Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Coupling general purpose PDE solvers with a Combination Technique Framework

David Damerow

Department of Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Coupling general purpose PDE solvers with a Combination Technique Framework

Kopplung von universellen PDE-Lösern mittels der Kombinationstechnik

Author:	David Damerow
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Michael Obersteiner
Submission Date:	April 15th, 2018

I hereby declare that this bachelor’s thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

April 15th, 2018 David Damerow

Abstract

This bachelor's thesis deals with the coupling of the combination technique and a general purpose PDE solver in Python. For the FEniCS project which was chosen as the PDE solver in this thesis and all other solvers the curse of dimensionality limits the number of dimensions that can be simulated. Solving PDEs with the finite element method in high dimensions quickly leads to memory problems. The combination technique provides an approximation of high quality for these large-scale problems. It uses the solutions of coarse grids to approximate the solution on fine-mesh grids. However, it is not commonly agreed upon for which PDEs the combination technique does or does not work. The coupling of a general purpose PDE solver and the combination technique makes it possible to investigate this matter further. This thesis implements a basis for this coupling. In this thesis no PDE was found for which the combination technique did not work properly. However, only a few of them - the Poisson, diffusion, linear elasiticy and the Navier-Stokes equations - were tested. The combination technique found the expected solutions for all of them with a small exception of a nonlinear Poisson equation and the Navier-Stokes equation. For both equations the results of the component grids of the combination technique were more precise than the result of the combination technique itself.

Zusammenfassung

Diese Bachelorarbeit behandelt die Kopplung der Kombinationstechnik mit einem universellen PDE Löser in Python. Als universeller PDE Löser wurde das FEniCS Framework gewählt. Diesem Framework und aber auch allen anderen verfügbaren PDE Lösern sind durch den Fluch der Dimensionalität bei der Größe der simulierbaren Probleme Grenzen gesetzt. Das ist der Fall, da das Lösen von PDEs mit der finite Elemente Methode in hohen Dimension schnell zu Speicherproblemen führt. Die Kombinationstechnik stellt eine qualitativ hochwertige Abschätzung für diese hoch-dimensional Probleme zur Verfügung.

Dabei werden mithilfe von gröberen Gittern die Lösung für feinere Gitter abgeschätzt. Allerdings sind die Arten der PDEs, die mithilfe der Kombinationstechnik gelöst werden können, noch nicht vollständig definiert. Die Kopplung dieser Kombiationstechnik mit einem universellen PDE Löser wie dem FEniCS Framework ermöglicht es diesen Bereich besser und genauer zu erforschen. In dieser Bachelorarbeit wurde keine PDE gefunden, für die die Kombinationstechnik nicht funktioniert. Jedoch wurden nur ein paar wenige, wie die Poisson, Diffusions, lineare Elastizität und die Navier-Stokes Gleichung getesetet. Die Kombinationstechnik löste alle Probleme wie erwartet, nur in Bezug auf eine nicht-lineare Poisson und die Navier-Stokes Gleichung ist sie auf kleine Schwierigkeiten gestoßen. Bei beiden Gleichungen waren das beste Ergebnis der Komponentengitter der Kombinationstechnik genauer als die von der Kombinationstechnik gefundene Lösung.

Contents

1. Introduction

The python programming language provides a wide range of possibilites for solving partial differential equations (PDE). Many different frameworks can be found for this issue. They range from FiPy over SfePy to the FEniCS project, just to name a few. All of them feature different advantages and strengths. The FEniCS project for example makes it possible to solve even complicated PDEs with a minimal number of lines of code. Furthermore, PDEs can be solved on complicated domains as well. However, limits are set for all of these frameworks. Admittedly the FEniCS project provides a parallel calculation of PDEs, but only to some extent. This is were the combination technique comes into play. It can reduce the needed time for solving PDEs rapidly and simultaneously keep up a result with a only slightly worse approximation than the actual result. This is achieved by solving the PDE on coarse component grids and combine the results of them. A coupling of those two makes it possible to test the combination technique with a broad range of PDEs. By doing this, new application areas of the combination technique might be identified.

It is structured as the following. Chapter 2 provides a brief introduction in how the FEniCS framework works and can be used. A part of this will be a short introduction to the variational formulation. In chapter 3 the combination technique will be explained shortly; the focus will in particular lay on the truncated combination technique. Chapter 4 contains the main part of this thesis. Thereby a interface between the the FEniCS project and combination technique will be provided and demonstrated with two examples. In chapter 5 more examples of PDEs will be provided. These examples will be solved with the coupling of the combination technique and the FEniCS project and their errors will be analyzed.

2. The FEniCS Project

The FEniCS Project is an open source software project which is used for solving PDEs by the finite element method. FEniCS has been started in 2003 and has been advanced through cooperation between various universities and research institutes. A number of software components are found in the FEniCS software. One of them is DOLFIN; the C++ backend of FEniCS. It offers several data structures, interfaces to the linear solvers and makes it possible to use FEniCS for solving PDEs in both Python and C++. In the following a brief introduction to FEniCS and its software components will be given.[6]

2.1. Finite Element Method

Explaining the finite element method will not be part of this thesis and the implementations of this thesis can be understood with only small knowledge about the inside mechanics of the FEniCS. However, a good recommendation of books about this topic can be found in the FEniCS tutorial.[6] The only part of the finite element method which will be described here, is the transformation of a PDE in its variational formulation. In order to achieve this, the approach will be illustrated with a example PDE: the Poisson equation. This equation and its solution will be specified in section 2.3. The starting point is the original problem formulation (see equation 2.10). Firstly, the PDE has to be multiplied with a so-called *test function* v and integrated over the problem domain Ω. (see equation 2.1).

$$-\int_{\Omega}(\nabla^2 u)v dx = \int_{\Omega} f v dx \tag{2.1}$$

Secondly, integration by parts of terms with second-order derivatives is applied to the equation. It is reasonable to minimize the order of the derivatives and a first-order derivative can be received by integrating by parts:

$$-\int_{\Omega}(\nabla^2 u)v dx = \int_{\Omega} \nabla u \cdot \nabla v dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v ds, \tag{2.2}$$

with $\frac{\partial u}{\partial n} = \nabla u \cdot n$ and n the outward normal direction on the boundary. It pertains to this example that $v = 0$ on the boundary $\partial\Omega$, since the test function v is zero where the solution u is predefined (for more explanations about this see the FEniCS tutorial.[6]) As it can be seen in section 2.3 the wanted function u is defined on the boundary $\partial\Omega$. With this in mind, the equation 2.2 is simplified to the following:

$$-\int_{\Omega}(\nabla^2 u)v dx = \int_{\Omega} \nabla u \cdot \nabla v dx \tag{2.3}$$

Combining the results from equations 2.1 and 2.3 the *weak* or *variational form* in a not yet defined *test space* $\hat{V}$ can be received:

$$\int_{\Omega} \nabla u \cdot v dx = \int_{\Omega} f x dx \quad \forall v \in \hat{V}. \tag{2.4}$$

Additionally to the test space a function space which contains the solution u has to be defined. Those two function spaces do not have to be necessarily the same. For this example problem these function spaces can be described as

$$V = \{v \in H^1(\Omega) : v = u_D \ on \ \partial\Omega\}, \tag{2.5}$$

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \ on \ \partial\Omega\}. \tag{2.6}$$

Here, H^1 denotes the Sobolev space in which the finite integrals over Ω of v^2 and $|\nabla v|^2$ are defined. The PDE has to be committed to FEniCS in the following notation. The unknown function $u \in V$ has to be found so that

$$a(u, v) = L(v) \ \forall v \in \hat{V} \tag{2.7}$$

is fulfilled. $a(u, v)$ and $L(v)$ are called the *bilinear form* and the *linear form*, respectively. Following the above explained rules and strategies a given PDE can be transformed to the variational formulation and passed over to FEniCS in the correct way.[6]

2.2. Main Features of the FEniCS Project

2.2.1. Generating Meshes

FEniCS offers different ways to generate meshes. Firstly, DOLFIN provides multiple classes to create easy and regular meshes. By default rectangular cells form each mesh and these rectangles are divided into two triangles.[6] The following code (see source code 2.1) creates a mesh, which containts 5 x 5 cells, over the unit square [0,1] x [0,1]. Arbitrary rectangular domain regions can be defined by using the class *RectangleMesh* and is plotted in figure 2.1.

```
mesh = UnitSquareMesh(5,5)
```

Source Code 2.1.: Unit Square Mesh

Three dimensional meshes can be generated as the two dimensional spaces before (see source code 2.2).

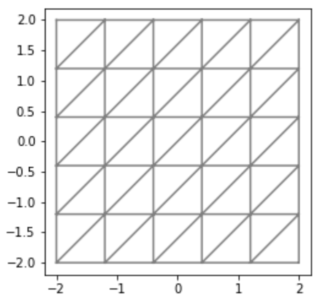

Figure 2.1.: Rectangular mesh over square [-2,-2] x [2,2]

```
mesh = UnitCubeMesh(5,5,5)
```

Source Code 2.2.: Unit Cube Mesh

Secondly, meshes can be defined by using one of FEniCS's software components: *mshr*. This way, more complicated meshes can be generated (see source code 2.3). Here, a rectangular and a circular domain are defined. Afterwards the circular domain is placed in the rectangular domain by substracting it from the rectangular domain. With this domain a mesh can be generated. The argument 16 specifies the number of cells across the domain's diameter (see figure 2.2).[6]

```
from mshr import *
channel = Rectangle(Point(0, 0), Point(2.0, 1.0))
cylinder = Circle(Point(0.4, 0.5), 0.3)
domain = channel - cylinder
mesh = generate_mesh(domain, 16)
```

Source Code 2.3.: Empty circle in rectangular grid

Thirdly, predefined meshes can easily be imported from a file (see source code 2.4).

```
mesh = Mesh(filename)
```

Source Code 2.4.: Import mesh from a file

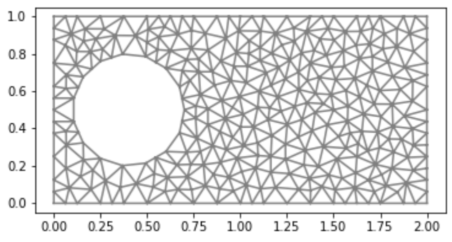

Figure 2.2.: Empty circle in rectangular grid

2.2.2. Function Spaces

Before any function can be formulated, it is necessary to define a function space. These can either be a scalar or a vectorial function space. Multiple function spaces per mesh can be defined (see source code 2.5).

```
V = FunctionSpace(mesh,'P',1)
Q = VectorFunctionSpace(mesh,'P',1)
```

Source Code 2.5.: Function spaces

The second and third argument indicates the type and the degree of the finite elements, which is used for solving the PDE. In this case the type of element is P; the Lagrange family of elements.[6] All of FEniCS supported types of elements can be found in the FEniCS documentation[1].

