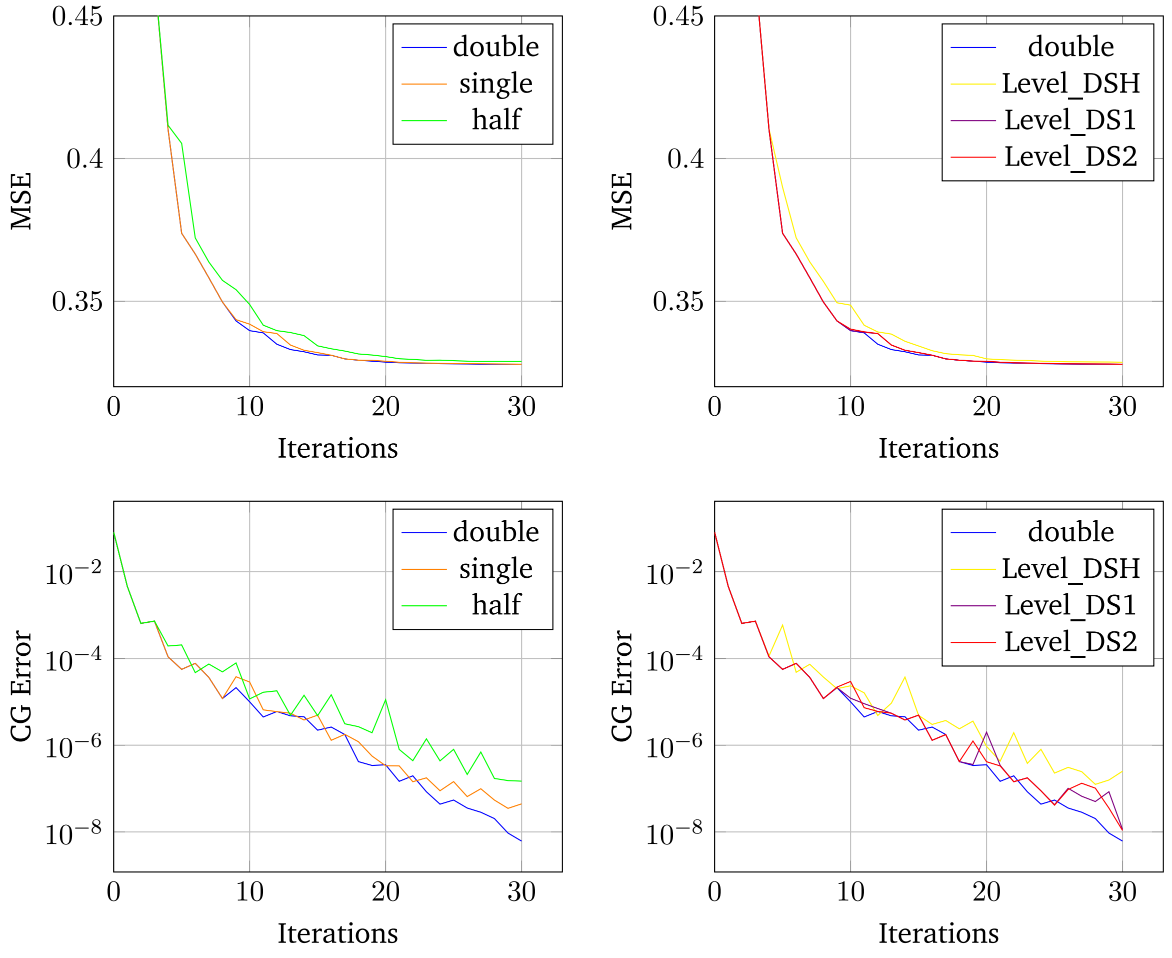


Figure 5.7: MSE (top) and error norm of the CG method (bottom) of the regression are shown over the number of executed CG iteration. Tests were conducted on a regular, four-dimensional sparse grid, containing 769 grid nodes, without boundary nodes or adaptive refinement. The Friedman set containing 90000 training and 10000 testing point was used. A regularization parameter of $\lambda = 0.0001$ was chosen. Different floating-point precision strategies were tested. For the sake of clearness, the result have been split into homogeneous (left) and level-dependent strategies (right).

