FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Specification and Partitioning of Computational Domains for Generated Geometric Multigrid Solvers

Jeremias Isnardy

Master Thesis

Specification and Partitioning of Computational Domains for Generated Geometric Multigrid Solvers

Jeremias Isnardy
Master Thesis

Aufgabensteller:	PD Dr.-Ing. habil. Harald Köstler
Betreuer:	Sebastian Kuckuk, Harald Köstler
Bearbeitungszeitraum:	01.2015 – 07.2015

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Master Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 7. September 2015 .....

Contents

Listings

List of Figures

List of Tables

1 Introduction and Motivation

A lot of mathematical problems of a high complexity exist in the field of engineering, which can not be solved analytically but numerically, for example partial differential equations (PDE). Mathematical skills are mandatory, to use a numerical approach of solving them, the same as skills in computer science to implement this approach. The knowledge, mediated by each of this fields, is often barely sufficient to understand the problem completely and to implement a solution method. Projects as ExaStencils (Advanced Stencil-Code Engineering) emerge for that reason. With it, it is possible to define a mathematical problem and to get an approximated solution by an automated code generation. By this means engineers or mathematicians can get simulation results without knowing anything about programming.

This thesis wants to contribute to ExaStencils. It deals with specifications and partitioning of computational domains so this project is able to solve PDEs on a more complex structure. As matters stand, only computation domains of the structure unit square (2D) or unit cube (3D) are possible. The goal of this thesis is to extend this assortment taking account of mathematical limitations.

2 Mathematical Background

ExaStencil uses different approaches to generate a solution of a defined mathematical problem. This chapter provides a short overview of the ones, which had been used during the work on this thesis.

2.1 Finite Difference Method

The finite difference discretization method is be used to solve partial differential equations in particular. It is one of the simplest forms of discretization methods.

2.1.1 Basic Principle

The basic principle is to approximate derivatives of a function by linear combinations of function values.

$$u'(x) = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h} \tag{2.1}$$

Based on this principle certain differentation formulas can be expressed for first derivatives, assuming the function u gets discretized on a grid at grid point i with a constant mesh size Δx.

$$\text{forward difference scheme:} \qquad \left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1} - u_i}{\Delta x} \tag{2.2a}$$

$$\text{backward difference scheme:} \qquad \left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_i - u_{i-1}}{\Delta x} \tag{2.2b}$$

$$\text{central difference scheme:} \qquad \left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x} \tag{2.2c}$$

Analogously, a scheme for the second derivative can be derived.

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} \tag{2.3}$$

2.1.2 Example: Poisson Equation

The practical use of the finite differences gets demonstrated by discretizing the 2D Poisson's equation:

$$-\Delta u = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{2.4}$$

Using equation 2.3 and assuming a constant mesh size h in each direction, equation 2.4 can be approximated at the grid point i,j as follows:

$$(-\Delta u)_{i,j} = -\frac{1}{h^2}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}) = f_{i,j} \tag{2.5}$$

This result can be written in a sparse linear system of equations.

$$A^h u^h = f^h \tag{2.6}$$

With discretization matrix $A^h \in \mathbb{R}^{NxN}$, vector of unknowns $u^h \in \mathbb{R}^N$ and the right hand side (RHS) vector $f^h \in \mathbb{R}^N$. N denotes the number of unknowns. The discretization matrix in this case would look like this:

$$A = \frac{1}{h^2}\begin{bmatrix} 4 & -1 & 0 & 0 & 0 & ... & 0 \\ -1 & 4 & -1 & 0 & 0 & ... & 0 \\ 0 & -1 & 4 & -1 & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & -1 & 4 & -1 & 0 \\ 0 & ... & ... & 0 & -1 & 4 & -1 \\ 0 & ... & ... & ... & 0 & -1 & 4 \end{bmatrix} \tag{2.7}$$

For reasons of efficiency A can be expressed and stored as a so called stencil. In this situation a 5-point stencil of the form:

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} \tag{2.8}$$

2.2 Finite Volume Method

Using more complex shapes of domains lead to the problem of finite differences, that it is getting hard to implement correctly. The stencil (2.8) can not be applied anymore. As long the grid is still a rectilinear one, the step sizes could be adjusted accordingly to the given position. But when the generated grid is a more complex structured one, another approach needs to be used. This could be the finite volume method. It discretizes the governing equation in integral form, in contrast to the finite differences method, which is applied to the governing equation in differential form.

Starting again from the Poisson's equation 2.4 and applying the integral form and using Green's theorem leads to:

$$-\int_{ABCD} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} dxdy = -\int_{ABCD} (\frac{\partial u}{\partial x} dy - \frac{\partial u}{\partial y} dx) = f(x,y) \tag{2.9}$$

This can now be discretized by:

$$
\begin{aligned}
-\frac{1}{S_{ABCD}}\Big(&\left[\frac{\partial u}{\partial x}\right]_{i,j-\frac{1}{2}} \Delta y_{AB} - \left[\frac{\partial u}{\partial y}\right]_{i,j-\frac{1}{2}} \Delta x_{AB} \\
&+ \left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2},j} \Delta y_{BC} - \left[\frac{\partial u}{\partial y}\right]_{i+\frac{1}{2},j} \Delta x_{BC} \\
&+ \left[\frac{\partial u}{\partial x}\right]_{i,j+\frac{1}{2}} \Delta y_{CD} - \left[\frac{\partial u}{\partial y}\right]_{i,j+\frac{1}{2}} \Delta x_{CD} \\
&+ \left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2},j} \Delta y_{DA} - \left[\frac{\partial u}{\partial y}\right]_{i-\frac{1}{2},j} \Delta x_{DA}\Big)
\end{aligned}
\tag{2.10}
$$

The notation $ABCD$ describes the control volume. Using a cell-centered finite volume method, this looks as follows:

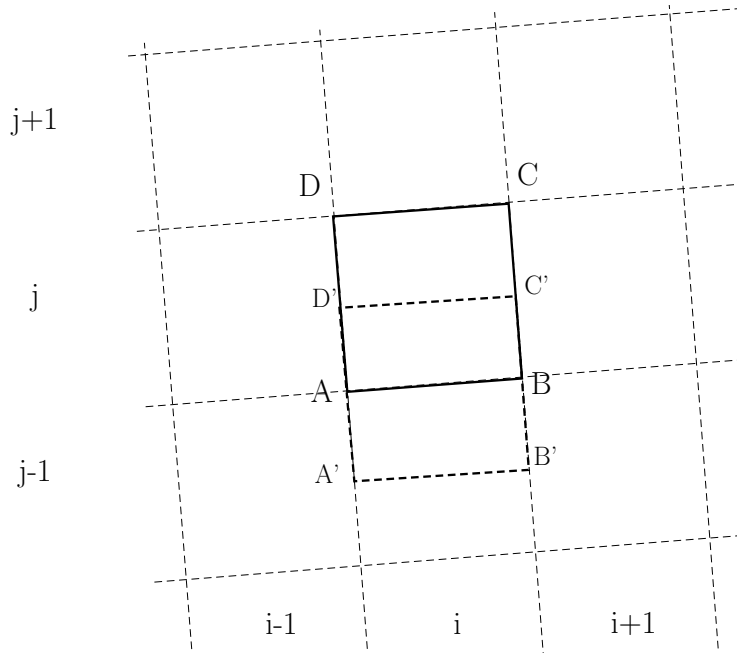


Figure 2.1: Concept of control volume

The cell (i, j) gets framed by the points A, B, C and D, which are creating a control volume for this cell. For the calculation of the terms of equation 2.10 the evaluation of e.g. $\frac{\partial u}{\partial x}$ at the position $(i, j-\frac{1}{2})$ are required. This can be done by evaluating this term my the mean value over the area $A'B'C'D'$, also illustrated in figure 2.1.

$$
\left[\frac{\partial u}{\partial x}\right]_{i,j-\frac{1}{2}} = \left(\frac{1}{S_{A'B'C'D'}}\right) \iint \left(\frac{\partial u}{\partial x}\right) dxdy = \left(\frac{1}{S_{A'B'C'D'}}\right) \oint\limits_{A'B'C'D'} u dy \tag{2.11a}
$$

$$
\left[\frac{\partial u}{\partial y}\right]_{i,j-\frac{1}{2}} = \left(\frac{1}{S_{A'B'C'D'}}\right) \iint \left(\frac{\partial u}{\partial y}\right) dxdy = -\left(\frac{1}{S_{A'B'C'D'}}\right) \oint\limits_{A'B'C'D'} u dx \tag{2.11b}
$$

Again using Green's theorem:

$$\oint_{A'B'C'D'} udy = u_{i,j-1}\Delta y_{A'B'} + u_B \Delta y_{B'C'} + u_{i,j}\Delta y_{C'D'} + u_A \Delta y_{D'A'} \tag{2.12a}$$

$$\oint_{A'B'C'D'} udx = u_{i,j-1}\Delta x_{A'B'} + u_B \Delta x_{B'C'} + u_{i,j}\Delta x_{C'D'} + u_A \Delta x_{D'A'} \tag{2.12b}$$

If the distortion of the mesh is not too big, the following relations ca be assumed:

$$\Delta y_{A'B'} \approx -\Delta y_{C'D'} \approx \Delta y_{A,B} \tag{2.13a}$$

$$\Delta y_{B'C'} \approx -\Delta y_{D'A'} \approx \Delta y_{i(j-1,j)} \tag{2.13b}$$

$$\Delta x_{A'B'} \approx -\Delta x_{C'D'} \approx \Delta x_{A,B} \tag{2.13c}$$

$$\Delta x_{B'C'} \approx -\Delta x_{D'A'} \approx \Delta x_{i(j-1,j)} \tag{2.13d}$$

$$S_{A'B'C'D'} = S_{A,B} = |\det\left(\vec{A'B'} \;\; \vec{B'C'}\right)| = \Delta x_{AB}\Delta y_{i(j-1,j)} - \Delta y_{AB}\Delta x_{i(j-1,j)} \tag{2.13e}$$

Using equations 2.12 and 2.13 the first terms of equation 2.10 can be expressed as:

$$\begin{aligned}\left[\frac{\partial u}{\partial x}\right]_{i,j-\frac{1}{2}} \Delta y_{AB} - \left[\frac{\partial u}{\partial y}\right]_{i,j-\frac{1}{2}} \Delta x_{AB} = \frac{\left(\Delta x_{AB}^2 + \Delta y_{AB}^2\right)\left(u_{i,j-1} - u_{i,j}\right)}{S_{AB}} \\ + \frac{\left(\Delta x_{AB}\Delta x_{i(j-1,j)} + \Delta y_{AB}\Delta y_{i(j-1,j)}\right)\left(u_B - u_A\right)}{S_{AB}}\end{aligned} \tag{2.14}$$

This steps can be done analogously for all the other terms of equation 2.10. This leads to:

$$\begin{aligned}M_{AB}\left(u_{i,j-1} - u_{i,j}\right) + N_{AB}\left(u_B - u_A\right) + M_{BC}\left(u_{i+1,j} - u_{i,j}\right) + N_{BC}\left(u_C - u_B\right) \\ + M_{CD}\left(u_{i,j+1} - u_{i,j}\right) + N_{CD}\left(u_D - u_C\right) + M_{DA}\left(u_{i-1,j} - u_{i,j}\right) + N_{DA}\left(u_A - u_D\right) = f\end{aligned} \tag{2.15}$$

With the geometrical paramaters:

$$M_{AB} = \left(\Delta x_{AB}^2 + \Delta y_{AB}^2\right)/S_{AB} \quad N_{AB} = \left(\Delta x_{AB}\Delta x_{i(j-1,j)} + \Delta y_{AB}\Delta y_{i(j-1,j)}\right)/S_{AB} \tag{2.16a}$$

$$M_{BC} = \left(\Delta x_{BC}^2 + \Delta y_{BC}^2\right)/S_{BC} \quad N_{BC} = \left(\Delta x_{BC}\Delta x_{(i+1,i)j} + \Delta y_{BC}\Delta y_{(i+1,i)j}\right)/S_{BC} \tag{2.16b}$$

$$M_{CD} = \left(\Delta x_{CD}^2 + \Delta y_{CD}^2\right)/S_{CD} \quad N_{CD} = \left(\Delta x_{CD}\Delta x_{i(j+1,j)} + \Delta y_{CD}\Delta y_{i(j+1,j)}\right)/S_{CD} \tag{2.16c}$$

$$M_{DA} = \left(\Delta x_{DA}^2 + \Delta y_{DA}^2\right)/S_{DA} \quad N_{DA} = \left(\Delta x_{DA}\Delta x_{(i-1,i)j} + \Delta y_{DA}\Delta y_{(i-1,i)j}\right)/S_{DA} \tag{2.16d}$$

Additionally, u_A,u_B,u_C and u_D need to be evaluated. This can be done by interpolating them between their surrounding values, exemplarily done for u_A:

$$u_A = 0.25\,(u_{i,j} + u_{i-1,j} + u_{i-1,j} + u_{i,j-1}) \tag{2.17}$$

Inserting them into equation 2.15 leads to a nine-point discretization. This can be again expresses in stencil form:

$$\frac{1}{S_{ABCD}} \begin{bmatrix} 0.25\,(N_{DA} - NCD) & -M_{CD} + 0.25\,(N_{DA} - N_{BC}) & 0.25\,(N_{CD} - N_{BC}) \\ -M_{DA} + 0.25\,(N_{AB} - N_{CD}) & M_{AB} + M_{BC} + M_{CD} + M_{DA} & -M_{BC} + 0.25\,(N_{CD} - N_{AB}) \\ 0.25\,(N_{AB} - NDA) & -M_{AB} + 0.25\,(N_{BC} - N_{DA}) & 0.25\,(N_{BC} - N_{AB}) \end{bmatrix} \tag{2.18}$$

When using a uniformly rectangular grid, this equations reduces to the form of equation 2.5.