2.2.3. Boundary Conditions

The FEniCS project implements three different types of boundary conditions: Dirichlet, Neumann and Robin conditions. Furthermore, different boundary conditions in one mesh can be defined.[1] However, in this thesis, in order to keep it clear, only Dirichlet conditions will be covered. The boundary condition is defined by the simple function

$$u_D = 1 + x_0^2 + 2x_1^2 \tag{2.8}$$

which is applied to every cell at the boundary (see source code 2.6). Therefore a function is defined which returns True if a point is on the boundary (see lines 1-2). In this function

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

x corresponds to the spatial coordinate and *on_boundary* is a boolean value which denotes whether this point lays on the boundary or not. The Dirichlet boundary conditions are initialized with this function and u_D. x_0 and x_1 represent the two spatial coordinates in this 2D example. This function is declared with another one of FEniCS' software components: *Unified Form Language* (UFL). A string formulation of the function in C++ syntax has to be forwarded to the Expression class.[1]

```
def boundary(x, on_boundary):
        return on_boundary
u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
bc = DirichletBC(V, u_D, boundary)
```

Source Code 2.6.: Boundary condition

2.2.4. Defining Functions

As mentioned in section 2.1, a trial and a test function are essential for solving PDEs with FEniCS. They can easily be defined with a function space V (see source code 2.7).

```
u = TrialFunction(V)
v = TestFunction(V)
```

Source Code 2.7.: Test and trial funtion

More source terms can be defined as in section 2.2.3 or as Constants (see source code 2.8). The actual variational problem is declared as the previous expression in UFL; however, a slightly different syntax has to be used (see source code 2.8). In this example the following equations in the style of the variational formulations from section 2.1 are formulated:

$$\begin{aligned} a &= \int_\Omega \nabla u \cdot \nabla v dx \\ L &= \int_\Omega f v dx \end{aligned} \tag{2.9}$$

```
f = Constant(-6.0)
a = dot(grad(u), grad(v))*dx
L = f*v*dx
```

Source Code 2.8.: Variational Problem

After declaring the test and trial function, FEniCS can easily solve the PDE (see source code 2.9).[6]

```
u = Function(V)
solve(a==L,u,bc)
```

Source Code 2.9.: Solving the PDE

2.3. Poisson Example

All of the previously mentioned functions can be used to solve simple PDEs, however, more complicated examples require more complex functions and multiple terms. For simplicity we restrict ourselve in this overview to the solution of the boundary value problem of the Poisson equation:

$$\begin{aligned} -\nabla^2 u(x) &= f(x), x \ in \ \Omega, \\ u(x) &= u_D(x), x \ on \ \partial\Omega. \end{aligned} \tag{2.10}$$

In this equation u denotes the unknown function, f an predefined function, Ω the domain and u_D the boundary condition. According to section 2.1 the equations have to be rewritten as a variational formulation. The Poisson equations formulated as a variational problem are described as

$$a = \int_\Omega \nabla u \cdot \nabla v dx, \tag{2.11}$$

$$L = \int_\Omega f v dx. \tag{2.12}$$

So far only the general Poisson equation has been discussed. For an actual example the boundary values u_D and the function f have to be defined. By inserting a quadratic function $u_e(x, y) = 1 + x^2 + 2y^2$ in the equation 2.10 both of them can be calculated:

$$f(x, y) = -6 \qquad u_D = u_e = 1 + x^2 + 2y^2 \tag{2.13}$$

In this example the domain shall be a two dimensional unit square over [0,1] x [0,1] with 10 cells in each directions (see source code 2.10).[6] By running the program the following solution can be calculated:

```python
def boundary(x, on_boundary):
    return on_boundary
mesh = UnitSquareMesh(9,9)
V = FunctionSpace(mesh, 'P', 1)
u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
bc = DirichletBC(V, u_D, boundary)
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = dot(grad(u), grad(v))*dx
L = Constant(-6.0)*v*dx
u = Function(V)
solve(a == L, u, bc)
plot_solution = plot(u)
colorbar(plot_solution)
```

Source Code 2.10.: Implementation of the Poisson equation in 2D

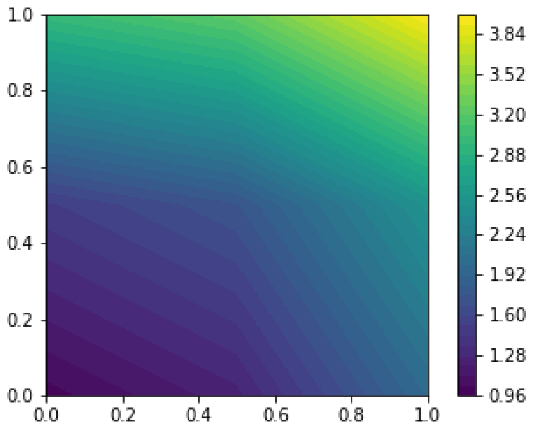

Figure 2.3.: Solution for Poisson equation

3. The Combination Technique

The finite element method is an useful and wide spread method for solving PDEs. However, it is restricted to rather small problems due to its high number of computations; especially high-dimensional problems are affected by these restrictions. This issue is known as the so-called *curse of dimensionality*.[2] For a uniform mesh with N grid points in one dimension, the complexity of the program increases exponentially to $O(N^d)$ with d dimensions. The number of operations and the time which is necessary for them increases exponentially as well. If a so-called *sparse grid* is used instead of the full grid, the complexity can be reduced to $O(N \cdot log(n)^{d-1})$. By using the so-called *combination technique* $O(d \cdot log(N)^{d-1})$ grids with the grid size $O(N)$ have to be used to solve the problem. This technique was established by Griebel et al.[4]

The Sparse Grid Combination Technique is based on the following two assumptions. The first one is that the solution of a PDE on a sparse grid is only slightly less accurate than the solution on the corresponding full grid. Provided certain smoothness conditions, it can be assumed that the solution of the combination technique corresponds to the solution of the full grid to a large extent.[3] The second assumption is that solution of a grid with a higher resolution can be composed by the solutions of grids with lower resolutions. This can be useful for problems that can only be solved for the component grids and not for the full grid.[4] The underlying theory and how it can be implemented will be introduced in the following briefly.

3.1. Sparse Grids and Hierarchical Subspaces

The goal of the combination technique is to approximate a multivariate function u, $u(\vec{x}) \in \mathbb{R}$, $\vec{x} := (x_1, ..., x_d) \in \Omega$ on a d-dimensionl unit interval $\Omega := [0,1]^d$. This approximation can be received by linearly combining basis function $\varphi_i(x) : \Omega \to \mathbb{R}$:

$$u(\vec{x}) \approx \sum_i \alpha_i \varphi_i(\vec{x}) \tag{3.1}$$

The function u is solved on a so-called *full grid* and therefore the unit interval Ω corresponds to a regular grid Ω_n with equidistant spacing between its grid points. $n \in \mathbb{N}_d$ represents a so-called *levelvector*. For levelvectors we define the following rules:

$$\vec{a} \leq \vec{b} \iff a_k \leq b_k \; and \; \vec{a} < \vec{b} \iff a_k < b_k \tag{3.2}$$

A grid $\Omega_{\vec{n}}$ has

$$N = |\Omega_{\vec{n}}| = \prod_{k=1}^{d} (2^{l_k} \pm 1) \tag{3.3}$$

grid points and a meshwidth of

$$h_k = 2^{-n_k} \tag{3.4}$$

(-1 if the grid has no boundary, +1 if it has one).[5] A example *full grid* $\Omega_{(4,4)}$ is plotted in figure 3.1. The simplest choice for a basis function for a one-dimensional grid is the

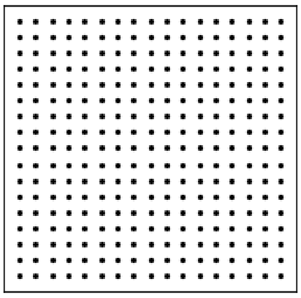

Figure 3.1.: Full grid with 17 cells in each direction

so-called *hat function*

$$\varphi_i(x) = max\left(1 - \frac{|x - x_i|}{h}, 0\right). \tag{3.5}$$

The equation 3.5 can be extended to d-dimensions with a tensor product:

$$\varphi_i(\vec{x}) = \prod_{k=1}^{d} max(1 - \frac{|(\vec{x})_k - (\vec{x_i})_k|}{h_k}, 0) \tag{3.6}$$

With the aid of these basis function a to the grid $\Omega_{\vec{n}}$ matching functionspace can be devised:

$$V_{\vec{n}} := span\{\varphi_i : 0 \le i \le N - 1\} \tag{3.7}$$

This function space can be broke down in its so-called *hierarchical subspaces*. The coherence between these two is described as the following:

$$V_{\vec{n}} = \bigoplus_{\vec{l} \le \vec{n}} W_{\vec{l}} \tag{3.8}$$

A so-called *sparse grid* $\Omega_{\vec{n}}^{(s)}$ composes itself by a combination of the hierarchical subspaces $W_{\vec{l}}$. This combination is described by the equation 3.9. In order to get the sparse grid, only a few of these subspaces have to be combined. These subspaces are defined by the following equation:

$$V_{\vec{n}}^{(s)} := \bigoplus_{|\vec{l}| \leq n+d-1} W_{\vec{l}} \tag{3.9}$$

Here, $|\vec{l}|$ denotes the L1-norm of $\vec{l}$:

$$|\vec{l}| := \sum_{k=1}^{d} l_k \tag{3.10}$$

In figure 3.2 all hierarchical subspaces of a grid $\Omega_{(3,3)}$ and the corresponding sparse grid $\Omega_{(3,3)}^{(s)}$ can be seen. All hierarchical subspaces which are not necessary for the sparse grid are drawn in grey. If all subspaces of figure 3.2 are combined a full grid $\Omega_{(3,3)}$ will be received.

