Solving Poisson's Equation in High Dimensions by a Hybrid Monte-Carlo Finite Difference Method

by

Wilson Au

B.Sc., Simon Fraser University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN THE DEPARTMENT
OF
MATHEMATICS

SIMON FRASER UNIVERSITY
Fall 2006

APPROVAL

Name:	Wilson Au
Degree:	Master of Science
Title of thesis:	Solving Poisson's Equation in High Dimensions by a Hybrid Monte-Carlo Finite Difference Method
Examining Committee:	Dr. Ralf Wittenberg Chair
	Dr. Adam Oberman Senior Supervisor
	Dr. Steve Ruuth Supervisor
	Dr. JF Williams External Examiner
Date Approved:	September 1, 2006

Abstract

We introduce and implement a hybrid Monte-Carlo finite difference method for approximating the solution of Poisson's equation. This method solves smaller problems multiple times to collectively solve a larger main problem, when the solution of the main problem is unattainable by known regular direct and iterative methods. The method thereby resolves features that a single smaller problem may not. This hybrid Monte-Carlo finite difference method achieves second order accuracy on generic problems, and on problems with sharp features.

Keywords:
Finite difference method; high dimensions; parallel computing; Poisson's equation

Acknowledgments

I would like to thank my senior supervisor Dr. Adam Oberman for suggesting this interesting topic for my thesis. I would also like to thank Dr. Steve Ruuth, Dr. JF Williams, Dr. Ralf Wittenberg and Dr. Jim Verner for their valuable comments to improve the style of this thesis. Special thanks to my colleagues Thomas Humphries and Ryo Takei for their constructive comments and suggestions.

Last but not least, many thanks to my family and Vivian Chen. This would not have been possible without their continuous support, patience and encouragement throughout my bachelor's and master's degrees at Simon Fraser University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The aim of this thesis is to construct a hybrid Monte-Carlo finite difference method for the use in computing numerical solutions to Poisson's equation in multiple dimensions. This method solves smaller problems multiple times to collectively solve a larger, often high cost main problem.

A standard technique for solving Poisson's equation is by using the finite difference method, which is essentially equivalent to solving a large system of linear equations. The two standard classes of methods for solving system of equations are the direct methods and the iterative methods. The work required in solving a general system of N equations with N unknowns by direct methods is $O(N^3)$, whereas it is $O(N^2)$ by iterative methods. In general, direct methods require larger memory and more work, but are more robust. The iterative methods require less memory and work, but are also less robust. The drawback of both is the rapid increase in computational complexity as the number of dimensions increases, an effect known as the *curse of dimensionality*[1]. Table 1.1 from [17] gives a rough approximation to what dimensions might have been considered "very large" for direct methods over the years.

For high dimensions, the commonly used method is the Monte-Carlo method. A classic use of the Monte-Carlo method is for the evaluation of definite integrals, particularly multiple dimensional integrals with complicated boundary conditions. Moreover, the Monte-Carlo method can be used to compute the stochastic processes. Furthermore, it can be used to compute solutions of partial differential equations (PDEs), based on the well-known

[1] A term coined by Richard Bellman to describe the rapid growth of volume as the number of dimensions increases.

Table 1.1: History of matrix computations over the years.

1950:	N = 20
1965:	N = 200
1980:	N = 2000
1995:	N = 20000

Feynman-Kac formula and Itô's formula. They are powerful tools that allow one to represent the solutions of elliptic and parabolic PDEs as the expected values over a stochastic processes under some assumptions (see [3], [5]). Unfortunately, the Monte-Carlo method required M times as much work to reduces the numerical approximation by a factor of $1/\sqrt{M}$, where M is the number of simulations. This property holds independently of the number of dimensions, but the rate of convergence is still the fatal drawback.

These two methods offer a tradeoff. The first gives good accuracy if we can solve a large linear system. The second requires only inexpensive (unit cost) simulation, but many of them to obtain comparable accuracy. The hybrid Monte-Carlo finite difference method introduced here aims to find a compromise between the two.

In this thesis, we are interested in solving what first appears to be a simple problem, namely Poisson's equation over a d-dimensional domain Ω.