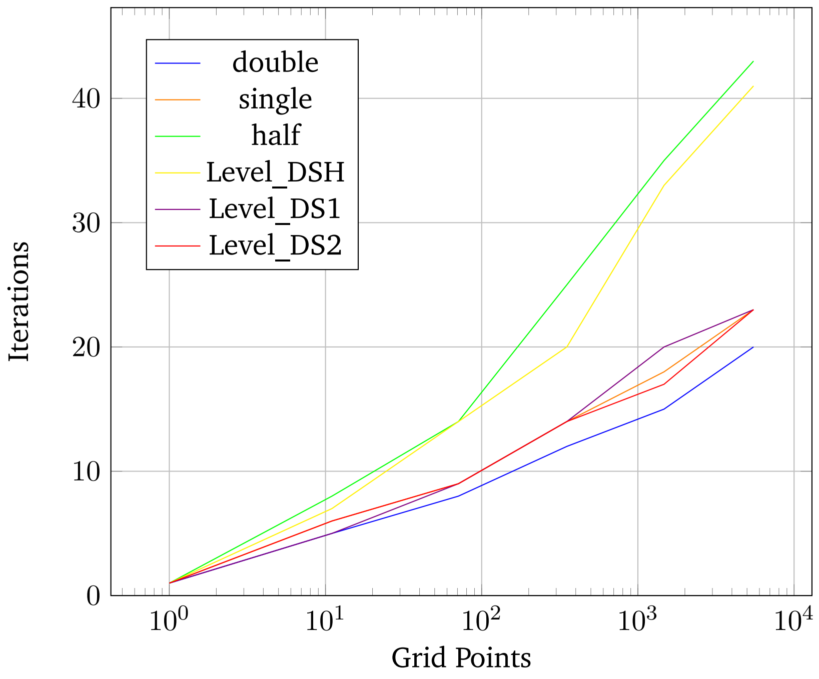
5.3 DR5 Data Set

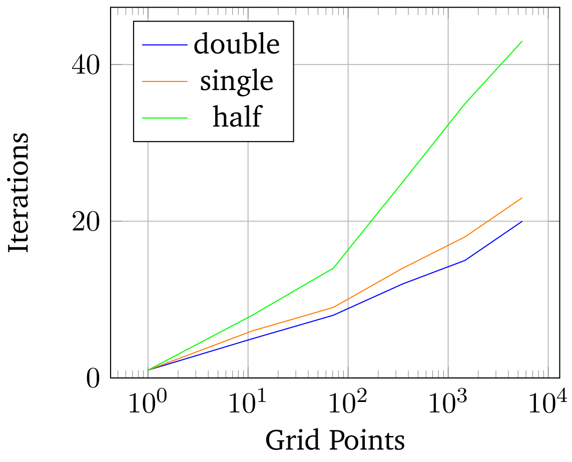
Another well-known data set in the data mining community is DR5, recorded by the *Sloan Digital Sky Survey* (SDSS) [AAA$^+$07], a astrophysical sky survey project. The real-world data set is composed of photometric measurements taken in order to gain information about the travel speed of distant galaxies. It contains a multitude of five-dimensional data. In this work, 300000 points for training, and 60000 points for the testing set were extracted, each of which was scaled to fit in our domain $[0, 1]^5$. The regularization parameter has been chosen as $\lambda = 5 \cdot 10^{-4}$. One thing to note about the DR5 set is, however, that its point only lie in a small portion of the approximation space. That makes the regression a difficult task for sparse grids, especially when no adaptive refinement is applied [Pfl10].

5.3.1 Convergence Tests

Figure 5.8 shows the test results in the earlier established style. On the DR5 data set, the curves seem to never reach a remarkable inflection point. This is probably due to the data set being so complex and concentrated to a smaller area. Many grid node are wasted on ”empty” space, while in the important regions, even on higher levels the grid is not dense enough to approximate the function adequately. In a practical scenario, an adaptive refinement of the sparse grid would be inevitable. Again, Level_DS1 and Level_DS2 perform identical to single precision for the most part. Considering the complexity of the DR5 data set, it is remarkable that all strategies incorporating single precision usually do not need more than three extra iterations when compared to homogeneous double precision.

Nevertheless, the result coincide much with the observations made on the Friedman set in Section 5.2. For the sake of integrity, the MSE during the convergence test is shown in Figure 5.9.