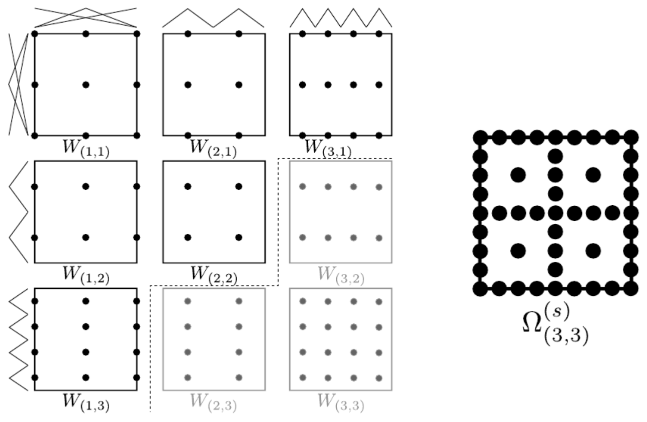

Figure 3.2.: Hierarchical subspaces of a grid $\Omega_{(3,3)}$ and the corresponding sparse grid (Heene 2017, p. 11)

A to the grid of figure 3.1 corresponding sparse grid can be seen in figure 3.3. The advantage of sparse grids towards full grids is the lower amount of necessary computations to solve a function in it. The number of grid points in a sparse grid $O(N \cdot (logN)^{d-1})$ is significantly lower than for full grids $O(N^d)$. However, the accuracy of sparse grids $O(N^{-2} \cdot (logN)^{d-1})$ is only slightly worse than for full grids $O(N^{-2})$.[3]

3.2. Sparse Grid Combination Technique

As mentioned in section 3.1, the combination technique is used to approximate a function u on a full grid $\Omega_{\vec{n}}$. Therefore the approximation $u_{\vec{n}}^{(s)}$, which denotes the function u on a sparse grid $V_{\vec{n}}^{(s)}$, is determined. In order to receive this function, the approximations u_i on

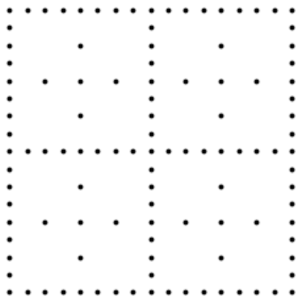

Figure 3.3.: Sparse grid of $\vec{l} = (4, 4)$

all hierarchical subspaces of the full grids Ω_i are calculated and afterwards all of them are combined using a certain weigh c_i:

$$u_n^{(c)} = \sum_{i \in I} c_i u_i. \tag{3.11}$$

Here, I denotes the index set of all grids that are included in the combination. From this point on u_i will be called *component solution*, Ω_i *component grid*, c_i *component coefficient*. The component coefficients have to be chosen deliberately in order to work for the combination technique.[5] One way to choose the component coefficients is the so-called *Truncated Combination technique*:[8]

$$u_{n,\tau}^{(c)} \approx \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{i \in I_{q,\tau}^{d,n}} u_i \tag{3.12}$$

with a binomial coefficient $\binom{d-1}{q}$ and the index set

$$I_{q,\tau}^{d,n} = \{i : |i|_1 = n + (d-1) + |\tau|_1 - q, \quad i > \tau\}. \tag{3.13}$$

According to this equation, the combination technique for a grid $\Omega_{(4,4)}$ would be implemented as shown in figure 3.4.

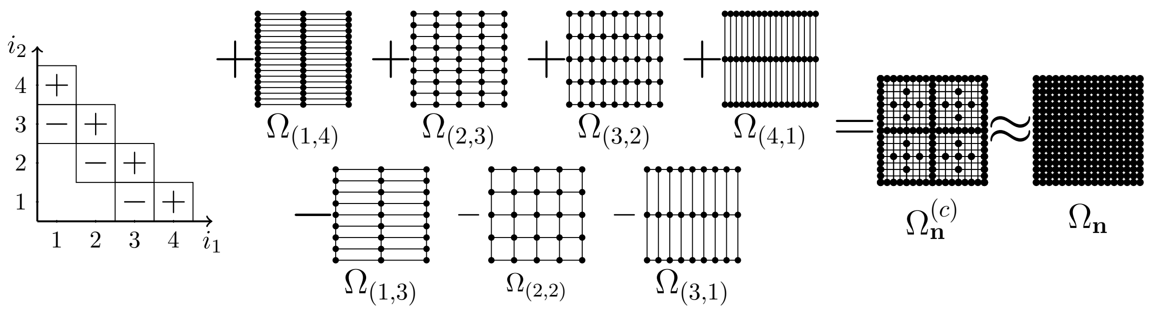

Figure 3.4.: Result of the combination technique with a grid $\Omega_{(4,4)}$ and $n = 4$ and $\tau = (0,0)$(Obersteiner et al., 2017)

However, there are problems which cannot be solved for all component grids, especially not the ones with the lowest resolution. For this case, the truncation paramter τ can be for example set to $(1,1)$ to enforce a minimal level of 2 for each dimension. For the same grid as in figure 3.4 and therefore $n = 3$ the solution would be composed by the following functions:[8]

$$u^{(c)}_{3,(1,1)} = u_{(2,4)} + u_{(3,3)} + u_{(4,2)} - u_{(2,3)} - u_{(3,2)}. \tag{3.14}$$

Solving a time dependent problem with the combination technique is slightly more complex. A general formulation of this concept can be found in equation 3.15.

$$u^{(c)}_{n,t+\Delta t,\tau} = \sum_{i \in I^{d,n}_{q,\tau}} c_i F_i\{P_i(u^{(c)}_{n,t,\tau})\} \tag{3.15}$$

Here, P_i denotes a projection of the combined solution $u^{(c)}_n$ in order to fit into the original component solution u_i. F_i is an individual factor which is used to develop the component solution by the timestep Δt. In this thesis in every timestep the combined solution of the previous timestep is interpolated to the size of each component grid and the simulation in each component grid is resumed with the result of this interpolation. Therefore P_i is in this case a simple interpolation to the size of the component grid, and F_i just the solution of the PDE which is used for calculating the component solution in each timestep (see figure 3.5).[5]

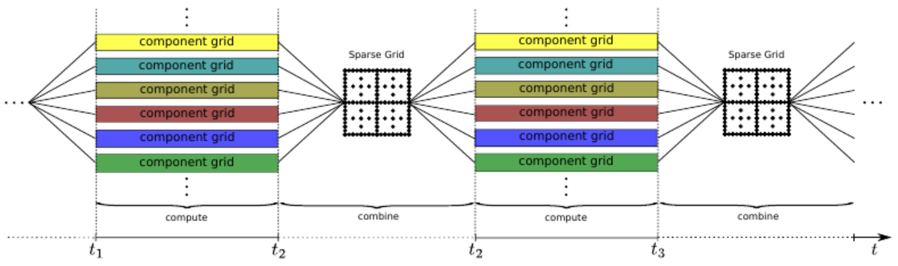

Figure 3.5.: Concept of time stepping with the combination technique (Heene 2017, p. 16)

3.3. Combination Strategies

There are a few strategies which can be used for actual combining the component solution. Here, only two of them will be introduced briefly. Firstly, each component grid can be interpolated to the size of the full or sparse grid. For the combination we then just need to pointwise add up the interpolated component grids. However, the number of arithmetic operations and the amount of needed memory is quite high for this approach. Especially the amount of memory rises remarkably; it rises up to $O(N^d)$ for each component grid of the full grid. Yet, a big advantage of this approach is its easy implementation. A different and more cost-saving approach is to combine in the hierarchical basis. Here, hierarchical coefficients are added in the sparse grid's hierarchical basis. Thus, the amount of data is minmized to $O(N)$ per each component grid. Therefore the memory costs for this approach are reduced to $O(N \cdot log(N)^{(d-1)})$ as the resulting grid is the corresponding sparse grid. For a more in-depth look of this approach, the dissertation of Mario Heene can be consulted.[5] Consequently, this approach seems to be more promising for solving large scale problems with the combination technique. However, in this thesis, each component grid is interpolated to the full grid.

4. Implementation

The main topic of this thesis is to combine the FEniCS framework and the combination technique. Most of the basic functions for the combination techqniue were implemented by the Chair of Scientific Computing in Computer Science of Technical University of Munich's informatics departement. The FEniCS project is, as previously mentioned, an open source project for solving PDEs. Hence, this bachelor thesis has to concern itself with an interface between these technologies. In the following all implemented classes and functions will be presented

Grid with arbitrary dimensions
So far only a two dimensional dummy grid was implemented for the combination technique. The class *DummyGridArbitraryDim* provides grids with arbitrary dimensions (see Appendix A). It inherits from the class *combiGrid* and implements its abstract methods. By initializing a instance of this class, the levelvector, the size of the data grid and the boundaries are set. For filling the grids with data the method *fillData* can be used. If no additional parameter is passed onto the function, the grid is fill with dummy data. As additional parameters either a function or a *numpy array* are accepted. *fillData* uses the passed function to fill the grid or sets it to the passed array. For the here used combination technique, the component grids need to be able to be interpolated to the full grid. In order to fulfill this purpose the function *extrapolateData* was implemented. Firstly, it calculates the factor by which the component grid has to be "zoomed in" to get the interpolated component grid. Secondly, it uses the *zoom* function from the SciPy framework to receive the interpolation. Here, it is important that the interpolation is linear, since the combination technique has had issues with non linear interpolations.