PDE version:

$$\begin{cases} -\triangle u(\boldsymbol{x}) = f(\boldsymbol{x}), & \forall \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) = g(\boldsymbol{x}), & \forall \boldsymbol{x} \in \partial\Omega \end{cases} \tag{1.1}$$

Stochastic version:

$$u(\boldsymbol{x}) = E_{\boldsymbol{x}}\left[g(W_{\tau_{\partial\Omega}}) + \frac{1}{2}\int_0^{\tau_{\partial\Omega}} f(W_t)\,dt\right], \tag{1.2}$$

where $(W_t,\ t \geq 0)$ is a Brownian path and $\tau_{\partial\Omega} = \inf(t : W_t \in \partial\Omega)$, is the stopping time. The solution at each point is given as an average of a functional over the Brownian paths (see [4], [9], [11], [10], [13]). In this thesis, we will not go into detail about the stochastic version. Everything we do can be thought of as a special kind of finite difference method. The stochastic version is the motivation of the hybrid Monte-Carlo finite method; it gives a different way of thinking of the problem.

The concept of the hybrid Monte-Carlo finite difference method involves solving an approximate solution on a coarse grid, then making a refinement of it using a fine grid. Although the method has the appearance of a multigrid method, because we are solving on two different scales, it is designed to generalize to nonlinear equations which can be written as expectations of stochastic processes, as in (1.2). We hope to generalize this method to solving nonlinear elliptic PDEs.

A summary of the contents of the chapters is as follows:

The first half of Chapter 2 consists of a quick review of the discretization of the one dimensional Poisson's equation with Dirichlet boundary conditions on a uniform grid, and the matrix representation of this discretization. The second half is on a non-uniform grid.

Chapter 3 begins with a discussion of the discretization of the multiple dimensional Poisson's equation and the corresponding matrix representation. It then touches briefly on the difficulty of solving problems in high dimensions.

In Chapter 4, we introduce a hybrid Monte-Carlo finite difference method, with numerical results presented for the one, two, three and four dimensional problems.

In Chapter 5, we investigate the rate of convergence in one, two and three dimensions. All the data throughout this thesis is collected using an Apple Power Mac G5 with dual 2.5Ghz processors and 2GB RAM.

Finally, Chapter 6 is divided into two sections: The first section is a summary, followed by a discussion on future work.

Chapter 2

One Dimensional Poisson's Equation

The purpose of this chapter is to review the finite difference discretization for the one dimensional Poisson's equation. Let us first recall the one dimensional Poisson's equation with Dirichlet boundary conditions on the domain $[a, b] \subset \mathbb{R}$:

$$\begin{cases} -u''(x) = f(x), & \forall x \in (a, b) \\ u(a) = g_1, \quad u(b) = g_N. \end{cases} \tag{2.1}$$

2.1 On a Uniform Grid

Finite difference discretization consists of replacing each derivative by a difference quotient. The most standard difference quotient for the second derivative is the centered second order difference, which could be derived from the Taylor expansion of $u(x - \Delta x)$ and $u(x + \Delta x)$:

$$u(x - \Delta x) = u(x) - u'(x)\Delta x + \frac{1}{2}u''(x)\Delta x^2 - \frac{1}{6}u'''(x)\Delta x^3 + O(\Delta x^4), \tag{2.2}$$

$$u(x + \Delta x) = u(x) + u'(x)\Delta x + \frac{1}{2}u''(x)\Delta x^2 + \frac{1}{6}u'''(x)\Delta x^3 + O(\Delta x^4). \tag{2.3}$$

By adding (2.2) and (2.3), subtracting $2u(x)$ and dividing by Δx^2, we deduce that

$$u''(x) = \frac{u(x + \Delta x) - 2u(x) + u(x - \Delta x)}{\Delta x^2} + O(\Delta x^2). \tag{2.4}$$

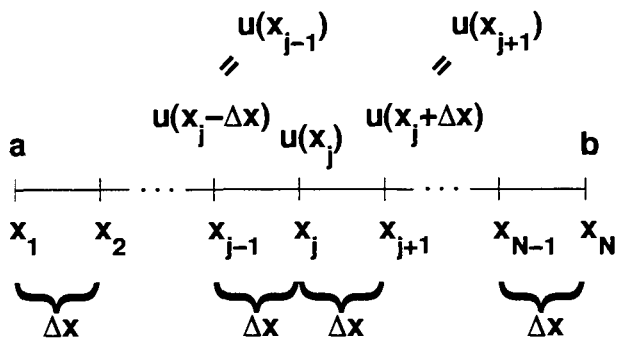

Figure 2.1: Uniform grid.