2.3 Multigrid

To solve linear system of equations of the form shown in equation 2.6 certain iterative solvers can be used, as Jacobi iteration method or Gauss-Seidel iteration method. These iterative methods are often referred to as relaxation methods, which first guess an initial value for the solution and relax towards the true value of it, by reducing the error in each iteration step. These approaches have some disadvantages. On the one hand, the convergence rate depends on the mesh size and on the other hand they are having difficulties smoothing both, high-frequency and low-frequency parts of the error, at the same time. The essential multigrid principle is to approximate the smooth (long wavelength) part of the error on coarser grids (coarse grid principle). The non-smooth part gets reduced by a small number of iterations with these basic iteration methods as Jacobi or Gauss-Seidel on the fine grid (smoothing principle).[HCS+15][P91]

2.3.1 Restriction and Prolongation

The following figure shows the different versions of a grid, depending on the current multigrid level.

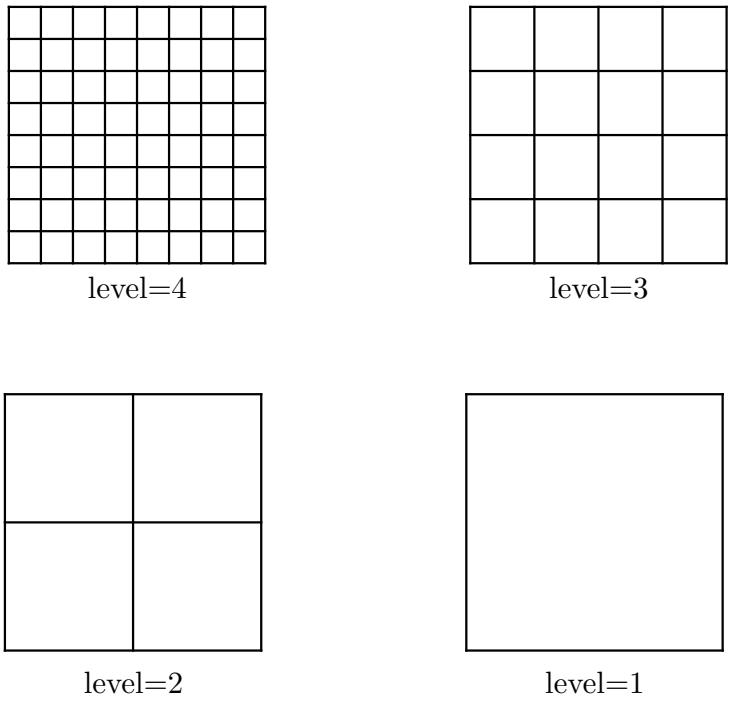


Figure 2.2: Different multigrid level of a mesh

For the transfer from a finer grid level (as level 2 in figure 2.2) to a coarser one (level 1) a linear operator $\mathcal{R} = \mathcal{I}_h^H$ gets defined, representing the restriction. H denotes the coarser grid, h the finer one. When using a cell based grid R can be expressed as a stencil, depending on the fact, which grid points should be used for defining the value of the coarser grid cell. A possible stencil could look like this:

$$\begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix} \tag{2.19}$$

The interpolation, respectively the prolongation from the coarser grid to the finer with the linear operator $\mathcal{P}$ can also be written in stencil form. The simplest one would just use the value of the coarser grid cell to set all the values of the finer grid cells. Written in stencil form it would look as follows:

$$\begin{bmatrix} 1 \end{bmatrix} \tag{2.20}$$

2.3.2 Algorithm

The general multigrid algorithm can be expressed as follows:

Listing 2.1: Multigrid Algorithm

```
1  if level == 1 then                          //coarsest grid level
2          solve A_h u_h = f_h by (parallel) direct solver or by CG iterations
3  else
4          u_h^k = S^{ν1}(u_h^k, A_h, f_h, ν1)  // presmoothing
5          r_h = f_h − A_h · u_h^k              // compute residual
6          r_H = R · r_h                        // restrict residual
7          for i=1 to μ by step 1
8                  A_H · e_H = r_H              // get error
9          e_h = P · e_H                        // prolongate error
10         ũ_h^k = u_h^k + e_h                  // coarse grid correction
11         u_h^{k+1} = S^{ν2}(ũ_h^k, A_h, f_h, ν2) //postsmoothing
12 end if
```

Two successive grid levels are denoted by Ω_h and Ω_H and the following components are used in the multigrid solver:

- ν_1 pre- and ν_2 postsmoothing steps of S^{ν_1} and S^{ν_2}, both representing a smoother as Jacobi or Gauss-Seidel
- restriction operator $\mathcal{R}$ and prolongation operator $\mathcal{P}$
- iteration step k
- multigrid parameter μ

The parameter μ influences the type of the algorithm, which can be distinguished by the so-called *V-cycle* ($\mu = 1$) and *W-cycle* ($\mu = 2$). Assuming the maximal multigrid level would be 3, the cycles would look as follows:

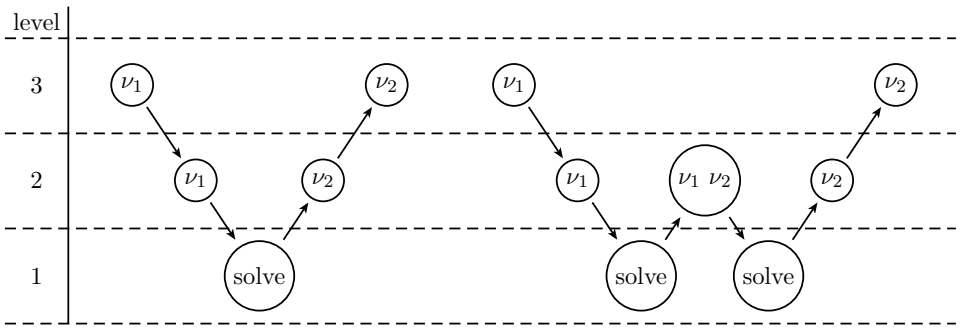


Figure 2.3: V-Cycle and W-Cycle of Multigrid algorithm

3 ExaStencils

ExaStencils is a software project developed by the universities of Passau, Wuppertal and Erlangen–Nuremberg. It is narrowed to a small but very important application domain. The goal is to allow a simple specification of a mathematical problem and by adding some specific domain and hardware parameters to achieve a C++ code, which is able to approximate a solution of the problem. The application domain chosen is that of stencil codes, i.e., compute-intensive algorithms in which data points in a grid are redefined repeatedly as a combination of the values of neighboring points. The neighborhood pattern used is called a stencil. ExaStencils can be seen as a so called domain-specific language (DSL). An additional goal is to achieve an exascale performance, so the name of the project results in ExaStencils.[Exa]

3.1 Workflow

As mentioned above the handling of ExaStencils should be as simple as possible. The following figure 3.1 illustrates the workflow how to achieve it.

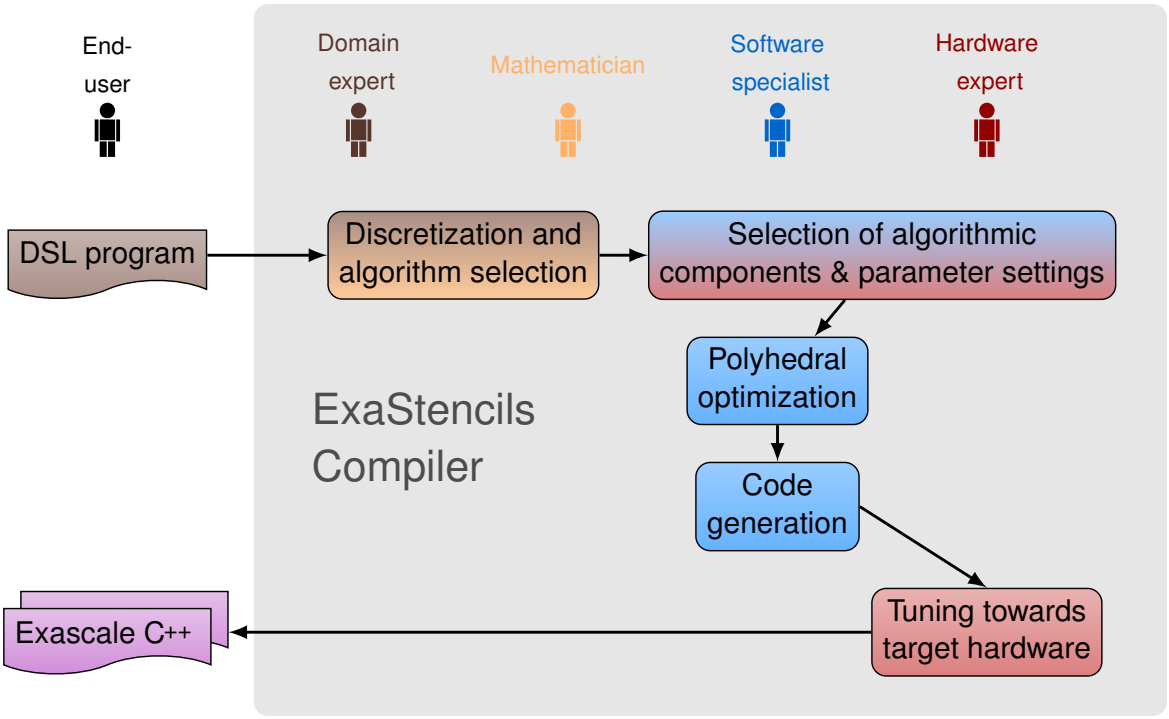


Figure 3.1: Workflow of ExaStencils programming paradigm [LAB+14]

The end-user gets a software, which takes a certain amount of specialized input, as the exact PDE, the computational domain, some settings regarding multigrid, some hardware settings etc. The ExaStencils compiler converts this input to a C++ code, which can be used for instance on a cluster. It does so by discretizating the PDE, choosing and adjusting an appropriate algorithm, optimizing it and tuning it towards the chosen hardware. For all these steps some experts are necessary. One for the given physical domain, who has an idea what is important in this domain. One for the mathematics, choosing the right numerical method, one for the software, implementing the algorithm correctly and one, who optimizes the code regarding hardware concerns. For a more detailed description of the workflow see [LAB+14].

3.2 Layer

The implementation of ExaStencils can be divided logically into four layers, each representing an own domain specific language (DSL) and an additional hardware description. The concept is shown in the following figure.

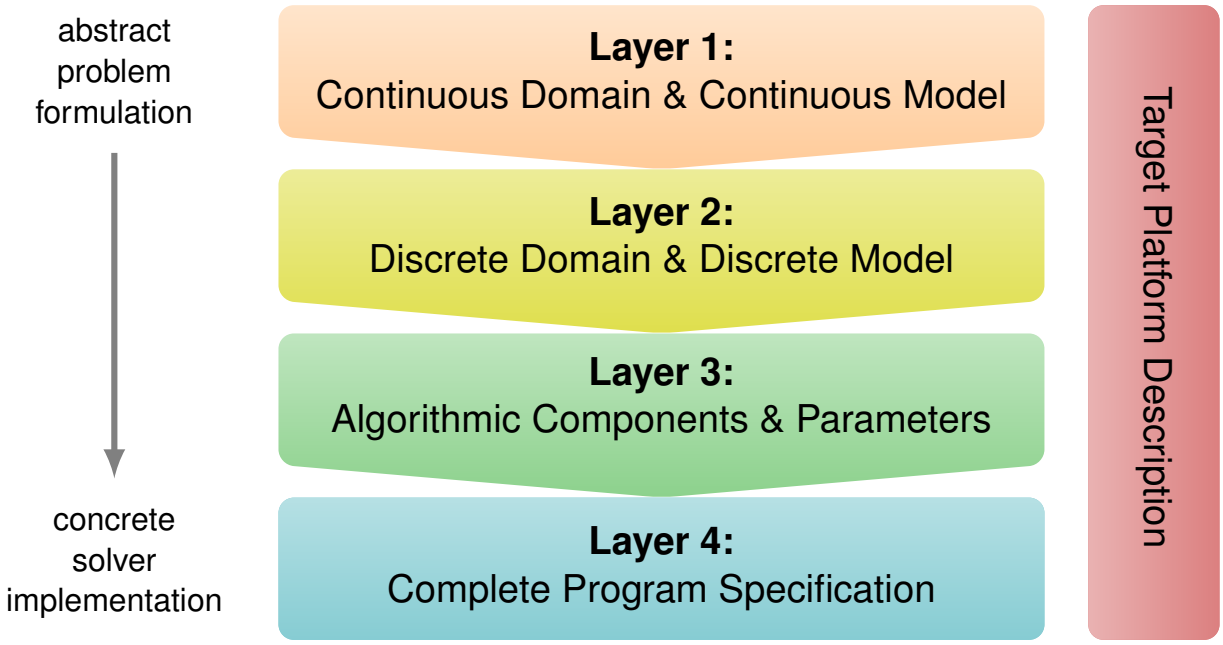


Figure 3.2: Layer of ExaStencils [SKH+14b]

Layer 1 is the most abstract layer concerning the choice of the algorithms and the implementation. It describes the mathematical domain and model of the problem, which needs to be solved. Right up to layer 4 the level of a more concrete treatment of the problem increases. Layer 2 takes care of the discretization, layer 3 of the mathematical method and layer 4 arranges the code elements. There also exists an additional layer describing the hardware, which influences in some circumstances all the other layers, but has no direct part in the logical procedure.