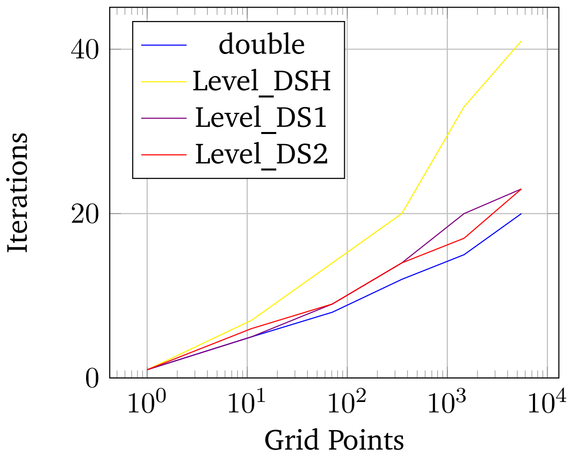


Figure 5.8: CG iterations needed to meet an error bound of 10^{-7}. Tests were conducted on a regular, five-dimensional sparse grid without boundary nodes or adaptive refinement. A subset of the real-world DR5 set containing 300000 data points was used to train the regression algorithm. A regularization parameter of $\lambda = 5 \cdot 10^{-4}$ was chosen. Different floating-point precision strategies were tested over an increasing size of the grid. The upper plot contains all different strategies. Below, the results are split into multiple plots, each comparing strategies of different approaches, homogeneous (left) and level-dependent precision (right).

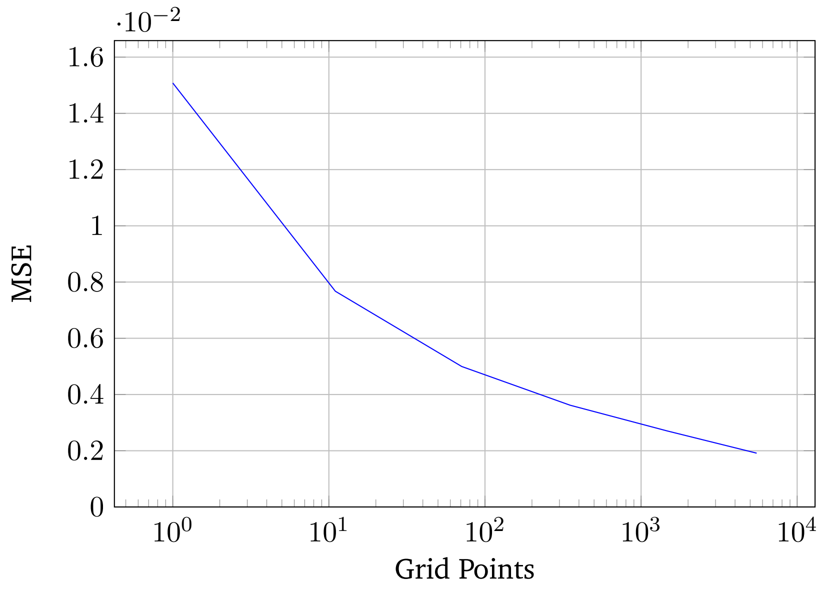


Figure 5.9: Mean-Square-Error (MSE) in the convergence test (see Figure 5.8), plotted over the number of grid nodes. From the DR5 data set, the training set with 300000 data points was used to train the regression algorithm, and the testing set with 60000 data points was used to measure the MSE. All precision strategies perform practically identical, therefore only one curve is drawn.

5.3.2 Error Tests

On the DR5 data set, the sparse grid has been chosen to be bigger than in the previous error test, due to the fact that the set is more complex and concentrated. The grid has a size of 1471 nodes. As can be seen in Figure 5.10, the achievement of homogeneous double precision is clearly superior to the rest. Where single, Level_DS1 and Level_DS2 performed similar to double on the Friedman and checkerboard set, they converge much later for DR5. Double precision comes close to its optimal MSE after around 15 CG iterations. Single needs about five to seven, Level_DS1 and Level_DS2 only around three to five extra iterations to reach a comparable accuracy. Half and Level_DSH on the other hand, both need over 25 iterations to converge, and always produce a MSE considerably larger than the other strategies. On a more general note, the MSE on DR5 is overall much smaller than compared to Friedman and checkerboard. That is probably due to the fact, that the DR5 set contain less data points which lie close to the domain's boundary.