The next step is to choose an integer N, the number of grid points, and define the grid size $\Delta x = (b-a)/(N-1)$. This partitioned the domain $[a,b]$ into $(N-1)$ equal parts of length Δx. We define a particular grid point x_j by

$$x_j = a + (j-1)\Delta x, \qquad j = 1, 2, ..., N \tag{2.5}$$

and the value u_j by

$$u_j \simeq u(x_j). \tag{2.6}$$

For example, if $u(x)$ is approximated by u_j, then $u(x+\Delta x)$ and $u(x-\Delta x)$ are approximated by u_{j+1} and u_{j-1} respectively. Then (2.4) becomes,

$$u''(x_j) = \frac{u_{j+1} - 2u_j + u_{j-1}}{\Delta x^2} + O(\Delta x^2). \tag{2.7}$$

This scheme is $O(\Delta x^2)$ accurate; in other words, $\frac{u_{j+1} - 2u_j + u_{j-1}}{\Delta x^2}$ approximates $u''(x_j)$ up to terms proportional to Δx^2.

When we replace $u''(x)$ by (2.7) in (2.1), then the discretized one dimensional Poisson's equation with Dirichlet boundary conditions is:

$$\begin{cases} \dfrac{-u_{j+1}+2u_j-u_{j-1}}{\Delta x^2} = f_j, & j = 2,3,...,N-1 \\ u_1 = g_1, \quad u_N = g_N. \end{cases} \tag{2.8}$$

The domain $[a,b]$ is partitioned into $(N-1)$ equal parts by N grid points, for which $j=1$ and $j=N$ are the boundary points where the solution is given, and $j=2,...,(N-1)$ are the interior points where the solution is to be computed. In order to solve this discretized one dimensional Poisson's equation, we need $(N-2)$ equations for the $(N-2)$ unknowns. We can obtain these $(N-2)$ equations by expanding (2.8) from $j=2$ to $j=(N-1)$, and together with the boundary points. This leads to a linear system of equations in the form of $A\mathbf{u}=\mathbf{b}$, where A is the matrix representation of the one dimensional Poisson's equation with Dirichlet boundary conditions:

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} & \frac{-1}{\Delta x^2} & 0 & & & 0 \\ 0 & \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} & \frac{-1}{\Delta x^2} & 0 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} & \frac{-1}{\Delta x^2} & 0 \\ \vdots & & & 0 & \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} & \frac{-1}{\Delta x^2} \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix}, \tag{2.9}$$

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_{N-3} \\ u_{N-2} \\ u_{N-1} \\ u_N \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} g_1 \\ f_2 \\ f_3 \\ f_4 \\ \vdots \\ f_{N-3} \\ f_{N-2} \\ f_{N-1} \\ g_N \end{bmatrix}. \tag{2.10}$$

For numerical purposes, when solving such a system, we often rewrite the boundary conditions as

$$\begin{aligned} \frac{u_1}{\Delta x^2} &= \frac{g_1}{\Delta x^2}, \\ \frac{u_N}{\Delta x^2} &= \frac{g_N}{\Delta x^2}. \end{aligned} \tag{2.11}$$

This ensures a lower condition number[1]. Notice that A is sparse[2], tridiagonal[3] and positive definite[4]. There are many methods that have been designed to solve these types of linear system for the vector $\mathbf{u}$.

2.2 On a Non-uniform Grid

For non-uniform grids, the corresponding centered second order difference can also be derived from the Taylor expansion of $u(x-\Delta x_L)$ and $u(x+\Delta x_R)$, except now, Δx_L is not necessary equal to Δx_R.

$$u(x-\Delta x_L) = u(x) - u'(x)\Delta x_L + \frac{1}{2}u''(x)\Delta x_L^2 - \frac{1}{6}u'''(x)\Delta x_L^3 + O(\Delta x_L^4), \tag{2.12}$$

$$u(x+\Delta x_R) = u(x) + u'(x)\Delta x_R + \frac{1}{2}u''(x)\Delta x_R^2 + \frac{1}{6}u'''(x)\Delta x_R^3 + O(\Delta x_R^4). \tag{2.13}$$

After multiplying Δx_R^2 to (2.12) and Δx_L^2 to (2.13), taking the sum and subtract $(\Delta x_L + \Delta x_R)u(x)$; then dividing both by $\frac{1}{2}(\Delta x_L \Delta x_R^2 + \Delta x_L^2 \Delta x_R)$ to deduce:

$$u''(x) = \frac{\Delta x_L u(x+\Delta x_R) - (\Delta x_L + \Delta x_R)u(x) + \Delta x_R u(x-\Delta x_L)}{\frac{1}{2}(\Delta x_L \Delta x_R^2 + \Delta x_L^2 \Delta x_R)} + O(\Delta x_L - \Delta x_R). \tag{2.14}$$

Define a particular grid point x_j by

$$x_j = a + \sum_{k=1}^{j-1} \Delta x_k, \qquad j = 1, 2, ..., N \tag{2.15}$$

[1] The condition number of matrix A measures the number of digits lost in solving a linear system with that matrix. It is defined by $\|A\|\|A^{-1}\|$. A problem with a low condition number is said to be well-conditioned, whereas a problem with a high condition number is said to be ill-conditioned.

