

UNIVERSITY OF CALIFORNIA  
Santa Barbara

# A Parallel Preconditioner for Octree Meshes

A Dissertation submitted in partial satisfaction  
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Vikram Aggarwal

Committee in Charge:

Professor John R. Gilbert, Chair

Professor Linda R. Petzold

Professor Frédéric G. Gibou

June 2008

The Dissertation of  
Vikram Aggarwal is approved:

---

Professor Linda R. Petzold

---

Professor Frédéric G. Gibou

---

Professor John R. Gilbert, Committee Chairperson

May 2008

A Parallel Preconditioner for Octree Meshes

Copyright © 2008

by

Vikram Aggarwal

To Neha.

## Acknowledgements

Research work entails a collaboration with many individuals and an exchange of ideas with a wide variety of people. This thesis is no exception. John Gilbert has been a wonderful advisor, keeping me on track, and helping me focus on the important questions. I also thank Joseph Papac for his timely help with fluid flow computation.

Aydin Buluç and Viral Shah provided many stimulating discussions. Viral Shah had valuable insight into matrix computations whenever I needed it. Min Roh and Viral Shah were particularly supportive when minor pieces of hell broke loose. Many other students at UCSB provided input, insight and intellectual company.

I thank my parents and my brother for their love and assistance throughout my PhD.

Half the effort in this work is due to my lovely wife: Neha Pandey. She gave me reasons to smile through the toughest phases and was a loving companion in times of need.

# Curriculum Vitæ

Vikram Aggarwal

## Education

- |      |  |
|------|--|
| 2004 | Master of Science in Computer Science, Florida State University,<br>Tallahassee, FL. |
| 2002 | Master of Science in Mathematics, Indian Institute of Technology,<br>Bombay, India.  |

## Experience

- |             |  |
|-------------|--|
| 2006 – 2008 | Graduate Research Assistant, University of California, Santa Bar-<br>bara. |
| 2004 – 2006 | Teaching Assistant, University of California, Santa Barbara.               |

## Selected Publications

J. R. Gilbert, V. B. Shah, I. Patel, V. Aggarwal. Measuring Productivity in a Graduate Parallel Computing Course *Siam Conference on Parallel Processing for Scientific Computing*, March 2008.

# Abstract

## A Parallel Preconditioner for Octree Meshes

Vikram Aggarwal

Fully adaptive octree grids are a promising development in the solution of fluid flow problems. While fully adaptive grids make it easier to discretize the domain, solving the linear system is harder, due to the linear system being unsymmetric. Researchers would find it beneficial to solve large systems of equations with such grids, to simulate phenomena that occur at very fine grid spacing.

In this work, we develop an octree grid generator which is meant to be a general purpose prototyping framework for researchers interested in octree grids. Using the grid generator, we analyze the structure of fully-adaptive octree grids to identify their salient properties. We evaluate various methods of solving linear systems arising from these grids and find iterative linear solvers like BiCGSTAB to be the most promising. We demonstrate that the performance of iterative solvers is improved considerably by the use of no-fill incomplete factors.

We demonstrate that for large octree grids, the preconditioner forms the dominant computation that limits the performance of iterative linear systems on parallel machines. To attempt to speed up the performance of the preconditioner, we exploit the graph structure of the octree grids to develop a novel coloring. This

coloring leads to an upper bound on the chromatic number of octree grids for the Poisson problem. Using this coloring, we develop a triangular solver that shows good performance on parallel computers. We demonstrate the performance of the triangular solver as a preconditioner for the BiCGSTAB iteration, which greatly improves the performance of the linear system for these grids.



# Contents

<b>Acknowledgements</b>	<b>v</b>
<b>Curriculum Vitæ</b>	<b>vi</b>
<b>Abstract</b>	<b>vii</b>
<b>List of Figures</b>	<b>xii</b>
<b>List of Tables</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Poisson Equation . . . . .	2
1.1.1 Relevance to Fluid Flow Simulations . . . . .	4
1.1.2 Poisson Equation for Level Set Methods . . . . .	5
1.2 Octree Discretization . . . . .	6
1.3 Finite Difference Approximation . . . . .	9
1.4 Problem Statement . . . . .	13
<b>2 Matrix Structure and Serial Solution</b>	<b>16</b>
2.1 Graph Structure . . . . .	20
2.1.1 Symmetry . . . . .	21
2.1.2 Condition Number . . . . .	29
2.2 GMRES and BiCGSTAB . . . . .	30
2.3 Preconditioners . . . . .	32
<b>3 Graph Coloring and Parallel Approaches</b>	<b>39</b>
3.1 Triangular solves dominate performance . . . . .	42
3.2 Matrix Reordering . . . . .	44

3.3	Graph Coloring . . . . .	52
3.4	Parallel Triangular Solves . . . . .	59
3.4.1	Partitioned Inverses . . . . .	60
3.4.2	Graph Coloring . . . . .	61
3.4.3	Oski . . . . .	65
3.4.4	Memory Bandwidth . . . . .	66
3.5	Performance results . . . . .	68
3.5.1	Scaling Observations . . . . .	72
<b>4</b>	<b>Alternate Approaches</b>	<b>75</b>
4.1	Graph Methods . . . . .	76
4.1.1	Multigrid . . . . .	76
4.1.2	Support Graph Preconditioning . . . . .	84
4.2	Numerical Methods . . . . .	90
<b>5</b>	<b>Grid Generator</b>	<b>95</b>
5.1	Overview . . . . .	96
5.2	Stand-alone program . . . . .	102
5.3	Extensibility . . . . .	103
5.4	Convergence . . . . .	106
5.5	Modifications for Coloring . . . . .	109
5.6	Focus on Experimentation . . . . .	111
<b>6</b>	<b>Conclusion</b>	<b>112</b>
	<b>Bibliography</b>	<b>114</b>
	<b>Appendices</b>	<b>124</b>
<b>A</b>	<b>GeomOct Reference</b>	<b>125</b>
A.1	Installation . . . . .	125
A.2	Detailed Reference . . . . .	126
A.2.1	ColorNumberer< Depth > Class Template Reference . . .	126
A.2.2	ColorRed< Data, Depth > Class Template Reference . . .	130
A.2.3	Corner< Data, Depth > Class Template Reference . . . .	131
A.2.4	CornerKeeper< Data, Depth > Class Template Reference	137
A.2.5	CornerList< Data, Depth > Class Template Reference . .	142
A.2.6	CornerOperation< Data, Depth > Class Template Reference	146
A.2.7	CornerPosition< Depth > Class Template Reference . . .	148

A.2.8	ForcePrinter< Depth > Class Template Reference . . . . .	154
A.2.9	FredMatrixPrinter< Depth > Class Template Reference . . . . .	156
A.2.10	NodeInterpolant Class Reference . . . . .	160
A.2.11	NodePosition< Depth > Class Template Reference . . . . .	163
A.2.12	Octree< Data, Depth > Class Template Reference . . . . .	166
A.2.13	OctreeNode< Data, Depth > Class Template Reference . . . . .	168
A.2.14	OctreeNodeOperation< Data, Depth > Class Template Reference . . . . .	175
A.2.15	SetPoisson< Data, Depth > Class Template Reference . . . . .	177
A.2.16	SplitNotNull< Data, Depth > Class Template Reference . . . . .	179
A.2.17	SplitShape< Data, Depth > Class Template Reference . . . . .	181
A.2.18	SplittingCriterion< Data, Depth > Class Template Reference . . . . .	184
A.2.19	TrueSolutionPrinter< Depth > Class Template Reference . . . . .	185

# List of Figures

2.1	Quadtree grid demonstrating discretization . . . . .	26
2.2	The source of unsymmetry . . . . .	29
3.1	Percentage of time spent in the preconditioner in serial . . . . .	43
3.2	Percentage of time spent in the preconditioner in parallel . . . . .	44
3.3	Percentage of time spent in the preconditioner for 4M grid . . . . .	45
3.4	Percentage of time spent in the preconditioner for 16M grid . . . . .	45
3.5	Single quadtree showing the different heights of $\hat{C}(v_i)$ . . . . .	49
3.6	Quadtree numbered according to the heights of grid points . . . . .	49
3.7	Spy plot of the simple quadtree grid renumbered . . . . .	50
3.8	Spy plot of matrix reordered with the colored ordering . . . . .	59
3.9	Performance of linear solver with circular grid containing 4M (top), 16M (middle) and 64M (bottom) grid points . . . . .	70
3.10	Performance of linear solver with circular grid containing 163M grid points . . . . .	71
3.11	Scaling of the coloring-based linear system solver with circular grid . . . . .	74
5.1	Single octree showing nodes and corners . . . . .	98
5.2	Octree grid refined around origin . . . . .	105
5.3	One half of a spherical octree grid . . . . .	105

# List of Tables

1.1	The Poisson solver dominates the Navier-Stokes computation . . .	5
2.1	Growth of the number of grid points for spherical grid . . . . .	18
2.2	Growth of the number of grid points for other grids . . . . .	18
2.3	Measures of symmetry of the spherical grid . . . . .	22
2.4	Structural symmetry for other grids . . . . .	22
2.5	Numerical symmetry for other grids . . . . .	22
2.6	Measure of symmetry of the spherical grid . . . . .	25
2.7	$\ A_K\ $ as a fraction of $\ A\ $ for the other grids . . . . .	26
2.8	Condition number estimates for the spherical grid . . . . .	30
2.9	Condition number estimates for other grids . . . . .	30
2.10	Comparison of GMRES and BiCGSTAB with ILU(0) preconditioner	32
2.11	Convergence of ILUT, with drop tolerance = $10^{-3}$ . . . . .	36
2.12	Convergence of ILUT, with drop tolerance = $10^{-4}$ . . . . .	36
2.13	Convergence of ILUT, with drop tolerance = $10^{-5}$ . . . . .	36
2.14	Iterations of BiCGSTAB for matrix generated with various symmetric permutations, preconditioned with ILU(0) . . . . .	38
2.15	Iterations of GMRES for matrix generated with various symmetric permutations, preconditioned with ILU(0) . . . . .	38
3.1	Growth of the number of colors for spherical grid . . . . .	55
3.2	Growth of the number of colors for grid of hyperboloid of 1 sheet	56
3.3	Growth of the number of colors for grid of hyperboloid of 2 sheets	56
3.4	Growth of the number of colors for cylindrical grid . . . . .	56
3.5	Number of grid points of each color for the spherical grid with 65M points . . . . .	58
3.6	Triangular solve timings with Oski on 65M grid . . . . .	66
3.7	Triangular solves with Oski on 16M grid . . . . .	66

3.8	Time in seconds for solving a linear system with the 4M mesh . .	71
3.9	Time in seconds for solving a linear system with the 16M mesh . .	71
3.10	Time in seconds for solving a linear system with the 64M mesh . .	71
3.11	Time in seconds for solving a linear system with the 263M mesh .	72
4.1	Discrete $L^2$ norm error for multigrid v-cycle with 63k spherical grid using the identity matrix as an operator . . . . .	79
4.2	Discrete $L^2$ norm error for multigrid v-cycle with 256k spherical grid using the identity matrix as an operator . . . . .	80
4.3	Discrete $L^2$ norm error for multigrid v-cycle with finest grid containing 256k spherical grid and coarsest grid containing 16k points using interpolation operators and full weighting . . . . .	81
4.4	Discrete $L^2$ norm error for multigrid v-cycle with finest grid containing 256k spherical grid and coarsest grid containing 469 points using interpolation operators and full weighting . . . . .	82
4.5	Discrete $L^2$ norm error for multigrid v-cycle with 256k spherical grid using interpolation operators, full weighting and Galerkin coarse grid approximation . . . . .	82
4.6	Relative difference between eigenvalues of the symmetric matrices	92
4.7	Condition numbers of $S_i^{-1}A$ . . . . .	92
4.8	Iterations of GMRES with $S_i$ used to generate IC(0), and ILU(0) factors . . . . .	92
5.1	Convergence of focussed grid with $u = e^{-(x+y+z)}$ . . . . .	108
5.2	Convergence of spherical grid with $u = e^{-(x+y+z)}$ . . . . .	108
5.3	Convergence of focussed grid with $u = e^{-(x^2+y^2+z^2)}$ . . . . .	108
5.4	Convergence of spherical grid with $u = e^{-(x^2+y^2+z^2)}$ . . . . .	109

# Chapter 1

## Introduction

*To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty, of nature ... If you want to learn about nature, to appreciate nature, it is necessary to understand the language that she speaks in.*

Richard Feynman (1918-1988) [32]

The mathematical problem of interest in this research is the variable coefficient Poisson problem. This problem is the computational kernel of many important applications like incompressible flows, dendritic growth, electrostatics, and many fields in mechanical engineering [40].

In the general formulation, no analytical solution exists, and thus the problem must be solved using numerical methods. There is a lot of previous work in this area. Many previous studies [41, 42, 59, 71, 73] have investigated efficient solvers of the Poisson equation in various computational domains.

Our interest in this problem is motivated by the work of Gibou et al. [42, 75, 76] in which a second order accurate projection method for the incompressible

## *Chapter 1. Introduction*

Navier-Stokes equation is described. Our main focus is to improve the parallel performance of a linear solver for such a method. We focus on the linear systems obtained from these Poisson equations, and look at various ways to improve their running time, and the prospect of running these linear system solvers in parallel. We identify the hurdles in a straightforward implementation of a parallel solver for this method, and then use the graph structure of these matrices to obtain an efficient parallel solver that solves the problem on distributed architectures. This shall lead to an advance in both the size of problem that can be modeled, and the accuracy to which we can model these problems.

In order to motivate the topic, we state the problem formally, and then cover the general method of solving these systems using finite differences and octree discretizations.

### **1.1 Poisson Equation**

Consider a three dimensional domain, and define  $f$ , a force function,  $v$ , the velocity, and a function  $\rho$  that varies over the domain. In this setting, the variable coefficient Poisson problem can be formally written as follows:  $\nabla(\rho\nabla v) = f$

This equation is found in a fluid flow where  $f$  and  $\rho$  are functions that describe the vector fields of force on the computational domain, and  $v$  is the unknown value



## *Chapter 1. Introduction*

to be calculated [69]. Domains of interest are often a two dimensional area, or a three dimensional volume, over which the values  $f$  and  $\rho$  can be computed. The general method of solving a Poisson equation proceeds as follows. First the domain of interest is discretized. This involves dividing the physical domain into computational elements at which the computation proceeds. The discretization is called a grid, and the computational elements are called grid points. Once the domain has been discretized, a variety of numerical methods can be used to yield an approximate solution. In our work, we focus on finite differences; other possibilities include finite elements and finite volumes. These methods lead to a mathematical relationship between the grid points. These relationships are usually between adjacent grid points and relate the values of  $f$ ,  $v$ , and  $\rho$  at various grid points. The unknown values in this relationship are  $v$ , the velocities at the specific grid points, and the resulting linear system takes the form  $A\bar{v} = \bar{f}$ , where  $\bar{v}$  is the vector of velocities at the grid points, and  $\bar{f}$  is the vector of the force function, evaluated at the grid points. The vectors  $\bar{v}$  and  $\bar{f}$  are the discretized analogues of the continuous functions  $v$  and  $f$ , and  $A$  is a square matrix. Solving the Poisson equation is thus reduced to solving a linear system. The work required to solve the linear system depends largely on the properties of the matrix  $A$ . Matrices obtained from finite difference methods are usually sparse, since the physical values at most grid points depend on a small number of other grid points.

### 1.1.1 Relevance to Fluid Flow Simulations

Table 1.1 demonstrates the time required for fluid flow simulations, and provides justification for the interest in the Poisson problem. These numbers are from a Navier-Stokes fluid flow calculation based on a level set method run on a two dimensional grid. This work is done by Joseph Papac, a researcher working on immiscible multiphase flow in porous media [83]. The work constitutes a study of fluids through porous media, where the interface is tracked using a level set method, and the region is discretized using a Cartesian grid. In this example, Papac reports the amount of time spent for a single time step in the simulation. The times in this were averaged over five time steps of the Navier-Stokes solver, and only the average time per time-step is reported.

As we can see in table 1.1, 42.63 seconds are spent on the entire computation that updates the interface value at a single time step for a grid containing 262k points. Of that, 27.81 seconds are spent by the Poisson solver. As the grid size increases to 1,048k grid points, 260.98 seconds are spent in the Poisson solver out of 353.74 seconds. Thus, the Poisson solver takes at least 73.77% of the total time, at  $10^{-10}$  accuracy for the solver. These results provide a quantitative example of a phenomenon that is well known in the fluid dynamics community, that the Poisson solver is the dominant calculation in most fluid flow simulations.

Grid Size	Total time	Poisson solve	Poisson contribution
16,384	7.31	4.39	60.05%
262,144	42.63	27.81	65.22%
1,048,576	353.74	260.98	73.77%

**Table 1.1:** The Poisson solver dominates the Navier-Stokes computation

Since the Poisson problem can be discretized in many ways, and a variety of numerical methods can be used to solve it, we motivate our particular choice in the sections that follow.

### 1.1.2 Poisson Equation for Level Set Methods

Our interest in the Poisson equation is motivated by the work on solving the Poisson equation for octree grids of level set methods due to Min et al. [76]. Level set methods were pioneered by Osher and Sethian [91], and are used to solve moving boundary problems. In the level set method, the boundary to be tracked is evolved into a higher dimension set given by the level set function  $\varphi_t$ . The value of the function  $\varphi_t$  at  $t = 0$  gives the position of the boundary. Instead of tracking the boundary, the level set function  $\varphi_t$  is evolved instead. Osher and Fedkiw [81] and Sethian [95] provide a good overview of the level set method.

In the work by Losasso et al. [69], the level set function  $\varphi$  is updated using the velocity field  $u$ . The velocity field itself is computed using the Navier-Stokes

## Chapter 1. Introduction

equation given by equation 1.1. In the equation,  $u$  is the velocity field,  $f$  is the external force function, and  $p$  is the pressure.

$$\begin{aligned} u_t + u \cdot \nabla u &= -\nabla p + f \\ \nabla \cdot u &= 0 \end{aligned} \tag{1.1}$$

To solve this equation, first an intermediate velocity field  $\hat{u}$  is computed, ignoring the pressure term  $\nabla p$ . Then, we compute the pressure update for the time increment  $\Delta_t$  as the solution to the Poisson equation given below.

$$\nabla^2 p = \frac{\nabla \cdot \hat{u}}{\Delta_t}$$

Finally, the velocity field is computed using the pressure computed above.

$$u = \hat{u} - \Delta_t \nabla p$$

Since the level set is to be updated, the grid points for this discretization are ideally located along the interface. This is exactly what octree discretizations provide, and we will continue this theme in the next section.

## 1.2 Octree Discretization

The number and position of the grid points depends on the desired amount of accuracy. The time required for the computation is directly proportional to the number of these grid points, as an increase in grid points corresponds to more

## *Chapter 1. Introduction*

computational work. For this reason, a smaller number of grid points leads to a quicker solution. On the other hand, an increase in accuracy usually requires an increase in number of these grid points. Thus, there is a tradeoff in the number of grid points we would like to place in the domain. Since the computational cost relies on the number of grid points, this number is usually abbreviated by  $n$ , a convention we shall follow throughout in this work. This is also called the “size” of the problem. Later, when this discretization leads to a linear system, the matrix obtained is square and has dimension  $n$ , leading to a direct relationship between the size of the physical problem and the size of the linear system that solves it.

A common choice for placing the grid points is coordinate grids, in which elements are placed along regular, coordinate grid patterns. This has the advantage of simplifying the discretization, and grid points can be easily referenced by their coordinate position. Such a scheme might be too simplistic in applications where a lot of accuracy is desired in a small part of the domain, while a loss of accuracy in the remaining domain is permissible. Since a decrease in the number of grid points leads to a quicker solution, the accuracy of the method would improve if fewer grid points are assigned to the regions where the required accuracy is low, and more grid points are assigned to regions where the accuracy required is higher. Sometimes, the underlying physical process governs this assignment of grid points, where one part of the region might not exhibit interesting phenomenon,

## *Chapter 1. Introduction*

while another part might have a lot of intricate physical interactions which need accurate modelling. Consider the evaporation of a fluid from a beaker, where the interesting phenomenon occurs near the surface, and the fluid far away from the surface does not offer any interesting physical insight. In such cases, octree grids have demonstrated promise. Recent work [69, 87] suggests that such grids can be easily generated, and offer far superior convergence compared to coordinate grids containing a comparable number of grid points. The grids that we generate are octree grids, with automatic mesh refinement using non-graded grids [76]. No constraint is imposed on the size of adjacent cells, and the mesh generation is dictated entirely by the interface of interest. In an octree discretization, the 3D domain is usually a cuboid. This is divided into 8 cuboids by bisecting each coordinate axis. If any cuboid is to be refined, then it is again divided into 8 identical cuboids by coordinate bisection. The cuboids are called subvolumes, or octree nodes, since they are arranged in an octree structure. Each cuboid is a node on an octree, and on refinement has 8 children.

There is a great deal of interest in the numerical solution of the Poisson equation. The seminal paper in this research is by Chorin [19], which describes an early attempt at solving this problem using coordinate grids. Quadtree grids in two dimensions and octree grids in three dimensions are a result of recursive spatial splitting, and this was first developed by Shephard et al. [96, 112, 113].

## *Chapter 1. Introduction*

Future work extended this technique for other problem areas [84, 97]. Octree and quadtree grids are gaining importance, and recent work covers a lot of development relating to the use of octree and quadtree grids for fluid flows [69, 87]. In addition, there are many options beyond coordinate grids and octree or quadtree grids. Various types of triangulations could be used in the case of finite volume or finite element computations. Grid generation is a vast field, and a variety of concerns influence the choice of grid used. Since our work focusses on computational aspects, we refer the reader to a comprehensive survey of grid generation techniques by Owen [82].

The choice of discretization and the choice of the numerical method often influence each other. We list the possible numerical methods in the next section, with a motivation for our particular choice of method.

### **1.3 Finite Difference Approximation**

There are a variety of numerical methods that can be used to solve the Poisson problem. Finite elements are a popular method for some problems, especially in two dimensions. Other possibilities are finite differences and finite volumes. Finite elements have an attractive theoretical framework, and the matrices obtained from this method are often symmetric and positive definite. This alone

## *Chapter 1. Introduction*

is a huge advantage, since the conjugate gradient method [52] is a robust solver for such matrices. As we mention in section 1.1.2, we are motivated by octree discretizations of level set methods. In level set methods, the region of interest is a boundary or an interface between two fluids. The discretization aims to surround the interface by a large number of grid points, while keeping the remaining region relatively free of grid points. Then, the level set interface  $\varphi_t$  is evolved using the vector field obtained at the grid points. Since the level set interface  $\varphi_t$  defines the boundary at  $t = 0$ , the regions farther away from the boundary do not exhibit interesting physical phenomena. For this reason, Min et al. [76] choose to use fully adaptive grids to allocate grid points. This technique ensures that the highest concentration of grid points stays near the interface, a set of measure zero in the computational domain. In such problems, where the region of interest forms a very small part of the computational domain, the finite element method is challenging to implement. If the finite element method is chosen, constraints on the size of elements can help implement the method, though at a considerable loss of accuracy. This is due to the introduction of a large number of additional elements to maintain the size constraint between neighboring elements. These additional elements reduce the overall accuracy of the method.

Multigrid techniques have been used for fluid flow equations with quite some success. The interest in multigrid techniques started with the work of Brandt



## *Chapter 1. Introduction*

[11], which demonstrated the use of multiple grids to speed convergence. The first known use of multigrid techniques for solving fluid flow equations is by Ni [77]. In this work, multiple grids were used to accelerate a one-step Lax-Wendroff algorithm for steady state inviscid problems. This work was extended to the Navier-Stokes equation by Johnson [62]. The work of Koeck and Chattot demonstrated the use of multiple grids to speed the solution of inviscid three-dimensional flows [66]. The first application of multigrid methods to three dimensional viscous flows is due to Johnson and Swisshelm [61]. Finally, Jespersen demonstrated the use of multiple grids to accelerate problems with time-dependent solution [58]. The first demonstrated use of multigrid schemes for the Navier-Stokes equation on parallel computers was by Johnson et al. [99]. The development of fully adaptive meshes started with an interest in embedded meshes. Bassi et al. [9] pioneered the use of embedded meshes for the two dimensional Navier-Stokes equation. Johnson and Swisshelm [60, 98] extended this work to three dimensional inviscid and viscous flows. Bai and Brandt [7] demonstrated the use of adaptive grids with multigrid. R  de [90] discussed modifications to the relaxation and the multilevel cycling to accelerate the convergence on adaptive grids. A general purpose multigrid based software for adaptive meshes called Gerris has been developed by Popinet [87]. A general overview of the multigrid method can be found in the excellent introduc-

## *Chapter 1. Introduction*

tory book by Briggs [13]. Other texts giving an overview of the multigrid method are Hackbusch and Trottenberg [48] and McCormick [72].

Multigrid methods have the advantage that their convergence is independent of the size of the matrix. This is a remarkable property and has resulted in a large body of research. Multigrid, however, suffers from a few drawbacks. The biggest drawback of multigrid is that it is hard to make a general purpose multigrid code. Multigrid codes are usually finely tuned to a specific problem, and do not work outside that domain. The prolongation and restriction operators used with one partial differential equation might not work on another, or even on the same PDE with a different discretization. Automatic mesh refinement offers some challenges as well, and Popinet uses graded grids to avoid such problems [87]. In many cases, algebraic multigrid might be used where the original grid is not available, but algebraic multigrid has many tunable parameters, and getting a working code is not always easy. Moreover, algebraic multigrid requires a symmetric and positive definite matrix, and this limits their utility for some problems. In the matrices obtained by Min et al. [76], the matrices are not symmetric, and thus algebraic multigrid cannot be directly applied. In chapter 4 we present a preliminary investigation of geometric multigrid for the Poisson problem.

The physical problems that we focus on are large fluid flow problems over 3D domains. The equations governing these fluid flows are the Navier-Stokes

## *Chapter 1. Introduction*

equations, with a Poisson equation being solved at every iteration. In recent work [69], the level set method [95] and adaptive grids are used to solve this system to first order accuracy. In this case, the matrix is symmetric, and the authors use the conjugate gradient method to solve the linear system. An improvement to this work discretizes the Navier-Stokes equation on completely adaptive grids, and obtains second order accuracy [75]. In the newer work, however, the higher order accuracy makes the matrix non-symmetric and thus conjugate gradients cannot be used anymore. This second-order technique is the main focus of this paper. The matrices of interest are generated using the technique by Min et al. [75], for 3D domains. The matrix sizes can get quite large, as increasing the size allows researchers either to model larger computational domains, or to refine the existing domain to a much finer detail.

### **1.4 Problem Statement**

In our work, we seek to increase the size of the problems that can be modeled by researchers using finite difference methods with octree discretizations to solve the Poisson equation. We focus on the computationally intensive part of the problem: the linear system solve that occurs after the Poisson equation has been discretized using octree grids. This linear system is of the form  $Ax = b$ , where the

## *Chapter 1. Introduction*

vectors  $x$  and  $b$  are of length  $n$  and the matrix  $A$  is a square matrix of dimension  $n$ . Greater refinement in the grid and a larger grid both result in an increase in  $n$ . Further, with an increase in the size of the problem, the linear system can no longer fit in the memory of a single computer. For these reasons, it is beneficial to find a parallel algorithm to perform the linear solve, so that larger systems can fit on current hardware, and can be solved efficiently.

Since physical systems exhibit interesting phenomena at very high resolutions, or on longer time scales, it is also important to look for parallel algorithms with a view towards performance. Multiprocessor computers are increasingly becoming the norm, and it is beneficial to speed these calculations on commodity hardware as well. So, while we aim to solve larger systems, we are also interested in solving existing problems faster, especially on multicore or multiprocessor architectures.

We believe the novel contribution of this work to be the following.

- Identifying and eliminating the computational hurdle in the efficient parallel solution of linear systems using octree grids of level set methods.
- Demonstrating that gain in efficiency makes it possible to solve larger linear systems than were previously possible.

Having motivated the physical problem, and the various components that form the solution, the layout of the remaining thesis will be as follows. In chapter [2](#),

## *Chapter 1. Introduction*

we show the structure of the matrices obtained. We list the previous work in solving such linear systems, and identify the computationally intensive part of the calculation. Later, in chapter 3, we list our graph coloring algorithm, and show how we use this approach to successfully solve these systems on parallel computers. In the same chapter, section 3.5 lists our performance results. We briefly review other approaches to solve such systems in chapter 4. In chapter 5, we describe our octree grid generator. The grid generator can be used to generate octree grids for arbitrary domains, and we list some of its salient points. Finally, we close with the conclusions of our work in chapter 6.

## Chapter 2

# Matrix Structure and Serial Solution

*You can know the name of a bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird... So let's look at the bird and see what it's doing — that's what counts.... I learned very early the difference between knowing the name of something and knowing something.*

Richard Feynman (1918-1988) [33]

In order to develop a parallel solution for the linear system, it is important to look at the properties of the matrices. We illustrate the matrix properties by analyzing a few different octree grids. The domain in all these problems is the cubic volume  $V = \{(x, y, z) \mid 0 \leq x, y, z \leq 4\}$ . Since our motivation is from level set methods to track interfaces, we generate octree grids to track specific interfaces. The interface in all these examples is the zero set of the function  $\Gamma(x, y, z)$ .

- A sphere,  $\Gamma(x, y, z) = (x - 2)^2 + (y - 2)^2 + (z - 2)^2 - 1$ .

## Chapter 2. Matrix Structure and Serial Solution

- A cylinder,  $\Gamma(x, y, z) = (x - 2)^2 + (y - 2)^2 - 1$ .
- A hyperboloid of one sheet,  $\Gamma(x, y, z) = (x - 2)^2 - (y - 2)^2 - (z - 2)^2$ . This is referred to as hyperboloid (1) in tables.
- A hyperboloid of two sheets,  $\Gamma(x, y, z) = (x - 2)^2 - (y - 2)^2 - (z - 2)^2 - 1$ . This is referred to as hyperboloid (2) in tables.

The grid structure is independent of the specific Poisson problem being solved. The matrix values, however, do depend on the Poisson problem. In order to generate representative grids, we picked the variable coefficient Poisson problem  $\nabla(\rho \nabla u) = f$ , with the force function  $f$  and the variable coefficient function  $\rho$ . The exact solution for this problem is given by  $u = e^{-(x^2+y^2+z^2)}$ . This particular problem was used to demonstrate second order convergence in the work by Min et al. [76].

$$\begin{aligned} u &= e^{-(x^2+y^2+z^2)} \\ \rho &= \sin(x + y + z) + 2 \\ f &= -6u\rho + 4u\rho(x^2 + y^2 + z^2) - 2u(x + y + z) \cdot \cos(x + y + z) \end{aligned} \tag{2.1}$$

We generate the grids at various levels of refinement. The highest level of refinement corresponds to greater accuracy, as we can place smaller octree cells near the surface of the interface: the region of interest. Table 2.1 shows the growth

Chapter 2. Matrix Structure and Serial Solution

Level of refinement	Grid size
3	469
4	1,293
5	4,277
6	16,141
7	64,005
8	256,589
9	1,027,813
10	4,113,837
11	16,465,541
12	65,873,965

**Table 2.1:** Growth of the number of grid points for spherical grid

Level	Cylinder	Hyperboloid (1)	Hyperboloid (2)
3	605	577	511
4	2,185	2,613	2,057
5	7,893	11,257	7,723
6	29,537	46,877	30,789
7	113,773	191,409	123,607
8	446,073	773,045	494,097

**Table 2.2:** Growth of the number of grid points for other grids



## *Chapter 2. Matrix Structure and Serial Solution*

of the number of grid points for the spherical grid as the level of refinement is increased. The level of refinement corresponds to the height of the octree. At a height  $h$ , the smallest cube has a side  $\frac{L}{2^h}$ , where  $L$  is the length of the side of the original cube. For comparison, table 2.2 shows the growth in the number of grid points for the other grids. It is interesting to note that the growth in the number of grids points is roughly by a factor of four. At every level, there are roughly four times as many grid points as the level before it. This is due to the surfaces under study being two dimensional manifolds in a three dimensional volume. If every cell was being refined, the grids would grow with a factor of eight. Instead, due to the refinement near a two dimensional manifold in a three dimensional space, they grow at a factor of the two-thirds power of eight, which is four.

A short word about the terminology: in the grids we generate, the data is stored at the corners. Each of these corners corresponds to an unknown value in the  $x$  vector. Hence, the corner of the octree is a grid point. When we look at the graph properties of these matrices, the grid points correspond to vertices of the graph, and dependencies between them correspond to edges. Since the discretization is generated using an octree, the nodes of the octree shall correspond to cubic volumes inside the original domain.

In this chapter, we first highlight a few properties of the matrix to give an insight into its structure. We link these matrix properties to the properties of the

## *Chapter 2. Matrix Structure and Serial Solution*

underlying partial differential equation, or the properties of the grid structure. Then, we look at the existing method of solving the linear system. This motivates our investigation into the behavior of various preconditioners to locate a good preconditioner for these systems.

In the sections that follow, we choose to demonstrate the matrix properties with a reasonably sized matrix. In areas where we are interested in the growth of some metric, we demonstrate our point with larger matrices. In some of the sections, we rely on the graph structure of the grids to illustrate some point. Since it is easier to depict quadrees in diagrams rather than octrees, we settle for diagrams of quadrees whenever they suffice, and use octrees in all other cases.