Helper Functions
The combination of the FEniCS projects and the combination technique need a couple of helper functions to work properly. Their purpose range from plotting results over calculating errors to interpolating grids. However, the most important functionality of this thesis - the transformation of the data for the combination technique - is implemented with library functions. It can be done by the following function calls:

```
data = np.reshape(convert_dof(von_Mises.vector().get_local(),\
        dof_to_vertex_map(V)),2**np.array(key)+1)
```

It comprises the functions *get_local*, *convert_dof*, *dof_to_vertex_map* and *reshape*. By calling the function *get_local* all of the grid data is received from the FEniCS project. However, the data is in the wrong order; this is where *dof_to_vertex_map* is deployed. It returns an array which contains the so-called *degress of freedom* map. Using this, the data can be brought in the correct order by using the function *convert_dof*. The implementation of this function can be found in appendix B. Finally, the shape of the data has to be changed to the corresponding shape of the component grid. After getting the data this way, the component grid can easily be filled with the above mentioned functions. However, transforming the data of a vector function space is a bit more complicated; especially if only one component of the vector is used for the combination technique (see Source Code 4.1). Firstly, the *dof_map* and

```
1 dof_map = V.sub(0).dofmap().dofs(mesh,0)
2 data_fenics = u.vector().get_local()
3 data = np.zeros(dof_map.shape)
4 index = 0
5 for i in df_map:
6     data[index] = data_fenics[i]
7     index+=1
8 data = np.reshape(data,(2**np.array(key)+1))
```

Source Code 4.1.: Getting the data from a vector function space

all data is requested from FEniCS (see line 1 and 2). The *dof_map* contains all indices of the data in the subspace *sub(0)* which is the x-components of the vector function space. This is achieved by calling a few functions with the *function space* V and the grid $mesh$. The data is directly received from the function u. Afterwards a *numpy array* with the shape of the *dof_map* is intialized. This *numpy array* is filled with the data by a simple for-loop where the right indices can be found in the *dof_map*. Finally, as seen above, the array has to be brought in the right shape and can afterwards be passed to the combination technique. In the following two examples for the combination of these two technologies will be shown, a time dependent and a non-time dependent example.

4.1. Non-Time Dependent Example

The example which was chosen for the non-time dependent example is the simulation of a deformation of an elastic body. The deformation of elastic bodies can be described and calculated with PDEs. A small deformation of an elastic body Ω can be modeled by

$$-\nabla \cdot \sigma = f \; in \; \Omega, \tag{4.1}$$

$$\sigma = \lambda tr(\epsilon) I + 2\mu\epsilon, \tag{4.2}$$

$$\epsilon = \frac{1}{2}(\nabla u + (\nabla u)^T), \tag{4.3}$$

with the tensor stress σ, the body force per unit volume f, the Lamé's elasticity parameters λ and μ for the body's material, the identity tensor I, the trace operator on a tensor tr and the displacement vector field u. The transformation to the variational form will be omitted in this thesis and can be found in the FEniCS book.[7] The variational formulation of the deformation of elastics bodies reads as follows:

$$a(u, v) = \int_\Omega \sigma(u) : \nabla v dx, \tag{4.4}$$

$$\sigma(u) = \lambda(\nabla \cdot u) I + \mu(\nabla u + (\nabla u)^T), \tag{4.5}$$

$$L(v) = \int_\Omega f \cdot v dx + \int_{\partial\Omega_T} T \cdot v ds. \tag{4.6}$$

After the deformation vectors are calculated, a stress measure can be computed. In this example the von Mises stress is chosen:

$$\sigma_M = \sqrt{\frac{3}{2} s : s}, \tag{4.7}$$

$$s = \sigma - \frac{1}{3} tr(\sigma) I. \tag{4.8}$$

In this example a fastened beam, which is deformed under its own weight will be modeled. The body force per unit is set to $f = (0, 0, \varrho g)$. Here g is the gravitational acceleration and ϱ the density of the beam. At the fastened end the deformation vector is set to $u_D = (0, 0, 0)$ and at the rest of the beam there is no traction and therefore set to 0. The beam is shaped like a cuboid.[7] The full implementation of the problem can be found in source code 4.2

```
 1 def clamped_boundary(x, on_boundary):
 2     return on_boundary and x[0] < tol
 3 def epsilon(u):
 4     return 0.5*(nabla_grad(u) + nabla_grad(u).T)
 5 def sigma(u):
 6     return lambda_*nabla_div(u)*Identity(d) + 2*mu*epsilon(u)
 7 tol = 1E-14
 8 mu = 1
 9 rho = 1
10 delta = 0.2/1
11 gamma = 0.4*delta**2
12 beta = 1.25
13 lambda_ = beta
14 g = gamma
15 lmin = 2,1,1
16 lmax = 5,4,4
17 factory = ActiveSetFactory.ClassicDiagonalActiveSet(lmax, lmin, 0)
18 activeSet = factory.getActiveSet()
19 scheme = combinationSchemeArbitrary(activeSet)
20 keys = scheme.dictOfScheme.keys()
21 combiGrid = combineGrids(scheme)
22 for key in keys:
23         solve_problem_with_fenics(key)
24         grid = DummyGridArbitraryDim(key,tuple(\
25                 [True for __ in range(len(key))]))
26         data = np.reshape(convert_dof(von_Mises.vector().get_local(),\
27             dof_to_vertex_map(V)),2**np.array(key)+1)
28         grid.fillData(f=data)
29         combiGrid.addGrid(grid)
30 erg_combi = real(combiGrid.getCombination())
```

Source Code 4.2.: Solving the linear elasticity problem with FEniCS and the combinatino technique

Firstly, the functions for the boundary, ϵ and σ and a few constants for the FEniCS implementation have to be defined (see lines 1-14). Secondly, the combination technique has to be prepared (see lines 15-16). Here, the full grid is characterized with the levelvector $\vec{l} = (5,4,4)$ and the lowest resolution is set to $\tau = (1,0,0)$. Afterwards the class which is responsible for combining the component grids (*combineGrids*) is initialized and all levelvectors of the component grids need to be passed onto it (see line 20). After the preparation is finished, each component grid can be solved (see line 23). In this source code the simulation is shortened and implied by a dummy function in order to keep it short. After computing the unknown function u, the von Mises stress can be calculated (see equation 4.7).The result of this can be passed over in the way of chapter 4 (see line 26), each grid can be filled with the data (see line 26) and the component grid can be added to the combination technique (see line 28). Finally, the result of the simulation is accessed by combining the grids (see line 29). The resulting plot can be seen in figure 4.1.

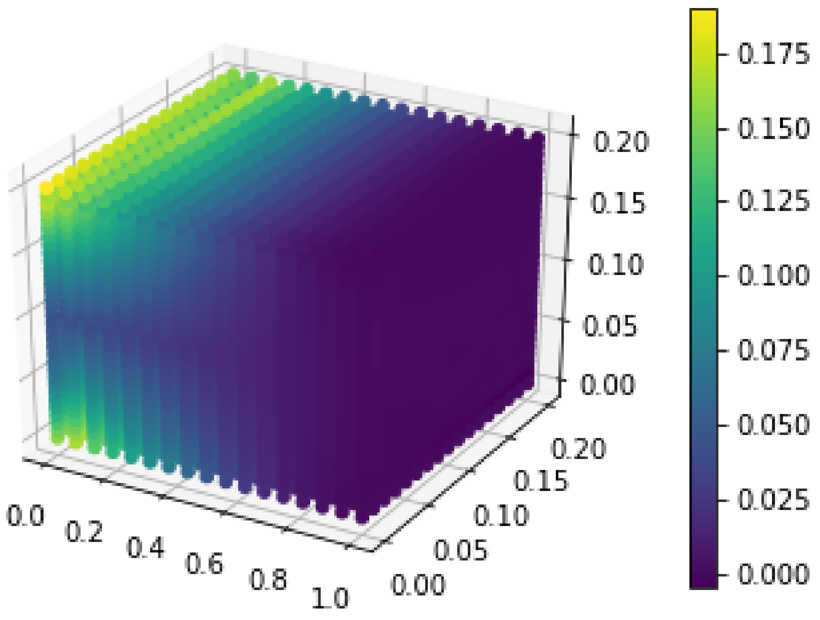

Figure 4.1.: Solution for the linear elasticity example

4.2. Time Dependent Example

The time dependent problems are a bit complexer, since the results of the combined component grids have to be reused in the FEniCS simulation (see section 3.2). Furthemore, all variables and functions of the FEniCS simulation are required in every timestep, thus, they have to be stored separately. The example which will be demonstrated here is the time-dependent diffusion equation. It is described by the following equations:

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \qquad \text{in } \Omega\, x\, (0,T], \tag{4.9}$$

$$u = u_D \qquad \text{on } \partial\Omega\, x\, (0,T], \tag{4.10}$$

$$u = u_0 \qquad \text{at } t = 0. \tag{4.11}$$

In this case, u is always depending on the spatial coordinatees and the time, the boundary condition u_D and the source function f are possibly depending on the spatial coordinates and time; the intial condition u_0 only depends on the spatial coordinates. The variational formulation of the above equations is given by:

$$a(u,v) = \int_\Omega (uv + \Delta t \nabla u \cdot \nabla v) dx, \tag{4.12}$$

$$L_{n+1}(v) = \int_\Omega (u^n + \Delta t f^{n+1}) v dx. \tag{4.13}$$

Additionally, the initial condition has to be approximated:

$$a_0(u,v) = L_0(v), \tag{4.14}$$

$$a_0(u,v) = \int_\Omega uv dx, \tag{4.15}$$

$$L_0(v) = \int_\Omega u_0 v dx. \tag{4.16}$$

The example presented here is the diffusion of a so-called *gaussian hill*. The grid will be initiated with the following function:

$$u_0(x,y) = e^{-ax^2 - ay^2} \tag{4.17}$$

with $a = 5$. The grid $\Omega_{(5,5)}$ will be solved on the domain [-2,2] x [2,2] with the Dirichlet boundary conditions $u_D = 0$.[6] The whole implementation of this problem can be found in source code 4.3. As in section 4.1 FEniCS and the combination techqniue have to be initialized (see lines 1-4). The only difference is that the FEniCS function and variables have to be stored seperately for all timesteps; this will be realised with the aid of a dicitionary (see line 1). The preparation of FEniCS and the combination technique is, in order to keep it clear, replaced with dummy functions (see line 2-4). After all preparations are finished the actual simulation of the diffusion can be performed (see lines 7-23). For every time step after the first on each component grid has to be resetted to the interpolated solution of the combination grid (see lines 8-14). Therefore a grid is initialized in lines 25-28. It is filled with the result of the combination technique and the solution is applied to every of the component grids. This is achieved by interpolating the result to the right size (see line 28). So, after every component grid contains the updated solution, the data of the FEniCS function has to be changed as well. This is done by using the functions *set_local*. The *dofmap* is necessary to bring the data in the right order. In contrary to the function *dof_to_vertex_map*, its indices denote the position of each grid point in the FEniCS data. Using this *dofmap* the data in the FEniCS simulation can easily be changed (see line 14). Hereafter, every component grid can be solved like previously in section 4.1. The only additional line of code for time dependent PDEs prepares the next time step by updating the solution of it with the current time step (see line 23). Three different timesteps of the diffusion of a gaussian hill

```
1   grids = {}
2   for key in keys:
3           grids[key].fill_with_fenics_functions(key)
4   prepare_combination_technique((5,5),(1,1))
5   for n in range(num_steps):
6           t+=dt
7           for key in keys:
8                   if n > 0:
9                           keyDict = ((key),)
10                          data = combiGrid.gridsDict[keyDict].getData()
11                          dofs = grids[key]['V'].dofmap()./
12                          dofs(grids[key]['mesh'],0)
13                          grids[key]['u'].vector()./
14                          set_local(data.flatten(),dofs)
15                  solve(grids[key]['a']==grids[key]['L'],/
16                          grids[key]['u'],grids[key]['bc'])
17                  grid = DummyGridArbitraryDim(key,tuple(\
18                          [True for __ in range(len(key))]))
19                  data = np.reshape(convert_dof(grids[key]['u'].vector().\
20                          get_local(),grids[key]['d2v']),2**np.array(key)+1)
21                  grid.fillData(f=data)
22                  combiGrid.addGrid(grid)
23                  grids[key]['u_n'].assign(grids[key]['u'])
24          erg_combi = combiGrid.getCombination()
25          grid = DummyGridArbitraryDim(lmax,/
26                  tuple([True for __ in range(len(key))]))
27          grid.fillData(erg_combi)
28          combiGrid.applyCombinedResultToComponentGrids(grid)
```

Source Code 4.3.: Solving the diffusion of a gaussian hill

are plotted in figure 4.2.