[2] A is sparse if most of the elements of A are zero.

[3] A is tridiagonal matrix, if A is square matrix with nonzero elements only on the diagonal, subdiagonal and superdiagonal.

[4] A is positive definite if $\mathbf{x}^T A\mathbf{x} > 0$ for all $\mathbf{x} \neq 0$.

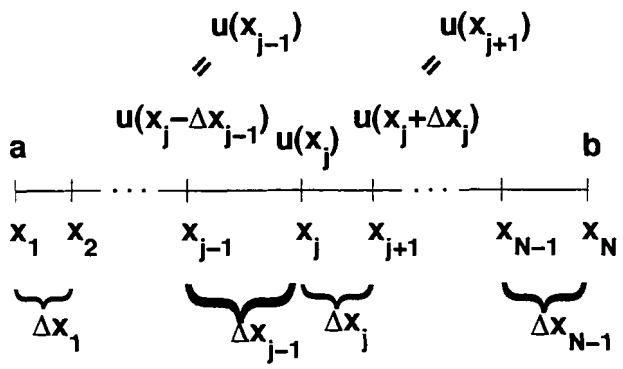

Figure 2.2: Non-uniform grid.

where $\Delta x_j = x_{j+1} - x_j$, and we approximate the value u_j by (2.6). Then (2.14) becomes,

$$u''(x_j) = \frac{\Delta x_{j-1}u_{j+1} - (\Delta x_{j-1} + \Delta x_j)u_j + \Delta x_j u_{j-1}}{\frac{1}{2}(\Delta x_{j-1}\Delta x_j^2 + \Delta x_{j-1}^2 \Delta x_j)} + O(\Delta x_{j-1} - \Delta x_j). \tag{2.16}$$

This is the centered second order difference for a non-uniform grid. In general, this scheme is first order accurate unless the mesh is smoothly graded, or $\Delta x_L = \Delta x_R$, in which case it would be second order accurate as described in Section 2.1.

Replace $u''(x)$ by (2.16) in (2.1), and the discretized one dimensional Poisson's equation with Dirichlet boundary conditions on a non-uniform grid becomes:

$$\begin{cases} \dfrac{-\Delta x_{j-1}u_{j+1} + (\Delta x_{j-1} + \Delta x_j)u_j - \Delta x_j u_{j-1}}{\frac{1}{2}(\Delta x_{j-1}\Delta x_j^2 + \Delta x_{j-1}^2 \Delta x_j)} = f_j, & j = 2, 3, ..., N-1 \\ u_1 = g_1, \qquad u_N = g_N. \end{cases} \tag{2.17}$$

As in Section 2.1, we need $(N-2)$ equations which are all obtained by expanding (2.17), from $j = 2$ to $j = (N-1)$; together with the boundary points, this leads to a linear system of equations in the form of $A\mathbf{u} = \mathbf{b}$, where A is sparse, tridiagonal and positive definite:

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ \frac{-2\Delta x_2}{d_2} & \frac{2\Delta_2}{d_2} & \frac{-2\Delta x_1}{d_2} & 0 & & & 0 \\ 0 & \frac{-2\Delta x_3}{d_3} & \frac{2\Delta_3}{d_3} & \frac{-2\Delta x_2}{d_3} & 0 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & & 0 & \frac{-2\Delta x_{N-2}}{d_{N-2}} & \frac{2\Delta_{N-2}}{d_{N-2}} & \frac{-2\Delta x_{N-3}}{d_{N-2}} & 0 \\ 0 & & & 0 & \frac{-2\Delta x_{N-1}}{d_{N-1}} & \frac{2\Delta_{N-1}}{d_{N-1}} & \frac{-2\Delta x_{N-2}}{d_{N-1}} \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix}, \tag{2.18}$$

where $\Delta x_j = x_{j+1} - x_j$, $\Delta_j = \Delta x_{j-1} + \Delta x_j$ and $d_j = \Delta x_{j-1}\Delta x_j^2 + \Delta x_{j-1}^2 \Delta x_j$. The vectors $\mathbf{u}$ and $\mathbf{b}$ are the same as in (2.10).