### **2.1 Graph Structure**

The matrices that we obtain are generated by a discretization of a 3D volume. This mesh consists of grid points that are linked together by their data dependency. The grid points are modelled as nodes of the graph. When element  $A_{ij}$  is nonzero in the matrix, it is due to a data dependency between nodes  $i$  and  $j$ , which is modelled as a directed edge from node  $j$  to node  $i$ , also depicted as edge  $(j, i)$ . When both edges  $(i, j)$  and  $(j, i)$  exist, we call both the edges undirected, and denote them as a single edge without arrows. The numerical entry  $A_{ij}$  is the

## Chapter 2. Matrix Structure and Serial Solution

coefficient with which the value  $v_i$  depends on the value  $v_j$ . In some analyses, numerical values are modelled as weights on the edges of the graph, but this does not add much to our insight. Thus, when analyzing the graph structure, we concern ourselves with the edge structure alone, and not the numerical values that form the edges.

This relation between the graph and the matrix gives us valuable insight into the structure of the matrix. In the sections that follow, we shall refer to the structure of the problem using both matrix and graph terminology.

### 2.1.1 Symmetry

Symmetry is a property of the matrix. A matrix is symmetric when

$$A_{ij} = A_{ji} \qquad \forall i, j$$

This is a useful property since symmetric matrices have real eigenvalues, and a variety of robust techniques can be applied to them. For the discretization that we are motivated by [75], the matrices obtained are not symmetric. This means that the dependency between nodes  $i$  and  $j$  might not be equal. In some cases, node  $i$  might depend on node  $j$  but with a different coefficient. A lot of dependencies are directional; a dependency might be only from node  $j$  to node  $i$  with no dependency between node  $i$  and node  $j$ . While symmetry is a boolean property, we can use

Chapter 2. Matrix Structure and Serial Solution

Grid size	Structural Symmetry	Numerical Symmetry
469	79.1%	56.5%
1,293	78.5%	39.0%
4,277	78.3%	38.2%
16,141	78.4%	38.2%
64,005	78.4%	38.5%
256,589	78.4%	38.5%
1,027,813	78.5%	38.5%

**Table 2.3:** Measures of symmetry of the spherical grid

Level	Cylinder	Hyperboloid (1)	Hyperboloid (2)
3	84.4%	79.2%	76.3%
4	81.8%	78.1%	77.2%
5	80.4%	78.0%	77.3%
6	79.5%	78.2%	77.7%
7	78.8%	78.4%	78.1%
8	78.4%	78.5%	78.3%

**Table 2.4:** Structural symmetry for other grids

this insight to quantify the deviation from symmetry by looking at the following metrics.

- Symmetric structure. Given the sparsity structure of  $A$ , how many  $A_{ij}$  edges are also  $A_{ji}$  edges, irrespective of their numerical value. In the graph setting,

Level	Cylinder	Hyperboloid (1)	Hyperboloid (2)
3	72.2%	55.3%	46.2%
4	46.5%	46.8%	46.7%
5	38.6%	43.8%	40.2%
6	35.4%	42.4%	40.0%
7	34.5%	41.8%	39.6%
8	33.7%	41.6%	39.4%

**Table 2.5:** Numerical symmetry for other grids

## Chapter 2. Matrix Structure and Serial Solution

this relates to whether a directed edge from node  $i$  to node  $j$  implies an edge from node  $j$  to node  $i$ . The value of the coefficient might be different. We can quantify this measure of structural symmetric by comparing the nonzero structure of the matrix  $A$  with the nonzero structure of the matrix  $A^T$ . We report the ratio of the number of edges that are symmetric in structure to the total number of nonzeros in the matrix. If the matrix is symmetric, then all the edges have symmetric structure, and thus the number of symmetric edges is identical to the number of nonzeros of the matrix. We consider diagonal elements to be symmetric self-edges to the node itself. In the worst case, only the diagonal edges are symmetric in structure. Since we have a full diagonal, the worst possible value of symmetric structure is the number of diagonal elements, which is equal to the number of rows in the matrix:  $n$ . As we can see in table 2.3, the structural symmetry is quite high: around 75% of the edges are symmetric in structure. Table 2.4 list the structural symmetry for the other grids as well. We can see that the value of structural symmetry for all grids is comparable.

- Number of symmetric edges. If an edge  $A_{ij}$  exists, then we calculate if it is also an  $A_{ji}$  edge, where the weights on the two edges are within machine epsilon ( $\epsilon$ ) of each other. For a symmetric matrix, the number of symmetric edges should be equal to the total number of nonzeros in the matrix. Again,

## Chapter 2. Matrix Structure and Serial Solution

the worst possible value is  $n$ , the total number of diagonal elements in the matrix. Table 2.3 contains the observations from the various measures of symmetry. The columns 3 and 4 contain the measures of symmetry. As we increase the refinement of the grid, the symmetry drops at first, since the spherical grid at lower levels of refinement is very close to a coordinate grid. At higher levels of refinement, we notice that the numerical symmetry is roughly 39%, which means that 39% of the  $A_{ij}$  edges are also  $A_{ji}$  edges with an equal magnitude. We include the diagonals in this computation. There are around  $8n$  nonzero edges and  $n$  diagonals, so we expect the diagonals to contribute  $\frac{1}{8}$ , or 12.5% to this numerical symmetry. The rest, 27%, is due to nondiagonal edges. Table 2.5 list the numerical symmetry for the other grids as well. Again, all grids under consideration show very similar behavior.

- The final measure of symmetry we evaluate is the Frobenius norm of the skew symmetric and the symmetric components of the matrix. Every matrix  $A$  can be written as a sum of a symmetric and skew symmetric matrix  $A_H$  and  $A_K$ , where  $A_H = (A + A^T)/2$  and  $A_K = (A - A^T)/2$ . These norms give an idea of the relative significance of the symmetric and the skew symmetric part of the matrix. Table 2.6 contains the results from the norms of these matrices. From the table, we can see that  $\|A_H\|$  forms a significant part

Grid size	nnz(A)	$\ A\ $	$\ A_H\ $	$\ A_K\ $
469	2,335	$7.97 \times 10^2$	$7.94 \times 10^2$	$7.05 \times 10^1$
1,293	8,991	$5.61 \times 10^3$	$5.60 \times 10^3$	$2.54 \times 10^2$
4,277	33,311	$4.20 \times 10^4$	$4.19 \times 10^4$	$1.94 \times 10^3$
16,141	129,919	$3.34 \times 10^5$	$3.33 \times 10^5$	$1.56 \times 10^4$
64,005	520,191	$2.68 \times 10^6$	$2.68 \times 10^6$	$1.28 \times 10^5$
256,589	2,090,591	$2.15 \times 10^7$	$2.15 \times 10^7$	$1.03 \times 10^6$
1,027,813	8,377,183	$1.72 \times 10^8$	$1.72 \times 10^8$	$8.27 \times 10^6$

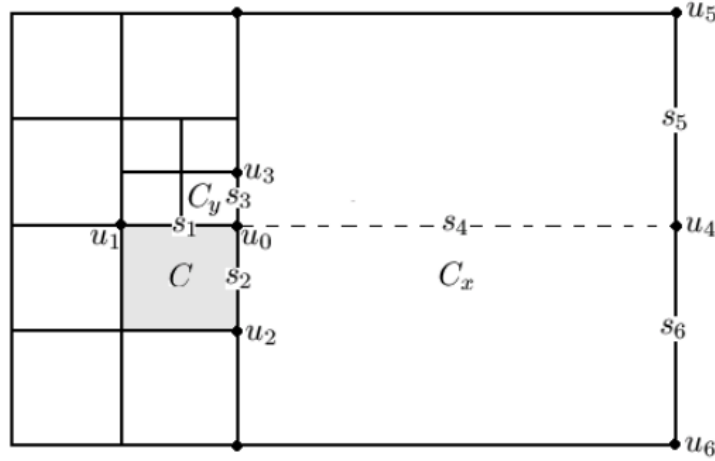
**Table 2.6:** Measure of symmetry of the spherical grid

of  $\|A\|$ , and is usually an order of magnitude larger than  $\|A_K\|$ . This can be partly attributed to the diagonals being significantly larger than the off-diagonal elements. For the finite difference discretization, the row sums are zero, and the magnitude of the diagonal values equals the sums of the magnitude of the off-diagonal elements across the row. This leads to the diagonal values being large and consequently, the norm of the symmetric part of the matrix is significant. This is also true of the other grids that we considered. Table 2.7 lists  $\|A_K\|$  as a fraction of  $\|A\|$  for the other grids. In all grids,  $\|A_K\|$  forms a insignificant part of  $\|A\|$ . Further, as the grid size increases,  $\|A_K\|$  diminishes relative to  $\|A\|$ .

As we can see from the previous analysis, the non-symmetric part of the matrices is smaller than the symmetric part. In this particular case, it helps to look at the graph structure of the matrix. This helps us identify the reasons for non-symmetry, and gives a better insight into their graph structure. To aid with the

Level	Cylinder	Hyperboloid (1)	Hyperboloid (2)
3	9.81%	10.9%	12.9%
4	5.88%	7.96%	8.94%
5	5.39%	6.36%	6.77%
6	5.29%	5.47%	5.86%
7	5.40%	5.00%	5.30%
8	5.51%	4.73%	4.97%

**Table 2.7:**  $\|A_K\|$  as a fraction of  $\|A\|$  for the other grids



**Figure 2.1:** Quadtree grid demonstrating discretization

understanding of the graph structure, we begin by describing the finite difference method developed by Min et al. to solve the Poisson equation on the octree grids.

Let us describe the finite difference method for a two dimensional quadtree grid, and a constant coefficient Poisson equation. The method is easily extended to three dimensions and the variable coefficient Poisson equation. Complete details are available in the original work describing this method [76]. Consider grid point  $u_0$  in figure 2.1. To form the finite difference discretization, the missing value  $u_4$



## Chapter 2. Matrix Structure and Serial Solution

is interpolated from the grid points  $u_5$  and  $u_6$  as follows:

$$u_4 = \frac{s_5 u_6 + s_6 u_5}{s_5 + s_6}$$

The discretization of  $\Delta u$  at the grid point  $u_0$  is then written as follows:

$$\begin{aligned} \left( \frac{u_4 - u_0}{s_4} - \frac{u_0 - u_1}{s_1} \right) \cdot \frac{2}{s_4 + s_1} &= u_{xx} \\ \left( \frac{u_3 - u_0}{s_3} - \frac{u_0 - u_2}{s_2} \right) \cdot \frac{2}{s_3 + s_2} \cdot w &= u_{yy} \end{aligned}$$

where  $w$  is

$$w = 1 - \frac{s_5 s_6}{(s_4 + s_1) s_4}$$

In the presence of the point  $u_4$ , no interpolation is required, and the scheme is identical to the standard central scheme.

$$\begin{aligned} \left( \frac{u_4 - u_0}{s_4} - \frac{u_0 - u_1}{s_1} \right) \cdot \frac{2}{s_4 + s_1} &= u_{xx} \\ \left( \frac{u_3 - u_0}{s_3} - \frac{u_0 - u_2}{s_2} \right) \cdot \frac{2}{s_3 + s_2} &= u_{yy} \end{aligned}$$

In the above finite difference discretization, there are two causes of unsymmetries.

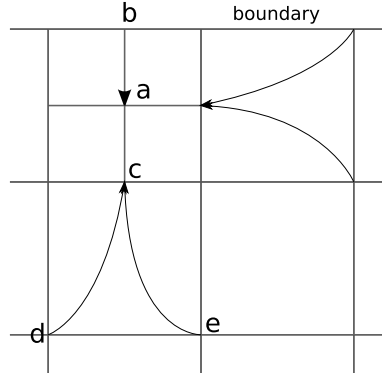
1. The first cause of unsymmetry is when a small octree node adjoins a larger octree node. In the figure 2.2, the smaller node has a corner  $c$ , which depends on points  $d$  and  $e$  for the finite difference approximation. However, points  $d$  and  $e$  do not depend on the point  $c$  for their approximations. This only

## *Chapter 2. Matrix Structure and Serial Solution*

happens when a smaller cell is adjacent to a larger cell. Due to the finite difference discretization, grid points are used as interpolants, introducing unsymmetric dependencies. From this, we can see that corners of smaller cells depend on corners of adjacent larger cells. The only way to eliminate this unsymmetry while retaining the finite difference scheme is by using coordinate grids. However, this would greatly increase the number of grid points, and so this form of unsymmetry is largely unavoidable.

2. The second cause of unsymmetry is Dirichlet boundary conditions. In figure 2.2, the point  $a$  depends on the boundary point  $b$ . Due to Dirichlet boundary conditions, the value of boundary point  $b$  is set independently. Thus point  $b$  is independent of  $a$ , and is independent of all its neighboring points as well. The unsymmetry due to Dirichlet boundary conditions can be eliminated. We could choose to use Neumann boundary conditions, or we could remove the boundary points altogether from the linear system.

These two reasons lead to directed edges between vertices in the graphs. We show a directed edge from grid point  $b$  to grid point  $a$  if the discretization of grid point  $a$  requires the grid point  $b$ , and the discretization of grid point  $b$  is independent of the grid point  $a$ .



**Figure 2.2:** The source of unsymmetry

### 2.1.2 Condition Number

Let us continue our look at the matrices by noting the growth in their condition numbers. The condition number of a matrix  $A$  is denoted by  $\kappa_2(A)$  and is defined as follows [44].

$$\kappa_2(A) = \|A\|_2 \cdot \|A^{-1}\|_2$$

Let the linear system be denoted by  $Ax = b$ , and let  $\Delta_b$  be the error in  $b$ . Let  $(x + \Delta_x)$  be the corresponding solution of the system  $A^{-1}(b + \Delta_b)$ . The condition number bounds the norm of  $\Delta_x$  given the norm of  $\Delta_b$  as follows:

$$\|\Delta_x\| \leq \kappa_2(A) \cdot \frac{\|\Delta_b\|}{\|b\|} \cdot \|x + \Delta_x\|$$

The condition number of  $A$  represents the maximum ratio of the relative error in  $x$  to the relative error in  $b$ .

Grid size	nnz(A)	Estimate of $\kappa_2(A)$
469	2,335	$4.87 \times 10^2$
1,293	8,991	$6.46 \times 10^3$
4,277	33,311	$9.20 \times 10^4$
16,141	129,919	$1.42 \times 10^6$
64,005	520,191	$2.28 \times 10^7$
256,589	2,090,591	$3.66 \times 10^8$
1,027,813	8,377,183	$5.88 \times 10^9$

**Table 2.8:** Condition number estimates for the spherical grid

Level	Cylinder	Hyperboloid (1)	Hyperboloid (2)
3	$4.67 \times 10^2$	$1.04 \times 10^3$	$5.92 \times 10^2$
4	$1.11 \times 10^4$	$1.97 \times 10^4$	$1.10 \times 10^4$
5	$1.78 \times 10^5$	$3.63 \times 10^5$	$1.82 \times 10^5$
6	$2.76 \times 10^6$	$6.32 \times 10^6$	$3.06 \times 10^6$
7	$4.30 \times 10^7$	$1.05 \times 10^8$	$5.03 \times 10^7$
8	$6.75 \times 10^8$	$1.73 \times 10^9$	$8.14 \times 10^8$

**Table 2.9:** Condition number estimates for other grids

Table 2.8 contains the condition numbers estimates generated using `condtest` [49, 55] in Matlab. As the number of grid points increases, the condition number increases rather rapidly. For comparison, table 2.9 shows that the condition number estimates from the other grids also increase rapidly.

## 2.2 GMRES and BiCGSTAB

To solve the grids obtained in original work by Gibou et al. [75], the authors used the BiCGSTAB iterative method with incomplete LU factor preconditioners. This method produces reasonable convergence, according to the authors. GMRES

## *Chapter 2. Matrix Structure and Serial Solution*

is an iterative method that can be used as an alternative to the BiCGSTAB iteration. We experimented with the GMRES method, with the accuracy of the final residual set to  $10^{-10}$  and the maximum number of iterations set to 1000. The method was set to restart after 50 iterations. This requires an extra memory overhead of  $50n$ , which is considerable. This overhead is larger than the space required for the matrix, since the matrix has around 10 nonzeros in every row, and thus has roughly  $10n$  nonzeros. The BiCGSTAB method does not have this memory overhead. Neither restarted GMRES nor BiCGSTAB guarantee convergence, and either may stagnate indefinitely.

Table [2.10](#) shows the results obtained with restarted GMRES and BiCGSTAB. Incomplete no-fill LU factors were used to precondition the linear system. While the number of iterations might differ, both GMRES and BiCGSTAB take approximately the same time when solving the linear systems, and work reliably. Since the memory overhead of GMRES is prohibitive and its convergence is not significantly better, we conclude that BiCGSTAB is to be preferred over GMRES for these linear systems.

Grid size	Time (s)		Iterations		nnz (L+U)
	GMRES	BiCGSTAB	GMRES	BiCGSTAB	
16,141	1	1	77	25	129,919
64,005	7	9	88	41	520,191
256,589	59	63	105	63	2,090,591
1,027,813	377	477	133	99.5	8,377,183
4,113,837	2,755	3,340	182	144.5	33,533,151
16,465,541	18,118	18,208	221	185.5	134,218,559

**Table 2.10:** Comparison of GMRES and BiCGSTAB with ILU(0) preconditioner

## 2.3 Preconditioners

Preconditioning involves modifying the linear system being solved, so that the new system has better convergence properties than the original system. The first mention of preconditioning techniques dates back to the work by Buleev [14] and Oliphant [79, 80]. Varga discussed a general class of matrix splitting techniques, which include incomplete factorizations [108, 109]. Evans is credited for the word “preconditioning” and evaluated the use of sparse LU factors for preconditioning [29]. Gustafsson proposed the modified incomplete factorization and demonstrated its improved spectral properties [46].

A preconditioner is a matrix,  $M$ , such that the matrix  $M^{-1}A$  has a smaller condition number than the matrix  $A$ . Formally, we aim to find a preconditioner  $M$  that satisfies two goals:

## Chapter 2. Matrix Structure and Serial Solution

- The eigenvalues of  $M^{-1}A$  are better clustered than that of the original matrix  $A$ . In turn, the condition number of the preconditioned system is lower than that of the original system.
- Applying  $M^{-1}$  should be computationally inexpensive.

Preconditioning theory is a large topic, and there are various preconditioners that can be used [93]. The simplest preconditioners are Jacobi or Gauss–Seidel preconditioners [109]. These mimic the fixed point iteration, and work well for some simple problems.

Convergence bounds for preconditioned iterative methods are known in some cases. When solving a preconditioned symmetric positive definite linear system using the conjugate gradient method, the number of iterations required to reduce the error by a constant  $\epsilon$  is bounded by  $\Theta\left(\sqrt{\kappa(M^{-1}A)}\right)$  [44, 65]. The conjugate gradient iteration converges in no more than  $n$  iterations in exact arithmetic. In practice  $n$  iterations is a prohibitive amount of work, so the bound is of little use, even if exact arithmetic could be used. GMRES iterations, if not restarted, also converge in no more than  $n$  steps. Again, this result is of little practical use, especially since GMRES without restarts would additionally require a large amount of space. Restarted GMRES and BiCGSTAB do not have any bounds on convergence. In the absence of theoretical performance bounds, the choice

## *Chapter 2. Matrix Structure and Serial Solution*

of a preconditioner is dictated largely by empirical performance. It has been historically difficult to obtain a robust preconditioner, in part due to a lack of theory about these methods [93]. For the matrices under consideration, no-fill incomplete LU factors, called ILU(0) [17], are a good preconditioner.

Incomplete factorizations are effective preconditioners for many problems. Meijerink and van der Vorst proved the existence of ILU factors for arbitrary fill patterns for M-matrices [74]. The work by Axelsson gives the bound on the eigenvalues of certain classes of matrices preconditioned with ILU and MILU factorizations [6]. While incomplete LU is an effective preconditioner, there are alternatives that might work better. Modified incomplete LU (MILU) [17, 46] is a simple modification to ILU, where the dropped values are added to the diagonal of the row. This modification helps in certain problems, where maintaining the row sum corresponds to a conservation of mass.

Incomplete LU factors can themselves be generated to different criteria. Incomplete LU can be performed with either levels of fill, denoted by ILU( $p$ ), or a fixed tolerance, denoted by ILUT. When using a fixed tolerance, a relative tolerance is calculated for every row. The relative tolerance for a row is the fixed tolerance multiplied by the 2-norm of the row. If a nonzero value is smaller than this relative tolerance, it is dropped [92]. In the levels of fill method [93], the first level, ILU(0), has a nonzero structure of the original matrix  $A$ . The level of every



## Chapter 2. Matrix Structure and Serial Solution

nonzero element of the matrix is defined as 0. The level of every zero element is set to  $\infty$ . Every element  $a_{ij}$  is obtained due to the update  $a_{ij} = a_{ij} - a_{ik} \times a_{kj}$ . The level of every element is defined as follows.

$$\text{lev}(a_{ij}) = \min\{\text{lev}(a_{ij}), \text{lev}(a_{ik}) + \text{lev}(a_{kj}) + 1\}$$

In this method, it is difficult to estimate the number of nonzeros at any given level  $p$ , for  $p > 0$ . At  $p = 0$ , the nonzero structure is identical to that of the original matrix. Due to its popularity, ILU( $p$ ) with  $p = 0$  is sometimes abbreviated to plain “ILU”. Table 2.11 shows the results of the performance of ILUT at a drop tolerance of  $10^{-3}$ , table 2.12 with  $10^{-4}$ , and table 2.11 with  $10^{-5}$ . Using MILU was not found to be beneficial, and we could not obtain convergence with the `milu` routine included with Matlab. ILU(0) factors were obtained using the `ilu` routine included in Matlab r2007a, with `setup.type = ‘nofill’` and all other variables set to their default values. The ILUT factors were obtained using the same `ilu` routine, with `setup.type = ‘ilutp’` and all other variables set to their default values. As we can see from the tables, reducing the drop tolerance increases the number of nonzero values in the factor. This significantly helps speed up the iterations of both GMRES and BiCGSTAB, though the gain in speed is offset by the loss of memory, and the time spent in generating the factor.

Grid size	GMRES		BiCGSTAB		ILUT	
	Iterations	Time	Iterations	Time	Gen time	nnz
469	54	.006	3	.005	0.0	13,412
1293	57	.048	4	.043	0.1	84,230
4277	61	.329	7.5	.361	3.1	453,954
16141	69	1.993	16.5	2.930	46.7	1,644,930

**Table 2.11:** Convergence of ILUT, with drop tolerance =  $10^{-3}$

Grid size	GMRES		BiCGSTAB		ILUT	
	Iterations	Time	Iterations	Time	Gen time	nnz
469	53	0.008	2	0.006	0.0	26,898
1293	54	0.048	2.5	0.041	0.2	155,285
4277	55	0.363	3.5	0.345	7.3	955,999
16141	58	1.945	5.5	2.110	114.1	4,088,419

**Table 2.12:** Convergence of ILUT, with drop tolerance =  $10^{-4}$

Grid size	GMRES		BiCGSTAB		ILUT	
	Iterations	Time	Iterations	Time	Gen time	nnz
469	52	0.008	1	0.005	0.1	38,952
1293	53	0.060	1.5	0.041	0.2	237,517
4277	54	0.485	2	0.369	14.3	1,627,881
16141	54	2.112	3	2.119	236.7	8,071,357

**Table 2.13:** Convergence of ILUT, with drop tolerance =  $10^{-5}$

## Chapter 2. Matrix Structure and Serial Solution

Our empirical observations in the previous section show that  $\text{ILU}(0)$  is a robust preconditioner for these problems.

$\text{ILU}(0)$  factors are sensitive to symmetric permutations of the matrix. We evaluate the impact of symmetric permutations on the convergence of  $\text{ILU}(0)$  preconditioners. For this, we generate three different types of permutations that are symmetrically applied to both the rows and the columns of the matrix.

- A random permutation generated with `randperm` in Matlab.
- An approximate minimum degree permutation generated with `symamd` in Matlab.
- A reverse Cuthill-McKee permutation generated with `symrcm` in Matlab.

Tables 2.14 and 2.15 list the number of iterations taken by BiCGSTAB and GMRES respectively. The accuracy of the iterative methods was set to  $10^{-6}$  and the maximum number of iterations was set to a hundred. The GMRES method was set to restart after 50 iterations. As the table demonstrates, there is very little difference in the convergence of preconditioners generated from symmetric permutations of the matrix.

$n$	Original	randperm	symamd	symrcm
469	59	61	61	58
1,293	63	63	63	63
4,277	69	69	69	69
16,141	77	77	77	77
64,005	88	88	88	88
256,589	105	105	105	105

**Table 2.14:** Iterations of BiCGSTAB for matrix generated with various symmetric permutations, preconditioned with ILU(0)

$n$	Original	randperm	symamd	symrcm
469	6.0	8.0	8.5	6.0
1,293	10.5	10.5	10.5	10.5
4,277	17.5	17.5	17.5	17.5
16,141	25.0	25.0	25.0	25.0
64,005	41.0	41.0	41.0	41.0
256,589	63.0	63.0	63.0	63.0

**Table 2.15:** Iterations of GMRES for matrix generated with various symmetric permutations, preconditioned with ILU(0)

## Chapter 3

# Graph Coloring and Parallel Approaches

*The worthwhile problems are the ones you can really solve or help solve, the ones you can really contribute something to. ... No problem is too small or too trivial if we can really do something about it.*

Richard Feynman (1918-1988) [[35](#)]

Our goal is to enable researchers to generate and solve large systems of the Poisson equation discretized with octrees. As seen in earlier chapters, 3D grids grow very quickly at higher levels of refinement. To continue modelling physical phenomena at higher levels of refinement, we hit two distinct limitations.

- **Space:** No single computer has sufficient main memory to hold the entire linear system.
- **Time:** The amount of time spent in solving the linear system grows beyond reasonable limits. Since the Poisson solver forms the computationally intensive kernel of many processes, a slow solver impacts the entire simulation.

### *Chapter 3. Graph Coloring and Parallel Approaches*

The conventional response to both these limits is to seek to parallelize the computation. First, the computation is partitioned into pieces that can fit in the storage of individual processors. This alleviates the space problem, since sub-problems are smaller than the original problem. Further, the processors work largely independently, and thus can calculate the solution much quicker than a single processor working by itself. Current architecture technologies also motivate looking for parallelism in computational processes. Multicore processors are popular in workstations, which suggests that speed improvements could be achieved for small problems as well.

In the previous chapter, we established BiCGSTAB with the ILU(0) preconditioner as the iterative method of choice for our finite difference grids. The BiCGSTAB algorithm involves matrix-vector products and inner product calculations [8]. Many studies have investigated the computation of matrix-vector products in parallel [15, 23, 85, 100, 103]. Triangular solvers, on the other hand, are relatively harder to parallelize. There is a lot of research pertaining to parallel solution of dense triangular systems. Romine and Ortega [89] demonstrated a parallel fan-in algorithm for dense parallel matrices, and Li and Coleman [68] demonstrated a radically different parallel algorithm based on a ring structure of computation. Heath and Romine authored a good survey of preliminary techniques [51], and the stability of parallel dense solvers was investigated by Higham [54].

### *Chapter 3. Graph Coloring and Parallel Approaches*

Reordered matrices are often used to preserve sparsity in sparse triangular factorization [4, 24, 102]. The early work by Tinney and Walker [102] shows how the serial computation of a linear solver can benefit from a reordered matrix, and from storing the triangular factors of the reordered matrix. This research has been extended to parallel systems [28].

In many cases, the ILU factors themselves must be generated in parallel, and there is active research [56] in this area. Matrix reorderings and graph colorings are often a starting point for distributing the computation. Graph colorings for parallel computation have been a subject of much research. Schreiber and Tang [94] developed the use of coloring to partition matrices into blocks in which the diagonal blocks are diagonal matrices. Jones and Plassman [63, 64] discuss the use of colorings to speed up the parallel performance of conjugate gradient iterations with incomplete Cholesky used as a preconditioner.

An alternative technique is due to Alvarado and Schreiber [3]. In this work, the triangular matrix is decomposed into a product of triangular matrices. The goal is to achieve a small number of triangular factors, each of which can be inverted in place, i.e., the inverse is as sparse as the factor. Much research has been done on this elegant idea, which was extended to include Cholesky factors in [2]. Recent work by Van Duin extended the result to reduce the number of partitions [106, 107].

### *Chapter 3. Graph Coloring and Parallel Approaches*

For symmetric matrices, the Cholesky factors are used, and existing methods speed up solutions using the elimination tree of the Cholesky factors [45]. Pothén and Alvarado [88] develop a specialization of the ILU factor work to Cholesky factors.

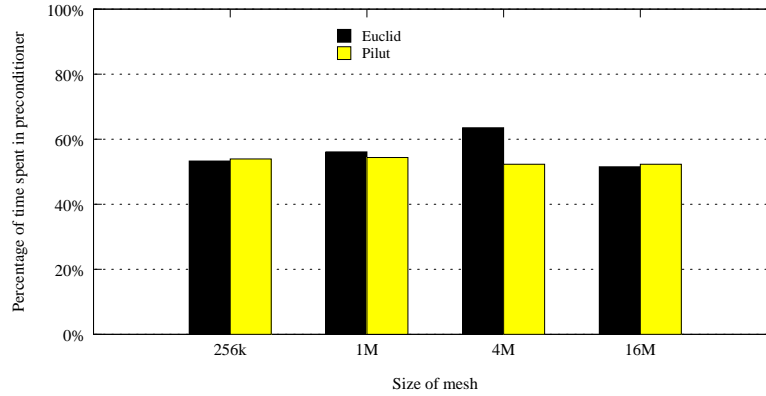
Gallivan and Ng [38] present an extensive survey of the development of parallel algorithms for matrix computations.

This chapter is arranged as follows. First we demonstrate that the triangular solver is the dominant computation in the preconditioned iterative linear solve. Then, we propose a matrix reordering that makes it easy to bound the chromatic number of the triangular factors of the octree grids. We propose a coloring to solve the triangular system in parallel, and then demonstrate the parallel performance of this algorithm.

## **3.1 Triangular solves dominate performance**

To investigate the dominant computation in parallel, we use the Pilut and the Euclid preconditioners from the popular HyPRE package [31]. In all these experiments, we use a spherical grid at increasing levels of refinement. The graphs show the time spent in the triangular solver as a fraction of the total linear solve time.

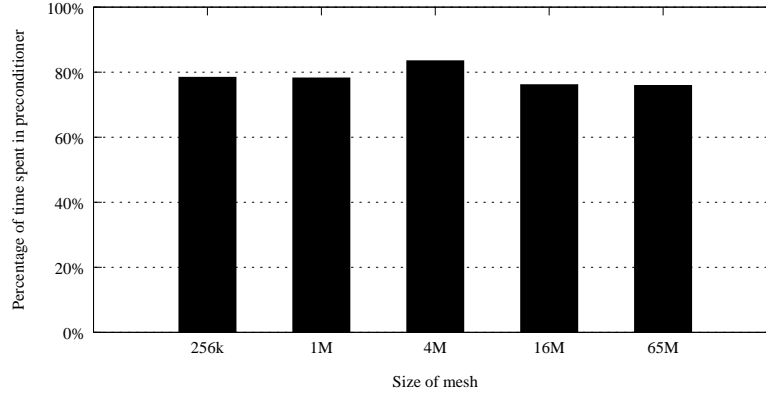




**Figure 3.1:** Percentage of time spent in the preconditioner in serial

First, we show the fraction of time taken by the preconditioner for serial runs. The preconditioner used is a triangular solver, and so both “time spent in the preconditioner” and “time spent in the triangular solver” refer to the same thing. Figure 3.1 shows the fraction of triangular solve time spent in Pilut and Euclid. Pilut is an ILUT implementation, in which we set the drop tolerance to 0.1 in order to generate triangular factors with about  $8n$  nonzeros. Euclid is an ILU(p) implementation, in which we generated ILU (0) factors. In serial, the triangular solves are half the computation of the entire linear solve.

Next, we show the fraction of time taken by the triangular solver as larger grids are solved in parallel. Figure 3.2 shows the fraction of time spent in the triangular solver for an increasing size of grids, for eight processors. The fraction of time spent in the preconditioner stays relatively constant at 75%–85%.

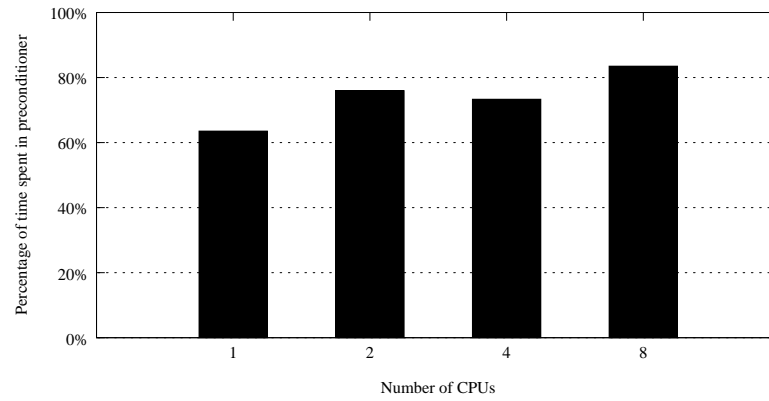


**Figure 3.2:** Percentage of time spent in the preconditioner in parallel

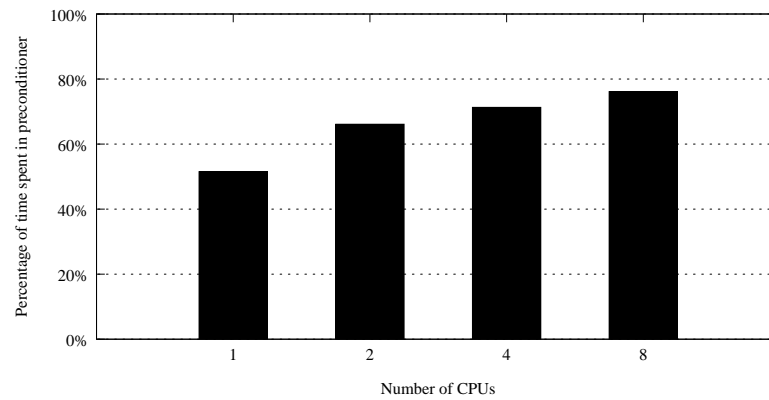
Finally, we demonstrate that the fraction of time taken by the triangular solver increases as the number of processors is increased. For these experiments, we used ILU(0) factors generated through Euclid on a varying number of processors. Figures 3.3 and 3.4 show the fraction of time spent in the triangular solver for grids with four million and sixteen million grid points respectively. The graphs demonstrate that the amount of time spent in the preconditioner forms a larger fraction of the total linear solve time as the number of processors is increased. In order to efficiently solve the linear systems in parallel, we must therefore speed up the triangular solver.

## 3.2 Matrix Reordering

We seek to exploit the graph structure of the octree grids in order to parallelize the triangular solver. As mentioned in section 2.1, the grids are unsymmetric, and



**Figure 3.3:** Percentage of time spent in the preconditioner for 4M grid



**Figure 3.4:** Percentage of time spent in the preconditioner for 16M grid

### *Chapter 3. Graph Coloring and Parallel Approaches*

the lack of symmetry is inherent to the octree grids. Unsymmetries in the matrix correspond to unidirectional edges from one octree corner to another in the graph structure corresponding to the matrix, while symmetric edges are bidirectional edges. We propose a matrix reordering that has the attractive property of directing all the unidirectional edges from lower numbered grid points to higher numbered grid points. In the matrix setting, this corresponds to limiting all the nonsymmetric edges to the lower triangle of the matrix. Let us define our ordering formally. Later we shall illustrate the ordering with an example.

Let  $v_i$  denote the octree grid points, and  $C_j$  denote octree cells. Let the height of the octree cell  $C_j$  in the octree be denoted by  $h(C_j)$ , with the height of the root set to 0, the height of its children set to 1, and so on. The volume of an octree cell is inversely related to its height. Grid points that are at the centroid of any octree cell are called “central” grid points.

Every octree cell has exactly 8 corners, being a cuboid. Every grid point is the corner of at least one octree cell, though it might form the corner of multiple octree cells. If a grid point  $v_i$  forms the corner of octree cell  $C_j$ , then we say that  $C_j$  adjoins  $v_i$ . Let  $C(v_i)$  denote the set of all octree cells that adjoin a grid point  $v_i$ . Among set  $C(v_i)$ , let the octree cell of least height (thus largest volume) be called the largest adjoining cell and be denoted by  $\hat{C}(v_i)$ . Let the height of a grid point  $h(v_i)$  be defined as follows.

$$h(v_i) = \begin{cases} 2 \times h(\hat{C}(v_i)) + 1 & v_j \text{ is a central point} \\ 2 \times h(\hat{C}(v_i)) & \text{otherwise} \end{cases}$$

The height of grid points  $h$  defines a partial ordering on grid points. This is a partial ordering since there are many grid points at any particular height. For a simple example, at height 0, there are 8 grid points — corners of the root cell. We define a reordering for the matrix as any reordering which satisfies this partial ordering.

To illustrate the reordering, we use a quadtree grid, rather than an octree grid. The ideas are nearly identical, and quadtree grids are easier to represent in figures.

The height of  $\hat{C}(v_i)$  are demonstrated in figure 3.5. Squares mark grid points with  $h(\hat{C}(v_i)) = 0$ . These are corners of the root cell. Circles mark the grid points with  $h(\hat{C}(v_i)) = 1$ , and stars mark the grid points with  $h(\hat{C}(v_i)) = 2$ .

The numbering proceeds according to the height of the grid points. Consider the quadtree in the figure 3.6. Figure 3.7 shows a spy plot of the matrix resulting from the numbering. Dots represent a nonzero element and whitespace represents zero values. The grid points with the least height are the corners of the original domain. There are four such grid points in a quadtree, and eight in an octree. These are points 1, 2, 3, and 4, which are boundary points. These can be renumbered among themselves without any loss since they are at the same height. Their

### *Chapter 3. Graph Coloring and Parallel Approaches*

corresponding rows in the matrix only have diagonal elements as shown in the spy plot in figure 3.7.

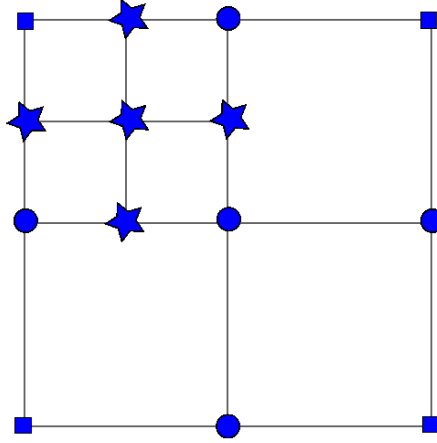
The grid points at height 2 are 5, 6, 7, and 8. Again, their rows are only nonzero in the diagonal elements since they are all boundary points.

The grid point at height 3 is 9. It is a central point, lying at the centroid of the root quadtree cell. It has symmetric dependencies with grid points 12 and 13. It has nonsymmetric dependencies on grid points 7 and 8, both of which are numbered lower than itself. The nonsymmetric dependencies are the elements  $A(9, 7)$  and  $A(9, 8)$  in the matrix.

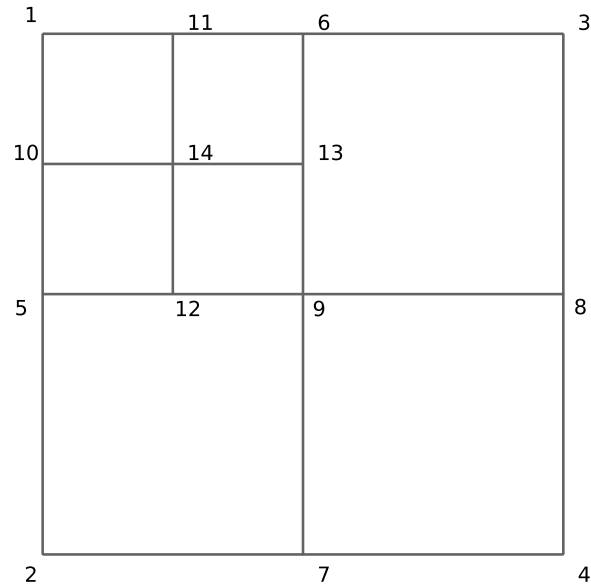
The grid points at height 4 are 10, 11, 12 and 13 and the grid point at height 5 is 14.

In the final matrix, the diagonal is nonzero, and all the unsymmetric values are now in the lower diagonal. Notice that every nonzero value  $A(i, j)$  in the upper triangle of the matrix is also a corresponding nonzero value  $A(j, i)$ . This can be seen for symmetric pairs  $A(9, 12), A(12, 9)$  and  $A(9, 13), A(13, 9)$ .

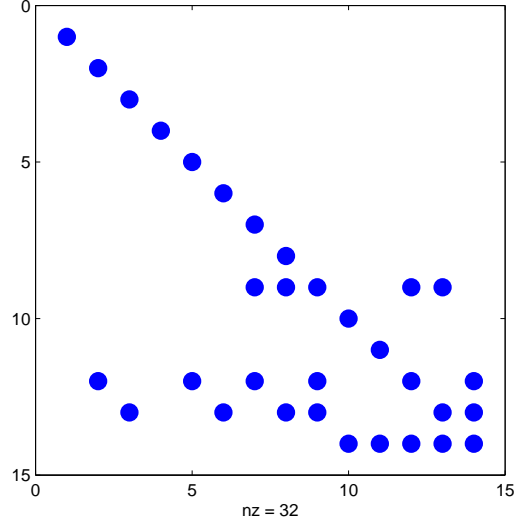
Recall that section 2.1 contained the two reasons for the existence of unsymmetric edges. We now prove that this matrix reordering constrains the unsymmetric edges to edges that are directed from lower numbered vertices to higher numbered vertices.



**Figure 3.5:** Single quadtree showing the different heights of  $\hat{C}(v_i)$



**Figure 3.6:** Quadtree numbered according to the heights of grid points



**Figure 3.7:** Spy plot of the simple quadtree grid renumbered

First let us prove that if edge  $(v_i, v_j) \in E$  is also the side of an octree cell, and  $v_i$  is a boundary point, then  $h(v_i) < h(v_j)$ .

*Proof.* Since  $(v_i, v_j) \in E$  is also the side of an octree cell, the grid points  $v_i$  and  $v_j$  are neighbors along some coordinate direction. Since  $v_i$  is a boundary point,  $h(\hat{C}(v_i)) \leq h(\hat{C}(v_j))$ .

If  $h(\hat{C}(v_i)) < h(\hat{C}(v_j))$ , then  $h(v_i) < h(v_j)$ .

If  $h(\hat{C}(v_i)) = h(\hat{C}(v_j))$ , then  $v_j$  must be a central point, since  $v_i$  is a boundary point, and thus  $h(v_i) = h(\hat{C}(v_i))$ , and  $h(v_j) = h(\hat{C}(v_i)) + 1$ . Again  $h(v_i) < h(v_j)$ .

In both cases  $h(v_i) < h(v_j)$  and the result holds.  $\square$



### Chapter 3. Graph Coloring and Parallel Approaches

We know that the unsymmetric edges are due to boundary values and due to large cells being adjacent to small cells. Let us show in each case that the unsymmetric edge is now being directed from a lower numbered grid point to a higher numbered grid point.

*Proof.* Consider the unsymmetric edge introduced due to large octree cells being adjacent to small octree cells. Let  $(v_i, v_j) \in E$  be an unsymmetric edge, where grid point  $v_i$  forms an interpolant for a missing grid point adjacent to  $v_j$ . Since interpolants are formed from corners of larger cells (according to the finite difference discretization),  $v_i$  adjoins a larger octree cell, and  $v_j$  adjoins a smaller octree cell. Thus,  $h(v_i) < h(v_j)$ , and  $v_i$  must be numbered lower than  $v_j$ .

Consider the unsymmetry introduced due to Dirichlet boundary values. Let  $v_i$  be a grid point that is on the boundary. Let  $v_j$  be a grid point that is not on the boundary, and  $(v_i, v_j) \in E$ . There are two possibilities.

- Edge  $(v_i, v_j)$  does not form the side of an octree cell. In this case,  $v_i$  forms an interpolant for a missing grid point adjacent to  $v_j$ . By the previous result,  $v_i$  adjoins a larger octree cell, and  $v_j$  adjoins a smaller octree cell. Thus,  $h(v_i) < h(v_j)$ , and  $v_i$  must be numbered lower than  $v_j$ .
- Edge  $(v_i, v_j)$  forms the side of an octree cell. This octree cell adjoins both  $v_i$  and  $v_j$ . By proof [3.2](#)  $v_i$  must be numbered lower than  $v_j$ .

□

Due to the finite difference discretization of the Poisson equation in the octree grids, we know that the number of nonzeros in a row cannot be greater than 11. With this numbering, the upper triangle has bounds on the row nonzero counts as well as the column nonzero counts: the row nonzero count for the upper triangle cannot exceed 11, and the column nonzero count cannot exceed 6. The limit on the row nonzero count for the lower triangle is 10, and there is no bound on the column nonzero count, due to the unsymmetric edges.

The above proof limits the number of edges incident on a grid point from lower numbered grid points. To see this, notice that the indegree of a grid point now corresponds directly to the edges from lower numbered grid points. Since the indegree is bounded above by 10, the number of edges from lower numbered grid points is also bounded above by 10. The real advantage appears when we couple this matrix reordering with graph coloring.

### 3.3 Graph Coloring

Graph coloring [26] is a method of assigning labels, or colors, to the vertices in a graph so that no like-colored vertices have an edge joining them. Let  $G = (V, E)$  be a graph with vertex set  $V$  and edge set  $E$ . Let  $S$  be a set, and  $c : V \rightarrow S$

### *Chapter 3. Graph Coloring and Parallel Approaches*

be a map such that  $c(x) \neq c(y)$ , when  $(x, y) \in E$ . When  $|S| = k$ , the graph is said to be  $k$ -colorable. We are interested in the smallest  $k$ , for which the graph is  $k$ -colorable. The smallest such  $k$  is called the chromatic number of a graph. In general, obtaining the chromatic number of a graph is an NP-hard problem [39]. If the octree grids are uniformly refined to make coordinate grids, the graph structure is identical to that of the 2D and 3D model problem. The graph structure of both the 2D and the 3D model problem are bipartite and hence 2-colorable. The quadtree grids in 2D without unsymmetric edges are planar, and it has been shown that planar graphs are 4-colorable [5]. If a bound exists on the degree of the vertices, a constructive greedy algorithm [26] can bound the chromatic number of the graph, and provide the coloring as well. Graphs obtained from quadtrees are not 2-colorable in general: Figure 3.6 shows an example of a simple quadtree which is not 2-colorable. Graphs of octree grids have a bounded indegree, since the finite difference approximation cannot depend on more than 10 corners, and thus the row nonzero count is bounded above by 11: 10 for the dependencies, and 1 for the diagonal nonzero value. There is no bound on the outdegree: we can construct octree grids with a large octree cell adjacent to any arbitrary number of small octree cells. Such a construction would lead to a large number of outgoing edges from the corners of the larger octree cell to all adjacent corners of small octree cells. The outdegree of the corners on the large octree cells is therefore

### Chapter 3. Graph Coloring and Parallel Approaches

unbounded. This corresponds to the lack of a bound on column nonzero counts in the coefficient matrix.

In this section, we use the matrix reordering from the previous section to propose a coloring of arbitrary octree grids. We follow the greedy graph coloring algorithm in the order of the numbering. At every grid point, we only examine the grid points that are labelled lower than the current grid point, and choose the smallest color that can be assigned without conflict.

The greedy algorithm combined with the previous numbering allows us to color the octree meshes of the Poisson equation using no more than 11 colors.

*Proof.* The octree mesh already has a bounded indegree:  $\Delta(G) = 10$ . From the previous numbering, the indegree corresponds to edges from lower numbered grid points. Since a grid point can have a maximum of 10 edges from lower numbered grid points, we need  $\Delta(G) + 1 = 11$  colors. Hence the graph requires no more than 11 colors.  $\square$

This result is independent of the number of grid points in the octree mesh, and the particular variable coefficient Poisson equation being solved. Grid points corresponding to a single color are independent of each other, and their values can be solved completely in parallel. The coloring translates directly into a coloring for the triangular factors, since the graph structure of  $\text{ILU}(0)$  is identical to that

Grid Size	Colors
469	4
1,293	6
4,277	7
16,141	7
64,005	7
256,589	7
1,027,813	8
4,113,837	8
16,465,541	9
65,873,965	9

**Table 3.1:** Growth of the number of colors for spherical grid

of  $A$ . In particular,  $L$  has edges from  $A$  which are from lower numbered vertices to higher numbered vertices, and  $U$  contains edges from higher numbered vertices to lower numbered vertices. At the time of generation of the coefficient matrix  $A$ , we can immediately color the grid points, and the coloring can be provided to the triangular solver to help speed up both the triangular solves. Table 3.1 lists the number of colors for various levels of refinement of the spherical grid, along with the number of grid points at that level. Fewer than 11 colors suffice, even when the grids are quite large. We also list the number of colors required for the other grids in tables 3.2, 3.3, and 3.4. Even in the largest grids, only 9 colors are required, and we have been unable to generate grids requiring more than 9 colors.

For  $m$  colors, the matrix has a block structure shown below.

Chapter 3. Graph Coloring and Parallel Approaches

Grid Size	Colors
577	4
2,613	6
11,257	7
46,877	7
191,409	8
773,045	8
3,106,249	8
12,454,029	8
49,870,401	9

**Table 3.2:** Growth of the number of colors for grid of hyperboloid of 1 sheet

Grid Size	Colors
511	4
2,057	6
7,723	7
30,789	8
123,607	8
494,097	8
1,971,731	8
7,880,573	9
31,503,823	9

**Table 3.3:** Growth of the number of colors for grid of hyperboloid of 2 sheets

Grid Size	Colors
605	4
2,185	6
7,893	7
29,537	8
113,773	8
446,073	8
1,766,021	8
7,027,345	8
28,035,741	8

**Table 3.4:** Growth of the number of colors for cylindrical grid

$$A = \begin{pmatrix} D_1 & U_{1,2} & \dots & U_{1,m} \\ L_{2,1} & D_2 & \dots & U_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ L_{1,m} & L_{2,m} & \dots & D_m \end{pmatrix}$$

$D_i$  are diagonal block matrices while  $U_{i,j}$  and  $L_{i,j}$  are rectangular block matrices.  $L_{2,1}$  is the dependency of color 1 grid points on color 2 grid points, while  $U_{1,m}$  is the dependency of color  $m$  grid points on color 1 grid points. The greedy algorithm orders the colors, giving rise to a notion of a “lower” color and a “higher” color, according to this ordering. Thus, color 1 is lower than any other color. In practice, the greedy algorithm assigns many more grid points to the lower colors than the higher colors. Table 3.5 shows the number of grid points in each color for a spherical grid containing 65 million grid points. The first two colors, with 40 million grid points, make up roughly two thirds of the grid.

When the ILU (0) factors of the matrix  $A$  are generated, they share the same nonzero structure as the matrix. Thus, the incomplete no-fill LU factors have the following structure.

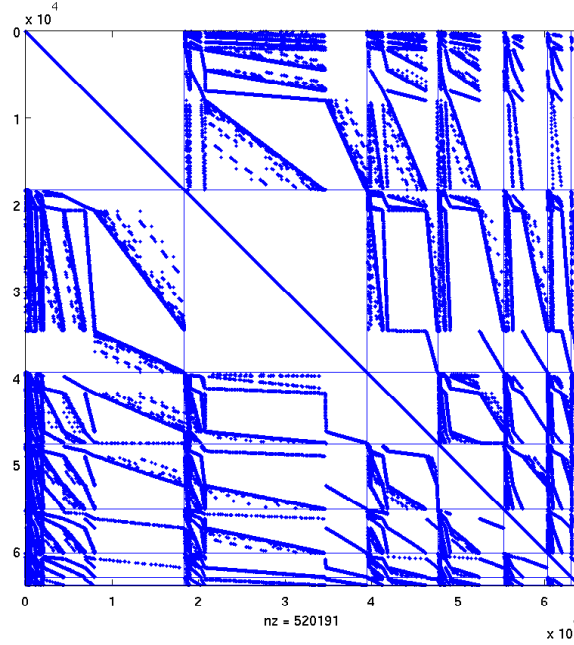
Color	Grid points
1	19,043,115
2	21,647,326
3	8,184,300
4	6,950,712
5	5,636,454
6	3,480,702
7	890,388
8	40,788
9	180

**Table 3.5:** Number of grid points of each color for the spherical grid with 65M points

$$L = \begin{pmatrix} D_1 & & & \\ L_{2,1} & D_2 & & \\ \vdots & \vdots & \ddots & \\ L_{1,m} & L_{2,m} & \dots & D_m \end{pmatrix}, \quad U = \begin{pmatrix} D_1 & U_{1,2} & \dots & U_{1,m} \\ & D_2 & \dots & U_{2,m} \\ & & \ddots & \vdots \\ & & & D_m \end{pmatrix}$$

The data dependency is now limited. Within a single color, there is no dependency, and the linear solver can work in parallel. The only data dependency is across colors. In the lower triangle,  $L$ , for instance, nodes colored with the second color only depend on nodes colored with the first color. A visual representation of the reordered matrix is shown in figure 3.8. The figure is a sparse matrix plot generated by the `spy` command in Matlab. Every dot represents a nonzero value in the matrix. Vertical and horizontal lines have been added to demarcate the boundary between the colors. The figure shows the pictorial representation of the





**Figure 3.8:** Spy plot of matrix reordered with the colored ordering

matrix of a spherical grid with 64,005 grid points, after applying the coloring re-ordering. For the grid shown, the greedy algorithm requires seven colors, of which the first two colors are the most prominent, and are clearly visible as the first two block diagonals. Later colors have successively fewer points, with the last color containing just 64 points.

### 3.4 Parallel Triangular Solves

In order to solve the triangular systems in parallel, we use the fan-out vector-sum algorithm first described by Kuck [67]. The paper describes the working of

### *Chapter 3. Graph Coloring and Parallel Approaches*

a vector-sum algorithm on a dense system, but the idea applies equally well to a sparse linear system. A promising variant, the dense parallel wavefront algorithm first outlined by Evans and Dunbar [30] can also be used, though the implementation might be considerably more difficult for arbitrary sparse matrices. The difficulty arises from the uncertain nonzero structure of a row in a sparse matrix. For dense matrices, the data dependency is known at the outset—when solving  $Ly = b$ , the computation of  $y_i$  depends on **all** points  $y_1, \dots, y_{i-1}$ . For a general sparse linear system,  $y_i$  depends on a very few of the  $y_1, \dots, y_{i-1}$  values, making the data dependency much harder to guess. For a survey of dense matrix triangular solver techniques, we refer the reader to the survey by Heath and Romine [51].

#### **3.4.1 Partitioned Inverses**

An alternative technique called the method of partitioned inverses is due to Alvarado and Schreiber [3]. Their main idea consists of decomposing the triangular matrix into a product of triangular matrices which can be inverted in place. Such a decomposition always exists. Let  $L$  be a triangular matrix. The elementary triangular factors  $L_1 L_2 \dots L_n$  described below constitute such a decomposition.

$$L = L_1 L_2 \dots L_n$$

where

$$\begin{aligned}(L_k)_{ik} &= L_{ik} \quad \text{for } k \leq i \leq n, \\ (L_k)_{jj} &= 1 \quad \text{for } j \neq k, \\ (L_k)_{ij} &= 0 \quad \text{otherwise.}\end{aligned}\tag{3.1}$$

The number of factors is also called the number of partitions. In this example, there are  $n$  partitions. The number of partitions represents the communication cost, and with partitioned inverses, we are interested in minimizing the number of partitions. The most recent work in this area is due to Van Duin [106, 107], in which algorithms RPO2 and PO1 are shown to reduce the number of partitions.

### 3.4.2 Graph Coloring

In the partitioned inverses method, extra work is required to compute partitions after the triangular factors are formed. The graph coloring method produces a reordered matrix at the grid generation stage. Once the matrix has been reordered, no extra computation is needed to color the triangular factors, if no-fill incomplete LU factors are used.

We illustrate the graph-coloring triangular solver for the lower triangular matrix  $L$ . The equation to be solved is  $Ly = b$  where  $L$  has the structure shown

below.

$$\begin{pmatrix} D_1 & & & \\ L_{2,1} & D_2 & & \\ \vdots & \vdots & \ddots & \\ L_{1,m} & L_{2,m} & \dots & D_m \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Each color is divided equally among the processors available, and rows of vectors  $y$  and  $b$  are also distributed among the processors. The rows of vector  $b$  are distributed in the same manner as the distribution of the rows of the factors  $L$  and  $U$ . The local rows of color  $i$  at processor  $j$  are denoted by  $b_{ij}$ . We refer to all the rows of color  $k$  in vector  $y$  as  $y_k$ . Algorithm 1 shows the algorithm for solving  $Ly = b$ .

The structure of the matrix  $U$  is given below. It also has a block structure, which allows us to perform triangular solves with minimum computation.

$$U = \begin{pmatrix} D_1 & U_{1,2} & \dots & U_{1,m} \\ & D_2 & \dots & U_{2,m} \\ & & \ddots & \vdots \\ & & & D_m \end{pmatrix}$$

The algorithm for solving  $Uy = b$  is nearly identical to the algorithm for solving  $Ly = b$ , with the colors being solved in decreasing order rather than increasing order, and is outlined in algorithm 2.

**Input:** Block diagonal  $L$  and vector components  $b_j$

**Output:** Vector  $y$

**foreach** *Processor  $i$  in parallel* **do**

**for**  $j \leftarrow 1$  to  $m$  **do**

**foreach**  $k, k < j$  **do**

$b_{ij} = b_{ij} - L_{kj} \cdot y_k$  ;

**end**

$y_{ij} = D_i^{-1} b_{ij}$  ;

**barrier**

**end**

**end**

**Algorithm 1:** Solving  $Ly = b$  in parallel

**Input:** Block diagonal  $U$  and vector components  $b_j$

**Output:** Vector  $y$

**foreach** *Processor  $i$  in parallel* **do**

**for**  $j \leftarrow m$  **to** 1 **do**

**foreach**  $k, k < j$  **do**

$b_{ij} = b_{ij} - L_{kj} \cdot y_k$  ;

**end**

$y_{ij} = D_i^{-1} b_{ij}$  ;

**barrier**

**end**

**end**

**Algorithm 2:** Solving  $Uy = b$  in parallel

### 3.4.3 Oski

Many improvements to the above implementation can be made, depending on the structure of the octree grids. In both algorithms 1 and 2, the step  $b_{ij} = b_{ij} - L_{kj} \cdot y_k$  constitutes a matrix-vector product. This step can be tuned by the use of an efficient matrix vector library like Oski [110]. The Oski library includes many impressive techniques that tune the matrix-vector product to the architecture of the machine used.

We implemented the triangular solver with the Oski library, and our results are presented below. We measure the time taken for the triangular solver alone, to quantify the gain in performance when using the Oski library. Table 3.6 presents the time taken (in seconds) by the Oski library on the lower triangular factor of the spherical grid containing 65 million rows. The triangular solver was run 100 times, and the times reported are the average run time per triangular solve. This was done since Oski needs a few iterations to learn enough about the structure of the matrix to tune the matrix-vector product. The results are inconclusive due to our lack of experience with tuning matrix vector products using the Oski library. A careful implementation using the Oski library might provide more speedups.

Processors	Trisolve (L)	Oski (L)	Trisolve (U)	Oski (U)
1	5.48	5.61	5.39	5.68
2	3.03	3.16	2.93	3.39
4	1.85	1.74	1.74	1.36
8	1.23	1.33	1.19	1.34

**Table 3.6:** Triangular solve timings with Oski on 65M grid

Processors	Trisolve (L)	Oski (L)	Trisolve (U)	Oski (U)
1	1.35	1.39	1.43	1.36
2	0.78	0.83	0.77	0.87
4	0.54	0.57	0.53	0.66
8	0.32	0.48	0.31	0.40

**Table 3.7:** Triangular solves with Oski on 16M grid

### 3.4.4 Memory Bandwidth

We performed some tests to calculate the memory transfer rates on the hardware that we were using. The aim of this was to measure the memory transfer rates that we observed and to compare them to the maximum possible memory transfer rates on the hardware. The computer in use is an 8-socket computer, with 8 dual core 2Ghz processors. The processors are connected together by a HyperTransport interconnect. The sockets are directly connected to 8 Gb of local memory, and the bandwidth of this processor-to-memory link is 200M transactions per second, where each transaction contains 144 bits [104, 105]. This corresponds to a memory bandwidth of 28.8 gigabits per second.

We first experimented with the time taken for solving a diagonal system. 8.86 seconds were taken to solve the diagonal system containing 600 million rows. Let



### *Chapter 3. Graph Coloring and Parallel Approaches*

us analyze this computation. Solving a diagonal system is equivalent to performing the element-wise division of two vectors. In this case, two double vectors with 600 million rows are used, and the result is placed in a third vector of doubles. Each vector has 8 bytes, or 64 bits. This results in a memory bandwidth of:

$$\frac{1}{8.86} \times 3 \times 64 \times 600 \times 10^6 = 12.991 \times 10^9$$

Thus, we obtain a memory bandwidth of 13 gigabits per second, which compares favorably with the peak memory bandwidth of 28.8 gigabits per second.

We now measure the memory bandwidth for a triangular solve. The diagonal of the triangular matrix is stored as a dense vector. This takes  $n$  double values. Let  $w$  be the number of off-diagonal nonzeros in the matrix. The off-diagonal matrix blocks are stored in the CSR format. This CSR structure takes  $w$  double values for the entries in the matrix,  $w$  integers for the column indices, and  $n - c_1$  integers for the row pointers, where  $c_1$  is the number of elements in the first color. This is because the triangular system representing the first color is a diagonal block, and does not have any off-diagonal elements. In addition,  $n$  doubles are read and written in the triangular solve. Thus the total memory accessed for the triangular solve is  $8(w + 3n) + 4(n - c_1)$  bytes. When  $t$  seconds are spent in the triangular solve, the memory bandwidth in bits per second is given by:

$$\frac{1}{t} \times 8 \times ((w - c_1) \times 4 + (w + 3n) \times 8)$$

The time taken for the lower triangular system containing 4113837 elements is 0.35 seconds. In this case  $t = 0.35$ ,  $n = 4,113,837$ ,  $w = 32,344,588$  and  $c_1 = 1,188,563$ . Thus the memory bandwidth is 10.6 gigabits per second.

### 3.5 Performance results

In this section, we present our results from running the parallel triangular solver as a preconditioner for an iterative linear solver. We demonstrate speedups obtained by using this algorithm with a varying number of processors. We used an 8 socket, 16 core shared memory computer running CentOS 5.0 [101] for our tests. Each socket had a dual core Opteron 2.2 Ghz. Each socket could access eight gigabytes of local memory, and a HyperTransport bus connected the sockets.

To evaluate the performance of the linear solver, we generated spherical meshes at varying depths of refinement. Figure 3.9 shows the performance of the linear solver on spherical grids containing 4 million, 16 million and 65 million points respectively.

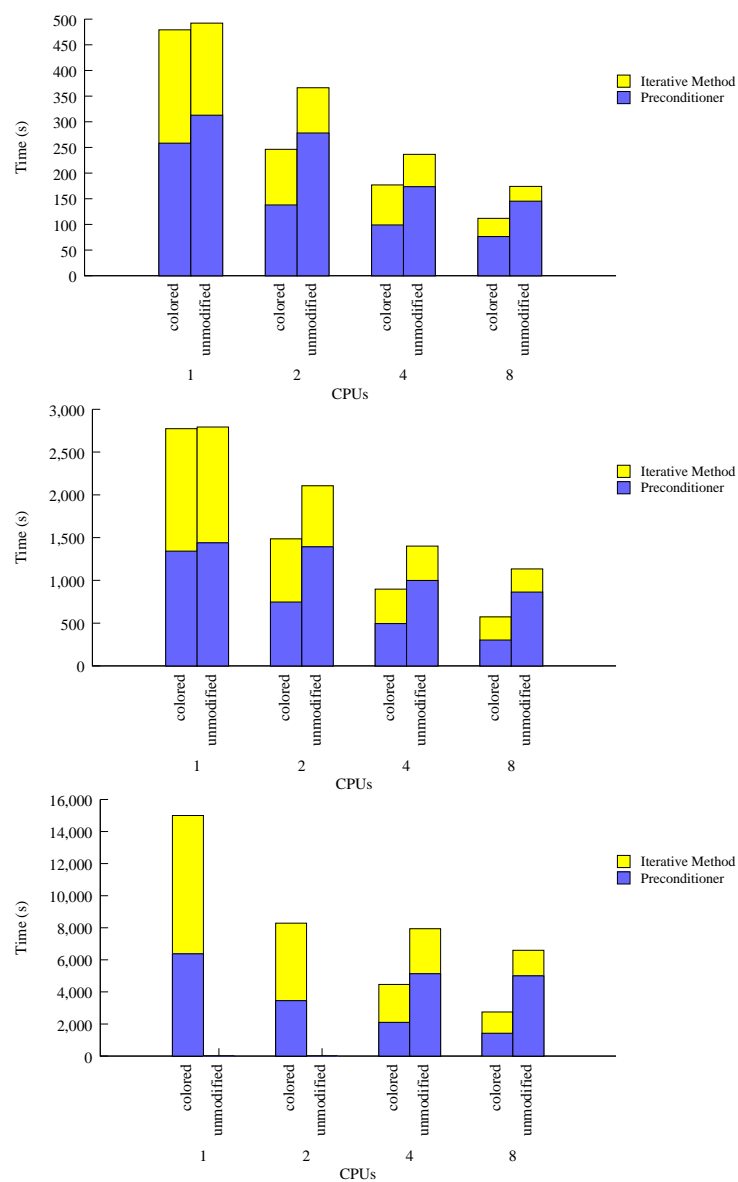
The bar plots show the performance of the two linear solver codes. The left bar, titled “colored” indicates the time taken by the coloring-based triangular solver

### *Chapter 3. Graph Coloring and Parallel Approaches*

used as a preconditioner with the BiCGSTAB iterative method from HyPRE. The right bar, titled “unmodified” indicates unmodified HyPRE code, with the ILU(0) preconditioner called Euclid, and the BiCGSTAB iterative method. We report the final solution time, which includes the time spent in the preconditioning solves. In each case, the times shown are averages over four runs. For the spherical grid with 65 million points, we were not able to run HyPRE with 1 or 2 processors, and thus we only have data for 4 and 8 processors.

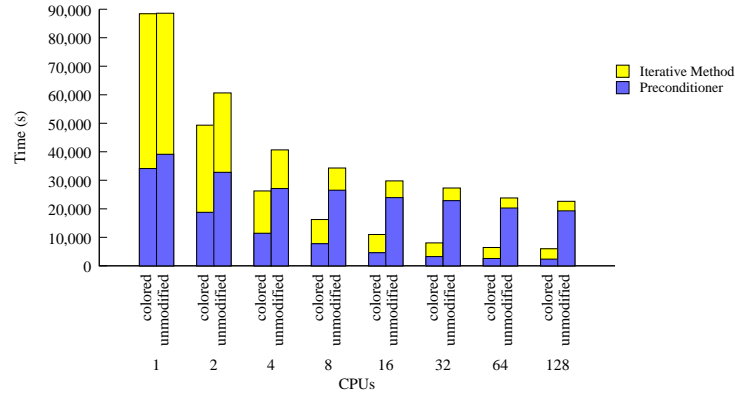
We also ran the linear solver on a grid containing 263 million grid points and figure 3.10 shows the performance of the linear solve on this grid. The code was run on an SGI Altix containing two nodes. Each node contained 512 processors, and each processor could address 4 Gigabytes of main memory. The results from this run indicates that the triangular solver parallelizes well. On 128 processors, the triangular solver was 14.5 times faster than on a single processor. Also, its performance was 8 times better than unmodified HyPRE on the same problem with the same number of processors.

The performance improves as the number of CPUs is increased. The serial performance of both methods is almost identical. As the number of CPUs is increased, the triangular solver performs much better. As the grid size increases, the coloring based triangular solver becomes more attractive. At 4 million grid points, the coloring-based solver is only 1.5 times as fast as the unmodified HyPRE



**Figure 3.9:** Performance of linear solver with circular grid containing 4M (top), 16M (middle) and 64M (bottom) grid points

### Chapter 3. Graph Coloring and Parallel Approaches



**Figure 3.10:** Performance of linear solver with circular grid containing 163M grid points

CPUs	Preconditioner		Iterative Method	
	Colored	Unmodified	Colored	Unmodified
1	258.45	312.79	220.83	179.54
2	138.06	278.24	108.28	88.03
4	98.88	173.38	78.11	63.08
8	76.28	145.39	35.63	28.77

**Table 3.8:** Time in seconds for solving a linear system with the 4M mesh

CPUs	Preconditioner		Iterative Method	
	Colored	Unmodified	Colored	Unmodified
1	1342.01	1438.49	1431.20	1354.31
2	747.35	1392.48	738.28	713.77
4	494.77	998.87	402.44	402.44
8	301.99	863.83	270.57	270.57

**Table 3.9:** Time in seconds for solving a linear system with the 16M mesh

CPUs	Preconditioner		Iterative Method	
	Colored	Unmodified	Colored	Unmodified
1	6374.89	-	8629.18	-
2	3454.34	-	4832.82	-
4	2105.44	5132.29	2363.85	2807.58
8	1421.63	5003.24	1328.39	1591.35

**Table 3.10:** Time in seconds for solving a linear system with the 64M mesh

CPUs	Preconditioner		Iterative Method	
	Colored	Unmodified	Colored	Unmodified
1	34,115.01	39,126.33	54,310.53	49,476.61
2	18,753.54	32,796.72	30,587.98	27,865.49
4	11,417.28	27,110.75	14,865.48	13,542.37
8	7,753.73	26,509.06	8,483.47	7,795.02
16	4,591.64	23,927.82	6,382.38	5,864.44
32	3,216.44	22,832.38	4,842.74	4,449.74
64	2,582.32	20,283.34	3,829.28	3,518.53
128	2,346.40	19,283.28	3,639.23	3,343.90

**Table 3.11:** Time in seconds for solving a linear system with the 263M mesh

code. For the mesh containing 65 million grid points, the coloring triangular solver is 2.4 times as fast as the unmodified HyPRE code on 8 CPUs.

### 3.5.1 Scaling Observations

The scaling of the coloring-based linear system solver is shown in figure 3.11.

We fit functions of time  $t$ , and number of grid points  $n$  to the scaling graphs:

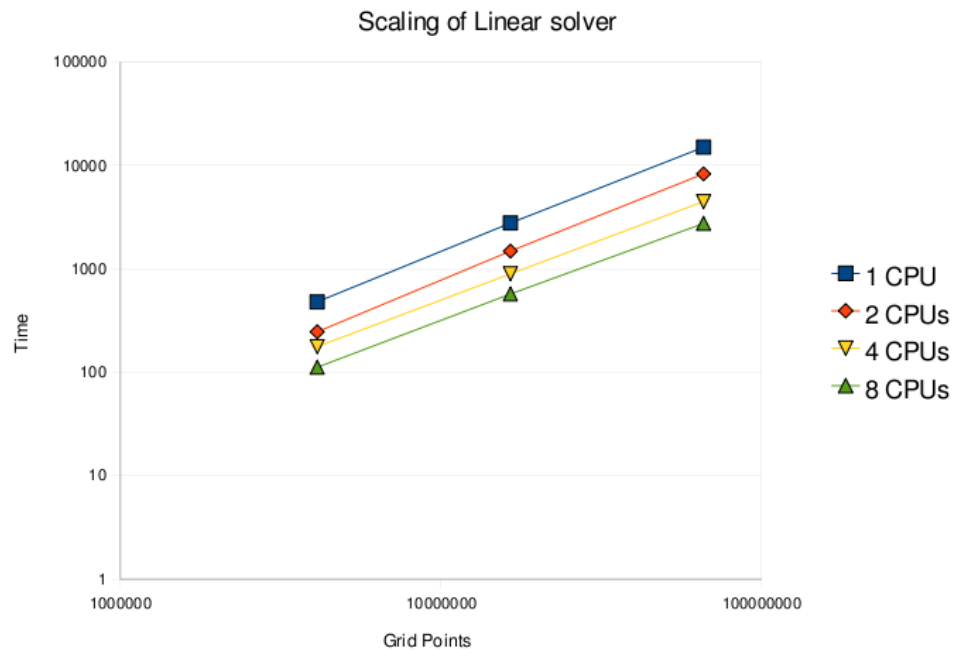
- 1 CPU:  $t = .0047 \times n^{1.22}$
- 2 CPUs:  $t = .0031 \times n^{1.24}$
- 4 CPUs:  $t = .0045 \times n^{1.16}$
- 8 CPUs:  $t = .0044 \times n^{1.13}$

In the equation  $t = c \times n^s$ , the coefficients  $c$  and  $s$  give us valuable clues about the performance of the parallel code.

### *Chapter 3. Graph Coloring and Parallel Approaches*

The value  $s$  marks the slope of the line in the log-log plot of figure 3.11. This coefficient dominates when the grid sizes get large. A smaller value of  $s$  indicates better scaling with increasing grid size. As we can see from the equations above, the constant  $s$  is 1.13 at 8 CPUs and 1.22 at 1 CPU. This suggests that as the grid size increases, the performance of the 8 CPU code will increasingly get better than the 1 CPU code.

The value  $c$  marks the  $y$ -intercept on the log-log plot, which indicates the work that is performed when  $n = 0$ . This coefficient dominates the time calculation when the grid sizes are small. Again, a smaller value indicates lower computation time and hence better performance. Looking again at the coefficients above, at 8 CPUs the value of  $s$  is 0.0044, which is less than that at 1 CPU, which is 0.0047. This indicates that at a higher number of processors, the time taken by the code is lower, even for smaller grids.



**Figure 3.11:** Scaling of the coloring-based linear system solver with circular grid



## Chapter 4

# Alternate Approaches

*It is scientific only to say what is more likely and what less likely, and not to be proving all the time the possible and impossible.*

Richard Feynman (1918-1988) [32]

There are many options when solving linear systems, and our work deals with a parallel version of only one of them. A valid argument could be made for investigating this wide variety of options to ensure that there is no method that performs significantly better than the combination of BiCGSTAB and ILU(0) that has been described in the previous chapters. In this chapter, we take a look at some of the many alternative approaches that could be tried, and evaluate their suitability to the task. Along the way, we isolate issues specific to the matrices under study that foil a naïve implementation of these techniques. This chapter could be used as a starting point for future research, and helps identify various areas where alternate techniques might be promising.

## Chapter 4. *Alternate Approaches*

This chapter looks at possible approaches from two perspectives: methods that deal largely with the graph structure, or methods that work on the numerical structure of the final matrix.

### 4.1 Graph Methods

The most promising solver for structured grids is multigrid. This is a well-studied method, with impressive theoretical convergence bounds. Since our grids are generated in a hierarchical way, multigrid sounds promising. Support graph preconditioners are also a promising class of preconditioners that rely on the graph structure of the problem to come up with effective preconditioners, that are largely independent of the numeric properties of the original matrix [18]. Since both these are largely graph based methods, we discuss them in the first section.

#### 4.1.1 Multigrid

Multigrid [111] is an efficient method for solving linear systems of space discretizations. The observation that leads to multigrid is that smoothers (like the Jacobi fixed point iteration) smooth out high frequency errors very efficiently and quickly. The long convergence time of these smoothers is due to their slow smoothing of low frequency errors. In order to accelerate the smoothing, multigrid

## *Chapter 4. Alternate Approaches*

methods perform the smoothing on various hierarchies of the grid. At a coarser grid, the smooth component of the error turns oscillatory, and can again be effectively reduced by a simple smoother. By translating the solution between these grids, we can effectively reduce all the components of the error using a computationally inexpensive smoother. This method has been demonstrated to work very well in a wide variety of physical problems. Multigrid methods require a good understanding of the graph structure, and the physical problem. An exception to this is algebraic multigrid, which requires just a positive semidefinite matrix and no extra knowledge of the grid structure.

The three main elements of multigrid are the coarsening operator, the refinement operator, and a smoother. For our initial tests we tried Jacobi and Gauss-Seidel smoothers, and also relaxed versions of these. As a first test, we generated spherical grids at various levels of refinement. Due to the octree structure, every point in a grid at  $i$  level of refinement is also a point in the grid at every level of refinement higher than  $i$ . This leads to an obvious coarsening and refinement strategy: generate the spherical grid at various levels of refinement at the outset. At level 0 the grid consists of the original cuboid, and at level 1 it consists of eight nodes, going all the way to the desired level of refinement. The grid generator, called GeomOct, was also modified to return the mapping of the points from one level of refinement to the next. Chapter 5 describes GeomOct, our grid genera-

## Chapter 4. Alternate Approaches

tor. Since all the points in the coarse grid are present in the fine grid, we copy their values from the coarse grid to the fine grid. The newly created points in the fine grid are given a value of 0. The coarsening operator is the transpose of this refinement operator, where values of the fine points are essentially thrown away, and values of points that continue being in the coarse grid are retained from the fine grid. This corresponds to the simple method of using a rectangular identity matrix as an coarsening and refinement operator. A more realistic approach: that of averaging neighboring values when refining was also tried, and its results are also presented later in this section. The performance of this identity-based code was rather dismal. We measure the discrete  $L^2$  norm of the vector  $e$ , where  $e$  is the error vector at the various levels of the grids. At the finest grid,  $e$  is defined as  $Ax - b$ , and at the coarse levels, it is defined as  $Au - f$ , where  $Au = f$  is the residual equation at that coarse grid. This is the preferred method of measuring multigrid errors [111, page 58]. We expect a working multigrid solver to reduce this error gradually to machine epsilon.

Table 4.1 shows the error at the various grid sizes when performing a multigrid v-cycle. For comparison, 3190 iterations of Jacobi on the finest grid produces an error of  $4.11 \times 10^{-2}$ . This comparison shows that the Jacobi iteration itself is more effective in reducing the error than a combination of the Jacobi iteration and translating values through the coarser grids using the multigrid v-cycle. The

Grid size	Iterations	Error
63811	1595	$9.87 \times 10^0$
15947	398	$3.91 \times 10^1$
4083	102	$3.38 \times 10^2$
1099	27	$4.45 \times 10^3$
4083	102	$2.20 \times 10^3$
15947	398	$9.06 \times 10^1$
63811	1595	$6.90 \times 10^1$

**Table 4.1:** Discrete  $L^2$  norm error for multigrid v-cycle with 63k spherical grid using the identity matrix as an operator

table also lists the number of iterations of Jacobi at every depth. The number of iterations of the Jacobi method used in this example are very large, and used to demonstrate the poor error reduction at the coarser levels. In a reasonable multigrid code, a few rounds of smoothing (much less than 10) are usually sufficient. In this experiment, a small number of rounds of smoothing usually led to the error diverging widely. Table 4.2 shows the results from running the same computation using a small number of rounds of smoothing at every stage. At every grid, ten iterations of Jacobi were run to smooth the errors. For comparison, running twenty iterations of Jacobi on the finest grid produces an error of  $4.52 \times 10^3$ .

Since the naïve application of multigrid was not beneficial, we investigated a more involved application. For this, we used the octree structure to help interpolate neighboring nodes when refining the grid. Since the grids are constructed using octrees, every grid point that is newly created during refinements of the grid is guaranteed to have neighbors along some co-ordinate axis, or plane. These

Grid size	Error
256395	$2.08 \times 10^{03}$
63811	$3.51 \times 10^{06}$
15947	$9.96 \times 10^{08}$
4083	$1.07 \times 10^{11}$
1099	$2.08 \times 10^{12}$
4083	$5.85 \times 10^{13}$
15947	$1.12 \times 10^{14}$
63811	$2.94 \times 10^{14}$
256395	$8.92 \times 10^{14}$

**Table 4.2:** Discrete  $L^2$  norm error for multigrid v-cycle with 256k spherical grid using the identity matrix as an operator

neighbors were used to interpolate values. We also implemented full weighting with a fraction of 0.125. The restriction operator  $R$  was  $0.125 \cdot E^T$ , where  $E$  is the elongation operator. Table 4.3 lists the performance of this experiment on a spherical grid containing 256k grid points where the coarsest grid contained 16k grid points. The performance is much better than the previous case, though the solution still diverges. Table 4.4 lists the performance of this code on the same grid, where the coarsest grid generated contained 469 grid points. Translating the solution to a coarser grid produced a very divergent solution.

In the previous experiments the matrix operators at the coarse grids were provided by GeomOct. An alternative technique called Galerkin coarse grid approximation uses matrix operators that are themselves generated using coarsening operators on fine grids [111]. This is beneficial in some cases, when only the operator for the finest grid is available. We experimented with Galerkin coarse grid

Grid size	Error
256589	$5.34 \times 10^{04}$
64005	$7.43 \times 10^{08}$
16141	$9.09 \times 10^{12}$
64005	$4.34 \times 10^{11}$
256589	$4.68 \times 10^{10}$

**Table 4.3:** Discrete  $L^2$  norm error for multigrid v-cycle with finest grid containing 256k spherical grid and coarsest grid containing 16k points using interpolation operators and full weighting

approximation and the results are presented in Table 4.5. The results were much better than previous results, while remaining divergent. In these experiments, Jacobi was used as a smoother, and ten iterations of Jacobi were run at every grid.

In all these experiments, Dirichlet boundary conditions were used, and the boundary points were left in the calculation, even though their values are explicitly known.

This initial experiment with multigrid suggests that the obvious multigrid technique for these grids does not work. Multigrid research observes [111] that smoothers need to work across the dimension where the errors oscillate. In these grids, the anisotropy is due to both the automatic mesh refinement, where the size of the grid cells can change dramatically, and also because of the variable coefficient  $\rho$ . A fully adaptive octree mesh has the attractive property that grid points can capture a very small interface at a high amount of detail. This also leads

Size(N)	Error
256589	$5.34 \times 10^{04}$
64005	$7.43 \times 10^{08}$
16141	$9.09 \times 10^{12}$
4277	$4.13 \times 10^{18}$
1293	$1.35 \times 10^{31}$
469	$1.57 \times 10^{44}$
1293	$2.10 \times 10^{50}$
4277	$8.39 \times 10^{49}$
16141	$4.81 \times 10^{49}$
64005	$3.52 \times 10^{49}$
256589	$3.14 \times 10^{49}$

**Table 4.4:** Discrete  $L^2$  norm error for multigrid v-cycle with finest grid containing 256k spherical grid and coarsest grid containing 469 points using interpolation operators and full weighting

Size(N)	Error
256589	$5.34 \times 10^{04}$
64005	$9.29 \times 10^{07}$
16141	$1.77 \times 10^{10}$
64005	$6.76 \times 10^{09}$
256589	$5.83 \times 10^{09}$

**Table 4.5:** Discrete  $L^2$  norm error for multigrid v-cycle with 256k spherical grid using interpolation operators, full weighting and Galerkin coarse grid approximation



#### *Chapter 4. Alternate Approaches*

to very few grid points elsewhere in the domain. Due to this, grid spacing varies widely across any given plane or any given co-ordinate axis. A smoother that works along the interface of interest might work well, but such an implementation would be quite involved. In our experimentation with multigrid, we were unable to obtain any convergence.

Recent research by Zhang [114, 115] suggests that multigrid methods can be successfully applied to similar problems. In this work, multilevel grids are used to successfully discretize the domain, and inter-grid transfer operators successfully transfer the components of errors between various grids. Research by Haber and Heldmann [47] demonstrates a working multigrid method for an octree grid, though their work deals with Maxwell’s equation rather than Poisson’s equation.

Algebraic multigrid is also an option when the grid structure is unknown. We tried algebraic multigrid with these grids, using BoomerAMG from the HyPRE package [31]. Since AMG requires a symmetric, positive definite matrix, we used  $\max(A, A^T)$ , which is symmetric and positive definite. We were unable to obtain convergence using this specific symmetric approximation.

Our experimentation with multigrid leads us to conclude that multigrid solvers for these grids are difficult to obtain. We can isolate the following areas that need work to develop a good multigrid code for these matrices.

- The obvious prolongation and restriction operators might work with a better smoother, though we cannot suggest a possible smoother based on our experimentation. Further, robust and efficient smoothers are difficult to obtain in 3D, as mentioned in “The Introduction to Multigrid Methods” [111, page 92],
- Fully adaptive meshes on level set problems have the advantage of capturing a small interface in the domain with great accuracy. This produces a large number of grid points in a small region of interest in the domain, and a small number of grid points elsewhere. Such a discretization has large anisotropies which make multigrid implementations tricky.
- The graph structure is unsymmetric, which makes a straightforward application of algebraic multigrid difficult. The matrix obtained is neither symmetric nor positive definite.

#### 4.1.2 Support Graph Preconditioning

Another promising approach is that of support theory [10]. Support graph preconditioners are constructed from the graph structure of the matrix, and generate preconditioners that are, in general, sparser than the original graph. Previous experimental results [18] have shown that support graph preconditioners work well

#### *Chapter 4. Alternate Approaches*

on the model problem with discontinuous coefficients. These experimental results suggest that problems on which ILU stagnates might be ones on which support graph preconditioners perform better.

A difficulty in the direct application of support graph preconditioners is that the support theory results only hold for symmetric, positive definite matrices. We do not have such matrices, and thus we try to approximate the generation of support graph preconditioners by mimicking their construction on symmetric, positive definite matrices.

In symmetric, positive definite matrices, the matrix  $A$  is decomposed as  $A = V \times V^T$ , where  $V$  is a matrix containing the individual edges of the original graph. Edge  $(i, j)$  leads to a nonzero element in the  $i$  and the  $j$  column of  $V$ , and the nonzero elements have opposite signs. This cannot be done for matrices under consideration here. However, we extend this result to M-matrices with non-negative row sums which might not be symmetric. Let  $V_1$  and  $V_2$  be matrices with equal number of rows, and  $[V_1 \ V_2]$  represent a matrix with columns composed of columns from matrix  $V_1$  followed by columns from matrix  $V_2$ . The lemma below shows a constructive proof of how any M-matrix  $A$  with non-negative row sum can be decomposed as  $A = [V_1 \ V_2] \times [V_1 \ V_3]^T$ , where the edges in the first part,  $V_1$ , correspond to the symmetric edges. The second set of edges are the non-symmetric edges, and these are responsible for the asymmetry in the original matrix.

## Chapter 4. Alternate Approaches

First, we write the M-matrix  $A$  as a sum of three components: the symmetric positive definite component  $A_s$ , the unsymmetric component  $A_u$ , and the diagonal component  $A_d$  such that:  $A = A_s + A_u + A_d$ . For symmetric positive definite matrices,  $A_u$  and  $A_d$  are zero. For M-matrices,  $A_d$  consists of diagonal elements that are positive or zero.

We use the standard support theory tools to write the matrix  $A_s$  as  $A_s = V \times V^T$  [10]. To write the original matrix  $A$  as  $A = [V_1 \ V_2] \times [V_1 \ V_3]^T$ , we need to demonstrate that we can split  $A_u$  in terms of some matrices  $U_1$ ,  $U_2$  and  $U_3$  such that  $[U_1 \ U_2] \times [U_1 \ U_3]^T$ .

First, we show how matrices  $U_1$ ,  $U_2$  and  $U_3$  can be constructed so that the off-diagonal elements of  $[U_1 \ U_2] \times [U_1 \ U_3]^T$  are identical to  $A_u$ . For M-matrices with non-negative row sums,  $A_u$  consists of rows with negative off-diagonals alone. We show how such a matrix  $A_u$  can be split.

**Lemma 4.1.** *Given a matrix  $B$  with negative off-diagonal elements, we can construct matrices  $U_1$ ,  $U_2$  and  $U_3$  such that the matrix  $[U_1 \ U_2] \times [U_1 \ U_3]^T$  has the same off-diagonal elements as  $U$ .*

*Proof.* Let  $B$  be a matrix containing negative off-diagonal elements.

We demonstrate how a single negative off-diagonal element  $B_{ij}$  can be written in terms of columns vectors  $a_{ij}$ ,  $b_{ij}$  and  $c_{ij}$  such that  $[a_{ij} \ b_{ij}] \times [a_{ij} \ c_{ij}]^T$  produces

Chapter 4. Alternate Approaches

the off-diagonal element  $B_{ij}$  in its original location, and a diagonal element  $-B_{ij}$  in position  $B_{ii}$ .

Let  $\alpha = \sqrt{-B_{ij}/2}$ .

Construct vector  $a_{ij}$  with  $-\alpha$  in the  $j$ th location and  $\alpha$  in the  $i$ th location.

Construct vector  $b_{ij}$  with  $\alpha$  in both the  $j$ th location and the  $i$ th location.

Let vector  $c_{ij}$  be identical to  $a_{ij}$ .

Now the product  $[a_{ij} \ b_{ij}] \times [a_{ij} \ c_{ij}]^T$  produces the off-diagonal element  $B_{ij}$  in its original location, and a diagonal element  $-B_{ij}$  in position  $B_{ii}$ .

This can be done for every nonzero in the row of the matrix  $B_i$ . We collect all the columns vectors  $a$  to form  $U_1$ ,  $b$  to form  $c$  and  $w_3$  to form  $U_3$ , preserving their order so that the  $p$ th column vector in all three matrices  $U_1$ ,  $U_2$  and  $U_3$  corresponds to the contribution from the  $p$ th nonzero element of  $B$ .

□

To demonstrate this splitting, we show how a single nonzero can be split. Here  $B$  is a matrix with a single off-diagonal nonzero element and column vectors  $a_1$ ,  $b_1$  and  $c_1$  produced from the previous result.

$$B = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -2 & 0 & 0 \end{pmatrix}, a_1 = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}, b_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, c_1 = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$$

## Chapter 4. Alternate Approaches

The product  $[a_1 \ b_1] \times [a_1 \ c_1]^T$  produces

$$\begin{pmatrix} -1 & 1 \\ 0 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -2 & 0 & 2 \end{pmatrix}$$

The previous splitting produces spurious positive elements in the diagonal elements which were not present in the original matrix  $B$ . For M-matrices, the row sum in the diagonal is non-negative and thus these diagonal elements are present in the matrix  $A_d$ . The row sum might be strictly positive, and in that case, we need the following lemma to handle the diagonal values alone.

**Lemma 4.2.** *A matrix  $D$  with a positive diagonal elements can be written as  $D = U \times U^T$ , where  $U$  is an  $n \times n$  matrix.*

*Proof.* Let  $A$  be a matrix with a positive diagonal elements  $d_{ii}$ . Then, the matrix  $U$  with diagonal elements  $\sqrt{d_{ii}}$  in the  $i$ th locations achieves the result.  $\square$

Finally, we list the entire decomposition.

**Theorem 4.1.** *Any M-matrix  $A$ , with non-negative row sums can be written as  $A = [V_1 \ V_2] \times [V_1 \ V_3]^T$  where  $V_1$ ,  $V_2$ , and  $V_3$  are matrices with at most two nonzero elements per column.*

#### Chapter 4. Alternate Approaches

*Proof.* Let  $A$  be an M-matrix with non-negative row sums. Decompose  $A$  into its symmetric, unsymmetric and diagonal parts  $A_s$ ,  $A_u$  and  $A_d$ , such that  $A_s$  is symmetric positive definite and  $A = A_s + A_u + A_d$ .

Generate matrix  $V$  such that  $A_s = V \times V^T$  using support theory.

Generate matrices  $U_1$ ,  $U_2$  and  $U_3$  using lemma 4.1 on  $A_u$ .

Let  $D = A - V \times V^T - ([U_1 \ U_2] \times [U_1 \ U_3]^T)$ .

Since  $A$  is an M-matrix with non-negative row sums,  $D$  is a diagonal matrix with non-negative elements.

Construct  $U$  using lemma 4.2 on  $D$ .

Construct matrices  $V_1 = [V \ U \ U_1]$ ,  $V_2 = U_2$  and  $V_3 = U_3$ .

This ensures the matrix  $A = [V_1 \ V_2] \times [V_1 \ V_3]^T$ . □

Using this construction, we can use support theory tools to find a good approximation for  $V_1 \times V_1^T$ . Using the above construction, we can construct symmetric, positive definite matrices from M-matrices. This should allow the evaluation of support graph tools in areas where they have not been tried before. Support graph preconditioners can be generated in parallel using the parallel rooted spanning tree approach proposed by Cong and Bader [20].

## 4.2 Numerical Methods

The previous section described our efforts at using multigrid and support graph tools. Both these techniques can be used if the matrices generated are symmetric and positive definite. The matrices obtained from the second order discretization of the Poisson problem are neither symmetric nor positive definite. However, as noted in section 2.1.1, the unsymmetry in the matrices is small. In this section, we try to develop means to construct approximations of these matrices that are symmetric and positive definite. These approximations could then be used in preconditioners that work well on positive definite, symmetric matrices.

The goal is to evaluate different methods to generate a symmetric matrix  $S$  from the original matrix  $A$ . We experiment with a variety of constructions that can help approximate the original matrix by a symmetric, positive definite matrix. The symmetric part of the matrix  $A$ ,  $A_S = (A + A^T)/2$  is sparse but not positive definite. This construction would work if the original matrix was positive definite, but that is not the case for the matrices under consideration. Another possible construction involves the polar decomposition, and is explained in the work by Higham [53]. The work describes a construction for a symmetric positive semidefinite matrix that is the closest (in terms of Frobenius norm) to the original matrix  $A$ . In this case,  $A_H = (A + A^T)/2$  and  $A_H = U \times P$  is the polar



#### Chapter 4. Alternate Approaches

decomposition of the symmetric part of the matrix  $A$ . The symmetric positive definite matrix is then constructed as  $S$ ,  $S = (A_H + P)/2$ .

These are the various possibilities under study.

- $S_1 = \max(A, A^T)$ . This construction is positive definite and sparse for the matrices under study. It has the advantage of being easy to implement, and computationally efficient.
- $S_2 = V_1 \times V_1^T$  where  $A = [V_1 \ V_2] \times [V_1 \ V_3]^T$ .  $V_1, V_2, V_3$  are all matrices with at most two non-zero elements per column. We have previously shown that this construction is always possible. We now look at it purely from the perspective of the approximation it provides.

In order to evaluate how close these symmetric matrices approximate the original matrix  $A$ , we look at a few numerical metrics that aim to quantify the approximation. We look at how closely the eigenvalues of the symmetric matrices approximate the eigenvalues of the original matrix. We look at the condition number of  $S_i^{-1} \cdot A$ , to see how close the inverse of the symmetric matrix is to the inverse of  $A$  itself. We look at the eigenvalues of the symmetric matrix and check how they approximate the eigenvalues of the original matrix. Finally, we use the symmetric matrix to precondition the original matrix and see how well it works as a preconditioner.

Chapter 4. Alternate Approaches

$n$	$\ \sigma(S_1) - \sigma(A)\ /\ \sigma(A)\ $	$\ \sigma(S_2) - \sigma(A)\ /\ \sigma(A)\ $
469	$1.18 \times 10^{-2}$	$7.53 \times 10^{-2}$
1293	$1.49 \times 10^{-2}$	$2.51 \times 10^{-2}$
4277	$1.40 \times 10^{-2}$	$1.91 \times 10^{-2}$

**Table 4.6:** Relative difference between eigenvalues of the symmetric matrices

Grid size	$\kappa(S_1^{-1}A)$	$\kappa(S_2^{-1}A)$	$\kappa(A)$
469	$9.67 \times 10^0$	$1.45 \times 10^3$	$4.87 \times 10^2$
1293	$1.22 \times 10^3$	$5.73 \times 10^5$	$6.46 \times 10^3$
4277	$2.59 \times 10^3$	$2.05 \times 10^7$	$9.20 \times 10^4$

**Table 4.7:** Condition numbers of  $S_i^{-1}A$

For all the tests below, we use a spherical grid that was generated using the partial differential equation 2.1.

Table 4.6 shows the relative difference between the eigenvalues of the symmetric approximations, and that of the original matrix  $A$ . We report  $\|\sigma(S_i) - \sigma(A)\|/\|\sigma(A)\|$ , where  $\sigma$  denotes the vector containing the sorted eigenvalues of the matrix.  $\sigma_1$  is the eigenvalue with the smallest magnitude and  $\sigma_n$ , the largest. If the relative difference of this norm is small, then the symmetric matrix forms a good approximation to the original matrix.

Grid size	$S_1$	$S_2$	ILU(0)
469	68	99	64
1293	79	122	74
4277	93	133	86
16141	132	163	102
64005	203	254	163

**Table 4.8:** Iterations of GMRES with  $S_i$  used to generate IC(0), and ILU(0) factors

## Chapter 4. Alternate Approaches

Table 4.7 shows the comparison of the condition numbers of  $S_i^{-1} \cdot A$  for the different approximations. As we can see,  $S_1$  forms a closer approximation to the original matrix  $A$ . The condition number of  $A^{-1}A$  is 1, and we are interested in a symmetric matrix that keeps this condition number small. The condition number of the matrix  $A$  is shown for comparison.

Table 4.8 shows the comparison of the preconditioning ability of  $S_i^{-1} \cdot A$  for the different approximations. We use two methods to evaluate this. First, we generate Cholesky factors from the symmetric matrices. These are then solved with the GMRES iterative method. The method is set to restart every 50 iterations, and to an accuracy of  $10^{-10}$ , and with maximum number of iterations set to 1000. For a comparison, we generate incomplete LU factors, and use them to precondition the same iterative method. We are interested in reducing the number of iterations of GMRES. The table shows that  $S_1$  again forms a good approximation, and the number of iterations is 25-30% more than that for the ILU factors. The storage required for the Cholesky factors is significantly less, since we only store one triangular factor rather than two. In memory constrained environments, it might be beneficial to use the incomplete Cholesky factors generated with the  $S_1$  approximation rather than incomplete LU factors.

In our matrices, the off-diagonal values are negative, thus the construction of  $S_1$  involves dropping nonzero values that are not symmetric in structure. This

#### *Chapter 4. Alternate Approaches*

results in the nonzero structure of  $S_1$  being a subset of the nonzero structure of the original matrix  $A$ . The nonzero structure of incomplete no-fill Cholesky factors generated with  $S_1$  is identical to the nonzero structure of  $S_1$ . Hence, the nonzero structure of the incomplete no-fill Cholesky factors generated with  $S_1$  is a subset of the nonzero structure of  $A$ . Our coloring, described in section 3.3, is designed for the nonzero structure of the matrix  $A$  and will be a valid coloring for  $S_1$  and its incomplete no-fill Cholesky factors. Thus, the graph coloring described earlier can still be applied in the use of incomplete no-fill Cholesky factors generated with the symmetric approximation  $S_1$ .

# Chapter 5

## Grid Generator

*What I cannot create, I do not understand.*

Richard Feynman (1918-1988)<sup>1</sup> [50].

In this chapter, we describe our grid generator, called GeomOct, which is a robust, flexible tool to generate large octree grids. GeomOct generates general octree grids, over a 3D domain. The tool is written with a view towards flexibility. It can either be used as a library, or as a stand-alone program, making it attractive to researchers burdened with generating octree grids. Information can be stored at either the cell centers, or the cell corners, to allow for the simulation of a wide variety of physical processes. Once the octree grid is generated, helper methods are provided to iterate over all the octree cells, or all the corners.

For most common uses, a single program is included, where the user provides a signed distance function specifying the boundary. This program is easy to use and extend: all a casual user needs to provide is a signed distance function. The

---

<sup>1</sup>On his blackboard at time of death in 1988.

## *Chapter 5. Grid Generator*

target audience is a researcher evaluating whether automatically refined octree grids will be suitable to their problem.

The code is written in C++, extensively using the C++ template mechanism, and the standard template library [86]. For the basic functionality, no libraries are needed. Extra libraries help with the unit testing framework and the visualization but are not required for the functioning of the code. It also helps to have a matrix analysis environment (such as Matlab [70], Octave [27], or R [22, 37, 57]) to process the final grids.

In this chapter, we discuss the grid generator from the perspective of a user of the library. The library is provided in source form and is designed to be modified. Towards this goal, we provide programmers with a fairly detailed guide of the library internals in appendix [A](#).

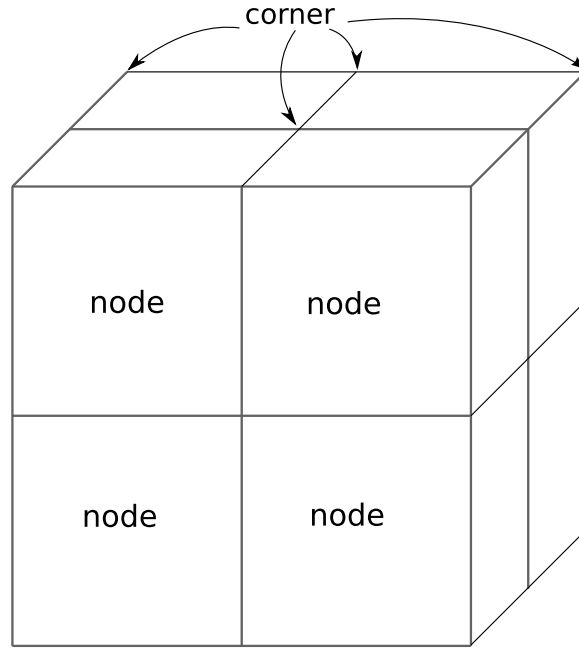
### **5.1 Overview**

GeomOct is designed to be extensible and easy to use. One of the prime motivations was the lack of a good publicly available octree grid generator that created fully automatically refined grids. We aim to keep the library as simple as possible, to ensure that it is flexible. We begin with an overview of the generator.