Each layer can be controlled by a specially built DSL file, which will be later on created completely by the layers themselves. So each layer can be developed independently. In the following each layer will be described with some examples, based on [HCS+15].

3.2.1 Layer 1 – Continuous Domain and Model

The following Poisson's equation is the one which shall be solved.

$$\Delta u = f \quad \text{in } \Omega \tag{3.1a}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{3.1b}$$

Assuming homogeneous Dirichlet boundary conditions, the domain $\Omega = [0,1]^2$, the right hand sided function $f = 0$ and mesh size of 2^{-12}, the DSL of layer 1 would look like the following listing:

Listing 3.1: DSL example of layer 1

```
1
2  Domain d = [0,1] x [0,1]
3
4  Function f = 0
5  Unknown solution = initrandom
6  Operator Lapl = Laplacian
7
8  PDE pde{ Lapl(solution) = f }
9  PDEBC bc { solution = 0 }
10
11 Accuracy = 12
```

Having these information all other layers can be generated by adding some internal domain knowledge. However adapting them, after they have been created, is still possible.

3.2.2 Layer 2 – Discrete Domain and Model

When equation 3.1 gets discretized by finite differences as described in section 2.1, the DSL description is:

Listing 3.2: DSL example of layer 2

```
1
2  Fragment f1 = Regular_Square
3
4  Discrete_Domain d levels 12 {
```

```
  xsize  = 4096
  xoarsefac  = 2
  ysize  = 4096
  ycoarsefac  = 2
}

field <Double,1>@nodes f
field <Double,1>@nodes solution
stencil <Double,FD,2>@nodes Lapl
```

Based on some internal domain knowledge values and settings some parameters had been chosen here. The computational domain is partitioned in so called fragments, each basically representing a cell in the grid on the coarsest multigrid level. In combination with the accuracy of layer 1 the discrete domain can be defined. For the function f and the solution fields, double precision floating point numbers had been chosen and a stencil field, additionally marked with finite differences, got created.

3.2.3 Layer 3 – Algorithmic Components and Parameters

Despite the fact that ExaStencils is currently restricted to the multigrid algorithm (see section 2.3), there exist some parameter which need to be adjusted, e.g. the smoothing property, which could be Gauss-Seidel (GS).

Listing 3.3: DSL example of layer 3

```
mgcomponents {
    smoother = GaussSeidel
    interpolation   = interpolatecorr
    restriction     = Restrict
    coarsesolver    = GaussSeidel
    cycle     = VCycle
}

mgparameter {
    nlevel       = 7
    rest_order   = 2
    int_order    = 2
    ncoarse      = 10
    nprae    = 2
    npost    = 1
    iters    = 10
    omega    = 1.0
}
```

The parameter of *mgcomponents* are defining all the important steps of the algorithm. The assigned names correspond to them on layer 4. The *mgparameter* setup defines the behavior of the algorithm in detail, the number of multigrid levels, order of restriction and interpolation operator, number of coarse grid solver iterations, pre-smoothing and post-smoothing steps, maximum number of V-cycles and a smoothing parameter.

3.2.4 Layer 4 – Complete Program Specification

Having the generated code of the previous layers it is now possible to link modules from external frameworks, add and change algorithms or formulate own functions and classes in a kind of pseudo code, but quite similar to Scala syntax. Even using local variables, statements, expressions, specialized loops and simple I/O like printing is possible. For the example of the subsections above all multigrid components were generated automatically using internal domain knowledge. The following listing shows the main application, V-Cycle function, smoother and the transfer operator functions.

Listing 3.4: DSL example of layer 4

```
1
2  def cpu Application ( ) : Unit
3  {
4      decl res0 : Double = L2Residual ( 0 )
5      decl res : Double = res0
6      decl resold : Double = 0
7      print ( 'startingres' res0 )
8      repeat up 10
9          resold = res
10         VCycle ( 0 )
11         res = L2Residual ( 0 )
12         print ( 'Residual:' res 'residual reduction:' (res0/res) )
13     next
14 }
15
16 def cpu VCycle ( lev:Int ) : Unit
17 {
18     if coarsestlevel {
19         repeat up ncoarse
20             GaussSeidel ( lev)
21         next
22     } else {
23         repeat up nprae
```

```
24              GaussSeidel( lev)
25          next
26      Residual ( lev )
27      Restrict ( (lev+1) f[(lev+1)] Res[lev])
28      set( (lev+1) solution[(lev+1)] 0)
29      VCycle (lev+1)
30      interpolatecorr( lev solution[lev] solution[(lev+1)] )
31          repeat up npost
32              GaussSeidel ( lev )
33          next
34      }
35  }
36
37  def cpu GaussSeidel ( lev:Int ) : Unit
38  {
39      loop innerpoints level lev order rb block 1 1
40          solution = solution[lev] + (inverse( diag(Lapl[lev]) ) * omega
41              * ( f[lev] - Lapl[lev] * solution[lev] ) )
42      next
43  }
44
45  def cpu Restrict ( lev:Int coarse:Array fine:Array) : Unit
46  {
47      loop innerpoints level coarse order lex block 1 1
48          coarse = RestrictionStencil * fine | ToCoarse
49      next
50  }
51
52  def cpu interpolatecorr( lev:Int uf:Array uc:Array ) : Unit
53  {
54      loop innerpoints level uf order lex block 1 1
55          uf += transpose(RestrictionStencil) * uc | ToFine
56      next
57  }
```

The basic structure of the syntax is quite similar to the ones of programming languages. But there are some features especially developed for this purpose. For one, the keyword *loop* (lines 39, 47, 54), which is based on a for-loop construct. The first modifier defines the set of gridpoints the statement is executed on (possible values: *allpoints, innerpoints, boundarypoint*), the second one *level* defines the desired grid level, *order* the order of traversal through the grid points, and *block* if a point-wise or block-wise update is done. By adding *ToCoarse* or *ToFine* to the statement it

is also possible to control inter-grid transfers. For another there is a matrix-vector (stencil-field) product implemented, when one gridpoint requires data from neighboring ones.

3.2.5 Hardware Description

Additionally to the four layers there is some need of hardware information, which do not influence the DSL of the layers, but their conversion to C++ code.

Listing 3.5: "Hardware description"

```
1
2  Hardware cpu {
3      bandwith = 60
4      pwak = 118
5      cores = 4
6  }
7
8  Node {
9      sockets  = 1
10 }
11
12 Cluster {
13     nodes = 1
14     networkbandwith = 10
15 }
```

With this additional settings it is possible to decide e.g. if a parallalization is possible (OpenMP, MPI). Of course this has affect on the generated code, but it does not change the procedure of the layers.

3.3 Feature Model

As described above in the layer examples, most of the internal necessary information for each layer can be obtained by the defined problem on layer 1 and some additional domain knowledge. These are basically the degrees of freedom ExaStencils provides, like the type of hardware, the dimension of the problem, the exact PDE or the concrete components of the multigrid solver. All these parameters the user can define are collected in a so called *feature model*.

Feature	Level	Values
Computational Domain	1	UnitSquare, UnitCube
Operator	1	Laplacian, ComplexDiffusion
Boundary Conditions	1	Dirchlet, Neumann
Location of grid points	2	node-based, cell-centered
Discretization	2	finite differences, finite volumes
Data type	2	single/double accuracy, complex numbers
Multigrid smoother	3	ω-Jacobi, ω-GS, red-black variants
Multigrid inter-grid transfer	3	constant and linear interpolation and restriction
Multigrid coarsening	3	direct (re-discretization)
Multigrid parameters	3	various
Implementation	4	various code optimization strategies
Platform	Hardware	CPU, GPU
Parallelization	Hardware	serial, OpenMP

Table 3.1: A feature model for the first prototype

Features in bold font need to be specified, settings for all the other ones can be derived from them. Their input is optional.

3.4 Configuration

The current version of ExaStencils gets controlled by two files, both being passed as command line arguments. The first argument is the *settings* file, which includes primarily information about all the files (generated code files, DSL files or configuration files) being created and where to save them. The second argument is the *knowledge* file, obtaining all the important internal domain knowledge information. The following listing shows the most important parameters concerning this thesis, containing their default values. Parameter in bold font are the ones, added during this thesis and will be explained more detailed in upcoming sections.

Listing 3.6: "Knowledge parameter"

```
1
2 var targetCompiler : String                = "MSVC"
3
4 var useDblPrecision : Boolean              = true
5
6 var dimensionality : Int                   = 3
7
8 var minLevel : Int                         = 0
9 var maxLevel : Int                         = 6
```

```
10
11 var domain_readFromFile : Boolean          = false
12 var domain_onlyRectangular : Boolean        = true
13 var domain_rect_generate : Boolean          = true
14
15 var domain_rect_numBlocks_* : Int           = 1
16 var domain_rect_numFragsPerBlock_* : Int    = 1
17
18 var domain_numBlocks : Int                  = 1
19 var domain_numFragmentsPerBlock : Int       = 1
20
21 var domain_useCase : String                 = ""
22
23 var domain_generateDomainFile : Boolean     = false
24
25 var domain_fragmentTransformation : Boolean  = false
26 var domain_fragmentInterpolation : Boolean  = false
27
28 var fragmentFile_config_output : Int        = 2
29
30 var comm_strategyFragment : Int             = 6
31
32 var mpi_enabled : Boolean                   = true
33 var mpi_numThreads : Int                    = 1
34
35 var l3tmp_generateL4 : Boolean              = true
36
37 var l3tmp_smoother : String                 = "Jac"
38 var l3tmp_cgs : String                      = "CG"
39 var l3tmp_numRecCycleCalls : Int            = 1
40 var l3tmp_numPre : Int                      = 3
41 var l3tmp_numPost : Int                     = 3
42 var l3tmp_omega : Double                    = 1.0
43 var l3tmp_genHDepStencils : Boolean        = false
44
45 var l3tmp_exactSolution : String           = "Zero"
46 var l3tmp_printFieldAtEnd : Boolean        = false
```

- **targetCompiler:** Indicates which compile the generated code uses. Possible choices are *"MSVC"* (Microsoft Visual C++), *"GCC"* (GNU Compiler Collection), *"IBMXL"* (IBM XL C/C++ Compiler) and *"IBMBG"* (IBM Blue Gene).

- **useDblPrecision:** Option to use floating point numbers with double precision, otherwise single precision.
- **dimensionality:** Defines the dimension of the mathematical problem, specified on layer 1.
- **minLevel / maxLevel:** The used multigrid levels (coarsest and finest).
- **domain_readFromFile:** Indicates if the domain gets defined by a file, see subsection 5.1.3
- **domain_onlyRectangular:** Specifies if only rectangular domains are used.
- **domain_rect_generate:** Specifies if dynamic domain setup code gets generated for rectangular domains. Only possible when *domain_onlyRectangular* is true and *domain_readFromFile* is false.
- **domain_rect_numBlocks_*:** Number of blocks in each dimension (marked with *, meaning *x*/*y*/*z*). One block is usually mapped to one MPI thread. This option does only matter when generating a rectangular domain.
- **domain_rect_numFragsPerBlock_*:** Number of fragments for each block per dimension (* means again *x*/*y*/*z*). Again, this option does only matter when generating a rectangular domain.
- **domain_numBlocks:** Total number of blocks of the domain, important when using a domain specified by a file.
- **domain_numFragmentsPerBlock:** Number of fragments per block.
- **domain_useCase:** Temporary parameter, indicates the use of some specified shaped domains. Possible values are *"L-Shape","X-Shape","Plus-Shape"*.
- **domain_generateDomainFile:** Specifies if the current used domain will be saved in a domain file, see subsection 5.1.3.
- **domain_fragmentTransformation:** Specifies if the transformation matrices, submitted by the domain file, will be used to transform the fragments. See subsection 4.2.3
- **domain_fragmentInterpolation:** Indicates if calculations regarding the position of deformed fragments, submitted by the domain file, will be done by a bi-/trilinear interpolation, see subsection 4.2.3. Note: Only one of this options can be used.
- **fragmentFile_config_output:** Controls how to save all the fragment information. Possible values are *0*, when only a binary file should be created (see subsection 5.2.2). This in mandatory when *domain_rect_generate* is false. Value *1* means, only a readable version of the fragment file gets created. This file, whose syntax is similar to JSON, is only useful for debugging reasons (see subsection 5.2.1). Value *2* indicates that both files get created.