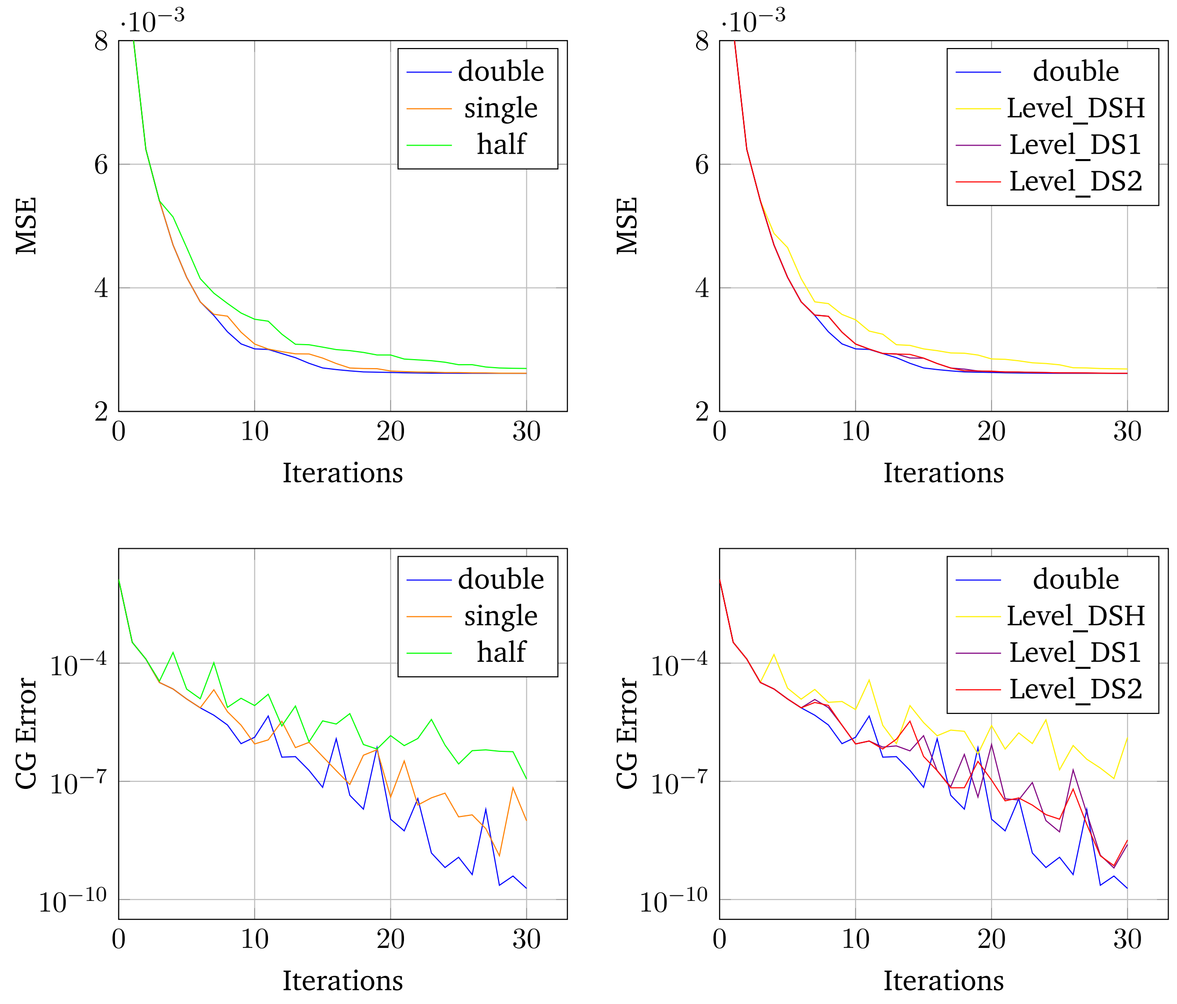


Figure 5.10: MSE (top) and error norm of the CG method (bottom) of the regression are shown over the number of executed CG iteration. Tests were conducted on a regular, five-dimensional sparse grid, containing 1471 grid nodes, without boundary nodes or adaptive refinement. The real-world DR5 set containing 300000 training and 60000 testing point was used. A regularization parameter of $\lambda = 5 \cdot 10^{-4}$ was chosen. Different floating-point precision strategies were tested. The results are split into homogeneous (left) and level-dependent strategies (right).

Chapter 6

Discussion and Outlook

At the beginning of this work, the basics of the sparse grid approach and notations used throughout this work were presented. It was explained how complex functions can be approximated with a set of simple d-linear basis functions on a sparse grid. Considering the hierarchical structure of sparse grids, some established algorithms for function approximation and data mining tasks were discussed. One essential component is the hierarchical evaluation, which has an improved computational complexity compared to the naive approach. It was the main focus of the floating-point considerations. Methods to solve the classification and regression problem with sparse grids were investigated. The conjugate gradient method is considered to find an approximated solution of the linear system produced by these methods. Multiple hierarchical evaluations and transposed evaluations are performed in each iteration, which dominate the runtime of the CG algorithm. In a practical implementation these algorithms perform many expensive floating-point operation. It was shown that the hierarchical structure of sparse grids often leads to significantly smaller basis coefficients on higher levels, which consequently have a smaller contribution to the evaluation result. Therefore, a reduction of the floating-point precision in conjunction with these coefficients was explored. Besides homogeneous double (FP64), single (FP32) and half (FP16) precision, different mixed-precision strategies were developed, in order to reduce the computation time and storage size while ensuring appropriate accuracy.