## *Chapter 5. Grid Generator*

The two central classes in the code are `OctreeNode` and `Corner`. The `OctreeNode` class keeps information relevant to the tree structure, through which the grids are generated. This tree structure is used repeatedly, whether to find cells, to perform automatic refinement, or when calculating the coefficients needed by the finite difference calculation. The `OctreeNode` objects are stored in a tree data structure, where every `OctreeNode` object has either eight children or none. In case the `OctreeNode` has no children, it is called a leaf node, since it is a leaf in the octree. The initial domain is represented by one `OctreeNode` called the root, and this is returned when constructing an octree. The eight children of any `OctreeNode` are called siblings of each other.

Since finite difference calculations can store data in the cell (the `OctreeNode` center) or the cell corners, each `OctreeNode` has associated data in structures called a `Corner`. `Corner` and `OctreeNode` objects share an important relationship. A single `OctreeNode` has information about the corners of that particular octree node. While octree nodes are unique, due to their tree structure, their corners are not. For example, consider a unit volume split into eight children. Each of the corners of the parent are inherited by exactly one child, and thus the parent and the children share corners. The storage of corners is abstracted away in a class called `CornerList`, which provides the functionality to retrieve corners or query for specific corners. Every octree is associated with a `CornerKeeper` object,



**Figure 5.1:** Single octree showing nodes and corners

which creates and deletes corners, and makes sure that a corner is created only once. Thus there is a hierarchical relationship between `OctreeNode` objects but not between corner objects.

Figure 5.1 shows an octree node, which is split into eight nodes, and the corresponding corners. The corner in the center of the octrees is shared by all four octrees. Specifically, the corner in the center of the domain is shared by all eight siblings. Since corners are shared among octree nodes, any data stored at them is visible to all octree nodes that have that corner.

Despite a lack of a hierarchical arrangement, the corner objects are organized to make it easy to perform finite difference schemes. In most finite difference



## *Chapter 5. Grid Generator*

codes, it is important to find adjacent corners. Given a specific corner, the physical system is formed by evaluating values at the neighbors of the corners in all three coordinate directions. Since this is such a common operation, the GeomOct library data structures are designed to make the operation quick and efficient. In order to do this, every corner is a part of a doubly linked list, with two links for each coordinate axis. Each corner is connected to its neighbor in the positive X direction, and its neighbor in the negative X direction. Such connections exist for every coordinate direction, at every corner. This makes it easy to write finite difference codes, since the finite difference approximations are all created by accessing the physical values at the corners that neighbor the corner of interest. The way to do this in the library is to call `somecorner.getNext(axis)` and `somecorner.getPrevious(axis)` for the desired coordinate axis. Creating corners and inserting them at specific locations preserves these doubly linked lists, making it easy to traverse a particular coordinate direction, or to find all the neighbors of any node.

For creating octrees and performing the finite difference calculation, the library makes extensive use of functors [86]. A functor is a generalization of a function. In C++, a functor is an object that implements the `operator()`, and thus can be called as a function. Being an object, a functor can be created at run-time, deleted, or passed as an argument to a method. Also, a functor can contain arbitrary

## Chapter 5. Grid Generator

state information, and can perform initialization when created and clean-ups when deleted. Functors are extensively used in this library, due to their flexibility over functions. They also make the code much easier to debug by separating the functionality.

GeomOct allows for two distinct ways to create octrees. A class called **SplitShape** is provided, which takes as input a shape in rectangular coordinates. To create a spherical octree, the programmer must define the sphere, (Say  $f(x, y, z) = x^2 + y^2 + z^2 - 4$ ), and then create a **SplitShape** object with this function. Using this object, the root node of the octree can be split. Since this is the most common technique for creating an octree, it is designed to be as simple as possible. The other possibility is to pass an functor of the type **SplittingCriterion**. In this case, the object must define the `operator()`. The call `octreeNode.splitOnDecider()` takes a **SplittingCriterion** functor as an argument, and evaluates the functor at every octree node, to choose whether the node is to be split. If the functor returns true, then the cell is split, otherwise it is not. This is a completely flexible way to split an octree. Both these ensure that octrees can be created without the internal knowledge of the octree data structures.

The library hides much of the internal representation of the octrees and the corners. This is both to reduce the complexity for implementing octree based

## *Chapter 5. Grid Generator*

finite difference codes, and to prevent deep changes to the internal data structures. Instead, helper classes are provided to make it easy to change the internal data for the entire octree at once. For the `OctreeNode` class, the method is `octreeNode.performOnAll()`. This accepts an `OctreeNodeOperation` object. This object can perform any modifications that are required at the specified node. Using this, octree nodes can be iterated on without knowledge of the underlying data structures.

The internal representation of the octree and corner data structure is not made visible. Since the library is available in source form, willing programmers can peek inside the data structures and change them, but this is not the suggested mode of operation. Rather, there are a variety of classes that act on `OctreeNode` and `Corner` objects. For a full description of the classes, we point the reader to appendix [A](#) where the complete class documentation is listed. The code includes many extra features, like unit testing of the various components, to verify that the library is built correctly, hooks for visualization tools, and sample code that performs finite difference calculations for many grids.

## 5.2 Stand-alone program

For generating a finite difference octree grid, we provide a stand-alone program which calls the GeomOct library to create a grid. This stand-alone program is called GridGen. This section describes the working of this stand-alone program.

In 3D, the volume is automatically discretized using octrees. The grid generator requires a signed distance function  $\phi$ , which indicates the distance from the fluid interface.  $\phi(x)$  is negative when  $x$  is inside the fluid, positive outside, and zero on the interface itself. GridGen refines the volume around the interface to the maximum possible depth allowed by the user. While this formulation is motivated by the level set method approach, it is not limited to the level set method. Complex grids, for example a hull of ship, could be modelled similarly by considering the hull-water interface, where the value of  $\phi(x)$  is negative for all the points inside the hull. In either case, the refinement is carried out to surround the interface with fine octree nodes. Elsewhere in the domain, there is no refinement. This ensures that all the computational effort is spent in the region of interest. We do not impose a graded grid, where the volume of neighboring octree node differs only by a constant factor.

Once the volume is discretized, the user can choose to run iterators over all the octree nodes or all the corners, in a straightforward manner. One such iterator

assigns pressure and density values to all the node corners. Another computes the matrix from a finite difference formulation. A third could print out the dependencies between the corners, for a multigrid-like method. The sequence of iterators depends on the user's choice, and many iterators could be run on the same grid, once it is constructed.

## 5.3 Extensibility

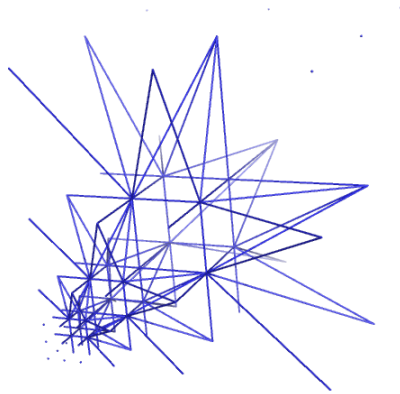
While we have demonstrated the utility of the GeomOct library, and the accompanying program, the library and program are meant to be extensible.

Since the octree is just a simple data object, many octrees can be kept simultaneously in memory. This could be to model different parts of the same computation, or to compare two octrees. The number of octrees, or the amount of refinement, is limited only by the memory on the machine. We have successfully generated grids of up to 65 million grid points with the current code. Figure [5.3](#) shows the corners of a spherical grid. The grid has been cut away to reveal the spherical shell to the right. One can notice that the density of points is the greatest near the spherical shell, and sparse when we move away from the shell. The shell is the region of interest in most computations, and this method devotes the maximum computational elements (and resources) to the region of interest.

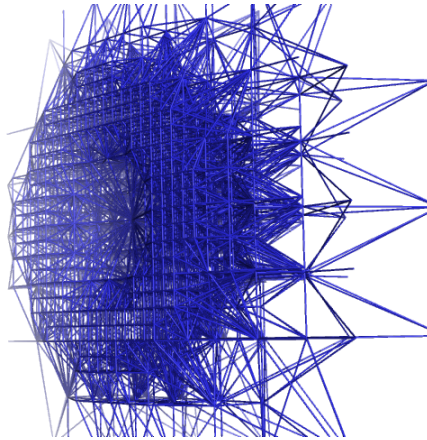
## *Chapter 5. Grid Generator*

Our tools have been designed to be of use in both matrix analysis and visualization. As a result, we generate matrices that can be imported into common matrix analysis tools like Matlab, Octave, or R. We also generate grids that can be imported into freely available visualization software, to allow researchers to generate graphically appealing images and movies. This tool and all supporting code is available through our website [21] under the GNU general public license [36].

In order to quickly generate visually appealing images and movies, a special utility (`PyMolPrinter.py`) is provided to take the output of the `GeomOct`. It is a simple Python script that produces Protein Data Bank (PDB) files as output. These can be visualized using the freely available PyMol program. The choice of PyMol [25] as a visualization tool was due to its robust implementation, free availability, and the ability to handle large files. In addition to generating static images, PyMol allows for interactive visualization of the data. In PyMol, the grid can be zoomed into, rotated, the grid point numbering can be turned enabled or disabled and the dependencies between grid points can be shown or hidden. Further, movies can be created containing visualizations of the grid. These can be very helpful since 3D grids become difficult to visualize on a 2D surface of a computer screen (or a sheet of paper). In order to completely understand their structure, it is valuable to be able to interact with the grid and view it from various angles. PyMol is available on a wide variety of platforms.



**Figure 5.2:** Octree grid refined around origin



**Figure 5.3:** One half of a spherical octree grid

In the next section, we demonstrate the second order accuracy of the solution on grids produced by GeomOct.

## 5.4 Convergence

Octree grids can be generated for a wide variety of surfaces that define the interface. Along with this variability, we can also choose the partial differential equation to be solved over this domain. We show that the finite difference method is accurate, to demonstrate the correctness of the grid generator.

We generate grids using the second order method described in the work by Min et al. [76]. We demonstrate that the grids generated using this method are indeed second order accurate. For this example, we choose two types of grids:

- Grid created by focussed refinement: the grid is formed by refining the cell containing the origin. Figure 5.2 shows the focussed grid.
- Spherical grid: the grid is formed by positioning a sphere inside the domain of computation. Refinement is carried out to capture the surface of this sphere. Figure 5.3 shows a piece of the spherical grid.

Along with these two grids, we can introduce anisotropies by considering either a variable coefficient Poisson equation or a constant coefficient equation. We consider two possibilities for the partial differential equations,



## Chapter 5. Grid Generator

- A constant coefficient equation:  $u = e^{-(x+y+z)}$ ,  $\rho = 1$ , and force function  $f = e^{-(x+y+z)}$ .
- A variable coefficient equation:  $u = e^{-(x^2+y^2+z^2)}$ ,  $\rho = \sin(x+y+z) + 2$ , and force function  $f = -6u\rho + 4u\rho(x^2 + y^2 + z^2) - 2u(x+y+z) * \cos(x+y+z)$ .

Both these equations are chosen because of their use in previous research [76]. Tables 5.1 and 5.3 show the errors for the focussed grid with increasing levels of refinement, while tables 5.2 and 5.4 show the errors in the spherical grid with increasing levels of refinement. As the grid spacing decreases, the accuracy of the finite difference method increases, and the convergence is second order, as expected.

We describe the method for verifying the accuracy, since the number of grid points at the various levels differs between tables 2.1 and 5.2. When generating octree grids for table 2.1, we generate a grid so that the smallest cells border the interface. Thus, at height 4, the sides of the cells near the interface measure  $\frac{L}{2^4}$ , where  $L$  is the length of the side of the original volume. No other constraint is imposed, and this is a kind of a bottom up approach to grid construction.

By contrast, when we measure the accuracy, we construct a grid at a specified level of refinement, say level 3 for table 5.2. Then, for the next level, **every** octree node is refined. This certainly puts the smallest octree cells near the interface,

Grid Points	Height	Error ( $ \hat{u} - u $ )
122	6	0.0069
615	7	0.0019
3749	8	4.2652e-04
25833	9	1.1178e-04

**Table 5.1:** Convergence of focussed grid with  $u = e^{-(x+y+z)}$

Grid Points	Height	Error ( $ \hat{u} - u $ )
469	3	0.0114
2977	4	0.0028
21025	5	6.7038e-04
157633	6	1.6233e-04

**Table 5.2:** Convergence of spherical grid with  $u = e^{-(x+y+z)}$

but also creates smaller cells in regions far away from the interface. This is why tables 2.1 and 5.2 agree on the number of cells at height 3, where the grids are identical. However, at height 4, table 2.1 has fewer points than table 5.2, and this disparity increases as the refinement proceeds.

Grid Points	Height	Error ( $ \hat{u} - u $ )
122	6	0.0332
615	7	0.0085
3749	8	0.0020
25833	9	4.9409e-04

**Table 5.3:** Convergence of focussed grid with  $u = e^{-(x^2+y^2+z^2)}$

Grid Points	Height	Error ( $ \hat{u} - u $ )
469	3	0.0527
2977	4	0.0108
21025	5	0.0029
157633	6	7.3246e-04

**Table 5.4:** Convergence of spherical grid with  $u = e^{-(x^2+y^2+z^2)}$ 

## 5.5 Modifications for Coloring

Our grid generator constructs octree grids, which can be numbered in a variety of ways. For our coloring, we wish to generate grids that have been reordered with the coloring scheme mentioned in section 3.3.

Let us highlight the modifications required to the GeomOct code to reorder the matrix with the coloring described in section 3.3. All this code is contained in file `ColorNumberer.h` in the source directory `Octree`. The code is all contained within a single class `ColorNumberer`, which is a functor of type `OctreeNodeOperation`. In order to color the node, we pass a `ColorNumberer` object as follows: `root.performOnAll (numbererObject)`. Then, the `numberer` is called once on every octree node, in breadth-first order.

In order to assign colors, the greedy graph coloring algorithm is used. The colors are ordered, and we try to assign the minimum possible color to every grid point. Every color also has a linked list, which links all the grid points of that color, in the order in which they were colored. Once this is set up, the

## Chapter 5. Grid Generator

`ColorNumberer` object itself does very little. It identifies the data dependencies, and then assigns each grid point the minimum color that is possible. This is done by assigning the color number to a variable, and also adding the grid point to a list for that particular color. Once the numberer has run on all the nodes, at the end of the `performOnAll` call, the lists have been populated, but the actual numbering has not been performed. In order to perform the numbering, we call the `assignNumbers` method of the `ColorNumberer` object. This method iterates over the lists for each color in order, assigning increasing numbers to the grid points in that list. This simple operation numbers all the grid points. An outline of the pseudo-code of the greedy algorithm is given below.

The computational complexity of coloring the nodes is  $\Theta(n)$ , since each grid point is traversed exactly once by the octree breadth-first routine. Searching for data dependencies is a constant time operation due to the bidirectional coordinate linked lists that join all grid points to their immediate neighbors. Once the grid points are colored, numbering them is also a  $\Theta(n)$  operation, since we simply iterate over all the lists, which together contain  $n$  grid points. Thus, the overall complexity of the coloring and numbering is  $\Theta(n)$ .

## **5.6 Focus on Experimentation**

In the analysis of octree grids, we have found the grid generator to be a valuable prototyping tool. New ideas can be tried and evaluated quickly. We have successfully used the grid generator to generate reordered matrices, grid colorings, and even elongation and restriction operators for multigrid.

We expect our grid generator to be of use to researchers who plan to experiment with octree grids. It is easy to change the numbering of the cells, to generate grids with a variety of finite difference or finite volume methods. We hope that this tool helps users prototype octree grids for their applications. Once a suitable method is found, an efficient grid generator can be designed for the target architecture and the physical problem.

The source code of the GeomOct library is comprehensively documented, to ease the task of programmers who might need to extend it. While the provided stand-alone program demonstrates one possible application of the library, we expect programmers to utilize it for their specific applications. This requires a detailed understanding of the internal structure of the GeomOct library, which is provided in [appendix A](#).

# Chapter 6

## Conclusion

*If that's the world's smartest man, God help us.*

Lucille Feynman, referring to her son, Richard Feynman <sup>1</sup>

We have successfully demonstrated a method to speed up the linear system solution of octree grids for unsymmetric, second-order accurate, finite difference discretizations of the Poisson's equation. We have demonstrated that for these grids, incomplete no-fill factorizations form the most robust preconditioners, and their use with iterative methods like BiCGSTAB yields good results. We have also demonstrated that within the iterative method, the preconditioned solve forms the dominant computation, and is the bottleneck to the efficient solution of these grids on parallel computers.

To eliminate this bottleneck, we propose a novel numbering for the grid points. This numbering is independent of the Poisson equation, and is relatively straightforward to implement. It can be easily incorporated in the grid generator, since it

---

<sup>1</sup>In response to Omni magazine naming Richard Feynman the world's smartest man [43].

## *Chapter 6. Conclusion*

requires only local information. Further, this numbering, coupled with the well-known greedy graph coloring algorithm, gives an upper bound on the chromatic number of the octree grids.

Using this coloring, we have successfully implemented a parallel triangular solver that exhibits good speedups. In section 3.5 we have demonstrated that the parallel linear solver parallelizes well, and its performance on 128 processors is 14.5 times as fast as the run on a single processor. This offers a speedup of 8 times over the previous method.

We believe that the simplicity of the octree grid generation, coupled with good parallel performance of ILU(0) should make the octree grids a more attractive solution to researchers. Towards this aim, we make our grid generator available to the research community. We hope its flexibility and strength in prototyping will encourage the use of octree meshes in new applications.

# Bibliography

- [1] P. Allen. Putting UML to work: Strategies and techniques. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 33–43, 1998.
- [2] F. L. Alvarado, A. Pothen, and R. S. Schreiber. Highly parallel sparse triangular solution. *Graph Theory and Sparse Matrix Computation*, 56:141–158, 1993.
- [3] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal of Scientific Computing*, 14(2):446–460, 1993.
- [4] P. Amestoy, T. A. Davis, , and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.
- [5] K. Appel and W. Haken. Every planar map is four colorable. *Illinois Journal of Mathematics*, 21:429–567, September 1977.
- [6] O. Axelsson. Bounds of eigenvalues of preconditioned matrices. *SIAM Journal on Matrix Analysis and Applications*, 13:847–862, 1992.
- [7] D. Bai and A. Brandt. Local mesh refinement multilevel techniques. *SIAM Journal on Scientific and Statistical Computing*, 8:109–134, 1987.
- [8] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [9] F. Bassi, F. Grasso, and M. Savini. Solution of the compressible Navier-Stokes equations by using embedded adaptive meshes. *Lecture Notes in Physics*, 264:113–119, 1986.



## Bibliography

- [10] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo. Support-graph preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 27:930–951, 2006.
- [11] A. Brandt. Multilevel adaptive solutions to boundary value problems. *Mathematics and Computation*, 31:333–390, 1977.
- [12] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, views and models of UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 93–108. Physica-Verlag, Heidelberg, 1998.
- [13] W. Briggs. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, 1987.
- [14] N. Buleev. A numerical method for the solution of two-dimensional and three-dimensional equations of diffusion. *Sbornik: Mathematics*, 51:227–238, 1960.
- [15] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. Technical report, Numerical Analysis Group, Oxford University Computing laboratory, 1995.
- [16] X. Castellani. An overview of the version 1.1 of the UML defined with charts of concepts. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML’98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 13–24, 1998.
- [17] T. C. Chan and H. A. van der Vorst. Approximate and incomplete factorizations. Technical Report 871, Department of Mathematics, University of Utrecht, The Netherlands, 1994.
- [18] D. Chen and S. Toledo. Vaidya’s preconditioners: Implementation and experimental study. *Electronic Transactions on Numerical Analysis*, 16:30–49, 2003.
- [19] A. J. Chorin. A numerical method for solving incompressible viscous flows. *Journal of Computational Physics*, 1967.
- [20] G. Cong and D. A. Bader. The Euler tour technique and parallel rooted spanning tree. In *33rd International Conference on Parallel Processing*, pages 448–457, August 2004.

## Bibliography

- [21] Combinatorial Scientific Computing lab website. <http://gauss.cs.ucsb.edu/>.
- [22] P. Dalgaard. *Introductory Statistics with R*. Springer, 2002. ISBN 0-387-95475-9.
- [23] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. *American Institute for Aeronautics and Astronautics*, 32:489–496, 1994.
- [24] T. A. Davis, J. R. Gilbert, S. Larimore, and E. Ng. A column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30(3):353–376, 2004.
- [25] W. L. DeLano and S. Bromberg. *The PyMOL User’s Manual*. DeLano Scientific LLC, San Carlos, CA, USA., 2004.
- [26] R. Diestel. *Graph Theory, Third Edition*. Springer-Verlag, July 2005.
- [27] J. W. Eaton. *Octave User’s Guide*, 2007.
- [28] M. Enns, W. F. Tinney, and F. L. Alvarado. Sparse-matrix inverse factors. *Ieee Transactions On Power Systems*, 5(2):466–473, 1990.
- [29] D. J. Evans. The use of pre-conditioning in iterative methods for solving linear equations with symmetric positive definite matrices. *Journal of the Institute of Mathematics and Applications*, pages 295–314, 1968.
- [30] D. J. Evans and R. C. Dunbar. The parallel solution of triangular systems of equations. *IEEE Transactions on Computers*, C-32:201–204, 1983.
- [31] R. D. Falgout, J. E. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. *Numerical Solution of Partial Differential Equations on Parallel Computers*, 51:267–294, 2006.
- [32] R. P. Feynman. *The Character of Physical Law*. Modern Library, 1965.
- [33] R. P. Feynman. What is Science? *The Physics Teacher*, 7(6), 1969.
- [34] R. P. Feynman. *The Pleasure of Finding Things Out*. Perseus Books Group, 2000.

## Bibliography

- [35] R. P. Feynman and M. Feynman. *Perfectly Reasonable Deviations from the Beaten Track : The Letters of Richard P. Feynman*. Basic Books, 2005.
- [36] Free Software Foundation. GNU general public license. <http://www.gnu.org/copyleft/gpl.html>, June 2007.
- [37] Free Software Foundation. *R User's Guide*, 2007.
- [38] K. Gallivan and E. W. Ng, editors. *Parallel Algorithms for Matrix Computations*. SIAM, 1990.
- [39] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, New York, 1979.
- [40] F. Gibou. A level set approach for the numerical simulation of dendritic growth. *Journal of Scientific Computing*, 13, December 2003.
- [41] F. Gibou and R. Fedkiw. A fourth order accurate discretization for the Laplace and heat equations on arbitrary domains, with applications to the Stefan problem. *Journal of Computational Physics*, 202:577–601, 2005.
- [42] F. Gibou, R. Fedkiw, L.-T. Cheng, and M. Kang. A second order accurate symmetric discretization of the Poisson equation on irregular domains. *Journal of Computational Physics*, 176:202–227, 2002.
- [43] J. Gleick. *Genius: The Life and Science of Richard Feynman*. Vintage, 1992.
- [44] G. Golub and V. Loan. *Matrix Computations*. Johns Hopkins University Press, 1996.
- [45] A. Gupta and V. Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 74, New York, NY, USA, 1995. ACM.
- [46] I. Gustafsson. A class of first-order factorization methods. *BIT Numerical Mathematics*, 18:142–156, 1978.
- [47] E. Haber and S. Heldmann. An octree multigrid method for quasi-static Maxwell's equations with highly discontinuous coefficients. *Journal of Computational Physics*, 223(2):783–796, 2007.

## Bibliography

- [48] W. Hackbusch and U. Trottenberg. *Multigrid Methods*. Springer-Verlag, 1982.
- [49] W. W. Hager. Condition estimates. *SIAM Journal on Scientific and Statistical Computing*, 5:311–316, 1984.
- [50] S. W. Hawking. *The Universe in a Nutshell*. Bantam, 2001.
- [51] M. T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9(3):558–588, 1988.
- [52] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 1952.
- [53] N. J. Higham. Matrix nearness problems and applications. *Applications of Matrix Theory*, pages 1–27, 1989.
- [54] N. J. Higham. Stability of parallel triangular system solvers. *SIAM Journal of Scientific Computing*, 16(2):400–413, 1995.
- [55] N. J. Higham and F. Tisseur. A block algorithm for matrix 1-norm estimation with an application to 1-norm pseudospectra. *SIAM Journal of Matrix Analysis and Applications*, 21:1185–1201, 2000.
- [56] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal of Scientific Computing*, 22(6):2194–2215, 2000.
- [57] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, Sept 1996.
- [58] D. Jespersen. A time-accurate multiple-grid algorithm. *American Institute for Aeronautics and Astronautics*, 1985.
- [59] H. Johansen and P. Colella. A Cartesian grid embedded boundary method for Poisson’s equation on irregular domains. *Journal of Computational Physics*, 147(1):60–85, 1998.
- [60] G. Johnson and J. Swisshelm. Multigrid for parallel-processing supercomputers. In *Proceedings of the Third Copper Mountain Conference on Multigrid Methods*, April 1987.

## Bibliography

- [61] G. Johnson and J. Swisshelm. Multiple-grid solution of the three-dimensional Euler and Navier-Stokes equations. *Lecture Notes in Physics*, 218:286–290, 1987.
- [62] G. M. Johnson. Multiple-grid convergence acceleration of viscous and inviscid flow computations. *Journal of Applied Mathematics and Computation*, 18(3-4):375–398, 1983.
- [63] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.
- [64] M. T. Jones and P. E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20(5):753–773, 1994.
- [65] S. Kaniel. Estimates for some computational techniques in linear algebra. *Mathematics of Computation*, pages 369–378, 1966.
- [66] C. Koeck and J. Chattot. Computation of three-dimensional vortex flows past wings using the Euler equations and a multiple-grid scheme. *Lecture Notes in Physics*, 218:308–313, 1985.
- [67] D. J. Kuck. Parallel processing of ordinary programs. *Advances in Computers*, 15:119–179, 1976.
- [68] G. Li and T. F. Coleman. A parallel triangular solver for a hypercube multiprocessor. Technical Report TR86-787, Cornell University Ithaca, NY, USA, 1986.
- [69] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. *SIGGRAPH*, pages 457–462, 2004.
- [70] Mathworks Inc. *MATLAB User’s Guide*, 2007. version 2007a.
- [71] A. Mayo. The fast solution of Poisson’s and the biharmonic equations on irregular regions. *SIAM Journal of Numerical Analysis*, 21:285–299, 1984.
- [72] S. McCormick. *Multigrid Methods*. Society for Industrial and Applied Mathematics, 1987.
- [73] P. Mccorquodale, P. Colella, D. P. Grote, and J.-L. Vay. A node-centered local refinement algorithm for Poisson’s equation in complex geometries. *Journal of Computational Physics*, 201(1):34–60, November 2004.

## Bibliography

- [74] J. Meijerink and H. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31:148–162, 1977.
- [75] C.-H. Min and F. Gibou. A second order accurate projection method for the incompressible Navier-Stokes equations on non-graded adaptive grids. *Journal of Computational Physics*, 219(2):912–929, 2006.
- [76] C.-H. Min, F. Gibou, and H. Ceniceros. A supra-convergent finite difference scheme for the variable coefficient Poisson equation on fully adaptive grids. *Journal of Computational Physics*, 202:577–601, 2006.
- [77] R.-H. Ni. A multiple-grid scheme for solving the Euler equations. *American Institute for Aeronautics and Astronautics*, 20:1565–1571, 1982.
- [78] Object Management Group. Unified Modeling Language: Infrastructure and superstructure specification. <http://www.omg.org/spec/UML/2.1.2/>, 2007.
- [79] T. Oliphant. An implicit numerical method for solving two-dimensional time-dependent diffusion problems. *Quarterly Applied Mathematics*, 19:221–229, 1961.
- [80] T. Oliphant. An extrapolation process for solving linear systems. *Quarterly Applied Mathematics*, 20:257–267, 1962.
- [81] S. J. Osher and R. P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.
- [82] S. J. Owen. A survey of unstructured mesh generation technology. In *Proceedings, 7th International Meshing Roundtable, Sandia National Lab*, pages 239–267, October 1998.
- [83] J. Papac. Personal Communication, May 2008.
- [84] M. S. Perucchio, Renato and A. Kela. Automatic mesh generation from solid models based on recursive spatial decompositions. *International Journal For Numerical Methods In Engineering*, 28:2469–2501, 1989.
- [85] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 30, New York, NY, USA, 1999. ACM.

## Bibliography

- [86] P. Plauger, M. Lee, D. Musser, A. A. Stepanov, and A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.
- [87] S. Popinet. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *Journal of Computational Physics*, 190:572–600, 2003.
- [88] A. Pothen and F. L. Alvarado. A fast reordering algorithm for parallel sparse triangular solution. *SIAM Journal on Scientific and Statistical Computing*, 13(2):645–653, 1992.
- [89] C. H. Romine and J. M. Ortega. Parallel solution of triangular systems of equations. *Parallel Computing*, 6:109–114, 1988.
- [90] U. Rüde. Fully adaptive multigrid methods. *SIAM Journal of Numerical Analysis*, 30:230–248, 1993.
- [91] O. S. and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.
- [92] Y. Saad. ILUT: a dual threshold incomplete *LU* factorization. *Numerical linear algebra with applications*, 1(4):387–402, 1994.
- [93] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.
- [94] R. Schreiber and W. P. Tang. Vectorizing the conjugate gradient method. In *Proceedings of the Symposium on CYBER 205 Applications*, 1982.
- [95] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University, 1999.
- [96] M. Shephard, F. Guerinoni, J. Flaherty, R. Ludwig, and P. Baehmann. Finite octree mesh generation for three-dimensional flow analysis. *Computers and Structures*, 20(1):31–39, 1985.
- [97] M. S. Shephard and M. K. Georges. Three dimensional mesh generation by finite octree technique. *International Journal For Numerical Methods In Engineering*, 32:704–749, 1991.

## Bibliography

- [98] J. Swisshelm and G. Johnson. Development of a aerodynamics algorithm for parallel-processing supercomputers. *Computational Fluid Dynamics*, pages 689–698, 1988.
- [99] J. Swisshelm, G. Johnson, and S. Kumar. Parallel computation of Euler and Navier-Stokes flows. *Journal of Applied Mathematics and Computation*, 19:321–331, 1986.
- [100] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing 92*, 1992.
- [101] The Centos Project. Centos website. <http://www.centos.org/>, April 2008.
- [102] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.
- [103] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.
- [104] Tyan Computer Corporation. *Tyan Thunder M4985 CPU card installation manual*.
- [105] Tyan Computer Corporation. *Tyan Thunder n4250QE S4985G3NR motherboard installation manual*.
- [106] A. C. N. van Duin. Optimal DAG partitioning for partially inverting triangular systems. Technical report, Universiteit Leiden, Netherlands, 1997.
- [107] A. C. N. van Duin. Sparse triangular system partitioning. Technical report, Universiteit Leiden, Netherlands, 1997.
- [108] R. Varga. Factorization and normalized iterative methods. In *Boundary problems in differential equations*, pages 121–142. University of Wisconsin Press, 1960.
- [109] R. S. Varga. *Matrix Iterative Analysis*. Prentice Hall, Englewood Cliffs, 1962.
- [110] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series*, 16:521–530, Jan. 2005.



## Bibliography

- [111] P. Wesseling. *An Introduction to Multigrid Methods*. John Wiley and Sons, 1992.
- [112] M. A. Yerry and M. S. Shephard. Three dimensional mesh generation by modified octree technique. *International Journal For Numerical Methods In Engineering*, 20:1965–1990, 1984.
- [113] M. A. Yerry and M. S. Shephard. Trends in engineering software and hardware: Automatic mesh generation for three-dimensional solids. *International Journal For Numerical Methods In Engineering*, 32:704–749, 1991.
- [114] J. Zhang. Acceleration of five-point red-black Gauss-Seidel in multigrid for Poisson equation. *Applied Mathematics and Computation*, 80(1):73–93, 1996.
- [115] J. Zhang. Accelerated multigrid high accuracy solution of the convection-diffusion equations with high reynolds number, 1997.

# Appendices

# Appendix A

## GeomOct Reference

The GeomOct library is intended to be used as a quick prototyping tool for octree meshes. While its primary goal is to generate octree meshes, it is capable of providing much more. As an example, it has been successfully used to generate multigrid prolongation and restriction operators, compute graph colorings, and verify specific finite difference methods. This is possible due to the flexibility of the library and its modular design.

We are providing this library to the research community to evaluate the use of octree grids in new areas. We encourage users to utilize this library for their specific applications. The clean modular nature and the use of templates makes it easy to tailor GeomOct for novel applications. To aid the adoption of GeomOct, we offer a complete documentation of the library for authors of finite difference codes. This appendix contains comprehensive documentation of all the classes comprising the GeomOct library. All public members and public methods are exhaustively documented in this chapter.

This documentation is arranged in sections, where each class of GeomOct is explained in sufficient detail to make the task of library programmers easy. Each section has complete detail about the relation between classes, and inheritance information. To aid in the understanding of class relationships, complete relationship and inheritance diagrams are provided in standard UML notation [1, 12, 16, 78].

### A.1 Installation

The library may be obtained from the Combinatorial Scientific Computing lab website [21]. It is available as a gzipped GNU tar archive. The latest version at the time of publication is 0.80. Despite the sub-1.0 version numbering, the code is stable and reliable. The source package ships with a variety of tests which

## Appendix A. *GeomOct* Reference

ensure that the library was successfully built and is working correctly on the target hardware. Installation is very easy due to the use of GNU autoconf and automake. The `configure` script included with the source package tunes the library to the specific C++ compiler and machine architecture.

Here are the steps involved in unpacking and installing the software. In this example, `$` denotes the shell prompt.

```
$ tar -zxvf octree-0.80.tar.gz
$ cd octree-0.80
$ ./configure
  <...long output containing tuning results...>
$ make
```

## A.2 Detailed Reference

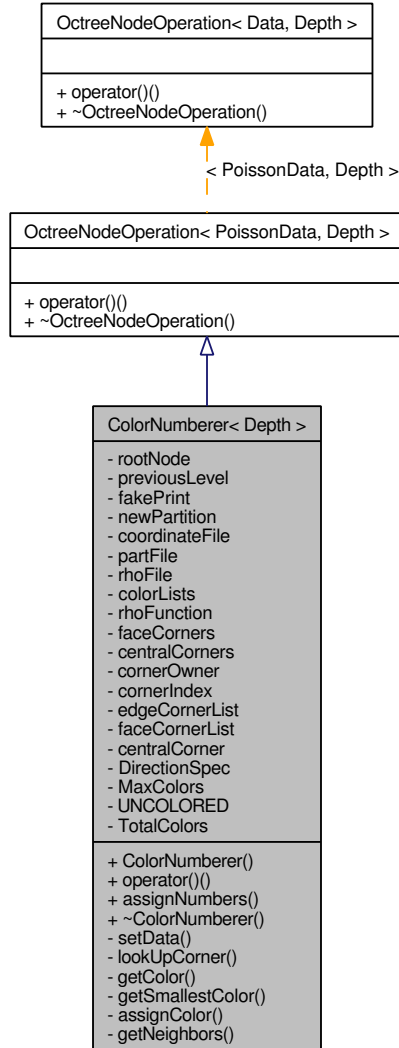
Complete technical detail about every class is provided here. The information is structured as follows. First, we list the preprocessor directives required to include the class. This is a single line in most cases. Next, we list the inheritance diagram for the class, in standard UML notation [78]. This is followed by the collaboration diagram, again in UML notation. The collaboration diagram is useful in determining the dependencies of a particular class. Due to the highly modular nature of *GeomOct*, a single class may depend on many others for its functionality, and the collaboration diagram helps in visualizing these relationships. The collaboration diagram is followed by a list of the public members of the class. This is followed by a detailed description of the class, which includes helpful suggestions to library users. Finally, constructors and public methods are explained in complete detail.

### A.2.1 `ColorNumberer< Depth >` Class Template Reference

```
#include <ColorNumberer.h>
```

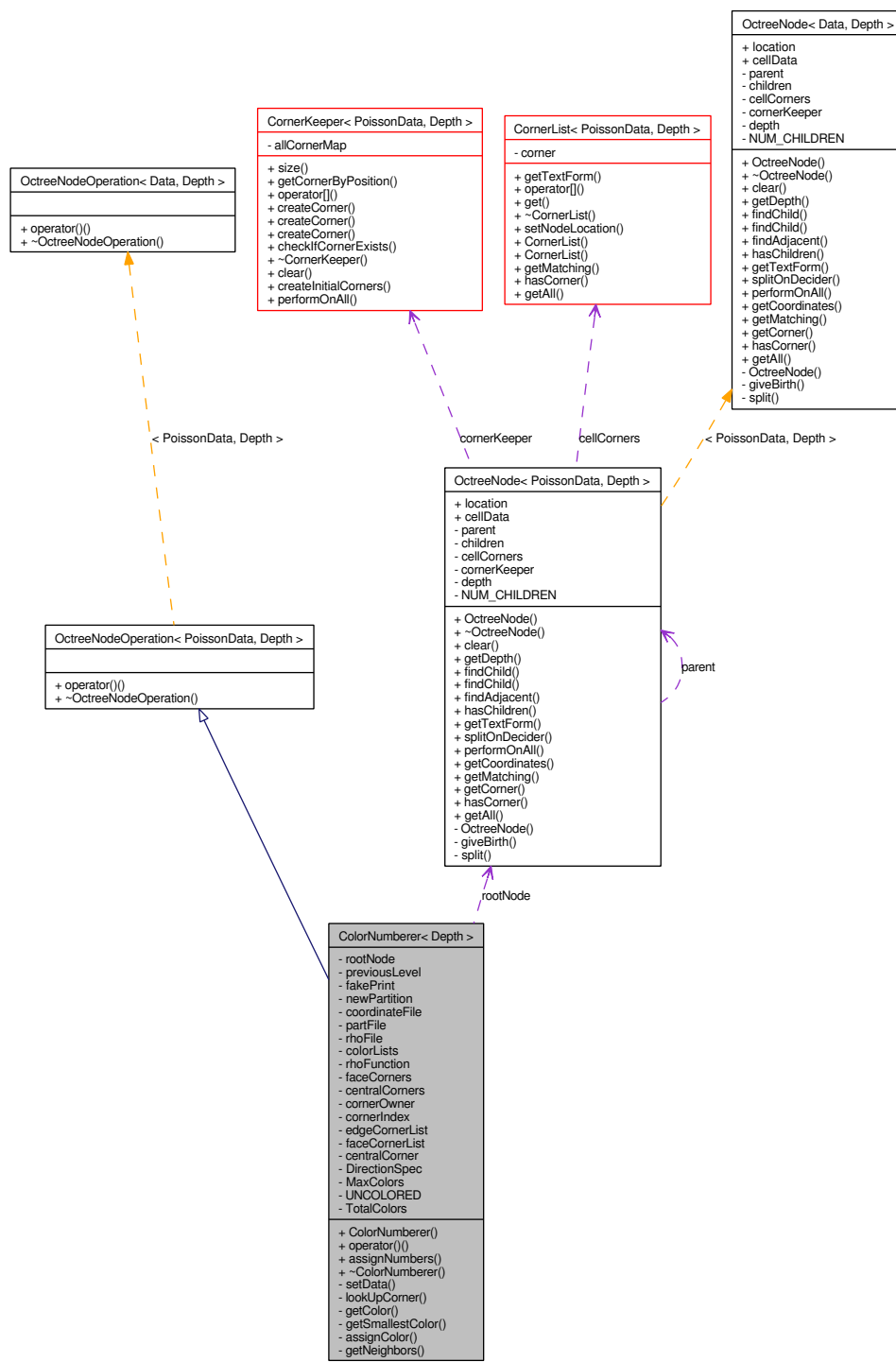
## Appendix A. *GeomOct* Reference

Inheritance diagram for ColorNumberer< Depth >:



## Appendix A. GeomOct Reference

Collaboration diagram for ColorNumberer< Depth >:



## Public Member Functions

- `ColorNumberer` (`OctreeNode`< `PoissonData`, `Depth` > `*octreeRoot`, `double(*rhoFunc)(double, double, double)`, `std::string coordName=""`, `std::string partName=""`, `std::string rhoName=""`)
- virtual `bool operator()` (`OctreeNode`< `PoissonData`, `Depth` > &`node`)
- void `assignNumbers` (void)

## Detailed Description

`template<unsigned short Depth> class ColorNumberer< Depth >`

Color the corners in a specific way that guarantees 11 colors will be sufficient.

The way this works is as follows: the corners are traversed in order of their heights. Instead of immediately assigning the numbers, we assign colors to the corners, and add the corner to a linked list for that color. There is an 11-coloring for these graphs, which means that 11 colors are enough while fewer might be required in practice. After the colors have been assigned, the method `assignNumbers` should be called. This method assigns the numbers based on the colors (and the linked lists) that were created earlier. This is a two-pass algorithm.

After running the functor, run the `assignNumbers()` method to perform the numbering.

This class is not required for the functioning of *GeomOct* and is provided as a reference for writing graph coloring algorithms using *GeomOct*.

## Constructor & Destructor Documentation

`template<unsigned short Depth> ColorNumberer< Depth >::ColorNumberer` (`OctreeNode`< `PoissonData`, `Depth` > `* octreeRoot`, `double(*)`(`double`, `double`, `double`) `rhoFunc`, `std::string coordName = ""`, `std::string partName = ""`, `std::string rhoName = ""`)

Construct a functor using all the information about this octree. This arguments are as follows. The root cell of the octree needs to be provided. The argument `rhoFunc` is a function to compute `rho` at every point in the domain. The remaining arguments specify filenames: `coordName` specifies a file in which coordinate information is written, `partName` specifies the file in which the partitions are written, and `rhoName` specifies the file in which the variable coefficients are written.

## Appendix A. *GeomOct* Reference

Once this is specified, the functor colors all the corners when called as a function.

### Member Function Documentation

```
template<unsigned short Depth> bool ColorNumberer< Depth
>::operator() (OctreeNode< PoissonData, Depth > & node) [virtual]
```

This method performs all the coloring. It requires no input. To generate the node numbering, call the associated [assignNumbers\(\)](#) method after the functor has completed coloring all the corners in the octree.

Implements [OctreeNodeOperation< PoissonData, Depth >](#).

```
template<unsigned short Depth> void ColorNumberer< Depth
>::assignNumbers (void)
```

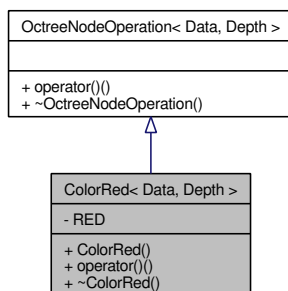
Assign a number to every corner.

This method must be called for the indexing to take place. If this method is not called, then the IDs will be the colors of the nodes rather than the indices.

### A.2.2 ColorRed< Data, Depth > Class Template Reference

```
#include <ColorRed.h>
```

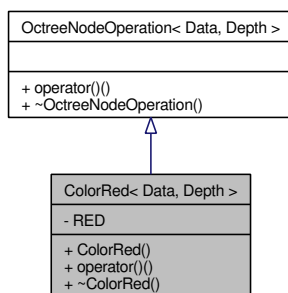
Inheritance diagram for ColorRed< Data, Depth >:





## Appendix A. *GeomOct* Reference

Collaboration diagram for `ColorRed< Data, Depth >`:



### Public Member Functions

- virtual bool `operator()` (`OctreeNode< Data, Depth > &node`)

### Detailed Description

`template<class Data, unsigned short Depth> class ColorRed< Data, Depth >`

This class is used to color all the nodes a specific color. This is required for some graph colorings where all the vertices must start out colored in a specific color at the start of the coloring.

This class is useful, and it also marks a very simple demonstration of the implementation of `OctreeNodeOperation` functors.

### Member Function Documentation

`template<class Data, unsigned short Depth> bool ColorRed< Data, Depth >::operator() (OctreeNode< Data, Depth > & node) [virtual]`

Color all the nodes red.

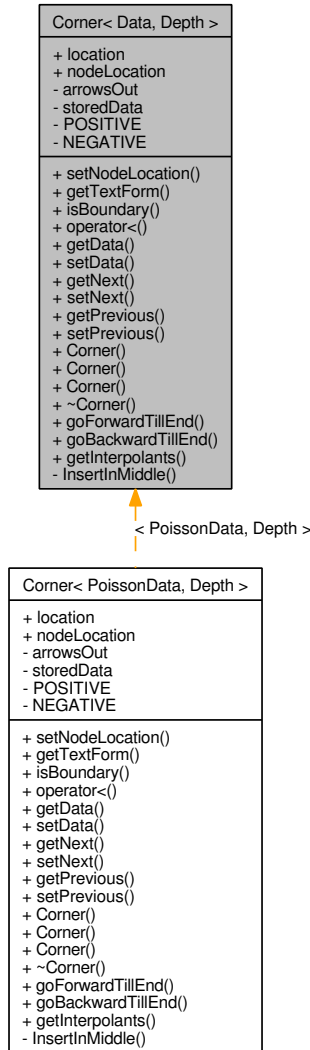
Implements `OctreeNodeOperation< Data, Depth >`.

### A.2.3 `Corner< Data, Depth >` Class Template Reference

```
#include <Corner.h>
```

## Appendix A. *GeomOct* Reference

Inheritance diagram for `Corner< Data, Depth >`:



### Public Member Functions

- void `setNodeLocation` (`NodePosition< Depth > newPosition`)
- `std::string` `getTextForm` (void) const
- bool `isBoundary` (void) const
- bool `operator<` (const `CornerPosition< Depth > &other`) const
- `Data *` `getData` () const
- void `setData` (`Data *newValue`)

## Appendix A. *GeomOct* Reference

- `Corner< Data, Depth > * getNext` (Direction D) const
- void `setNext` (Direction D, `Corner< Data, Depth > *newNext`)
- `Corner< Data, Depth > * getPrevious` (Direction D) const
- void `setPrevious` (Direction D, `Corner< Data, Depth > *newPrevious`)
- `Corner` ()
- `Corner` (const `Corner< Data, Depth > &original`)
- `Corner` (`Corner< Data, Depth > *xPrevious`, `Corner< Data, Depth > *xNext`, `Corner< Data, Depth > *yPrevious`, `Corner< Data, Depth > *yNext`, `Corner< Data, Depth > *zPrevious`, `Corner< Data, Depth > *zNext`, `CornerPosition< Depth > location`)
- `~Corner` ()
- void `goForwardTillEnd` (Direction D, `CornerOperation< Data, Depth > &op`)
- void `goBackwardTillEnd` (Direction D, `CornerOperation< Data, Depth > &op`)

### Static Public Member Functions

- static `std::vector< Corner< Data, Depth > > getInterpolants` (const `std::vector< Corner< Data, Depth > > &matchLists`, const `CornerPosition< Depth > &toCalculate`, short unsigned int direction)

### Public Attributes

- `CornerPosition< Depth > location`
- `NodePosition< Depth > nodeLocation`

### Detailed Description

`template<typename Data, unsigned short Depth> class Corner< Data, Depth >`

The `Corner` class is used to store information about a corner. Corners are also called grid points. The template parameter `Data` is used to store a pointer to any arbitrary data. The `Corner` class does not use the `Data` field at all, so it can be modified as required by the programmer. The argument `Depth` is the same argument given to `OctreeNode`. It is required since we keep the position of a corner as a bitset, and we need to know how many bits to keep.

Corners are linked to each other. This linking helps write finite difference codes in which coefficients depend on values on neighboring grid points.

## Constructor & Destructor Documentation

**template<typename Data, unsigned short Depth> Corner< Data, Depth >::Corner ()**

Create a corner which doesn't point to anything, has no stored data, and has the zero position. This is a way of creating a null node, either for experimentation, or for testing out the qualities of corners.

This method has limited use outside of testing and debugging.

**template<typename Data, unsigned short Depth> Corner< Data, Depth >::Corner (const Corner< Data, Depth > & *original*)**

Create a disconnected corner with just a location and a stored data pointer. No connections are created on this corner either. This is most often used for checking the properties of corners, or for making a copy of a corner's position in a harmless manner.

**template<typename Data, unsigned short Depth> Corner< Data, Depth >::Corner (Corner< Data, Depth > \* *xPrevious*, Corner< Data, Depth > \* *xNext*, Corner< Data, Depth > \* *yPrevious*, Corner< Data, Depth > \* *yNext*, Corner< Data, Depth > \* *zPrevious*, Corner< Data, Depth > \* *zNext*, CornerPosition< Depth > *inputLocation*)**

Create a corner with all the previous and next nodes specified, and the position of this corner.

This constructor gives programmers complete control in inserting grid points.

**template<typename Data, unsigned short Depth> Corner< Data, Depth >::~~Corner ()**

The destructor ensures that the data structures associated with corners stay consistent. For example, when a corner is removed, the previous and next pointers are updated accordingly so that there are no dangling pointers.

This destructor is called automatically, so everything is handled for the programmer. The library does not delete the storedData, since that could be either a pointer to something which is on the stack, or could be a new() object that is being delete()'ed elsewhere. The programmer is responsible for clearing the storedData, especially when using Valgrind.

## Member Function Documentation

**template<typename Data, unsigned short Depth> void Corner< Data, Depth >::setNodeLocation (NodePosition< Depth > *newPosition*)**

Set the location of the octree node containing this corner. *newPosition* is an octree node which is guaranteed to be adjoint to this corner. A single corner may lie in many octree nodes, and we need just one for our calculation. The library needs to keep the location of some adjoint cell. This method allows us to store the position *newPosition* as the location of an adjoint cell for this [Corner](#).

No checking is performed, and so the programmer is expected to set this value correctly.

**template<typename Data, unsigned short Depth> bool Corner< Data, Depth >::isBoundary (void) const**

Returns true if this corner is on a boundary, and false otherwise.

**template<typename Data, unsigned short Depth> bool Corner< Data, Depth >::operator< (const CornerPosition< Depth > & *other*) const**

This method orders the corners. The following is guaranteed: either  $A < B$ , or  $B < A$ . If neither of those hold true, then  $A$  is equal to  $B$ . This is a complete ordering.

**template<typename Data, unsigned short Depth> Data\* Corner< Data, Depth >::getData () const**

Operation to get the stored data. The stored data is not manipulated by the library. It is intended solely for the programmer's internal use.

**template<typename Data, unsigned short Depth> void Corner< Data, Depth >::setData (Data \* *newValue*)**

Operation to store data at this corner. The stored data is not manipulated by the library. It is intended solely for the programmer's internal use.

**template<typename Data, unsigned short Depth> Corner<Data, Depth>\* Corner< Data, Depth >::getNext (Direction *D*) const**

Get the next node in the direction  $D$  indicated. Returns NULL if there is no node linked in the direction indicated.

## Appendix A. *GeomOct Reference*

**template<typename Data, unsigned short Depth> void Corner< Data, Depth >::setNext (Direction *D*, Corner< Data, Depth > \* *newNext*)**

Set the next node in the indicated direction to be the *newNext* node. It also changes the previous node of *newNext* to be this node. This corresponds to an insertion in the linked list of direction *D*.

**template<typename Data, unsigned short Depth> Corner<Data, Depth>\* Corner< Data, Depth >::getPrevious (Direction *D*) const**

Get the previous node in the direction indicated. Returns NULL if there is no node linked in the direction indicated.

**template<typename Data, unsigned short Depth> void Corner< Data, Depth >::setPrevious (Direction *D*, Corner< Data, Depth > \* *newPrevious*)**

Set the previous node in the indicated direction to be the *newPrevious* node. It also changes the next node of *newPrevious* to be this node. This corresponds to an insertion in the linked list of direction *D*.

**template<typename Data, unsigned short Depth> void Corner< Data, Depth >::goForwardTillEnd (Direction *D*, CornerOperation< Data, Depth > & *op*)**

Keep following the next pointer in the direction indicated and apply the operation to all the nodes along the way.

**template<typename Data, unsigned short Depth> void Corner< Data, Depth >::goBackwardTillEnd (Direction *D*, CornerOperation< Data, Depth > & *op*)**

Keep following the previous pointer in the direction indicated and apply the operation to all the nodes along the way.

**template<typename Data, unsigned short Depth> static std::vector<Corner <Data, Depth> > Corner< Data, Depth >::getInterpolants (const std::vector< Corner< Data, Depth > > & *matchLists*, const CornerPosition< Depth > & *toCalculate*, short unsigned int *direction*) [static]**

Return nodes in the vector *matchLists* that do not match the location *toCalculate* in the direction indicated. The directions can be 0, 1, or 2, for X, Y, and Z axis, respectively. This method is required for finite difference methods in which

## Appendix A. *GeomOct Reference*

a missing value prompts an interpolation between points on the same plane. The return value of this method is a list of corners than can be used as interpolants for a finite difference method. If no interpolants are found, then an empty vector will be returned instead.

### Member Data Documentation

**template<typename Data, unsigned short Depth>  
CornerPosition<Depth> Corner< Data, Depth >::location**

This value specifies the location of this point. This variable is provided to ease the task of implementers of finite difference codes. Please treat this variable with respect: copying it is fine, but modifying this is strongly discouraged.

**template<typename Data, unsigned short Depth>  
NodePosition<Depth> Corner< Data, Depth >::nodeLocation**

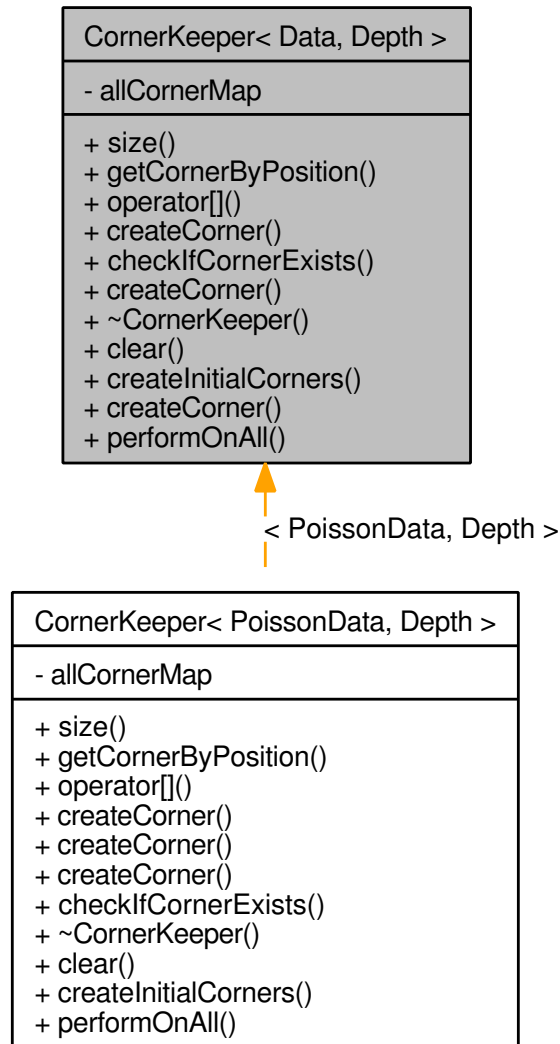
The location of a cell which is adjoint to this corner. This is one octree node that contains this corner. No guarantee on which node it is, it could be any node, but this corner is on it for sure. Formally, this corresponds to some adjoint cell, but there is no guarantee on the height of that adjoint cell.

### A.2.4 CornerKeeper< Data, Depth > Class Template Reference

```
#include <CornerKeeper.h>
```

## Appendix A. *GeomOct* Reference

Inheritance diagram for `CornerKeeper< Data, Depth >`:



### Public Member Functions

- `int size ()`
- `Corner< Data, Depth > * getCornerByPosition (const CornerPosition< Depth > &toSearch) const`
- `Corner< Data, Depth > * operator[] (const CornerPosition< Depth > &toSearch) const`



## Appendix A. *GeomOct* Reference

- `Corner< Data, Depth > * createCorner (CornerPosition< Depth > &toMake)`
- `bool checkIfCornerExists (const CornerPosition< Depth > &toSearch) const`
- `std::vector< Corner< Data, Depth > * > createCorner (std::vector< CornerPosition< Depth > > newLocationsAndOld) throw (CornerException)`
- `~CornerKeeper ()`
- `void clear (void)`
- `std::vector< Corner< Data, Depth > * > createInitialCorners (bool forceDelete) throw (CornerException)`
- `std::vector< Corner< Data, Depth > * > createCorner (const CornerList< Data, Depth > &toSplit) throw (CornerException)`
- `bool performOnAll (CornerOperation< Data, Depth > &op)`

### Detailed Description

**template<typename Data, unsigned short Depth> class CornerKeeper< Data, Depth >**

The `CornerKeeper` class keeps track of all the corners of the octree. The specifics of how the corners are stored is not important and might be subject to change. The keeper structure provides iterators and other method to ensure that no programmer would need to know the details of its implementation.

Since it occupies such a pivotal role in the creation of corners, it is essential that every octree have just one `CornerKeeper`, so that all the corners of the octree are centrally managed. All corners are doubly linked to the elements that are in the same cartesian plane as them. The class `CornerList` is a class that manages corners on the behalf of `OctreeNode`.

If more than one octree is being manipulated, a separate `CornerKeeper` should be created for each octree. Corners of two separate octrees can be stored in a single `CornerKeeper` object as well, though this would not be required in most circumstances.

### Constructor & Destructor Documentation

**template<typename Data, unsigned short Depth> CornerKeeper< Data, Depth >::~~CornerKeeper ()**

The destructor removes all the corners that are held by this keeper.

## Appendix A. *GeomOct Reference*

This necessitates that the [CornerKeeper](#) objects do not go out of scope, for that would mean all their kept corners are cleaned out. If corners mysteriously disappear, this is a good place to look.

### Member Function Documentation

**template<typename Data, unsigned short Depth> int CornerKeeper<Data, Depth >::size ()**

Returns the number of corners that are stored in the keeper.

**template<typename Data, unsigned short Depth> Corner< Data, Depth > \* CornerKeeper< Data, Depth >::getCornerByPosition (const CornerPosition< Depth > & *toSearch*) const**

Search all the corners for this [CornerPosition](#). If a corner is found, return it. If not, a NULL is returned. The returned data structure can be read, but modifications to it are not allowed.

**template<typename Data, unsigned short Depth> Corner<Data, Depth>\* CornerKeeper< Data, Depth >::operator[] (const CornerPosition< Depth > & *toSearch*) const**

Given a [CornerPosition](#), the [CornerKeeper](#) returns a pointer to the corner that matches this position, if one is stored. If no such corner exists, then a NULL is returned.

**template<typename Data, unsigned short Depth> Corner<Data, Depth > \* CornerKeeper< Data, Depth >::createCorner (CornerPosition< Depth > & *toMake*)**

Given the position of a corner, this method first checks if the corner exists. If the corner already exists with the keeper, then it is immediately returned. If no such corner exists, then it is created, and a pointer to it is returned. Also, this [CornerKeeper](#) keeps the newly created corner. This method always returns a corner.

**template<typename Data, unsigned short Depth> bool CornerKeeper< Data, Depth >::checkIfCornerExists (const CornerPosition< Depth > & *toSearch*) const**

## Appendix A. *GeomOct Reference*

Check if this specific corner exists in this [CornerKeeper](#), given the corner's position. Returns true if this [CornerKeeper](#) has the corner, and false otherwise.

```
template<typename Data, unsigned short Depth> std::vector<
Corner< Data, Depth > * > CornerKeeper< Data, Depth
>::createCorner (std::vector< CornerPosition< Depth > > newLoca-
tionsAndOld) throw (CornerException)
```

Create the entire set of 19 corners when an octree cell is refined. This method takes a vector of exactly 27 elements, which are the [CornerPosition](#) objects representing the location of the 27 corners. The last 8 of these are old corners: corners that are already with the parent. When the parent cell is split, this split will contain 27 corners in all. The first 19 elements in the argument are the [CornerPosition](#) objects for these 19 new corners.

The [CornerKeeper](#) iterates over all of these, creating a new [Corner](#) if one is required. After all the nodes are either found or created, we link them together.

Returns a vector of 27 corners: 19 new, and 8 old.

This method is very important, and all finite difference methods use it in some form or the other.

```
template<typename Data, unsigned short Depth> void CornerKeeper<
Data, Depth >::clear (void)
```

Delete all elements in this [CornerKeeper](#). Use with extreme caution and restraint. All references to corners become invalid after this method, and it should be used only when winding down a computation.

```
template<typename Data, unsigned short Depth> std::vector<
Corner< Data, Depth > * > CornerKeeper< Data, Depth
>::createInitialCorners (bool forceDelete = false) throw (CornerEx-
ception)
```

Create the initial eight corners that are adjoint to the root cell. Returns a vector of all the eight corners in standard order. For standard order, look up the documentation of the class [CornerList](#). The initial eight corners are formally defined as corners with height 1. Also, these are the only corners that are adjoint to the root octree cell.

If the optional argument `forceDelete` is set to true, then any information in the [CornerKeeper](#) is deleted, and a new set of eight elements are created. This is the same as deleting the [CornerKeeper](#) and creating a new one. Please do not pass this option unless the [CornerKeeper](#) is to be purged of all the corners that it currently holds. If this option is not given, or is specified as false (the default),

## Appendix A. *GeomOct Reference*

then this method cannot be called more than once. The second time it is called (and `forceDelete` is false), then it will throw an exception, since the [CornerKeeper](#) already has elements.

When unsure, do not specify `forceDelete`, and the right thing will happen.

```
template<typename Data, unsigned short Depth> std::vector<
Corner< Data, Depth > * > CornerKeeper< Data, Depth
>::createCorner (const CornerList< Data, Depth > & input) throw
(CornerException)
```

This creates new corners for a node that is about to be split. The [CornerList](#) passed as an argument belongs to the node to be split. In preparation for the split, new corner elements have to be created. This method takes as an argument the corners that belong to the node being split. It calculates which 19 corners will be created, and then creates the ones that do not exist. It returns a vector of all the 27 corners: 19 new, and 8 old, of points to [Corner](#) elements, which have either been created or searched.

A lot is happening in this method, but all the programmer needs to know is that this is called when an octree cell is split. Provide the existing corners as input. The output will contain the existing corners followed by the newly created corners. Some of the newly created corners might already exist but to the programmer, this minor detail is inconsequential.

```
template<typename Data, unsigned short Depth> bool
CornerKeeper< Data, Depth >::performOnAll (CornerOperation<
Data, Depth > & op)
```

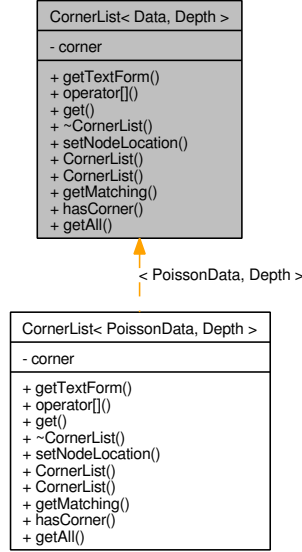
Perform this operation on all the nodes. The operation could modify the nodes, so the operation is not defined `const`. Modifying a corner is safe only if the method performing the modifications is correctly written. This simple method can cause large scale destruction of the octree corners. A good check would be to code up the [CornerOperation](#) and check it very thoroughly before unleashing it on all the corners kept in a [CornerKeeper](#) structure.

### A.2.5 `CornerList< Data, Depth >` Class Template Reference

```
#include <CornerList.h>
```

## Appendix A. *GeomOct* Reference

Inheritance diagram for `CornerList< Data, Depth >`:



### Public Member Functions

- `std::string` **getTextForm** (void) const
- `Corner< Data, Depth > * operator[]` (unsigned short index)
- `Corner< Data, Depth > * get` (unsigned short index) const
- void **setNodeLocation** (`NodePosition< Depth > newPosition`)
- `CornerList` (const `std::vector< Corner< Data, Depth > * >` &listOf27Corners, bool xDirection, bool yDirection, bool zDirection) throw (`CornerException`)
- `CornerList` (const `std::vector< Corner< Data, Depth > * >` &listOfCorners) throw (`CornerException`)
- `std::vector< Corner< Data, Depth > >` **getMatching** (const `CornerPosition< Depth > &toMatch`) const
- bool **hasCorner** (const `CornerPosition< Depth > &toMatch`) const
- `std::vector< Corner< Data, Depth > >` **getAll** (void) const

## Detailed Description

```
template<typename Data, unsigned short Depth> class CornerList<
Data, Depth >
```

Each [OctreeNode](#) has to keep track of its corners. This is the class that keeps track of all the corners for the [OctreeNode](#). The operation of this class should be invisible to most programmers. The corners themselves are stored in the [CornerKeeper](#), and this class keeps pointers to the corners allocated there. Also, the corners themselves are [Corner](#) elements, and we keep exactly eight pointers to them. This helps since most corners appear in more than one octree, and the number of octree nodes that a single [Corner](#) can appear in is fairly large.

The corners are numbered in order of increasing coordinates. This is the order of corners: 0: (0,0,0), 1: (0,0,1), 2: (0,1,0), 3: (0,1,1), 4: (1,0,0), 5: (1,0,1), 6: (1,1,0), 7: (1,1,1), where the tuples are (x,y,z) values, 0 means low end, and 1 means high end. This numbering follows the binary system. This is also why the numbering has been retained as 0..7 instead of 1..8. [Corner](#) (0,0,0) is on the low end of all three axes, and (1,1,1) is on the high end of all three axes.

Most users of this library will never need to instantiate a [CornerList](#). This is around for the clean separation of the library more than an entry point for use of the user of the library.

This class is of use to programmers modifying the GeomOct library rather than programmers using the GeomOct library. As such, the goal is to allow flexibility and control rather than ease of understanding. In particular, this class requires a very detailed understanding of the corner storage, which is of little concern to most programmers.

## Constructor & Destructor Documentation

```
template<typename Data, unsigned short Depth> CornerList< Data,
Depth >::CornerList (const std::vector< Corner< Data, Depth > * >
& listOf27Corners, bool x, bool y, bool z) throw (CornerException)
```

Create a corner list given the direction in which this child was created from the parent, and the list of the 27 corners that will be passed from augmenting the vector after newCornersAfterSplit(). The listOfCorners has 19 new corners, and the 8 old ones, making it a grand total of 27 total corners. Directions are as discussed earlier.

## Appendix A. *GeomOct* Reference

This method returns a corner list containing the corners of the cell in the direction indicated.

```
template<typename Data, unsigned short Depth> CornerList< Data,  
Depth >::CornerList (const std::vector< Corner< Data, Depth > * >  
& listOf8Corners) throw (CornerException)
```

Create the corners of the root cell. There are exactly 8 corners in this list, since at the root, there are exactly 8 corners, and no choice of directions. Formally, create the eight corners that are adjoint to the root cell.

### Member Function Documentation

```
template<typename Data, unsigned short Depth> Corner<Data,  
Depth>* CornerList< Data, Depth >::operator[] (unsigned short in-  
dex)
```

Get a pointer to a particular corner. The index must be between 0 and 7, inclusive. Returns a pointer to the specified corner. Refer to the class documentation to learn about the numbering of corners.

```
template<typename Data, unsigned short Depth> Corner<Data,  
Depth>* CornerList< Data, Depth >::get (unsigned short index) const
```

Get a particular corner. The index must be between 0 and 7, inclusive. Returns a pointer to the specified corner. Used in contexts where the [] operator is not allowed.

```
template<class Data, unsigned short Depth> void CornerList< Data,  
Depth >::setNodeLocation (NodePosition< Depth > newPosition)
```

This method instructs all the corners of this list of a cell that is adjoint to them. There is no check performed, and so the programmer is expected to understand the nature of adjoint cells, and ensure that the location passed is truly of a cell that is adjoint to all the corners in this list.

```
template<typename Data, unsigned short Depth> std::vector<  
Corner< Data, Depth > > CornerList< Data, Depth >::getMatching  
(const CornerPosition< Depth > & toMatch) const
```

## Appendix A. *GeomOct* Reference

Return copies of all corner elements that match any coordinate of the given [CornerPosition](#). These copied out corners do not have link information but only have the correct location and data.

This method exists to make it easy to copy out location and data without destroying the link information. Corners are linked along all coordinate directions, and copying a corners will certainly cause errors in this link structure. This method offers an easy way out of this quandry. The returned corner contains valid location information, and the correct data, but does not inherit the link structure. These returned corners are crippled. They can be read and modified, but the modifications do not impact the octree.

```
template<typename Data, unsigned short Depth> bool CornerList<
Data, Depth >::hasCorner (const CornerPosition< Depth > & toMatch)
const
```

Given a particular [CornerPosition](#), check to see if any corner in this list matches the particular position. Return value is true if some corner in this list has that position and false otherwise.

```
template<typename Data, unsigned short Depth> std::vector<
Corner< Data, Depth > > CornerList< Data, Depth >::getAll (void)
const
```

Get all the corners copied out. These copied out corners do not have link information but only have the correct location and data. Read the documentation related to [getMatching\(\)](#) to find why this would be beneficial.

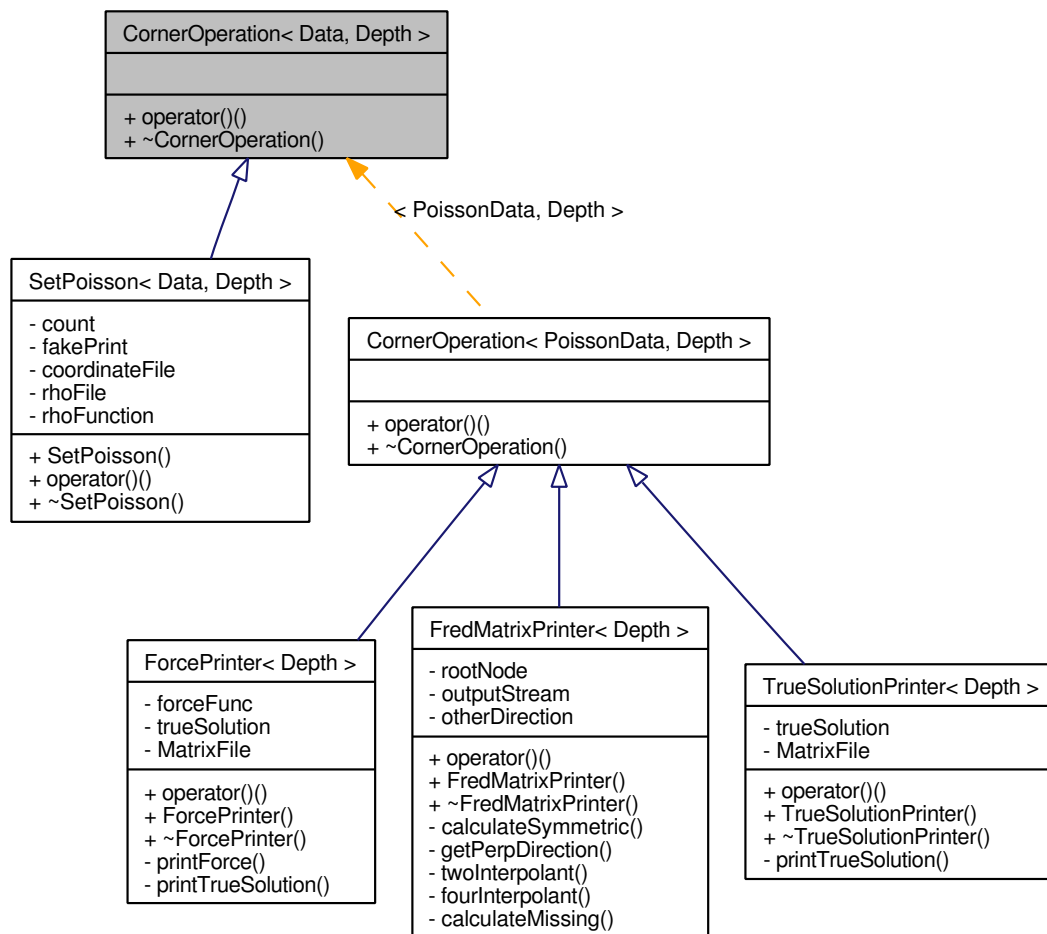
### A.2.6 `CornerOperation< Data, Depth >` Class Template Reference

```
#include <Corner.h>
```



## Appendix A. *GeomOct* Reference

Inheritance diagram for `CornerOperation< Data, Depth >`:



### Public Member Functions

- virtual bool `operator()` (`Corner< Data, Depth > &node`)=0

### Detailed Description

```
template<typename Data, unsigned short Depth> class
CornerOperation< Data, Depth >
```

This class specifies a functor for all the operations that can be performed on the corners. To make a new operation, use this as the base class and overload the

## Appendix A. *GeomOct Reference*

() operator. The only method in this is the () operator, which enables this class to act as a functor.

Functors subclassing this class will be run on all the corners of the octree in some order. The order is unspecified and may be implementation-specific.

### Member Function Documentation

```
template<typename Data, unsigned short Depth> virtual bool
CornerOperation< Data, Depth >::operator() (Corner< Data, Depth
> & node) [pure virtual]
```

Overload this to make an operator for nodes. Return true in this method by default. Return false only if the [CornerOperation](#) computation is to be interrupted. If this operator ever returns false, then there is no guarantee that it will be run on all corners.

Return true if unsure. Returning false in method can mean that this method does not run on the remaining nodes.

Implemented in [ForcePrinter< Depth >](#), [FredMatrixPrinter< Depth >](#), [SetPoisson< Data, Depth >](#), and [TrueSolutionPrinter< Depth >](#).

### A.2.7 [CornerPosition< Depth >](#) Class Template Reference

```
#include <CornerPosition.h>
```

#### Public Member Functions

- bool [isBoundary](#) () const
- bool [operator==](#) (const [CornerPosition](#) &other) const
- [CornerPosition](#) (bool xEnd, bool yEnd, bool zEnd)
- **CornerPosition** (unsigned int xVal, unsigned int yVal, unsigned int zVal)
- [CornerPosition](#) (const [CornerPosition](#)< Depth > &xComponent, const [CornerPosition](#)< Depth > &yComponent, const [CornerPosition](#)< Depth > &zComponent)
- [CornerPosition](#) (const [CornerPosition](#)< Depth > &toCopy)
- [CornerPosition](#) & [operator=](#) (const [CornerPosition](#)< Depth > &toCopy)
- bool [operator<](#) (const [CornerPosition](#)< Depth > &right) const
- std::string [getTextForm](#) (void) const

## Appendix A. *GeomOct Reference*

- `std::vector< double > getLocation` (double xMax, double yMax, double zMax) const
- `const NodePosition< Depth > getNodeLocation` () const
- `CornerPosition< Depth > getNext` (bool xNext, bool yNext, bool zNext) const throw (PositionException)
- `CornerPosition< Depth > getPrevious` (bool xPrevious, bool yPrevious, bool zPrevious) const throw (PositionException)
- `bool matchesAnyCoordinate` (const `CornerPosition< Depth >` &toMatch) const
- `bool matchesXCoordinate` (const `CornerPosition< Depth >` &toMatch) const
- `bool matchesYCoordinate` (const `CornerPosition< Depth >` &toMatch) const
- `bool matchesZCoordinate` (const `CornerPosition< Depth >` &toMatch) const

### Static Public Member Functions

- `static std::vector< CornerPosition< Depth > > newCornersAfterSplit` (const `CornerPosition< Depth >` &smallestCorner, const `CornerPosition< Depth >` &largestCorner)
- `static const unsigned short * returnCornerSpecification` (void)
- `static CornerPosition< Depth > getMidPoint` (const `CornerPosition< Depth >` &A, const `CornerPosition< Depth >` &B)

### Static Public Attributes

- `static const unsigned short cornerSpec` [64]

### Detailed Description

`template<unsigned short Depth> class CornerPosition< Depth >`

This class keeps information relevant to the position of a corner. The implementation of this class is hidden from the user since the details of the implementation are unnecessary.

This code is self-sufficient, and provides all the methods required to implement finite difference codes on arbitrary octree meshes. The methods `getNext()` and `getPrevious()` are most useful, since they obtain the position of the next and

## Appendix A. *GeomOct* Reference

previous grid points. Much of the functionality provided here will not be required in finite difference codes.

### Constructor & Destructor Documentation

```
template<unsigned short Depth> CornerPosition< Depth  
>::CornerPosition (bool xEnd = false, bool yEnd = false, bool zEnd  
= false)
```

Create a corner position. The optional arguments should be true if this corner is at the extreme end of that coordinate or not. So passing (false, true, true) specifies that this corner is at the beginning of the x axis, and at the ends of the y and z axes.

```
template<unsigned short Depth> CornerPosition< Depth  
>::CornerPosition (const CornerPosition< Depth > & xComponent,  
const CornerPosition< Depth > & yComponent, const  
CornerPosition< Depth > & zComponent)
```

Create a corner position using components from the given corner positions.

```
template<unsigned short Depth> CornerPosition< Depth  
>::CornerPosition (const CornerPosition< Depth > & toCopy)
```

This is a standard copy constructor.

### Member Function Documentation

```
template<unsigned short Depth> bool CornerPosition< Depth  
>::isBoundary () const
```

Return true if this position is on the boundary, false otherwise.

```
template<unsigned short Depth> bool CornerPosition< Depth  
>::operator== (const CornerPosition< Depth > & other) const
```

Compare two corner positions. Return true if they are equal, and false otherwise.

## Appendix A. *GeomOct Reference*

```
template<unsigned short Depth> CornerPosition< Depth > &  
CornerPosition< Depth >::operator= (const CornerPosition< Depth  
> & rhs)
```

This is a standard assignment operator.

```
template<unsigned short Depth> bool CornerPosition< Depth  
>::operator< (const CornerPosition< Depth > & right) const
```

Order two [CornerPosition](#) objects.

```
template<unsigned short Depth> static std::vector<  
CornerPosition<Depth> > CornerPosition< Depth  
>::newCornersAfterSplit (const CornerPosition< Depth > & smallest-  
Corner, const CornerPosition< Depth > & largestCorner) [static]
```

Return a vector of 19 new corner positions when a cell is split. The calling routine can check if some of these exist, or choose to make them if they don't. The arguments are the two extreme corners of the octree cell. The extreme corners are the one with least x,y,z coordinates, and the one with the highest x,y,z coordinates. These are also referred to as the least and the highest corner of any cell, as ordered by the operator< in this class.

```
template<unsigned short Depth> static const unsigned short*  
CornerPosition< Depth >::returnCornerSpecification (void) [static]
```

This is tied to the implementation of the [newCornersAfterSplit\(\)](#) method. We return the numbering of the corners according to the split, where numbers 20...27 are the original corners, also indexed in the same x, y, z way as the corners 1...19.

The numbering refers to the corner indices that are required to choose the correct corners when creating a new cell. Say that this class needs to create the 5th child cell of a particular parent. The new cell will need 8 corners, and we have the 19 corner positions that are made by the [newCornersAfterSplit\(\)](#) method. So to the 19 corner positions, we add the 8 positions of the parent. These 27 are passed to some object, which creates the corners for us. After these 27 nodes are obtained, this class needs to choose the 8 corners that are relevant to the 5th child cell.

This method will return a long vector (of exactly 64) values. Contiguous lines of 8 numbers specify the corners for the child cells. So for the 5th cell the values are the range  $8*4=32 \dots 8*4+7=39$ . (We want the 5th cell, and cells are numbered 0...7, so we want the cell numbered 4) Thus, the range in the returned vector is

## Appendix A. *GeomOct Reference*

of indices 32...39. That range specifies the indices of the corners that this 5th cell needs to have.

```
template<unsigned short Depth> static CornerPosition<Depth>
CornerPosition< Depth >::getMidPoint (const CornerPosition< Depth
> & A, const CornerPosition< Depth > & B) [static]
```

Get the corner in the middle of these two positions. A's coordinates must necessarily be either all smaller than or equal to B's coordinates

```
template<unsigned short Depth> std::vector< double >
CornerPosition< Depth >::getLocation (double xMax, double
yMax, double zMax) const
```

Get a vector containing 3 double values representing the decimal value of coordinates that this corner position represents. The original volume spans (0,0,0) to (xMax, yMax, zMax). The returned value should be indexed to get the (x, y, z) components of this corner position.

```
template<unsigned short Depth> const NodePosition< Depth >
CornerPosition< Depth >::getNodeLocation () const
```

Return the node that this missing corner is guaranteed to lie in.

```
template<unsigned short Depth> CornerPosition< Depth >
CornerPosition< Depth >::getNext (bool xNext, bool yNext, bool
zNext) const throw (PositionException)
```

Find the next position, by making the smallest possible increment on this position.

```
template<unsigned short Depth> CornerPosition< Depth >
CornerPosition< Depth >::getPrevious (bool xPrevious, bool yPrevi-
ous, bool zPrevious) const throw (PositionException)
```

Find the previous position, by making the smallest possible decrement on this position.

```
template<unsigned short Depth> bool CornerPosition< Depth
>::matchesAnyCoordinate (const CornerPosition< Depth > &
toMatch) const
```

Return true if the position matches the given position in any coordinate.

## Appendix A. *GeomOct* Reference

```
template<unsigned short Depth> bool CornerPosition< Depth
>::matchesXCoordinate (const CornerPosition< Depth > & toMatch)
const
```

Return true if the position matches the given position in the X coordinate.

```
template<unsigned short Depth> bool CornerPosition< Depth
>::matchesYCoordinate (const CornerPosition< Depth > & toMatch)
const
```

Return true if the position matches the given position in the Y coordinate.

```
template<unsigned short Depth> bool CornerPosition< Depth
>::matchesZCoordinate (const CornerPosition< Depth > & toMatch)
const
```

Return true if the position matches the given position in the Z coordinate.

## Member Data Documentation

```
template<unsigned short Depth> const unsigned short
CornerPosition< Depth >::cornerSpec [static]
```

Initial value:

```
{
    19, 0, 1, 2, 5, 6, 8, 9,
    0, 20, 2, 3, 6, 7, 9, 10,

    1, 2, 21, 4, 8, 9, 11, 12,
    2, 3, 4, 22, 9, 10, 12, 13,

    5, 6, 8, 9, 23, 14, 15, 16,
    6, 7, 9, 10, 14, 24, 16, 17,

    8, 9, 11, 12, 15, 16, 25, 18,
    9, 10, 12, 13, 16, 17, 18, 26
}
```

## Appendix A. *GeomOct* Reference

This is tied to the implementation of the `newCornersAfterSplit()` method. We return the numbering of the corners according to the split, where numbers 20...27 are the original corners, also indexed in the same x, y, z way as the corners 1...19.

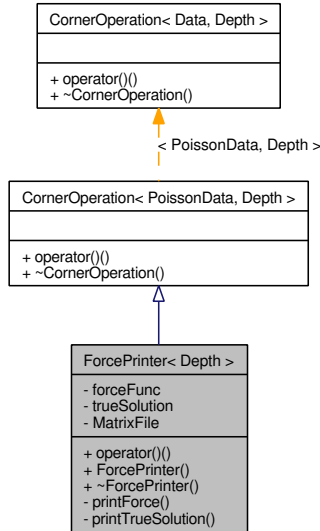
The numbering refers to the corner indices that are required to choose the correct corners when creating a new cell. Say that this class needs to create the 5th child cell of a particular parent. The new cell will need 8 corners, and we have the 19 corner positions that are made by the `newCornersAfterSplit()` method. So to the 19 corner positions, we add the 8 positions of the parent. These 27 are passed to some object, which creates the corners for us. After these 27 nodes are obtained, this class needs to choose the 8 corners that are relevant to the 5th child cell.

This method will return a long vector (of exactly 64) values. Contiguous lines of 8 numbers specify the corners for the child cells. So for the 5th cell the values are the range  $8*4=32 \dots 8*4+7=39$ . (We want the 5th cell, and cells are numbered 0...7, so we want the cell numbered 4) Thus, the range in the returned vector is of indices 32...39. That range specifies the indices of the corners that this 5th cell needs to have.

### A.2.8 `ForcePrinter< Depth >` Class Template Reference

```
#include <ForcePrinter.h>
```

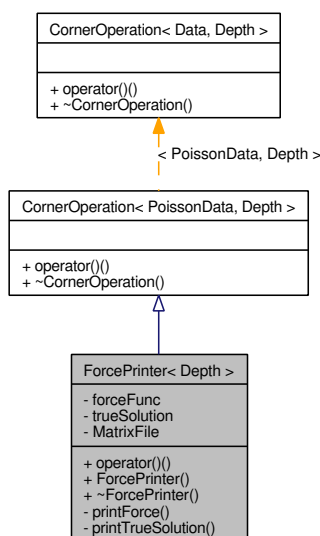
Inheritance diagram for `ForcePrinter< Depth >`:





## Appendix A. *GeomOct* Reference

Collaboration diagram for ForcePrinter< Depth >:



### Public Member Functions

- virtual bool `operator()` (`Corner< PoissonData, Depth > &myNode`)
- `ForcePrinter` (double(\*forceFunc)(double, double, double), double(\*trueSolution)(double, double, double), std::string matrixName)

### Detailed Description

`template<size_t Depth> class ForcePrinter< Depth >`

Make a functor that will print out the force function for solving this mesh.

The reason for the size of this class is to ensure that it is clean reusable code. Much of the flexibility derives from it. For example, the force function is specified outside this class which keeps the details of the force function separate from the implementation of this class.

The implementation of `operator()` here forms a very good demonstration of functors and CornerOperations.

### Constructor & Destructor Documentation

## Appendix A. *GeomOct* Reference

```
template<size_t Depth> ForcePrinter< Depth >::ForcePrinter  
(double(*) (double, double, double) forceFunc, double(*) (double,  
double, double) trueSolution, std::string matrixName)
```

Create a new [ForcePrinter](#) with the specified forceFunction. The expected forceFunction should be a pointer to a function that takes three arguments (double x, double y, double z), and returns the force at the point (x,y,z). The function trueSolution, is also a pointer to a function that gives the true solution at every point. Both the true solution and the forceFunction are needed because at the boundary points, the forceFunction is set to the true solution, assuming Dirichlet boundary conditions.

### Member Function Documentation

```
template<size_t Depth> bool ForcePrinter< Depth >::operator()  
(Corner< PoissonData, Depth > & myNode) [virtual]
```

From each node, write down the forcing function at this point. Since everything is specified in the constructor, no information is needed at this stage.

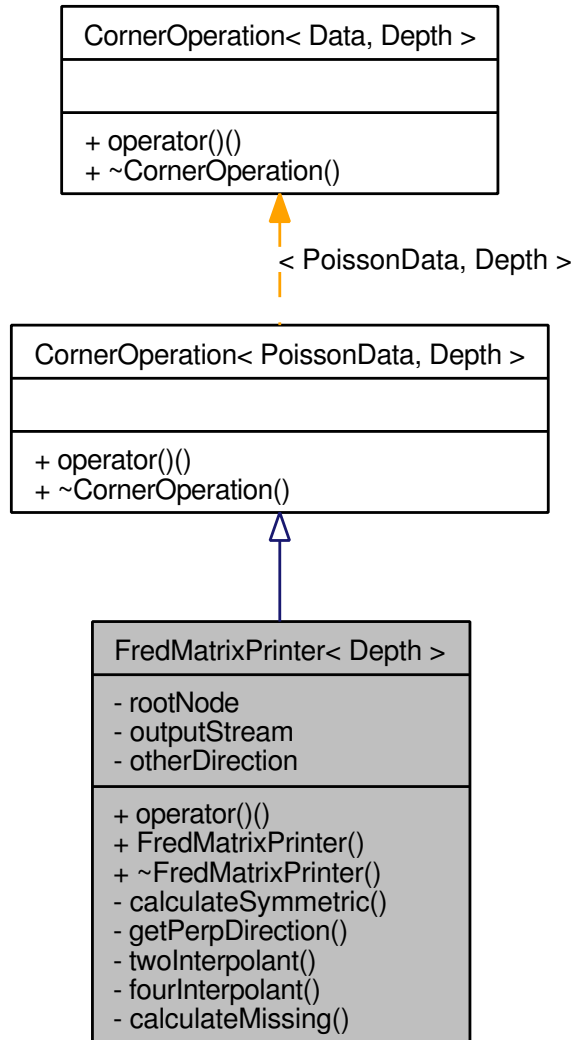
Implements [CornerOperation< PoissonData, Depth >](#).

### A.2.9 FredMatrixPrinter< Depth > Class Template Reference

```
#include <FredMatrixPrinter.h>
```

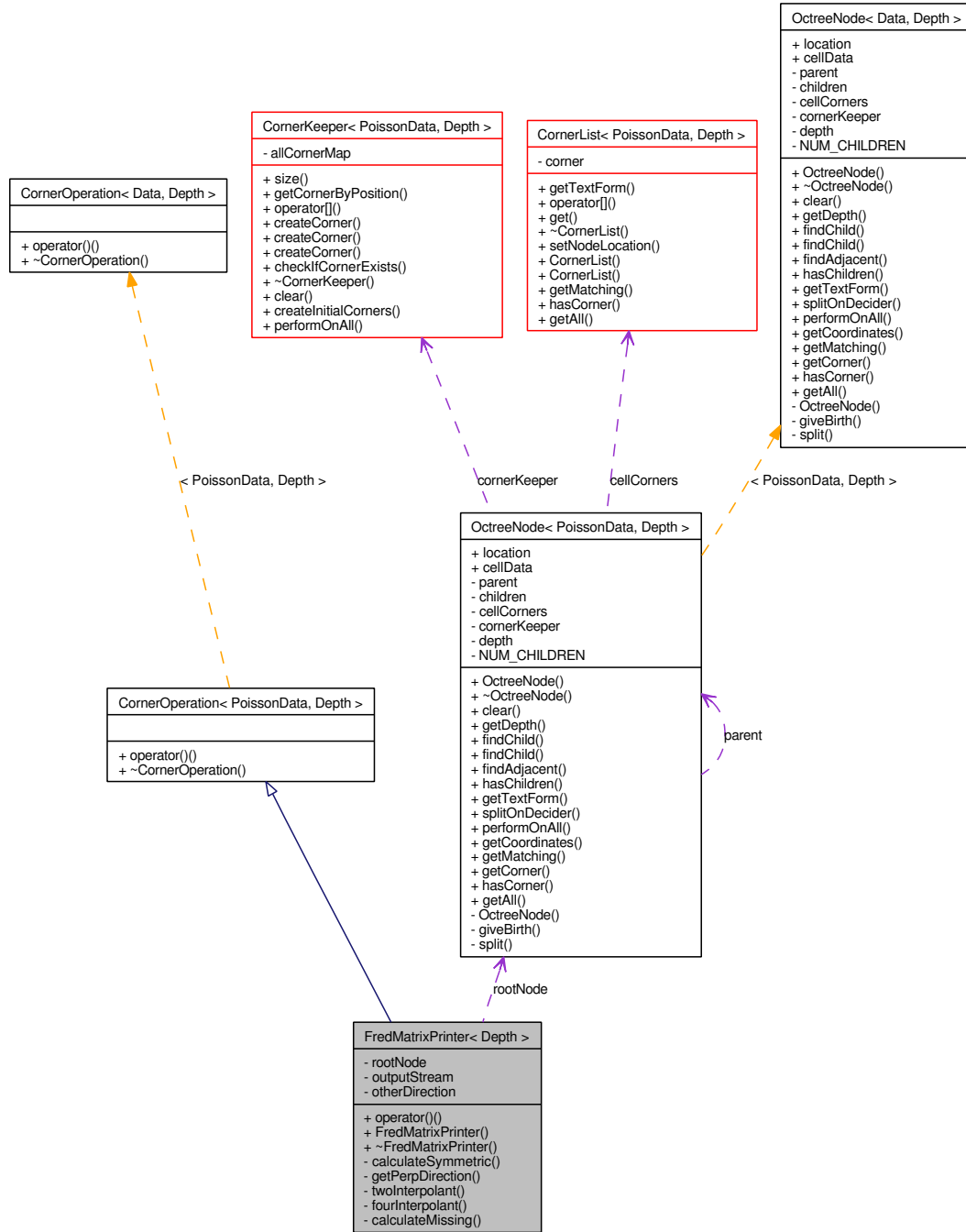
## Appendix A. *GeomOct Reference*

Inheritance diagram for FredMatrixPrinter< Depth >:



## Appendix A. *GeomOct* Reference

Collaboration diagram for FredMatrixPrinter< Depth >:



## Public Member Functions

- virtual bool `operator()` (`Corner`< `PoissonData`, `Depth` > &`myNode`)
- `FredMatrixPrinter` (`OctreeNode`< `PoissonData`, `Depth` > \*`octreeRoot`, `std::string` `matrixName`)

## Detailed Description

`template<size_t Depth> class FredMatrixPrinter< Depth >`

Functor that prints out the matrix corresponding to the second order Poisson discretization described in the work by Min, Gibou Cenicerros. All the heavy lifting is done in this class, which locates the dependencies, and calculates the interpolants. The associated class `NodeInterpolant` specifies what is to be done with the interpolants and the data dependencies.

If other finite difference or finite volume discretizations need to be calculated, this class forms a good starting point. A copy of this class can be used as a template to generate the data dependencies and the node interpolants. Once the corner dependencies are resolved, a copy of the `NodeInterpolant` class can be used to perform the finite difference discretization.

This class is not required for the functioning of *GeomOct* and marks a demonstration of using *GeomOct* for finite difference codes.

## Constructor & Destructor Documentation

```
template<size_t      Depth>      FredMatrixPrinter<      Depth
>::FredMatrixPrinter (OctreeNode< PoissonData, Depth > * oc-
treeRoot, std::string matrixName)
```

Create a new functor with `octreeRoot` as the root cell, and the indicated `matrixName` for creating the output file.

## Member Function Documentation

```
template<size_t      Depth>      bool      FredMatrixPrinter<      Depth
>::operator() (Corner< PoissonData, Depth > & myNode) [virtual]
```

## Appendix A. *GeomOct Reference*

This method does everything useful in this class. This method fires up all the remaining methods in this class. The general structure of this is quite easy to follow.

From each corner, this method first tracks down the corners that it depends on. This is referred to as data-dependency between corners, or edges between vertices. Once the data dependency has been obtained, we need the specific coefficients associated with the dependencies. The specific implementation of the coefficient generation is in the [NodeInterpolant](#) class. This is of great help when the edges between the vertices is identical, but the finite difference discretization requires a different weighting scheme.

This method writes each nonzero of the matrix as an *ijv* pair, for use by Matlab, or any other matrix analysis software.

Implements [CornerOperation< PoissonData, Depth >](#).

### A.2.10 **NodeInterpolant Class Reference**

```
#include <NodeInterpolant.h>
```

#### **Public Member Functions**

- `std::vector< CoefficientTerms > getTerms` (unsigned short direction) const
- [NodeInterpolant](#) (unsigned long number, double rho, std::ofstream &outputStream)
- [NodeInterpolant](#) (unsigned long number, std::ofstream &outputStream)
- `void printEdges` (void) const
- `void calculateNoInterpolant` (unsigned short direction, unsigned long prevId, double prevRho, double prevDistance, unsigned long nextId, double nextRho, double nextDistance)
- `void calculateTwoInterpolant` (unsigned short direction, unsigned long prevId, double prevRho, double nodePrevDistance, unsigned long upId, double upRho, double upMissingDistance, unsigned long downId, double downRho, double downMissingDistance, double nodeMissingDistance, short unsigned perpDirection)
- `void calculateFourInterpolant` (unsigned short direction, unsigned long prevId, double prevRho, double nodePrevDistance, unsigned long downLeftId, double downLeftRho, unsigned long downRightId, double downRightRho, unsigned long upLeftId, double upLeftRho, unsigned long upRightId, double upRightRho, unsigned short horizDirection, double leftMissingDistance,

## Appendix A. *GeomOct* Reference

- double rightMissingDistance, unsigned short vertDirection, double downMissingDistance, double upMissingDistance, double nodeMissingDistance)
- void [handleBoundary](#) () const

### Detailed Description

This class represents the mathematical details of a finite difference method. In this case, it refers to the second order accurate method devised by C-H Min et al. This class alone contains all the mathematical details. The remaining *GeomOct* code is devoid of any mathematical subtlety. As such, it forms a very elegant separation of mathematical logic, and graph details. *GeomOct* is filled with graph details, with some classes painfully specifying data structures to ensure that graph details are carefully encapsulated. This class alone marks the mathematical roots of *GeomOct*. Its separation from the graph details is of great benefit to programmers, since they can modify this one class and generate very different finite difference methods.

This greatly helps with the testing of finite difference methods, since a class containing straight-line mathematical code is very easy to debug. The heavy lifting is all done outside this class. The implementation of this class is nearly devoid of any *GeomOct*-specific data structures.

Some implementation specific notes: throughout the class *rho* specifies the value of the variable coefficient in a Poisson equation. *Id* specifies the node index, or the number associated with every grid point. The three co-ordinate axes are listed in their usual analogy with the physical world: up/down, left/right and front/back. Missing refers to a missing grid point which is interpolated using two or four values.

This class is not required for the functioning of *GeomOct*. It is provided as a demonstration of finite difference codes.

### Constructor & Destructor Documentation

**NodeInterpolant::NodeInterpolant** (unsigned long *number*, double *rho*, std::ofstream & *outputStream*)

Create a new node interpolant by specifying the index of the current node and the value of the variable coefficient *rho*.

**NodeInterpolant::NodeInterpolant** (unsigned long *number*, std::ofstream & *outputStream*)

## Appendix A. *GeomOct* Reference

Create a boundary node, by specifying their index alone. The value of the variable coefficient is un-necessary.

### Member Function Documentation

**std::vector<CoefficientTerms> NodeInterpolant::getTerms (unsigned short *direction*) const**

Get all the coefficients that determine this finite difference calculation, along with their associated coefficient values.

**void NodeInterpolant::printEdges (void) const**

After having populated all the values, print out the data. This method prints out a single coefficient per line in the standard ijk format.

**void NodeInterpolant::calculateNoInterpolant (unsigned short *direction*, unsigned long *prevId*, double *prevRho*, double *nodePrevDistance*, unsigned long *nextId*, double *nextRho*, double *nodeNextDistance*)**

The symmetric dependency is calculated here. This method requires the direction, the index of the previous and next nodes, the rho values of the previous and next nodes, and the distance of the current node from the previous and next, in the order specified.

**void NodeInterpolant::calculateTwoInterpolant (unsigned short *direction*, unsigned long *prevId*, double *prevRho*, double *nodePrevDistance*, unsigned long *upId*, double *upRho*, double *upMissingDistance*, unsigned long *downId*, double *downRho*, double *downMissingDistance*, double *nodeMissingDistance*, short unsigned *perpDirection*)**

Calculate the weights when the dependence is on two points. The previous node is present and is referred to by 'prev'. The next node is absent, and is interpolated from up and down. The interpolation yields the missing point. The direction along which the up and down nodes lie is called the *perpDirection*.

**void NodeInterpolant::calculateFourInterpolant (unsigned short *direction*, unsigned long *prevId*, double *prevRho*, double *nodePrevDistance*, unsigned long *downLeftId*, double *downLeftRho*, unsigned long *downRightId*, double *downRightRho*, unsigned long *upLeftId*, double *upLeftRho*, unsigned long *upRightId*, double *upRightRho*, unsigned**



**short *horizDirection*, double *leftMissingDistance*, double *rightMissingDistance*, unsigned short *vertDirection*, double *downMissingDistance*, double *upMissingDistance*, double *nodeMissingDistance*)**

Calculate the weights when the dependance is on four points. This function has a lot of input arguments. The direction is the one along which the partial derivative of u is being approximated. Prev is the node that is present, and is called the previous node, in keeping with the [calculateNoInterpolant\(\)](#) method. Missing is the point that is interpolated from the four interpolants.

**void NodeInterpolant::handleBoundary () const**

Set the boundary conditions.

### A.2.11 NodePosition< Depth > Class Template Reference

```
#include <NodePosition.h>
```

#### Public Member Functions

- [NodePosition](#) ([NodePosition](#)< Depth > parent, bool x, bool y, bool z)
- [NodePosition](#) (PORTABLE\_BITSET< Depth > x, PORTABLE\_BITSET< Depth > y, PORTABLE\_BITSET< Depth > z, unsigned short int bits)
- [NodePosition](#) (Coordinate< Depth > x, Coordinate< Depth > y, Coordinate< Depth > z, unsigned short int bits)
- std::vector< double > [getRange](#) (double xEnd, double yEnd, double zEnd) const
- [NodePosition](#)< Depth > [getNext](#) (bool xNext, bool yNext, bool zNext) const throw (PositionException)
- [NodePosition](#)< Depth > [getPrevious](#) (bool xPrevious, bool yPrevious, bool zPrevious) const throw (PositionException)
- bool [isMyAncestor](#) ([NodePosition](#) potential) const
- bool [equalsNBits](#) (const [NodePosition](#) &toCompare, unsigned short N) const
- std::string [getTextForm](#) () const
- bool [operator==](#) (const [NodePosition](#)< Depth > &other) const
- unsigned short [getLevel](#) () const
- unsigned short int [getIndexOfEntry](#) (unsigned short int level) const

## Static Public Member Functions

- static unsigned short **tripleToIndex** (bool x, bool y, bool z)
- static bool **indexToX** (unsigned short index)
- static bool **indexToY** (unsigned short index)
- static bool **indexToZ** (unsigned short index)

## Detailed Description

**template<unsigned short Depth> class NodePosition< Depth >**

This is the position of any node in the octree. The boolean values can be thought of as left (false) and right (true).

The specific implementation of this class is hidden from the programmer since the implementation is subject to change. This class provides a long list of utility methods that ensure that its internal representation is not required.

## Constructor & Destructor Documentation

**template<unsigned short Depth> NodePosition< Depth >::NodePosition (NodePosition< Depth > *parent*, bool *xNext*, bool *yNext*, bool *zNext*)**

Given the position of the parent, create the position of the child from it. Also required is the displacement of the child from the parent, which forms the specific part of the location, since prefixes are identical.

**template<unsigned short Depth> NodePosition< Depth >::NodePosition (PORTABLE\_BITSET< Depth > *x*, PORTABLE\_BITSET< Depth > *y*, PORTABLE\_BITSET< Depth > *z*, unsigned short int *bits*)**

Create a position, given the three components, in the three directions, and a short integer specifying how many of the bits are valid.

**template<unsigned short Depth> NodePosition< Depth >::NodePosition (Coordinate< Depth > *x*, Coordinate< Depth > *y*, Coordinate< Depth > *z*, unsigned short int *bits*)**

Create a position, given the three components, in the three directions, and a short integer specifying how many of the bits are valid.

## Member Function Documentation

```
template<unsigned short Depth> std::vector< double > NodePosition<  
Depth >::getRange (double xEnd, double yEnd, double zEnd) const
```

Since this represents the position of an [OctreeNode](#), this method returns the range of decimal values that the node occupies. The octree is always started at the origin (0,0,0), and the three arguments here represent the end of the octree. The return value is a vector of doubles containing six values, which are three two-pair values. The first two pairs: values [0] and [1] are the range (low and high) of the X coordinate. The next two: [2], [3] are the range of Y coordinate, and the last two [4], [5] are the range of the Z coordinate. This is a range since the [OctreeNode](#) occupies some volume, and so a complete description requires at least six points. The values [0],[2],[4] form the decimal coordinates of the smallest numbered corner, and the values [1], [3], [5] form the decimal coordinates of the highest numbered corner. Using these, it is easy to come up with the decimal coordinates of all the corners of this node.

```
template<unsigned short Depth> NodePosition< Depth >  
NodePosition< Depth >::getNext (bool xNext, bool yNext, bool  
zNext) const throw (PositionException)
```

Find the next position, by making the smallest possible increment on this position.

```
template<unsigned short Depth> NodePosition< Depth >  
NodePosition< Depth >::getPrevious (bool xPrevious, bool yPrevious,  
bool zPrevious) const throw (PositionException)
```

Find the previous position, by making the smallest possible decrement on this position.

```
template<unsigned short Depth> bool NodePosition< Depth  
>::isMyAncestor (NodePosition< Depth > potential) const
```

Check if the given position could possibly be the position of this node's parent. Returns true if the position is a child of the object with the position passed as an argument. Return false otherwise.

## Appendix A. *GeomOct Reference*

```
template<unsigned short Depth> bool NodePosition< Depth
>::equalsNBits (const NodePosition< Depth > & toCompare, unsigned
short N) const
```

Compare the *N* most relevant bits of position between these two objects. Returns true if the positions match upto *N* most significant bits, and false otherwise.

```
template<unsigned short Depth> bool NodePosition< Depth
>::operator== (const NodePosition< Depth > & other) const
```

Check if two position objects contain the same data, by checking their positions, and the number of bits they hold.

```
template<unsigned short Depth> unsigned short NodePosition< Depth
>::getLevel () const
```

Retruns the level of refinement for this node. The original volume has level=1, and every child of an octree node with level *i* has a level *i*+1. Formally, this method returns one more than the height of the octree cell.

### A.2.12 Octree< Data, Depth > Class Template Reference

```
#include <Octree.h>
```

#### Public Member Functions

- [Octree](#) (double(\*levelFunc)(double, double, double), double xEnd, double yEnd, double zEnd)
- void [createTree](#) () throw (OctreeException)
- void [performOnCorners](#) ([CornerOperation](#)< Data, Depth > &op) throw (OctreeException)
- std::string [getTextForm](#) (void) const throw (OctreeException)
- [~Octree](#) ()

#### Detailed Description

```
template<class Data, size_t Depth> class Octree< Data, Depth >
```

The [Octree](#) data structure holds 3D spatial information in a structure that is designed to cut up a volume into eight pieces. This is provided as a utility class for generating octrees from specific level set functions. It is meant to be used as

## Appendix A. *GeomOct* Reference

a quick way to generate octrees when a level set function is known and nothing else is required.

The template argument `depth` decides the maximum depth of the octree and `data` specifies the finite difference method data to be stored at the nodes.

This class is provided to enable programmers to generate octrees extremely quickly. No knowledge of the grid generation is required. The octrees will be refined to capture the interface specified by the level set function. The boundary of the interface is defined to be the points at which the level set function has value zero.

### Constructor & Destructor Documentation

```
template<class Data, size_t Depth> Octree< Data, Depth >::Octree  
(double (*)(double, double, double) levelFunc, double x, double y,  
double z)
```

Create the root of a tree, given the function to use as the level set function, and the end points of the volume. The level set function should ideally be a signed distance function. After creating an octree, the `createTree()` creates the octree. This allocates storage for all the nodes, the corners, and sets up all data structures. So a call to `createTree()` should immediately follow this, or occur between the constructor and any use of the octree.

```
template<class Data, size_t Depth> Octree< Data, Depth >::~~Octree  
(void)
```

Delete a previously defined `OctreeNode`. This just deletes the root, and nothing else. In general, a lot more might be required to ensure no memory leak. Since this class is meant as a prototype and a quick octree generator, this destructor is left empty.

### Member Function Documentation

```
template<class Data, size_t Depth> void Octree< Data, Depth  
>::createTree (void) throw (OctreeException)
```

Create the octree using the level set function provided in the constructor.

## Appendix A. *GeomOct* Reference

```
template<class Data, size_t Depth> void Octree< Data, Depth
>::performOnCorners (CornerOperation< Data, Depth > & op) throw
(OctreeException)
```

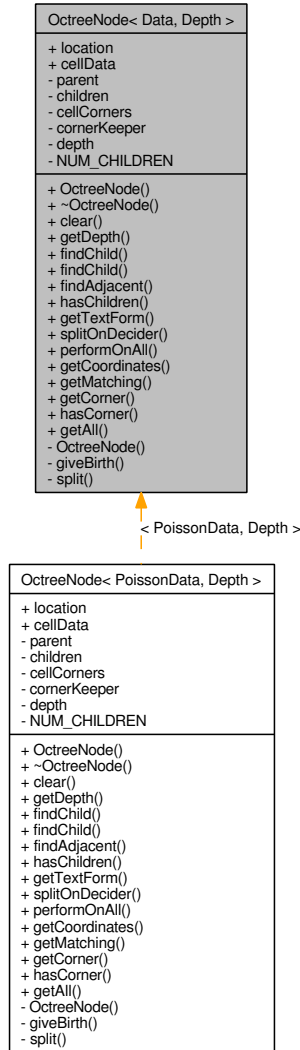
Allows the user of this library to perform any arbitrary operation on all the corners. The operation can modify the corners, or their data.

### A.2.13 **OctreeNode< Data, Depth > Class Template Reference**

```
#include <OctreeNode.h>
```

## Appendix A. *GeomOct* Reference

Inheritance diagram for `OctreeNode< Data, Depth >`:



### Public Member Functions

- `OctreeNode` (`CornerKeeper< Data, Depth > *keeper`)
- `void clear` (`void`)
- `const unsigned short getDepth` (`() const`)
- `const OctreeNode< Data, Depth > * findChild` (`const NodePosition< Depth > &childPosition`) `const`

## Appendix A. *GeomOct Reference*

- const [OctreeNode](#)< Data, Depth > \* [findChild](#) (const [CornerPosition](#)< Depth > &toFind) const
- const [OctreeNode](#)< Data, Depth > \* [findAdjacent](#) (const [NodePosition](#)< Depth > &nodeSuggestion, unsigned short direction, bool lookBehind) const
- bool [hasChildren](#) (void) const
- std::string [getTextForm](#) (void) const
- void [splitOnDecider](#) ([SplittingCriterion](#)< Data, Depth > &op) throw ([OctreeException](#))
- bool [performOnAll](#) ([OctreeNodeOperation](#)< Data, Depth > &op)
- std::vector< double > [getCoordinates](#) (double xEnd, double yEnd, double zEnd) const
- std::vector< [Corner](#)< Data, Depth > > [getMatching](#) (const [CornerPosition](#)< Depth > &toMatch) const
- [Corner](#)< Data, Depth > \* [getCorner](#) (unsigned short which) const
- bool [hasCorner](#) (const [CornerPosition](#)< Depth > &toMatch) const
- std::vector< [Corner](#)< Data, Depth > > [getAll](#) (void) const

### Public Attributes

- [NodePosition](#)< Depth > [location](#)
- Data \* [cellData](#)

### Friends

- class [SetPartitioned](#)< Depth >
- class [ColorNumberer](#)< Depth >

### Detailed Description

**template<class Data, unsigned short Depth> class OctreeNode< Data, Depth >**

The [OctreeNode](#) data structure holds 3D spatial information in a structure that is designed to cut up a volume into eight pieces. The template argument depth decides the maximum depth of the octree. The template argument data allows the storing of a pointer to any arbitrary information at this node. OctreeNodes are designed to be used from the [Octree](#) class, where the root node is stored explicitly. This is also the node that will get created when the public constructor is called. All other nodes are created by their parents, using the explicit split() call. The



## Appendix A. *GeomOct* Reference

recommended way of splitting the root node is to write a functor which inherits from [SplittingCriterion](#). This functor can be passed by reference to the method [splitOnDecider\(\)](#), which calls this functor at every point to decide whether to split at the current node or not. This is the civilized way of making an octree.

The template argument `depth` specifies the maximum depth of the tree. This is restricted to be unsigned, since a depth of a negative number makes no sense. It is short because we don't expect a depth limitation of 256 to be a big constraint.

This class is at the heart of *GeomOct*. It provides many methods for the manipulation of octrees. It is advisable to use these methods to ensure consistency of data structures. A lot of octree manipulation can be done by functors such as [OctreeNodeOperation](#). We recommend programmers to use functors as far as possible, and investigate the internal structure of octrees only if functors will not suffice. As a demonstration, an entire second order difference method has been programmed using functors. Refer to the classes [FredMatrixPrinter](#) and [NodeInterpolant](#) for more guidance on how to do this.

This class has many implementation-specific issues that are specified in the source code. Many of these are to ensure memory correctness, and to deallocate all memory.

### Constructor & Destructor Documentation

```
template<class Data, unsigned short Depth> OctreeNode< Data,  
Depth >::OctreeNode (CornerKeeper< Data, Depth > * keeper)
```

Create the root of a tree. This constructor needs a pointer to the keeper of corners which should either be [CornerKeeper](#), or inherit from [CornerKeeper](#). All other values are set to make this the root of the tree. The node data at this point is guaranteed to be NULL.

### Member Function Documentation

```
template<class Data, unsigned short Depth> void OctreeNode< Data,  
Depth >::clear (void)
```

Clear an [OctreeNode](#) and all its children.

## Appendix A. *GeomOct Reference*

**template<class Data, unsigned short Depth> const unsigned short OctreeNode< Data, Depth >::getDepth () const**

Return the depth of this node from the root. A depth of 0 specifies the root. Formally, this is also referred to as level of an octree cell.

**template<class Data, unsigned short Depth> const OctreeNode< Data, Depth > \* OctreeNode< Data, Depth >::findChild (const NodePosition< Depth > & *toFind*) const**

Find a child with the given coordinates. If the child is not found, then return its immediate parent.

The caller is expected to verify that the match has been exact. If the match is not exact, the caller is responsible for handling the case where a parent might be returned instead. While this code is called findChild, its true return value is either the child, or its closest ancestor that exists. If an ancestor is returned, it is guaranteed to be of highest depth. That is, if we draw the tree on a sheet of paper, and then draw the missing node, then the value returned by this routine is closest to the missing node.

**template<class Data, unsigned short Depth> const OctreeNode< Data, Depth > \* OctreeNode< Data, Depth >::findChild (const CornerPosition< Depth > & *toFind*) const**

Find an octree node which contains the corner position given here.

**template<class Data, unsigned short Depth> const OctreeNode< Data, Depth > \* OctreeNode< Data, Depth >::findAdjacent (const NodePosition< Depth > & *startingNode*, unsigned short *direction*, bool *lookBehind*) const**

Find an octree node which is located either behind (*lookBehind* = true) or ahead (*lookBehind* = false) of the node position given by *startingNode* in the direction indicated (0=X, 1=Y, 2=Z) by the second argument.

**template<class Data, unsigned short Depth> bool OctreeNode< Data, Depth >::hasChildren (void) const**

Returns true if this node has children, and false otherwise.

**template<class Data, unsigned short Depth> void OctreeNode< Data, Depth >::splitOnDecider (SplittingCriterion< Data, Depth > & *op*) throw (OctreeException)**

## Appendix A. *GeomOct Reference*

Split based on the functor provided here. The functor could have state, and this allows the functor to modify its own state. This function must throw an `OctreeException` if the splitting encountered some error. This is the suggested way to split the octree. The [SplittingCriterion](#) cannot modify the nodes, but is permitted to modify itself.

```
template<class Data, unsigned short Depth> bool OctreeNode< Data,  
Depth >::performOnAll (OctreeNodeOperation< Data, Depth > & op)
```

This method runs the specified const operation on all the nodes below this level. The [OctreeNodeOperation](#) must return true. If even a single operation returns false, then this loop might exit prematurely. The final return value is true if all operations returned true, and false if even a single one returned false. To guarantee all nodes have this operation run on them, make your operation return true.

```
template<typename Data, unsigned short Depth> std::vector< double  
> OctreeNode< Data, Depth >::getCoordinates (double xEnd, double  
yEnd, double zEnd) const
```

Convert the three locations into coordinates. This method requires a triple of coordinates marking the end of the region. The domain is assumed to begin at (0,0,0) and end at (xEnd, yEnd, zEnd). The return value is a vector of doubles containing six values, which are three two-pair values. The first two pairs: values [0] and [1] are the range (low and high) of the X coordinate. The next two: [2], [3] are the range of Y coordinate, and the last two [4], [5] are the range of the Z coordinate. This is a range since the octree cells occupies volume, and so a complete description requires at least six points. The values [0],[2],[4] form the decimal coordinates of the smallest numbered corner, and the values [1], [3], [5] form the decimal coordinates of the highest numbered corner. Using these, it is easy to come up with the decimal coordinates of all the corners of this node.

```
template<typename Data, unsigned short Depth> std::vector<  
Corner< Data, Depth > > OctreeNode< Data, Depth >::getMatching  
(const CornerPosition< Depth > & toMatch) const
```

Return copies of all corners that match any coordinate of the given [CornerPosition](#). These copied out corners do not have link information but only have the correct location and data.

## Appendix A. *GeomOct Reference*

```
template<typename Data, unsigned short Depth> Corner< Data,  
Depth > * OctreeNode< Data, Depth >::getCorner (unsigned short  
index) const
```

Given an index of a corner, return the corner. Corners are labelled 0..7, so the index should be within this range. They are also labelled in increasing order of coordinates Z, Y, and X, in that order, so the corners are (0,0,0), (0,0,1), (0,1,0), ..., (1,1,1), just like the binary expansion of the numbers. 0 means lesser, and 1 means higher.

```
template<typename Data, unsigned short Depth> bool OctreeNode<  
Data, Depth >::hasCorner (const CornerPosition< Depth > & toMatch)  
const
```

Return true if this [CornerPosition](#) matches any of the corners of this octree node. Return false if none of the corners match the given corner position. Formally, this method checks if the corner position specified matches the position of a corner that this cell adjoins.

```
template<typename Data, unsigned short Depth> std::vector<  
Corner< Data, Depth > > OctreeNode< Data, Depth >::getAll (void)  
const
```

Get all the corners copied out. These copied out corners do not have link information but only have the correct location and data.

## Member Data Documentation

```
template<class Data, unsigned short Depth> NodePosition<Depth>  
OctreeNode< Data, Depth >::location
```

This location corresponds to the range that the octree cell occupies.

```
template<class Data, unsigned short Depth> Data* OctreeNode<  
Data, Depth >::cellData
```

The data stored at this cell is of type Data which is the first argument to the template. This code just keeps the data, it does not do anything else with it. Programmers are responsible for any allocation or cleaning up of this data structure if you do plan to use it. The library maintains its consistency, and the programmer is tasked with maintaining its correctness and utility.

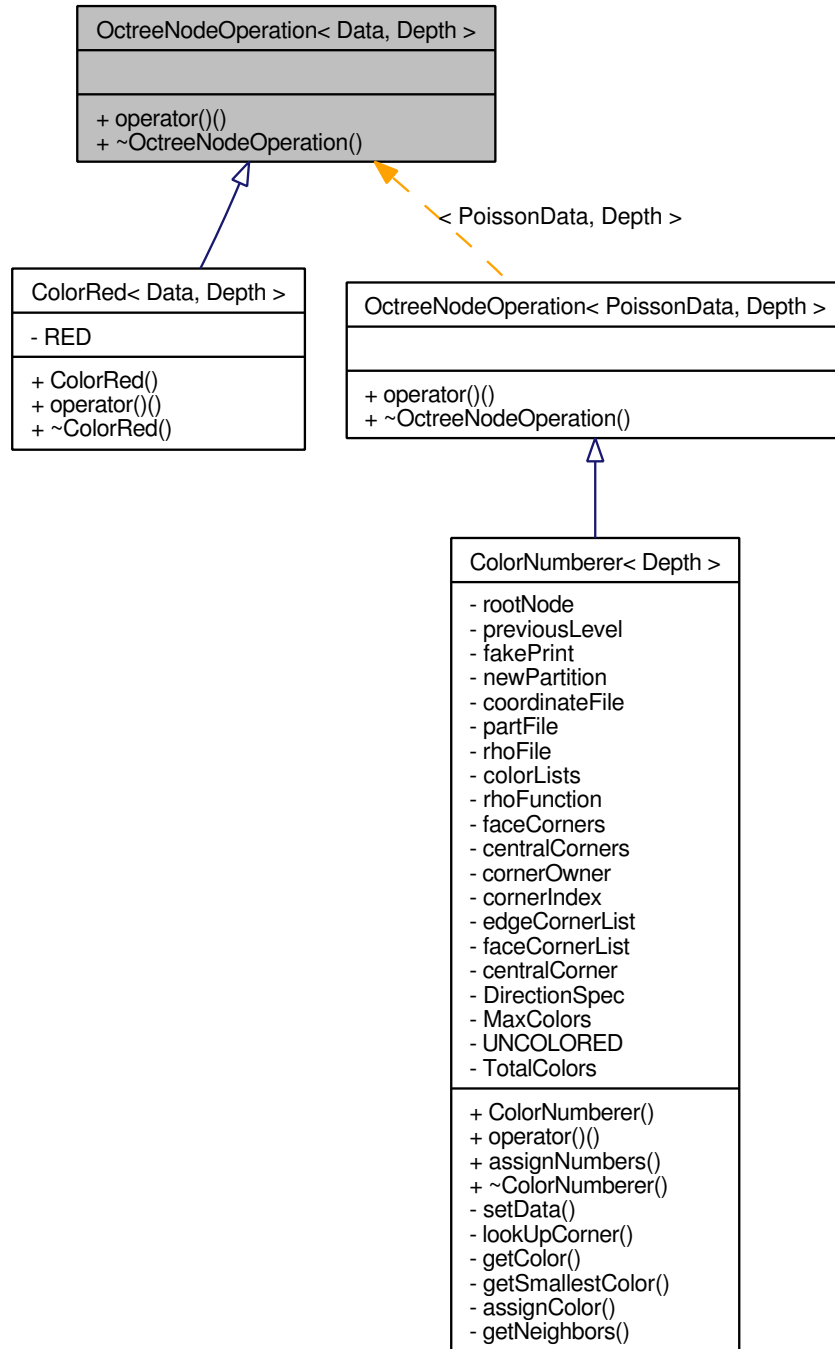
*Appendix A. GeomOct Reference*

**A.2.14 OctreeNodeOperation< Data, Depth > Class Template Reference**

```
#include <OctreeNode.h>
```

## Appendix A. *GeomOct Reference*

Inheritance diagram for OctreeNodeOperation< Data, Depth >:



## Appendix A. *GeomOct Reference*

### Public Member Functions

- virtual bool `operator()` (`OctreeNode< Data, Depth > &node`)=0

### Detailed Description

`template<typename Data, unsigned short Depth> class OctreeNodeOperation< Data, Depth >`

This class the the base class on which are built all the operations that can be performed on the Nodes. To make a new operation, use this as the base class and overload the `()` operator.

### Member Function Documentation

`template<typename Data, unsigned short Depth> virtual bool OctreeNodeOperation< Data, Depth >::operator() (OctreeNode< Data, Depth > & node) [pure virtual]`

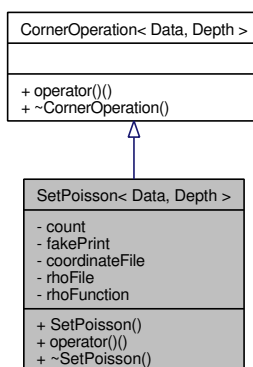
Overload this to make an operator for nodes.

Implemented in `ColorNumberer< Depth >`, and `ColorRed< Data, Depth >`.

### A.2.15 `SetPoisson< Data, Depth >` Class Template Reference

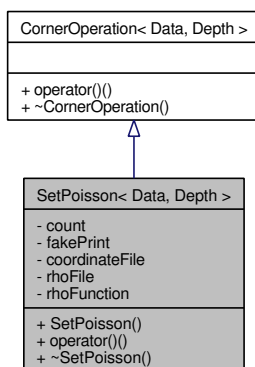
```
#include <SetPoisson.h>
```

Inheritance diagram for `SetPoisson< Data, Depth >`:



## Appendix A. *GeomOct* Reference

Collaboration diagram for `SetPoisson< Data, Depth >`:



### Public Member Functions

- `SetPoisson` (`double(*rhoFunc)(double, double, double)`, `std::string coordName=""`, `std::string rhoName=""`)
- virtual bool `operator()` (`Corner< Data, Depth > &node`)

### Detailed Description

```
template<typename Data, size_t Depth> class SetPoisson< Data,
Depth >
```

Simple functor to set the data at the grid point. Its purpose is to set the value of the variable coefficient at all the grid points.

### Constructor & Destructor Documentation

```
template<typename Data, size_t Depth> SetPoisson< Data, Depth
>::SetPoisson (double(*) (double, double, double) rhoFunc, std::string
coordName = "", std::string rhoName = "")
```

Given a function to compute rho at every point in the domain (`rhoFunc`), and a matrix name, create a functor which can iterate over all the corners and set the relevant values required at each corner.

If the coordinates are to be printed to a file, specify the `coordName` as the name of the file that must be written. If the values of the variable coefficient are



## Appendix A. *GeomOct* Reference

to be printed to a file, specify the rhoName as the name of the file that must be written. In the absence of these file names, no output will be generated.

This method accepts a pointer to a function. Read the C++ documentation to learn how to achieve this.

### Member Function Documentation

```
template<typename Data, size_t Depth> virtual bool SetPoisson<
Data, Depth >::operator() (Corner< Data, Depth > & node) [virtual]
```

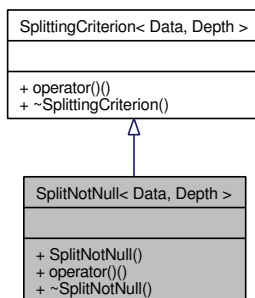
Set the value of the variable coefficient on this node according to its location. Also assign it a unique node number. If specified in the constructor, the coordinates are printed to a file, along with the node number. Also, if required, the values of the variable coefficients are also printed to file.

Implements [CornerOperation< Data, Depth >](#).

### A.2.16 SplitNotNull< Data, Depth > Class Template Reference

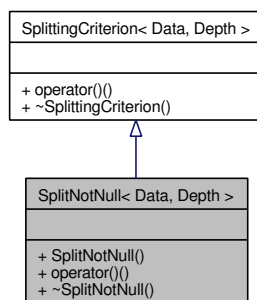
```
#include <Splitting.h>
```

Inheritance diagram for SplitNotNull< Data, Depth >:



## Appendix A. *GeomOct* Reference

Collaboration diagram for `SplitNotNull< Data, Depth >`:



### Public Member Functions

- `SplitNotNull()`
- virtual bool `operator()` (const `OctreeNode< Data, Depth >` &node)

### Detailed Description

`template<typename Data, unsigned short Depth> class SplitNotNull< Data, Depth >`

Split every node that does not have NULL as its Data. This is useful in the cases where every child has to be refined, and the children have some data.

While this class is useful, it is also a very good demonstration of how to write `SplittingCriterion` functors, and forms a great starting point for such functors.

### Constructor & Destructor Documentation

`template<typename Data, unsigned short Depth> SplitNotNull< Data, Depth >::SplitNotNull()`

Default constructor. No arguments, and nothing to do.

### Member Function Documentation

## Appendix A. *GeomOct Reference*

**template<typename Data, unsigned short Depth> bool SplitNotNull<Data, Depth >::operator() (const OctreeNode< Data, Depth > & *node*)**  
[virtual]

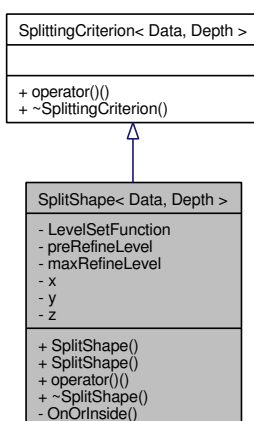
This function does the splitting. It is not meant to be called by the user. Rather, it is called by [OctreeNode](#) as a [SplittingCriterion](#). It examines the node, and if there is any data attached to this node, it chooses to split it. It relies on the [OctreeNode](#) creation making a new child have data NULL.

Implements [SplittingCriterion< Data, Depth >](#).

### A.2.17 SplitShape< Data, Depth > Class Template Reference

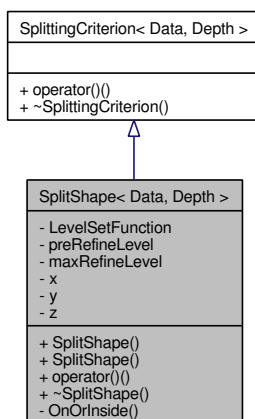
```
#include <Splitting.h>
```

Inheritance diagram for SplitShape< Data, Depth >:



## Appendix A. *GeomOct* Reference

Collaboration diagram for `SplitShape< Data, Depth >`:



### Public Member Functions

- [SplitShape](#) (`double(*levelFunc)(double, double, double)`, unsigned int `maxRefine=Depth`, unsigned int `preRefine=1`)
- [SplitShape](#) (`double(*levelFunc)(double, double, double)`, double `xEnd`, double `yEnd`, double `zEnd`, unsigned int `maxRefine=Depth`, unsigned int `preRefine=1`)
- virtual bool [operator\(\)](#) (`const OctreeNode< Data, Depth > &node`)

### Detailed Description

**template<typename Data, unsigned short Depth> class SplitShape< Data, Depth >**

Split the volume based on a level set function which returns a negative value if we are inside the interface, 0 if we are on the interface, and positive if we are outside the interface. This class takes the level set function as an argument to the constructor and splits based on that function.

This class is provided as a demonstration of the ease with which splitters can be written and forms a good starting point for splitters.

### Constructor & Destructor Documentation

```
template<typename Data, unsigned short Depth> SplitShape< Data,
Depth >::SplitShape (double(*) (double, double, double) levelFunc, un-
signed int maxRefine = Depth, unsigned int preRefine = 1)
```

Create a [SplitShape](#) object, by specifying a pointer to a level function that takes (x,y,z) coordinates, and an optional preRefine criteria.

The function is a level set signed distance function. It requires three doubles, which are the (x,y,z) values of the point. It returns a negative value or zero if the current point is inside the interface, and a positive  $> 0$  if the point is outside it. The interface divides the two fluids.

The preRefine criteria is to refine the tree a few times before the operator() is run on the nodes. This prerenement is carried out because initially there is just one node: the root. If the interface lies entirely in the original volume, then no refinement will take place, since all corners of the root node will be outside the interface. By default the preRefine is set to 1.

In this constructor, the limits of the cube are defined to be (0,0,0) - (1,1,1)

```
template<typename Data, unsigned short Depth> SplitShape< Data,
Depth >::SplitShape (double(*) (double, double, double) levelFunc,
double xEnd, double yEnd, double zEnd, unsigned int maxRefine
= Depth, unsigned int preRefine = 1)
```

This constructor allows specifies the end points of the volume as well. The first argument is the level set signed distance function, and the next three are the coordinates of the point farthest from the origin. The origin is the other end of the volume.

## Member Function Documentation

```
template<typename Data, unsigned short Depth> bool SplitShape<
Data, Depth >::operator() (const OctreeNode< Data, Depth > & node)
[virtual]
```

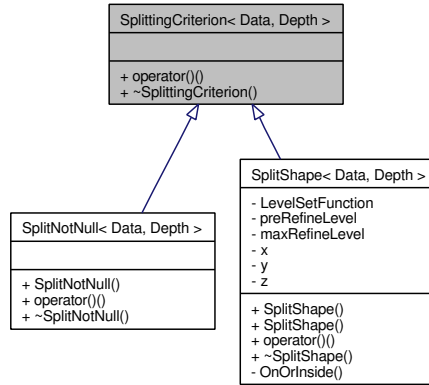
This method does the splitting. It is not meant to be called by the user. Rather, it is called by [OctreeNode](#) as a [SplittingCriterion](#).

Implements [SplittingCriterion< Data, Depth >](#).

### A.2.18 SplittingCriterion< Data, Depth > Class Template Reference

```
#include <Splitting.h>
```

Inheritance diagram for SplittingCriterion< Data, Depth >:



#### Public Member Functions

- virtual bool `operator()` (const `OctreeNode< Data, Depth > &node`)=0

#### Detailed Description

```
template<typename Data, unsigned short Depth> class
SplittingCriterion< Data, Depth >
```

This is the operation that can be accepted as criterion for splitting a node. In order to make an operation like this, extend `SplittingCriterion` and overload `operator()`. The operator returns true if that particular node is to be split, and false otherwise.

#### Member Function Documentation

```
template<typename Data, unsigned short Depth> virtual bool
SplittingCriterion< Data, Depth >::operator() (const OctreeNode<
Data, Depth > & node) [pure virtual]
```

The function called to decide if a split should be made. It is given a const reference to an `OctreeNode`. If that node is fit for splitting, then this method

## Appendix A. *GeomOct* Reference

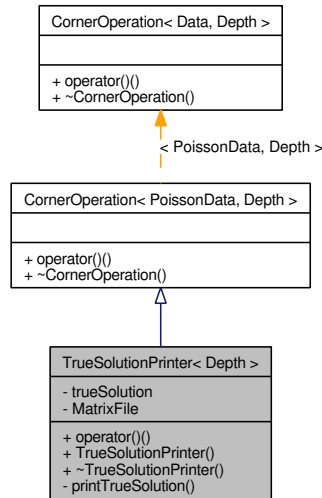
must return true, otherwise it must return false. This is a pure abstract method: all implementers must override it.

Implemented in [SplitShape< Data, Depth >](#), and [SplitNotNull< Data, Depth >](#).

### A.2.19 **TrueSolutionPrinter< Depth >** Class Template Reference

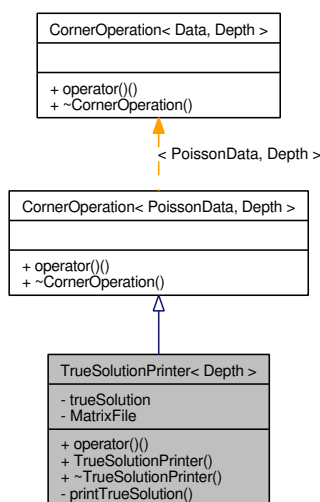
```
#include <TrueSolutionPrinter.h>
```

Inheritance diagram for TrueSolutionPrinter< Depth >:



## Appendix A. *GeomOct* Reference

Collaboration diagram for `TrueSolutionPrinter< Depth >`:



### Public Member Functions

- virtual bool `operator()` (`Corner< PoissonData, Depth > &myNode`)
- `TrueSolutionPrinter` (`double(*trueSolution)(double, double, double)`, `std::string matrixName`)

### Detailed Description

`template<size_t Depth> class TrueSolutionPrinter< Depth >`

Make a functor that will print out the true solution for this mesh.

The reason for the size of this class is to ensure that it is clean reusable code. Much of the flexibility derives from it. For example, the force function is specified outside this class which keeps the details of the force function separate from the implementation of this class.

The implementation of `operator()` here forms a very good demonstration of functors and `CornerOperations`.

### Constructor & Destructor Documentation



## Appendix A. *GeomOct* Reference

```
template<size_t Depth>      TrueSolutionPrinter<      Depth
>::TrueSolutionPrinter (double*)(double, double, double) trueS-
olution, std::string matrixName)
```

Create a new functor, which requires a pointer to a function that gives the true solution at every point. The second argument is the name of a file in which the true solution is written. This second argument is required.

### Member Function Documentation

```
template<size_t Depth>      bool      TrueSolutionPrinter<      Depth
>::operator() (Corner< PoissonData, Depth > & myNode) [virtual]
```

From each node, write down the true solution at this point. Forms an excellent study in the use of functors in modular code.

Implements [CornerOperation< PoissonData, Depth >](#).