- **comm_strategyFragment:** Specifies if communication is only performed along edges in 2D or faces in 3D, when setting this parameter to *6*, or along every Vertex, when the value is *26*.
- **mpi_enabled & mpi_numThreads:** Specifies if a MPI parallelization and how many MPI processes will be used. This should be in conformity with *domain_numBlocks*, which needs to be an multiple of it.
- **l3tmp_*:** All the parameter starting with it are only temporary ones, as long as both, layer 3 and layer 4 are not consistently working together:
  - **generateL4:** Option to generate a new Layer4.exa file
  - **smoother:** Choice of the multigrid smoother, possible values are *"Jac"* (Jacobi), *"GS"* (Gauss-Seidel) and *"RBGS"* (Red-Black Gauss-Seidel).
  - **cgs:** The generated coarse grid solver, currently only *"CG"* (Conjugate Gradient) possible.
  - **numRecCycleCalls:** Differs between V-cycle (value *1*) and W-cycle (*2*).
  - **numPre & numPost:** Number of pre- and post-smoothing steps.
  - **omega:** Relaxation parameter.
  - **genHDepStencils:** Generates a stencil, which depends on the gridsize. Important when using finite differences.
  - **exactSolution:** Defines which function is used for the RHS. Allowed options are *"Zero","Polynomial","Trigometric","Kappa","Kappa_VC"*.
  - **printFieldAtEnd:** Indicates if the result gets printed in a seperated .dat file.

4 Definition of Domain

The current version of ExaStencils allows only strictly regular rectangular grids, in 2D as an unit square and in 3D as unit cube. Both grids are shown in figure 4.1

Figure 4.1: Current possible computational domains

4.1 Goal

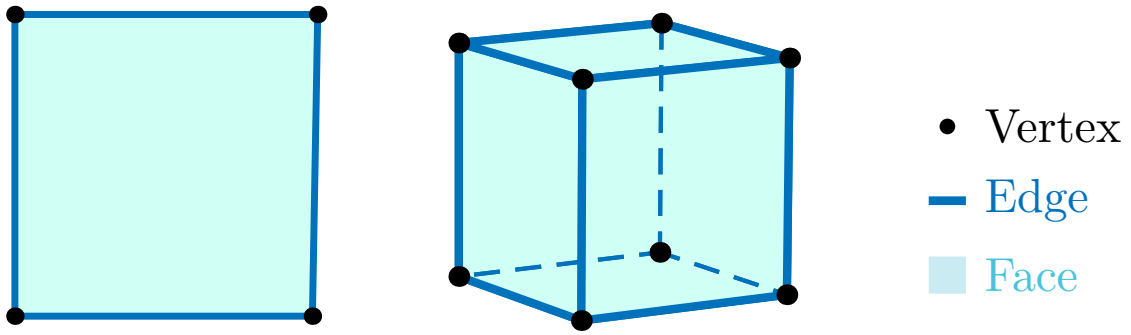
It is desirable to define the computational domain as flexible as possible to cover a maximum of e.g. geometrical situations a pde should be solved on. Depending on the numerical solution method (finite differences/ elements/ volumes) there are some restrictions regarding the geometry the domain can be adopted to. This thesis creates a base providing the possibility to define several different geometrical domains, without violating any mathematical restrictions. It should be seen as first step to make ExaStencils more flexible.

4.2 Concept of fragments

To get a more flexible way of defining its geometry, the domain gets divided into so called fragments.

4.2.1 Geometrical idea

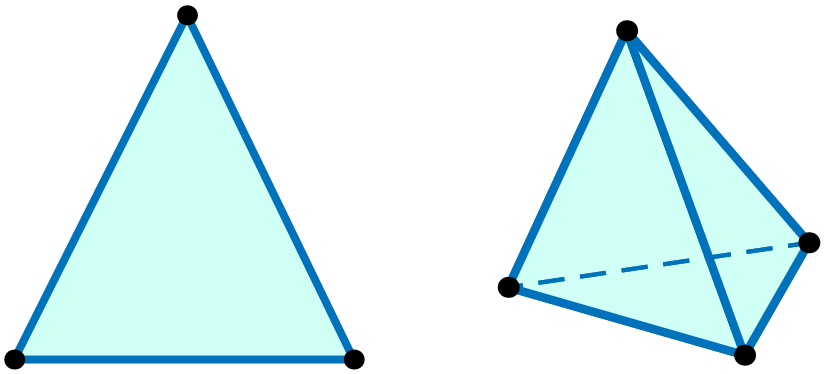
A fragments consists of faces (in 2D one face, in 3D several faces), a face consists of edges and an edge is defined by two vertices. This way all possible straight-lined forms are possible to define a fragment, respectively a domain. The following figure 4.2 illustrates the concept of fragments with faces, spanned by four vertices.



(a) 2D (b) 3D

Figure 4.2: Structure of a fragment

For reasons of simplification this thesis takes only faces containing four edges into account as shown in figure 4.2. Nevertheless the code is designed in a way that an upgrade to other geometrical forms can be done quite easy. These geometrical forms can be e.g. triangles in 2D or tetrahedrons in 3D, as shown in the following figure 4.3



(a) 2D (b) 3D

Figure 4.3: Further possible structures of a fragment

The concept of faces, edges and vertices stays the same.

4.2.2 Setup

Every fragment needs to save some specific information for the creation of a consistent generated code. The following listing explains all the parameter of one fragment.

- **Vertices:** A list of all vertices defining the fragment. A vertex contains explicit coordinates, depending of their dimensionality.
- **Edges:** A list of all edges of the fragment. As mentioned before, an edge is defined by two vertices.

- **Faces:** A list of all faces. Depending of the geometry, a face is defined by a certain number of edges.
- **Rank:** In case the generated code is parallelized by OpenMPI [Ope] , each fragment is assigned to one mpi rank. The subsection 4.3.2 explains it more in detail how this is arranged.
- **Global ID:** This is an unique id for each fragment. The id is a positive integer number
- **Local ID:** This id is linked to the rank. Each fragment has an unique id inside of one rank block.
- **Neighbor IDs:** This list stores the global ids of each relevant neighbor (depending on the settings regarding at each edge or at each corner). In case there is a boundary instead of a fragment, the id gets marked by a negative integer number.
- **Domain IDs:** Each fragment belongs to one or several number of different mathematical subdomains, all stored in this list.
- **Transformation matrix:** In case the user wishes to change the shape of a fragment, it is possible to commit a 4x4 transformation matrix. The default one is a identity matrix.
- **Binary size:** It is important to know for each fragment the binary size to save all the fragment information in one data file — see subsection 5.2

4.2.3 Deformation

It is the goal, not only to define rectangular shapes, but also shapes of arbitrary quadrilaterals. When this is done the arrangement of the gridpoints of each multigrid level need to be adjusted accordingly. The following figure shows a deformed fragment an their grid cells after some steps of refining the original fragment cell.

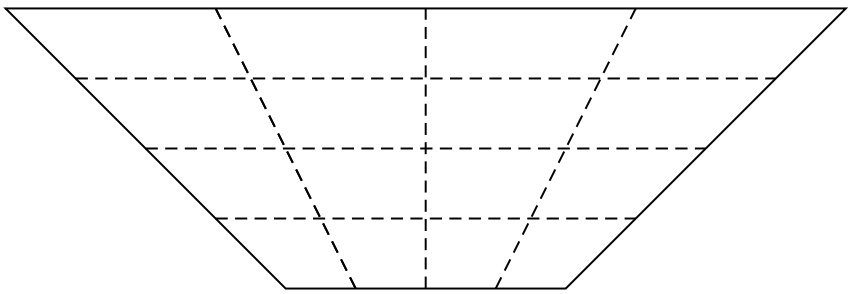


Figure 4.4: Deformed fragment with corresponding grid cells after two refinment steps

To achieve the possibility to generate quadrilateral fragments of any shape, two different ways had been implemented.

Transformation matrix

The user has the option to pass a transformation matrix M, which deforms a unit square or unit cube.

$$M = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & 1 \end{bmatrix} \tag{4.1}$$

The transformation matrix is composed of sub-matrix for linear transformation, spanning from a_{00} to a_{22} , a translation sub vector a_{03} to a_{23} and a projection row a_{30} to a_{32}. To generate a homogeneous 4x4 matrix an additional *1* needs to be set at position a_{33}. Every point inside a domain, e.g. a specified grid point, can be transformed by multiplying the transformation matrix to it. A matrix, which does not change the shape of a fragment, respectively the coordinates of points inside of it, is the 4x4 identity matrix.

The green marked sub-matrix can be used for the following linear transformations:

- **Scale:** Setting the parameter s_* on die diagonal of the matrix changes the size of the fragment.

$$\begin{bmatrix} scale_x & 0 & 0 \\ 0 & scale_y & 0 \\ 0 & 0 & scale_z \end{bmatrix}$$

- **Rotation:** There are different sets of rotation matrices for the three dimensions, one for each axis that can be rotated around.

$$\text{x-axis rotation: } \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}$$

$$\text{y-axis rotation: } \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}$$

$$\text{z-axis rotation: } \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Shearing:** Shears the shape along one axis, marked by a capital letter.

$$\begin{bmatrix} 1 & shearX_y & shearX_z \\ shearY_x & 1 & shearY_z \\ shearZ_x & shearZ_y & 1 \end{bmatrix}$$

Additionally it is possible to change the physical position of a fragment by setting the translation sub vector.

$$\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

The projection row is also implemented, to keep the structure of a 4x4 matrix, but has basically no use in this kind of situation.

Note: It is possible to compose multiple transformation matrices into one matrix my multiplying them together.

To obtain a transformed coordinate vector pos', the original vector pos need to be multiplied to the transformation matrix M.

$$pos' = M \cdot pos \tag{4.2}$$

Interpolation

Another option is to specify the shape of a deformed fragment directly and calculate all the points inside by a bi-linear (in 2D), respectively a tri-linear (3D) interpolation.

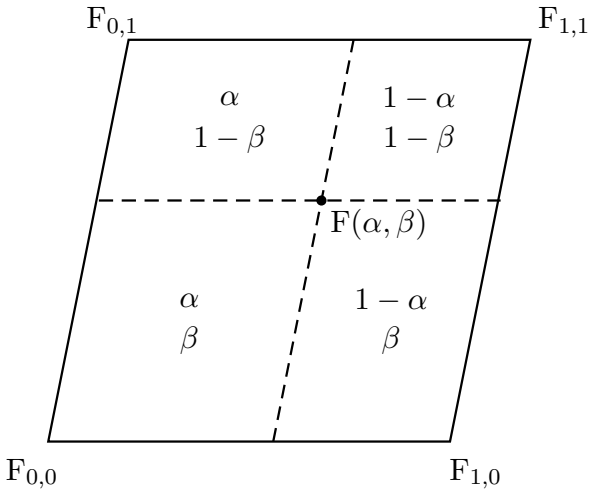


Figure 4.5: Bi-linear interpolation

The concept of a bi-linear interpolation is to calculate a function value $\mathrm{F}(\alpha, \beta)$ by using the given values $\mathrm{F}_{*,*}$ at the corners of the area. This is done by dividing the are into 4 parts by using the weighting parameter $0 \leq \alpha \leq 1$ for the x-direction and $0 \leq \beta \leq 1$ for the y-direction. The function value of interest is in this situation the coordinates at the point F. Regarding figure 4.5 the functional value $\mathrm{F}(\alpha, \beta)$ can by calculated by the equation:

$$F(\alpha, \beta) = F_{0,0}(1-\alpha)(1-\beta) + F_{1,0}\alpha(1-\beta) + F_{0,1}(1-\alpha)\beta + F_{1,1}\alpha\beta \tag{4.3}$$

Analogously, it is possible to derive an equation for the tri-linear interpolation by introducing an additional parameter $0 \leq \gamma \leq 1$ for the z-direction. This results in the following equation:

$$\begin{aligned} F(\alpha,\beta,\gamma) = F_{0,0,0}(1-\alpha)(1-\beta)(1-\gamma) + F_{1,0,0}\alpha(1-\beta)(1-\gamma) + \\ F_{0,1,0}(1-\alpha)\beta(1-\gamma) + F_{0,0,1}(1-\alpha)(1-\beta)\gamma + F_{0,1,1}(1-\alpha)\beta\gamma + \\ F_{1,0,1}\alpha(1-\beta)\gamma + F_{1,1,0}\alpha\beta(1-\gamma) + F_{1,1,1}\alpha\beta\gamma \end{aligned} \tag{4.4}$$

When setting the parameter $\gamma = 0$ as constant, equation 4.4 reduces to equation 4.3. So this equation could be implemented independent of the dimension of the mathematical problem.

To apply this approach on a given multigrid, the parameter α,β,γ need to be defined as the index values of each direction, normalized to the maximum number of index points of the corresponding direction of the given grid.

$$\alpha = i/i_{max} \tag{4.5a}$$

$$\beta = j/j_{max} \tag{4.5b}$$

$$\gamma = k/k_{max} \tag{4.5c}$$

So with all these steps, this interpolation procedure can be used to define a mapping between indices of an arbitrary structured (but not necessarily uniformly regular) grid to physical coordinates.

