DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Adaptive Romberg-Quadrature for the Sparse Grid Combination Technique

Pascal Resch

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Adaptive Romberg-Quadrature for the Sparse Grid Combination Technique

Adaptive Romberg-Quadratur für die Sparse Grid Kombinationstechnik

Author:	Pascal Resch
Supervisor:	Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisor:	M.Sc. Michael Obersteiner
Submission Date:	01.09.2020

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

Erklärung zur Hausarbeit gemäß § 29 (Abs. 6) LPO I: Hiermit erkläre ich, dass die vorliegende Arbeit von mir selbstständig verfasst wurde, und dass keine anderen als die angegebenen Hilfsmittel benutzt wurden. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen sind, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Diese Erklärung erstreckt sich auch auf etwa in der Arbeit enthaltene Grafiken, Zeichnungen, Kartenskizzen und bildliche Darstellungen.

München, 01.09.2020

Pascal Resch

Acknowledgements

Above all I would especially like to thank my advisor Michael Obersteiner for his great inspirations, suggestions and constant support. Thank you for the time you invested into the regular meetings. I enjoyed our conversations very much and they wouldn't have been the same without you!
Next, I would like to thank Prof. Dr. Bungartz for providing me the opportunity to write this bachelor's thesis at the *Chair of Scientific Computing in Computer Science*. I had a great time working on this bachelor's thesis and learned a lot during the process.
Finally, I would like to thank my parents for their constant support and making my studies possible.

Pascal Resch, 01.09.2020
München

Abstract

Many problems nowadays require approximations of high-dimensional integrals, because their analytical solutions cannot be provided. Among others, possible fields of application are Uncertainty Quantification and Machine Learning. Unfortunately high-dimensional problems suffer from the *Curse of Dimensionality*: Due to the exponential increase of the number of grid points the computation time grows rapidly. This makes calculations infeasible when the number of dimension grows. Possible solutions are *Sparse Grids* and the *Sparse Grid Combination Technique*, which are non-adaptive. However, many real-world applications have highly varying characteristics and thus require strategies that adapt to the given problem. A well-known approach is a dimension adaptive variant of the *Combination Technique* [GG03]. Another approach has been presented in [OB20]: a spatially adaptive variant with dimension-wise refinement. So far there has been little research towards high order methods for the above-mentioned approach. The goal of this thesis is to combine adaptive order methods and spatial adaptivity (with dimension-wise refinement). Therefore we investigate the well-known *Romberg-Quadrature* and generalize it for the application on adaptive grids. Thereafter, various variants of these theoretical results are incorporated into the sparseSpACE-framework and compared to adaptive as well as non-adaptive implementations of quadrature rules, such as Gauß-Legendre. The numerical results show that our adaptive extrapolation can significantly reduce the total number of distinct function evaluations to achieve a certain approximation tolerance threshold. This leads to shorter runtimes and a lower total number of refinements.

Zusammenfassung

Viele Problemstellungen benötigen heutzutage hochdimensionale Berechnungen. Mögliche Einsatzgebiete hierfür sind u.a. Maschinelles Lernen oder Uncertainty Quantification. Hierbei kann oft die analytische Lösung nicht bestimmt werden, weswegen eine Approximation von hochdimensionalen Integralen notwendig wird. Dabei tritt allerdings der *Fluch der Dimensionalität* auf. Mit steigender Dimensions-Anzahl erhöht sich die Anzahl der Gitterpunkte exponentiell, wodurch Berechnungen sehr lange dauern können. Um diesem Effekt entgegenzuwirken wurden *Sparse Grids* (Dünne Gitter) und die *Sparse Grid Kombinationstechnik* untersucht. Allerdings weisen Alltagsprobleme oftmals sehr unterschiedliche Charakteristiken auf, welche stark variieren können. Daher benötigt man adaptive Verfahren, die sich an die Charakteristiken des Problems anpassen. Ein bekannter Algorithmus hierfür ist eine dimensionsadaptive Version der *Kombinationstechnik* [GG03]. Außerdem wurde in [OB20] eine räumlich-adaptive Variante mit Verfeinerung von einzelnen Dimensionen vorgestellt. Bisher gab es wenig Forschung, welche High-Order Methoden mit dem vorherigen Verfahren zur räumlichen Verfeinerung verknüpft. Das Ziel dieser Arbeit ist ein Verfahren, welches Ordnungsadaptivität und räumliche Adaptivität (mit dimensionsweiser Verfeinerung) vereint. Dazu untersuchen wir die bekannte *Romberg-Quadratur* und verallgemeinern diese anschließend für adaptive Gitter. Wir implementieren verschiedene Varianten dieser Verallgemeinerung und integrieren diese in das sparseSpACE-Framework. Außerdem führen wir numerische Untersuchungen durch und vergleichen unsere Ergebnisse mit sowohl adaptiven als auch nicht-adaptiven Verfahren, wie z.B. einem Gauß-Legendre Gitter. Unsere Ergebnisse zeigen, dass durch adaptive Extrapolation die Gesamtanzahl eindeutiger Funktionsevaluationen signifikant verringert werden kann, wobei weiterhin eine vergleichbare Approximation des Integralwerts gewährleistet wird. Diese Verbesserung hat, unter anderem, kürzere Laufzeiten und weniger Verfeinerungsschritte zur Folge.

Contents

1 Introduction

"Squaring the circle" [1]. This statement is used when someone is trying to do the impossible. Already the ancient greek geometers were interested in the problem called *Squaring the circle*. It is also known as *Quadrature of the circle* [DR07, p.1]. Given a circle with a radius $r = 1$ one aims to construct a square which has the same area as the circle within a finite amount of steps. It was proven that this problem is unsolvable using only straightedge and compass. One geometer who investigated this issue with deeper interest was Archimedes of Syracuse. He approximated the circle by inscribing and circumscribing regular n-sided polygons for increasing $6 \leq n_i \leq 96, \quad n_{i+1} = 2n_i$. Using this technique Archimedes proved that π is bounded by [Eng80, p.7]: $\frac{22}{7} < \pi < \frac{223}{71}$.
Another problem that emerged was the *Quadrature of the Parabola*. Archimedes asked himself the question: "What is the size of the area enclosed by an inverted unit parabola and the x-axis?" [Dei].

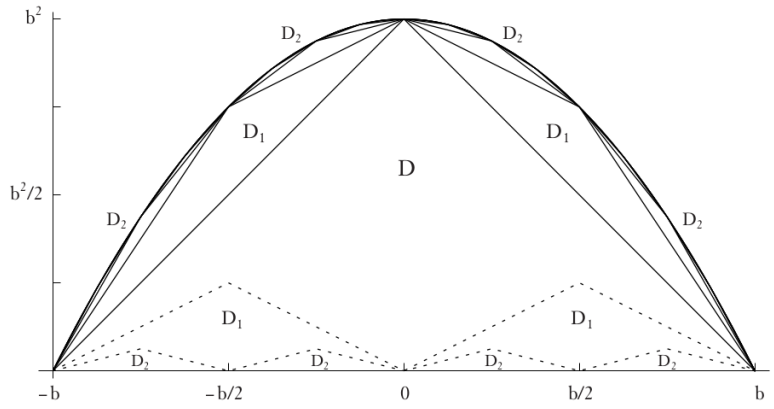


Figure 1.1: Method of exhaustion.
Taken from [Dei, p.37].

He solved this problem using the method of exhaustion by inscribing the parabola with triangles of decreasing size: Let $D, D_1, D_2, \ldots, D_n, \ldots$ be the area of the corresponding triangle in the figure 1.1 above. By applying Cavalieri's principle and the geometric series it follows with $D = \frac{1}{2} \cdot 2b \cdot b^2 = b^3$:

$$\begin{aligned} A &= D + 2 \cdot D_1 + 4 \cdot D_2 + \cdots + 2^n \cdot D_n + \ldots \\ &= D + 2 \cdot \frac{D}{8} + 4 \cdot \frac{D}{8^2} + \cdots + 2^n \cdot \frac{D}{8^n} + \ldots \\ &= D \cdot \sum_{n \geq 0} \left(\frac{1}{4}\right)^n = D \cdot \frac{1}{1 - \frac{1}{4}} = \frac{4}{3} b^3 \end{aligned}$$

On the other hand, using the fundamental theorem of calculus we obtain instantly

$$\begin{aligned} A = \int_{-b}^{b} f(x)\, dx = \int_{-b}^{b} b^2 - x^2\, dx = 2 \cdot \int_{0}^{b} b^2 - x^2\, dx = 2 \cdot \left[b^2 \cdot x - \frac{1}{3} x^3 \right]_0^b \\ = 2 \cdot b^3 - 2 \cdot \frac{1}{3} b^3 = \frac{4}{3} b^3, \end{aligned}$$

since the function is symmetric to the y-axis. Generalizing the idea of Archimedes, one possible method to approximate integrals is to bound the calculation using only n triangles.

[1] `https://www.dictionary.com/browse/square--the--circle`

The generic problem of quadrature

Now we want to introduce the general core concepts of numerical quadrature. In the following let $f : \Omega \subset \mathbb{R} \to \mathbb{R}$ be a scalar function for a close interval Ω. Numerical quadrature normally uses a linear combination of function values $f(x_i)$ and corresponding weights w_i as an approximation of the integral with $n \geq 1$ support points:

$$\int_a^b f(x)\, dx \approx w_1 \cdot f(x_1) + \cdots + w_n \cdot f(x_n) = \sum_{i=1}^{n} w_i \cdot f(x_i) \tag{1.0.1}$$

for $\min(\Omega) \leq a \leq b \leq \max(\Omega)$, $w_i \in \mathbb{R}$, and $x_i \in \Omega$ for all $1 \leq i \leq n$. One regularly strives to use non-negative weights, since the problem of quadrature can become ill-conditioned otherwise. Following this approach we obtain primitive quadrature rules like the *Midpoint rule* and *Trapezoidal rule* by introducing a step width $h = \frac{b-a}{n} > 0$ for $n \geq 1$.

Visually speaking, one can consider h as the width of each sub-interval. The Midpoint rule inscribes n rectangular stripes under the function curve using the function value in the middle of each sub-interval to compute the corresponding height:

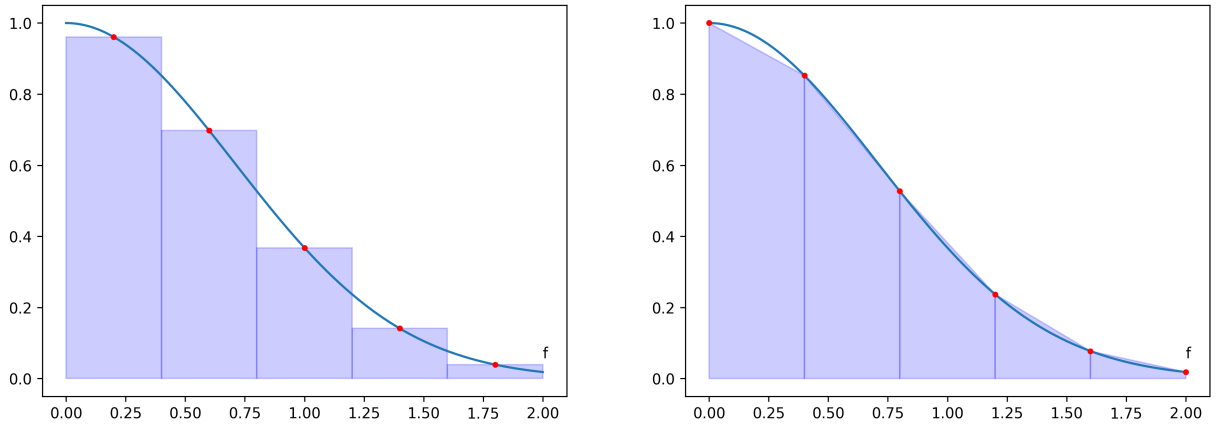
$$\begin{aligned} M_f(h) &= h \cdot f(a + (0 + \frac{1}{2}) \cdot h) + h \cdot f(a + (1 + \frac{1}{2}) \cdot h) + \cdots + h \cdot f(a + ((n-1) + \frac{1}{2}) \cdot h) \\ &= h \cdot \sum_{i=0}^{n-1} f(a + (i + \frac{1}{2}) \cdot h) \end{aligned} \tag{1.0.2}$$

By inscribing trapezoids instead of rectangles, we obtain the Trapezoidal rule for $h = \frac{b-a}{n}$:

$$\begin{aligned} T_f(h) &= \frac{h}{2} \cdot f(a + 0 \cdot h) + h \cdot f(a + 1 \cdot h) + \cdots + h \cdot f(a + (n-1) \cdot h) + \frac{h}{2} \cdot f(b) \\ &= h \cdot \left[\frac{1}{2} f(a) + \left(\sum_{i=1}^{n-1} f(a + i \cdot h) \right) + \frac{1}{2} f(b) \right] \end{aligned} \tag{1.0.3}$$

Futhermore, the weights are defined as $w_i = \frac{h}{2}$ for $i \in \{0, n\}$ and $w_i = h$ for $1 \leq i \leq n-1$.

The following figure compares these rules for $f : \mathbb{R} \to \mathbb{R}, f(x) = \exp(-x^2)$, and $a = 0, b = 2, n = 5$.



(a) Midpoint rule for $n = 5$

(b) Trapezoidal rule for $n = 5$

Figure 1.2: Comparison of Midpoint and Trapezoidal rule

These two methods belong to a class of Quadrature rules called Newton-Cotes formulas [DR07]. Among others this class also contains *Simpson's rule*. The results are approximated by dividing the integration area into several sub-intervals. The Trapezoidal rule inscribes trapezoids whereas Simpson's rule inscribes parabolas. The sum of these parts approximates the area below the curve in the given interval. Theoretically it would be possible to increase the degree of accuracy of the underlying interpolation function arbitrarily. Unfortunately, for $n \geq 9$ negative weights occur [Eng80, p.268] which increases the numerical condition and therefore detoriates the result.
Nowadays, many problems require an approximation of integrals, because analytical solutions cannot be provided. This happens, for example, if there are only discrete values of scientific experiments available. Another example is the function $f : [a, b] \to \mathbb{R}, f(x) = e^{-x^2}$ where the integral $\int f(x)\, dx$ "cannot be expressed in finite terms by combinations of algebraic, logarithmic, or exponential operations" [DR07, p.2].

Sparse Grids
High-dimensional applications occur for example in physics or finance. Often one has to compute integrals that are approximated by increasing the resolution of points in each dimension. Using Archimedes quadrature the method's accuracy depends on how many triangles are used. The more triangles are used, the more accurate the result will be.
The drawback of increasing the grid's resolution for higher dimensions ($d > 4$, see [Pfl10]) is that the computation time soars rapidly. This is due to the fact that the function must be evaluated at an exponentially growing number of points for increasing d. More specifically, let us consider a grid with n grid points in each of the d dimensions. Then the *Full Grid* contains $\mathcal{O}(n^d)$ grid points. The computing complexity in such high dimensions is also called *Curse of Dimensionality*.
One possible remedy are Sparse Grids. They require significantly less evaluation points, while maintaining a similar asymptotic error decay as full grids (up to a logarithmic factor) [Pfl10]. This is achieved by selecting only those points which contribute most to the result. Points whose contributions are too small are neglected. With this technique the number of points is reduced to $\mathcal{O}(n \cdot log(n)^{d-1})$ for n grid points in each dimension [BG04]. Further important aspects of sparse grids will be presented in chapter 3.
Unfortunately, when using Sparse Grids one has to transform existing algorithms to work with the new grid type. To mitigate these efforts Zenger et al. developed the *Combination Technique* [GSZ92]. Their idea is to represent a Sparse Grid by a linear combination of full grids (each is called a *Component Grid*) with different resolutions. This technique enables us to take profit of already existing algorithms that operate separately on component grids. These individual results are combined according to the combination scheme. It is an ideal example for using parallel computations for the component grids.

Adaptivity strategies
Not all functions are as smooth as the function mentioned in the example above. They might rather have different properties in some areas or dimensions. Therefore one should additionally leverage adaptive algorithms.
There are two strategies of adaptivity: order adaptivity and refinement adaptivity.
The latter aims to exploit local smoothness of functions by partitioning the integration domain into smaller regions. Here the main idea is to use more grid points in intervals where the integrand has high oscillation or many different characteristics. In contrast one wants to use fewer grid points in those areas where the function is moderate. Several authors have implemented different types of adaptivity: dimensional adaptivity [GG03], spatial adaptivity [Pfl10] [OB19] and spatial adaptivity with dimension-wise refinement [OB20].

The second strategy is adaptivity of order [Bon94]. Here, the idea is again to exploit the smoothness of a function. But this time by increasing the order in promising sub-areas. Bonk proposed a method that combines both adapitivity strategies [Bon95].
The main contribution of the thesis will be based on Romberg-Quadrature [Rom55]. This method leverages Richardson extrapolation and combines trapezoidal rules of decreasing step size in each iteration to retain an approximation of higher order. A detailed introduction to this method is given in chapter 2. Afterwards another generalization of Romberg's method (than the one of [Bon95]) is proposed.

Thesis overview
The thesis will have the following structure: In chapter 2, the original *Romberg-Quadrature* will be introduced. The new results of this thesis are based on the original idea of Romberg. After the intuition of Romberg's method is explained, we will formally construct and derive this quadrature method. Subsequently a new generalization of Romberg's method is derived. It will enable extrapolation on adaptive grids and is the main contribution of this thesis. Finally, we propose some optimizations and variants of the generalized Romberg method. Chapter 3 presents the intuition and construction of the *Sparse Grid* method. Building upon those grids, we are going to explain the *Combination technique* and some adaptive variants of it. After the theoretical foundations are laid we will summarize important aspects of the SparseSpace framework and the integration of the new method in chapter 4. In chapter 5 the methods are tested and compared with various other adaptive as well as non-adaptive quadrature techniques. Finally, we draw a conclusion of our work and propose further ideas for improvement in chapter 6 which concludes this thesis.

2 Romberg-Quadrature

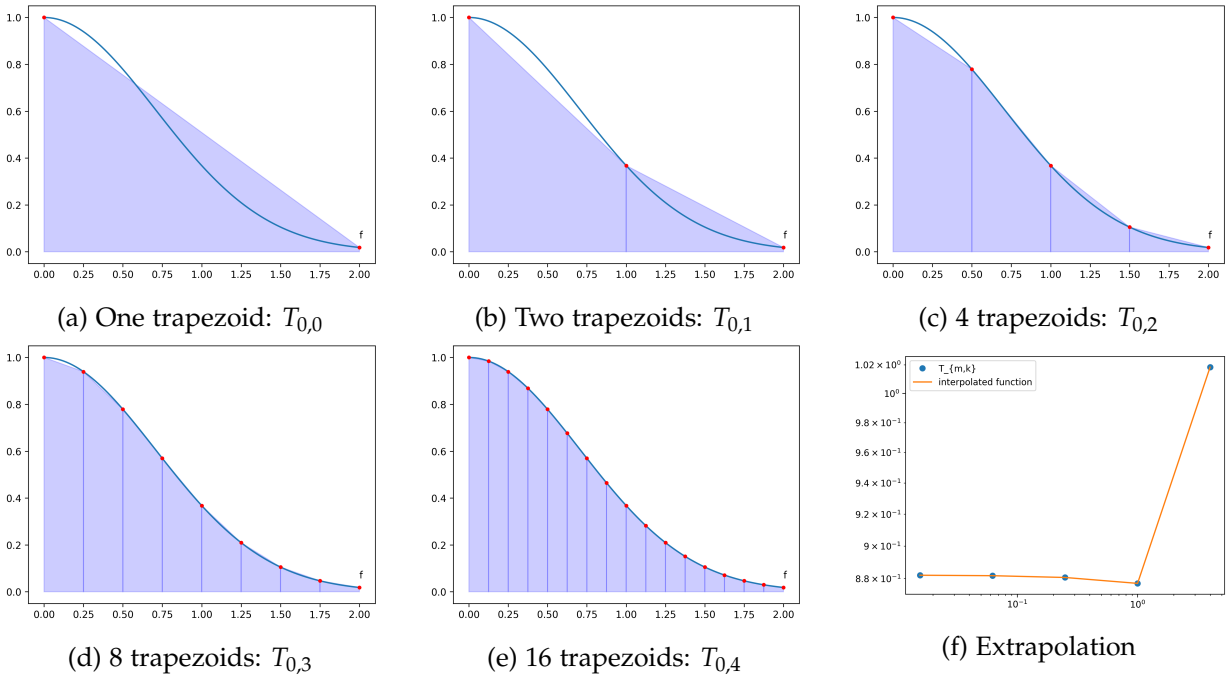
Romberg's method initially computes trapezoidal rules with decreasing step size. The sequence of results are then combined by exploiting the error expansions of the trapezoidal rules - following the idea of extrapolation. Thus a higher order of accuracy is obtained.
In this chapter we will first explain the intuition of the method and summarize some important results in this area. Afterwards, the theoretical background of Romberg's method is explained. Finally, Romberg's method will be generalized, enabling higher order quadrature on adaptive grids. In this chapter we will focus on the theoretical foundation and intuition. The associated implementation will be presented in chapter 4.

2.1 Original method

Let $f \in \mathcal{C}^{2m+2}([a,b])$ for $m \geq 0$. We define the step size $h_k = \frac{b-a}{n_k}$ with $n_k = 2^k$ for all $k \geq 0$. In this section we are following [Eng80, p.369]. In addition, let us introduce a new notation:

$$T_{0,k} = T_f(h_k) = h_k \cdot \left[\frac{1}{2} f(a) + \left(\sum_{i=1}^{n_k - 1} f(a + i \cdot h_k) \right) + \frac{1}{2} f(b) \right] \tag{2.1.1}$$

Romberg's method proficiently combines these trapezoidal sums $T_{0,k}$. Before we explain the formal extrapolation process we visualize the idea of the method. Let $f : \mathbb{R} \to \mathbb{R}, f(x) = \exp(-x^2)$, and $a = 0, b = 2$. We approximate the integral $\int_0^2 f(x)\, dx$ by using Romberg's method:



(a) One trapezoid: $T_{0,0}$ (b) Two trapezoids: $T_{0,1}$ (c) 4 trapezoids: $T_{0,2}$

(d) 8 trapezoids: $T_{0,3}$ (e) 16 trapezoids: $T_{0,4}$ (f) Extrapolation

Figure 2.1: Romberg's method visualized

Figure 2.1f visualizes the extrapolation process using formula 2.1.2. Informally speaking, we interpolate through the points $(h_k^2, T_{0,k})$ for $0 \leq k \leq 4$. The extrapolated result is the intersection of the interpolated curve and $x = 0$.

In each iteration $k \geq 1$, we initially compute $T_{0,k}$. Afterwards the trapezoidal rules are combined, by applying the following formula (according to the Aitken-Neville-Scheme):

$$T_{m,k} = \frac{4^m \cdot T_{m-1,k+1} - T_{m-1,k}}{4^m - 1} \tag{2.1.2}$$

where $1 \leq m \leq k$ specifies the number of extrapolations. This way we are able to combine the entries $T_{m,k}$ and can fill a table row by row. Hence, the order of accuracy is increased in each iteration of k:

$$\begin{array}{llll} T_{0,0} & & & \\ T_{0,1} & T_{1,0} & & \\ T_{0,2} & T_{1,1} & T_{2,0} & \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

Table 2.1: Romberg's table or T-table

We will refer to this table as *Romberg's table* in the following sections. Assuming we stop at row $r \geq 0$, the result of the extrapolation process is given by $T_{r,0}$. Using *Lagrange interpolation* we are able to rewrite this extrapolation process [Bon95, p.21] to:

$$T_m(0) = T_{m,0} = \sum_{j=0}^{m} c_{m,j} \cdot T(h_j), \quad \text{with} \quad c_{m,j} = \prod_{\substack{i=0, \\ i \neq j}}^{m} \frac{h_i^2}{h_i^2 - h_j^2} \tag{2.1.3}$$

As one notices, the function is evaluated multiple times at the same support points. In the example above we would evaluate the function at points a, b for each $0 \leq k \leq 4$. By exploiting the property of $T_{0,k}$ we get a more efficient form of the trapezoidal rule for $k \geq 1$ when using the results from previous steps:

$$\begin{aligned} T_{0,k} &= h_k \left[\frac{1}{2} f(a) + \sum_{l=1}^{n_k - 1} f(a + l \cdot h_k) + \frac{1}{2} f(b) \right] \\ &= h_k \left[\frac{1}{2} f(a) + \sum_{\substack{l=1, \\ l \text{ odd}}}^{n_k/2} f(a + l \cdot h_k) + \sum_{\substack{l=1, \\ l \text{ even}}}^{(n_k/2)-1} f(a + l \cdot h_k) + \frac{1}{2} f(b) \right] \\ &= \frac{T_{0,k-1}}{2} + h_k \sum_{\substack{l=1, \\ l \text{ odd}}}^{n_k/2} f(a + l \cdot h_k) \end{aligned} \tag{2.1.4}$$

Properties of Romberg's method:
Engels [Eng80, p.381] summarized and proved some important properties about the Romberg-Quadrature method:

> **Theorem 2.1.1**
> Romberg's quadrature contains only positive weights.

Negative weights would increase the numerical condition and therefore detoriate the result.

Theorem 2.1.2

Let $f \in C^0([0,1])$. Then the Romberg-Quadrature scheme converges for every f.

Bauer [Eng80] proved a theorem about the error representation of the method:

Theorem 2.1.3: Error representation of Romberg-Quadrature

The Romberg-quadrature formulas $T_{m,k}$ have the error representation

$$\int_0^1 W(x) \cdot f(x)\, dx - T_{m,k} = \frac{(-1)^{i+1} \cdot B_{2m+2}}{(2m+2)!} \cdot 4^{-(m+1)k} \cdot 2^{-m(m+1)} \cdot f^{(2m+2)}(t)$$

for $t \in (0,1)$, a weight function $W(x)$, and the Bernoulli numbers B_i.

Romberg's method operates on the sequence $n_k = 2^k$. This has the disadvantage that n_k is decreasing rapid. This is reflected by rapidly increasing computation time because the number of sub-intervals for the trapezoidal rule increases exponentially. Bulirsch [Bul64] showed that another sequence may be used:

$$n_0 = 1, \quad n_k = \left\{ \begin{array}{ll} \frac{1}{3} \cdot 2^{-\frac{k}{2}+1}, & \text{for } k \text{ even} \\ 2^{-\frac{k+1}{2}} & \text{for } k \text{ odd} \end{array} \right\} \text{ for } k > 0 \tag{2.1.5}$$

When using other sequences than the one proposed by Romberg, it is necessary to use another formula instead of 2.1.2. An example is retained by application of Aitken-Neville as explained in [Eng80, p.372]:

$$T_{m,k} = \frac{T_{m-1,k} \cdot h_{k+m} - T_{m-1,k+1} \cdot h_k}{h_{k+m} - h_k} \tag{2.1.6}$$

By comparing $delta := T_{i,0} - T_{i-1,0}$ for $i > 0$ after each step with a given tolerance $tol > 0$ one can define a termination criterion for this method: Stop the iteration if $delta < tol$. The Romberg method may then be summarized as follows:

Algorithm 1: Romberg-Quadrature in the 1-dimensional case

Input: Function $f : \mathbb{R} \to \mathbb{R}$, error tolerance tol, maximum iteration bound max_i, lower and upper bounds $a, b \in \mathbb{R}$.

Output: Approximation of the integral $\int_a^b f(x)\, dx$

$h_k \leftarrow b - a$; $\quad n_k \leftarrow 1$; $\quad delta \leftarrow \infty$; $\quad i \leftarrow 0$;

while $(|delta| > tol)$ ***and*** $(i < max\_i)$ **do**

  T[i] $\leftarrow h_k \left[\frac{1}{2} f(a) + \sum_{l=1}^{n_k - 1} f(a + l \cdot h_k)\right]$

  for $k \leftarrow 1$ **to** i **do**

    T[i].append($\frac{4^m \cdot T_{m-1,k+1} - T_{m-1,k}}{4^m - 1}$)

  end

  if $i > 0$ **then**

    delta $\leftarrow T_{i,0} - T_{i-1,0}$

  end

  $h_k \leftarrow h_k / 2$; $\quad n_k = n_k * 2$; $\quad i \leftarrow i + 1$;

end

2.2 Derivation of the original method

In this section, we will explain why the Romberg method works the way presented in the section above. The argumentation is as follows: First we are going to analyze the error of the trapezoidal rule based on a Taylor expansion. Thereafter we are generalizing this result to the composite trapezoidal rule. Thus, we are able to specify the error in terms of the step width. Finally we compose the steps mentioned above by combining the error representation of trapezoidal rules with different step widths. This procedure gives us the main formula of Romberg's method and its corresponding error. First of all, let $\Omega = [a, b] \subset \mathbb{R}$, $f : \Omega \to \mathbb{R}$ a function with $f \in \mathcal{C}^\infty(\Omega)$. Furthermore, let $m \in \Omega$. Then there exists a Taylor expansion of f in the development point m:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!}(x-m)^k \tag{2.2.1}$$

2.2.1 Trapezoidal rule error

We develop a Taylor series around the midpoint $m = \frac{a+b}{2}$ of the integration domain D, following [Hea02] [Obe19]. Starting with the default formula for the trapezoidal rule $T(f)$ with step width $H = b - a$ we insert the Taylor expansion:

$$\begin{aligned} T(f) &= H\frac{f(a)+f(b)}{2} = \frac{H}{2}(f(a)+f(b)) \\ &= \frac{H}{2}\left[\sum_{k=0}^{\infty}\frac{f^{(k)}(m)}{k!}(a-m)^k + \sum_{k=0}^{\infty}\frac{f^{(k)}(m)}{k!}(b-m)^k\right] \\ &= \frac{H}{2}\sum_{k=0}^{\infty}\frac{f^{(k)}(m)}{k!}((a-m)^k + (b-m)^k) \end{aligned}$$

Besides, we know

$$\begin{aligned} a - m &= a - \frac{a+b}{2} = \frac{2a}{2} - \frac{a+b}{2} = \frac{a-b}{2}, \\ b - m &= b - \frac{a+b}{2} = \frac{2b}{2} - \frac{a+b}{2} = \frac{b-a}{2} = -\frac{a-b}{2} = -(a-m) \end{aligned}$$

Thus, it holds that $(a-m)^k = (-1)^k(b-m)^k = (-\frac{H}{2})^k = (-1)^k\frac{H^k}{2^k}$ and we can simplify

$$(a-m)^k + (b-m)^k = \left\{ \begin{array}{ll} 0 & \text{, for } k \text{ odd} \\ 2\frac{H^k}{2^k} & \text{, for } k \text{ even} \end{array} \right\} \text{ for } k \geq 0$$

The odd terms eliminate themselves. By inserting in $T(f)$ above it holds:

$$\begin{aligned} T(f) &= \frac{H}{2}\sum_{\substack{k=0 \\ k \text{ even}}}^{\infty}\frac{f^{(k)}(m)}{k!}\cdot 2\cdot\frac{H^k}{2^k} = \sum_{\substack{k=0 \\ k \text{ even}}}^{\infty}\frac{f^{(k)}(m)}{k!}\cdot\frac{H}{2}\cdot 2\cdot\frac{H^k}{2^k} \\ &= \sum_{\substack{k=0 \\ k \text{ even}}}^{\infty}\frac{f^{(k)}(m)\cdot H^{k+1}}{2^k\cdot k!} \end{aligned}$$

By integrating the Taylor expansion of f in m we generate a second equation:

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_a^b \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!}(x-m)^k\,dx = \sum_{k=0}^{\infty} \int_a^b \frac{f^{(k)}(m)}{k!}(x-m)^k\,dx \\ &= \sum_{k=0}^{\infty} \left[\frac{f^{(k)}(m)}{(k+1)\cdot k!}(x-m)^{k+1}\right]_a^b \\ &= \sum_{k=0}^{\infty} \left[\frac{f^{(k)}(m)}{(k+1)!}(b-m)^{k+1} - \frac{f^{(k)}(m)}{(k+1)!}(a-m)^{k+1}\right] \\ &= \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{(k+1)!}\left[(b-m)^{k+1} - (a-m)^{k+1}\right]\end{aligned}$$

This expression can be simplified. With the same idea as presented above we get:

$$(b-m)^{k+1} - (a-m)^{k+1} = \left\{ \begin{array}{ll} 0 & \text{, for } k \text{ odd} \\ 2\frac{H^{k+1}}{2^{k+1}} & \text{, for } k \text{ even} \end{array} \right\} \text{ for } k \geq 0$$

In the end we can rewrite the integral expression to:

$$\int_a^b f(x)\,dx = \sum_{\substack{k=0 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m)}{(k+1)!} \cdot 2 \cdot \frac{H^{k+1}}{2^{k+1}} = \sum_{\substack{k=0 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m)\cdot H^{k+1}}{2^k \cdot (k+1)!}$$

This derivation provided us two equations that are summarized in the following:

$$T(f) = H \cdot f(m) + \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m)\cdot H^{k+1}}{2^k \cdot k!} \tag{I}$$

$$\int_a^b f(x)\,dx = H \cdot f(m) + \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m)\cdot H^{k+1}}{2^k \cdot (k+1)!} \tag{II}$$

Reordering of equation (I) enables us to insert (I) in (II):

$$\begin{aligned}\int_a^b f(x)\,dx &= \underbrace{\left[T(f) - \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m)\cdot H^{k+1}}{2^k \cdot k!}\right]}_{=H\cdot f(m)} + \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m)\cdot H^{k+1}}{2^k \cdot (k+1)!} \\ &= H\frac{f(a)+f(b)}{2} + \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} f^{(k)}(m) \cdot \left[\frac{H^{k+1}}{2^k\cdot(k+1)!} - \frac{H^{k+1}}{2^k\cdot k!}\right] \\ &= H\frac{f(a)+f(b)}{2} + \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} H^{k+1} \cdot \underbrace{f^{(k)}(m) \cdot \left[\frac{1}{2^k\cdot(k+1)!} - \frac{1}{2^k\cdot k!}\right]}_{=:C_k} \\ &= H\frac{f(a)+f(b)}{2} + \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} H^{k+1}\cdot C_k\end{aligned}$$

Through rearranging we obtain the following error expansion of the trapezoidal rule:

$$\begin{aligned}\int_a^b f(x)\,dx - T(f) &= \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} H^{k+1}\cdot C_k \\ &= H^3\cdot C_2 + H^5\cdot C_4 + H^7\cdot C_6 + \ldots\end{aligned} \tag{2.2.2}$$

2.2.2 Composite trapezoidal rule error

The derivation of this error expansion is a little bit more involved. An important result is the well-known Euler-Maclaurin formula, which is summarized in the following theorem [Sto71, p.168]:

Theorem 2.2.1: Euler-Maclaurin formula (Trapezoidal rule)
Let $\Omega = [a,b]$, $f \in C^{2m+2}([a,b])$, and $x_i = a + i \cdot h$ for $0 \leq i \leq n$ and $h = \frac{b-a}{n}$. Then it holds

$$T_f(h) = \int_a^b f(t)dt + \sum_{k=1}^{m} \tau_{2k} \cdot h^{2k} + R_{m+1}(h) \cdot h^{2m+2}$$

with $\tau_{2k} = \frac{B_{2k}}{(2k)!}(f^{(2k-1)}(b) - f^{(2k-1)}(a))$, $1 \leq k \leq m$ and $R_{m+1}(h) = \frac{B_{2m+2}}{(2m+2)!}(b-a)f^{2m+2}(\xi)$, $a < \xi < b$.