The presented strategies were tested in the discussed algorithms. A two-dimensional checkerboard set was consulted for the classification problem. The four-dimensional Friedman set and the five-dimensional DR5 set served as input for the regression problem. The CG convergence over a growing number of grid node, and the accuracy progression for a reasonable sparse grid size were investigated. Multiple conclusions can be drawn from the test results:

- The homogeneous or partial use of half is not recommended. Compared to double precision, the convergence of the CG method takes significantly longer. On a complex data set, the regression needs to perform twice as much iterations to reach an acceptable error bound. And still, the MSE is never able to reach the MSE of strategies with single and/or double precision.
- Both mixtures of single and double precision perform very similar to uniform single precision. The anticipated benefits of mixed-precision could not be observed.

- The tests with single precision promise potential. The CG method usually converges after only 3 extra iterations. Possibly, the faster runtime of single-precision operations could more than compensate this computational overhead. In the tests, full or partial single precision can very well compete with homogeneous double precision regarding accuracy.
- All precision choices achieve the exact same MSE and CG error in the first few iterations of the CG algorithm. On the most complex data set, single can keep up with double precision over the first 5 iterations. On simpler functions it performs equally until iteration 10. Therefore, single precision could be used in the first few iteration, before switching to double precision in all operations. With this approach, the runtime can be improved notably considering that the full convergence takes between 20 and 25 iterations on all data sets.

It has to be noted, that boundary nodes or adaptive refinement of the sparse grid was not considered in this work, which both increase the effectiveness of sparse grid. More representative results can be produced by incorporating these techniques.

One problem that was not discussed in this work is the realization of a practical, storage-efficient data structure. When it is known beforehand that certain values are only computed in lower precision, these variable can be stored accordingly in order to save storage space. However, a traditional array does not allow a mixed-precision. A data structure based on pointers is also not a good idea, since individual entries should be accessible via index, and pointers produce an additional storage overhead.

Though the benefits of mixed-precision strategies were quite limited in the tests, the concept of mixed-precision can be extended to other approaches. We have examined different level-dependent strategies. However, surplus values on higher level do not have to be smaller necessarily, since the approximated function may contain individual spikes or oscillations. Another approach hence is to choose precision depending on the actual value of the hierarchical surplus. As we have discussed, a temporary use of lower precision is also possible.

Nevertheless, further investigation will be needed to uncover the full potential of mixed-precision data mining for sparse grids.

Appendix

Bibliography

[AAA$^+$07] J. K. Adelman-McCarthy, M. A. Agüeros, S. S. Allam, et al. The Fifth Data Release of the Sloan Digital Sky Survey. *Astrophysical Journal Supplement Series*, 172:634–644, 2007. (Cited on pages 37 and 47)

[Bel61] R. E. Bellman. *Adaptive control processes: a guided tour*. Princeton university press, 1961. (Cited on page 7)

[BG04] H.-J. Bungartz, M. Griebel. Sparse grids. *Acta numerica*, 13:147–269, 2004. (Cited on pages 7 and 9)

[Bru16] M. Brunn. Data-mining on adaptively refined sparse grids with higher order basis functions on GPUs, 2016. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-2479&engl=1. (Cited on pages 9, 11, 16, 17, 19, 20, 21, 22, 23, 37, 38 and 43)

[Bun98] H.-J. Bungartz. *Finite elements of higher order on sparse grids*. Ph.D. thesis, Technische Universität München, 1998. (Cited on pages 9, 12 and 16)

[Fri91] J. H. Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pp. 1–67, 1991. (Cited on pages 37 and 43)

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991. (Cited on pages 26 and 27)

[Pfl10] D. M. Pflüger. *Spatially adaptive sparse grids for high-dimensional problems*. Ph.D. thesis, Technische Universität München, 2010. (Cited on pages 7, 8, 9, 15, 19, 20, 21, 37, 38 and 47)

[ZCA$^+$08] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, et al. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pp. 1–70, 2008. (Cited on page 27)

[Zen90] C. Zenger. *Sparse grids*. Technische Universität, 1990. (Cited on pages 7 and 25)

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature