Implementation of an Adaptive Refinement and Coarsening Strategy Within Peano Framework

Kaveh Rahnema
Implementierung einer dynamisch adaptiven
Verfeinerungs- und Vergröberungsstrategie im Peano
Framework

Implementation of an Adaptive Refinement and Coarsening
Strategy Within Peano Framework

1st Examiner:  Univ.–Prof. Dr. Hans-Joachim Bungartz
2nd Examiner: Univ.–Prof. Dr. Thomas Huckle
Adviser: Dr. Miriam Mehl, Dr. Tobias Weinzierl
I hereby declare that this 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.

30.12.2009
Datum Kaveh Rahnema
Acknowledgments

During the past six months that I have been working on this thesis, several people have helped and supported me in many different ways therefore, I would like to appreciate their efforts. First, I would like to thank Dr. Tobias Weinzierl explicitly, for all the help and support he has given me throughout this six months, from the start up phases up to the writing corrections. Next, I would like to thank Dr. Miriam Mehl for all the help and suggestions she had through this time. Daniel Butnaru, a colleague of mine who also contributed to the Peano framework as part of his Master’s thesis, helped me constantly with different technical difficulties. I would like to apologize for not having enough time to spend with my family and my partner during this six months and thank them for their understanding. And Finally, I would like to thank my examiners, Prof. Dr. Hans-Joachin Bungartz and Prof. Dr. Thomas Huckle for their time spent to examine my thesis.
Abstract

Even though the computers face an incredible evolution throughout the last couple of years, still memory requirements and run time restrict and render many computations unsatisfying even on massively parallel super computers. In Computational Sciences, most of the times we deal with different type of discretizations of time or space. On one hand the finer our discretizations is the more memory and run time is needed to do our computation. On the other hand, the coarser our discretization is the higher our error. Therefore there is higher demand to use an adaptively refined grid. An adaptively refined grid is a grid which has been refined in some specific regions of interest while the remaining domain is resolved at a coarser level.

This thesis, implements two refinement and one coarsening criteria to provide the possibility to have an adaptively refined grid within the Peano framework. To find the regions of interest for refinement or coarsening linear surplus has been chosen. After each refinement one needs to initialize the new vertices. For this, two different interpolation schemes have been implemented, bilinear interpolation and cubic interpolation. To be able to use the refinement and coarsening strategies for all the already implemented solvers within the Peano framework, a new component has been introduced within the framework which works as a refinement/coarsening black-box.

Different experiments show that, by the use of refinement and coarsening strategies one can reduce memory consumption and runtime of the simulation by a great factor. In this thesis the benchmark experiments are different Poisson equations, but several other experiments using these refinement and coarsening strategies have been done, i.e. CFD[5].
## Contents

1 Introduction .............................................................................................................. 11

2 Challenges ................................................................................................................. 13

3 Peano Framework ....................................................................................................... 15
   3.1 Peano Framework Structure .................................................................................. 15
   3.2 Grid Component ................................................................................................... 17
   3.3 K-spacetree ......................................................................................................... 17
   3.4 Grid Traversal ....................................................................................................... 19
      3.4.1 Element-Wise Operations ........................................................................... 19
   3.5 Poisson and Poisson Scenario Components ......................................................... 20
      3.5.1 The Gauss-Seidl and The Jacobi Iterative Solvers ....................................... 22
      3.5.2 Element-wise Operations and the Stencils ............................................... 23

4 Linear Surplus ............................................................................................................ 25
   4.1 Regions of Interest ............................................................................................ 25
   4.2 Computation of the Linear Surplus .................................................................... 29
   4.3 Elementwise Evaluation .................................................................................... 33
1 Introduction

To solve Partial Differential Equations numerically with the Finite Element Method (FEM), we discretize our domain. The finer the grid the more accurate the representation of the domain and the better the numerical approximation of the solution. There is however always a trade-off between a finer grid and runtime or memory usage. The finer the grid is the more memory is needed, and our runtime will increase due to a longer grid traversal. Since we have in the best case an $O(n)$ solver we will have an increase in the number of required iterations as well as, the $n$ from above is the total number of grid points. Knowing this, one typically switches from very fine Cartesian regular grids to a partially refined adaptive grid. A partially refined adaptive grid is a grid which is fine only in some specific regions while the rest of the domain has been left at a coarser level of refinement. By refinement we try to bring the local error at each grid point (vertex) to the same order through the whole domain. It does not make sense to spend too much time on partitions of the grid with already a rather good solution because the small errors in these partitions do not have much effect on the global error due to the higher order error within other regions.

An adaptive grid is a grid which is only refined in the those regions which need more computational effort to obtain a given accuracy. In return, if there exists any part of the grid which is finer than needed, an intelligent adaptive scheme also coarsens this to reduce the computational load. Adaptive schemes however come along with at least two drawbacks: They are more difficult to realize than regular schemes, and the regions where to coarse and where to refine have to be chosen carefully. In this thesis, we realize a dynamic adaptive scheme that identifies regions of interest automatically within the PDE framework Peano. We demonstrate its power for simple experiments with the Poisson equation, but the same idea (and the same source code) were already used successfully for computational fluid dynamics codes, too.

Peano provides a multiscale view of the computing grid, i.e. several refinement levels of the grid are available simultaneously. In this thesis, we look for an approach which tells us how much the solution profits locally from current refinement level. By knowing this we assume that the next refinement level could be of value, too, and refine further. To implement this idea, we need a metric. The metric chosen here is the linear surplus. There exists a one to one relation between the linear surplus and the second
derivative of our solution: it is basically the difference between the computed value at one grid point and the linear approximation of the value using the two neighboring points. Where the absolute value of this difference is big, the second derivative is big, too. Where the second derivative of a continuous solution is big, the solution is changing rapidly. If we approximate this solution by bi- or trilinear shape functions, we need a very fine step/grid size in such regions, i.e. regions with a high second derivative are regions of interest where a black-box refinement criterion should refine the solution automatically. The linear surplus shows us the regions which need more refinement levels to bring down the discretization error to the same level throughout the whole domain.

By the time I started this thesis, the calculation of the linear surplus had already been implemented and was being used for refinement, but there was no coarsening implemented, and the interpolation for refinement needed to be extended. Through this Master’s Thesis I added another refinement strategy to the Peano framework, completed the already implemented refinement strategy and interpolation schemes, and implemented also a coarsening strategy.

The remainder of this Master’s Thesis is organized as follows: Chapter 2 presents the challenges. Chapter 3 of this Thesis is an introduction into the Peano Framework. This chapter will describe the main features of this framework. The dynamical adaptivity has been implemented within this framework. Chapter 4 introduces the linear surplus, its relation with the second derivative, its calculation and stencils and how it is computed in Peano framework. Chapter 5 is about the two different refinement strategies implemented using the linear surplus. Chapter 6 discusses the coarsening strategy implemented within Peano framework. After each refinement we need to initialize the new vertices. For this, we use two different strategies for interpolation, as is discussed in detail in chapter 7. Chapter 8 shows some numerical experiments using the implemented strategies and some comparisons between adaptively refined grids and regularly refined grids.
2 Challenges

To define a well-suited adaptively refined grid, we need to know which regions of our domain benefit from more computational effort. If it is a time discretization we should know which time steps are the critical ones. As an example for this, one could think about the big bang theory simulation where in the first couple of hours after the big bang things happen at a fast pace. After a while things slow down. Therefore, to do such a simulation one might need a rather small time stepping strategy for the beginning but after a while a rather larger time step may be sufficient. The same holds for space discretizations. Let us assume one is solving a Poisson Equation for a point charged particle in the domain, as it is known from physics. This charged point can be described by a delta function in the Poisson equation. This point will be a singularity in the domain. In order to deal with such singularities in the domain, we need to have a rather smaller mesh size around the singularities to reduce the discretization error, while on the other regions of our domain where no singularities exist we are satisfied with a coarsed mesh size.

The challenge for having an adaptively refined grid is that we normally do not know the singularities or the regions of interest which exist in the domain before solving the equations. Unlike the examples mentioned above, this is the case most of the times. As a result, we need to have some special strategies to find out where our domain requires a very accurate representation. For this purpose, in this thesis, we study the linear surplus as it is defined independently of the actual differential equation to be solved.