Here $T_f(h)$ stands for the composite trapezoidal rule as defined in chapter 1 and B_k denotes the k-th Bernoulli number which is defined recursively as $B_k = -\frac{1}{n+1}\sum_{i=0}^{k-1}\binom{k+1}{i}B_i$ with $B_0 = 1$.
This provides us with an expansion of $T_f(h)$ in powers of h where τ_{2k} is independent of n and h.
The remainder $R_{m+1}(h)$ is bounded [Sto71, p.170]: $\left|R_{m+1}(h)\right| \leq M_{m+1}$ for all $h = \frac{b-a}{n}$, $n = 1, 2, \ldots$ with $M_{m+1} = \left|\frac{B_{2m+2}}{(2m+2)!}(b-a)\right| \cdot \max_{x\in[a,b]}\left|f^{(2m+2)}(x)\right|$.

2.2.3 Extrapolation

Now we are going to combine composite trapezoidal rules of decreasing step widths. For convenience we will assume $\Omega = [a,b] = [0,1]$ in the following. This has, among others, the advantage that $H = b - a = 1$. By rearranging 2.2.1 we obtain

$$T(h) := T_f(h) = \underbrace{\int_0^1 f(x)\,dx}_{=:I(f)} + \tau_2 h^2 + \tau_4 h^4 + \tau_6 h^6 + \ldots \tag{2.2.3}$$

Now we set up two equations, each one with another step width ($h_0, h_1 > 0$):

$$T(h_0) = I(f) + \tau_2 h_0^2 + \tau_4 h_0^4 + \ldots \tag{I}$$
$$T(h_1) = I(f) + \tau_2 h_1^2 + \tau_4 h_1^4 + \ldots. \tag{II}$$

Subtracting $h_0^2 \cdot (II)$ from $h_1^2 \cdot (I)$ yields

$$\begin{aligned} h_1^2 \cdot T(h_0) - h_0^2 \cdot T(h_1) &= (h_1^2 - h_0^2) \cdot I(f) + 0 + h_0^2 h_1^2 (h_0^2 - h_1^2)\tau_4 + \mathcal{O}(h^6) \\ &= (h_1^2 - h_0^2) \cdot I(f) - h_0^2 h_1^2 (h_1^2 - h_0^2)\tau_4 + \mathcal{O}(h^6) \end{aligned}$$

Now we divide by $h_1^2 - h_0^2$ and obtain

$$\frac{h_1^2 \cdot T(h_0) - h_0^2 \cdot T(h_1)}{h_1^2 - h_0^2} = I(f) - h_0^2 h_1^2 \tau_4 + \mathcal{O}(h^4)$$

After isolation of $I(f)$ it holds that

$$\begin{aligned} I(f) &= \frac{h_1^2 \cdot T(h_0) - h_0^2 \cdot T(h_1)}{h_1^2 - h_0^2} - h_0^2 h_1^2 \tau_4 + \mathcal{O}(h^4) \\ &\approx \frac{h_1^2 \cdot T(h_0) - h_0^2 \cdot T(h_1)}{h_1^2 - h_0^2} + \mathcal{O}(h^4) \end{aligned} \tag{2.2.4}$$

by omitting the error in $\mathcal{O}(h_0^2 h_1^2)$. This procedure can be executed with an increasing number of different step widths. The error is in $\mathcal{O}(h_0^2 h_1^2 h_2^2 \cdots)$ [Obe19].
We derive formula 2.1.3 for the special case $m = 1$ using formula 2.2.4:

$$\begin{aligned} I(f) &\approx \frac{h_1^2 \cdot T(h_0) - h_0^2 \cdot T(h_1)}{h_1^2 - h_0^2} = \frac{h_1^2}{h_1^2 - h_0^2} \cdot T(h_0) - \frac{h_0^2}{h_1^2 - h_0^2} \cdot T(h_1) \\ &= \frac{h_1^2}{h_1^2 - h_0^2} \cdot T(h_0) + \frac{h_0^2}{-(h_1^2 - h_0^2)} \cdot T(h_1) \\ &= \underbrace{\frac{h_1^2}{h_1^2 - h_0^2}}_{=c_{1,0}} \cdot T(h_0) + \underbrace{\frac{h_0^2}{h_0^2 - h_1^2}}_{=c_{1,1}} \cdot T(h_1) = \sum_{j=0}^{1} c_{1,j} \cdot T(h_j) \end{aligned}$$

This results can be generalized for $h_0, h_1, h_2, \dots$ which results in formula 2.1.3 [Bon95, pp. p.21].

2.3 Weight-based variant of the original method

In this section we propose a new variant of Romberg's method by following a weight-based approach. Our goal is to obtain a formula similar to equation 1.0.1:

$$\int_a^b f(x)\, dx \approx w_1 \cdot f(x_1) + \cdots + w_n \cdot f(x_n) = \sum_{i=1}^{n} w_i \cdot f(x_i)$$

Hence, we strive to obtain the coefficients (weights) w_i for each $1 \leq i \leq n$. To achieve our goal we transform equation 2.1.3:

$$\begin{aligned} T_m(0) &= \sum_{j=0}^{m} c_{m,j} \cdot T(h_j) = \sum_{j=0}^{m} c_{m,j} \cdot \left(\frac{h_j}{2} \cdot (f(a) + f(b)) + h_j \cdot \sum_{i=1}^{n_j - 1} f(a + i \cdot h_j) \right) \\ &= \underbrace{\sum_{j=0}^{m} \left(c_{m,j} \cdot \frac{h_j}{2} \cdot (f(a) + f(b)) \right)}_{=:A} + \underbrace{\sum_{j=0}^{m} \left(c_{m,j} \cdot h_j \cdot \sum_{i=1}^{n_j - 1} f(a + i \cdot h_j) \right)}_{=:B} \end{aligned}$$

Thus, by looking at part A, we obtain by comparison of coefficients between equation 1.0.1 and

$$\begin{aligned} A &= \sum_{j=0}^{m} \left(c_{m,j} \cdot \frac{h_j}{2} \cdot (f(a) + f(b)) \right) = (f(a) + f(b)) \cdot \sum_{j=0}^{m} \frac{c_{m,j} \cdot h_j}{2} \\ &= \underbrace{\left(\sum_{j=0}^{m} \frac{c_{m,j} \cdot h_j}{2} \right)}_{=w_0} \cdot f(a) + \underbrace{\left(\sum_{j=0}^{m} \frac{c_{m,j} \cdot h_j}{2} \right)}_{=w_n} \cdot f(b) \end{aligned}$$

the weights $w_0 = w_n = \sum_{j=0}^{m} \frac{c_{m,j} \cdot h_j}{2}$ of the boundary points a, b. The weights of the other points require a little more effort. We consider the second sum, denoted by B. Since the default Romberg method operates on the sequence $(h_k)_{k \in \mathbb{N}_0} = (\frac{H}{n_k})_{k \in \mathbb{N}_0} = (\frac{H}{2^0}, \frac{H}{2^1}, \frac{H}{2^2}, \dots)$ for $(n_k)_{k \in \mathbb{N}_0} = (2^k)_{k \in \mathbb{N}_0}$ and $H = b - a$ we consider this specific sequence at first. Thus the boundary weights simplify to

$$w_0 = w_n = \sum_{j=0}^{m} \frac{c_{m,j} \cdot H}{2 \cdot 2^j} = \sum_{j=0}^{m} \frac{c_{m,j} \cdot H}{2^{j+1}} = \sum_{j=0}^{m} c_{m,j} \cdot \frac{h_j}{2}$$

Consequently, let us introduce a new notation: In the following we will abbreviate $x_{l,i} = a + i \cdot h_l$.

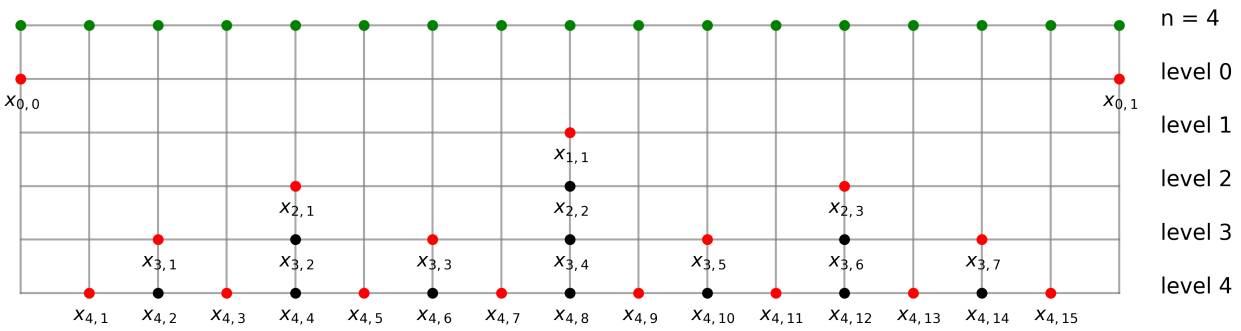


Figure 2.2: Example of a nested grid (level $l = 4$)

By inserting Romberg's sequence h_k we obtain:

$$
\begin{aligned}
B &= \sum_{j=0}^{m} c_{m,j} \cdot h_j \cdot \sum_{i=1}^{n_j-1} f(a + i \cdot h_j) = \sum_{j=0}^{m} c_{m,j} \cdot \frac{H}{2^j} \cdot \sum_{i=1}^{n_j-1} f\left(a + i \cdot \frac{H}{2^j}\right) \\
&= \sum_{j=0}^{m} c_{m,j} \cdot \frac{H}{2^j} \cdot \left(f\left(a + \frac{H}{2^j}\right) + \cdots + f\left(a + \frac{H \cdot (2^j - 1)}{2^j}\right)\right) \\
&= \quad c_{m,0} \cdot \frac{H}{2^0} \cdot \left(0\right) \\
&\quad + c_{m,1} \cdot \frac{H}{2^1} \cdot \left(f(x_{1,1})\right) \\
&\quad + c_{m,2} \cdot \frac{H}{2^2} \cdot \left(f(x_{2,1}) + f(x_{2,2}) + f(x_{2,3})\right) \\
&\quad + c_{m,3} \cdot \frac{H}{2^3} \cdot \left(f(x_{3,1}) + f(x_{3,2}) + f(x_{3,3}) + f(x_{3,4}) + f(x_{3,5}) + f(x_{3,6}) + f(x_{3,7})\right) \\
&\quad + c_{m,4} \cdot \frac{H}{2^4} \cdot \Big(f(x_{4,1}) + f(x_{4,2}) + f(x_{4,3}) + f(x_{4,4}) + f(x_{4,5}) + f(x_{4,6}) + f(x_{4,7}) + f(x_{4,8}) \\
&\qquad\qquad + f(x_{4,9}) + f(x_{4,10}) + f(x_{4,11}) + f(x_{4,12}) + f(x_{4,13}) + f(x_{4,14}) + f(x_{4,15})\Big) \\
&\quad + \ldots \\
&\quad + c_{m,m} \cdot \frac{H}{2^m} \cdot \left(f\left(a + \frac{H}{2^m}\right) + \cdots + f\left(a + \frac{H \cdot (2^m - 1)}{2^m}\right)\right)
\end{aligned}
$$

The red grids points ($x_{l,i}$) in the formula above correspond to the red points in figure 2.2. These are the new contributions on their level. All other points have already occurred in previous levels. If one groups same support values $f(x_{l,i})$ it is now possible to deduce a formula for the non-boundary points: Let $l_i \geq 0$ denote the level of point $x_{l,i} = a + i \cdot h_l$ for $1 \leq i \leq 2^l - 1$.
The weights of Romberg's method are determined by:

$$
w_i = \left\{ \begin{array}{ll} \sum_{j=0}^{m} \frac{c_{m,j} \cdot H}{2^{j+1}}, & \text{for } i = 0, 2^m \\ \sum_{j=l_i}^{m} \frac{c_{m,j} \cdot H}{2^j}, & \text{else} \end{array} \right\} = \left\{ \begin{array}{ll} \sum_{j=0}^{m} c_{m,j} \cdot \frac{h_j}{2}, & \text{for } i = 0, 2^m \\ \sum_{j=l_i}^{m} c_{m,j} \cdot h_j, & \text{else} \end{array} \right. \tag{2.3.1}
$$

We can also interpret this weight formula visually: For each inner point we construct a sum of extrapolation coefficients factored with the step width on each level $l_i \leq j \leq m$.

2.4 Adaptive sliced method

This section aims to develop a generalized Romberg's method for adaptive grids. The default Romberg method works flawlessly on equidistant grids to whom we will refer in the following as *full grids*. However, this method is in general not suitable for *adaptive grids*.

(a) Full grid of level 2

(b) Adaptive grid with max. level 3

Figure 2.3: Comparison of full grids and adaptive grids (Example).

The error cancellation process of Romberg's method does not work that well on adaptive grids. Due to the non-equidistant nature of adaptive grids not all error terms cancel each other out. We will develope a technique that tries to mitigate this effect.

2.4.1 Sliced trapezoidal rule

In the following we want to find a formula for a sliced trapezoid. The main idea is to linearly interpolate between the support points of a trapezoid and then integrate a slice of this area. Let $f : D \subset \mathbb{R} \to \mathbb{R}$ a continuous function. Moreover, let $\Omega = [a, b] \subset D$ be the integration domain. Firstly, we interpolate f in through the two points $(a, f(a))$, $(b, f(b))$:

$$p : \mathbb{R} \to \mathbb{R}, \quad p(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a) =: f(a) + \Delta f_a^b \cdot (x - a)$$

Secondly, let $S_i = [x_i, x_{i+1}] \subset \Omega$ be a slice of the integration domain with step width $h_i = x_{i+1} - x_i$.

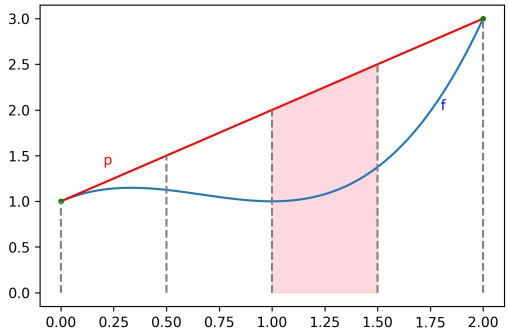


Figure 2.4: Slice $S_2 = [1, 1.5]$ with the support element $(a, b) = (0, 2)$ of level 0.

To obtain the area of the sliced trapezoid on S_i we integrate the interpolated function p:

$$\begin{aligned}
\int_{x_i}^{x_{i+1}} p(x)\, dx &= \int_{x_i}^{x_{i+1}} \left(f(a) + \Delta f_a^b \cdot (x - a) \right) dx = \left[f(a) \cdot x + \Delta f_a^b \cdot \left(\frac{1}{2} \cdot x^2 - a \cdot x \right) \right]_{x_i}^{x_{i+1}} \\
&= (x_{i+1} - x_i) \cdot f(a) + ({x_{i+1}}^2 - {x_i}^2) \cdot \frac{1}{2} \cdot \Delta f_a^b + (-x_{i+1} + x_i) \cdot a \cdot \Delta f_a^b \\
&= (x_{i+1} - x_i) \cdot f(a) + (x_{i+1} + x_i)(x_{i+1} - x_i) \cdot \frac{1}{2} \cdot \Delta f_a^b - (x_{i+1} - x_i) \cdot a \cdot \Delta f_a^b \\
&= h_i \cdot \left(f(a) - a \cdot \Delta f_a^b + \frac{x_{i+1} + x_i}{2} \cdot \Delta f_a^b \right)
\end{aligned}$$

This result can also be achieved by applying the trapezoidal rule $T(p) = H\frac{p(x_i)+p(x_{i+1})}{2}$ on the function p with $H = x_{i+1} - x_i$:

$$\begin{aligned} T_a^b(x_i, x_{i+1}) &:= H\frac{p(x_i) + p(x_{i+1})}{2} \\ &= (x_{i+1} - x_i) \cdot \frac{1}{2} \cdot \left(f(a) + \Delta f_a^b \cdot (x_i - a) + f(a) + \Delta f_a^b \cdot (x_{i+1} - a) \right) \\ &= (x_{i+1} - x_i) \cdot \frac{1}{2} \cdot \left(2 \cdot f(a) + \Delta f_a^b \cdot (x_{i+1} + x_i) - 2 \cdot \Delta f_a^b \cdot a \right) \\ &= h_i \cdot \left(f(a) - a \cdot \Delta f_a^b + \frac{x_{i+1} + x_i}{2} \cdot \Delta f_a^b \right) \end{aligned}$$

It can easily be shown that insertion of $x_i = a$, and $x_{i+1} = b$ generates the default trapezoidal rule for the support points a and b. Thus it holds by construction:

$$T_a^b(a, b) = H\frac{f(a) + f(b)}{2} = T(f), \text{ for } H = b - a$$

These results are summarized in the following definition:

Definition 2.4.1: Sliced trapezoidal rule

Let $f : D \subset \mathbb{R} \to \mathbb{R}$ a continuous function. Furthermore, let $\Omega = [a, b] \subset D$, $S_i = [x_i, x_{i+1}] \subset \Omega$, $h_i = x_{i+1} - x_i$, and $m_i = \frac{x_{i+1}+x_i}{2}$. Then we define the *Sliced trapezoidal rule* as

$$T_a^b(S_i) = T_a^b(x_i, x_{i+1}) = h_i \cdot \left(f(a) - a \cdot \Delta f_a^b + m_i \cdot \Delta f_a^b \right) \tag{2.4.1}$$

Furthermore the following theorem holds:

Theorem 2.4.1: Area of the sliced trapezoidal rule

Let $f : D \subset \mathbb{R} \to \mathbb{R}$ a continuous function, $\Omega = [a, b] \subset D$, and $S_i = [x_i, x_{i+1}] \subset \Omega$.
Then it yields: $T_a^b(S_i)$ is the area of slice S_i under the trapezoid spanned by a and b.

Proof. Compare with the derivation of $T_a^b(x_i, x_{i+1})$ above. ■

Now we deduce a weight-based variant of 2.4.1. Let $[l, r] \subset \Omega$, where l denotes the left support point of the trapezoid and r respectively the right support point. Rearranging the terms yields:

$$
\begin{aligned}
T_l^r(x_i, x_{i+1}) &= (x_{i+1} - x_i) \cdot \left(f(l) - l \cdot \Delta f_l^r + \frac{x_{i+1} + x_i}{2} \cdot \Delta f_l^r \right) \\
&= x_{i+1} \left[f(l) - l \cdot \left(\frac{f(l)}{l-r} - \frac{f(r)}{l-r} \right) + \frac{x_{i+1} + x_i}{2} \left(\frac{f(l)}{l-r} - \frac{f(r)}{l-r} \right) \right] \\
&\quad - x_i \left[f(l) - l \cdot \left(\frac{f(l)}{l-r} - \frac{f(r)}{l-r} \right) + \frac{x_{i+1} + x_i}{2} \left(\frac{f(l)}{l-r} - \frac{f(r)}{l-r} \right) \right] \\
&= x_{i+1} \cdot f(l) - x_{i+1} \cdot l \cdot \frac{f(l)}{l-r} + x_{i+1} \cdot l \cdot \frac{f(r)}{l-r} + x_{i+1} \cdot \frac{x_{i+1} + x_i}{2} \frac{f(l)}{l-r} - x_{i+1} \cdot \frac{x_{i+1} + x_i}{2} \frac{f(r)}{l-r} \\
&\quad - x_i \cdot f(l) + x_i \cdot l \cdot \frac{f(l)}{l-r} - x_i \cdot l \cdot \frac{f(r)}{l-r} - x_i \cdot \frac{x_{i+1} + x_i}{2} \frac{f(l)}{l-r} + x_i \cdot \frac{x_{i+1} + x_i}{2} \frac{f(r)}{l-r} \\
&= \quad f(l) \left[x_{i+1} - x_{i+1} \cdot l \cdot \frac{1}{l-r} + x_{i+1} \cdot \frac{x_{i+1} + x_i}{2} \frac{1}{l-r} - x_i + x_i \cdot l \cdot \frac{1}{l-r} - x_i \cdot \frac{x_{i+1} + x_i}{2} \frac{1}{l-r} \right] \\
&\quad + f(r) \left[x_{i+1} \cdot l \cdot \frac{1}{l-r} - x_{i+1} \cdot \frac{x_{i+1} + x_i}{2} \frac{1}{l-r} - x_i \cdot l \cdot \frac{1}{l-r} + x_i \cdot \frac{x_{i+1} + x_i}{2} \frac{1}{l-r} \right] \\
&= \quad f(l) \left[(x_{i+1} - x_i) - (x_{i+1} - x_i) \cdot \frac{l}{l-r} + (x_{i+1} - x_i) \cdot \frac{x_{i+1} + x_i}{2} \frac{1}{l-r} \right] \\
&\quad + f(r) \left[(x_{i+1} - x_i) \cdot \frac{l}{l-r} - (x_{i+1} - x_i) \cdot \frac{x_{i+1} + x_i}{2} \frac{1}{l-r} \right] \\
&= \quad f(l) \cdot (x_{i+1} - x_i) \cdot \left(1 - \frac{l}{l-r} + \frac{x_{i+1} + x_i}{2} \frac{1}{l-r} \right) \\
&\quad + f(r) \cdot \underbrace{(x_{i+1} - x_i)}_{=:h_i} \cdot \left(\frac{l}{l-r} - \underbrace{\frac{x_{i+1} + x_i}{2}}_{=:m_i} \underbrace{\frac{1}{l-r}}_{=-\frac{1}{r-l}=-\frac{1}{H}} \right) \\
&= f(l) \cdot h_i \cdot \left(1 + \frac{l}{H} - \frac{m_i}{H} \right) + f(r) \cdot h_i \cdot \left(\frac{m_i}{H} - \frac{l}{H} \right) \\
&= f(l) \cdot h_i \cdot \frac{H + l - m_i}{H} + f(r) \cdot h_i \cdot \frac{m_i - l}{H}
\end{aligned}
$$

This is our sought-for weight-based representation of 2.4.1. The result is summarized in the next definition and subsequent theorem:

Definition 2.4.2: Left/right weight of the sliced trapezoidal rule

Let $f : D \subset \mathbb{R} \to \mathbb{R}$ be a continuous function, $\Omega = [a, b] \subset D$, and $S_i = [x_i, x_{i+1}] \subset \Omega$, $h_i = x_{i+1} - x_i$. In addition, let (l, r) be the current support thus $H = r - l$.
Then we define the *left weight of slice* S_i as

$$
w_l \Big|_{x_i}^{x_{i+1}} = h_i \cdot \frac{H + l - m_i}{H} \tag{2.4.2}
$$

Analogously we define the *right weight of slice* S_i as

$$
w_r \Big|_{x_i}^{x_{i+1}} = h_i \cdot \frac{m_i - l}{H} \tag{2.4.3}
$$

Theorem 2.4.2: Weight-based area of a trapezoidal slice

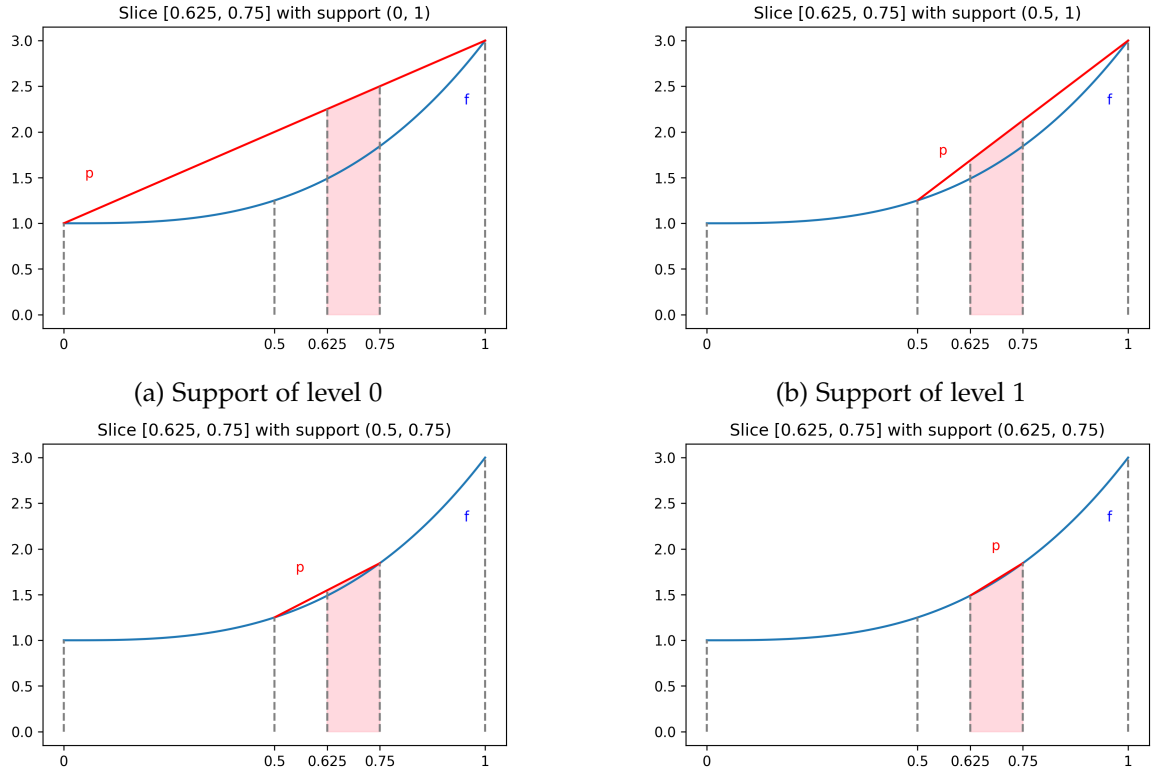
Let $f : D \subset \mathbb{R} \to \mathbb{R}$ be a continuous function, $\Omega = [a, b] \subset D$, and $S_i = [x_i, x_{i+1}] \subset \Omega$.
Then it yields:

$$T_a^b(S_i) = w_l \Big|_{x_i}^{x_{i+1}} \cdot f(l) + w_r \Big|_{x_i}^{x_{i+1}} \cdot f(r) \tag{2.4.4}$$

Proof. Compare with the derivation above. ■

With this theorems at hand it is now possible to approximate the area of a slice using a sequence of support. Before we formalize this procedure, we want to give a visual demonstration of the idea.