4.2.4 Code conversion

The implementation of this concept of fragments is shown partly in the following listing

Listing 4.1: implementation of a fragment class and its sub classes

```
1  class Fragment( localId : Int,
2                  globalId : Int,
3                  domainIds : ListBuffer[Int],
4                  faces : ListBuffer[Face],
5                  edges : ListBuffer[Edge],
6                  vertices : ListBuffer[Vertex],
7                  neighborIds : ListBuffer[Int],
8                  rank : Int,
9                  transformationMatrix : ListBuffer[Double] = ListBuffer (1,0,{...},0,1) ){
10         ...
11 }
12
13 class Vertex(coords : ListBuffer[Double]){
14         ...
```

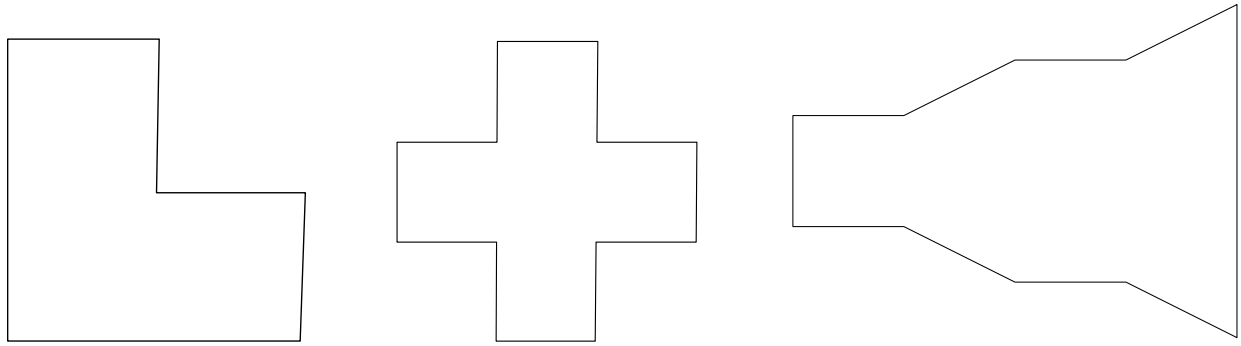
```
15 }
16
17 class Edge(vertex1 : Vertex, vertex2: Vertex) {
18         def contains(v : Vertex) : Boolean = {...}
19          ...
20 }
21
22 class Face(edges : ListBuffer[Edge], vertices : ListBuffer[Vertex]){
23          ...
24 }
```

4.3 Domain setup

A combination of fragments defines a computational subdomain, the set of all subdomains is the global computational domain. It is possible that subdomains overlap each other, respectively include each other.

4.3.1 Shape

The shapes used in this thesis are restricted to these ones, which can be merged by quadrilateral forms. The following figure 4.6 shows some examples of domain shapes that would be possible.



(a) L-Shaped Domain (b) Plus-Shaped Domain (c) Arbitrary-Shaped Domain

Figure 4.6: Different examples of domain shapes

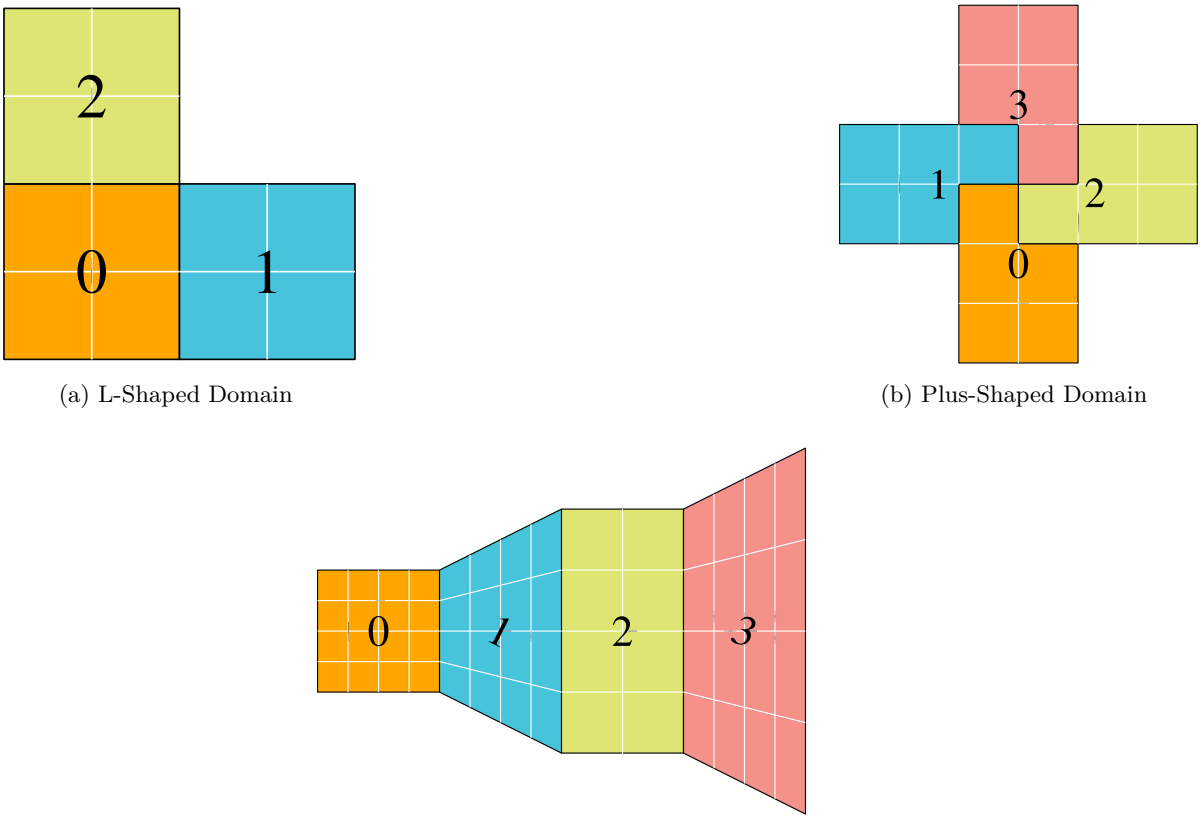
It should be mentioned that the shape of these domains are all possible, but they influence the discretization and the choice of the numerical method. With the shapes of figure 4.6a and 4.6b it is possible to generate a regular cartesian grid, shape 4.6c on the other hand needs to be discretized with an structered grid, which makes it difficult to use Finite Differences. So the preferred method would be the Finite Element Method or Finite Volume Method.

4.3.2 Separation

When the generated C++ code gets created it is possible by setting the parameter *mpi_enabled = true* to generate a MPI parallelized code. The idea is to assign each mpi process a certain amount of fragments (a so called block). Each process tries to solve the defined problem on this block. At the boundaries of each block or fragment, the mpi process communicates with the neighboring fragment. Depending on the fact whether the neighbor is a local one of the same mpi rank or a remote one of a different one, the communication needs to use the mpi communication methods, as *MPI_Irecv* or *MPI_Isend*. The number of fragments inside a block can vary for each mpi rank. But for reasons of efficiency it should be as evenly distributed as possible, otherwise one process will idle for some time while another one has to work much longer.

The separation of the domain to the mpi ranks is only based on the arrangement of the fragments into blocks. It is decoupled from arrangement of the domain into subdomains. Whether or not a processor needs to solve the problem on a certain fragment depends on the parameter *DomainIds* as mentioned in Subsection 4.2.2.

The following figure demonstrates possible separations of the domains, showed in figure 4.6.



(a) L-Shaped Domain

(b) Plus-Shaped Domain

(c) Arbitrary-Shaped Domain

Figure 4.7: Different examples of domain separations

Figure 4.7a demonstrates one possible way of separating a L shaped domain. There are three blocks, marked by different color and number, each containing four fragments. Each block is intrinsically regular and square, so using the Finite Differences Method is possible without any issues. The same is true for figure 4.7b, even so the shape of each block is not a square one. In this case each of the four blocks contains five fragments. The arrangement of fragments and blocks is worth discussing, depending on the gridpoints inside of a fragment it could be in this situation more efficient to assign two blocks with six fragments and two blocks with four fragments, because the number of remote communication edges would decrease. Figure 4.7c shows a more complicated domain structure. Block 0 and block 2 are regular parts, containing 16, respectively 8 fragments. The geometrical size of them differs, but since the calculation is independent of each other this is not a problem. Block 1 and block 3 are distorted, which makes it more difficult to apply Finite Difference Method. It would be still possible to use it, in case the stencil would be adjusted accordingly, but for this domain it is more reasonable to use the Finite Volume Method instead.

Figure 4.7 illustrates that there are a lot of possible ways to create a domain, as long as it consists (for now only) of quadrilateral areas.

5 Integration into ExaStencils

Including the functionality described in chapter 4 into ExaStencils is the main practical part of this thesis. This chapter describes how this has been done and what needs to be considered. ExaStencils is written in Scala [Sca], it is a programming language, based on Java, combining both, object-oriented and functional programming. One big advantage of Scala are the powerful parser combinators. The reasons for choosing Scala are explained in [SKH+14a].

5.1 Declaration of Domain

First of all the method of declaring the computational domain needs to be introduced. This is done in the layer 4 .exa file, which is providing all the information needed in layer 4.

5.1.1 Current State

At the current version of ExaStencils the domain gets defined by a line similar to this one:

Listing 5.1: Layer 4 default domain definition

```
1 Domain global< [ 0, 0 ] to [ 1, 1 ] >
```

There are only variations possible regarding the size and the dimensionality of the domain. The following listing shows some example settings of the knowledge parameters to generate a code without the extensions of this thesis.

Listing 5.2: Settings for code generation of current state

```
1  dimensionality                  = 2
2  domain_onlyRectangular          = true
3
4  domain_rect_numBlocks_x         = 2
5  domain_rect_numBlocks_y         = 2
6  domain_rect_numFragsPerBlock_x = 2
7  domain_rect_numFragsPerBlock_y = 2
8
9  mpi_enabled                     = true
10 mpi_numThreads                  = 4
```

This setting creates a code, using four mpi processes, each dealing with four fragments. The geometrical shape, the size and the amount of the fragments stays at each mpi process the same. For that reason is it possible to generate a dynamic global domain setup code, valid for each mpi process. The grid is getting generated automatically, the meshsize gets determined by number of multigrid levels and the amount of fragments on the coarsest grid, based on the fact that one fragment equals one cell in the mesh on the coarsest level.

5.1.2 Extension of layer 4 file

This subsection describes a way to declare domain shapes direct in layer 4 internally. The .exa file would look like this:

Listing 5.3: Extended domain definition in layer 4

```
Domain global< [ 0, 0 ] to [ 1, 1 ] >
Domain LShaped< [ 0.0, 0.0 ] to [ 0.5, 0.5 ], [ 0.5, 0.0 ] to [ 1.0, 0.5 ],
                              [ 0.0, 0.5 ] to [ 0.5, 1.0 ] >
```

This domain definition creates one L-shaped domain, as in figure 4.6a. There is still a *global* domain defined. This is necessary, because all the settings regarding number of fragments, blocks, mpi threads etc (see the following listing 5.4) are applied to the global domain. Through the definition of an additional domain the code only considers the fragments, which are inside of the *LShaped* domain (or any other/additional domain) and ignores the ones outside. Some example parameter injected by the knowledge file could look like this:

Listing 5.4: Settings for code generation of a L shaped domain

```
dimensionality                  = 2

domain_rect_generate            = false
domain_readFromFile             = false
domain_onlyRectangular          = false


domain_useCase                  = "L-Shape"

domain_rect_numBlocks_x         = 2
domain_rect_numBlocks_y         = 2
domain_numBlocks                = 4

domain_rect_numFragsPerBlock_x = 2
domain_rect_numFragsPerBlock_y = 2
domain_numFragmentsPerBlock = 4
```

```
17
18 mpi_enabled                    = true
19 mpi_numThreads                 = 4
```

- **domain_rect_generate = false:** Must be *false* because it can only be used in squared strictly regular domains
- **domain_readFromFile = false:** In this situation it is possible to generate a slightly variant domain without reading it from a file.
- **domain_onlyRectangular = false:** Even so a L-shaped domain consists of rectangular part the domain itself is not rectangular, because of the 270° angle.
- **domain_useCase = "L-Shape" :** Right now this kind of definition works only for some use cases, whose parameter are defined in the code.
- **domain_rect_numBlock_* and mpi_numThreads:** As mentioned before, all these parameters are applied to the global domain first. They are necessary to create the initial fragment arrangement. When this is done, these parameters will be changed to their correct values concerning the L-shape (respectively another chosen use case) automatically.

The settings from listing 5.3 and 5.4 result in the domain, illustrated in the following figure 5.1

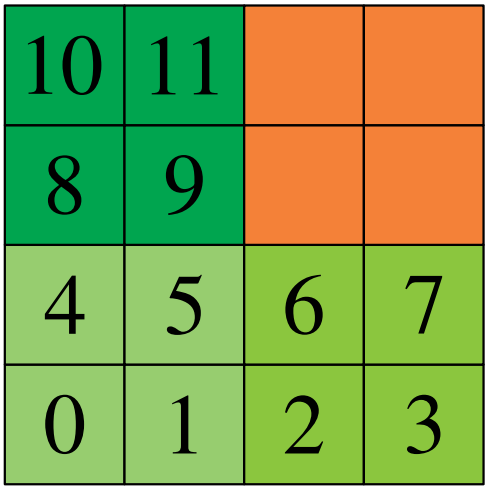


Figure 5.1: Generated L-shaped domain

This time the fragments are numerated. The used mpi blocks are colored in green. All 16 fragments are initiated, but the ones in the upper right corner will be ignored when passing the fragments (see section 5.2) to the generated C++ code. In this scenario the number of mpi threads gets reduced from initial four to three and the total number of used fragments from 16 to 12.

There are some things, which need to be considered, when a domain gets defined in this way. For one thing, only shaped domains, which are composed of rectangular areas, are possible.

And for another thing, the shaped domains need to be compatible with the segmentation of the *global* domain. So the starting point of an L-shaped domain can not be in between one fragment, it has to be at the boundary of the fragment.

5.1.3 Domain definition file

Additionally to the extension of the functionality there is now the possibility to pass a file, which describes the domain. This is necessary as soon as the domain is structured in a more complex way. Instead of defining a domain directly on layer 4, there is just the path to the domain file, as shown in the following listing.

Listing 5.5: layer 4 domain definition when using a file

```
1 Domain fromFile("DomainFile.cfg")
```

To use this option, the following parameter needs to be set:

Listing 5.6: Settings to read from domain file

```
1 domain_readFromFile          = true
```

Some parameters (e.g. dimensionality, see 5.1.3) do not need to be configured by the knowledge file, because they are domain specific and need to be set by the domain file.

A method to write a domain file is also implemented and its getting triggered by the following parameter in the knowledge file:

Listing 5.7: Settings to write domain file

```
1 domain_generateDomainFile          = true
```

Depending on the parameter *domainFile*, set by the settings file, the domain file gets named.

Layout

This domain file needs to be in a certain structure, shown in the following listing.

Listing 5.8: Layout of Domain Definition File

```
1 #exastencils domain file
2 DATA
3 dimensionality = x
4 numBlocks = x
5 numFragmentsPerBlock = x
6 mpi_numThreads = x
```

```
discr_hx = (h_x0,h_x1,...,h_xn)
discr_hy = (h_y0,h_y1,...,h_yn)
discr_hy = (h_z0,h_z1,...,h_zn)
domainIdentifier = "name0,name1,..."
DOMAINS
name0 = (b_0,b_1,...,b_m-1)
name1 = (b_m,b_m+1,...,b_n)
 ...
BLOCKS
b_n = (f_n, f_n-1 ,..., f_m)
b_n-1 = (f_m-1 ,..., f_j)
 ...
b_0 = (..., f_1, f_0)
FRAGMENTS
f_0 = (((v_0x,v_0y,v_0z),(v_1x,v_1y,v_1z)),((v_2x,v_2y,v_2z),(v_3x,v_3y,v_3z)),(...),(...)),(...),(...),(...))
f_1 = (...)
f_2 = (...)
 ...
f_n = (...)
TRAFOS
f_0 = ((a_00,a_01,a_02,a_03),(a_10,a_11,a_12,a_13),(a_20,a_21,a_22,a_23),(a_30,a_31,a_32,a_33))
f_1 = ( ... )
 ...
f_2 = ( ... )
```

- **DATA (2-10):** This part is basically the header of the file. The parameters listed here must be set by the user. The red marked x are integer values, h_{**} are floating point numbers. The parameter domainIdentifier lists all the different subdomains.
- **DOMAINS (12-14):** An assignment of blocks for each domain mentioned in domainIdentifier must take place in this part. The blocks are listed between brackets, separated by b a comma.
- **BLOCKS (16-19):** Each block contains fragments, as before they are listed between two brackets, separated by a comma. The id of each block starts with a *b*, followed by an integer value. When reading the file the integer part will become the id of the block and so the mpi rank. So it is mandatory that the lowest id begins with b0 and continues chronological. The arrangement of the ids inside of this block on the other hand does not matter.
- **FRAGMENTS (21-25):** The structure of the ids is similar to the ones of the blocks, except for the fact that the fragment ids start with *f*, followed by an integer (which will become later on the global id). Different to the block ids, those ids do not need to be

chronological. The value of each fragment consists of a list of faces, separated by comma. A face (marked with green brackets (), see line 21) consists of a list of edges (orange brackets ()) and a edge consists of two vertices (marked by blue brackets and values ()). This results in a nested but straightforwarded way for declarations of fragments.

- **TRAFO (27-30):** This section is optional. In case it is necessary to pass some transformation matrices for some fragments, this can be done here. The parameter consists of four lines, each containing four values. This results in a 4x4 matrix, which will be used to change the shape of a fragment. The ids of this section match the ones of the fragments section, so the matrices will be assigned correctly.

Example: L-shape

The following listing shows an example how a domain file could look like. This domain file creates the same L-shaped domain as in figure 5.1

Listing 5.9: L-shape example for domain file

```
1  #exastencils domain file
2  DATA
3  dimensionality = 2
4  numBlocks = 3
5  numFragmentsPerBlock = 4
6  mpi_numThreads = 3
7  discr_hx = (0.25,0.125,0.0625,0.03125,0.015625,0.0078125,0.00390625)
8  discr_hy = (0.25,0.125,0.0625,0.03125,0.015625,0.0078125,0.00390625)
9  domainIdentifier = "LShaped"
10 DOMAINS
11 LShaped = (b2,b1,b0)
12 BLOCKS
13 b2 = (f8,f9,f10,f11 )
14 b1 = (f4,f5,f6,f7 )
15 b0 = (f0,f1,f2,f3 )
16 FRAGMENTS
17 f0 = ((((0.00000,0.00000) , (0.25000,0.00000)),((0.00000,0.00000) ,
       (0.00000,0.25000)),((0.25000,0.00000) , (0.25000,0.25000)),((0.00000,0.25000) ,
       (0.25000,0.25000))))
18 f1 = ((((0.25000,0.00000) , (0.50000,0.00000)),((0.25000,0.00000) ,
       (0.25000,0.25000)),((0.50000,0.00000) , (0.50000,0.25000)),((0.25000,0.25000) ,
       (0.50000,0.25000))))
19 f2 = ((((0.00000,0.25000) , (0.25000,0.25000)),((0.00000,0.25000) ,
       (0.00000,0.50000)),((0.25000,0.25000) , (0.25000,0.50000)),((0.00000,0.50000) ,
       (0.25000,0.50000))))
```

```
20 f3 = ((((0.25000,0.25000) , (0.50000,0.25000)),((0.25000,0.25000) ,
       (0.25000,0.50000)),((0.50000,0.25000) , (0.50000,0.50000)),((0.25000,0.50000) ,
       (0.50000,0.50000))))
21 f4 = ((((0.50000,0.00000) , (0.75000,0.00000)),((0.50000,0.00000) ,
       (0.50000,0.25000)),((0.75000,0.00000) , (0.75000,0.25000)),((0.50000,0.25000) ,
       (0.75000,0.25000))))
22 f5 = ((((0.75000,0.00000) , (1.00000,0.00000)),((0.75000,0.00000) ,
       (0.75000,0.25000)),((1.00000,0.00000) , (1.00000,0.25000)),((0.75000,0.25000) ,
       (1.00000,0.25000))))
23 f6 = ((((0.50000,0.25000) , (0.75000,0.25000)),((0.50000,0.25000) ,
       (0.50000,0.50000)),((0.75000,0.25000) , (0.75000,0.50000)),((0.50000,0.50000) ,
       (0.75000,0.50000))))
24 f7 = ((((0.75000,0.25000) , (1.00000,0.25000)),((0.75000,0.25000) ,
       (0.75000,0.50000)),((1.00000,0.25000) , (1.00000,0.50000)),((0.75000,0.50000) ,
       (1.00000,0.50000))))
25 f8 = ((((0.00000,0.50000) , (0.25000,0.50000)),((0.00000,0.50000) ,
       (0.00000,0.75000)),((0.25000,0.50000) , (0.25000,0.75000)),((0.00000,0.75000) ,
       (0.25000,0.75000))))
26 f9 = ((((0.25000,0.50000) , (0.50000,0.50000)),((0.25000,0.50000) ,
       (0.25000,0.75000)),((0.50000,0.50000) , (0.50000,0.75000)),((0.25000,0.75000) ,
       (0.50000,0.75000))))
27 f10 = ((((0.00000,0.75000) , (0.25000,0.75000)),((0.00000,0.75000) ,
       (0.00000,1.00000)),((0.25000,0.75000) , (0.25000,1.00000)),((0.00000,1.00000) ,
       (0.25000,1.00000))))
28 f11 = ((((0.25000,0.75000) , (0.50000,0.75000)),((0.25000,0.75000) ,
       (0.25000,1.00000)),((0.50000,0.75000) , (0.50000,1.00000)),((0.25000,1.00000) ,
       (0.50000,1.00000))))
```

5.2 Fragment data file

This section describes the transfer of all the fragment and domain information into the generated C++ code. For that reason there are two possible ways to save the data into different files. What to save gets controlled by the parameter *fragmentFile_config_output* in the knowledge file. The values can be 0 (save date in binary file),1 (save data in readable file) or 2 (save data in both files)

5.2.1 Readable file

It is possible to generate a file which can be read, its structure is similar to JSON. The file name is the parameter *fragmentFile_path_readable* in the settings file. This is mostly implemented because of debugging reasons. It is recommended to create this file only when generating a

manageable amount of fragments, otherwise this file can get quite big. The following listing shows a segment from one file.

Listing 5.10: Readable fragment file

```
Fragment : [
        domainIds : 0,1
        globalId  :  0
        localId  :  0
        mpiRank : 0
        Vertices  :  [
                ((0.00000,0.00000))
                ((0.25000,0.00000))
                ((0.00000,0.25000))
                ((0.25000,0.25000))
        ]
        Position  :  [
                (0.125,0.125)
        ]
        PosMin : [
                 (0.0,0.0)
        ]
        PosMax : [
                (0.25,0.25)
        ]
        Neighbours : [
                left :  −1
                right :1
                bottom:−2
                top:2
        ]
]
```

This are all the information defining a fragment. Additional to the *Vertices* it is necessary to pass *PosMin* and *PosMax*, in case the fragment is distorted. There are also some different negative values in the *Neighbours* parameter. They are describing different boundaries.

5.2.2 Binary File

More important is the binary file and how it is getting read. This file serves the purpose to transfer all the fragment data to each mpi rank. The size of the file should be as small as possible, so only necessary data are stored.

There are also different ways to provide the data for each mpi rank. It would be also possible

to split the data and create one file for each mpi rank. But because of the fact that there will be projects with a big amount of different fragments this approach has been discarded, because there would be too many files and the cost of reading every single one would be too high. The following subsections about writing and reading the file are assuming this will be done with an active MPI parallelization. But of course this works too without one.

Writing File

This necessary data gets written in binary format to a data file, fragment by fragment, value by value, without any extra information. The path of the file gets set by the parameter *fragmentFile_binary* in the settings file. The information of a fragment are the ones listed in listing 5.10, but with the constraint that only valid neighbors will be written to the file. Saving a negative id for not valid ones is not necessary right now.

To save additional information on how to read the file, another file gets created, whose path gets set by the parameter *fragmentFile_config*. This config file includes the number of fragments and their binary size in the data file for each mpi rank.

Reading File

Having these both files, each mpi rank gets the information from the config file and knows from which point it has to start to read at the fragment data file. The following figure illustrates this concept.

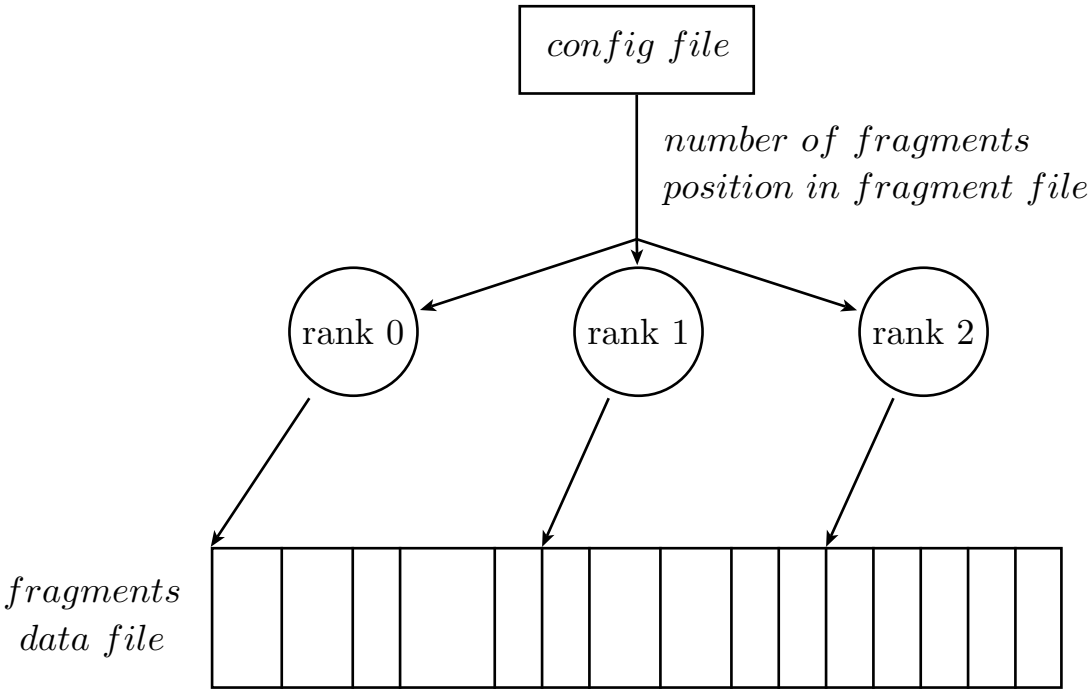


Figure 5.2: Concept of reading fragment data file in parallel

The size of a single fragment can vary, depending on the number of edges and vertices (e.g. when using an unstructured grid with triangle and quadrangles in 2D), respectively, as mentioned, the number of neighbors.

Each mpi rank uses the following code to access the data on the file.

Listing 5.11: Reading File with MPI

```
1 MPI_File fh;
2 MPI_File_open( mpiCommunicator, "./fragments.dat", MPI_MODE_RDONLY,
      MPI_INFO_NULL, &fh);
3 MPI_File_read_at(fh, fileOffset, buf, bufsize, MPI_BYTE,
      MPI_STATUSES_IGNORE);
```

- **mpiCommunicator** is the parameter *mpi_defaultCommunicator*, set in the knowledge file
- **fragments.dat** is the data file containing all the fragments (parameter *fragmentFile_binary* in settings file)
- **MPI_MODE_RDONLY** is the file access mode
- **MPI_INFO_NULL** some info object
- **fileOffset** is an integer, whose value is the starting position in the file for this mpi process

- **buf** is an char array of size *bufsize*

The logic of reading and setting the data in C++ must be aligned to the file writer in Scala. Since there is no information on the types, written to the file, just pure binary data, the file reader must know what and how to read. For that purpose it uses the following template function.

Listing 5.12: Template function for reading values from binary file

```
template <class T> T readValue (char*& memblock) {
    int size = sizeof(T);
    char bytes[size];
    for (int j = 0; j < size; ++j) {
        bytes[size-1-j] = memblock[j];
    }
    memblock+=size;
    return *(T *)&bytes;
}
```

5.3 Implementation of fragments and domains

This sections shows some code snippets of the implementation handling fragments and domains.

5.3.1 Domain implementation

The implementation of the most abstract basis is the domain, implemented as a trait.

Listing 5.13: Domain trait

```
trait Domain {
  def identifier : String
  def index : Int
  def shape : Any
}
```

The shape of the domain is initiated completely flexible and depends on the kind of the domain, followed in the next listing.

Listing 5.14: Different domain types

```
case class RectangularDomain(
  var identifier : String,
  var index : Int,
  var shape : RectangularDomainShape) extends Domain {}

case class ShapedDomain(
  var identifier : String,
  var index : Int,
  var shape : ShapedDomainShape) extends Domain {}

case class FileInputGlobalDomain(
  var identifier : String,
  var index : Int,
  var shape : List[FileInputDomain]) extends Domain {}

case class FileInputDomain(
  var identifier : String,
  var index : Int,
  var shape : FileInputDomainShape) extends Domain {}
```

Each domain type, implemented as a case class, uses a different kind of shape. When a rectangular or any shaped domain gets defined on layer 4, as in listing 5.1 or listing 5.3, then a *RectangularDomain*, respectively a *ShapedDomain* gets chosen. When using a domain definition

file, described in section 5.1.3, a *FileInputGlobalDomain* will be used with a list of each subdomain (*FileInputDomain*) as shape parameter. A single *FileInputDomain* uses the *FileInputDomainShape*.

Additionally there exists an object *DomainCollection*, holding all the defined domains and some methods.

Listing 5.15: Domain Collection object

```
object DomainCollection {
  var domains : ListBuffer[Domain] = ListBuffer()
  def getDomainByIdentifier(identifier : String) : Option[Domain] = {...}
  def initFragments() {...}
```

The method *initFragments* initializes all the fragments by calling the corresponding method at the chosen DomainShape.

5.3.2 Domain shape implementation

Similar to the domain handling, all the domain shapes base upon a trait. There is also a value – *shapeData* – which can be any kind of type.

Listing 5.16: Domain shape trait

```
trait DomainShape {
  def shapeData : Any
  def contains(vertex : Vertex) : Boolean
  def initFragments() : Unit
}
```

And again the concrete domain shapes are implemented as case classes.

Listing 5.17: Domain shape types

```
case class FileInputDomainShape(override val shapeData : String) extends DomainShape {
  var blocks : List[String] = List()
  var frags : List[String] = List()
  def contains(vertex : Vertex) : Boolean = {...}
  def initFragments() = {...}
}

case class ShapedDomainShape(override val shapeData : List[RectangularDomainShape])
    extends DomainShape {
  def contains(vertex : Vertex) : Boolean = {...}
  def initFragments() = {...}
}
```

```
12
13 case class RectangularDomainShape(override val shapeData : AABB) extends DomainShape {
14   def contains(vertex : Vertex) : Boolean = {...}
15   def initFragments() = {...}
16   case class Index(r : Int, k : Int, j : Int, i : Int) {...}
17   def calcGlobalFragmentId(indices : Index) : Int = { }
18   def calcLocalFragmentId() : Int = {...}
19   def getNeighbors(indices : Index) : ListBuffer[Int] = {...}
20   def getIntervalOfRank(rank : Int) : Map[String, Interval] = {...}
21   def getNeighbor(kStep : Int = 0, jStep : Int = 0, iStep : Int = 0) = {...}
22   def getGlobalId() : Int = {...}
23   def checkValidity(id : Int) : Int = {...}
24 }
```

- **FileInputDomainShape:**
  - shapeData: In this case shapeData serves just the purpose to name the subdomain.
  - blocks, frags: Both variables gets set, when reading the domain definition file. See section 5.1.3
  - initFragments: All fragments are already set by the parser, reading the domain definition file. So this method is just assigning the neighbors correctly.
- **ShapedDomainShape :**
  - shapeData: Since shaped domains consists of rectangular areas, this value is a list of RectangularDomainShape
  - initFragments: initializes all the fragments in regard to the global domain and filters all the fragments not belonging to the defined shaped domains.
- **RectangularDomainShape:**
  - shapeData: A axis-aligned bounding box, representing the domain or a part of the domain
  - initFragments: Accordingly to the knowledge file it creates vertices, edges, faces and fragments, sets the global and local ids and determines the neighbor relations and distributes the fragments to the mpi ranks. For that it uses all the functions listed afterwards.

5.3.3 Fragment Methods

All the methods regarding fragment logic are combined in the object *FragmentCollection*, which contains also the list of all fragments.

Listing 5.18: FragmentCollection

```
object FragmentCollection {
  var fragments : ListBuffer[Fragment] = ListBuffer()

  def getLocalFragId(globalId : Int) : Int = {...}
  def getMpiRank(globalId : Int) : Int = {...}
  def getDomainIds(globalId : Int) : ListBuffer[Int] = {...}
  def getRemoteRank(globalId : Int, domainId : Int) : Int = {...}
  def isValidForSubDomain(globalId : Int, domain : Int) : Boolean = {...}
  def getNumberOfNeighbors() : Int = {...}
  def isNeighborValid(globalId : Int, neighborId : Int, domain : Int) : Boolean = {...}
  def isNeighborRemote(globalId : Int, neighborId : Int, domain : Int) : Boolean = {...}
  def getFragPos(vertices : ListBuffer[Vertex]) : Vertex = {...}
  def getPosMin(vertices : ListBuffer[Vertex]) : Vertex = {...}
  def getPosMax(vertices : ListBuffer[Vertex]) : Vertex = {... }
  def getNeighborIndex(fragment : Fragment, neighbor : Fragment) : Option[Int] = {...}
}
```

6 Examples and Results

This chapter demonstrates the solutions of several different domains regarding a specified mathematical problem of the form:

$$\Delta u = f \quad \text{in } \Omega \tag{6.1a}$$

$$f = -6xy \tag{6.1b}$$

At the current version of ExaStencils only Dirichlet boundary conditions are possible. These are set by the function $g(x, y)$, which represents in this situation also the exact solution, which ca be used for validation of the generated solutions.

$$g(x, y) = x^3 + y^3 \tag{6.2a}$$

$$u = g(x, y) \quad \text{on } \partial\Omega \tag{6.2b}$$

This problem had been discretized by finite differences and the multigrid algorithm had been used, with the damped Jacobi smoother ($\omega = 0.8$) for the presmoothing and postsmoothing steps, which were conducted three times each. On the coarsest grid level the CG method had been chosen for the generation of the solution. The exit criterion of the solver loop had been set to $1 \cdot 10^{-5}$, representing the target reduction of the residual regarding the initial residual.

For the evaluation of the different domains, certain aspects had been considered:

- **Residual:** The discretization of the problem 6.1a results in a sparse linear system of equations $Au = f$ with the discretization matrix A, basically representing the stencil. When generating a approximated solution $\tilde{u}$, the residual is defined as

  $$r = f - A\tilde{u} \tag{6.3}$$

  To get a more distinctive quantity for the residual field, a normalized residual value of the iteration step had been determined by the L2 norm.

  $$r_{iteration} = \sqrt{\sum_{i,j,k} r_{i,j,k}^2} \tag{6.4}$$

- **Convergence Factor:** Represents a order of magnitude regarding to the quality of the solving procedure. Small values ($<< 1$) represent a very good approximation of the exact solution. When the values getting bigger (but still < 1) it needs more steps to get the best solution possible. Values bigger than 1 are indicating that the chosen solution procedure is not working.

$$\text{convergence factor} = \frac{r_{iteration}}{r_{iteration-1}} \tag{6.5}$$

  The factor gets determined by taking the ration of the normalized residual from the current iteration step to the last one.

- **Error:** The error gets determined by the difference of the exact solution and the approximated one:

$$\text{error} = |u - \tilde{u}| \tag{6.6}$$

  Contrary to the residual the error gets normalized by the maximum norm:

$$\text{error}_{iteration} = \max_{i,j,k} |u_{i,j,k} - \tilde{u}_{i,j,k}| \tag{6.7}$$

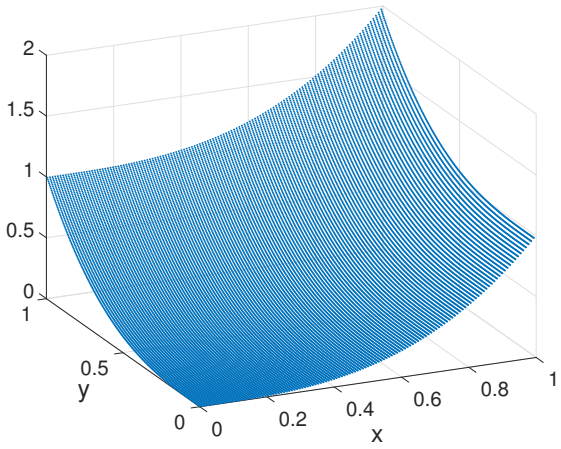
6.1 Unit Square

The calculation on a unit square domain had been conducted primarily to verify the correct handling of the fragment file. For that reason there were two runs of the same domain configuration, the first one with the option *domain_rect_generate* set to true, the second one without this option, so the fragment file could be generated.

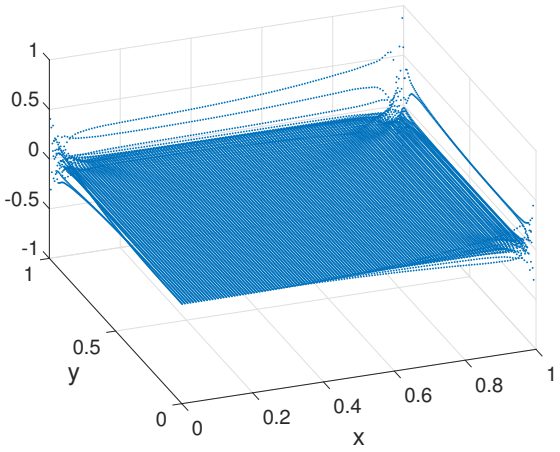
Both settings led to the same results:

Number of Iterations	Residual	Convergence Factor	Error
4	10.2748	0.044454	2.21956e-05

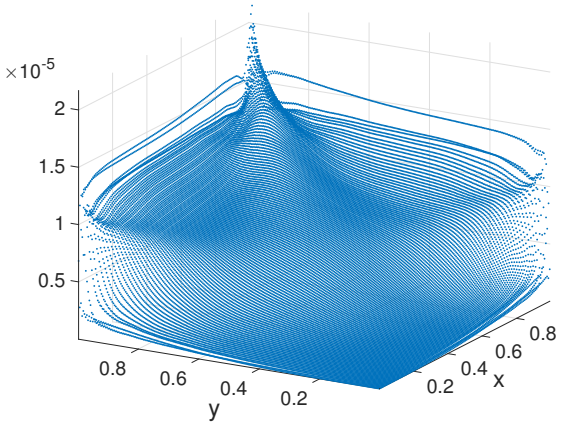
Table 6.1: Results of Calculation with Unit Square Domain



(a) Solution field of Unit Square



(b) Residual field of Unit Square



(c) Error field of Unit Square

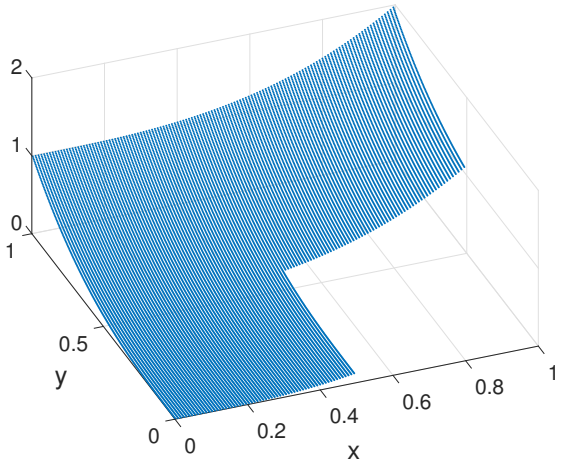
Figure 6.1: Results of Calculation with Unit Square Domain

6.2 L shaped domain

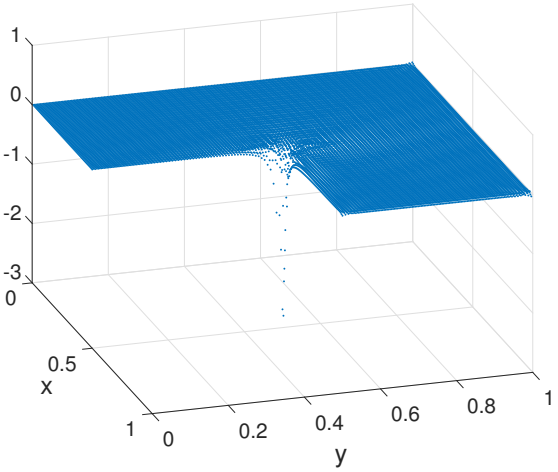
The results when using a L-shaped domain are the following ones:

Number of Iterations	Residual	Convergence Factor	Error
5	10.8018	0.198653	0.00016808

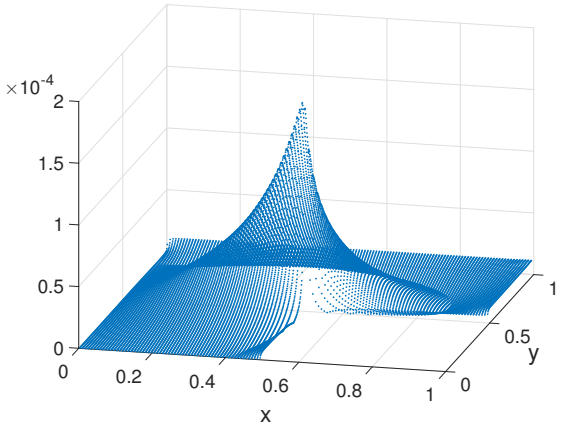
Table 6.2: Results of Calculation with L-shaped Domain



(a) Solution field of L-Shape



(b) Residual field of L-Shape



(c) Error field of L-Shape

Figure 6.2: Results of Calculation with L-Shaped Domain

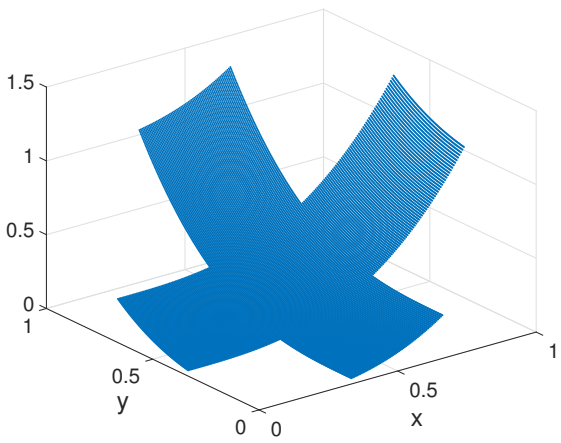
It can be observed that the solution of the L-shaped domain seems to be as smooth as the one of the unit square. Nevertheless, the residual and the error show increased values at the re-entrant corner of the domain. The number of needed steps before hitting the exit criterion also increased by one and the convergence factor is about 4.5 times higher.

6.3 Plus shaped domain

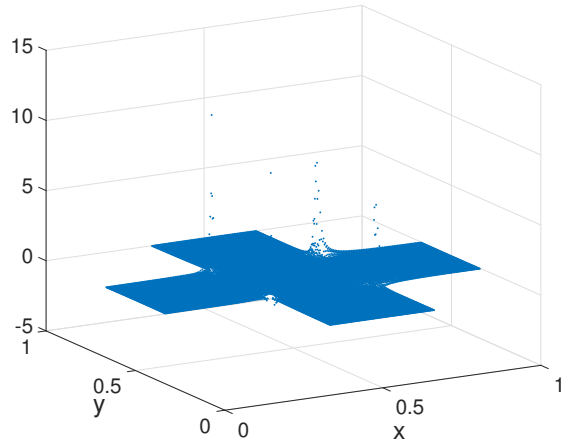
The number of those re-entrant corner gets increased by the Plus-shaped domain, creating the following results:

Number of Iterations	Residual	Convergence Factor	Error
6	45.0642	0.26797	0.000273816

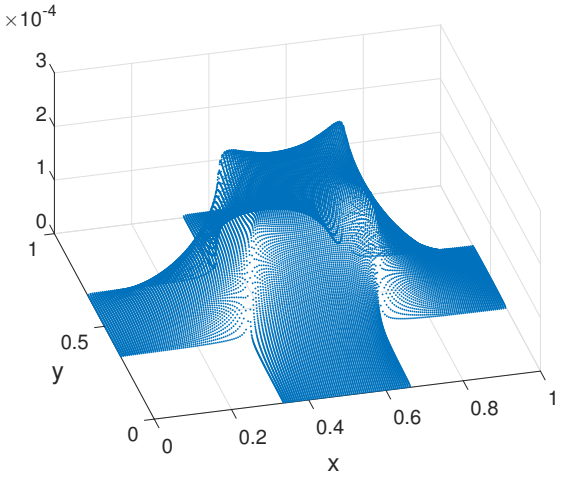
Table 6.3: Results of Calculation with Plus-shaped Domain



(a) Solution field of Plus-Shape



(b) Residual field of Plus-Shape



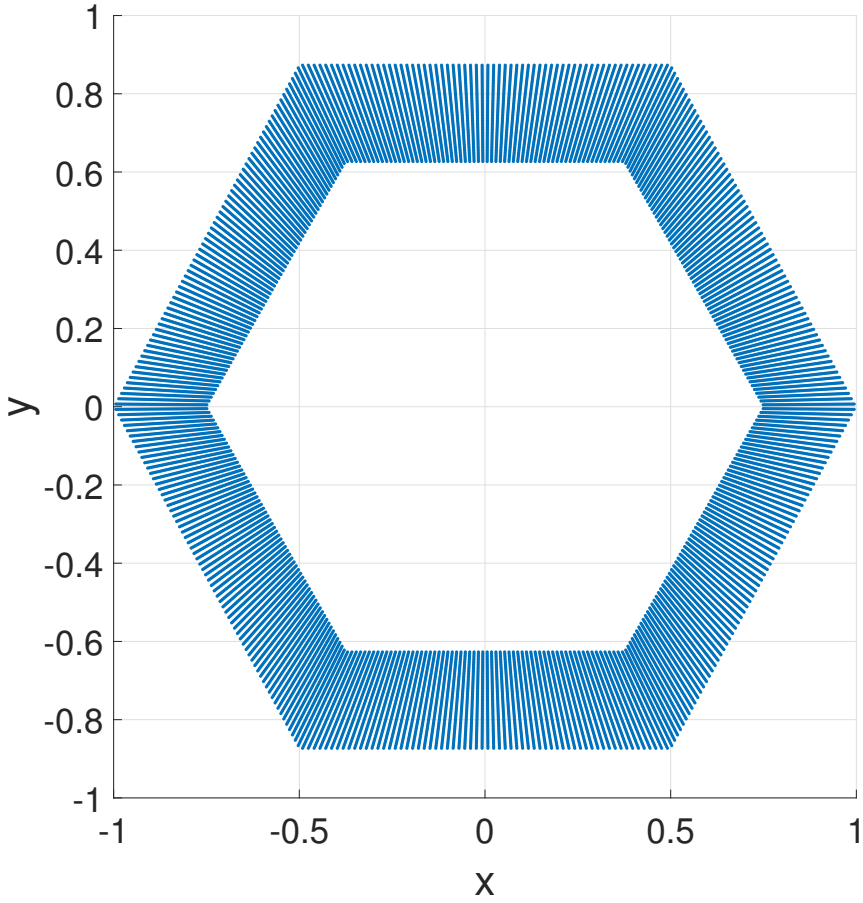
(c) Error field of Plus-Shape

Figure 6.3: Results of Calculation with Plus-Shaped Domain

Again, the number of iterations, the residual, the convergence factor and the error are showing worse results.

6.4 Arbitrary shaped domain

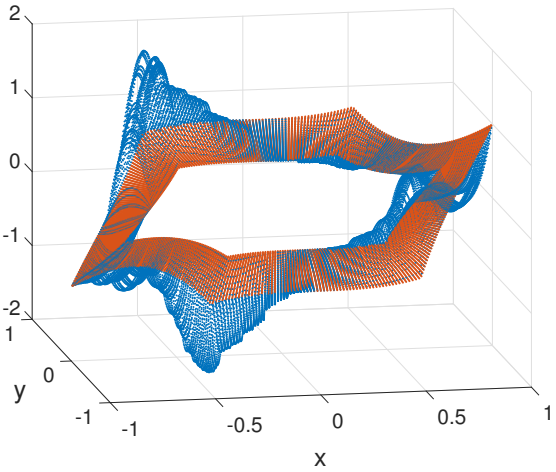
The last example is a more complexly shaped domain, using finite volume (see section2.2) instead of finite differences. It should be mentioned that finite volumes is currently not a verified part of ExaStencils and the made implementation is still an experimental one. First of all the following figure shows the generated domain.



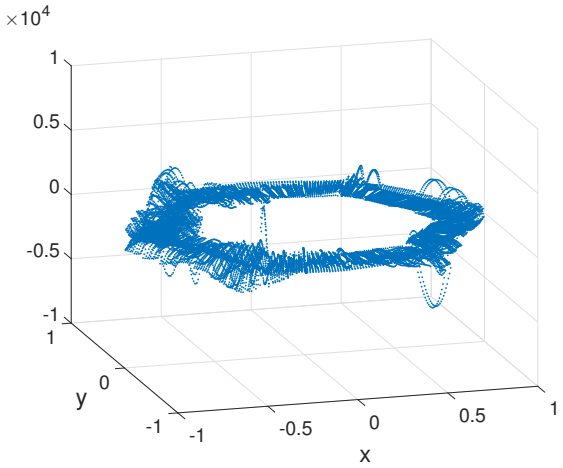
This figure shows that the domain was created correctly, however, the discretization of finite volume led to a bad convergence rate and big errors. Stopping the iterations after three steps (*) led to the following results.

Number of Iterations	Residual	Convergence Factor	Error
3*	102670	0.866198	0.1.86139

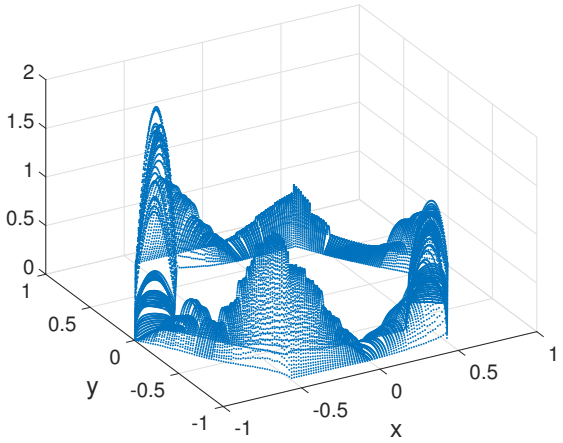
Table 6.4: Results of Calculation with arbitrary shaped Domain



(a) Solution field of arbitrary shape



(b) Residual field of arbitrary shape



(c) Error field of arbitrary shape

Figure 6.4: Results of Calculation with arbitrary shaped Domain

Figure6.4a shows in this case both, the approximated solution and the exact one. It can be observed that the approximation is clung to the exact solution, but there are high fractions of error oscillations. Also good to see in figure 6.4c. Despite this not converging results, it can be seen that the creation of the domain and the communication between the fragments is functional.

7 Conclusion and Future Work

The work described in this thesis has been concerned with the extension of ExaStencils with the requirement to provide the basis of creating more complex structures of domains and how to handle them. For that reason this thesis explained in the beginning the mathematical basics ExaStencils is based on, followed by the explanation of the principles and idea of the framework itself. Afterwards the basic goals were brought to mind by the explanation of the theory of domains and the concept of fragments which has been implemented. An overview how this has been done is given in the section afterwards, followed by some examples of what is possible now.

At this point is still a lot of room for future work. Especially with respect to finite elements or finite volumes the possibility to pass unstructured grids should also be provided, although the basic principles considered it. Nevertheless, the work done for this thesis provides some new features in ExaStencils and has a stake in improving it bit by bit.

Bibliography

[Exa] ExaStencils. Advanced stencil-code engineering (exastencils). http://www.exastencils.org/. Accessed: 2015-07-23.

[HCS+15] Köstler H, Schmitt C, Kuckuk S, Hannig F, Teich J, and Rüde U. A scala prototype to generate multigrid solver implementations for different problems and target multi-core platforms. *International Journal of Computational Science and Engineering (IJCSE)*, 2015.

[LAB+14] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. ExaStencils: Advanced Stencil-Code Engineering. Bericht MIP-1401, Universities of Passau,Wuppertal, Erlangen-Nürnberg, June 2014.

[Ope] OpenMPI. Open source high performance computing. http://www.open-mpi.org/. Accessed: 2015-07-23.

[P91] Wesseling P. *An Introductio To Multigrid Methods*. John Wiley and Sons, 1991.

[Sca] Scala. Object-oriented meets functional. http://scala-lang.org/. Accessed: 2015-07-23.

[SKH+14a] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. An evaluation of domain-specific language technologies for code generation. *International Conference on Computational Science and Its Applications (ICCSA)*, 2014.

[SKH+14b] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. *Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2014.