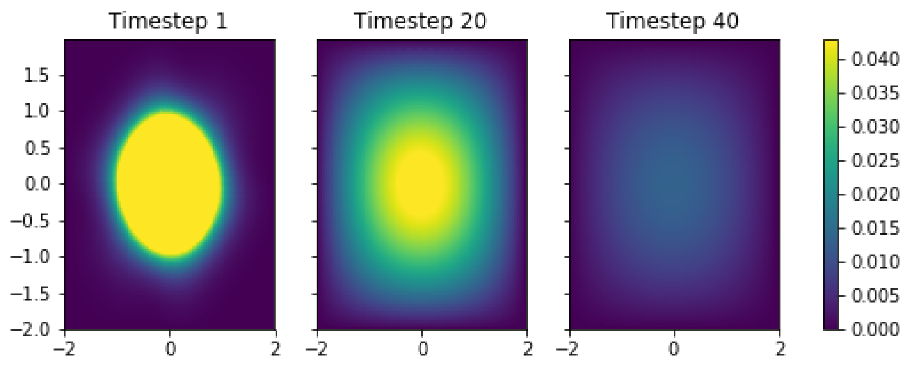

Figure 4.2.: Solution for the difussion of a gaussian hill

5. Results

In this chapter four different problems and their errors will be analyzed. Among these examples, are the ones, that have been mentioned in this thesis before. For analyzing the error the difference between a full grid and the sparse grid will be calculated. Furthermore the time which is needed for every calculation will be included in this analysis.

5.1. 2D Poisson Equation

The first example is the Poisson equation in a two dimensional domain (see section 2.3). It is solved for a grid $\Omega_{(5,5)}$ and the truncation parameter is $\tau = (0,0)$. The difference between the solution of the full grid $u_{(5,5)}$ and the combination technique $u^{(c)}_{(5,5)}$ is plotted in figure 5.1.

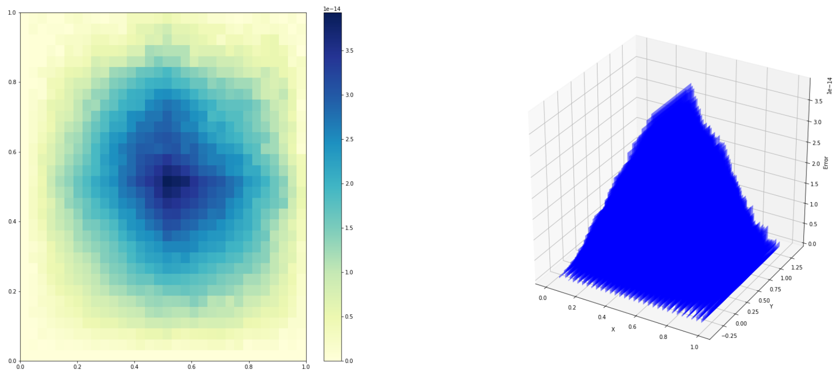

Figure 5.1.: Poisson error for a domain $\Omega_{(5,5)}$

As it can be seen in this plots, the error is the highest in the middle of the domain. This is due to the fixed definition of the boundaries.

A different approach to analyzing the error is by calculating the L2-Norm (see Equation 5.1).

$$S = \sum_{i=0}^{n} (y_i - x_i)^2 \tag{5.1}$$

Firstly, for the calculation of the error graph a full grid $\Omega_{(10,10)}$ is solved. The combination technique is applied to all levelvectors between $\vec{l} = (2,2)$ to $\vec{l} = (10,10)$. The truncation parameter for all the grids will be $\tau = (0,0)$. The results of all combinations will be interpolated to the size $\vec{l} = (10,10)$ and compared to the solution of the full grid $\Omega_{(10,10)}$. Additionally, the best component grid of each combination will be determined and compared to the full grid as well. The comparison will be performed by calculating the L2-Norm according to equation 5.1. The result of these calculation is plotted in figure 5.2.

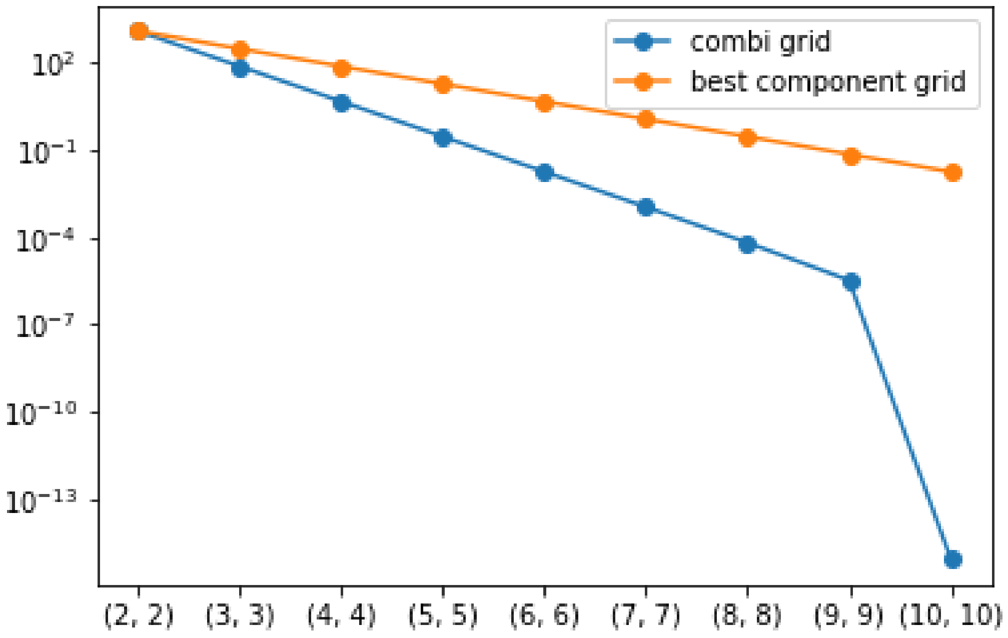

Figure 5.2.: Comparison of the error of the best component grid and the combi grid with a full grid $\Omega_{(10,10)}$

The result of this error analysis is almost as expected. With a higher resolution the results for the combination technique and the best component grid become more precise. Moreover, the solution of the combination technique $u^{(c)}_{(10,10)}$ seems to almost converge to the solution of the full grid. The L2-norm amounts approximately to 1.6025e-16, which is in the order of the machine epsilon (2.2204e-16). However, the difference between the accuracy of the best component grid and the combination technique is remarkable. Usually the combination technique does not improve the results of the component grids in such an intensity. Nevertheless, this seems to be so for this example.

In figure 5.3 the time which is needed for each calculation is plotted. Additionally the time for the full grid of each level is in the plot. At level $(7, 7)$ is the point where the time for solving the full grid intersects with the time for solving the combi grid. At this level the time for solving the max grid exceedes the time for solving the combi grid. With a lower level than that the time for actual combining the grids is higher than the time for solving the full grid. Since the time for solving the full grid rises exponentially with N instead of $log(N)$ (see chapter 3), there is a point where these two graphes intersect. As mentioned above the error of the combination technique is higher than the best component grid used in the combination. This is a prerequisite for using the combination technique as otherwise solving only one of the component grids would be more efficient.

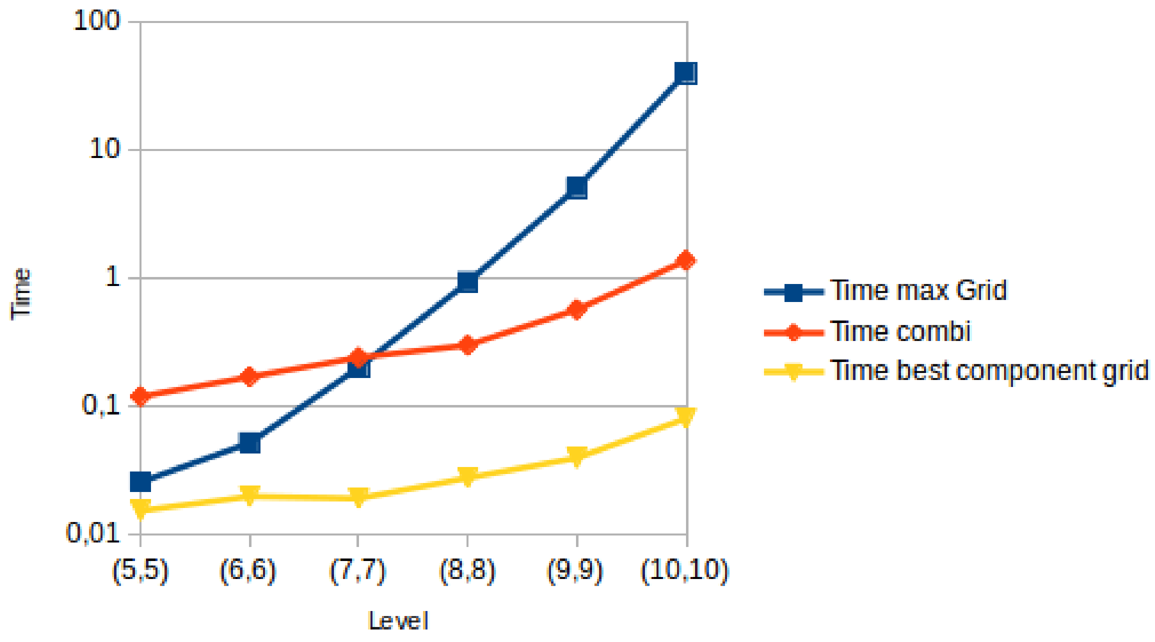

Figure 5.3.: Needed time for calculation of the full, sparse and best component grid

5.2. 3D Poisson Equation

This example solves the Poisson equation in a three dimensional domain. In contrary to the previous chapter a nonlinear Poisson equation will be solved. It is described by the following equations:

$$-\nabla \cdot (q(u)\nabla u) = f \quad in\ \Omega \tag{5.2}$$

$$u = u_D \ on\ boundary\ \partial\Omega \tag{5.3}$$

The variational formulation is given by:[6]

$$F(u;v) = 0 \qquad \forall v \in \hat{V}; \tag{5.4}$$

$$F(u;v) = \int_\Omega (q(u)\nabla u \cdot v - fv)dx; \tag{5.5}$$

$$q(u) = 1 + u^2; \tag{5.6}$$

$$f(x,y,z) = -42x - 84y - 168z - 42; \tag{5.7}$$

$$u_D(x,y,z) = 1 + x + 2y + 4z. \tag{5.8}$$

The result of the above defined equations is the following plot:

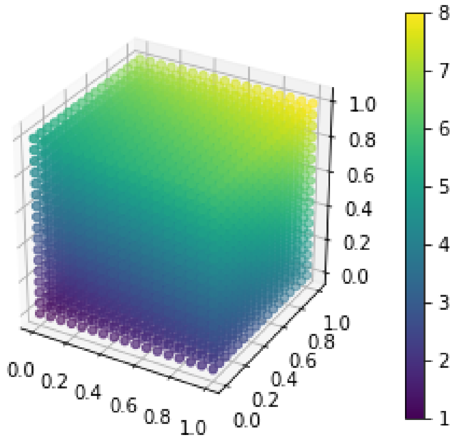

Figure 5.4.: Solving a nonlinear Poisson equation in a three dimensional domain

The error for this example will be analyzed as in the previous section. The full grid for comparison is $\Omega_{(5,5,5)}$ and the truncation parameter $\tau = (0,0,0)$. The resulting plot is shown in figure 5.5. In contrary to the previous chapter, here, the best component grid is better in quality than the combination technique. Additionally the error of the combination technique rises with higher resolutions. A reason for that may be the combination technique's weakness with nonlinear problems. However, the error of the combination grid may vanish for higher resolutions than the ones which are discussed here.

The plot which contains the needed time for calculating the solutions resembles the one from the previous chapter. Above a certain level the full grid time is higher than the combi grid time and the component grid time is logically always lower than the combi grid time. All in all it can be said that it is not reasonable to use the combination technique for this PDE. The best component grid is faster and more precise than the combination technique and therefore the better choice.

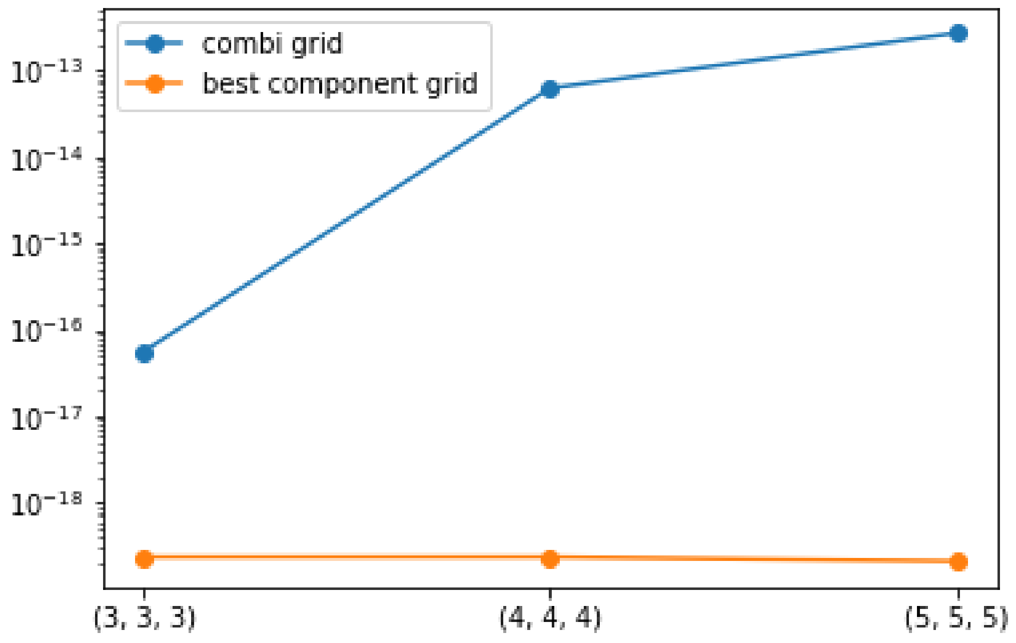

Figure 5.5.: Comparison of the error of the best component grid and the combi grid

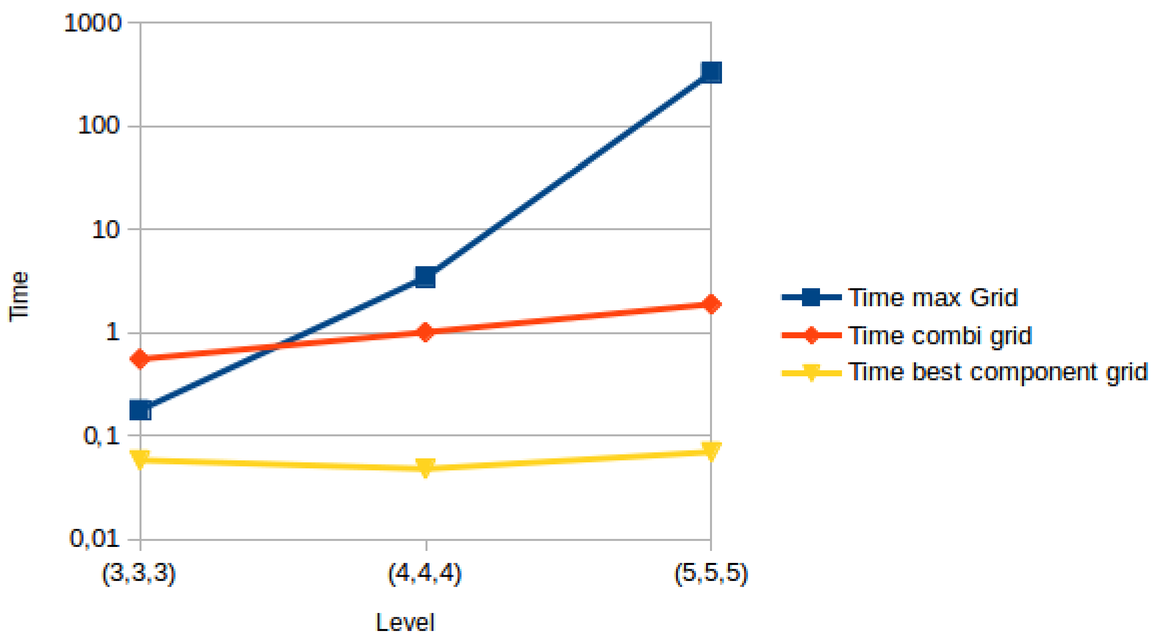

Figure 5.6.: Needed time for calculation of the full, sparse and best component grid

5.3. Diffusion of a Gaussian Hill

In this section the example from section 4.2 and its errors will be examined more thoroughly. Usually in time dependent problems the error rises in each timestep. Nevertheless, exceptions - like diffusion - exist since they can converge to a certain value. For this reason the error is analyzed for three different checkpoints during the simulation. For this analysis the checkpoints which were plotted in section 4.2 were chosen (timestep 0, 20, 40). The errors of these three timesteps are shown in figure 5.7. The errors are determind in the same way as in the previous sections with the full grid $\Omega_{(9,9)}$ and the truncation parameter $\tau = (0, 0)$.

As expected, over time the errors for the combi and the best component grid vanish. Remarkable is the level $\vec{l} = (3, 3)$. At this level the combination technique looses contrary to the expectations its accuracy towards the lower level $\vec{l} = (2, 2)$. Altogether it can be said that the combination technique's accuracy is higher than the one of the best component grid. In due consideration of the time plot (see figure 5.8) and the combination technique's accuracy it can be said that the use of the combination technique in this example is reasonable.