Example
Let $f : \mathbb{R} \to \mathbb{R}, f : 2 \cdot x^3 + 1$ with grid $[0, \frac{1}{2}, \frac{5}{8}, \frac{3}{4}, 1]$ be given. Hence, there are four slices $S_0 = [0, \frac{1}{2}]$, $S_1 = [\frac{1}{2}, \frac{5}{8}]$, $S_2 = [\frac{5}{8}, \frac{3}{4}]$, and $S_3 = [\frac{3}{4}, 1]$.
Slice S_2 entails the support sequence $(0, 1), (\frac{1}{2}, 1), (\frac{1}{2}, \frac{3}{4}), (\frac{5}{8}, \frac{3}{4})$ with max level 3:



(a) Support of level 0

(b) Support of level 1

(c) Support of level 2

(d) Support of level 3

Figure 2.5: Visualization of the support sequence for S_2

The other slices could be visualized analogously. To avoid repetition we will omit these figures. The following figures illustrate all slice refinements of support trapezoids for each level:

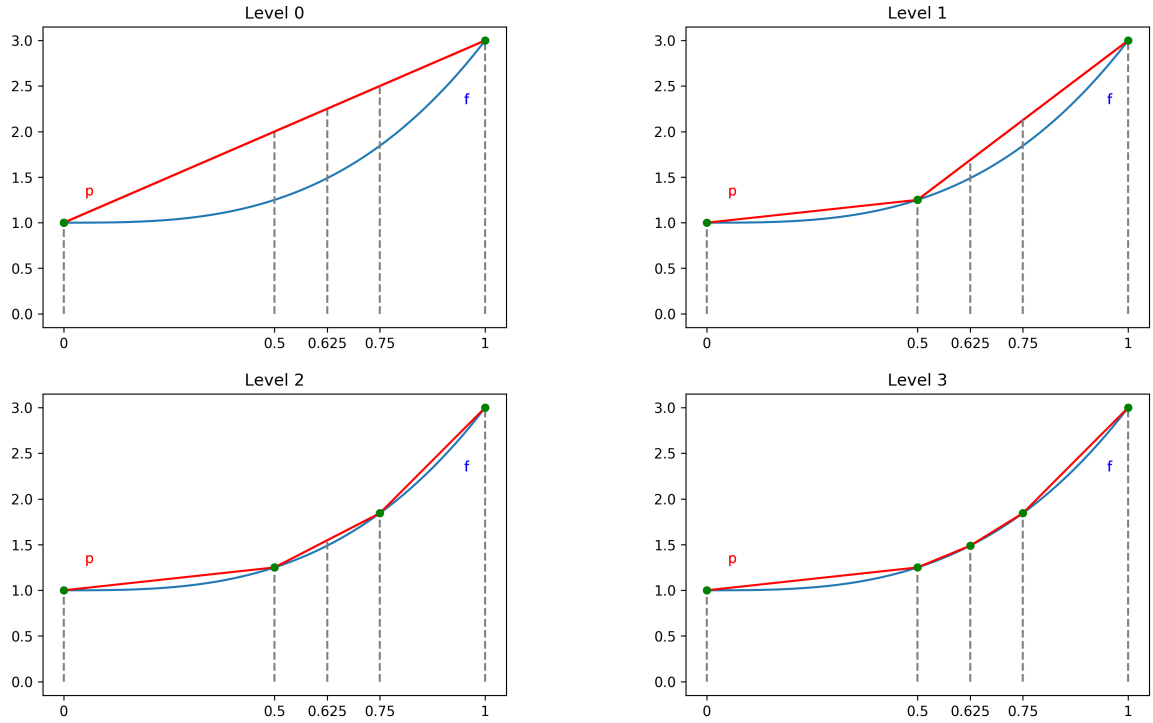


Figure 2.6: Visualization of support refinement for each level of the grid

With theorem 2.4.2 it yields for $S_0 = [0, \frac{1}{2}]$ and level 0:

$$T_0^1(S_0) = w_0\Big|_0^{\frac{1}{2}} \cdot f(0) + w_1\Big|_0^{\frac{1}{2}} \cdot f(1)$$

We compute the weight according to definition 2.4.2, with $H = (1-0) = 1$, $h_0 = (\frac{1}{2} - 0) = \frac{1}{2}$, and $m_0 = \frac{0+\frac{1}{2}}{2} = \frac{1}{4}$:

$$w_0\Big|_0^{\frac{1}{2}} = h_0 \cdot \frac{H + 0 - m_0}{H} = \frac{1}{2} \cdot \frac{1 + 0 - \frac{1}{4}}{1} = \frac{3}{8} \quad \text{and} \quad w_1\Big|_0^{\frac{1}{2}} = h_0 \cdot \frac{m_0 - 0}{H} = \frac{1}{2} \cdot \frac{\frac{1}{4} - 0}{1} = \frac{1}{8}$$

Finally, it follows $T_0^1(S_0) = \frac{3}{8} \cdot f(0) + \frac{1}{8} \cdot f(1) = \frac{3}{8} \cdot 1 + \frac{1}{8} \cdot 3 = \frac{3}{4}$. The results of all weights are summarized in this table:

	$S_0 = [0, \frac{1}{2}]$	$S_1 = [\frac{1}{2}, \frac{5}{8}]$	$S_2 = [\frac{5}{8}, \frac{3}{4}]$	$S_3 = [\frac{3}{4}, 1]$
Level 0	$w_0 = \frac{3}{8}, \quad w_1 = \frac{1}{8}$	$w_0 = \frac{7}{128}, \quad w_1 = \frac{9}{128}$	$w_0 = \frac{5}{128}, \quad w_1 = \frac{11}{128}$	$w_0 = \frac{1}{32}, \quad w_1 = \frac{7}{32}$
Level 1	$w_0 = \frac{1}{4}, \quad w_{\frac{1}{2}} = \frac{1}{4}$	$w_{\frac{1}{2}} = \frac{7}{64}, \quad w_1 = \frac{1}{64}$	$w_{\frac{1}{2}} = \frac{5}{64}, \quad w_1 = \frac{3}{64}$	$w_{\frac{1}{2}} = \frac{1}{16}, \quad w_1 = \frac{3}{16}$
Level 2	–	$w_{\frac{1}{2}} = \frac{3}{32}, \quad w_{\frac{3}{4}} = \frac{1}{32}$	$w_{\frac{1}{2}} = \frac{1}{32}, \quad w_{\frac{3}{4}} = \frac{3}{32}$	$w_{\frac{3}{4}} = \frac{1}{8}, \quad w_1 = \frac{1}{8}$
Level 3	–	$w_{\frac{1}{2}} = \frac{1}{16}, \quad w_{\frac{5}{8}} = \frac{1}{16}$	$w_{\frac{5}{8}} = \frac{1}{16}, \quad w_{\frac{3}{4}} = \frac{1}{16}$	–

Table 2.2: Sliced trapezoidal rule example: summary of all weights

For convenience and space purposes we abbreviated the notation $w_x := w_x\Big|_{x_i}^{x_{i+1}}$ for each column $S_i = [x_i, x_{i+1}]$. In the next sections we will derive an extrapolation formula using these weights.

2.4.2 Error expansion

In this section we will derive an error expansion of the sliced trapezoidal rule. The procedure is inspired by section 2.2.
As mentioned before, let $\Omega = [a, b] \subset \mathbb{R}$, $f : \Omega \to \mathbb{R}$ be a function with $f \in \mathcal{C}^\infty(\Omega)$. Additionally, let $m \in \Omega$. Then there exists a Taylor expansion of f in the development point m:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!}(x-m)^k \tag{2.4.5}$$

We define the slice with index $i \in \mathbb{N}_0$ as $S_i = [x_i, x_{i+1}]$. Let $m = \frac{x_{i+1}+x_i}{2}$ be the midpoint of slice S_i, $H = b - a$, and $h = x_{i+1} - x_i$.
The area of S_i can be expressed with theorem 2.4.2:

$$T_a^b(S_i) = h \cdot \frac{H+a-m}{H} \cdot f(a) + h \cdot \frac{m-a}{H} \cdot f(b)$$

Now the Taylor expansion with the midpoint m of the slice S_i is inserted. Subsequently the resulting expression is simplified:

$$\begin{aligned}
T_a^b(S_i) &= h \cdot \frac{H+a-m}{H} \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!}(a-m)^k + h \cdot \frac{m-a}{H} \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!}(b-m)^k \\
&= \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot h \cdot \left[(a-m)^k \cdot \frac{H+a-m}{H} + (b-m)^k \cdot \frac{m-a}{H} \right] \\
&= \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot h \cdot \left[(a-m)^k \cdot \left(1 + \frac{a-m}{H}\right) - (b-m)^k \cdot \frac{a-m}{H} \right] \\
&= h \cdot f(m) \cdot \left(1 + \frac{(a-m)}{H} - \frac{(a-m)}{H}\right) \\
&\quad + h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot (a-m)^k + h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot \frac{a-m}{H} \cdot \left[(a-m)^k - (b-m)^k \right] \\
&= h \cdot f(m) + h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot (a-m)^k + h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot \frac{a-m}{H} \cdot \left[(a-m)^k - (b-m)^k \right]
\end{aligned}$$

Integrating the Taylor series within the domain of S_i results in:

$$\begin{aligned}
\int_{x_i}^{x_{i+1}} f(x)\, dx &= \int_{x_i}^{x_{i+1}} \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{k!}(x-m)^k\, dx = \sum_{k=0}^{\infty} \int_{x_i}^{x_{i+1}} \frac{f^{(k)}(m)}{k!}(x-m)^k\, dx \\
&= \sum_{k=0}^{\infty} \left[\frac{f^{(k)}(m)}{(k+1)\cdot k!}(x-m)^{k+1} \right]_{x_i}^{x_{i+1}} \\
&= \sum_{k=0}^{\infty} \left[\frac{f^{(k)}(m)}{(k+1)!}(x_{i+1}-m)^{k+1} - \frac{f^{(k)}(m)}{(k+1)!}(x_i-m)^{k+1} \right] \\
&= \sum_{k=0}^{\infty} \frac{f^{(k)}(m)}{(k+1)!} \left[(x_{i+1}-m)^{k+1} - (x_i-m)^{k+1} \right]
\end{aligned}$$

To further simplify this expression we use the following trick: We know that
$x_{i+1} - m = \frac{2x_{i+1}}{2} - \frac{x_{i+1}+x_i}{2} = \frac{x_{i+1}-x_i}{2} = \frac{h}{2}$ and $x_i - m = \frac{2x_i}{2} - \frac{x_i+x_{i+1}}{2} = \frac{x_i-x_{i+1}}{2} = -(x_{i+1}-m) = -\frac{h}{2}$.
Thus, it holds that $(x_i - m)^{k+1} = (-1)^{k+1}(x_{i+1} - m)^{k+1}$ and the odd terms cancel each other out

$$(x_{i+1}-m)^{k+1} - (x_i-m)^{k+1} = \left\{ \begin{array}{ll} 0 & \text{, for } k \text{ odd} \\ 2\frac{h^{k+1}}{2^{k+1}} & \text{, for } k \text{ even} \end{array} \right\} \text{ for } k \geq 0$$

Inserting this expression into the integral of the taylor series yields:

$$\int_{x_i}^{x_{i+1}} f(x)\,dx = \sum_{\substack{k=0 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m)}{(k+1)!} \cdot 2 \cdot \frac{h^{k+1}}{2^{k+1}} = \sum_{\substack{k=0 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m) \cdot h^{k+1}}{2^k \cdot (k+1)!}$$
$$= h \cdot f(m) + \sum_{\substack{k=1 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m) \cdot h^{k+1}}{2^k \cdot (k+1)!} \tag{2.4.6}$$

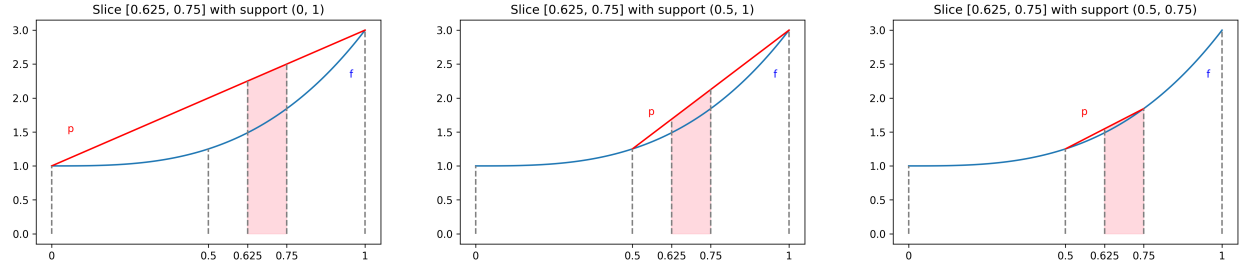
And now we have a system of two equations which is summarized here:

$$T_a^b(S_i) = h \cdot f(m) + h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot (a-m)^k + h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot \frac{a-m}{H} \cdot \left[(a-m)^k - (b-m)^k\right]$$
$$\int_{x_i}^{x_{i+1}} f(x)\,dx = h \cdot f(m) + \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m) \cdot h^{k+1}}{2^k \cdot (k+1)!}$$

Subtracting the second equation from the first one yields after rearrangement:

$$T_a^b(S_i) = \int_{x_i}^{x_{i+1}} f(x)\,dx$$
$$+ h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot (a-m)^k + h \sum_{k=1}^{\infty} \frac{f^{(k)}(m)}{k!} \cdot \frac{a-m}{H} \cdot \left[(a-m)^k - (b-m)^k\right]$$
$$- \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} \frac{f^{(k)}(m) \cdot h^{k+1}}{2^k \cdot (k+1)!}$$

Finally, we differentiate two cases: The refinement of the left boundary and respectively the refinement of the right boundary of support. This difference is illustrated in the following figures:



(a) Step 0: Starting point for the support refinement

(b) Step 1: Refinement of the **left** support boundary

(c) Step 2: Refinement of the **right** support boundary

Figure 2.7: Different cases of support refinement

Additionally, constant terms are factored out and grouped accordingly. This yields the following theorems about the error expansion of the sliced trapezoidal rule where $\int_{x_i}^{x_{i+1}} f(x)\,dx$ is abbreviated with $I_{x_i}^{x_{i+1}}(f)$.

Theorem 2.4.3: Sliced trapezoidal rule error expansion: right boundary refinement
Let $((a, b_j))_{j\in\mathbb{N}}$ be the support sequence of slice $S_i = [x_i, x_{i+1}]$. Thus it holds:

$$T_a^{b_j}(S_i) = I_{x_i}^{x_{i+1}}(f) + \sum_{k=1}^{\infty} C_{1,k}^{(a)} + \sum_{k=1}^{\infty} \frac{1}{H_j^{(a)}}(C_{2,k}^{(a)} - C_{3,k}^{(a)}(b_j - m)^k) - \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} C_k$$

for $H_j^{(a)} = b_j - a$ and the constants

$$C_{1,k}^{(a)} = h \cdot \frac{f^{(k)}(m)}{k!} \cdot (a-m)^k \qquad C_{3,k}^{(a)} = h \cdot \frac{f^{(k)}(m)}{k!} \cdot (a-m)$$
$$C_{2,k}^{(a)} = C_{1,k}^{(a)} \cdot (a-m) = h \cdot \frac{f^{(k)}(m)}{k!} \cdot (a-m)^{k+1} \quad \text{and} \quad C_k = \frac{f^{(k)}(m) \cdot h^{k+1}}{2^k \cdot (k+1)!}$$

Theorem 2.4.4: Sliced trapezoidal rule error expansion: left boundary refinement
Let $((a_j, b))_{j\in\mathbb{N}}$ be the support sequence of slice $S_i = [x_i, x_{i+1}]$. Thus it holds:

$$T_{a_j}^{b}(S_i) = I_{x_i}^{x_{i+1}}(f) + \sum_{k=1}^{\infty} C_{1,k}^{(b)}(a_j - m)^k + \sum_{k=1}^{\infty} \frac{a_j - m}{H_j^{(b)}}(C_{2,k}^{(b)} \cdot (a_j - m)^k - C_{3,k}^{(b)}) - \sum_{\substack{k=2 \\ k \text{ even}}}^{\infty} C_k$$

for $H_j^{(b)} = b - a_j$ and the constants:

$$C_{1,k}^{(b)} = h \cdot \frac{f^{(k)}(m)}{k!} \qquad C_{3,k}^{(b)} = C_{1,k}^{(b)} \cdot (b-m)^k = h \cdot \frac{f^{(k)}(m)}{k!} \cdot (b-m)^k$$
$$C_{2,k}^{(b)} = C_{1,k}^{(b)} \quad \text{and} \quad C_k = \frac{f^{(k)}(m) \cdot h^{k+1}}{2^k \cdot (k+1)!}$$

An important aspect to note is that the constant term C_k is the same for each of the two cases. However the expansion is more involved than one of the default trapezoidal rule.

2.4.3 Extrapolation

The previous results will be used to deduce an extrapolation formula for each refinement case. In contrast to Romberg's method we combine the approximations of only one slice using support trapezoids of decreasing support width. To achieve this result we follow a similar procedure as in section 2.2.3. Using the notation and definitions of the previous section we initially consider the first step of the extrapolation process.

Extrapolation with right boundary refinement
Let $(a, b) = (a, b_0)$ be the support of the first trapezoid. Furthermore, let (a, b_1) be the support of the second trapezoid for $b_1 = \frac{a+b_0}{2}$. Since the slice boundaries x_i, x_{i+1} stay constant during the extrapolation we abbreviate $I_{x_i}^{x_{i+1}}(f)$ with $I(f)$. For notational convenience we abbreviate $H_i := H_i^{(a)} = b_i - a$ for $i \in \{0, 1\}$.

We expand the sums of theorem 2.4.3 and obtain these two equations:

$$T_a^{b_0}(S_i) = I(f) + C_{1,1}^{(a)} + \frac{1}{H_0}C_{2,1}^{(a)} - \frac{1}{H_0}(b_0 - m)C_{3,1}^{(a)} \tag{I}$$
$$+ C_{1,2}^{(a)} + \frac{1}{H_0}C_{2,2}^{(a)} - \frac{1}{H_0}(b_0 - m)^2 C_{3,2}^{(a)} - C_2 + \dots$$
$$T_a^{b_1}(S_i) = I(f) + C_{1,1}^{(a)} + \frac{1}{H_1}C_{2,1}^{(a)} - \frac{1}{H_1}(b_1 - m)C_{3,1}^{(a)} \tag{II}$$
$$+ C_{1,2}^{(a)} + \frac{1}{H_1}C_{2,2}^{(a)} - \frac{1}{H_1}(b_1 - m)^2 C_{3,2}^{(a)} - C_2 + \dots$$

By multiplying the first equation with H_1^2 and the second equation with H_0^2 we receive:

$$H_1^2 \cdot T_a^{b_0}(S_i) = H_1^2 \cdot I(f) + H_1^2 \cdot C_{1,1}^{(a)} + \frac{H_1^2}{H_0}C_{2,1}^{(a)} - \frac{H_1^2}{H_0}(b_0 - m)C_{3,1}^{(a)} \tag{I*}$$
$$+ H_1^2 \cdot C_{1,2}^{(a)} + \frac{H_1^2}{H_0}C_{2,2}^{(a)} - \frac{H_1^2}{H_0}(b_0 - m)^2 C_{3,2}^{(a)} - H_1^2 \cdot C_2 + \dots$$
$$H_0^2 \cdot T_a^{b_1}(S_i) = H_0^2 \cdot I(f) + H_0^2 \cdot C_{1,1}^{(a)} + \frac{H_0^2}{H_1}C_{2,1}^{(a)} - \frac{H_0^2}{H_1}(b_1 - m)C_{3,1}^{(a)} \tag{II*}$$
$$+ H_0^2 \cdot C_{1,2}^{(a)} + \frac{H_0^2}{H_1}C_{2,2}^{(a)} - \frac{H_0^2}{H_1}(b_1 - m)^2 C_{3,2}^{(a)} - H_0^2 \cdot C_2 + \dots$$

The first extrapolation step can be written as $\Big((I^*) - (II^*)\Big) \div \Big(H_1^2 - H_0^2\Big)$:

$$\frac{H_1^2 \cdot T_a^{b_0}(S_i) - H_0^2 \cdot T_a^{b_1}(S_i)}{H_1^2 - H_0^2} = I(f) + C_{1,1}^{(a)} + \frac{\frac{H_1^2}{H_0} - \frac{H_0^2}{H_1}}{H_1^2 - H_0^2}C_{2,1}^{(a)} + \frac{\frac{H_0^2}{H_1}(b_1 - m) - \frac{H_1^2}{H_0}(b_0 - m)}{H_1^2 - H_0^2}C_{3,1}^{(a)}$$
$$+ C_{1,2}^{(a)} + \frac{\frac{H_1^2}{H_0} - \frac{H_0^2}{H_1}}{H_1^2 - H_0^2}C_{2,2}^{(a)} + \frac{\frac{H_0^2}{H_1}(b_1 - m)^2 - \frac{H_1^2}{H_0}(b_0 - m)^2}{H_1^2 - H_0^2}C_{3,2}^{(a)}$$
$$- C_2 + \dots$$

Solving for $I(f)$ yields the expression we searched for

$$I(f) = \frac{H_1^2 \cdot T_a^{b_0}(S_i) - H_0^2 \cdot T_a^{b_1}(S_i)}{H_1^2 - H_0^2}$$
$$- C_{1,1}^{(a)} - \frac{\frac{H_1^2}{H_0} - \frac{H_0^2}{H_1}}{H_1^2 - H_0^2}C_{2,1}^{(a)} - \frac{\frac{H_0^2}{H_1}(b_1 - m) - \frac{H_1^2}{H_0}(b_0 - m)}{H_1^2 - H_0^2}C_{3,1}^{(a)}$$
$$- C_{1,2}^{(a)} - \frac{\frac{H_1^2}{H_0} - \frac{H_0^2}{H_1}}{H_1^2 - H_0^2}C_{2,2}^{(a)} - \frac{\frac{H_0^2}{H_1}(b_1 - m)^2 - \frac{H_1^2}{H_0}(b_0 - m)^2}{H_1^2 - H_0^2}C_{3,2}^{(a)}$$
$$+ C_2 - \dots$$

where $H_i = b_i - a$ for $i \in \{0, 1\}$. These results are constrained to one extrapolation step. However, it is possible to generalize the formula above for $n > 1$ extrapolation steps under the condition that only the right support boundary is refined during the extrapolation. This implies that the left support point a does not change.
As one can see in comparison to Romberg there are more constant terms involved. These terms will be estimated and then eliminated from the equation (see subsection 2.5.2).

Extrapolation with left boundary refinement

Until now we proved the extrapolation only for the refinement of the right support point. Analogously, we obtain a formula for refinement of the left support point. Let $(a,b) = (a_0,b)$ the support of the first trapezoid. Furthermore, let (a_1,b) the support of the second trapezoid for $a_1 = \frac{a_0+b}{2}$. For notational convenience we abbreviate $H_i := H_i^{(b)} = b - a_i$ for $i \in \{0,1\}$.
We obtain the following with theorem 2.4.4:

$$T_{a_0}^b(S_i) = I(f) + (a_0-m)C_{1,1}^{(b)} + \frac{(a_0-m)^2}{H_0}C_{2,1}^{(b)} - \frac{a_0-m}{H_0}C_{3,1}^{(b)} \tag{I}$$
$$+ (a_0-m)^2C_{1,2}^{(b)} + \frac{(a_0-m)^3}{H_0}C_{2,2}^{(b)} - \frac{a_0-m}{H_0}C_{3,2}^{(b)} - C_2 + \dots$$
$$T_{a_1}^b(S_i) = I(f) + (a_1-m)C_{1,1}^{(b)} + \frac{(a_1-m)^2}{H_1}C_{2,1}^{(b)} - \frac{a_1-m}{H_1}C_{3,1}^{(b)} \tag{II}$$
$$+ (a_1-m)^2C_{1,2}^{(b)} + \frac{(a_1-m)^3}{H_1}C_{2,2}^{(b)} - \frac{a_1-m}{H_1}C_{3,2}^{(b)} - C_2 + \dots$$

To continue we multiply again crosswise with the support width H_i:

$$H_1^2 \cdot T_{a_0}^b(S_i) = H_1^2 \cdot I(f) + H_1^2 \cdot (a_0-m)C_{1,1}^{(b)} + \frac{H_1^2 \cdot (a_0-m)^2}{H_0}C_{2,1}^{(b)} - \frac{H_1^2 \cdot (a_0-m)}{H_0}C_{3,1}^{(b)} \tag{I*}$$
$$+ H_1^2 \cdot (a_0-m)^2C_{1,2}^{(b)} + \frac{H_1^2 \cdot (a_0-m)^3}{H_0}C_{2,2}^{(b)} - \frac{H_1^2 \cdot (a_0-m)}{H_0}C_{3,2}^{(b)} - H_1^2 \cdot C_2 + \dots$$
$$H_0^2 \cdot T_{a_1}^b(S_i) = H_0^2 \cdot I(f) + H_0^2 \cdot (a_1-m)C_{1,1}^{(b)} + \frac{H_0^2 \cdot (a_1-m)^2}{H_1}C_{2,1}^{(b)} - \frac{H_0^2 \cdot (a_1-m)}{H_1}C_{3,1}^{(b)} \tag{II*}$$
$$+ H_0^2 \cdot (a_1-m)^2C_{1,2}^{(b)} + \frac{H_0^2 \cdot (a_1-m)^3}{H_1}C_{2,2}^{(b)} - \frac{H_0^2 \cdot (a_1-m)}{H_1}C_{3,2}^{(b)} - H_0^2 \cdot C_2 + \dots$$

Finally, $\Big((I^*) - (II^*)\Big) \div \Big(H_1^2 - H_0^2\Big)$ yields:

$$\begin{aligned}
&\frac{H_1^2 \cdot T_{a_0}^b(S_i) - H_0^2 \cdot T_{a_1}^b(S_i)}{H_1^2 - H_0^2} \\
&= I(f) + \frac{H_1^2(a_0-m) - H_0^2(a_1-m)}{H_1^2 - H_0^2}C_{1,1}^{(b)} + \frac{\frac{H_1^2\cdot(a_0-m)^2}{H_0} - \frac{H_0^2\cdot(a_1-m)^2}{H_1}}{H_1^2 - H_0^2}C_{2,1}^{(b)} + \frac{\frac{H_0^2\cdot(a_1-m)}{H_1} - \frac{H_1^2\cdot(a_0-m)}{H_0}}{H_1^2 - H_0^2}C_{3,1}^{(b)} \\
&\quad + \frac{H_1^2(a_0-m)^2 - H_0^2(a_1-m)^2}{H_1^2 - H_0^2}C_{1,2}^{(b)} + \frac{\frac{H_1^2\cdot(a_0-m)^3}{H_0} - \frac{H_0^2\cdot(a_1-m)^3}{H_1}}{H_1^2 - H_0^2}C_{2,2}^{(b)} + \frac{\frac{H_0^2\cdot(a_1-m)}{H_1} - \frac{H_1^2\cdot(a_0-m)}{H_0}}{H_1^2 - H_0^2}C_{3,2}^{(b)} \\
&\quad - C_2 + \dots
\end{aligned}$$

By rearranging the terms we obtain the final result

$$\begin{aligned}
I(f) = &\frac{H_1^2 \cdot T_{a_0}^b(S_i) - H_0^2 \cdot T_{a_1}^b(S_i)}{H_1^2 - H_0^2} \\
&- \frac{H_1^2(a_0-m) - H_0^2(a_1-m)}{H_1^2 - H_0^2}C_{1,1}^{(b)} - \frac{\frac{H_1^2\cdot(a_0-m)^2}{H_0} - \frac{H_0^2\cdot(a_1-m)^2}{H_1}}{H_1^2 - H_0^2}C_{2,1}^{(b)} - \frac{\frac{H_0^2\cdot(a_1-m)}{H_1} - \frac{H_1^2\cdot(a_0-m)}{H_0}}{H_1^2 - H_0^2}C_{3,1}^{(b)} \\
&- \frac{H_1^2(a_0-m)^2 - H_0^2(a_1-m)^2}{H_1^2 - H_0^2}C_{1,2}^{(b)} - \frac{\frac{H_1^2\cdot(a_0-m)^3}{H_0} - \frac{H_0^2\cdot(a_1-m)^3}{H_1}}{H_1^2 - H_0^2}C_{2,2}^{(b)} - \frac{\frac{H_0^2\cdot(a_1-m)}{H_1} - \frac{H_1^2\cdot(a_0-m)}{H_0}}{H_1^2 - H_0^2}C_{3,2}^{(b)} \\
&+ C_2 - \dots
\end{aligned}$$

where $H_i = b - a_i$ for $i \in \{0,1\}$.

The above-mentioned derivation only showed one step of the extrapolation process for each of both cases. Now we would like to briefly sketch the idea for multiple extrapolation steps. Suppose we should perform an extrapolation with three step widths h_0, h_1, h_2. Similar to Romberg's table (see 2.1) we combined the result from h_0 with the one from h_1. The result of the previous extrapolation is combined with the result of the combination from h_1 and h_2. In total, we obtain the following result:

Extrapolation of a generic support sequence
Thus far we have distinguished between two cases of extrapolation, depending on the type of support boundary refinement. Now we are going to merge these two cases into one extrapolation formula. Similar to equation 2.1.3 we obtain for a slice $S_i = [x_i, x_{i+1}]$ of level $m \geq 0$:

$$\hat{T}(S_i) \coloneqq \sum_{j=0}^{m} c_{m,j} \cdot T_{a_j}^{b_j}(S_i), \quad \text{with} \quad c_{m,j} = \prod_{\substack{k=0,\\ k\neq j}}^{m} \frac{H_k^2}{H_k^2 - H_j^2} \tag{2.4.7}$$

where (a_j, b_j) is the support sequence element of level m for S_i.

Other possible extrapolations
We have also tried other extrapolation steps, such as

$$I(f) \approx \frac{H_0 \cdot T_a^{b_0}(S_i) - H_1 \cdot T_a^{b_1}(S_i)}{H_0 - H_1} \quad \text{and} \quad I(f) \approx \frac{\frac{H_0}{(a_0-m)^2} \cdot T_{a_0}^{b}(S_i) - \frac{H_1}{(a_1-m)^2} \cdot T_{a_1}^{b}(S_i)}{\frac{H_0}{(a_0-m)^2} - \frac{H_1}{(a_1-m)^2}}$$

for the right and left boundary refinement respectively. These formulas are obtained through different combinations of the above-mentioned equations (I) and (II). The idea was to factor different terms in equation 2.4.7, depending of the type of refinement. For $\tilde{H}_j \coloneqq \frac{H_j}{(a_j-m)^2}$ we investigated:

$$c_{m,j} = \prod_{\substack{k=0,\\ k\neq j}}^{m} \begin{cases} \frac{H_j}{H_j - H_k}, & \text{for a right boundary refinement} \\ \frac{\tilde{H}_j}{\tilde{H}_j - \tilde{H}_k}, & \text{for a left boundary refinement} \end{cases}$$

Despite having a simpler extrapolation representation they did not converge as quickly in our test cases as in the previous method.

Example (continued)
Now we are going to continue the example of section 2.4.1 by extrapolating the weights of table 2.2. The slice $S_0 = [0, \frac{1}{2}]$ has the weights $w_0^{(0)} = \frac{3}{8}$, $w_1^{(0)} = \frac{1}{8}$ for level 0 and $w_0^{(1)} = \frac{1}{4}$, $w_{\frac{1}{2}}^{(1)} = \frac{1}{4}$ for level 1. Using formula 2.4.7 we obtain:

$$\begin{aligned}
\hat{T}(S_0) &= \sum_{j=0}^{1} c_{1,j} \cdot T_{a_j}^{b_j}(S_i) = c_{1,0} \cdot T_{a_0}^{b_0}(S_i) + c_{1,1} \cdot T_{a_1}^{b_1}(S_i) \\
&= c_{1,0} \cdot (w_0^{(0)} \cdot f(0) + w_1^{(0)} \cdot f(1)) + c_{1,1} \cdot (w_0^{(1)} \cdot f(0) + w_{\frac{1}{2}}^{(1)} \cdot f(\tfrac{1}{2})) \\
&= \underbrace{\left(c_{1,0} \cdot w_0^{(0)} + c_{1,1} \cdot w_0^{(1)}\right)}_{=: \hat{w}_0^{[0,\frac{1}{2}]}} \cdot f(0) + \underbrace{c_{1,1} \cdot w_{\frac{1}{2}}^{(1)}}_{=: \hat{w}_{\frac{1}{2}}^{[0,\frac{1}{2}]}} \cdot f(\tfrac{1}{2}) + \underbrace{c_{1,0} \cdot w_1^{(0)}}_{=: \hat{w}_1^{[0,\frac{1}{2}]}} \cdot f(1)
\end{aligned}$$

Now we have found a respresentation of the extrapolated weights for S_0 which is grouped for each grid point (where $c_{0,1}, c_{1,0}, c_{1,1}$ are computed like in formula 2.4.7):

$$\hat{w}_0^{[0,\frac{1}{2}]} = -\frac{1}{3} \cdot \frac{3}{8} + \frac{4}{3} \cdot \frac{1}{4} = -\frac{1}{8} + \frac{1}{3} = \frac{5}{24}$$
$$\hat{w}_{\frac{1}{2}}^{[0,\frac{1}{2}]} = \frac{4}{3} \cdot \frac{1}{4} = \frac{1}{3}$$
$$\hat{w}_1^{[0,\frac{1}{2}]} = -\frac{1}{3} \cdot \frac{1}{8} = -\frac{1}{24}$$

The extrapolated weights of the other slices can be computed analogously. The results of these computations are summarized in the next table:

	$S_0 = [0, \frac{1}{2}]$	$S_1 = [\frac{1}{2}, \frac{5}{8}]$	$S_2 = [\frac{5}{8}, \frac{3}{4}]$	$S_3 = [\frac{3}{4}, 1]$	$\hat{w}_i = \sum_{k=0}^{3} \hat{w}_i^{S_k}$
$x_0 = 0$	$\hat{w}_0^{S_0} = \frac{5}{24}$	$-\frac{1}{51840}$	$-\frac{1}{72576}$	$\frac{1}{1440}$	$\frac{79}{378}$
$x_1 = \frac{1}{2}$	$\frac{1}{3}$	$\frac{2227}{45360}$	$-\frac{1}{80}$	$-\frac{1}{36}$	$\frac{194}{567}$
$x_2 = \frac{5}{8}$	–	$\frac{256}{2835}$	$\frac{256}{2835}$	–	$\frac{512}{2835}$
$x_3 = \frac{3}{4}$	–	$-\frac{2}{135}$	$\frac{26}{567}$	$\frac{8}{45}$	$\frac{592}{2835}$
$x_4 = 1$	$-\frac{1}{24}$	$\frac{53}{120960}$	$\frac{493}{362880}$	$\frac{143}{1440}$	$\frac{337}{5670}$

Table 2.3: Sliced trapezoidal rule example: summary of all extrapolated weights

Finally, we are able to compute the integral approximation:

$$\begin{aligned} I(f) &= \sum_{i=0}^{4} \hat{w}_i \cdot f(x_i) \\ &= \frac{79}{378} \cdot 1 + \frac{194}{567} \cdot \frac{5}{4} + \frac{512}{2835} \cdot \frac{381}{256} + \frac{592}{2835} \cdot \frac{59}{32} + \frac{337}{5670} \cdot 3 = \frac{1388}{945} \approx 1.469 \end{aligned}$$

Whereas the analytical solution would be $\int_0^1 2 \cdot x^3 + 1 dx = [\frac{1}{2}x^4 + x]_0^1 = \frac{3}{2} = 1.5$.

Exploitation of full grids: Thus far, we have ignored the fact that the grid $[0, \frac{1}{2}, \frac{5}{8}, \frac{3}{4}, 1]$ contains a partial full grid structure. Specifically, we partition the grid into containers:

Containers:	$C_0 = [0, \frac{1}{2}]$	$C_1 = [\frac{1}{2}, \frac{5}{8}, \frac{3}{4}]$	$C_2 = [\frac{3}{4}, 1]$
Levels:	$[0, 1]$	$[1, 3, 2]$	$[2, 0]$
Normalized levels:	$[0, 0]$	$[0, 1, 0]$	$[0, 0]$

Table 2.4: Example: Exploitation of the full grid structure

With this trick it is now possible to execute the sliced Romberg method in C_0 and C_2 separately.

We can furthermore execute the default weight-based Romberg method in C_1 with normalized levels:

$$\hat{w}_1^* := \hat{w}_3^* := \sum_{j=0}^{1} \frac{c_{1,j} \cdot H}{2^{j+1}} = c_{1,0} \cdot (\tfrac{3}{4} - \tfrac{1}{2}) \cdot \tfrac{1}{2} + c_{1,1} \cdot (\tfrac{3}{4} - \tfrac{1}{2}) \cdot \tfrac{1}{4} = \tfrac{1}{24}$$

$$\hat{w}_2^* := \sum_{j=1}^{1} \frac{c_{1,j} \cdot H}{2^{j}} = c_{1,1} \cdot (\tfrac{3}{4} - \tfrac{1}{2}) \cdot \tfrac{1}{4} = \tfrac{1}{6}$$

The new extrapolated total weights are given by the sum of some old weights from table 2.3 and the above computed new weights:

$$\hat{w}_0 = \hat{w}_0^{[0,\frac{1}{2}]} + 0 \ + \hat{w}_0^{[\frac{3}{4},1]} = \tfrac{5}{24} + \tfrac{1}{1440} = \tfrac{301}{1440}$$

$$\hat{w}_1 = \hat{w}_1^{[0,\frac{1}{2}]} + \hat{w}_1^* + \hat{w}_1^{[\frac{3}{4},1]} = \tfrac{1}{3} + \tfrac{1}{24} + (-\tfrac{1}{36}) = \tfrac{25}{72}$$

$$\hat{w}_2 = \hat{w}_2^{[0,\frac{1}{2}]} + \hat{w}_2^* + \hat{w}_2^{[\frac{3}{4},1]} = 0 + \tfrac{1}{6} + 0 = \tfrac{1}{6}$$

$$\hat{w}_3 = \hat{w}_3^{[0,\frac{1}{2}]} + \hat{w}_3^* + \hat{w}_3^{[\frac{3}{4},1]} = \tfrac{79}{360}$$

$$\hat{w}_4 = \hat{w}_4^{[0,\frac{1}{2}]} + 0 \ + \hat{w}_4^{[\frac{3}{4},1]} = \tfrac{83}{1440}$$

In total, we achieve a better approximation than before:

$$I(f) = \sum_{i=0}^{4} \hat{w}_i \cdot f(x_i) = \tfrac{79241}{53760} \approx 1.474$$

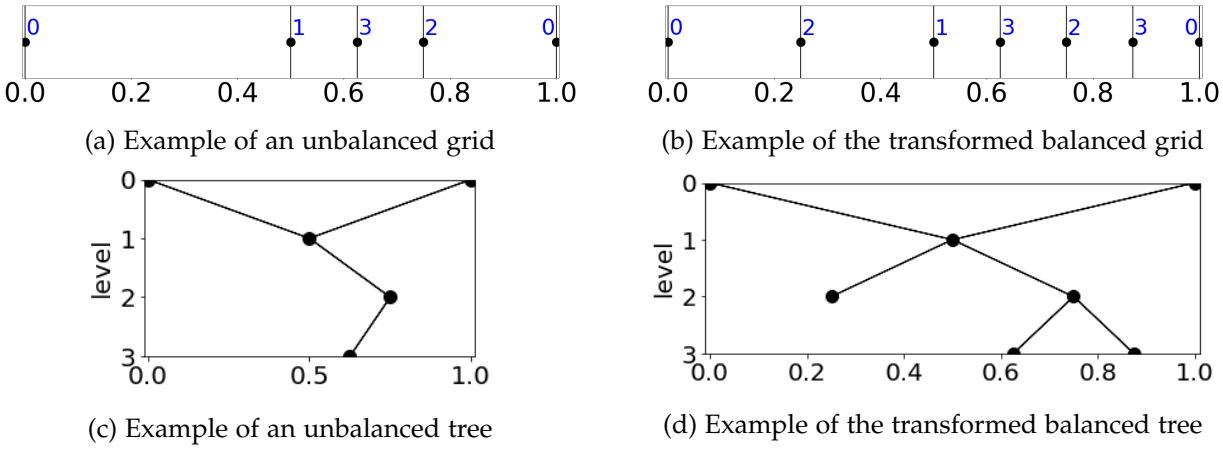
However, the default weight-based Romberg method on the full grid $F = [0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1]$ computes the exact analytical value. The sliced Romberg method is also exact on the grid F. Thus the missing point $\frac{1}{4}$ in the penultimate grid detoriates the approximations. In the following our goal is to further improve our adaptive extrapolation method by increasing the order in each slice.

2.5 Further improvements

Building upon the previous results about weight-based and adaptive extrapolation we want to propose several improvements of this technique. An naive but very effective approach is Grid Balancing, which is explained in subsection 2.5.1. Another possibility exploits the above derived error expansion of the sliced trapezoidal rule in the extrapolation process (subsection 2.5.2). Our third idea (subsection 2.5.3) uses an interpolatory approach of promising missing grid points in combination with a suitable grouping of subgrids. Finally, we investigate extrapolation using Simpon's rule instead of the trapezoidal rule as a base rule (subsection 2.5.4).

2.5.1 Grid balancing

In the previous example we mentioned that our sliced extrapolation method also achieves exactness on a full grid of level 2. Hence, we investigated how the method performs on balanced grids. This grid type is characterized by the following property: each inner grid point has either zero or two children in the refinement graph. This is illustrated in the next figure:



(a) Example of an unbalanced grid

(b) Example of the transformed balanced grid

(c) Example of an unbalanced tree

(d) Example of the transformed balanced tree

Figure 2.8: Comparison of unbalanced and balanced grids (and their refinement trees)

Although naturally this leads to more points it improves the results significantly in respect to the approximation error (see chapter 5). In the example of the previous section the balanced grid $[0, \frac{1}{4}, \frac{1}{2}, \frac{5}{8}, \frac{3}{4}, \frac{7}{8}, 1]$ achieves exactness using the adaptive method whereas without balancing there is still a small error. This idea of grid balancing has been investigated by other authors, e.g. from [Dir00, p.24], in the context of piece-wise Gauss basis functions. More information about this comparison is compiled in chapter 5.

2.5.2 Subtraction of extrapolation constants

Another proposition to improve the extrapolation results is subtracting appropriate terms of the error expansion. In section 2.4.3 we derived an extrapolation formula for the sliced trapezoidal rule. Its error representation depends on several constants. Each constant contains a factor of the form $\frac{f^{(k)}(m)}{k!}$ for $k \geq 0$, where $m \in \mathbb{R}$ is the midpoint of the slice. Since the function value of f at the point m is in general not available, it is necessary to approximate $f^{(k)}(m)$. This is, for example, achieved with Lagrange interpolation [Hea02, p.367].
To explain some important aspects about this extrapolation improvement we refer to the previous example. For the constant subtraction in $S_0 = [0, \frac{1}{2}]$ we need:

$$f(m) = \sum_{i=0}^{4} L_i(m) \cdot f(x_i) \quad \text{with} \quad L_i(m) = \prod_{\substack{j=0, \\ j \neq i}}^{4} \frac{m - x_j}{x_i - x_j}$$

where $m = \frac{0+\frac{1}{2}}{2} = \frac{1}{4}$. This expression can be differentiated, to obtain $f^{(k)}(m)$. Hence, each constant is now expressed as a weighted sum with factors that only depend on the support points.
In each extrapolation step these constants are taken into account. Depending on the extrapolation type (left or right) the necessary constants are constructed. Since these constants are a weighted sum of support points it is possible to distribute their contributions onto the extrapolation weights. Therefore this procedure adjusts the extrapolated weights according the error expansion of the extrapolation step.

2.5.3 Grid interpolation

In the last example it has been shown that the adaptive extrapolation method approximates the integral decently (taken into account that there are certainly fewer points than using a full grid). To improve this method, the idea of exploiting partial subgrids will now be extended by increasing the order of some subgrids. Promising missing grid points are interpolated and their contributions are distributed to the weights of the surrounding non-interpolated grid points.
Since the generic construction of this construction is rather complex we will only explain the underlying idea by continuing the example from above. Through interpolation of the missing point $x_1 = \frac{1}{4}$ of level 2 (colored in red), we increase the order in the first half of the grid.

(a) Adaptive grid

(b) Adaptive interpolated grid

Figure 2.9: Comparison of an adaptive grid and an interpolated adaptive grid

Obviously, it would be inefficient to interpolate all missing grid points which would in this example, provide a full grid of level 3. Instead, suitable candidates for interpolation points are determined by partitioning the grid in multiple subgrids. Some of those algorithms are explained in chapter 4. The philosophy is to interpolate as few points as possible but as many as necessary to achieve good results in a broad range of test functions.
For the interpolation there are a multitude of possibilities, for example Lagrange Interpolation [Hea02, p.316] or B-Splines [Hea02, p.330]. For simplicity we will explain the extrapolation method using Lagrange Interpolation. After suitable interpolation points are determined (see figure 2.9b), the Lagrange Basis is constructed with suitable support points (see chapter 4):

$$L_0(x) = \prod_{\substack{j=0,\\ j\neq 0}}^{4} \frac{x - x_j}{x_0 - x_j}, \qquad L_1(x) = \prod_{\substack{j=0,\\ j\neq 1}}^{4} \frac{x - x_j}{x_1 - x_j}, \qquad \ldots, \qquad L_4(x) = \prod_{\substack{j=0,\\ j\neq 4}}^{4} \frac{x - x_j}{x_4 - x_j}$$

where the grid is given by $G = [x_0, x_1, x_2, x_3, x_4] := [0, \frac{1}{2}, \frac{5}{8}, \frac{3}{4}, 1]$. Hence the interpolation polynomial is given by $p(x) = \sum_{i=0}^{4} L_i(x) \cdot f(x_i)$ and

$$p(\tfrac{1}{4}) = \tfrac{3}{20} \cdot f(x_0) + \tfrac{9}{2} \cdot f(x_1) + (-\tfrac{32}{5}) \cdot f(x_2) + 3 \cdot f(x_3) + (-\tfrac{1}{4}) \cdot f(x_4) = \tfrac{33}{32}$$
$$p(\tfrac{7}{8}) = -\tfrac{1}{160} \cdot f(x_0) + \tfrac{7}{16} \cdot f(x_1) + (-\tfrac{7}{5}) \cdot f(x_2) + \tfrac{7}{4} \cdot f(x_3) + \tfrac{7}{32} \cdot f(x_4) = \tfrac{599}{256}$$

Since $f(x) = 2x^3 + 1$ has degree $\deg(f) = 3$ the interpolation is exact in this example. Obviously this is not always the case for more complex functions. By including the interpolated points and partitioning the grid into two subgrids

$$G_1 = [x_0, x_{p_0}, x_1] := [0, \tfrac{1}{4}, \tfrac{1}{2}]$$
$$G_2 = [x_1, x_2, x_3, x_{p_1}, x_4] := [\tfrac{1}{2}, \tfrac{5}{8}, \tfrac{3}{4}, \tfrac{7}{8}, 1]$$

we can now execute two separate extrapolations on the full grids G_1 and G_2 with normalized levels. In this example we extrapolate using formula 2.3.1:
The weight proportion of the interpolated points $x \in \{\frac{1}{4}, \frac{7}{8}\}$ is distributed to the support points of the interpolation. Each non-interpolated grid point x_i $(0 \leq i \leq 4)$ has a total extrapolated weight of $\hat{w}_i = w_i + w_i'$, where w_i is given through formula 2.3.1 and $w_i' = w_i \cdot \sum_{x \in \{\frac{1}{4}, \frac{7}{8}\}} L_i(x)$.
The reason for this combination is best explained using an area-based extrapolation instead of

weight-based extrapolation: Firstly, we extrapolated on G_1. The composite trapezoidal rules for $h_0 = \frac{1}{2}$ and $h_1 = \frac{1}{4}$ are

$$T(h_0) = \tfrac{h_0}{2}(f(x_0) + f(x_1)) + h_0 \cdot \sum_{i=1}^{2^0-1} f(x_0 + i \cdot h_0) = \tfrac{h_0}{2}(f(x_0) + f(x_1)) = \tfrac{9}{16}$$

$$T(h_1) = \tfrac{h_1}{2}(f(x_0) + f(x_1)) + h_1 \cdot \sum_{i=1}^{2^1-1} f(x_0 + i \cdot h_1) = \tfrac{h_1}{2}(f(x_0) + f(x_1)) + h_1 \cdot f(\tfrac{1}{4})$$

$$\approx \tfrac{h_1}{2}(f(x_0) + f(x_1)) + h_1 \cdot p(\tfrac{1}{4}) = \tfrac{69}{128}$$

Through extrapolation with formula 2.1.3 we obtain:

$$T_1(0) = \sum_{j=0}^{1} c_{1,j} \cdot T(h_j) = c_{1,0} \cdot T(h_0) + c_{1,1} \cdot T(h_1) = -\tfrac{1}{3} \cdot \tfrac{9}{16} + \tfrac{4}{3} \cdot \tfrac{69}{128} = \tfrac{17}{32}$$

Instead of formula 2.1.3 it would also possible to use the sliced extrapolation from subsection 2.4.3. Since the interpolation is exact and an extrapolation on G_1 is exact ($\deg(f) = 3$), this "approximation" is equal to the analytical solution $\int_0^{1/2} f(x)\,dx$. By the previous calculation we recognize the structure of the weight-based interpolated extrapolation: Each non-interpolated grid point has, for one, an extrapolated weight w_i. In addition, all new weight proportions are added to w_i for which this point x_i is an support point for the interpolations. Thus we obtain $\hat{w}_i = w_i + w_i'$.
The extrapolation on G_2 is carried out analogously. For $h_0 = \frac{1}{2}$, $h_1 = \frac{1}{4}$, $h_2 = \frac{1}{4}$ we obtain

$$T(h_0) = \tfrac{h_0}{2}(f(x_1) + f(x_4)) + h_0 \cdot \sum_{i=1}^{2^0-1} f(x_1 + i \cdot h_0) = \tfrac{17}{16}$$

$$T(h_1) = \tfrac{h_1}{2}(f(x_1) + f(x_4)) + h_1 \cdot \sum_{i=1}^{2^1-1} f(x_1 + i \cdot h_1) = \tfrac{127}{128}$$

$$T(h_2) = \tfrac{h_2}{2}(f(x_1) + f(x_4)) + h_2 \cdot \sum_{i=1}^{2^2-1} f(x_1 + i \cdot h_2) = \tfrac{499}{512}$$

Again it holds that $p(\frac{7}{8}) = f(\frac{7}{8})$, for an interpolation polynomial p constructed with Lagrange Interpolation through $0, \frac{1}{2}, \frac{5}{8}, \frac{3}{4}, 1$, is similar to the previous construction on G_1. Extrapolation gives:

$$T_2(0) = \sum_{j=0}^{2} c_{2,j} \cdot T(h_j) = c_{2,0} \cdot T(h_0) + c_{2,1} \cdot T(h_1) + c_{2,2} \cdot T(h_2)$$
$$= \tfrac{1}{45} \cdot \tfrac{17}{16} + (-\tfrac{4}{9}) \cdot \tfrac{127}{128} + \tfrac{64}{45} \cdot \tfrac{499}{512} = \tfrac{31}{32}$$

In total we obtain the exact result on the domain $[0, 1]$ with $\frac{17}{32} + \frac{31}{32} = \frac{48}{32} = 1.5$, since both partial results are exact.
It is noteworthy that the extrapolation method with interpolation and without using constant subtraction is not always exact on an unbalanced interpolated grid (e.g. without the interpolated point $x_{p_1} = \frac{7}{8}$). This is due to the fact that not all errors cancel each other out.
This concludes the example. As mentioned previously, it is also possible to use other interpolation methods. For more information about the algorithmic implementation see chapter 4.

2.5.4 Romberg's method using Simpson's rule

Romberg's orginal method leverages the composite trapezoidal rule as base rule. These results are registered in the first column of Romberg's table (see table 2.1). The second column consists of composite Simpson's rules and the third column contains Boole's rules [Eng80, p.373].
Thus, one might try to skip the first column, which is the first extrapolation step, by using composite Simpson's rules as a base of the extrapolation process. The base rule is defined as [DR07, p.58]:

Definition 2.5.1: Composite Simpson's rule

Let $\Omega = [a, b]$ be the integration domain and $h = \frac{b-a}{n}$ for an even $n \in \{2, 4, 6, \dots\}$. Then we define the *composite Simpson's rule* by

$$S(h) = \frac{h}{3}\Big(f(x_0) + f(x_n) + 2\sum_{k=1}^{n/2-1} f(x_{2k}) + 4\sum_{k=1}^{n/2} f(x_{2k-1})\Big),$$

where $x_i = a + i \cdot h$ for all $0 \leq i \leq n-1$.

Similar to section 2.3, we obtain the following extrapolated weights:

$$w_i = \begin{cases} \frac{1}{3}\sum_{j=0}^{m} c_{m,j} \cdot h_j, & \text{for } i = 0, 2^m \\ \frac{4}{3}c_{m,i}h_i + \frac{2}{3}\sum_{j=l_i+1}^{m} c_{m,j} \cdot h_j, & \text{else} \end{cases} \tag{2.5.1}$$

for equidistant $x_0, \dots, x_{2^m}$ with $m \geq 0$. An Euler-Maclaurin formula for the Simpson's rule is given by [Luc, p.10]

Theorem 2.5.1: Euler-Maclaurin formula (Simpson's rule)

Let $\Omega = [a, b]$, $f \in C^\infty([a, b])$ and $x_i = a + i \cdot h$ for $0 \leq i \leq n$ and $h = \frac{b-a}{n}$. Then it holds

$$S(h) = \int_a^b f(t)dt + \frac{4}{3}\sum_{k=2}^{\infty} \tau_{2k} \cdot h^{2k}$$

with $\tau_{2k} = \frac{(4^{k-1}-1)B_{2k}}{(2k)!}\left[f^{(2k-1)}\right]_a^b$.

Thus the first order of Simpson's rule is $2 \cdot 2 = 4$. The error expansion contains the following powers of h: $h^4, h^6, h^8, h^{10}, \dots$ which differs from the expansion of the trapezoidal rule. This relationship must be taken into account when defining the $c_{m,j}$ for formula 2.5.1.

2.6 Related work

Many functions do not fulfill certain smoothness criteria or have different characteristics in some sub-intervals. Examples are, among others, functions that highly oscillate in one subinterval and are close to constant in other intervals. The result of the integral approximation can then be improved by using adaptive methods.

Adaptive Romberg method (from Prager): Prager introduced an adaptive variant of the Romberg-Quadrature [DR07, p.442] for one dimension.
Let f, a, b be defined as before and $n > 0$. First we select $h \geq 0$ so that $[a, a+h] \subset \Omega$ and $h \leq 4$. Now Romberg's table can be constructed to approximate $\int_a^{a+h} f(x)\, dx$. If $T_{0,k}$ and $T_{0,k-1}$ have at least n figures in common, then the computation for this subinterval is terminated and the result $T_{0,k}$ accepted. To determine the width h for the new subinterval the following cases are differentiated:

$$h := \begin{cases} \frac{3}{2} \cdot h & \text{if the sub-interval terminated with } k = 1 \\ \frac{3}{5} \cdot h & \text{if the sub-interval terminated with } k = 4 \\ 1 \cdot h & \text{else} \end{cases}$$

In each case the computation proceeds to the next subinterval. If however $T_{0,4}, T_{0,3}$ have less than n figures in common, h is decreased to $\frac{3}{5} \cdot h$ and Romberg's table for this smaller subinterval is constructed. Finally the amount of subintervals is restriced by $\frac{8 \cdot (b-a)}{h}$ otherwise the algorithm exits with an error.

Adaptive Romberg method (from Genz): Genz [Gen72] describes the method for one dimensional cases as follows: Initially the whole integration interval $[a, b]$ is considered. Subsequently we approximate the integral value using a suitable low order rule and denote the result with S_0. Afterwards the same rule is applied to $[a, \frac{a+b}{2}]$ and $[\frac{a+b}{2}, b]$ where S_1 denotes the sum of both sub-interval results. Thereafter each sub-interval is halved again and the sum of all four approximations is defined as S_2.
In general terms, let $i_k^{(p-1)}$ be defined as the k-th sub-interval in the iteration step $p-1$. Then this interval is halved into the two sub-intervals $i_{2k-1}^{(p)}$ and $i_{2k}^{(p)}$ in the next iteration step p. Furthermore $R_k^{(p)}$ is defined as result of a low order quadrature rule applied to $i_k^{(p)}$. With this notation it follows: $S_p = \sum_{k=1}^{2^p} R_k^{(p)}$ for each iteration step $p \geq 1$.
The approximation error $e_k^{(p)}$ in sub-interval k in iteration p is estimated using error estimators:

1. Terminate the computation on this sub interval, if $e_k^{(p)} < \alpha \cdot \left| R_k^{(p)} - (R_{2k-1}^{(p+1)} + R_{2k}^{(p+1)}) \right|$. Usually $\alpha = 1$. The result of the whole interval is compared with the sum of its two halved children.
2. Let R^* be another low order quadrature rule. The computation on a sub-interval is terminated if $e_k^{(p)} < \alpha \cdot \left| R_k^{(p)} - R_k^{*\,(p)} \right|$. In this case $\alpha > 1$ is usually chosen.

Let G_p contain the sum of all contributions of sub-intervals where the computation is already finished (convergence has already been obtained) for iteration step p. Furthermore, let E be the total error estimate in this step. We apply the low order quadrature rule only to sub-intervals that are still under consideration (meaning convergence has not been obtained).
We obtain the following relation for each iteration step p:

$$S_p = \sum_{k=1}^{n_p} R_k^{(p)} + G_{p-1} \text{ and } \epsilon_{p+1} = \frac{E - E_p}{n_{p+1}}$$

The subdivision in a region can be stopped if $e_k^{(p)} < \epsilon_p$. The previous method can be enhanced by using Richardson extrapolation for a specific type of functions (with high order continuous derivatives). After all the error expansion of S_p is not effected significantly by the adaptive method [Gen72, p.13]. Thus we can use Romberg's method on the results S_p.

Adaptive quadrature of higher order (from Bonk): T. Bonk presented a method which combines the generalized Romberg-Quadrature with the generalized Archimedes quadrature [Bon95, p.68] [Bon94, p.62] . Since both quadrature methods operate on Sparse Grids it is convenient to store grid points in a tree data structure. One major problem that needs to be addressed is how to handle missing grid points in the extrapolation process due to the adaptive nature of the Archimedes quadrature.
The one-dimensional case can be summarized, as follows, using a scheme based on the midpoint rule: Let $\Omega = [a, b]$. First we compute the midpoint rule of the root node M using $n = 1$ and $h = b - a$. The approximation $\hat{M}_0 = M_0(f) = M_f(h) = h \cdot f(a + \frac{1}{2}h)$ will be stored as first entry in the list of node M.

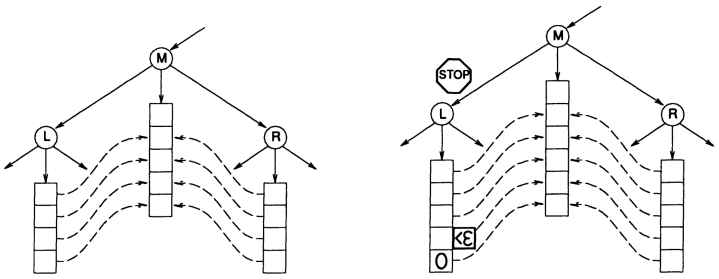


Figure 2.10: Adaptive quadrature of higher order (one-dimensional-case).
Taken from [Bon94, p.62].

In the second iteration, the integration domain Ω is halved into two sub-intervals $I_L = [a, \frac{a+b}{2}]$ and $I_R = [\frac{a+b}{2}, b]$, resulting in $\Omega = I_L \cup I_R$. For each sub-interval the midpoint sum is computed with $h = \frac{b-a}{2}$. The results are stored as the first list element in node L or respectively R.

$$M_0^{(L)} = h \cdot f\left(\frac{3a+b}{4}\right) \quad \text{or respectively} \quad M_0^{(R)} = h \cdot f\left(\frac{a+3b}{4}\right) \tag{2.6.1}$$

Based upon $M_0^{(L)}$ and $M_0^{(R)}$ we extrapolate $\hat{M}_1$ and store the difference $\hat{M}_1 - \hat{M}_0$ as second list item in node M. Consequently, the extrapolated value $\hat{M}_1$ is of order $\mathcal{O}(h^4)$. The third iteration halves each sub-interval I_L and I_R again. This results in extrapolated values of order $\mathcal{O}(h^4)$ in each sub-interval I_L and I_R. Each improvement is stored as second element in the respective list of node L or R. With this values one can extrapolate an approximation of order $\mathcal{O}(h^6)$ at node M and store the improvement as third element in the list of node M. This interval splitting is carried on recursively leading to results of higher order.
Finally, a termination criterion for the sub-interval splitting is required: When the absolute value of the last improvement is smaller than a given tolerance $\varepsilon > 0$, a stop flag will be set and this sub-tree will not be considered in the next iterations. Unfortunately some improvements for the extrapolation might be missing as a result of the adaptivity. Thus the improvements for all higher orders in the list of this node are set to zero. If all nodes have a stop flag assigned, the procedure stops and returns the approximation of highest order. This method can be generalized to multiple dimensions by splitting the domain in each dimension which results in a multidimensional tree.
Further information can be found in [Bon94, p.63] and [Bon95, p.75].

3 Sparse Grids

Generalizing the *Archimedes quadrature* and using the *Cavalieri Principle* leads naturally to Sparse Grids [Bon95]. First we will summarize an intuition about Sparse Grids before the generic construction of Sparse Grids will be presented. Subsequently, we summarize important aspects of the *Sparse Grid Combination Technique*.

3.1 Intuition: Generalized Archimedes quadrature

This section summarizes the method presented in [Bon95, pp. 6].

One-dimensional case
Let $f : \mathbb{R} \to \mathbb{R}$, $a, b \in \mathbb{R}$, $a < b$. The goal is to determine $F_1(f, a, b) = \int_a^b f(x)\, dx$. Firstly, we divide the enclosed area into two sub-problems T_1 and S_1.

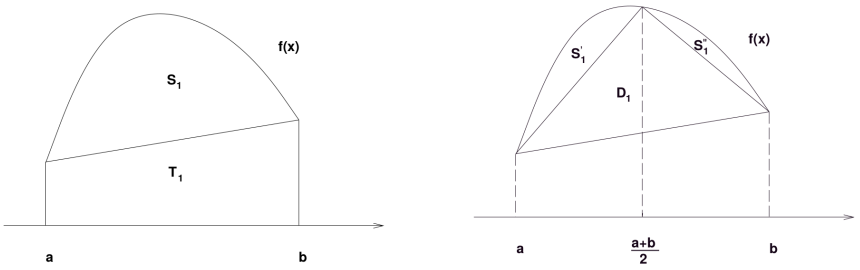


Figure 3.1: Method of exhaustion. Taken from [Bon95, p.7].