There are a couple of challenging points to construct such a grid dynamically. At first, we need to know how to evaluate our grid and find the regions which need to be refined or coarsened. After the evaluation has been done and if the grid needs to be refined we need to know how we are going to initialize the newly introduced grid points. Here we have to avoid to introduce a high error which increases the runtime and the number of iterations. Before this thesis, Peano already provided a black-box refinement strategy relying on the absolute value of the linear surplus in each individual vertex. The initial guess for values for newly introduced vertices was derived from a d-linear interpolation. However, first experiments revealed that such a refinement and initialization scheme does not fit to the requirements stated above: it introduces too much noise, i.e. new errors on the additional vertices. As a starting point
for this thesis, this leads back to the initial questions how to determine the refinement criterion: If we use the linear surplus, which is the method of choice as we wanted to implement a strategy independent of the actual PDE solved, another challenge within this Master’s thesis is how to calculate the linear surplus. At first the l2-norm of the linear surplus was calculated and stored for every vertex. But while dealing with the interpolation due to the need of the direction-wise linear surplus this is changed. With the direction-wise linear surplus, we show that it is possible to implement higher-order interpolation schemes within Peano straightforwardly. After having the linear surplus, the next challenge was how to decide if a vertex has to be deleted or a cell has to be coarsened. There were different options available for this, We choose the L2-norm of the sub vertices to decide if the cell needs to be coarsened. In the interpolation schemes it happens lots of the times when we would use the already implemented high order interpolation, that the solution would not converge. After a couple of weeks it got clear to us that the boundary cells where no linear surplus is stored were the origin of this and that we had to change our high order interpolation scheme for the cells with at least on boundary vertex, i.e. a higher order interpolation scheme is not of value if vertices near the boundary are not interpolated with higher order, too. In this thesis, we will present one way how to construct higher order schemes near the boundary. Another interesting but challenging fact which we observed was that even with using an adaptively refined grid, it would happen that we would run out of memory. To avoid this we introduced another approach for refinement which would lead to a less number of grid points. While the original refinement strategy of Peano is a local one where each vertex is refined whose criterion overruns a given threshold, this thesis introduces a scheme that takes the global refinement structure into account. Classical refinement scheme in literature first analyze all the vertices/cells which have to be refined, and then select the upper k percent to be actually refined. Thus, they restrict the number of cells and vertices added per iteration. Peano’s strict locality (see dissertation [?]) prohibits such a feature as globally resorting vertices or cells does not fit into the framework’s philosophy. In this thesis however, we introduce an approach scaling the local refinement criterion with the global solution. As a result, the maximum number of additional cells and vertices is not restricted, but the increase in vertices and cells per iteration can be fine-tuned and adopted to the current solution.
3 Peano Framework

Peano is a framework with unique selling points: low memory demands, parallelism, support for adaptivity etc.. Several PDE solvers already built within it e.g. Poisson solver, heat equation, CFD\[9][6], lattice Boltzmann and etc.. [5][8][3][1].

3.1 Peano Framework Structure

The Peano framework has been structured in a way that any new feature can be added to it as easy as possible. Each part of the framework has been implemented as an independent component, and the user can decide at the compile time which features are needed for the simulation and just compile those components. There exists some dependencies between different components, e.g. Poisson-scenario and Poisson, but beside such small dependencies the rest of the components can be used independently. The different components can be seen in Figure 3.1

Peano framework follows the Adapter architecture concept and due to this Architecture it is easy to add new components. After knowing the Adapter concept one can easily add his own component or modify an existing one to improve or implement new features. The components used through this Master’s thesis are, Poisson-scenario, Poisson, Grid, Stacks, Plotter, Geometry and the Solver-toolbox.

The Poisson-scenario includes some benchmark Poisson equation problems which have been used to test the refinement and coarsening criteria implemented for this Master’s thesis, The Poisson component includes different Poisson solvers such as Jacobi, Multigrid, etc, The Grid component as the name suggests takes care of the discretization of the domain, stacks component helps one to have an efficient memory usage, plotter plots the output of the simulation and finally the Geometry is needed to represent the geometry needed for any problem with some trivial geometries already implemented in it.

As there are a couple of different components which use similar solver functionali-
ties within Peano framework, those functionalities have been encapsulated into a new component called, The Solver-Toolbox. The main component for the refinement and coarsening is the Solver-toolbox. This component includes different methods to deal with the coarsening, refinement and interpolations schemes. beside this methods, there are a couple of methods which construct the stencils needed for the solver depending on the user specified input file.

The input files are some XML files which include the problem description, the Geometry and domain description and the method used to solve the problem.

Due to the existence of different components in Peano framework, and their dependencies, It is important to have a connection between them. For example it often happens that different components use the Grid component as their underlying computational domain. But due to the different degrees of freedom they might need on the grid and different operations they might perform on each grid vertex, the Peano framework needs to have a structure which lets this be possible in software engineering point of view.

The pattern used to achieve this in Peano framework is the Adapter pattern. Using this pattern, we can implement every component independent of the others, and then we will add a couple of adapter classed which all hold the same functions but with different implementations. Each component will carry it’s own adapters to communicate with
the other components. As an example, if the Poisson component needs to use the Grid component it will communicate with it through its adapter and on the other hand if Fluid component also needs to use the Grid component it has its own adapters to do so.[2]

3.2 Grid Component

Every Grid based PDE solver, describes the domain in order to solve the PDEs. As a result of a domain description, we will end up having a grid representation of our domain. In Peano framework exists one independent Grid component which takes care of every refinement and coarsening happening on the grid and the data carried by the grid vertices as well. The more vertices the more degrees of freedom we will have and thus our computation efforts will increase. As it will be discussed more in detail in the next couple of chapters, to gain more computational accuracy we need to refine our domain, and the more we refine the more memory we need and our run time will increase. Therefore the grid component should be able to handle grids with local refinement or coarsening as well as globally refined and coarsened grids. The Grid component in Peano framework handles both rather efficiently. The Grid component stores the vertices in using a space tree, space trees will be discussed in the next sub-chapter. Due to different components which use the Grid component as their underlying computational domain, Grid component should be able to cooperate with different components via different adapters. The Adapter concept will be described in a separate sub-chapter at the end of this chapter. Beside this connections between Grid component and different other components, the Grid component should have its own methods for refinement and coarsening. This means, it should be able to add more vertices to the grid and meanwhile delete the unnecessary ones. For this manner, two different methods have been implemented within the Grid component namely \texttt{refine()} and \texttt{coarse()} which take care of this.

3.3 K-spacetree

The Peano framework uses so called k-spacetrees. In this approach we will have a root cell which will be refined over and over and each refinement will be added as a new level to a tree like data structure. In this way it will be easy to access the parents of any refined cell. If the refinements are happening as bisections we end up having a quadtree.

As it can be seen in Figure 3.2, each refinement ends up adding four new children to the quadtree. Using this we can represent the whole grid on a tree and grid traversal
3.3. K-spacetree

is just a simple tree traversal.

A quadtree is a member of a larger collection called $k$-spacetrees. For $k$-spacetrees, $k$ represents the number of cuts in each direction happening due to the refinement within the grid. In Peano any refinement will cut the cell into three in every direction. Therefore in 2D each refinement will make nine sub-cells within one cell, and in 3D this number will be 27. Therefore in the Peano framework $k = 3$. In a $k$-spacetree, the root element corresponds to the coarsest cell representing the domain. The next refinements done on the root will be added as a new level immediately under the root element. Each leaf node in the $k$-spacetree represents the cells which haven’t been refined.[8][3]

Due to local refinements, frequently some hanging nodes are introduced. Hanging nodes are the nodes which lay on the edge between the two parent vertices when the neighboring cell is not refined. In such cases the normal stencils can not be applied to these vertices since the PDE solvers operators are not defined on them. Therefore we need to find another approach to update these vertices. In the Peano framework The solver is responsible to deal with these hanging nodes. Figure 3.3 shows two hanging nodes within an adaptively refined grid.

Such nodes are only created when needed, and therefore they are not stored and after the traversal leaves the cell they will be discarded.

After one cell is refined, the new vertices will be created within the cell as well as on the parent vertices position. In this way the grid will have multiple vertices in the same position which act independent of each other. This helps when dealing with complicated boundaries or geometries. It might happen that the parent vertex is a
boundary vertex but the sub vertex belongs to the computational domain. This will be discussed more in detail in linear surplus chapter.

3.4 Grid Traversal

As every PDE solver needs a grid to apply the corresponding stencils to, a PDE solver also needs some strategies to traverse the grid. Depending on the grid structure, different traversal strategies can be implemented. In the Peano Framework, the grid is traversed in an event-based way. Operations are triggered as events such as `enter-element` or `exit-element`. Exact traversal thus is hidden. In event-based traversals a vertex can be a couple of times by different events. Using the event-based traversal, every operation performing on the grid has to plug in into an event. Table 3.1 shows the events called when creating a grid.

As mentioned before, other components will interact with the Grid component via their adapters. their adapters need to trigger the necessary event performing the operations on the grid. Table 3.2 shows the events which have to be triggered from any adapter.

As an example, The Poisson component will set the every value from the last iterations on the vertices to zero when they are called for the first time within `touchVertexFirstTime()`, the stencils will be applied on the cell vertices within `handleElement()` and, finally, the vertices will be finalized within `touchVertexLastTime()`.

3.4.1 Element-Wise Operations

In the Peano framework, when an element is loaded, all of the adjacent vertices are loaded as well. For example, in 2D when a computational cell has been loaded the four
3.5 Poisson and Poisson Scenario Components

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createDegreeOfFreedom()</td>
<td>The method receives a vertex. The specific adapter initializes this vertex and returns it. Information about position and mesh size are also passed.</td>
</tr>
<tr>
<td>createDegreeOfFreedom()</td>
<td>The method receives a geometric element. The specific adapter initializes the element and returns it.</td>
</tr>
<tr>
<td>isElementOutsideDomain()</td>
<td>This method decides if the point identified by the passed position and surrounding environment is outside the computational domain or not.</td>
</tr>
<tr>
<td>isElementInsideDomain()</td>
<td>This method decides if the point identified by the passed position and surrounding environment is inside the computational domain or not.</td>
</tr>
<tr>
<td>refine()</td>
<td>This method decides if the point identified by the passed position should be refined.</td>
</tr>
</tbody>
</table>

Table 3.1: Grid creation and management events.

vertices which define this cell are loaded and therefore components which have stencils that use the neighboring vertices need to adapt their stencils to deal with the fact that at each time only three out of eight neighboring vertices are accessibly.

Having the different events mentioned before, this becomes easy to do. the other components need to split their stencils into four different stencils in 2D.

3.5 Poisson and Poisson Scenario Components

There are two components implemented in Peano framework To solve a certain type of PDEs, Poisson Equation. The Poisson component has a couple of different solvers implemented while Poisson Scenario component has a couple of predefined benchmarks to run the Poisson component. If another right hand side or boundary condition for the Poisson equation needs to be solved then all is needed to do is to add a new class to the Poisson Scenario component describing the right hand side and boundary conditions.

As mentioned before there are a couple of different solvers implemented within the Poisson component such as, Gauss-seidl, Damped Jacobi and Multigrid. Since for this Thesis the Multigrid solver has been chosen as the test solver for the Refinement and Coarsening criteria Therefore I will describe this Solver in more detail.
### 3.5. Poisson and Poisson Scenario Components

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginTraversal()</code></td>
<td>Start of an iteration.</td>
</tr>
<tr>
<td><code>endTraversal()</code></td>
<td>End of an iteration.</td>
</tr>
<tr>
<td><code>enterElement()</code></td>
<td>All necessary vertices for a cell are loaded.</td>
</tr>
<tr>
<td><code>leaveElement()</code></td>
<td>Counterpart to <code>enterElement()</code>.</td>
</tr>
<tr>
<td><code>touchVertexFirstTime()</code></td>
<td>The first time any cell next to this vertex is visited. The position, level and corresponding mesh width are passed.</td>
</tr>
<tr>
<td><code>touchVertexLastTime()</code></td>
<td>The last cell next to this vertex has been visited. At this step all geometric elements of this vertex have been visited.</td>
</tr>
<tr>
<td><code>loadSubElement()</code></td>
<td>Fired before going to the next finer level (top-bottom transition).</td>
</tr>
<tr>
<td><code>storeSubElement()</code></td>
<td>Fired before going to the next coarser level (bottom-top transition). This operation and its counterpart facilitate an interlevel information exchange.</td>
</tr>
<tr>
<td><code>startStepsUp()</code></td>
<td>Fired after the subelements have been processed and before the traversal will step up the tree.</td>
</tr>
<tr>
<td><code>startStepDown()</code></td>
<td>Fired right before the subelements are going to be entered.</td>
</tr>
<tr>
<td><code>createPersistentVertex()</code></td>
<td>Used to create new vertices when refining an existent vertex.</td>
</tr>
<tr>
<td><code>destroyPersistentVertex()</code></td>
<td>Used to destroy vertices following a coarsening.</td>
</tr>
<tr>
<td><code>createTemporaryVertex()</code></td>
<td>Used to create hanging nodes (touch first on hanging nodes).</td>
</tr>
<tr>
<td><code>destroyTemporaryVertex()</code></td>
<td>Used to destroy hanging nodes.</td>
</tr>
</tbody>
</table>

*Table 3.2: Adaptive regular Cartesian grid management and traversal events.* [3]
3.5. Poisson and Poisson Scenario Components

3.5.1 The Gauss-Seidel and The Jacobi Iterative Solvers

After discretization our domain we need to find a way to solve our Poisson Equation on the resulting grid. We could start by taking a closer look at the Laplace Operator, and try to bring it to a finite difference representation. Let’s assume while solving our physical problem we reach to a point that we need to solve the following Poisson equation. To solve this equation numerically we need to bring it into finite difference notation.

\[
\nabla u(x) = \frac{\partial^2 u(x)}{\partial x^2} = f(x) \quad \text{on} \quad \Omega = [a, b]
\]

\[f(x_i) = \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2}, \quad h = \frac{b-a}{n}, \quad n: \text{Number of intervals.} \quad (3.2)
\]

Now we can replace the continuous Poisson equation with a linear system of equations where the unknowns are \(u(x_i)\) at the grid points.

\[
\begin{pmatrix}
2 & -1 & & \\
-1 & 2 & -1 & \\
& \ddots & \ddots & \\
& & -1 & 2 & -1 & \\
& & & -1 & 2 & \\
\end{pmatrix}
\frac{1}{h^2}
\begin{pmatrix}
u_1 \\
u_2 \\
\vdots \\
u_{n-1} \\
u_n
\end{pmatrix}
=
\begin{pmatrix}
f_1 \\
f_2 \\
\vdots \\
f_{n-1} \\
f_n
\end{pmatrix}
\rightarrow Au = f \quad (3.3)
\]

Now we can use the equation (3.2) and solve it for \(u_i\).

\[
u_i = \frac{h^2f(x_i) + u_{i-1} + u_{i+1}}{2} \quad (3.4)
\]

Using equation (3.4) we can think of an iterative solver running through our entire grid, applying this equation and updating the new value for \(u_i\). Depending on how this update is being done we will have a Gauss-Seidel or Jacobi solver. For Gauss-Seidel the updated value for \(u_i\) will immediately be used in the same iteration when updating \(u_{i+1}\), as for Jacobi this value will be used in the next iteration and for updating \(u_{i+1}\) in the current iteration we will use the value for \(u_i\) from the last iteration.

Even though Gauss-Seidel and Jacobi solvers help us solve our Poisson equation, their convergence is of second order to the number of grid points, \(O(n^2)\), and therefore
the more grid points we have the slower our solver gets. For this reason another type of
solver which uses modified Jacobi or Gauss-Seidl as it’s smoother has been implemented
within Peano framework.[4]

3.5.2 Element-wise Operations and the Stencils

Let’s assume the five-point stencil which occurs often in Poisson equation iterative
solvers.

\[
\frac{1}{h^2} \begin{pmatrix} -1 & 4 & -1 \\ -1 & 4 & -1 \\ -1 & 4 & -1 \end{pmatrix}
\]

This stencil can be split up into the following 4 stencils:

\[
\frac{1}{h^2} \begin{pmatrix} 0 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{pmatrix}, \quad \frac{1}{h^2} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}
\]

\[
\frac{1}{h^2} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \quad \frac{1}{h^2} \begin{pmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 0 \end{pmatrix}
\]

Depending on the position of the vertex within the cell one of these stencils will
be applied to the vertex. when the touchVertexLastTime event has been triggered, the
vertex which is being touched for the last time has already had all of these stencils
applied and is ready to be finalized.
3.5. Poisson and Poisson Scenario Components
4 Linear Surplus

As the error of the solution to our problem is unknown, besides simple artificial benchmark problems, in order to be able to refine or coarsen our grid we need to have some estimates for the error. There are plenty of measures to decide where to refine. The measure used in this thesis is the linear surplus. Implementing Linear surplus is easy and it’s computation does not increase the run-time much. In this chapter I will talk about the regions of interest, how to find them, how to compute the linear surplus and how it is done in Peano framework.

4.1 Regions of Interest

A good measure to evaluate the computational discretization on the grid should find the regions which could be of interest to us. It should tell us which regions need to be refined more or can be coarsened. To obtain such a measure, we consider the difference between the computed solution on two different grids, grids with mesh sizes $h$ and $2h$. Having the solution on two different grids helps us to identify which regions are well-presented on which mesh size. On one hand comparing the coarser grids to the current fine grid helps us to understand how well the solution has been improve by this refinement and if another level of refinement is necessary: On the other hand, if we see not much of a change has happened to the solution due to the finer grid, we know that this level of refinement was unnecessary and we can coarsen the grid on that specific region of our domain.

On the other hand we can reconstruct the second derivative. We take the second derivative of the computed solution into account, and assume that regions with a high second derivative need more computational effort i.e. a finer grid. There is a couple of reasons to choose the second derivative as a measure, and the next couple of pages will be more focused on these reasons.

To find out the worthiness of the second derivative, lets take a look at a couple of case studies and see how the second derivative can help us to find the regions of
4.1. Regions of Interest

Figure 4.1: Coarse Grid Representation vs. Fine Grid Representation for Equation (4.3)

interest. Hereby we concentrate on the Poisson Equation, which is given by:

\[- \Delta u(x, y, z) = f(x, y, z) \quad \text{on} \quad \Omega = [a, b] \]

(4.1)

with \( \Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \).

(4.2)

In this thesis we always approximate our solution with piecewise linear approximations, therefore the second derivative tells us more about the difference between two successive approximations. When the second derivative is high in a certain region, this means that the solution in that region is changing with a rather high frequency, and therefore this region could be of interest. In other words, when the second derivative of our solution changes sign or changes rapidly between two grid points, then, depending on how big the change is, our linear approximation might not be sufficient enough to describe our solution in that region and therefore we might need to introduce more grid points to avoid this fact. To illustrate this, we can take a look at the next couple of examples.

\[ \Delta y(x) = 6x \]

with \( y(x) = x^3 \).

(4.3)

As it can be seen in this example, the second derivative is in the same order as \( x \). Therefore the grid to represent this equation should be the finer the more \( |x| \) increases. Figure 4.1 shows how a finer grid represents this function better. Comparing the coarse and fine grid representations one sees that the regions A and C have a high second derivative while region B which has a rather low second derivative. Therefore one should choose to refine regions A and C and leave region B as it is. Figure 4.2 shows the second derivative of Equation (4.3).

Another simple yet helpful example is (4.4). As it can be seen in this example, the
4.1. Regions of Interest

Figure 4.2: Second Derivative for Equation (4.3)

\[ \Delta y(x) = -2 \]
with \( y(x) = -x^2 \). \hspace{0.5cm} (4.4)

Figure 4.3: Coarse Grid Representation vs. Fine Grid Representation for Equation (4.4)

second derivative is always constant “6”, therefore, in contrast to the last example, we will need a uniformly refined grid to represent the solution for this example.

Figure 4.3 shows the solution represented on two different grids for (4.4). Since the second derivative of our solution is constant, there is no specific region which needs to be more refined than other regions, and, thus, the grid needs to be globally refined to the same accuracy.

Another interesting example is when there exists a point on the domain with second derivative of infinity. In such situations, no matter how fine the grid is already, the solution needs a finer grid. This example shows that in some rare cases considering the second derivative only may not be sufficient or may cause in a infinite sequence of refinements.
4.1. Regions of Interest

Figure 4.4: Coarse Grid Representation vs. Fine Grid Representation for Equation (4.5)

Figure 4.5: Second Derivative for Equation (4.5)

\[ \Delta y(x) = -\frac{1}{x^2} \]
with \( y(x) = \ln(|x|) \). \hspace{1cm} (4.5)

As it can be seen in Figure 4.5, the region close to \( x = 0 \) is the critical point since the second derivative at this region is infinity. No matter how fine the grid is, this value will still affect the computation and cause our simulation to refine more and more! Therefore, the regions A and C do not need a very fine grid but the region B needs to be very fine.

The last example is the Laplace Equation with the right hand side being zero, this could be an interesting example.

\[ \Delta y(x) = 0 \]
with \( y(x) = ax \). \hspace{1cm} (4.6)

Figure 4.6 shows the solution and the second derivative plot for this function. As
it is seen, to represent this function only two points would be sufficient as the second derivative is everywhere zero.

![Figure 4.6: Solution and the derivative for Equation (4.6)](image)

### 4.2 Computation of the Linear Surplus

As described in the last chapter, second derivative of our solution is a good measure to find the critical parts in our domain. There exists a one to one analogy between the second derivative and the linear surplus for linear elements and, therefore a linear surplus is also a good measure to evaluate refinement or coarsening criteria. In this section, we reveal the analogy. Then, all left for us to do is to calculate it through our grid traversal. The linear Surplus (in 1D) is just the average of the two left and right neighboring point of a vertex compared to the value of the vertex. This measure tells us how much we have improved our solution by this refinement. As described in the last section, the average of the two right and left neighbors is the piecewise linear solution to our middle grid point and therefore this value can be of much help to us.

The comparison can be done either by division or subtraction, the first one is not recommended, since if one of the values is close to zero then the result of the division is unknown or not useful. In this thesis, the later one has been used to calculate the linear surplus for each vertex.

The linear surplus gives us an idea on how accurate our representation of the grid is and where we need to refine or coarsen further. Using Taylor Series one can prove the relation between the linear surplus and the second derivative of the solution.
4.2. Computation of the Linear Surplus

Figure 4.7: Linear Surplus

\[
f(x_i) = f(x_{i+1}) + f'(x_{i+1})(x_i - x_{i+1}) + \frac{f''(x_{i+1})}{2}(x_i - x_{i+1})^2 + \ldots \tag{4.7}
\]

\[
f(x_{i+1}) \approx f(x_i) - f'(x_{i+1})(x_i - x_{i+1}) - \frac{f''(x_{i+1})}{2}(x_i - x_{i+1})^2 \tag{4.8}
\]

\[
f(x_{i+1}) \approx \frac{1}{2}(f(x_i) + f(x_{i+2}) - f'(x_{i+1})(x_i - x_{i+1}) - f'(x_{i+1})(x_{i+2} - x_{i+1})
- \frac{f''(x_{i+1})}{2}(x_i - x_{i+1})^2 - \frac{f''(x_{i+1})}{2}(x_{i+2} - x_{i+1})^2) \tag{4.9}
\]

Inserting \(x_{i+1} = x_i + \frac{h}{2}\) and \(x_{i+2} = x_i + h\) and some refactoring (4.9) leads to (4.10)

\[
f(x_{i+1}) - \frac{f(x_i) + f(x_{i+2})}{2} = \frac{h^2}{2} f''(x_{i+1}) = S(x_{i+1}) \tag{4.10}
\]

\[
\Rightarrow \tilde{u}_i = \left| \frac{\hat{u}_{i-1} + \hat{u}_{i+1}}{2} - \hat{u}_i \right| \tag{4.11}
\]

This in TUM leads to the so called 3-point stencil for linear surplus calculation in 1D. Using the same approach for 2D and 3D, one can derive the 5-point and 7-point stencils.

\[
f(r + a) = f(r) + (a \cdot \nabla r)f(r')|_{r'=r} + \frac{1}{2} a \cdot \left[ a \cdot \nabla r' \left( \nabla r' f(r') \right) \right]_{r'=r} + \cdots \tag{4.12}
\]
4.2. Computation of the Linear Surplus

Figure 4.8: Five Point Stencil Cell

\[
\begin{align*}
\quad f(E) &= f(A) + h(f(A) - f(D)) + \frac{h}{2} f''(E) \\
\quad f(E) &= f(D) - h(f(A) - f(D)) + \frac{h}{2} f''(E) \\
4f(E) &= f(A) + f(B) + f(C) + f(D) + 2hf''(E) \\
\frac{h}{2} f''(E) &= f(E) - \frac{f(A) + f(B) + f(C) + f(D)}{4}
\end{align*}
\] (4.13)

The same calculations can be done to derive the five point skewed stencil. Just a rotation transformation of 45 degrees has to be applied to our grid points.

\[
\begin{pmatrix}
\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\
\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}
\end{pmatrix}
\] (4.14)

\[
\begin{pmatrix}
0 & -1/4 & 0 \\
-1/4 & 1 & -1/4 \\
0 & -1/4 & 0
\end{pmatrix}
\begin{pmatrix}
-1/4 & 0 & -1/4 \\
0 & 1 & 0 \\
-1/4 & 0 & -1/4
\end{pmatrix}
\] (4.15)

By summing and normalizing the above two stencils we end up with full stencil.

\[
\begin{pmatrix}
-1/8 & -1/8 & -1/8 \\
-1/8 & 1 & -1/8 \\
-1/8 & -1/8 & -1/8
\end{pmatrix}
\] (4.16)
4.2. Computation of the Linear Surplus

One could calculate the linear surplus for any arbitrary point within the domain but the question is how useful would that be? In this Thesis we calculate it only at the grid vertices although it could be calculated for any point within the domain by interpolation. In Peano framework, this has been applied to every dimension independently. Meaning, for each vertex a vector for linear surplus is stored. In this way one can construct any norm of the linear surplus. Further more, having a vector of linear surplus will be of use in the higher order interpolation scheme.

Another feature which has been implemented within Peano framework is that the calculation of the linear surplus happens on $2^d$ coarser grids, which helps us evaluate if the next finer grid is necessary. In this way for example in 1D for each fine grid we will have 2 coarser grids which help us evaluate the fine grid and decide which regions are well presented on the fine grid, which regions need to be refined and which regions have to be coarsened to avoid unnecessary computational efforts. The two different coarse grids have been shown in Figure 4.10. As seen in the Figure 4.11 this is done for every vertex in the grid in every iteration.

There exists four different coarse grids in 2D for every fine grid, as for 3D this number is eight. Therefore for a d-dimensional problem, $2^d$ 1-level-coarser grid are needed to evaluate our fine grid.
4.3 Elementwise Evaluation

As described in the last chapter, in Peano the solver has only the data from one cell available at each time: there is no information from the neighboring cell available at any time. Therefore we need a strategy to evaluate the solution updates as well as the linear surplus for every vertex.

To do so, at every cell, $\frac{1}{2}$ of the contribution from each vertex to another one is calculated and accumulated. When the vertex is being touched for the last time ($4^{th}$ time in 2D and $8^{th}$ time in 3D), all the contributions have been accumulated and the value for the vertex can be finalized.

Let the cell numbers in Figure 4.12 define the order of the cell evaluations. Within cell number five the vertex number 5 is being touched the last time. Therefore the value accumulated for this vertex is the final value and the value for this vertex can be finally updated. For vertex number ten, this is not the case as it has just been touched for the first time. Therefore, at this point all the old values stored for it’s e.g. linear surplus.
surplus from the last iteration can be set to zero to start calculating the new values for this iteration. Vertex number 10 will be touched for the last time when the cell number 9 is entered.

With an element-wise processing we split our stencils up into 4 different stencils. Each corresponding to a different contribution. (4.17) show the resulting stencils after splitting the 5-point and 5-point-skewed stencils.

\[
\begin{pmatrix}
0 & -1/8 & 0 \\
0 & 1/4 & -1/8 \\
0 & 0 & 0 \\
0 & 1/4 & -1/8 \\
0 & -1/8 & 0 
\end{pmatrix}
\begin{pmatrix}
0 & -1/8 & 0 \\
-1/8 & 1/4 & 0 \\
0 & 0 & 0 \\
-1/8 & 1/4 & 0 \\
0 & -1/8 & 0 
\end{pmatrix}
\begin{pmatrix}
0 & 0 & -1/4 \\
0 & 1/4 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 
\end{pmatrix}
\begin{pmatrix}
0 & 1/4 & 0 \\
-1/4 & 0 & 0 \\
0 & 1/4 & 0 \\
-1/4 & 0 & 0 \\
0 & 1/4 & 0 
\end{pmatrix}
\]

(4.17)

Having these stencils (4.17), we can use the position of the vertex in the cell and use the corresponding stencil to accumulate the values from the other vertices. At the end, when the vertex is touched for the last time, we can update the value.

![Figure 4.13: Computational Cells in Peano](image)
4.4. Multiple Vertices and Boundary Treatments

As the grid gets refined, every existing vertex on the coarse grid will be created again as an individual vertex on the finer grid. If we have \( n \) levels of local refinements at a certain vertex, then that vertex will exist on all the \( n \) finer grids independent of the other ones. Therefore it carries its own value as well as linear surplus. Without having these multiple vertices the solver won’t distinguish between the different grid levels while solving the problem.

This fact will be even better seen when dealing with boundaries. If a vertex on the coarser grid belongs to the boundary but on the finer one belongs to the computational domain, then only the finer one will carry a linear surplus. This has been illustrated in Figure 4.14.

When such cases occur within the domain, the idea of having multiple independent vertices on different grids becomes helpful. As seen in Figure 4.14, the vertex marked is a boundary vertex on the coarse grid but on a one level finer grid it no longer belongs to the boundary and is lying within the computational domain. As a consequence of this feature one can easily represent any type of geometry by refining the boundary more and more.
4.4. Multiple Vertices and Boundary Treatments

Figure 4.14: Boundary elements in Coarse and Fine grids
5 Refinement Strategies

After having the Linear Surplus at every grid point now it’s time to find some strategies to use it as a measure for refinement. A good reason for the need of refinement is that due to the nature of discretization our error is normally at least of order two, \( O(h^2) \), and therefore no matter how hard we try to improve our solver this error will always be there and will often be the dominant part.

\[
\epsilon_{\text{total}} = \epsilon_{\text{computation}} + \epsilon_{\text{discretization}} \tag{5.1}
\]

\[
O(\epsilon_{\text{total}}) \geq O(\epsilon_{\text{discretization}}) \tag{5.2}
\]

5.1 Refinement Criteria

The aim for a refinement criterion would be to bring the local linear surpluses to the same order of magnitude through out the whole domain. By doing this, we can be sure that our domain representation is homogeneous with respect to the discretization error. There have been two different refinement criteria implemented in Peano Framework. Both of them result in a locally refined grid.

5.1.1 Refinement Veto

Before our solution converges there might exist some high frequency errors which can affect the linear surplus value for every vertex resulting in an incorrect refined grid. A feature implemented within Peano Framework is that the user can specify when the refinement is allowed to take place. The user also defines a threshold for the residual, and as long as the residual is above this threshold the grid is not allowed to be refined. With this feature the user can decide if the grid has to be refined in the beginning or before or after convergence.

Figure 5.1 shows a grid generated before the convergence for the Poisson equation...
5.1. Refinement Criteria

\[-\Delta u = 0.\] As expected and described in linear surplus chapter, one would expect a uniformly refined grid as the second derivative of our solution is constant.

![Figure 5.1: Generated Grid Before Convergence](image)

Using the Refinement Veto we make sure that the grid is refined once the solution has converged or is close to convergence. Figure 5.2 shows how this feature can change the generated grid.

5.1.2 Refinement Threshold

This approach lets the user define an upper limit for the linear surplus of any vertex. So any vertex with a higher linear surplus than the user-specified threshold is refined. The remaining will be left unchanged.

So, in the `touchVertexLastTime` the code checks the linear surplus of the vertex with the user given threshold. If it is larger than the threshold, the vertex will be flagged for refinement. The pseudo code of the function which checks this threshold is given by Algorithm 5.1:

For time-dependent scenarios, the grid has to be updated within every time step. The reason for that is, if we are dealing with a dynamic scenario then the domain changes at every time step and, therefore, the grid has to be adapted to the new situation. For the static scenarios constructing the adaptive grid in the beginning of
the calculation is sufficient since no changes happen to the domain through out our simulation.

Figure 5.5 shows the generated grid for a static scenario. The problem solved here is

\[- \Delta u = c \left( 128 \pi r^2 \sinh(64 \pi (2 - r^2)) - 3 \cosh(64 \pi (2 - r^2)) \right)\]

with \( r^2 = \sum_i 1^d \left( x_i - \frac{1}{3} \right)^2 , c = - \frac{128 \pi}{\sinh(128 \pi)} \).
5.1. Refinement Criteria

**Algorithm 5.1** Algorithm to Check the Refinement Threshold

```plaintext
getRefinementThreshold() { Gets the user defined Refinement Threshold }
ScaleLinearSurplus() { Scales the linear surplus of the vertex with respect to the mesh size }
3: CompareWithTheThreshold() { Compares every dimension of the Linear surplus with the Threshold }
UpdateFlag() { Updates the Refinement flag after each dimension has been checked }
if(flag) Refine() { Refines if the refinement flag is true for any direction }
```

The analytical solution to this problem is

\[ u = \frac{1}{\sinh(128\pi)} \sinh \left( 64\pi (2 - r^2) \right) \]  

(5.3)

The refinement threshold for this example has been set to $1e^{-05}$ and as it can be seen the starting grid has been refined three levels locally to bring the linear surplus to the same order through the domain.

![Initial Grid](image)

**Figure 5.4:** Initial Grid
5.1.3 Refinement Percentage

After we had implemented the refinement threshold, we decided to add another approach for refinement. Even though using adaptively refined grids helps us to reduce the number of vertices and degrees of freedom within our simulation which leads to a lower memory consumption, but if the domain gets larger and it has more critical regions, we will still be using a large portion of our memory and sometimes might even run out of memory. Therefore the need of another approach to refine the grid with less vertices was felt. To have a less refined grid, we used the linear surplus of the vertices. But this time, the user defines the ratio the linear surplus of a vertex should be to the max linear surplus on the grid to be refined. In other words, the user gives a percentage in his configuration file to the Peano Framework, and in touchVertexLastTime if the vertex has a linear surplus of that ratio or higher to the Max linear surplus, it will be refined.

Figure 5.5: Generated Adaptive Grid with Refinement Threshold
5.1. Refinement Criteria

Figure 5.6: Refinement Percentage

Figure 5.7: Generated Adaptive Grid with Refinement Percentage specified

Depending on the user specified ratio, this might lead to a less refined grid than the Refinement Percentage, and therefore a faster grid traversal. But on the other hand, it might happen that the discretization error from this approach get larger.

Figure 5.7 shows a refined grid. The initial grid can be seen in Figure 5.4. The
initial grid has been refined twice. As is can be seen in this Figure and mentioned before, each grid refinement in the Peano Framework divides the cell into 9 sub-cells.
5.1. Refinement Criteria
6 Coarsening

In contrast to refinement, we might be interested in coarsening our grid where the solution is smooth and can be represented by a coarse grid. There is no need to have a fine grid when the change from one coarse grid point to the other is not significant.

6.1 Coarsening Criterion

As for refinement, for the coarsening we also have a threshold. But this time, it specifies the lowest linear surplus a vertex should have. If a vertex has a lower linear surplus than the threshold, this means that more computational effort is put in that region of the domain than needed to reach the required accuracy.

There is a difference between this threshold and the one for refinement. For the refinement, the linear surplus of the vertex would be compared with the threshold, but for coarsening, we need to consider the sub-vertices.

In order to consider the sub-vertices we need to find a way to accumulate their linear surplus and compare it to the threshold. To do so, the L2-norm of the linear surplus of all the sub-vertices will be stored on the parent vertex. and at touchVertexLastTime
6.1. Coarsening Criterion

for the higher coarser vertex, we calculate the L2-norm of it and compare it to the threshold.

\[ S(X_{i,j}) = \frac{1}{N} \sqrt{\sum_{k=0}^{N} C^2(x_k)} \quad N: \text{Number of Sub Vertices} \quad (6.1) \]

\[ C(x_{i,j}) = \frac{1}{N} \sqrt{\sum_{k=0}^{D} S^2_k(x_{i,j})} \quad D: \text{Dimensions.} \quad (6.2) \]

After \( S(X_{i,j}) \) has been calculated for the parent vertex completely, this value will be compared to the threshold. If this is smaller than the threshold, then the vertex is coarsened.

If \( S(X_{i,j}) \leq Threshold \implies \text{Coarsen.} \)
Algorithm 6.1 Algorithm to Check the Coarsening Threshold

\begin{verbatim}
getCoarseningThreshold() { Gets the user defined Coarsening Threshold}
AccumulateSubVerticesLinearSurpluses() { Sums up the sub-Vertices Linear Surplus and Stores Them in the Parent Vertex}
3: CompareWithTheThreshold() { When the Parent Vertex is Touched the Last Time it checks the sum with the Threshold }
UpdateFlag() { Updates the Coarsening flag after each dimension has been checked }
if(flag) Coarse() { Coarsens if the Coarsening Flag is True}
\end{verbatim}

Another restriction which has been implemented within Peano is that, at any time, only one level of coarsening is allowed. If any vertex has more than one level of refinement, then the code is not allowed to trigger coarsening on that vertex before it’s sub-vertices have been coarsened. To do that we just set the \( S_k \) of the sub-vertices linear surplus to infinity through the coarsening analysis and therefore it will always be higher than the threshold.

\[
\text{if more than one level of children } \iff S(X_{i,j}) \rightarrow \infty \tag{6.3}
\]

The reason for this is that at each time the code has access to one level of refinement for each vertex, and if we coarsen this vertex then it might happen that some of the children of the sub-vertices for this vertex have high linear surplus and in this way we guarantee that those vertices won’t be lost.
6.1. Coarsening Criterion

Figure 6.4: Resulting Grid After Coarsening

Figure 6.4 shows the resulting grid after coarsening for (5.3). For this example the threshold has been chosen to be $1.0e^{-03}$. 

7 Interpolation

After a refinement, the new vertices have to be initialized. For this, there are two different approaches implemented in the Peano Framework: bilinear interpolation and cubic interpolation. The user can choose between these two approaches. With piecewise linear shape function, the bilinear/trilinear interpolation is a natural choice due to its simplicity. However, the simple linear interpolation has two significant drawbacks. First, the convergence estimates for F-cycles require the prolongation for a new grid level to belong to an order bigger than the smoothing stencil itself [7]. And, second, if we take a closer look to our solver, we will see that the data transport for our smoothers is one cell per iteration. Due to the fact that Peano implements three-partitioning the bilinear interpolation introduces a high order error in the middle of the newly refined cell. And it might take a lot of iterations till this error component is reduced. Therefore, we need a higher order interpolation scheme to avoid such scenarios. As described in the linear surplus chapter, the linear surplus for every vertex is stored in a vector approach to distinguish the different linear surplus directions. It will be shown in this chapter, why and how this vector helps us to construct a higher order interpolation scheme. First however, we discuss the standard bilinear scheme.

7.1 Bilinear Interpolation

The idea behind this interpolation is simple. Whenever a new vertex is introduced into the grid in 1D, the average of the neighboring points will be the value set for the new vertex.

Figure 4.1 shows this scheme in 1D, in 2D instead of linear interpolation bi-linear interpolation is applied.

In the bilinear interpolation, all the parent vertices contribute to the new vertex created Figure 7.2. The weight depends on their distance to the new vertex. The bilinear interpolation is of second order. Equation (7.1) shows this fact.
7.1. Bilinear Interpolation

Figure 7.1: Linear Interpolation

Figure 7.2: Bilinear Interpolation
7.2. Cubic Interpolation

\[
f(x, y) = (1 - \beta)[(1 - \alpha)f_{i,j} - \alpha f_{i+1,j}] + \beta[(1 - \alpha)f_{i,j+1} + \alpha f_{i+1,j+1}]
\]
\[
= (1 - \beta)f_j + \beta f_{j+1} \tag{7.1}
\]
\[
f_j = (1 - \alpha)f_{i,j} + \alpha f_{i+1,j} \tag{7.2}
\]
\[
f_{j+1} = (1 - \alpha)f_{i,j+1} + \alpha f_{i+1,j+1} \tag{7.3}
\]
\[
\alpha = \frac{x - x_i}{x_{i+1} - x_i} \quad \beta = \frac{y - y_i}{y_{i+1} - y_i} \quad \alpha, \beta \in [0, 1] \tag{7.4}
\]

As already mentioned in the previous chapters, in Peano framework any refinement splits the cell into 3 sub cells in 1D, thus this makes it easier to evaluate the bilinear interpolation. Knowing this we can derive the stencils with which the new vertices are going to be initialized and just apply it as soon as they are created. As the cell is divided into three equal intervals in every direction, therefore, the distance between the newly introduced vertices and the old ones is always known as a constant factor of our original mesh size \( h \). This fact can help us derive the weights with which the new vertices will be initialized once and use them as they are created. Equations (7.5), (7.6) and (7.7) show the corresponding stencils in 1D and eight of the 16 stencils for 2D.

\[
\begin{align*}
\frac{1}{3} \begin{bmatrix} 1 & 2 \\ \end{bmatrix} & \quad \frac{1}{3} \begin{bmatrix} 2 & 1 \\ \end{bmatrix} & \quad \begin{bmatrix} 1 & 0 \\ \end{bmatrix} & \quad \begin{bmatrix} 0 & 1 \\ \end{bmatrix} \\
\frac{1}{9} \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ \end{bmatrix} & \quad \frac{1}{9} \begin{bmatrix} 2 & 1 \\ 1 & 2 \\ \end{bmatrix} & \quad \frac{1}{9} \begin{bmatrix} 4 & 2 \\ 2 & 1 \\ \end{bmatrix} & \quad \frac{1}{9} \begin{bmatrix} 2 & 1 \\ \end{bmatrix} \\
\frac{1}{3} \begin{bmatrix} 2 & 1 \\ 0 & 0 \\ \end{bmatrix} & \quad \frac{1}{3} \begin{bmatrix} 0 & 0 \\ 2 & 1 \\ \end{bmatrix} & \quad \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ \end{bmatrix} & \quad \begin{bmatrix} 0 & 1 \\ \end{bmatrix}
\end{align*}
\tag{7.5}
\]

\[
\begin{align*}
\frac{1}{9} \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ \end{bmatrix} & \quad \frac{1}{9} \begin{bmatrix} 2 & 1 \\ 1 & 2 \\ \end{bmatrix} & \quad \frac{1}{9} \begin{bmatrix} 4 & 2 \\ 2 & 1 \\ \end{bmatrix} & \quad \frac{1}{9} \begin{bmatrix} 2 & 1 \\ \end{bmatrix} \\
\frac{1}{3} \begin{bmatrix} 2 & 1 \\ 0 & 0 \\ \end{matrix} & \quad \frac{1}{3} \begin{bmatrix} 0 & 0 \\ 2 & 1 \\ \end{matrix} & \quad \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ \end{bmatrix} & \quad \begin{bmatrix} 0 & 1 \\ \end{bmatrix}
\end{align*}
\tag{7.6}
\]

\[
\frac{1}{3} \begin{bmatrix} 2 & 1 \\ 0 & 0 \\ \end{bmatrix} & \quad \frac{1}{3} \begin{bmatrix} 0 & 0 \\ 2 & 1 \\ \end{bmatrix} & \quad \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ \end{bmatrix} & \quad \begin{bmatrix} 0 & 1 \\ \end{bmatrix}
\tag{7.7}
\]

7.2 Cubic Interpolation

The idea for having a higher order interpolation is to use more vertices than the four coarse vertices to calculate the value for the newly introduced vertex. We first restrict to the one-dimensional case: With four coarser points, we can approximate the value of the new vertex by finding the curve which passes through the four points. To construct this curve, we need to set the value of this curve to at the four given vertices to the actual value of these vertices. Equation (7.8) is the spline which passes through the four points shown in Figure 7.3.

\[
p(x) = \frac{1}{6}(1 - x)(2 - x)(3 - x)U_0 + \frac{1}{2}x(2 - x)(3 - x)U_1 \\
+ \frac{1}{2}x(3 - x)(x - 1)U_2 + \frac{1}{6}x(x - 2)(x - 1)U_3 \tag{7.8}
\]
7.2. Cubic Interpolation

Figure 7.3: Cubic Interpolation

$p(x)$ yields the original values at the existing grid points. In the Peano framework, every time a cell is refined in 2D it will be divided into 9 sub-cells. In 3D into 27 sub-cells. For simplicity, I will calculate the weights for the contribution of each vertex in 1D. Calculating them for 2D and 3D is a straightforward tensor product. As shown in Figure 7.3, we calculate the contribution of vertices number 0, 1, 2, 3 on the newly introduced vertices $a$ and $b$. Vertex $a$ is $\frac{1}{3}h$ to the right of vertex 1. Therefore we need to evaluate the value of $p(x)$ at $x = 1 + \frac{1}{3}$. Vertex $b$ however, is $\frac{2}{3}h$ to the right of vertex 1. Therefore we need to evaluate the value of $p(x)$ at $x = 1 + \frac{2}{3}$.

\[
p(1 + \frac{1}{3}) = \frac{1}{81}[-5U_0 + 60U_1 + 30U_2 - 4U_3] \quad (7.9)
\]
\[
p(1 + \frac{2}{3}) = \frac{1}{81}[-4U_0 + 30U_1 + 60U_2 - 5U_3] \quad (7.10)
\]
\[
p(1) = U_1 \quad (7.11)
\]
\[
p(2) = U_2 \quad (7.12)
\]

As in can be seen from Equations (7.10) and (7.11), in 1D we will have only four stencils to calculate the cubic interpolation. These stencils belong to the four possibilities a vertex can have within the parent cell. These four stencils are shown in Equation (7.13).

\[
A = \frac{1}{81}[-5 60 30 -4] \quad B = \frac{1}{81}[-4 30 60 -5] \quad C = [0 1 0 0] \quad D = [0 0 1 0] \quad (7.13)
\]

Now we need to do a tensor product between these two stencils to get the 16 different stencils for 2D. The 16 stencils come from 16 possibilities to do the tensor products, $A \otimes B$, $B \otimes A$, $A \otimes A$, $B \otimes B$, etc. Those give us the 16 different possibilities which occur for refinement in 2D. These first four tensor products result in the following four stencils.
7.2. Cubic Interpolation

These stencils are for the inner vertices, but for the vertices lying on the edge of the cell we have 12 additional stencils. The ones lying exactly on the parent vertices, will simply get their values from the parent vertices, the ones lying on the edge between two of the parent vertices will have eight stencils. In (7.16) and (7.17) I display four of them, the other four will be the same as the ones shown in (7.16) and (7.17) but with the corresponding rotation.

\[
\begin{align*}
\frac{1}{3^4} \begin{pmatrix} 20 & -240 & -120 & 16 \\ -150 & 1800 & 900 & -120 \\ -300 & 3600 & 1800 & -240 \\ 25 & -300 & -150 & 20 \end{pmatrix} & \quad \frac{1}{3^4} \begin{pmatrix} 20 & -150 & -300 & 25 \\ -240 & 1800 & 3600 & -300 \\ -120 & 900 & 1800 & -150 \\ 16 & -120 & -240 & 20 \end{pmatrix} \\
\frac{1}{3^4} \begin{pmatrix} 25 & -300 & -150 & 20 \\ -300 & 3600 & 1800 & -240 \\ -150 & 1800 & 900 & -120 \\ 20 & -240 & -120 & 16 \end{pmatrix} & \quad \frac{1}{3^4} \begin{pmatrix} 16 & -120 & -240 & 20 \\ -120 & 900 & 1800 & -150 \\ -240 & 1800 & 3600 & -300 \\ 20 & -150 & -300 & 25 \end{pmatrix}
\end{align*}
\]

\[(7.14)\]

In the Peano framework, to evaluate such stencils in 2D is rather tricky cause, at each time the data from one cell is accessible. Therefore, we need to find a new approach to calculate the value of the other neighboring points to calculate our interpolant.

In 2D and 3D, the same approach is used to calculate the neighboring points. After having the neighboring points, all that is left to do for us is to apply the interpolant using the coefficients and initialize the new vertex.

\[
\begin{align*}
\frac{1}{3^4} \begin{pmatrix} 0 & 0 & 0 & 0 \\ -5 & 60 & 30 & -4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} & \quad \frac{1}{3^4} \begin{pmatrix} 0 & 0 & 0 & 0 \\ -4 & 30 & 60 & -5 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \\
\frac{1}{3^4} \begin{pmatrix} 0 & -5 & 0 & 0 \\ 0 & 60 & 0 & 0 \\ 0 & 30 & 0 & 0 \\ 0 & -4 & 0 & 0 \end{pmatrix} & \quad \frac{1}{3^4} \begin{pmatrix} 0 & -4 & 0 & 0 \\ 0 & 30 & 0 & 0 \\ 0 & 60 & 0 & 0 \\ 0 & -5 & 0 & 0 \end{pmatrix}
\end{align*}
\]

\[(7.16)\]

\[(7.17)\]

\[
u_{i-1} = 2u_i - 2S_{i,x} - u_{i+1}
\]

\[(7.18)\]
However as it can be seen in Figure 7.5, we need the vertices number 0, 3, 12, 15 to do the interpolation. But since we do not have enough equations with respect to unknowns, we can not reconstruct the value for these vertices from the linear surpluses stored within our cell. Therefore, these values have been set to zero for the interpolation and as Figure 7.5 shows they do not contribute to the new vertex. As a result of this the stencils mentioned previously in Equations (7.14) and (7.15) will change as shown in Equations (7.19) and (7.20). By doing this, we need to make sure we do not introduce high order errors. By a closer look at equations (7.14) and (7.15) we see that, the weights for these vertices are rather smaller than other vertices. As a result we can be sure that setting these contributions to zero does not introduce high order errors to our initialization.
Another restriction we will have for this interpolation is when the parent vertex is already a boundary vertex. As in this case there is no linear surplus stored for that vertex, we can not reconstruct the neighboring vertex, and therefore we will have to set this value to be zero either.

Figure 7.6 shows the vertices which have to be set to zero in the cubic interpolation. As the vertices 5 and 9 are boundary vertices, no calculation happens for them and therefore there is no linear surplus available, thus we can not retrieve the values for vertices 1, 4, 8, 13 and have to set these to zero. Equation (7.21) shows the corresponding stencil for the scenario shown in Figure 7.6. Doing this we will lose 2 order of accuracy in the interpolation which seems to bring our convergence rate down, but on the other hand we know that the data travels from the boundary to the domain one cell per iteration and since we are close to the boundary the newly created vertex will be corrected immediately in the next iteration. Therefore we will not lose much by
setting these vertices to zero in the cubic interpolation stencils.

\[
\frac{1}{3^8 + 909} \begin{pmatrix} 0 & 0 & -120 & 0 \\ 0 & 1800 & 900 & -120 \\ 0 & 3600 & 1800 & -240 \\ 0 & 0 & -150 & 0 \end{pmatrix}
\]  

(7.21)
8 Numerical Experiments

In this chapter, I bring some numerical experiments that study the adaptivity and coarsening criteria yielding to an adaptively refined grid and compare them to a uniformly refined grid with the same depth of refinement. Four different problems have been chosen as benchmarks.

This experiments have been done on nodes with AMD Opteron 850, 2.4 GHz, processor with 1 MB of cache and 8 GB of Memory.

8.1 Experiment One

The first experiment is

\[-\Delta u = 0.\]

With the right hand side being zero, this experiment seems to be the same as the last one. But with the right hand side being zero we would expect no refinement to happen (chapter four). As a result all of the experiments regardless of the refinement threshold will lead to the same grid. Figure 8.2 shows a seven level computational grid for this example. The seven level grid is not produced due to refinement, but the initial mesh size has been changed. As expected no refinement happens for this experiment and the mesh size remains unchanged.
8.1. Experiment One

**Figure 8.1**: Initial Grid for Experiment One

**Figure 8.2**: Seven level Grid
8.2 Experiment two

The second Experiment is

\[-\Delta u = 1.\]

The initial grid is shown in Figure 8.3.

![Initial Grid for Experiment Two](image)

Figure 8.3: Initial Grid for Experiment Two

As the right hand side of this equation is constant we would expect to have a homogeneous refinement in every part of the domain. Figure 8.4 shows the resulting grid after refinement. Due to the right hand side and domain shape of this problem the regular and the adaptive grids will look the same and thus the run-times will be the same.

8.3 Experiment Three

This example has been taken from the dissertation of Prof. U. Rüde.

\[-\Delta u(x, y) = g(x, y) \quad \text{with} \quad g(x, y) = \frac{\cos(2\pi(\frac{1}{2}(x + 1) - \frac{1}{2}(y + 1))) \sinh(2\pi(\frac{1}{2}(x + 1) + \frac{1}{2}(y + 1) + 2))}{\sinh(8\pi)}.\]
This experiment has been solved on a unit square with boundaries set to be zero. The initial grid has is shown in Figure 8.5.

Figure 8.6 shows this problem solved on a five level adaptive grid. the runtime for this example is 1,79 second. the same problem on a five level regular grid needs 9,63 seconds.

The runtime for this experiment on a six level adaptive grid is 8,03 seconds while it lasts on a 6 level regular grid is 43,4 seconds. Figure 8.7 shows the resulting adaptive
8.3. Experiment Three

grid.

Figure 8.6: Five level grid for experiment Three

Figure 8.7: Six level grid for experiment Three
### 8.4 Experiment Four

Here, we pick up the experiment from refinement strategies chapter:

![Initial Grid for Experiment Four](image)

**Figure 8.8:** Initial Grid for Experiment Four

\[
- \Delta u = c \left( 128\pi r^2 \sinh(64\pi(2-r^2)) - 3 \cosh(64\pi(2-r^2)) \right)
\]

with \( r^2 = \sum_i = 1^d \left( x_i - \frac{1}{3} \right)^2 \), \( c = -\frac{128\pi}{\sinh(128\pi)} \).

The boundary conditions are Dirichlet and are set to be zero. Figure 8.8 shows the approximated solution to this problem on a regular grid.

As it can be seen from Figure 8.8, the regions on the bottom left of the domain are the regions which we would expect to have some refinement where the top right region might need coarsening.
To start I will switch the refinement switch on to just have this grid refined. The strategy used for this experiment is the refinement threshold and the threshold has been set to 1.0e – 3, Figure 8.9 shows the resulting adaptive grid. This grid has a depth of seven in its space-tree. The run-time for this experiment was, 15, 55 seconds.

![Figure 8.9: Seven level Adaptive grid](image)

To compare this result we need to solve the same problem with a depth seven regular grid and compare the run-times. As expected the run-time for the regular grid is higher than the adaptive grid and is 260, 91 seconds. The speed up for this experiment using the adaptive grid is 16, 8. As it is seen in this Figure the grid is very fine that the domain looks almost like a continuous computational domain.

The same experiment has been done with refinement threshold of 1.0e – 04 to get a depth of 8, Figure 8.10 shows the resulting adaptive grid. The run-time for the adaptive grid is 96, 89 seconds while the run-time for the regular grid is 1984, 24 seconds which leads to a speed up of 20, 5.

The run-time for the six level adaptive grid is 1, 35 seconds where the runtime for the six level regular grid is 34 seconds, which corresponds to a speed up of 3, 05. Figure 8.11 shows the six level regular and adaptive grids.

The next table shows some run-time comparisons for this experiment for the regular and the adaptive grid. Using different threshold we might have the same depth but different number of vertices, and due to this we might have more than one result for each depth.
8.4. Experiment Four

Figure 8.10: Eight level Adaptive grid

Figure 8.11: Six Level Regular and Adaptive grid

<table>
<thead>
<tr>
<th>Grid Depth</th>
<th>Regular grid run-time</th>
<th>Adaptive grid run-time</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>34,64</td>
<td>11,35</td>
<td>3,05</td>
</tr>
<tr>
<td>7</td>
<td>260,91</td>
<td>63,34</td>
<td>4,12</td>
</tr>
<tr>
<td>7</td>
<td>260,91</td>
<td>29,89</td>
<td>8,73</td>
</tr>
<tr>
<td>7</td>
<td>260,91</td>
<td>15,55</td>
<td>16,8</td>
</tr>
<tr>
<td>8</td>
<td>1984,24</td>
<td>136,21</td>
<td>14,57</td>
</tr>
<tr>
<td>8</td>
<td>1984,24</td>
<td>96,89</td>
<td>20,5</td>
</tr>
</tbody>
</table>
8.5 Computational Fluid Dynamics experiments

In this section I bring some of the experiments which were done within the Peano framework using the refinement black-box for computational fluid dynamics.\cite{ref5}

In this experiment an obstacle within a flow channel has been studied. The initial grid for this setup can be seen in Figure 8.12.

![Initial Grid for the CFD Simulation](image1)

Figure 8.12: Initial Grid for the CFD Simulation

As this experiment has been done out of the perspective of this thesis and only the refinement black-box implemented within the perspective of this thesis has been used, I only bring the resulting grids using the refinement black-box. The resulting grids are shown in Figure 8.13. For this experiment the grid has been refined as time evolved. These images show the grid at the 10100th and 10200th time step.

![Generated Grid for the CFD Simulation](image2)

Figure 8.13: Generated Grid for the CFD Simulation
9 Conclusion

This thesis has implemented refinement and coarsening strategies within the Peano framework. Using linear surplus as the estimate for the refinement and coarsening criteria we were able to implement a PDE independent refinement/coarsening black-box. Refinement and coarsening can be switched on and off by the user and the thresholds for those can also be defined by the user at the runtime. Two different schemes for interpolation have been implemented to initialize the newly introduced vertices. With the high order interpolation scheme one can guarantee to avoid possible high frequency errors introduced while initializing the vertices.

These features have already been applied to computational fluid dynamics simulations within the Peano framework[5]. As the Peano framework provides parallelisation, these features can also be run in parallel to decrease the runtime and distribute the work and memory load among different nodes[3].
Bibliography