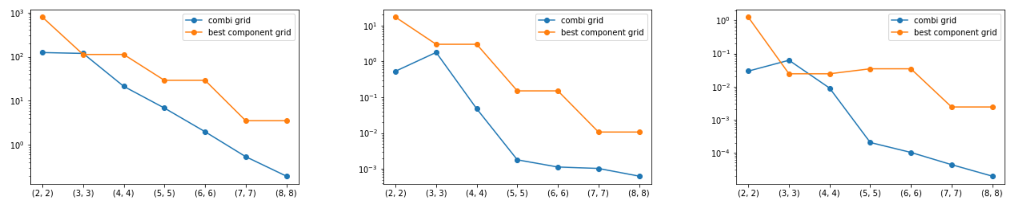

Figure 5.7.: Error of a diffusion of a gaussian hill for timestep 0, 20 and 40

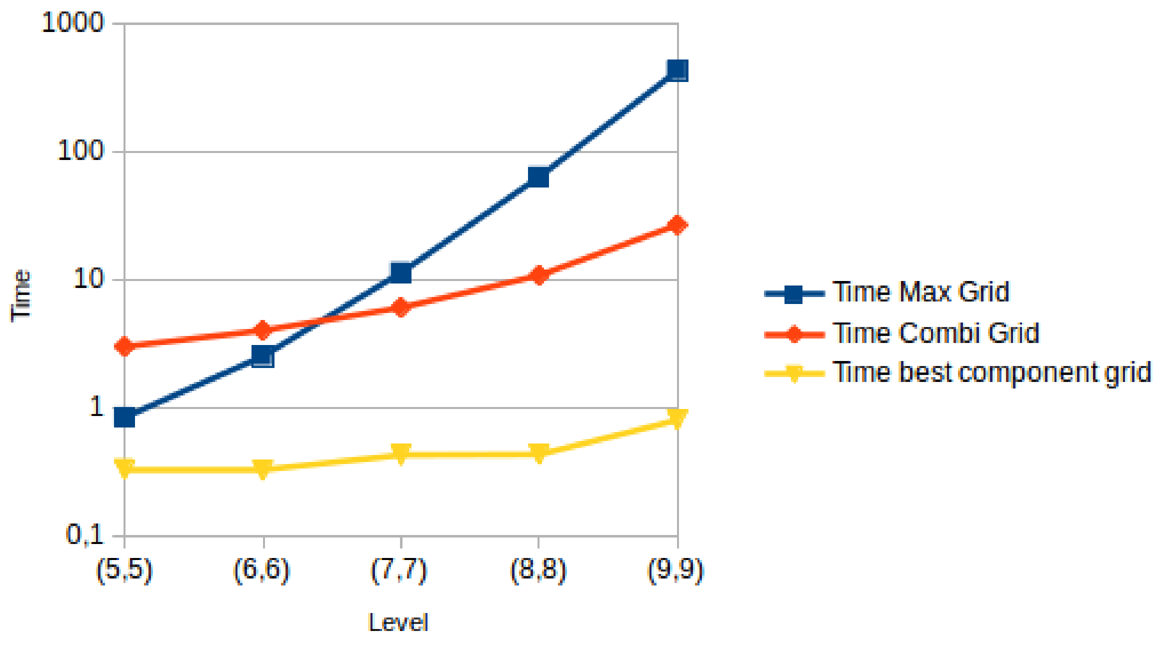

Figure 5.8.: Needed time for calculation of the full, sparse and best component grid

5.4. Linear Elasticity

The erros of the example from section 4.1 will be examined in this chapter. The errors are analyzed as in the previous chapters with the full grid $\Omega_{(5,4,4)}$ and the truncation parameter $\tau = (0, 0, 0)$. The errors for the combi and the best component grid are very similar (see figure 5.9). Yet, the difference between these two grows for the combination technique solution $u^{(c)}_{(5,4,4)}$. This may imply that this difference grows with an higher resolution and therefore the combination technique becomes more precise.

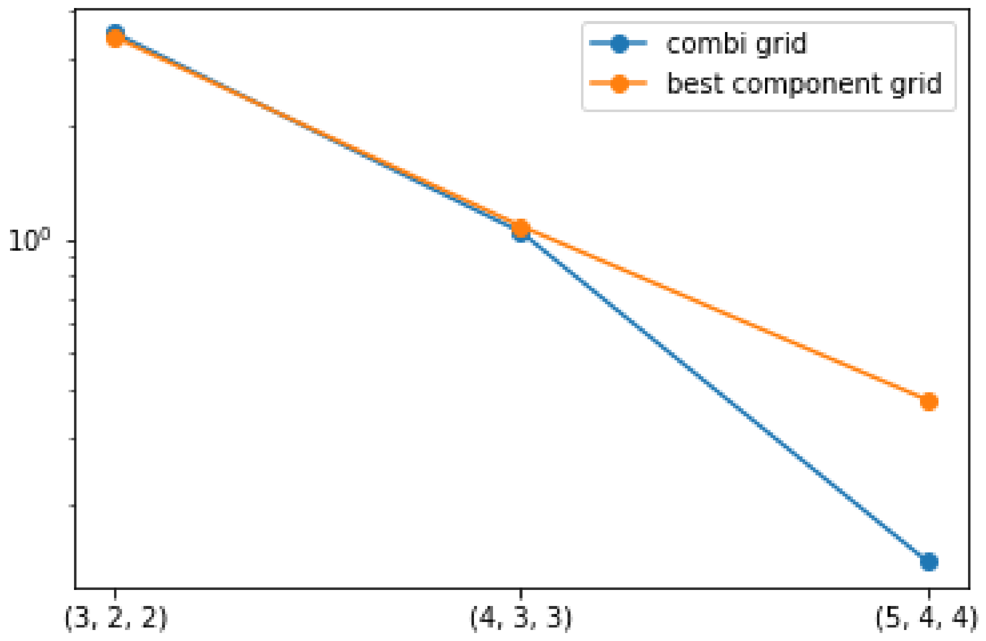

Figure 5.9.: Comparison of the error of the best component grid and the combi grid

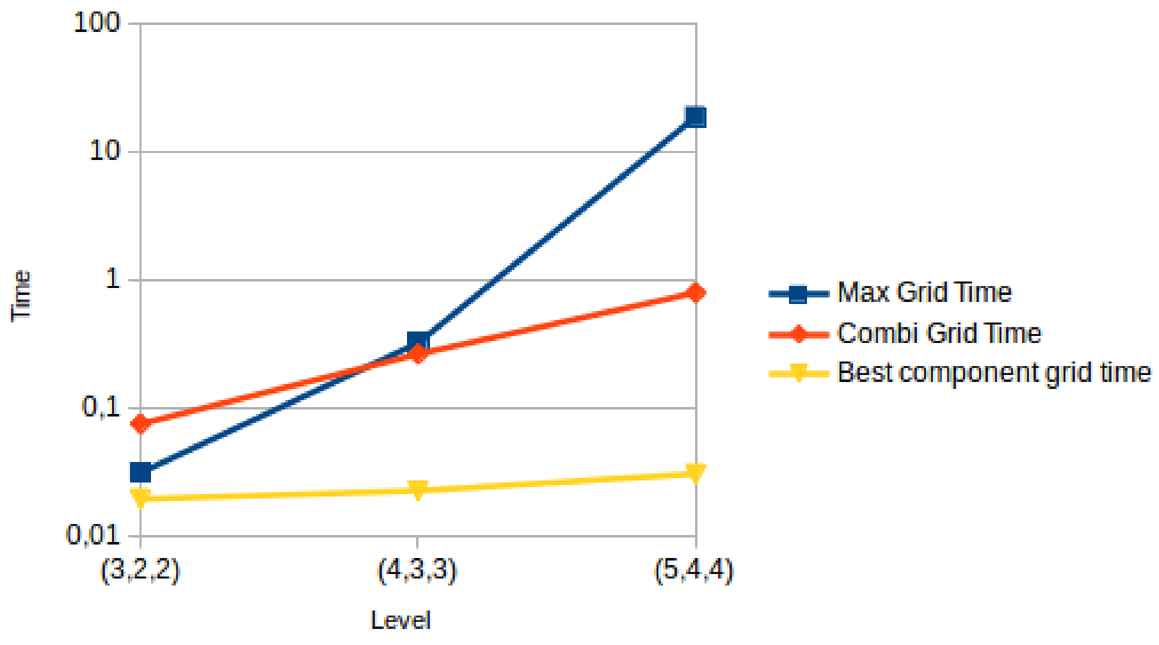

Figure 5.10.: Needed time for calculation of the full, sparse and best component grid

As the time which is needed for calculating the solutions (see figure 5.10) behaves as the one of the previous examples the application of the combination technique seems reasonable for this calculation.

5.5. Navier-Stokes Equation

In order to complete the examples, a time dependent three dimensional PDE needs to be solved. This example will be simulating the flow of an incompressible fluid in a three dimensional cube. The pressure p and the velocity v of the fluid can be determined by using the Navier-Stokes equation. These equations are described by the equations 5.9 - 5.12 with the stress tensor $\sigma(u,p)$, the force per unit volume f, the strain-rate tensor $\epsilon(u)$ and the dynamic viscosity μ.

$$\varrho\left(\frac{\partial u}{\partial t}+u\nabla u\right)=\nabla\cdot\sigma(u,p)+f, \tag{5.9}$$

$$\nabla\cdot u=0, \tag{5.10}$$

$$\sigma(u,p)=2\mu\epsilon(u)-pI, \tag{5.11}$$

$$\epsilon(u)=\frac{1}{2}\left(\nabla u+(\nabla u)^T\right). \tag{5.12}$$

This problem is solved by simulating three different linear problems according to the so-called incremental pressure correction scheme (IPCS). Firstly, the *tentative velocity u^** is calculated as a function of the pressure of the previous time step, afterwards the result of this is used to compute the new pressure and velocity (see equation 5.13)-5.15)[6].

$$\begin{aligned}\langle \varrho \left(u^* - u^n\right)/\Delta t, v\rangle + \langle \varrho u^n \cdot \nabla u^n, v\rangle + \langle \sigma(u^{n+\frac{1}{2}}, p^n), \epsilon(v)\rangle& \\ +\langle p^n n, v\rangle_{\partial\Omega} - \langle \mu \nabla u^{n+\frac{1}{2}} \cdot n, v\rangle_{\partial\Omega} = \langle f^{n+1}, v\rangle,& \qquad (5.13)\\ \langle \nabla p^{n+1}, \nabla q\rangle = \langle \nabla p^n, \nabla q\rangle - \Delta t^{-1}\langle \nabla \cdot u^*, q\rangle,& \qquad (5.14)\\ \langle u^{n+1}, v\rangle = \langle u^*, v\rangle - \Delta t\langle \nabla(p^{n+1} - p^n), v\rangle,& \qquad (5.15)\end{aligned}$$

with

$$\langle v, w\rangle = \int_\Omega vwdx, \qquad (5.16)$$

$$\langle v, w\rangle_{\partial\Omega} = \int_{\partial\Omega} vwds. \qquad (5.17)$$

In figure 5.11 the velocity in x-direction can be seen for the timesteps 1, 20 and 40.