Hence it holds: $F_1(f, a, b) = \underbrace{\frac{f(a) + f(b)}{2}(b - a)}_{T_1} + S_1(f, a, b)$.
Besides, the area of the inscribed triangle D_1 is given by

$$D_1(f, a, b) = \left(f\left(\frac{a+b}{2}\right) - \frac{f(a) + f(b)}{2} \right) \cdot \frac{b-a}{2}.$$

The following recursive relation holds with *hierarchical surplus* $g_1(f, a, b) = f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2}$:

$$S_1(f, a, b) = \begin{cases} g_1(f, a, b) \cdot \frac{b-a}{2}, & \text{if } \left|g_1(f, a, b)\right| \leq \epsilon_{\text{tol}} \\ g_1(f, a, b) \cdot \frac{b-a}{2} + S_1\left(f, a, \frac{a+b}{2}\right) + S_1\left(f, \frac{a+b}{2}, b\right), & \text{else} \end{cases}$$

for an $\varepsilon_{\text{tol}} > 0$. A noteworthy fact is that the result of the above rule is a trapezoidal rule with step width $h = \frac{b-a}{2^m}$ and $2^m + 1$ equidistant support points if the recursion is bounded by $m > 0$ incarnations.

Two-dimensional case
Let $f : \mathbb{R}^2 \to \mathbb{R}$, $a_i, b_i \in \mathbb{R}$, $a_i < b_i$ for $i \in 1, 2$. This time our goal is to determine
$F_2(f, a_1, a_2, b_1, b_2) = \int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x_1, x_2)\, dx_2\, dx_1$

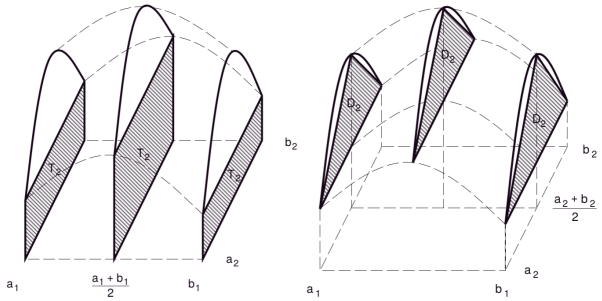


Figure 3.2: Cavalieri's Principle. Taken from [Bon95, p.9]

By applying Cavalieri's Principle we reduce the problem to the one-dimensional case:

$$F_2(f, a_1, a_2, b_1, b_2) = F_1\Bigg(\underbrace{\frac{f(x_1, a_2) + f(x_1, b_2)}{2} \cdot (b_2 - a_2)}_{T_2(x_1)}, a_1, b_1\Bigg) + S_2(f, a_1, a_2, b_1, b_2)$$

Furthermore, we get $D_2(x_1) = \left(f\left(x_1, \frac{a_2+b_2}{2}\right) - \frac{f(x_1,a_2)+f(x_1,b_2)}{2}\right) \cdot \frac{b_2-a_2}{2}$.
Let $g_2(f, a_1, a_2, b_1, b_2) = g_1(f(x_1, \frac{a_2+b_2}{2}), a_1, b_1) - \frac{1}{2} \cdot (g_1(f(x_1, a_2), a_1, b_1) + g_1(f(x_1, b_2), a_1, b_1))$ be the two dimensional hierarchical surplus. Then the following recursion holds:

$$S_2(f, a_1, a_2, b_1, b_2) = \begin{cases} F_1(D_2(x_1), a_1, b_1), & \text{for } \left|g_2(f, a_1, a_2, b_1, b_2)\right| \leq \epsilon_{\text{tol}} \\ F_1(D_2(x_1), a_1, b_1) & \\ \quad + S_2\left(f, a_1, a_2, b_1, \frac{a_2+b_2}{2}\right) & \\ \quad + S_2\left(f, a_1, \frac{a_2+b_2}{2}, b_1, b_2\right), & \text{otherwise} \end{cases}$$

for an $\varepsilon_{\text{tol}} > 0$.
Let us consider the examples $f : [-1, 1]^2 \to \mathbb{R}, f(x_1, x_2) = x_1^2 \cdot x_2^2$ with $\varepsilon_{\text{tol}} = 10^{-3}$ and $g : [-8, 8]^2 \to \mathbb{R}, f(x_1, x_2) = e^{x_1+x_2}$ with $\varepsilon_{\text{tol}} = 10^3$ [Bon94]. The following figure displays the evaluated points of f at the left side and g at the right:

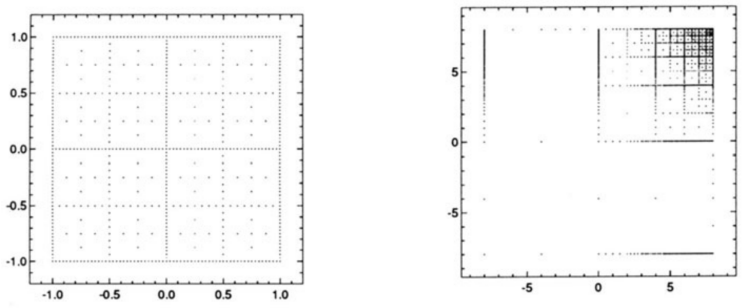


Figure 3.3: Sparse Grids for f (left) and g (right). Taken from [Bon94, p.58].

n-dimensional case
Let $f : \mathbb{R}^n \to \mathbb{R}$, $a_i, b_i \in \mathbb{R}$, $a_i < b_i$ for $1 \leq i \leq n$. Finally the goal is to determine the n-dimensional integral $F_d(f, a_1, \ldots, a_n, b_1, \ldots, b_n) = \int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f(x_1, \ldots, x_n)\, dx_n \ \ldots \ dx_1$. The above-mentioned formulas can be generalized by applying the idea of the two dimensional case using Cavalieri's Principle. For more information see [Bon95, p.9].

3.2 Hierarchical basis and Nodal basis

In this section we will explain the necessary concepts for the construction of Sparse Grids [Dir00] [Gar13] [GG10] [Pfl10]. Our first goal is to interpolate a function $f : \Omega \to \mathbb{R}$ using piece-wise d-linear functions for $d \geq 1$. Furthermore, we will restrict ourselves to $\Omega = [0,1]^d$, because other domains might be transformed to the d-dimensional unit cube using a suitable transformation function. Additionally, we assume $f(x) = 0$ for all $x \in \partial\Omega$ (all function values in boundary points are 0). Pursuing Archimedes' idea of inscribing triangles we first define the following term for $d = 1$.

Definition 3.2.1: Standard hat function

Let $\varphi : \mathbb{R} \to \mathbb{R}, \phi(x) := \max(1 - |x|, 0)$. Then we call ϕ the *Standard hat function*.

Using one-dimensional piece-wise linear basis functions $\varphi_i(x)$ and step size $h_n = 2^{-n}$ we are able to interpolate a function $f : [0,1] \to \mathbb{R}$ with $f(x) \approx u(x) := \sum_i \alpha_i \varphi_i(x)$ for certain coefficients $\alpha_i \in \mathbb{R}$. Through translation and dilatation we define:

Definition 3.2.2: One-dimensional hat function

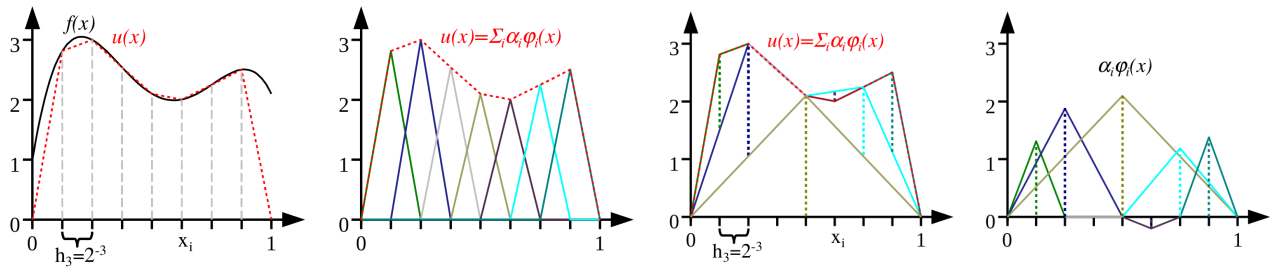
Let $l \geq 1$ denote a level, $1 \leq i \leq 2^l$ an index, and $h_l = 2^{-l}$ the step size of level l. Then we call

$$\varphi_{l,i}(x) := \varphi\left(\frac{x - i \cdot h_l}{h_l}\right) = \varphi(h_l^{-1} \cdot x - i) = \varphi(2^l \cdot x - i) \tag{3.2.1}$$

the *One-dimensional hat function* of level l and index i. Moreover, we define the *Space of piece-wise linear functions* of level $l \geq 1$ as

$$V_l := \operatorname{span}\{\varphi_{l,i} \mid 1 \leq i \leq 2^l - 1\}. \tag{3.2.2}$$

Let $x_{l,i}$ denote the grid point $x_{l,i} = i \cdot h_l$ for $0 \leq i \leq 2^l$. The function $\varphi_{l,i}$ has $x_{l,i}$ in its center of support given $[x_{l,i} - h_l, x_{l,i} + h_l]$. This family of basis functions is also called *Nodal basis* [GG10]. Unfortunately one disadvantage of the Nodal basis is that basis functions overlap.



(a) Using the Nodal basis. Taken from [Pfl10, p.7].

(b) Using the Hierarchical basis. Taken from [Pfl10, p.9].

Figure 3.4: Comparison of piece-wise linear interpolation for $d = 1$, $h_3 = 2^{-3}$.

If another basis is chosen, it is possible to reduce the number of basis functions and omit basis functions that have less contribution to the interpolation (small support).

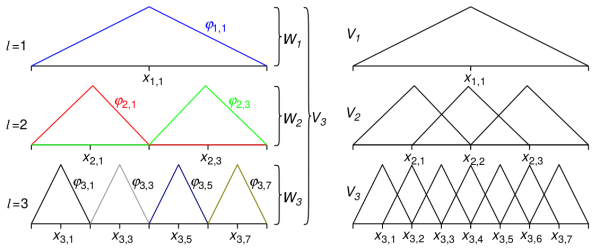


Figure 3.5: Comparison between the Hierarchical basis (left) and Nodal basis (right) for $\varphi_{l,i}$ where $d = 1$ and $1 \leq l \leq 3$. Taken from [Pfl10, p.9].

Now we are able to define the *Hierarchical increment spaces*.

Definition 3.2.3: Hierarchical increment spaces, Hierarchical basis

Let $l \geq 1$, $V_l := \text{span}\{\varphi_{l,i} \mid 1 \leq i \leq 2^l - 1\}$, and $I_l := \{i \in \mathbb{N} \mid 1 \leq i \leq 2^l - 1 \ \wedge \ i \text{ odd}\}$. We define the *Hierarchical increment space* of level l as

$$W_l := \{\varphi_{l,i} \mid i \in I_l\} \tag{3.2.3}$$

Moreover, we call the basis of W_l *Hierarchical basis* of level l.

For level $l \geq 1$ it holds $V_l = \bigoplus_{k \leq l} W_k$ (compare with figure 3.5). Thus it is possible to interpolate the above function f for a level $l \geq 1$ through u, defined by:

$$f(x) \approx u(x) := \sum_{k=1}^{l} \sum_{i \in I_k} \alpha_{k,i} \cdot \varphi_{k,i}(x) \tag{3.2.4}$$

with *hierarchical surpluses* $\alpha_{k,i} \in \mathbb{R}$. In more detail, this theorem holds [Gar13, p.6] [1]:

Theorem 3.2.1: Hierarchical surplus

Let f be a function with bounded second derivative. Then we can compute the hierarchical surpluses:

$$\alpha_{l,i} := f(x_{l,i}) - \frac{f(x_{l,i-1}) + f(x_{l,i+1})}{2} = \int_\Omega -\frac{h_l}{2} \cdot \varphi_{l,i}(x) \cdot f''(x)\, dx$$

[1] https://www5.in.tum.de/lehre/vorlesungen/asc/ss13/hierarch_integral_representation.pdf

Generalisation to n dimensions (using tensor product construction)
Now we consider $d > 1$ but still with the above restrictions to Ω and $\partial\Omega$. First of all, we introduce several notations for convenience:

Definition 3.2.4: Multi-dimensional level vector, step-width, grid point

Let $d > 1$. We define the multi-dimensional level vector $\boldsymbol{l} := (l_1, \ldots, l_d)$ where the level of dimension k is defined by $l_k \geq 1$ for all $1 \leq k \leq d$.
Analogously, the step width $\boldsymbol{h_l}$ of level $\boldsymbol{l}$ is given by the multi-index $\boldsymbol{h_l} := 2^{-\boldsymbol{l}} := (h_1, \ldots, h_d)$.
Furthermore, we define the grid point $\boldsymbol{x_{l,i}}$ of level $\boldsymbol{l}$ and index $\boldsymbol{i}$ as

$$\boldsymbol{x_{l,i}} := (x_{l_1,i_1}, \ldots, x_{l_d,i_d})$$

for $\mathbf{1} \leq \boldsymbol{i} \leq 2^{\boldsymbol{l}} - \mathbf{1}$ where $\mathbf{1}$ denotes the vector $(1, \ldots, 1) \in \mathbb{R}^n$, $2^{\boldsymbol{l}} := (2^{l_1}, \ldots, 2^{l_d})$
and $\forall \boldsymbol{o} = (o_1, \ldots, o_d), \boldsymbol{p} = (p_1, \ldots, p_d) : \ \boldsymbol{o} \leq \boldsymbol{p} \ \Leftrightarrow \ (\forall 1 \leq k \leq d : \ o_k \leq p_k)$

With this notational prepatory it is now possible to transfer the concepts of $d = 1$ to higher dimensions.

Definition 3.2.5: Multidimensional hat function

Let $\boldsymbol{l} \geq \mathbf{1}$, $\mathbf{1} \leq \boldsymbol{i} \leq 2^{\boldsymbol{l}}$, and $\boldsymbol{h_l} = 2^{-\boldsymbol{l}}$. Then we call

$$\varphi_{\boldsymbol{l},\boldsymbol{i}}(\boldsymbol{x}) := \prod_{k=1}^{d} \varphi_{l_k,i_k} \tag{3.2.5}$$

the *Multi-dimensional hat function* of level l and index i. Moreover, we define the *Space of piece-wise linear functions* of level $l \geq 1$ as

$$V_l := \operatorname{span}\{\varphi_{l,i} \mid \mathbf{1} \leq \boldsymbol{i} \leq 2^{\boldsymbol{l}} - \mathbf{1}\}. \tag{3.2.6}$$

This enables us to generalize the hierarchical increment spaces 3.2.3 to multiple dimensions:

Definition 3.2.6: Multidimensional hierarchical increment spaces, hierarchical basis

Let $\boldsymbol{l} \geq 1$, $V_{\boldsymbol{l}} := \operatorname{span}\{\varphi_{\boldsymbol{l},\boldsymbol{i}} \mid \mathbf{1} \leq \boldsymbol{i} \leq 2^{\boldsymbol{l}} - \mathbf{1}\}$, and
$I_{\boldsymbol{l}} := \{\boldsymbol{i} \in \mathbb{N}^d \mid \mathbf{1} \leq \boldsymbol{i} \leq 2^{\boldsymbol{l}} - \mathbf{1} \ \wedge \ \forall 1 \leq k \leq d : \ i_k \text{ odd}\}$. We define the *Hierarchical increment space* of level $\boldsymbol{l}$ as

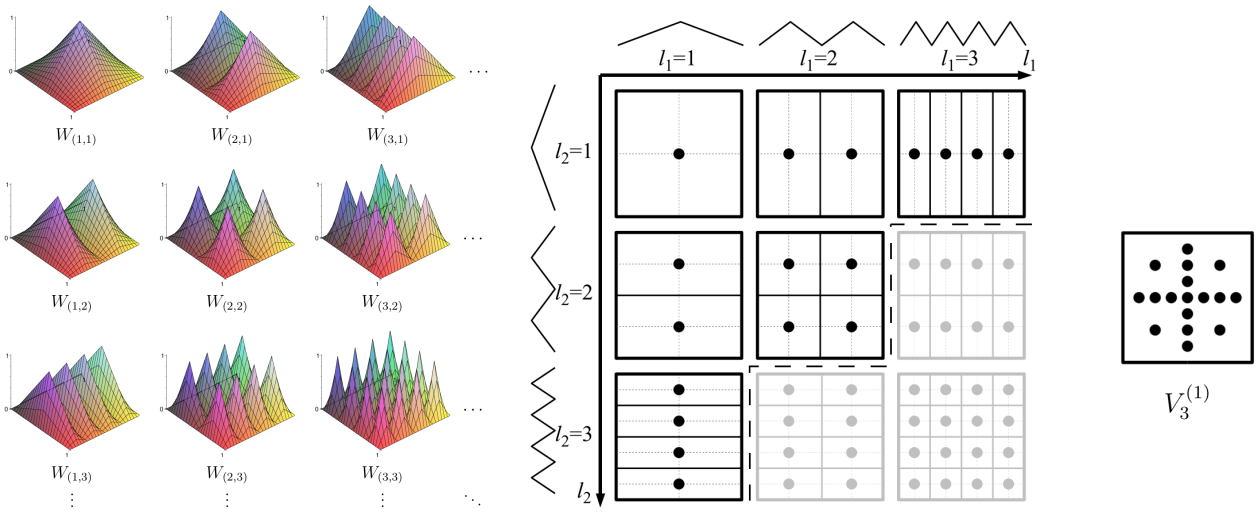
$$W_{\boldsymbol{l}} := \{\varphi_{\boldsymbol{l},\boldsymbol{i}} \mid \boldsymbol{i} \in I_{\boldsymbol{l}}\} \tag{3.2.7}$$

Moreover, we call the basis of $W_{\boldsymbol{l}}$ *Hierarchical basis* of level $\boldsymbol{l}$.

Just like in the one-dimensional case it holds by construction: $V_{\boldsymbol{l}} = \bigoplus_{\boldsymbol{k} \leq \boldsymbol{l}} W_{\boldsymbol{k}}$. Thus we can interpolate a function $f : \Omega \to \mathbb{R}$ with the properties mentioned above for a level $\boldsymbol{l} \geq \mathbf{1}$ through u defined by:

$$f(x) \approx u(x) := \sum_{\boldsymbol{k}=\mathbf{1}}^{\boldsymbol{l}} \sum_{\boldsymbol{i} \in I_{\boldsymbol{k}}} \alpha_{\boldsymbol{k},\boldsymbol{i}} \cdot \varphi_{\boldsymbol{k},\boldsymbol{i}}(\boldsymbol{x}) \tag{3.2.8}$$

with *hierarchical surpluses* $\alpha_{\boldsymbol{k},\boldsymbol{i}} \in \mathbb{R}$. The approximation error of this interpolations for sufficiently smooth functions is $\left\| f(\boldsymbol{x}) - u(\boldsymbol{x}) \right\|_2 \in \mathcal{O}(h_n^2)$ [Pfl10, p.10].



(a) Hierarchical increment spaces. Taken from [Pfl10, p.11].

(b) Selection of subspaces $W_{\boldsymbol{l}}$. Taken from [Pfl10, p.13].

Figure 3.6: Example of Hierarchical increment spaces and subspace selection for $\boldsymbol{l} \leq \mathbf{3}$ and $d = 2$

3.3 Construction of Sparse Grids

To obtain a Sparse Grid we follow the idea presented before: We select only those subspaces of $V_{\boldsymbol{l}}$, whose basis functions have a non-negligible area of support. Expressed with other words, this means basis functions with small support are ignored because they have small contribution to 3.2.8.
So, we obtain the following Sparse Grid space resulting from an optimization problem that was discussed in detail by [BG04]:

Definition 3.3.1: Classical Sparse Grid space

Let $n \geq 1$. We define the *Classical Sparse Grid space* of level n and dimension d as

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

In the definition above $|\cdot|_1$ denotes the l-1 norm. This subspace selection is arranged in figure 3.6b for $n = 2$ and $d = 2$. The left part of the figure displays the a priori selection of subspaces. The subspaces below the dashed line are neglected in the construction. Whereas the right part of figure 3.6b shows the resulting sparse grid $V_3^{(1)}$. In contrast to this is the *full grid* space [GG10, p.3] specified through $\widetilde{V}_n^{(1)} = \bigoplus_{|\boldsymbol{l}|_\infty \leq n} W_{\boldsymbol{l}}$. This space is created if the gray subspaces below the dashed line in figure 3.6b would be considered too. In [BG04, p.26] the authors showed that this selection of subspaces is optimal for the l-2 and l-∞ norm under a cost-benefit standpoint.
This construction diminishes the number of inner grid points to $\mathcal{O}(h_n^{-1} \cdot (\log h_n^{-1})^{d-1})$ [BG04, p.27] which is a significant improvement to $\mathcal{O}(h_n^{-d})$. So far we have only considered the special case of functions f with $f(x) = 0$ for all $x \in \partial\Omega$.
For many applications it is advantageous to extrapolate to the boundaries by modifying the underlying basis functions [Pfl10, p.14]. This technique is usually promising for applications where the accuracy at the boundary is not that important. Another solution to mitigate this boundary-problem is to introduce another level $l = 0$. However, this has the disadvantage of increasing the number of grid points drastically, because most of the points are located on the boundary $\partial\Omega$.

Original idea
The idea of sparse grids dates back to a Russian mathematician Smolyak [BG04, p.43 f.] [Dir00, p.10]. His method is based on a tensor product construction of quadrature formulas $Q_n^{(1)}$ for $n \in \mathbb{N}$. $Q_n^{(1)}$ is a sequence of simple quadrature rule on the subintervals $[\frac{i}{p^n}, \frac{i+1}{p^n}]$ for $0 \leq i \leq p^n - 1$ and $p \geq 2$. Based on these univariate rules d-dimensional quadrature formulas $Q_n^{(d)}$ are constructed. He investigated a class of the following type of quadrature rules:

$$Q_n^{(d)} f = \left(\sum_{i=0}^{n} (Q_i^{(1)} - Q_{i-1}^{(1)}) \otimes Q_{n-i}^{d-1} \right) f$$

where $Q_{-1}^{(1)}$ is the constant 0-function. Figure 3.7 visualizes the resulting grid for the trapezoidal rule. The left image grid has parameters $p = 2, n = 8$, the middle one has $p = 3, n = 5$, whereas the last image has $p = 4, n = 4$.

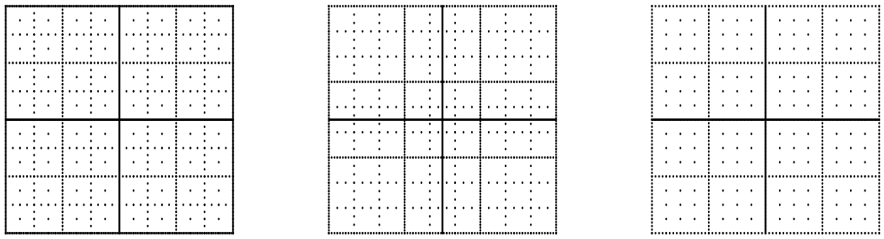
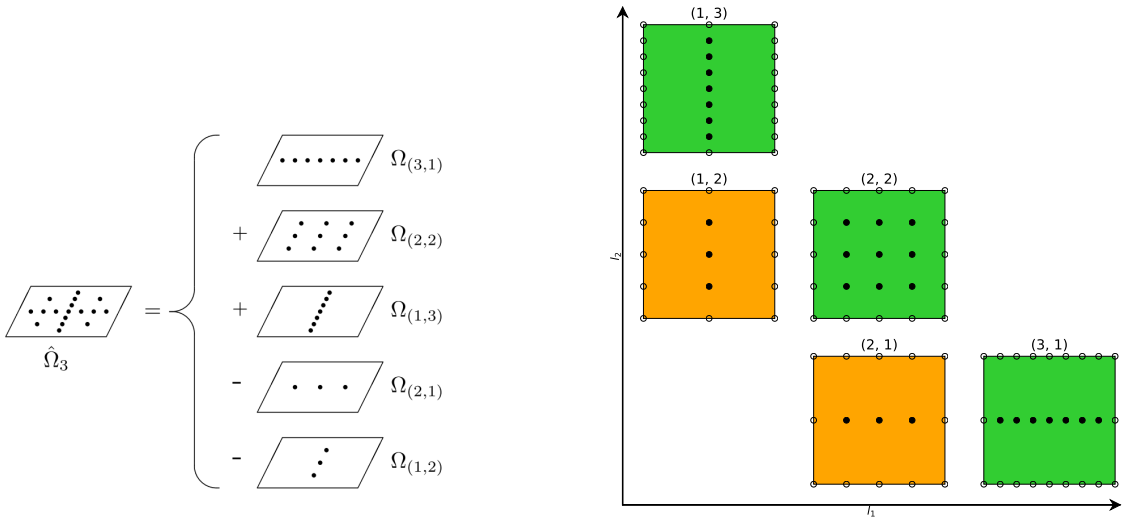


Figure 3.7: Smolyak quadrature with univariate trapezoidal rule.
Taken from [BG04, p.44].

3.4 Combination Technique

The *Combination technique* first introduced by [GSZ92] enables us to circumvent the sometimes difficult recursive data structures required by Sparse Grids. Each grid can be assembled by a linear combination of an-isotropic full grids of different levels. Hence, it is no longer necessary to transcribe already existing algorithms, so that they work on Sparse Grids, after all the algorithm may be executed for each component grid of the combination scheme separately. The results from the different component grids may then be combined using the appropriate coefficient for each grid. This is a perfect application domain for parallelism because the computations can be performed individually on each component grid [OB19].



(a) Relation between the subspace combination and the corresponding Sparse Grid. Taken from [GG10, p.4].

(b) Other visualization of the subspaces from figure 3.8a. Taken from sparseSpACE.

Figure 3.8: Visualizations of the Sparse Grid Combination technique

Let $|\boldsymbol{l}|_1 = n - q$ for $0 \leq q \leq d-1$ and $\boldsymbol{l} \geq 0$. The Combination technique considers grids Ω_l with the former restriction. From now on we will call those grids *component grids*. Leveraging piecewise d-linear functions $\phi_{l,i}$ (nodal basis) we obtain for each component grid the interpolant [Gar13, p.17]:

$$f_{\boldsymbol{l}}(\boldsymbol{x}) = \sum_{i_1=0}^{2^{l_1}} \cdots \sum_{i_d=0}^{2^{l_d}} \alpha_{\boldsymbol{l},\boldsymbol{i}} \phi_{\boldsymbol{l},\boldsymbol{i}}(\boldsymbol{x})$$

Finally, the results of each component grid are combined together by

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

By comparing the above-mentioned formula with figure 3.8b it is evident that some component grids have a negative sign.
Having discussed several grid variants, the following table summarizes the approximation error and corresponding grid points [Pfl10, p.20]:

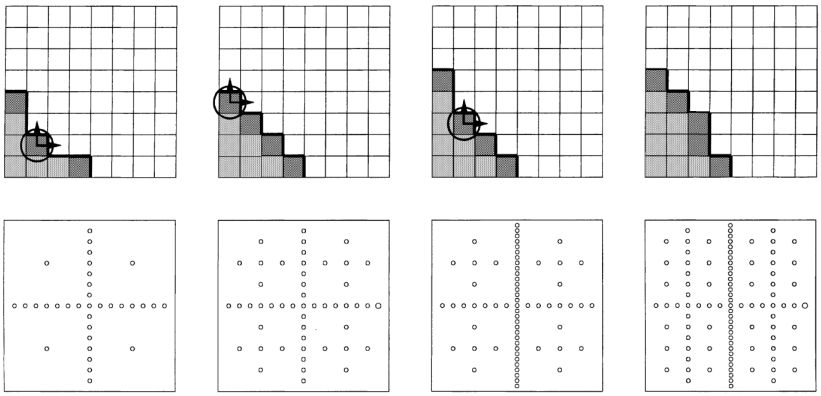
Grid type	Grid points	Approximation error
Sparse Grids	$\mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1})$	$\mathcal{O}(h_n^2(\log h_n^{-1})^{d-1})$
Sparse Grids Combination Technique	$\mathcal{O}(d \cdot (\log h_n^{-1})^{d-1}) \times \mathcal{O}(h_n^{-1})$	$\mathcal{O}(h_n^2(\log h_n^{-1})^{d-1})$
Full Grids	$\mathcal{O}(h_n^{-d})$	$\mathcal{O}(h_n^2)$

Table 3.1: Comparison of grid points and grid approximation error of different grid variants

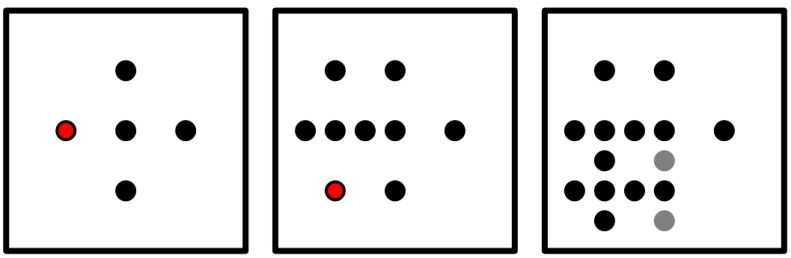
3.5 Adaptive refinement

Not all functions fulfill certain smoothness criteria. Some of them might have highly varying properties. In these cases Sparse Grids might perform not ideal regarding their approximation error and amount of evaluated grid points, due to the a priori choice of relevant subspaces. This is, for example, due to the fact that each dimension and every region is considered equally important.
Hence, adaptive variants of the combination technique have been investigated. The underlying idea is to decide during execution time which sub-spaces or grid points should be refined. This enables the algorithm to treat different regions or dimensions with varying amounts of grid points.
Specifically this means to "spend more effort on rough and less effort on smooth regions" [Dir00, p.20]. An important advantage is that usually fewer points are needed to achieve the same error as using general Sparse Grids.

Dimensional adaptivity
One of the first, who introduced the idea of dimensional adaptivity was [GG03]. In real problems some dimensions might contribute more to the solution than others. The dimension-adaptive approach solves this issue by modifying the current index set in each iteration of the algorithm. More specifically, certain indices are added to the active index set depending on the use of an appropriate error estimator. In each iteration the index with the largest error estimate in the active set is to be refined. The evolution of an exemplary refined grid is depicted in figure 3.9a. In the upper row different states of the index set are depicted. Active indices are coloured darker. The indices with the largest error estimate are encircled. In the second row the according sparse grids are shown using the midpoint rule.



(a) Evolution of the dimensional adaptive algorithm.
Taken from [GG03, p.74].



(b) Evolution of spatial refinement steps.
Taken from [Pfl10, p.21].

Figure 3.9: Comparison of dimensional and spatial adaptivity

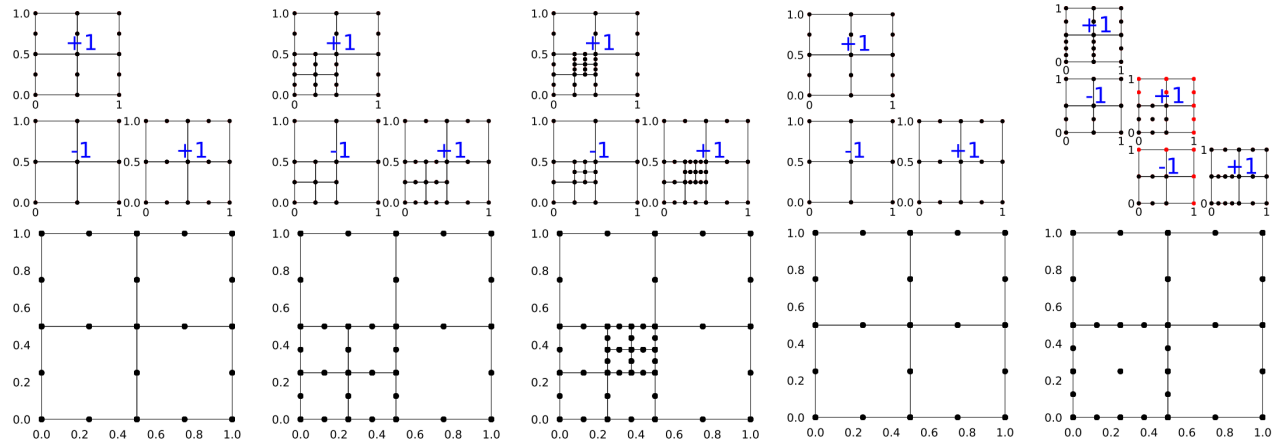
Spatial adaptivity

Whereas the previous technique tries to refine important dimensions, the spatial adaptive refinement tackles functions with varying smoothness conditions in different domains [Pfl10]. This is commonly achieved by refining individual grid points with large error estimates and adding all their hierarchical children. Sometimes it is necessary to add missing hierarchical parents recursively because they are necessary for the calculation of surpluses. An exemplary refinement process is shown in figure 3.9b. The red points are refined by adding all their hierarchical children recursively. In the second refinement iteration, it is necessary to add all hierarchical parents (gray points).

Unfortunately, this procedure cannot be as seamlessly incorporated into the Combination Technique as the previous dimensional adaptive procedure. Due to the nature of subspace selection, it is at first only possible to add full sub-spaces. This stands in contrast to the spatial adaptive idea of refining specific points individually.

However, M. Obersteiner developed the *Split-Extend Scheme* [OB19] that operates on a generalized Combination Technique using block adaptive full grids. The two main operations are, as the name suggests, splitting and extending. The former operates on refinement objects which are subareas of the domain. Based on error estimators a refinement object is split into 2^d subareas. Since this operation has the tendency to converge to a full grid structure another operation called *Extend* has been developed. By adding new sub-spaces (component grids) to the scheme and coarsening of subareas the operation mitigates the development of a full grid structure.

The following figure depicts these two operations side by side:



(a) Starting the inital grid (left), two sequential *Split* operations are executed. Taken from [OB19, p.9].

(b) *Extend* on the lower left quadrant. Taken from [OB19, p.11].

Figure 3.10: The two main operations of the Split-Extend Scheme

A detailed explanation of this algorithm can be found in [OB19].

Additionally, a spatial adaptive algorithm with dimension-wise refinement has been presented in [OB20]. Refinement objects are refined separately in each dimension instead of splitting the domain in 2^d children. After the refinement of each dimension the global scheme for the Combination Technique is constructed. Visually speaking, in every iteration of the algorithm each stripe is refined separately, and afterwards the combination scheme is constructed based on these stripes:

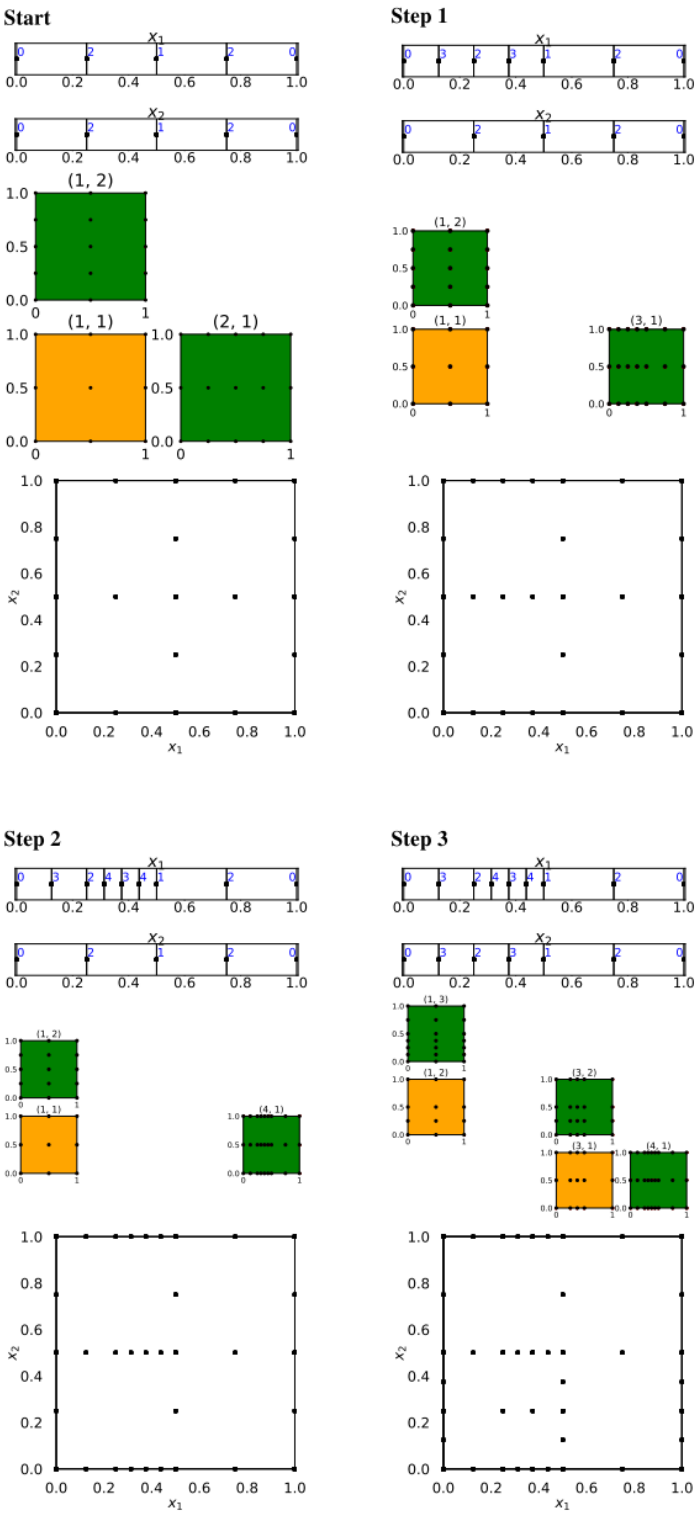


Figure 3.11: Refinement steps of the dimension wise spatial adaptivity.
Taken from [OB20, p.12].

The work of this thesis is based on the dimension-wise refinement of the above explained method. We will execute the extrapolation methods explained in chapter 2 individually on each one-dimensional grid stripe.

4 Implementation

In this chapter we explain the structure of our implementation. The contents are presented in the following order: First, we give an overview of the *sparseSpACE*-Framework[1]. Thereafter, we explain important aspects of the new contributions to the framework. We will address important aspects of our implementation and summarize most of the code structure.

4.1 sparseSpACE framework

The framework supports various types of grids. Each type inherits from

```
class Grid(object):
  def __init__(self, a, b, boundary=True):
    # ...
```

The parameters **a** and **b** are d-dimensional vectors which denote the boundaries of the rectangular integration domain $\Omega = [a_1, b_1] \times \cdots \times [a_d, b_d]$. Whereas the parameter **boundary** is `True` if the grid points on the boundary $\partial\Omega$ are included. The next method returns the weights of the grid:

```
def get_weights(self) -> Sequence[float]:
```

Every grid consists of one-dimensional grids that each inherit from

```
class Grid1d(object):
  def __init__(self, a: float=None, b: float=None, boundary: bool=True):
    #...
```

Here, the parameters **a**, **b**, and **boundary** specify the grid of current dimension. Already implemented grids are for example **TrapezoidalGrid** or **ClenshawCurtisGrid** with their corresponding one-dimensional component grids.
Based on the **Grid** class a generalized class has been implemented

```
class GlobalGrid(Grid):
```

which, among other things, enables the programmer to pass parameters such as **grid_1D** or **grid_levels_1D** to the computation of weights. Some grids like, for example, **GlobalTrapezoidalGrid** or **GlobalSimpsonGrid** already implement this interface and realize the weight computation for corresponding quadrature rules.
Another important component is the class

```
class Function(object):
```

which contains an interface for the numerical testing functions. Several functions from the Genz testing package [Gen87], like **GenzCornerPeak** or **GenzOszillatory** have been implemented.
Additionally there are several operations that can be performed on the grids. One of those operations is of course

```
class Integration(AreaOperation):
  def __init__(self, f: Function, grid: Grid, dim: int,
               reference_solution: Sequence[float] = None):
    # ...
```

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

which encapsulates methods like evaluation of an volume under a function. Most grids use the class

```
class IntegratorArbitraryGridScalarProduct(IntegratorBase):
  def __init__(self, grid):
    # ...
```

by default for integration after the grid weights have been computed. It computes the integral approximation using a scalar product of weights and corresponding function evaluations. After the weights and function values have already been computed this corresponds to the final step. In the one dimensional case:

$$\int_a^b f(x)\,dx \approx w_1 \cdot f(x_1) + \cdots + w_n \cdot f(x_n) = \sum_{i=1}^{n} w_i \cdot f(x_i)$$

The core of the framework encapsulates the logic for the spatially adaptive refinement for the Sparse Grid Combination technique. The class

```
class SpatiallyAdaptivBase(StandardCombi):
  def __init__(self, a: Sequence[float], b: Sequence[float],
               operation: GridOperation, norm: int=np.inf):
    # ...
```

provides the generic interface for all spatially adaptive refinement implementations. The refinement algorithm is initiated with

```
adaptiveCombi.performSpatiallyAdaptiv(lmin, lmax, errorOperator, tol, do_plot=True)
```

where **adaptiveCombi** is an object of an inheriting class of **SpatiallyAdaptivBase** and **lmin**/**lmax** are the corresponding minimal/maximal level of the (truncated) combination technique. Furthermore, an **errorOperator** of type **ErrorCalculator** computes the estimated error inside of an **RefinementContainer**. Finally, **tol** specifies the tolerance at which the refinement should terminate. There are several more options that could be provided. But those seem not so important for our work.
The two classes **SpatiallyAdaptiveExtendScheme** and **SpatiallyAdaptiveSingleDimensions2** are the main implementation classes of different adaptive refinement strategies. The former class is the implementation of the Split-extend scheme [OB19] whereas the latter class implements the spatially adaptive method with dimension-wise refinement [OB20].
More information about the calling conventions of sparseSpACE is compiled in the Jupyter-Notebooks[2] of the frameworks repository.

[2] https://github.com/obersteiner/sparseSpACE/tree/master/ipynb

4.2 Romberg Grid

This section gives an overview of the contribution of this thesis to the *sparseSpACE* framework. The classes that will be presented are important parts of the implementation from the developed theory proposed in the second part of chapter 2. Further information can be found in the Jupyter Notebook called *Tutorial_Extrapolation.ipynb* inside the sparseSpACE framework.

Main interface: sparseSpACE provides an interface that encapsulates the grids and weights. The weight computation of one-dimensional grid stripes is performed with the method **compute_1D_quad_weights**. Thus we implemented the main wrapper class that encapsulates the computation of weights

```
class GlobalRombergGrid(GlobalGrid):
  def __init__(self, a, b, boundary=True, modified_basis=False,
               slice_grouping=SliceGrouping.UNIT,
               slice_version=SliceVersion.ROMBERG_DEFAULT,
               container_version=SliceContainerVersion.ROMBERG_DEFAULT):
    # ...
```

As an example, we revisit the one of section 2.4 using a slightly more refined grid. Let $a = 0, b = 1$ and suppose that the spatial adaptive algorithm needs the extrapolated weights as explained in section 2.4 to decide which area should be refined in the next iteration. The current state of the concerning one-dimensional grid stripe is given by

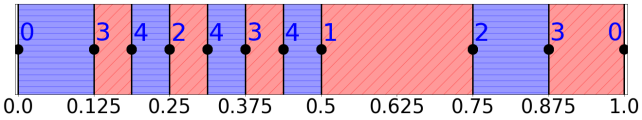
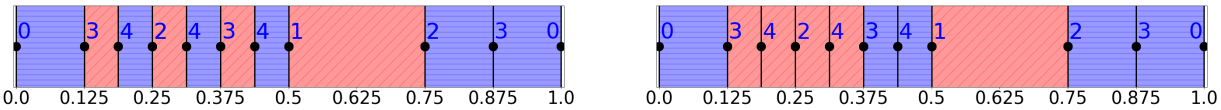


Figure 4.1: One-dimensional grid stripe example

This grid consists of the slices $S_0 = [0, \frac{2}{16}]$, $S_1 = [\frac{2}{16}, \frac{3}{16}]$, $S_2 = [\frac{3}{16}, \frac{4}{16}]$, $S_3 = [\frac{4}{16}, \frac{5}{16}]$, $S_4 = [\frac{5}{16}, \frac{6}{16}]$, $S_5 = [\frac{6}{16}, \frac{7}{16}]$, $S_6 = [\frac{7}{16}, \frac{8}{16}]$, $S_7 = [\frac{8}{16}, \frac{12}{16}]$, $S_8 = [\frac{12}{16}, \frac{14}{16}]$, $S_9 = [\frac{14}{16}, 1]$.
The previous class offers various options that can be set in arbitrary combination with each other:

Slice Grouping: This option determines whether the slices ($S_0, \ldots, S_9$ in the example above) are grouped into bigger containers. By default, each slice is in its own container, thus the containers $C_0, \ldots, C_9$ would be initialized. This version is called **UNIT** grouping and is illustrated in figure 4.1. Each slice is in its own container. A container generally consists of 2^k slices ($k \in \mathbb{N}_0$) which all have the same width. Hence, it is possible to execute the original Romberg's method in each container that has at least two slices. In particular each container with at least two slices has an even number of slices and therefore contains an odd amount of grid points. This enables extrapolation using Simpon's sums as base rules inside of containers with at least two slices. There are two more grouping options: **GROUPED** and **GROUPED_OPTIMIZED**. Using **GROUPED** the grid slices are scanned from left to right: when the previous grid slice has the same width as the current one the previous container is extended. Otherwise a new container is created. In a post processing step each container that does not contain 2^k slices is broken into unit containers (they each consist of a single slice). The **GROUPED_OPTIMIZED** option takes another approach. Suppose the container has $n \neq 2^k$ slices. Instead of breaking the partial full grid into unit containers, a binary decomposition of $n = n_1 + \cdots + n_j$ is executed. In a post processing step the containers are split into smaller ones with

sizes $n_1, \ldots, n_j$ respectively. This technique ensures that each container has the maximum amount of slices under the condition that it can only contain 2^i slices. The next figure illustrates the difference between the two grouping options:



(a) Default slice grouping

(b) Optimized slice grouping

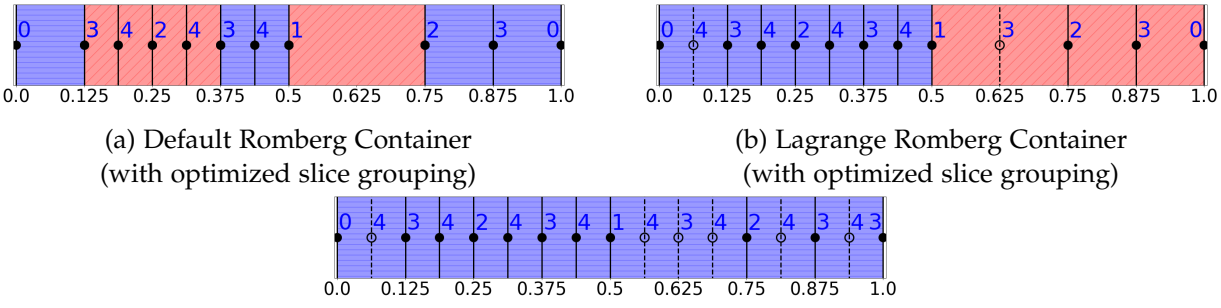
Figure 4.2: Slice grouping options

The alternating colors resemble the groupment of slices into containers. Hence, the left figure has 9 containers, whereas the right figure has only 5. This partitioning process brings some advantages: On the one hand, partial full grid structures are exploited and used for conventional extrapolation methods (e.g. Romberg extrapolation, section 2.3). On the other hand, unit containers are extrapolated using the sliced extrapolation (see section 2.4) or default trapezoidal rule. With this abstraction into containers and slices we provided an flexible interface that can easily be extended in the future using other extrapolation methods.

Slice Version: This option determines the (extrapolation) type for unit slices (these are slices that are alone in a container). Possible are, among others:

1. Sliced extrapolation: based on the support sequence of trapezoidal slices (see section 2.4)
2. Trapezoidal rule without extrapolation
3. Sliced extrapolation with constant subtraction (see subsection 2.5.2)

Container Version: This option allows to change the extrapolation type inside containers with more than one slice. Each container has a full equidistant grid. By default the container executes a weight-based Romberg extrapolation. Another possibility is an interpolatory approach, which is explained in section 2.5.3. The difference between the default Romberg containers (figure 4.3a) and interpolated containers (figure 4.3b) is illustrated in the next figure:



(a) Default Romberg Container (with optimized slice grouping)

(b) Lagrange Romberg Container (with optimized slice grouping)

(c) Lagrange Full Romberg Container (with optimized slice grouping)

Figure 4.3: Container version options

In figure 4.3b there are two interpolated grid points (characterized by the dashed vertical lines and empty points). This approach drastically enlarges full grid structures for extrapolation and reduces the amount of containers. We will explain some algorithmic ideas for the discovery of such

suitable interpolation grid points later on. Figure 4.3c depicts an interpolation of a full grid, which increases the order of the whole grid if the interpolation is sufficiently accurate. Another option is a weight-based extrapolation using Simpson's rule as a base rule, which is explained in subsection 2.5.4.

Main class: The next class handles the initialization of the grid, slices, and containers:

```
class ExtrapolationGrid:
  def __init__(self, slice_grouping: SliceGrouping = SliceGrouping.UNIT,
               slice_version: SliceVersion = SliceVersion.ROMBERG_DEFAULT,
               container_version: SliceContainerVersion
                  = SliceContainerVersion.ROMBERG_DEFAULT,
               force_balanced_refinement_tree=False,
               print_debug=False):
    # ...
```

By calling `extrapolation_grid.set_grid(grid, grid_levels)` on objects of this class, the correct slice and container objects are constructed using the *Factory Pattern*. Finally, the extrapolated weights are computed and returned with `extrapolation_grid.get_weights()`.

Slices: As mentioned above, we provide different types of slices: for example sliced extrapolation or trapezoidal rule. Each slice type inherits from the class

```
class ExtrapolationGridSlice:
    def __init__(self, interval, levels, support_sequence,
                 function: Function = None):
      # ...
```

Containers: Slices are grouped into containers. As explained previously we support multiple container types that all inherit from

```
class ExtrapolationGridSliceContainer:
    def __init__(self, function: Function = None):
      # ...
```

Supported container types are for example: default Romberg extrapolation, interpolatory Romberg extrapolation or extrapolation using Simpson's rule. Unit containers compute their weights based on the slice type. If a container consists of two or more slices, the weights are computed based on the container type.

Grid balancing: This technique is explained in subsection 2.5.1. The next class implements this behaviour using default algorithms

```
class GridBinaryTree:
  def __init__(self, print_debug=False):
    # ...
```

It also incorporates a caching ability that maps grids and their levels to the transformed balanced grid with its balanced levels. Whether this transformation is activated or not can be controlled in the constructor of the class **SpatiallyAdaptiveSingleDimensions2** with the boolean flag **force_balanced_refinement_tree**.

Now that we have introduced the most important interfaces and constructors of our implementation, we would like to present some fundamental algorithms of our implementation.
Firstly, we give an high-level overview of the main algorithm in **ExtrapolationGrid**:

Algorithm 2: Weight computation in ExtrapolationGrid

Input: slice_grouping, slice_version, container_version, force_balanced_refinement_tree, grid, grid_levels
Output: (Extrapolated) weights for the given grid and grid_levels

if *force_balanced_refinement_tree* **then**
  force_balanced_tree_grid()
end

initialized_grid_slices_and_containers()
initialize_weight_dictionary()

foreach *container* ***in*** *containers* **do**
  weights ← container.get_weights()
  update_weight_dictionary(weights)
end

Initialization of slices and containers: After the grid for the weight computation is set the slices as well as the containers are constructed. This is done by parsing the grid from left to right:

```
# Pseudo code
step_width_buffer = None

for i in range(0, len(grid)-1)
  step_width = grid[i+1] - grid[i]
  support_sequence = compute_support_sequence(i, i+1)

  slice_ = create_slice((grid[i], grid[i+1]), (grid_levels[i], grid_levels[i+1]),
                        support_sequence)
  create_containers(step_width, step_width_buffer, slice_)

  step_width_buffer = step_width

grid_init_post_processing()
```

The support sequence of a slice is necessary for the extrapolation process of the sliced trapezoidal rule (see section 2.4.1). Each support sequence always has the tuple of domain boundaries (a, b) as its first element.

```
def compute_support_sequence(self, final_slice_start_index, final_slice_stop_index):
  # Init sequence with indices for level 0 (whole integration domain)
  sequence = [(0, len(self.grid) - 1)] \
             + self.__compute_support_sequence_rec(0, len(self.grid) - 1,
                                                   final_slice_start_index,
                                                   final_slice_stop_index)

  return [(self.grid[element[0]], self.grid[element[1]]) for element in sequence]
```

Other support sequence elements are computed recursively with a sliding window method (start and stop indices). The recursive algorithm operates on grid levels by searching the index of the minimal level within the current window. After a new support tuple has been found the algorithm reduces the active window by halving the old window according to the position of the minimal level and the slice position.

```
def __compute_support_sequence_rec(self, start_index, stop_index,
                                   final_slice_start_index,
                                   final_slice_stop_index):
    if start_index >= stop_index:
        return []

    # Slice of the grid levels without the current level
    grid_levels_slice = self.grid_levels[(start_index + 1):stop_index]

    if len(grid_levels_slice) == 0:
        return []

    # Start or stop index of the next slice in the sequence
    new_boundary_index = (start_index + 1) \
                         + grid_levels_slice.index(min(grid_levels_slice))

    # Determine the boundary indices of the new slice in the sequence
    if new_boundary_index <= final_slice_start_index:
        start_index = new_boundary_index
        # stop_index does not change
    else:
        # start_index does not_change
        stop_index = new_boundary_index

    return [(start_index, stop_index)] \
        + self.__compute_support_sequence_rec(start_index, stop_index,
                                              final_slice_start_index,
                                              final_slice_stop_index)
```

If slice grouping is enabled, the slices are grouped into bigger containers. Depending on the width of the previous slice either a new container is constructed or the slice is appended to the previous container. If interpolation is enabled, it is possible that slices with different widths are grouped in one container, if their widths are below a certain threshold delta. Then the missing slices (those contain the interpolated grid points) are inserted in the post processing step.
Finally, in the post processing step those containers are divided if there exists no $k \in \mathbb{N}_0$ so that the slice count is 2^k. Figure 4.2 displays the default and optimized grouping.

Extrapolation: Another important aspect of our implementation is the extrapolation inside slices and containers. We will address only the core part of this implementation because the complete code is quite large. The next code excerpt shows the computation of the extrapolated weights from a slice using the support sequence. These partial weights are stored in a dictionary using grid points as keys. With this data structure it is convenient to merge and accumulate partial weights of different slices whose contributions might overlap with other slices.

The $c_{m,j}$ coefficients for the extrapolation are generated from a separate factory. This makes it convenient to change extrapolation methods easily.

```
# Returns a dictionary that maps grid points to a list of their extrapolated weights
def get_final_weights(self):
    # The 1st element of the support sequence contains the domain boundaries
    (a, b) = self.support_sequence[0]

    # Generate the extrapolation coefficients
    coefficient_factory = self.coefficient_factory.get(a, b, self.support_sequence)

    # Dictionary that maps grid points to their extrapolated weights
    weight_dictionary = defaultdict(list)

    for level in range(self.max_level + 1):  # 0 <= i <= max_level
        point_weight_pairs = self.get_support_points_with_their_weights(level)

        assert len(point_weight_pairs) == 2
        (left_point, left_weight) = point_weight_pairs[0]
        (right_point, right_weight) = point_weight_pairs[1]

        coefficient = coefficient_factory.get_coefficient(self.max_level, level)
        left_weight = coefficient * left_weight
        right_weight = coefficient * right_weight

        weight_dictionary[left_point].append(left_weight)
        weight_dictionary[right_point].append(right_weight)

    # Update dictionary of extrapolated weights
    self.extrapolated_weights_dict = weight_dictionary

    return weight_dictionary
```

As the name already suggests `self.get_support_points_with_their_weights(level)` returns the (**level**)-th support sequence element and the weights of the sliced trapezoidal rule (which have not been extrapolated yet).
The above code shows the weight extrapolation for one unit slice. When a container consists of more than one slice, the weights are computed using a classic full grid extrapolation technique, for example the original Romberg method. These partial weights of the non-unit containers are combined with the partial weights from all other unit slices to obtain the final total weights.

Subtraction of extrapolation constants: As explained in subsection 2.5.2, we use Lagrange interpolation for the approximation of the derivatives. Using the SymPy framework[3] the generic Lagrange basis is constructed and then symbolicly differentiated. These results are then used to approximate the extrapolation constants that will be subtracted. Unfortunately, the symbolic differentiation with SymPy needs a lot of computing time. Therefore, the method cannot be used in this form because many derivatives have to be determined.

[3] `https://www.sympy.org/`

Extrapolation with interpolated grid points: In subsection 2.5.3 we explained an interpolatory approach to increase the containers and the partial full grid structures for extrapolation. Figure 4.3 illustrates the difference between Romberg containers and interpolated Romberg containers. As explained above, the idea is to group slices with similar (but not the same) width into containers and replace some slices with smaller ones to obtain a full grid structure within the container. But how are these interpolatory slices determined? The goal is to interpolate as little as possible and consider only promising grid points that should be interpolated. Otherwise we would obtain an interpolated full grid on the whole integration domain.
An additional important aspect of this approach is the determination of support points for the interpolation. One possibility to determine these points is an geometrical approach. Starting from the interpolation point we select support points beginning with the geometrically closest ones. Whenever a new support point is found the condition of the interpolation is evaluated using the sum of the interpolated weights. The following pseudo-code illustrates the algorithmic idea:

Algorithm 3: Determine interpolation support points geometrically

```
Input: interp_point, max_points, adaptive
Output: List of geometrically closest support points

grid ← get_noninterpolated_grid()
(left, right) ← get_closest_neighbours(interp_point, grid)

support_points ← []

while len(support_points) < max_points and (0 ≤ left or right ≤ len(grid) - 1) do
    if 0 ≤ left then
        add_new_support_point(left, support_points)
    end
    if right ≤ len(grid) - 1 then
        add_new_support_point(right, support_points)
    end

    if is_ill_conditioned(interp_point, support_points) then
        revert_support_points(left, right, support_points)
    end
end
```

We presented a selection of some important algorithms of our implementation. For further information visit the Extrapolation classes in the sparseSpACE framework [4]. The next chapter finally presents the numerical results of our newly implemented methods.

[4] `https://github.com/obersteiner/sparseSpACE`

5 Numerical Results

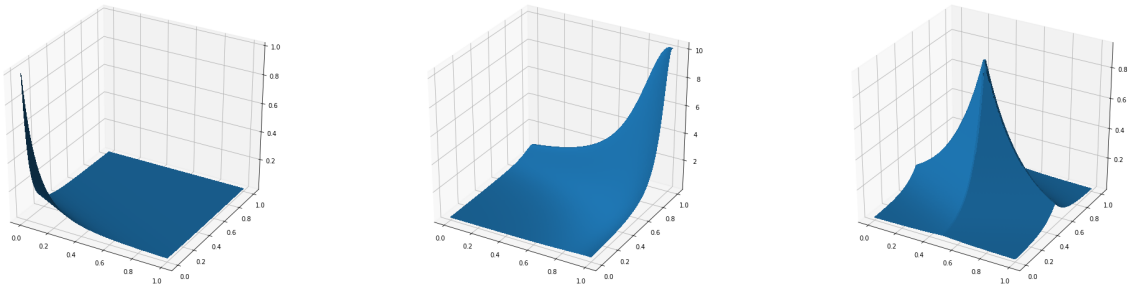
In this penultimate chapter we present the results from various versions of our implementation. Subsequently, we are going to compare these results with each other and reference solutions, which have already existed before. Finally, we will highlight important differences between the implemented methods. In the final chapter 6 further possible improvements of our implementation are discussed on the basis of these results.

5.1 Test functions

Let $\boldsymbol{x} = (x_1, \ldots, x_d)$, $d \geq 1$ and $s \in \{\text{corner}, \text{prod}, \text{cont}, \text{gauss}, \text{oscil}, \text{discont}, \text{expvar}\}$. Several numerical experiments have been conducted on the real value functions $f_s : \mathbb{R}^d \to \mathbb{R}$.
The first six functions $f_{\text{corner}}, f_{\text{prod}}, f_{\text{cont}}, f_{\text{gauss}}, f_{\text{oscil}}, f_{\text{discont}}$ are taken from the Genz test package [Gen87] and have been slightly adapted [OB19]. Whereas the last function f_{expvar} is taken from [GG98]. In the following let $\exp : \mathbb{R} \to \mathbb{R}$ denote $\exp(x) = e^x$.
The first function f_{corner} has one parameter $a \in \mathbb{R}^d$ which determines the growth of the function towards the corner.

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

The graph of f_{corner} is depicted in figure 5.1a for $a = \binom{4}{8}$:



(a) GenzCornerPeak f_{corner} (b) GenzProductPeak f_{prod} (c) GenzContinuous f_{cont}

Figure 5.1: GenzCornerPeak, GenzProductPeak and GenzContinuous for $d = 2$

f_{prod} has the parameters $a \in \mathbb{R}^d$ and $u \in \mathbb{R}^d$ that control the growth and position of the peak. It is arranged in figure 5.1b with $a = \binom{4}{8}$ and $u = \binom{0.99}{0.99}$:

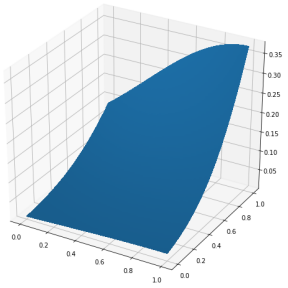
$$f_{\text{prod}}(\boldsymbol{x}) = \frac{10^{-d}}{\prod_{i=1}^{d} a_i^{-2} + (x_i - u_i)^2}$$

f_{cont} has parameters $a \in \mathbb{R}^d$ and $u \in \mathbb{R}^d$ and is displayed in figure 5.1c for $a = \binom{4}{8}$ and $u = \binom{0.5}{0.5}$

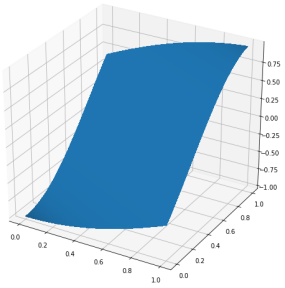
$$f_{\text{cont}}(\boldsymbol{x}) = \exp\left(-\sum_{i=1}^{d} a_i \cdot |x_i - u_i|\right)$$

f_{gauss} is controlled by $a \in \mathbb{R}^d$ and $u \in \mathbb{R}^d$ and is plotted in figure 5.2a for $a = \binom{1}{2}$ and $u = \binom{0.99}{0.99}$.

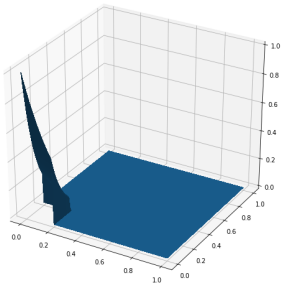
$$f_{\text{gauss}}(\boldsymbol{x}) = \exp\left(-\sum_{i=1}^{d} a_i \cdot (x_i - u_i)^2\right)$$



(a) GenzGaussian f_{gauss}



(b) GenzOscillatory f_{oscil}



(c) GenzDiscontinuous f_{discont}

Figure 5.2: GenzGaussian, GenzOscillatory and GenzDiscontinuous for $d = 2$

Now we are presenting some more difficult functions. Firstly, a highly oscillating function, for suitable parameters $a \in \mathbb{R}^d$ and $u \in \mathbb{R}$:

$$f_{\text{oscil}}(\boldsymbol{x}) = \cos\left(2\pi \cdot u + \sum_{i=1}^{d} i \cdot x_i\right)$$

This function is arranged in figure 5.2b for paramters $a = \binom{1}{2}$ and $u = \frac{1}{2}$.
Furthermore, we have investigated our implementation using a discontinuous function with $a \in \mathbb{R}^d$ and $u \in \mathbb{R}^d$, which is displayed for $a = \binom{4}{8}$ in figure 5.2c.

$$f_{\text{discont}}(\boldsymbol{x}) = \begin{cases} 0, & \text{for } \forall 1 \leq i \leq d : x_i \geq 0.2 \\ \exp\left(-\sum_{i=1}^{d} a_i \cdot x_i\right), & \text{otherwise} \end{cases}$$

The function f_{discont} has zero value for all $x \leq u$, where $x \in \mathbb{R}^d$ and is exponentially growing otherwise. Lastly, we present the function f_{expvar}:

$$f_{\text{expvar}}(\boldsymbol{x}) = \left(1 + \frac{1}{d}\right)^d \cdot \prod_{i=1}^{d} x_i^{\frac{1}{d}}$$

With f_{expvar} we want to evaluate how the adaptive refinement performs when the spatial adaptivity contributes only a little [OB20, p.18].

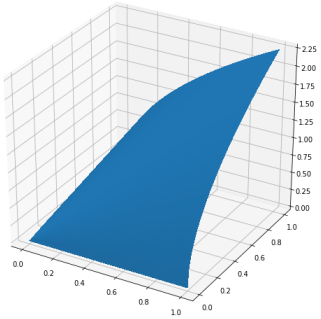
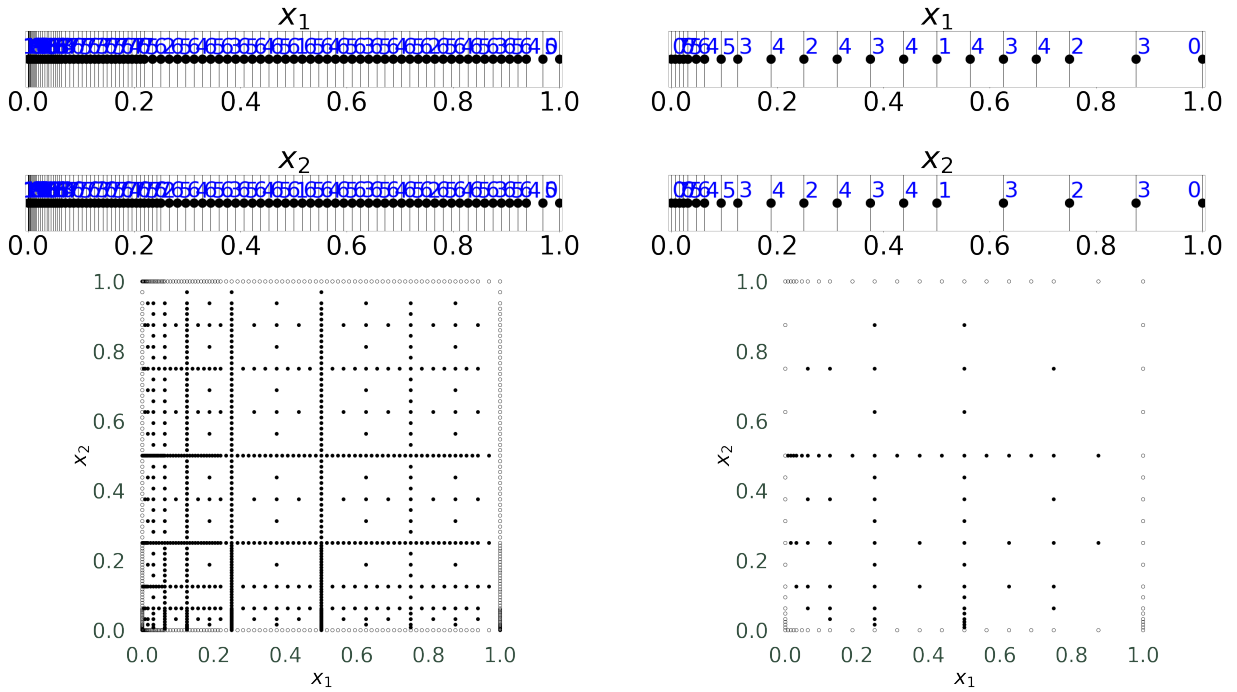


Figure 5.3: ExpVar for $d = 2$

5.2 Results

In this subsection we are presenting the numerical results of our implemented methods. To begin with, some two dimensional grids will be illustrated. Afterwards some convergence diagrams will be compared and discussed.

Resulting Sparse Grids: The following figures compare different Sparse Grids with and without extrapolation for different 2-dimensional functions using the same parameters as in the plots in section 5.1. These results are, among others, compared by the number of distinct function evaluations needed to achieve a certain tolerance threshold. In the following, we will abreviate "distinct function evaluations" with "function evaluation" and "distinct grid points" with "points".

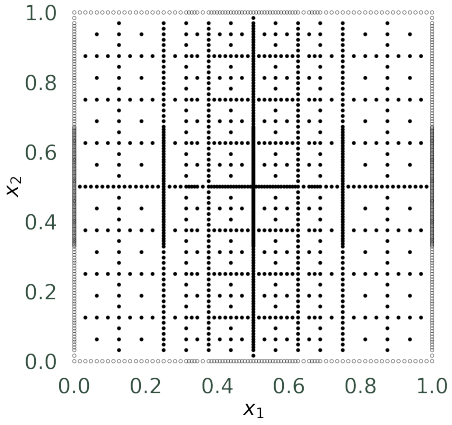


(a) Trapezoidal Grid (without rebalancing)

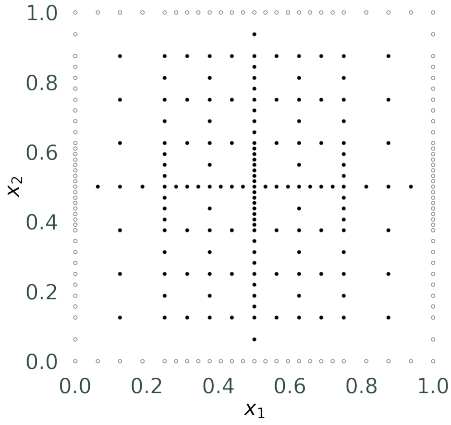
(b) Extrapolation Grid (Grouped Optimized, Romberg, Default Romberg)

Figure 5.4: Comparison of Sparse Grids and their stripes for $d = 2$, $tol = 10^{-4}$, and $f_{\text{exp\_var}}$

The above-mentioned example displays a big advantage of the extrapolation grid: only 148 distinct function evaluations are required to achieve the same approximation error as the trapezoidal grid with 1102 distinct function evaluations. Since significantly less points have to be evaluated with extrapolation this method is also significantly faster. Suprisingly, the trapezoidal grid with rebalancing requires as well 933 points which is much more than when using extrapolation. Additionally, the extrapolated grid requires only 30 refinements whereas the trapezoidal grids needs 178 refinements. The following figure illustrates the comparison of a trapezoidal grid and an extrapolated grid with optimized grouping of Romberg slices and execution of default Romberg inside containers. In this depiction you can easily see that the grid is refined towards the center of the integration domain. This area requires more points, because the function f_{cont} has its peak there, as one can see in figure 5.1c. Again the extrapolation grid requires only 273 points to fall below an approximation error of 10^{-4}. This number of points is significantly smaller than that of a trapezoidal grid which needs 1621 points. As a consequence only 52 refinements are required instead of 244.



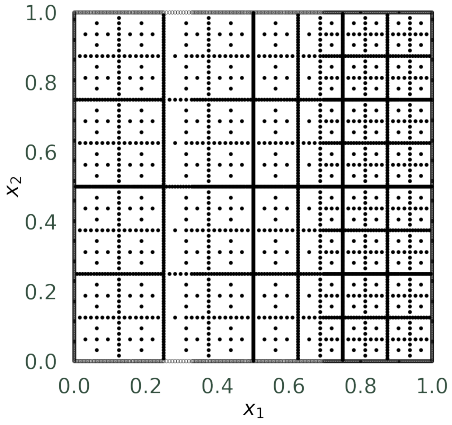
(a) Trapezoidal Grid
(without rebalancing)



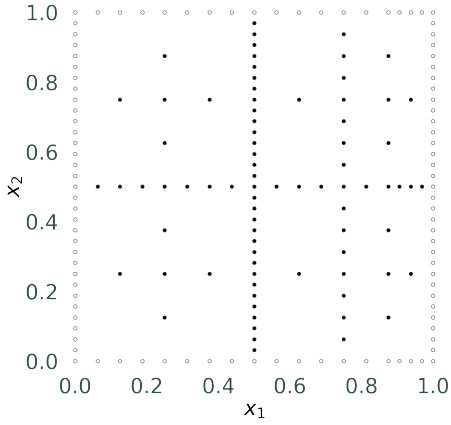
(b) Extrapolation Grid
(Grouped Optimized, Romberg, Default Romberg)

Figure 5.5: Comparison of Sparse Grids for f_{cont}, $d = 2$ and $tol = 10^{-4}$

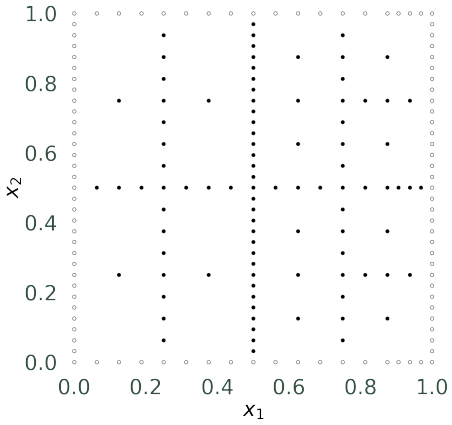
The next figure displays multiple variants of the extrapolation grid and a trapezoidal grid for f_{gauss}:



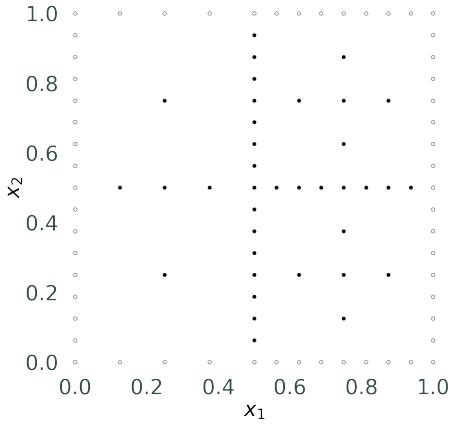
(a) Trapezoidal Grid
(without rebalancing)



(b) Extrapolation Grid
(Unit, Romberg, Default Romberg)



(c) Extrapolation Grid
(Unit, Romberg, Default Romberg, Balanced)



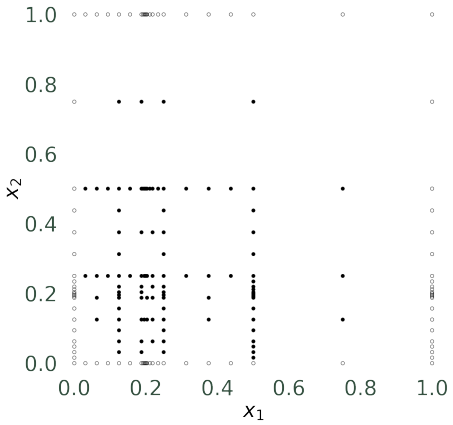
(d) Extrapolation Grid
(Unit, Romberg, Lagrange Romberg)

Figure 5.6: Comparison of Sparse Grids for f_{gauss}, $d = 2$ and $tol = 10^{-6}$

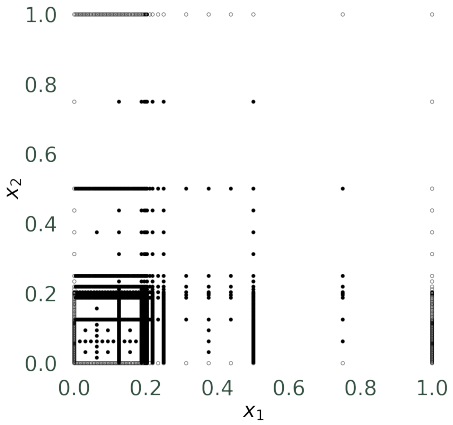
In all four grids a refinement towards the right boundary can be recognized. This behaviour corre-

sponds to the plot in figure 5.2a. The difference between trapezoidal grid and extrapolation grid is even more significant. We terminated the refinement of the trapezoidal grid manually after 5219 points were used resulting in an approximation error of $2 \cdot 10^{-6}$ because the approximation error decreased only slightly. However, using the extrapolation grid with unit grouping, Romberg slices, and default Romberg containers required only 200 points to achieve a tolerance of $tol = 10^{-6}$. An interesting observation is that the number of 42 refinements using the extrapolation grid are higher than the 18 refinements when using the trapezoidal grid. If balancing is enabled for the extrapolation grid, we obtain only slightly more points (203) whereas the number of refinements remains at 42. This means, balancing has no significant effect in this example.
The best results in terms of number of points are delivered by the extrapolation grid that uses Lagrange interpolation for suitable points. This method requires only 93 points and 20 refinements to achieve the same tolerance of $tol = 10^{-6}$.

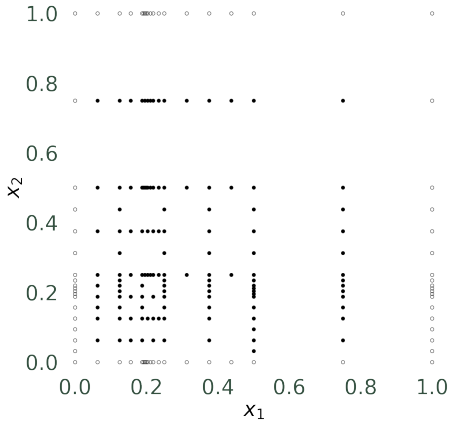
Finally, we present some Sparse Grids that are produced for f_{discont}:



(a) Trapezoidal Grid
(without rebalancing)



(b) Extrapolation Grid
(Grouped Optimized, Romberg, Default Romberg)



(c) Extrapolation Grid
(Grouped Optimized, Romberg, Default Romberg,
Balanced)

Figure 5.7: Comparison of Sparse Grids for f_{discont}, $d = 2$ and $tol = 10^{-3}$

All three figures show that the grid is refined towards the lower left corner where the function has discontinuities (see figure 5.2c). As follows, there are more points required than in the other more constant parts of the integration domain.

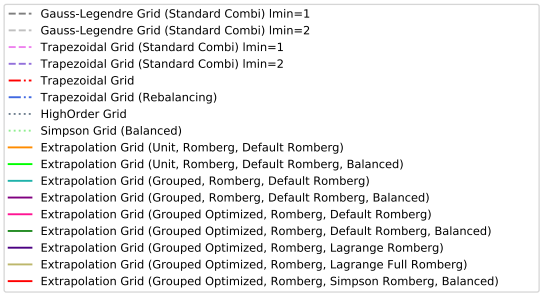
The trapezoidal grid requires 192 points to obtain the tolerance 10^{-3}. However, the extrapolation grid without rebalancing has its difficulties due to the discontinuities. Which is why we terminated the refinement after 1612 points manually and obtained only an approximation error of about $2 \cdot 10^{-1}$. Enabling grid balancing solves this problem in some extend. The algorithm then needs slightly more points (197) to achieve a tolerance of 10^{-3}. Despite requiring some more points the extrapolation with balancing has only 28 refinements whereas the trapezoidal grid needs 36 refinements. Thus, the extrapolation grid with balancing takes significantly less time than the trapezoidal grid.
In summary, we conclude that for all above-mentioned examples there exist an extrapolation grid that requires less points or less refinement steps than the trapezoidal grid. Among other things, this leads to shorter running times. However, it should be noted that this cannot be generalized for all functions and all variants of the extrapolation grid.

Convergence diagrams: Now the numerical results for the above mentioned functions are presented. Firstly, we will provide an overview of all compared variants. These variants are classified into the following groups:

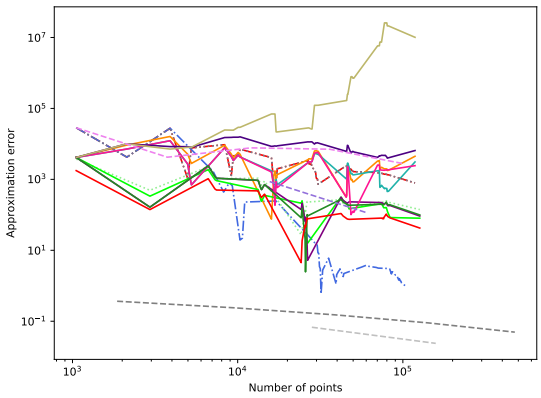
1. **Standard combination schemes:** These are already implemented grids in the sparseSpACE framework. Since those grids are executed with the standard combination scheme they don't use spatial adaptivity. The parameter *lmin* determines the minimum level of the combination technique.
   a) Gauss-Legendre Grid (Standard Combi) with lmin = 1 or lmin = 2.
   b) Trapezoidal Grid (Standard Combi) with lmin = 1 or lmin = 2.
2. **Spatial adaptivity with existing grids:** The following grids are used in combination with spatial adaptivity and dimension-wise refinement.
   a) Trapezoidal Grid with rebalancing enabled or disabled.
   b) High Order Grid.
   c) Simpson Grid (Balanced): The results of this grid have improved after we enabled our new grid balancing operation.

   Here rebalancing is an operation that modifies the grid levels to balance the refinement tree and even prevents from too much refinements of subtrees. The Balanced option however refers to the grid balancing from subsection 2.5.1 where each node has either zero or two children.
3. **Spatial adaptivity with new extrapolated grids:** In this group we again used spatial adaptivity with dimension-wise refinement. The evaluated grids are of the following form: Extrapolation Grid (<Grouping>, <Slice version>, <Container version>, <Balancing>). The options are defined, like explained in section 4.2.
   a) <Grouping>: e.g. Unit, Grouped or Grouped optimized.
   b) <Slice version>: e.g. Romberg, Trapezoid.
   c) <Container version>: e.g. Default Romberg, Lagrange Romberg, Simpson Romberg (subection 2.5.4), Lagrange Full Romberg (where a full grid is interpolated).
   d) <Balancing>: If enabled, each node in the refinement tree is forced to zero or two children.

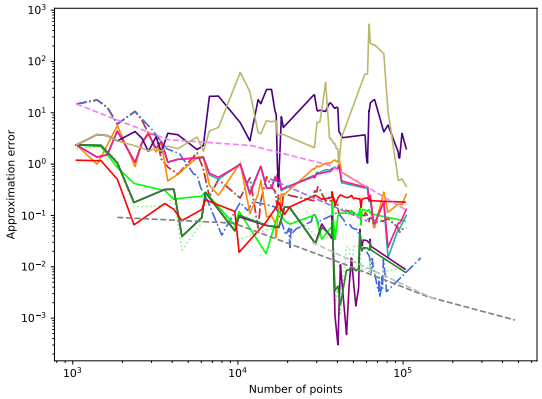
Because the following diagrams might be a little confusing due to the large number of selected variants, different variants are considered in isolation afterwards:



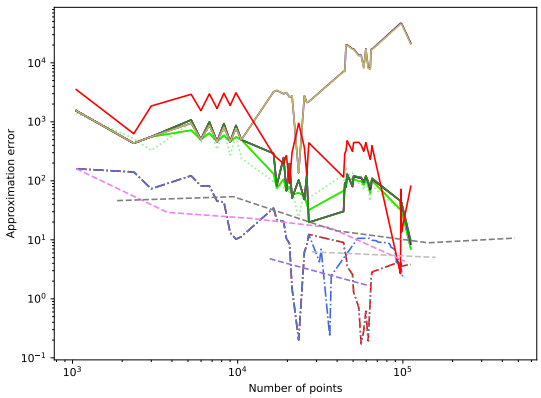
(a) Algorithm legend



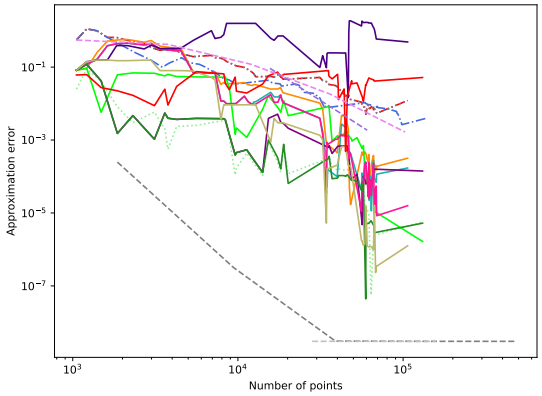
(b) GenzCornerPeak f_{corner}



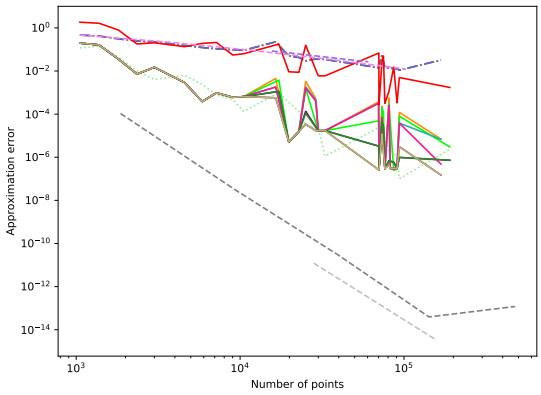
(c) GenzProductPeak f_{prod}



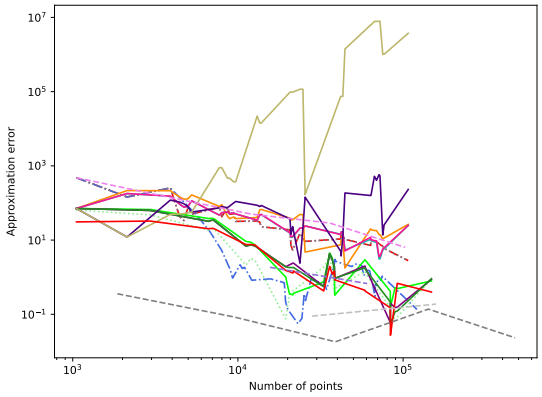
(d) GenzContinuous f_{cont}



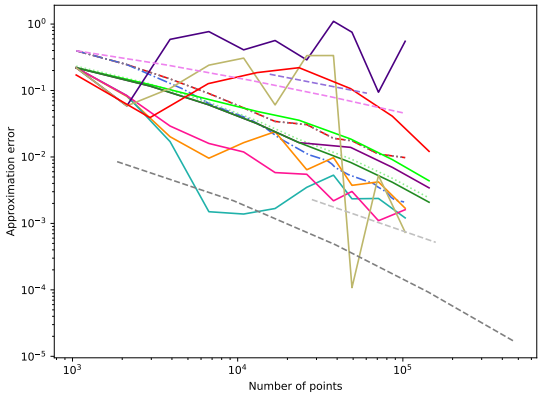
(e) GenzGaussian f_{gauss}



(f) GenzOscillatory f_{oscil}



(g) GenzDiscontinuous f_{discont}



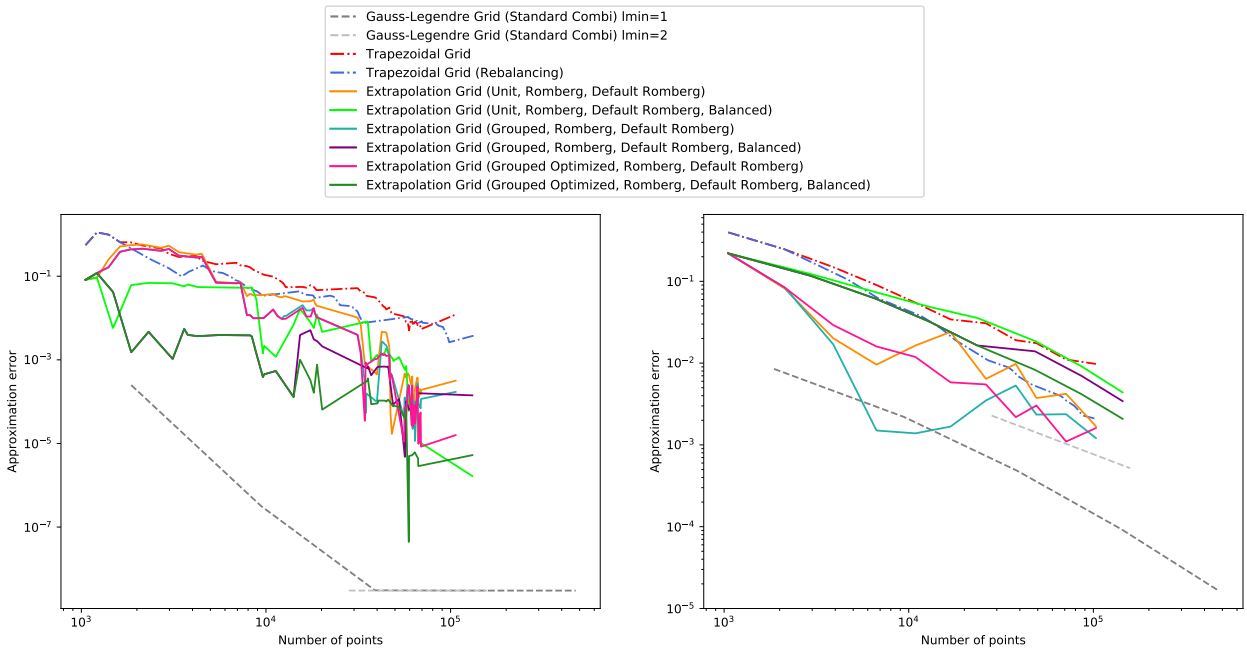
(h) ExpVar f_{expvar}

Figure 5.8: Overview of the approximation error for various grids

These diagrams were generated with 5-dimensional functions and the following parameters: For $f_{\text{corner}}, f_{\text{prod}}, f_{\text{cont}}$ and f_{discont} we have set the parameter $a = (a_i)_{1 \le i \le 5}$ with $a_i = 4 \cdot i$. Whereas we used $a = (a_i)_{1 \le i \le 5}$ with $a_i = i$ for f_{gauss} and f_{oscil}.
At first, it should be noted that most of the adaptive variants lie between the Standard Combination technique of the Trapezoidal rule and the Gauß-Legendre rule. The only significant exception is the GenzContinuous function. Here all algorithms that use extrapolation are worse than the previously implemented methods. The reason for this is that the GenzContinuous function is not continuously differentiable.
Another interesting observation is that the sliced extrapolation with unit grouping sometimes even outperforms the other extrapolation methods.

5.3 Comparison

Grid balancing: Although balancing potentially increases the number of grid points it mostly had a positive consequence on our tests. The following figures compare extrapolation with and without balancing for the function f_{gauss}.

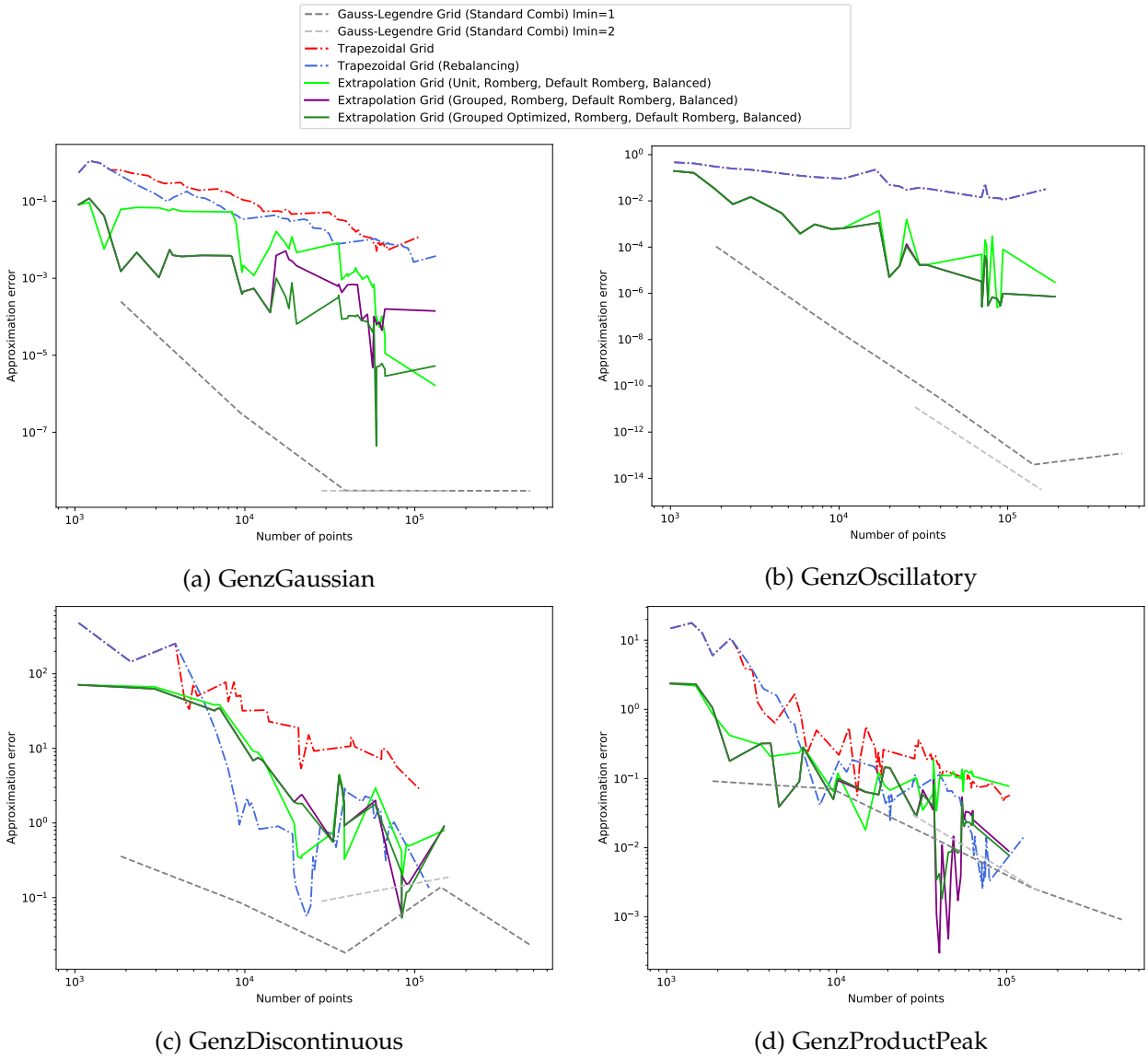


(a) f_{gauss} (b) f_{expvar}

Figure 5.9: Impact of unit extrapolation with and without balancing

No matter which slice grouping is used for f_{gauss}, the balanced extrapolation always outperforms the extrapolation without balancing. Most of the other functions behave similarly, i.e., if balancing is enabled, the error is reduced mostly at least equally or faster. However, the function f_{expvar} is an exception. Here the balancing seems to have a small negative effect on the extrapolation, which is slightly delaying the convergence. Whereas the trapezoidal grid with rebalancing outperforms the extrapolation grids with balancing at about 10^4 points. One conceivable cause for this behaviour could be the small benefit from spatial adaptivity for f_{expvar}. It should be noted that the approximation error is only slightly worse but mostly outperforms the trapezoidal rule.

Slice grouping: An important question one might ask is: Which slice grouping version performs the best? As the extrapolation grid performs better with balancing we will only compare the slice grouping with balancing enabled. But it should be noted that the extrapolation without balancing shows no significant differences between the grouping version. Nevertheless, the grouped optimized version mostly performs slightly better than the other versions (but not significantly).
The following figure compares unit, grouped, and grouped optimized with the trapezoidal rules:



(a) GenzGaussian

(b) GenzOscillatory

(c) GenzDiscontinuous

(d) GenzProductPeak

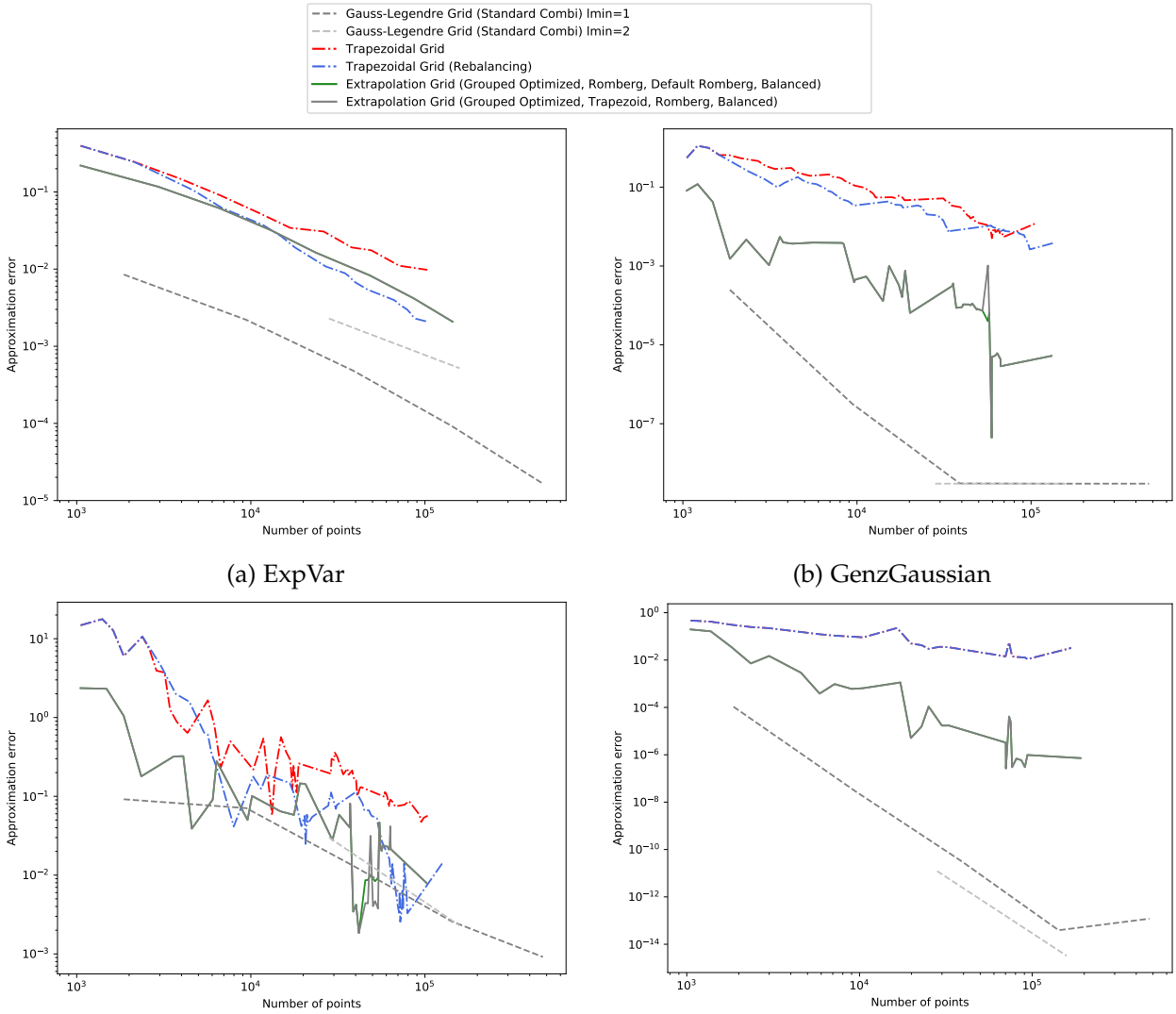
Figure 5.10: Comparison of slice grouping versions

Overall there are no significant differences of the three versions in the convergence diagrams. All three methods achieve roughly a similar order in all tests. Having said that, in most of the cases the grouped optimized version performs slightly better than the others.

Sliced extrapolation: Another important result concerns the sliced extrapolation (explained in subsection 2.4.3). We conducted numerical experiments with the extrapolation grid using unit grouping and sliced extrapolation. First it should be noted that sliced extrapolation is exact on those full grids where the original Romberg method is exact. This behaviour has been investigated using

many automatically generated polynomials. Therefore, from a numerical point of view, it seems that sliced extrapolation is equivalent to the original Romberg method on full grids. But it should be noted that this result has not been formally proven.
We numerically investigated the above-mentioned issue closer. We compared the approximation errors for polynomial and trigonometric functions with grids that were symmetrical to each other. For example $G_1 = [0, 0.5, 0.75, 1]$ and $G_1' = [0, 0.25, 0.5, 1]$, or $G_2 = [0, 0.5, 0.625, 0.75, 0.875, 1]$ and $G_2' = [0, 0.125, 0.25, 0.375, 0.5, 1]$ with the obvious levels. It appears that G_1 and G_1' have the same approximation error except that they have an inverted sign. This implies that the absolute error is equal. The same observation has been made with G_2 and G_2'.
We also investigated other unbalanced grids than G_1 and G_1', such as $G_3 = [0, 0.5, 0.625, 0.75, 1]$ and $G_3' = [0, 0.125, 0.25, 0.5, 1]$. Those grids mostly had not the same absolute error.

Impact of sliced extrapolation: The following figures compare the two slice versions with optimized slice grouping and grid balancing enabled.



(a) ExpVar

(b) GenzGaussian

(c) GenzProductPeak

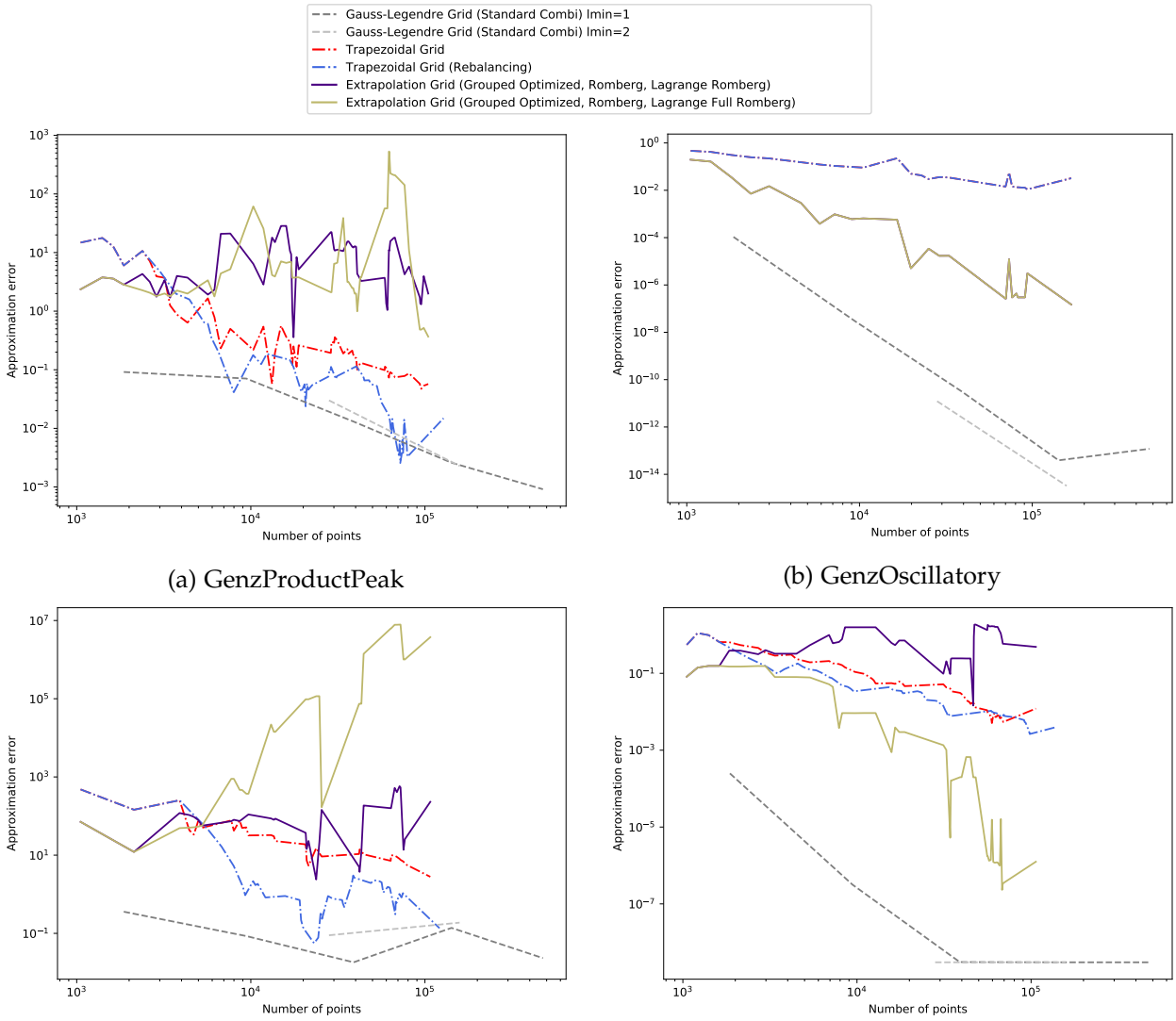
(d) GenzOscillatory

Figure 5.11: Impact of slice versions: Trapzoid vs. sliced Romberg

Apparently the containers with more than one slice contribute more to the result as the containers

with unit slices. However, figure 5.8 shows that sliced extrapolation with unit slicing and balancing performs very well in comparison to the other methods.

Interpolation containers: Now we want to investigate the impact of interpolating grid points for the extrapolation. First we compare the results of full grid interpolation (where all missing grid points of the closest full grid are interpolated) with the results of an adaptive grid interpolation (where only promising grid points are interpolated).



(a) GenzProductPeak

(b) GenzOscillatory

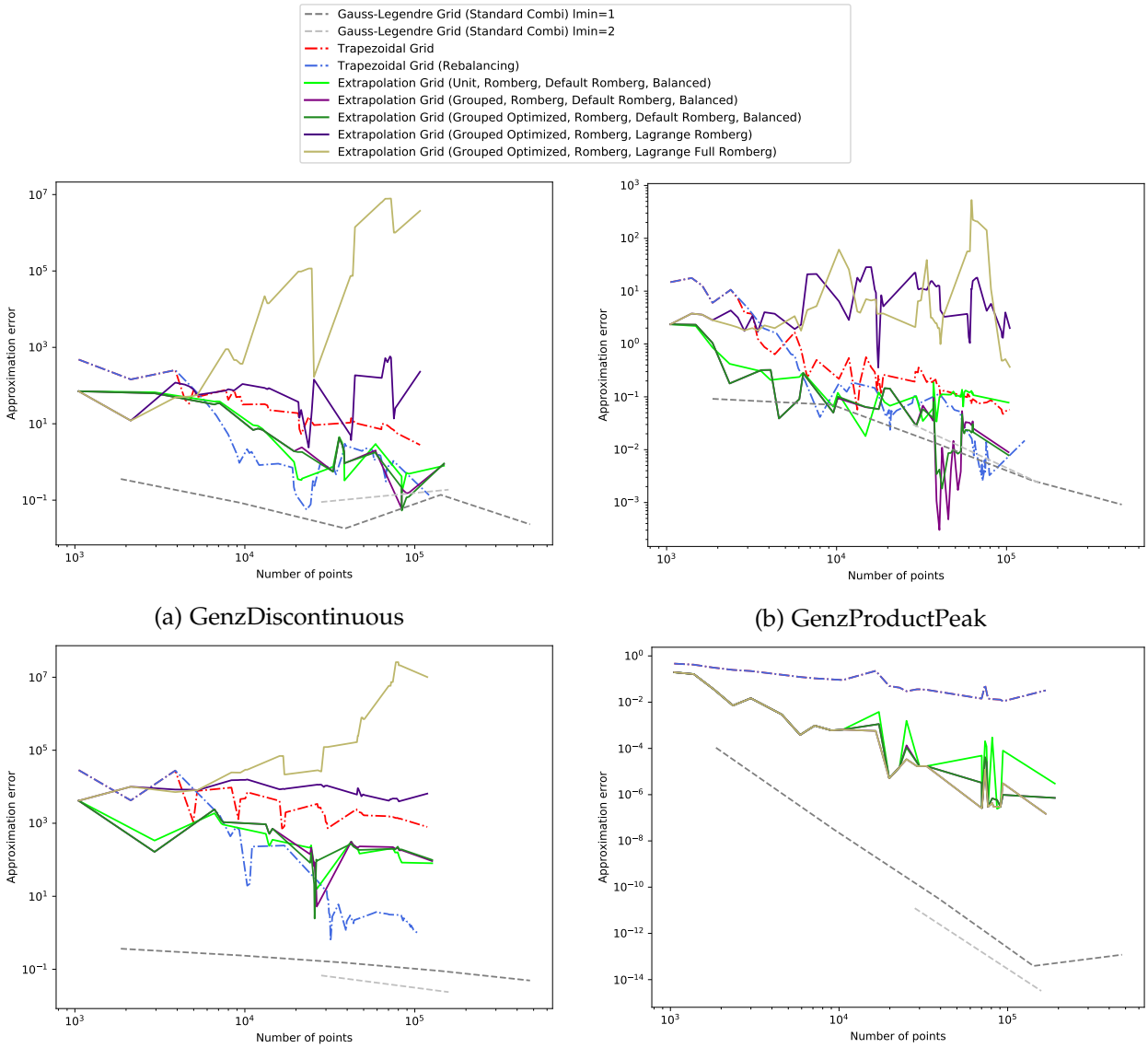
(c) GenzDiscontinuous

(d) GenzGaussian

Figure 5.12: Examples: Adaptive interpolating extrapolation vs. full grid interpolating extrapolation

As expected the full grid interpolation performs better in those cases where the interpolation is sufficiently accurate. This is due to the fact that it achieves a higher order on the whole grid in contrast to the adaptive interpolation that achieves only locally higher order. In the other cases both variants show similar characteristics.
An exception is the function f_{discont}: Due to its discontinuities the interpolation is not sufficient for the extrapolation. Hence, the full grid interpolation performs not as good as the adaptive extrapolation. Surprisingly f_{prod} is not resolved well either. Generally it can be said the adaptive interpolation is

the more appropriate choice under a runtime and computation complexity perspective. Especially for functions that require excessive refinement in one area and few points in others the interpolation to a full grid is not optimal.
Now we want to compare the interpolation approach with other extrapolation methods.



(a) GenzDiscontinuous

(b) GenzProductPeak

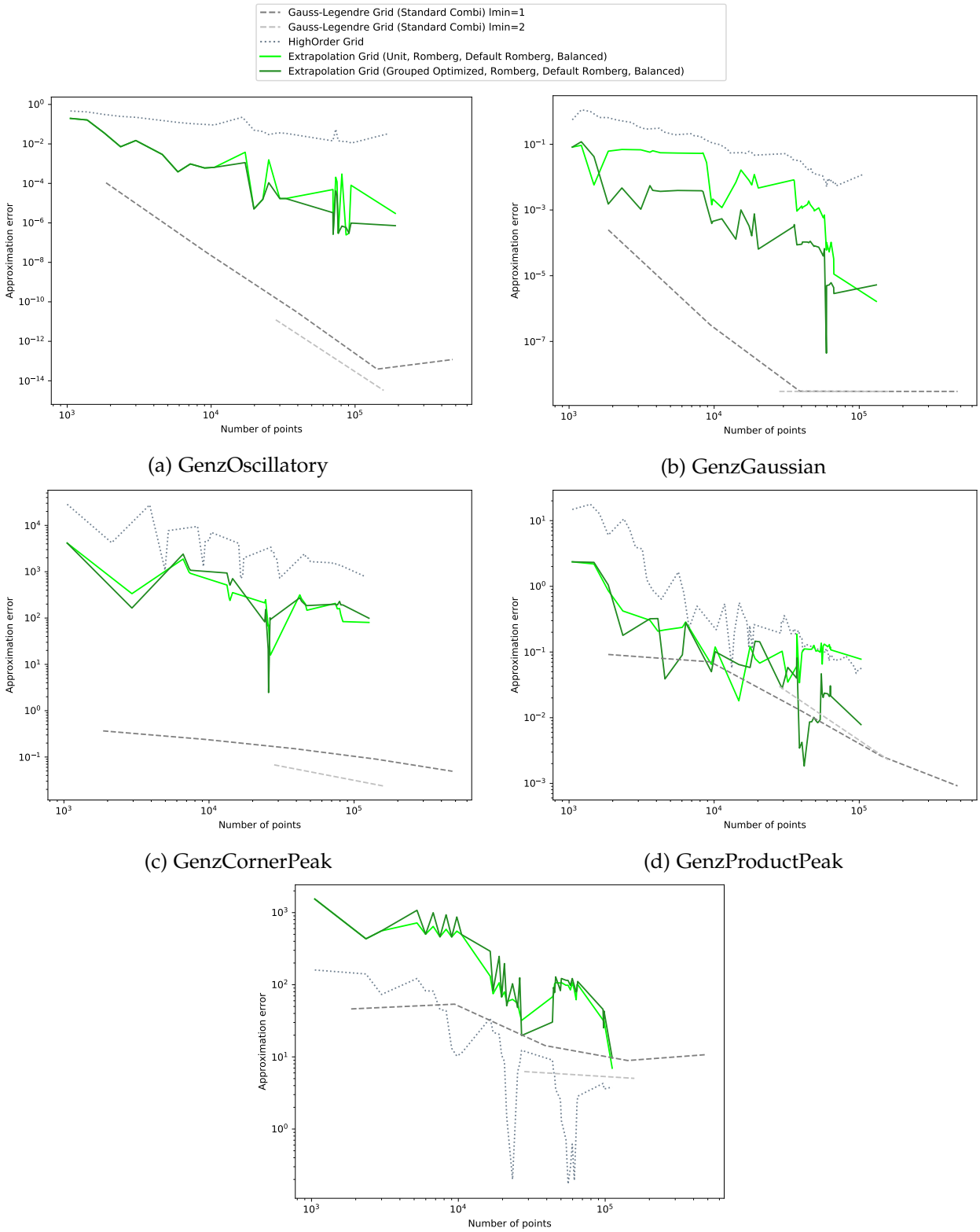
(c) GenzCornerPeak

(d) GenzOscillatory

Figure 5.13: Impact of interpolation on extrapolation

In most cases the interpolation methods perform not as good as the remaining extrapolation methods with grid balancing and optimized grouping.

High order comparsion: Finally, we decided to compare our new best extrapolation methods with another high order method. The HighOrder Grid was already implemented in sparseSpACE.



(a) GenzOscillatory

(b) GenzGaussian

(c) GenzCornerPeak

(d) GenzProductPeak

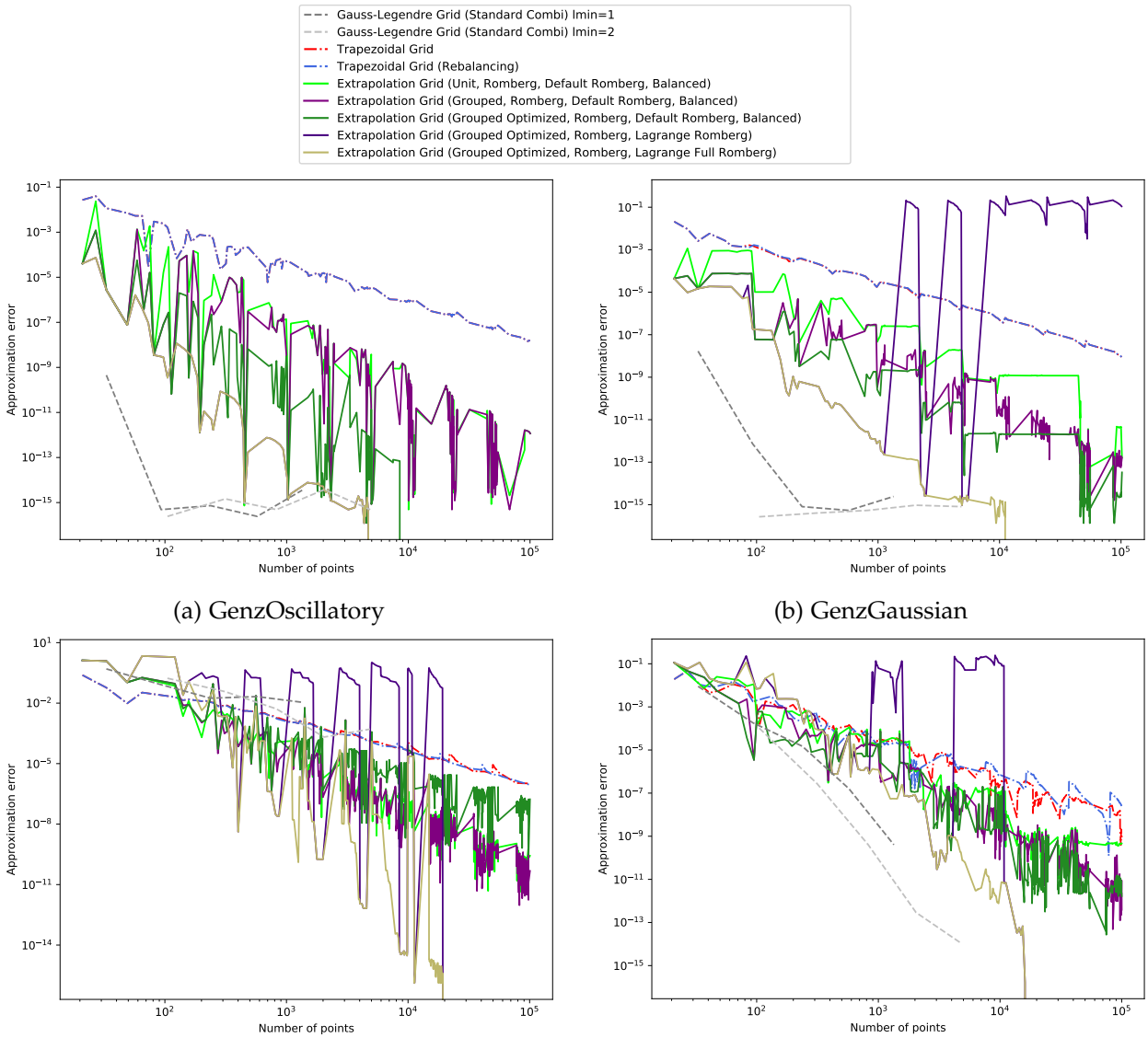
(e) GenzContinuous

Figure 5.14: Comparison of adaptive extrapolation with the previously implemented grid

With the exception of the GenzContinuous function our best new extrapolation variants outperformed the high order grid. It was to be assumed that extrapolation does not work well with GenzContinuous because it is not continuously differentiable.

In the other cases our new implementation improved the results mostly by one or two orders.

Influence of the dimensionality: It appears that the extrapolation grids work even better in lower dimensions where they achieve a significant higher order than non extrapolated methods. This result corresponds to the above-mentioned Sparse grids that are constructed from extrapolated one-dimensional grids. The following figures show convergence diagrams for selected variants in two dimensions:



(a) GenzOscillatory

(b) GenzGaussian

(c) GenzContinuous

(d) GenzProductPeak

Figure 5.15: Two dimensional convergence diagrams

Here one notices the oscillations of high order methods in convergence diagrams that occur because of the error cancelation process. In the four test cases above the extrapolation methods outperform both trapezoidal rules significantly.
The only exception is the extrapolation with adaptive interpolation which oscillates around the lines of the trapezoidal rules with a large amplitude.

Conclusion: We compared Sparse Grids that are based on extrapolated grid stripes with those based on non-extrapolated stripes. Thereby we noticed that the former grids mostly require significantly less points to obtain the same error tolerance. Even when the extrapolated grids required more points, there have been fewer refinement steps. This behaviour resulted mostly in a shorter runtime of the algorithms although the operations of the extrapolation grids are more involved than those of a simple trapezoidal rule.
Additionally the grid balancing had a positive effect on the error cancelation process in the extrapolation despite having the drawback of increasing the number of points slightly. The approximations improved up to one order.
Some of our extrapolation variants worked better than others depending on the test function. It can be said that optimized grouping with grid balancing achieved very good results in a wide range of test cases. Both interpolation approaches performed similar good for smooth functions. As expected the interpolation did not work as well for functions with discontinuities.
It turns out that extrapolation is not always the best approach. For the GenzDiscontinuous function some extrapolation methods performed surprisingly well even though there are discontinuities. However, the GenzContinuous test case showed that both trapezoidal rules perform significantly better than any extrapolation method. In two dimensions this can be explained with the steep peak in the middle of the integration domain which cannot be resolved that well using adaptive extrapolation. In general, the GenzContinuous is not continuously differentiable in this peak. Thus it is not resolved that well because extrapolation requires continuously differentiable functions. Consequently, extrapolation methods should not be used when the functions are not continuously differentiable.
To sum it up, if the conditions are fulfilled, most extrapolation variants perform at least as well as the trapezoidal rule and even surpass the standard combination Gauss-Legendre grids in some cases.

6 Conclusion

In this thesis the Romberg-Quadrature has been generalized to adaptive grids. Several versions of this method have been implemented and successfully integrated into the sparseSpACE framework. We derived error expansions for an adaptive slice extrapolation and proposed some algorithmic improvements to this approach.
The numerical results look promising: Using extrapolation the number of grid points can be significantly reduced in many tests. Additionally, the number of required refinement steps mostly decreases. Although the algorithmic computing complexity of extrapolated weights for an adaptive grid is pretty high the runtime of the algorithms can thereby be reduced.
Unfortunately, our implemented methods could not outperform the trapezoidal rule with rebalancing in every test case regarding the decay of the integral approximation error. When the conditions for extrapolation are not fulfilled the trapezoidal rules perform better. However, the convergence of the methods has been improved in a broad range of test cases. In some cases we even achieved to surpass the standard combination scheme using a Gauss-Legendre grid. These results show the potential of high order methods like a generalized Romberg method for high-dimensional problems. We compared our newly implemented extrapolation grids with another high order method that has been presented by other authors. Our methods outperform it in most of the test cases.
But there is always room for improvement: In future times one could investigate the implementation of a robust extrapolation grid for quadrature. We found out that some extrapolation variants perform better than others depending on the scenario. For example grid balancing improves the results for functions with discontinuities, and an interpolatory approach significantly improves the results for very smooth functions. If an appropriate detector of the scenario could be implemented, these versions could be combined or interchanged to obtain a robust quadrature for a broader range of application.
Another idea of improvement concerns the extrapolation using Simpson quadrature as a base rule. Instead of using this kind of extrapolation only inside containers one could derive an adaptive variant similar to our sliced extrapolation. Additionally the extrapolation constant subtraction could be investigated to increase the order of individual slices even more.
In summary, we showed how promising the potential of adaptive extrapolation for application in quadrature is and hope that this area will be investigated further in the future.

List of Figures

List of Tables

Bibliography

[BG04] H. J. Bungartz and M. Griebel. "Sparse grids." In: *Acta Numerica* 13 (2004), pp. 147–269. ISSN: 09624929. DOI: 10.1017/S0962492904000182.

[Bon94] T. Bonk. "A New Algorithm for Multi-Dimensional Adaptive Numerical Quadrature." In: *Adaptive Methods — Algorithms, Theory and Applications*. 1994.

[Bon95] T. Bonk. "Ein rekursiver Algorithmus zur adaptiven numerischen Quadratur mehrdimensionaler Funktionen." PhD thesis. TUM, 1995.

[Bul64] R. Bulirsch. "Bemerkungen zur Romberg-Integration." In: *Numerische Mathematik* 6 (1964).

[Dei] O. Deiser. *Analysis 2*.

[Dir00] S. Dirnstorfer. "Numerical quadrature on sparse grids." PhD thesis. 2000.

[DR07] P. J. Davis and P. Rabinowitz. *Methods of Numerical Integration*. 2nd ed. Mineola, New york: Dover Publications, Inc., 2007. ISBN: 978-0486453392.

[Eng80] H. Engels. *Numerical Quadrature and Cubature*. 1980.

[Gar13] J. Garcke. "Sparse grids in a nutshell." In: *Lecture Notes in Computational Science and Engineering* 88 (2013), pp. 57–80. ISSN: 14397358. DOI: 10.1007/978-3-642-31703-3-3.

[Gen72] A. Genz. "An adaptive multidimensional quadrature procedure." In: *Computer Physics Communications* 4.1 (1972), pp. 11–15. ISSN: 00104655. DOI: 10.1016/0010-4655(72)90024-0.

[Gen87] A. Genz. "A Package for Testing Multiple Integration Subroutines." In: *Numerical Integration* (1987), pp. 337–340. DOI: 10.1007/978-94-009-3889-2_33.

[GG03] T. Gerstner and M. Griebel. "Dimension – Adaptive Tensor – Product Quadrature." In: *Computing (Vienna/New York)* 71 (2003), pp. 65–87.

[GG10] T. Gerstner and M. Griebel. "Sparse Grids." In: *Encyclopedia of Quantitative Finance* (2010). DOI: 10.1002/9780470061602.eqf12011.

[GG98] T. Gerstner and M. Griebel. "Numerical integration using sparse grids." In: *Numerical Algorithms* 18.3/4 (1998), pp. 209–232. ISSN: 1017-1398. DOI: 10.1023/A:1019129717644.

[GSZ92] M. Griebel, M. Schneider, and C. Zenger. "A combination technique for the solution of sparse grid problems." In: *Iterative Methods in Linear Algebra* (1992), pp. 263–281.

[Hea02] M. T. Heath. *Scientific Computing*. 2002.

[Luc] S. K. Lucas. "An Euler-Maclaurin-like summation formula for Simpson ' s rule."

[OB19] M. Obersteiner and H.-J. Bungartz. "A Spatially Adaptive Sparse Grid Combination Technique for Numerical Quadrature." 2019.

[OB20] M. Obersteiner and H.-J. Bungartz. "A Generalized Spatially Adaptive Sparse Grid Combination Technique with Dimension-wise Refinement." In: (2020), pp. 1–25.

[Obe19] M. Obersteiner. *Übung 7 - Numerisches Programmieren*. 2019.

[Pfl10] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. 2010. DOI: 10.1016/j.jco.2010.04.001.

[Rom55] W. Romberg. "Vereinfachte Numerische Integration." In: *Det Kongelige Norske Videnskabers Selskabs Forhandlinger* 28.01 (1955).

[Sto71] J. Stoer. *Numerische Mathematik*. 1971.