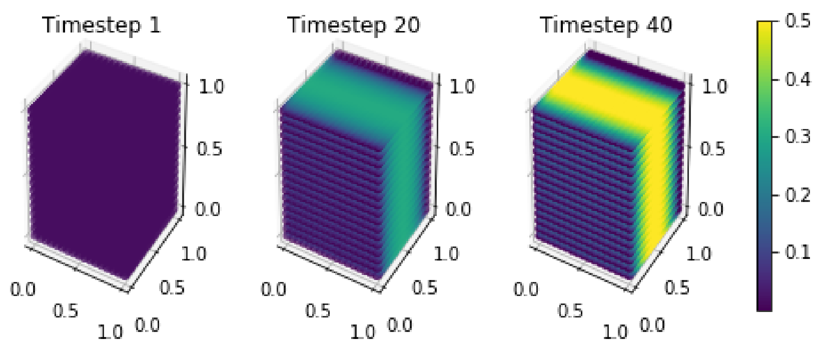

Figure 5.11.: Velocity in x-direction of a incompressible fluid in a three dimensional cube

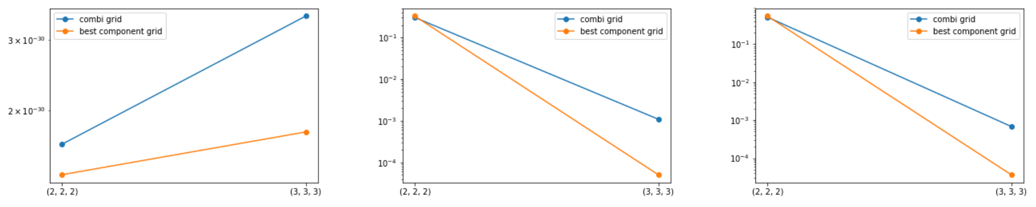

Figure 5.12.: Error of the navier-stokes equation for time step 1, 20 and 40

The errors for the time steps 1, 20 and 40 are analysed as in the previous sections (see figure 5.12). The full grid for calculating the errors is $\Omega_{(3,3,3)}$ and the truncation parameter $\tau = (1, 1, 1)$. However, the first plot can not be considered for this error analysis. All errors of the first time step are smaller than 2e-30 and hence, much smaller than the machine epsilon. The time steps 20 and 40 match the errors of the other examples much better. However, as in section 5.2 the best component grid performs better than the combination technique. This trend may change for higher resolution but could also stay the same. Overall it can be said that for this PDE the use of the combination technique was not rewarding. The best component grid solved the PDE more precise than the combination technique and logically faster as well.

The time which is needed for the calculation(see figure 5.13) resembles all the other results. The only difference is that the combination technique is even faster for the lowest resolution which was analyzed here.

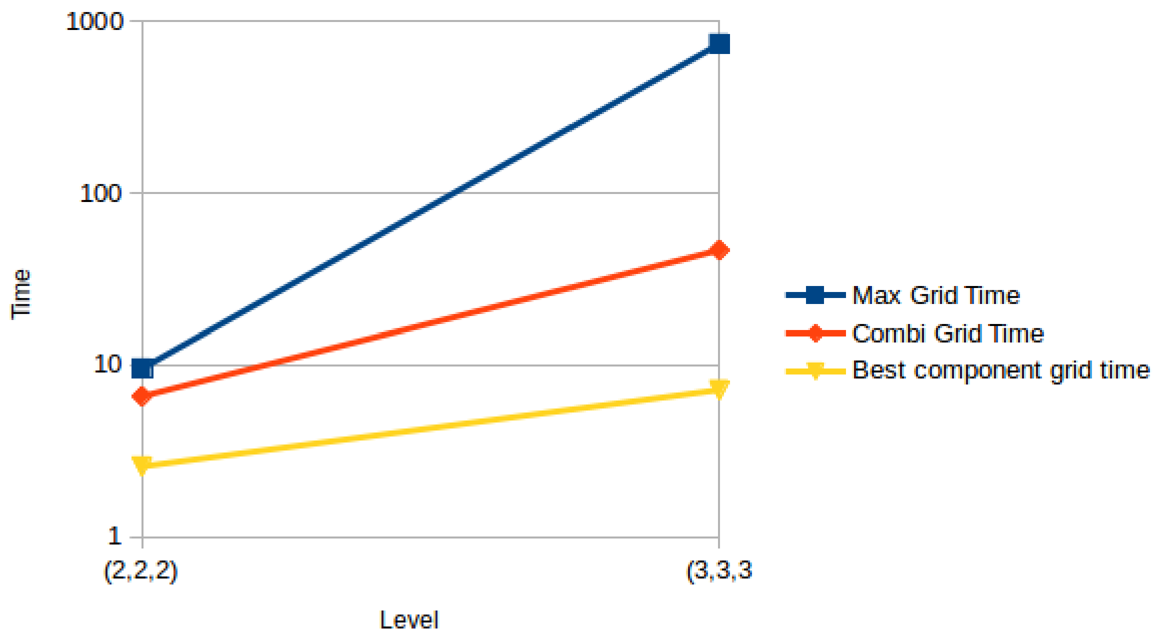

Figure 5.13.: Needed time for calculation of the full, sparse and best component grid

6. Conclusion

In conclusion, it can be said that the combination technique can be a valuable supplement to the FEniCS project. Almost all results of the examples show an improvement of the time which is need for solving the problem and only a rather small inaccuracy. The only PDEs which did not work properly were the nonlinear Poisson equation and the Navier-Stokes equation. However, the combination technique has had problems with nonlinear PDEs in the past anyway. The coupling of the Navier-Stokes equation and combination technique needs to be investigated further to make a conclusive judgement about it. A reason for the weakness of the combination technique could have been the low resolution for which the PDE was solved in this thesis. In contrary to that, it was possible to implement a coupling of the these two technlogies for time and non time dependet PDEs which produced only slightly inaccurate results. Therefore the combination of these two technlogies may provide a fast working solver for PDEs. This thesis can be taken as a model for more tests with the combination technique. By doing that the application areas of the combination technique can be narrowed down more exact.

Despite that, this thesis does not provide a full optimized framework for the coupling of these two technologies; furthermore, even limits are set for it. So far, FEniCS only provides meshes with up to three dimensions and the combination technique becomes the more useful the higher the dimension of the domain is. Additionally, FEniCS makes it easy to solve PDEs with complicated meshes, but the combination technique needs regular grids in order to work properly. Nevertheless, this obstacle could be broken through by extrapolating the results of the irregular grids on regular grids. Moreover, the solving of the component grids could be parallelised as well. The parallelisation of the combination technique can be found in further reading[5] which can be used as an example. Lastly, this thesis can be seen as a starting point for a further and more deeper coupling of the FEniCS project and the combination technique. Furthermore, it could be used as an example for future coupling of general purpose python PDE solvers and the combination technique.

Appendix

A. Implementation of the DummyGridArbitraryDim class

```
class DummyGridArbitraryDim(combiGrid):


    __metaclass__ =ABCMeta

    def __init__(self, levelvector, boundaries):
        combiGrid.__init__(self,levelvector,boundaries)
        self.surplus = None
        if self.boundaries!=/
        tuple([True for __ in xrange(len(self.levelvec))]):
            raise NotImplementedError/
            ('Dummy %dd just for grids with boundary'% len(self.levelvec))
        self.log.debug('Created %dd combigrid dummy' % len(self.levelvec))

    def getDim(self):
        return len(self.levelvec)

    def fillData(self,f=None,sdcFactor=0):
        if f is None:
            self.data = zeros(self.gridSizes)
            s= self.data.shape
            temp = [linspace(0,1.0,s[i]) /
            for i in xrange(len(self.levelvec))]
            grid = meshgrid(*temp,indexing='ij')
            self.data+=grid[0]
        elif isinstance(f,types.FunctionType):
            self.data = f(self.levelvec)
            if sdcFactor != 0:
                self.addSDC(sdcFactor)
        else:
            self.data = f

    def extrapolateData(self,level):
        buf = self.getData()
        cur = self.getGridSizes()
        new = tuple(map(float,[(2**i+1) for i in level]))
        fac = divide(new,cur)
        ret = zoom(buf,fac,order=1)
        return ret

    def getData(self):
        return self.data

    def getSurplus(self):
        return self.surplus

    def getLevelvector(self):
        return self.levelvec

    def getGridSizes(self):
        return self.gridSizes
```

B. Implementation of the Helper Functions

```python
import itertools as it
import numpy as np
from scipy.ndimage import zoom

def hierSubs_self(lmin,lmax):
        list_ranges = [range(lmin[i],lmax[i]+1) /
        for i in range(max(len(lmin),len(lmax)))]
        return it.product(*list_ranges)

def get_coordinates_for_plot(m):
    co_ord = m.coordinates()
    size = m.geometry().dim(),len(co_ord)
    ret = np.zeros(size)
    i = 0
    for xyz in co_ord:
        for j in range(size[0]):
            ret[j][i] = xyz[j]
        i+=1
    return ret.tolist()

def get_data_for_plot(combi):
    return combi

def compute_l1_error(a,b):
    if a.shape!=b.shape:
        print "shape"
        return -1
    if len(a.shape)!=len(b.shape)!=1:
        print "len"
        return -1
    ret = 0
    for x,y in zip(a,b):
        ret+=abs(x-y)
    return ret
```

```

def compute_l2_error(a,b):
    if a.shape!=b.shape:
        print "shape"
        return -1
    if len(a.shape)!=len(b.shape)!=1:
        print "len"
        return -1
    ret = 0
    for x,y in zip(a,b):
        ret+= ((x-y)*(x-y))
    return ret

def convert_dof(data,dof):
    ret = np.zeros(np.array(data).shape)
    for i in range(len(data)):
        ret[dof[i]]=data[i]
    return ret

def erg_is_zero(m):
    ret = True
    for i in m:
        if not np.isclose(i,0):
            ret = False
            break
    return ret

def extrapolateForError(buf,level):
    old_shape = np.array(buf.shape,dtype=np.float)
    new_shape = 2**np.array(level)+1
    fac = np.divide(new_shape,old_shape)
    ret = zoom(buf,fac,order=1)
    return ret
```

Bibliography

[1] The fenics tutorial volume i. `https://fenicsproject.org/pub/tutorial/sphinx1/`. Accessed: 10th April 2018.

[2] Richard Bellmann. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[3] Hans-Joachim Bungartz and Michael Griebel. *Sparse grids*. Cambridge University Press, 2004.

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

[5] Mario Heene. *A massively parallel combinatino technique for the solution of high-dimensional PDEs*. PhD thesis, Institut für Parallele und Verteilte Systeme der Universität Stuttgart, 2017.

[6] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Springer, 2017.

[7] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[8] Michael Obersteiner, Alfredo Parra Hinojosa, Mario Heene, Hans-Joachim Bungartz, and Dirk Pflüger. A highly scalable, algorithm-based fault-tolerant solver for gyrokinetic plasma simulations. In *Proceedings of ScalA 17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA 17)*, ACM, New York, NY, USA. 8 pages. https://doi.org/10.1145/3148226.3148229.