For numerical purposes, we often rewrite the boundary conditions as

$$\begin{aligned} \frac{\Delta_{floor(N/2)}}{d_{floor(N/2)}} u_1 &= \frac{\Delta_{floor(N/2)}}{d_{floor(N/2)}} g_1, \\ \frac{\Delta_{floor(N/2)}}{d_{floor(N/2)}} u_N &= \frac{\Delta_{floor(N/2)}}{d_{floor(N/2)}} g_N. \end{aligned} \tag{2.19}$$

In later chapters, we use this idea of non-uniform grid to perform grid shifting. The grid shifting shifts the interior grid point to a given direction by one unit (in term of Δx). Thereby the distance between each interior grid point is unchange, which is Δx. The distance between the interior grid point and the boundary point is either larger than or smaller than Δx. This can be thought as having a interior uniform grid cell with non-uniform boundary grid cell. By rewriting the boundary conditions as (2.19), this ensures the diagonal is approximately the same value, therefore resulting in a lower condition number.

Chapter 3

Multiple Dimensional Poisson's Equation

3.1 In Two Dimensions

Consider the two dimensional Poisson's equation

$$\begin{cases} -\triangle u(\boldsymbol{x}) = -\dfrac{\partial^2}{\partial x^2}u(\boldsymbol{x}) - \dfrac{\partial^2}{\partial y^2}u(\boldsymbol{x}) = f(\boldsymbol{x}), & \forall \boldsymbol{x} = (x, y) \in \Omega \\ u(\boldsymbol{x}) = g(\boldsymbol{x}), & \forall \boldsymbol{x} \in \partial\Omega \end{cases} \tag{3.1}$$

where $\Omega = (\alpha_0, \alpha_1) \times (\beta_0, \beta_1)$.

Similar to Section 2.1, we choose integers N_x and N_y and define the grid size $\Delta x = (\alpha_1 - \alpha_0)/(N_x - 1)$ and $\Delta y = (\beta_1 - \beta_0)/(N_y - 1)$, which partitions $[\alpha_0, \alpha_1]$ and $[\beta_0, \beta_1]$ into $(N_x - 1)$ and $(N_y - 1)$ equal parts of length Δx and Δy respectively. Define the grid point (x_i, y_j) by

$$x_i = \alpha_0 + (i-1)\Delta x, \qquad i = 1, 2, ..., N_x \tag{3.2}$$

and

$$y_j = \beta_0 + (j-1)\Delta y, \qquad j = 1, 2, ..., N_y \tag{3.3}$$

and the value $u_{i,j}$ by

$$u_{i,j} \simeq u(x_i, y_j). \tag{3.4}$$

The centered second order differences are

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + O(\Delta x^2), \tag{3.5}$$

$$\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} + O(\Delta y^2). \tag{3.6}$$

The discretized two dimensional Poisson's equation with Dirichlet boundary conditions is now

$$\begin{cases} -\frac{u_{i-1,j}}{\Delta x^2} - \frac{u_{i,j-1}}{\Delta y^2} + 2\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)u_{i,j} - \frac{u_{i,j+1}}{\Delta y^2} - \frac{u_{i+1,j}}{\Delta x^2} = f_{i,j}, \\ \qquad\qquad \text{for } i = 2,3,...,N_x - 1 \\ \qquad\qquad \text{and } j = 2,3,...,N_y - 1 \\ u_{1,j} = g_{1,j} \quad \text{and} \quad u_{N_x,j} = g_{N_x,j}, \quad \text{for } j = 1,2,...,N_y \\ u_{i,1} = g_{i,1} \quad \text{and} \quad u_{i,N_y} = g_{i,N_y}. \quad \text{for } i = 1,2,...,N_x \end{cases} \tag{3.7}$$

The domain is partitioned into $(N_x - 1)(N_y - 1)$ grid points of which $(2N_x + 2N_y - 4)$ are the boundary points where the solution is given, and $(N_x - 2)(N_y - 2)$ are the interior points where the solution is unknown. In order to solve the discretized two dimensional Poisson's equation, we need $(N_x - 2)(N_y - 2)$ equations. We obtain those by expanding (3.7) for each of the interior points from top to bottom and left to right, see Figure 3.1.

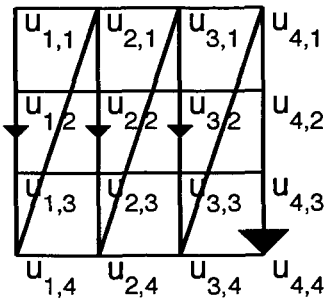

Figure 3.1: Example of reshaping a 2d matrix into a vector.

Together with the boundary values, this leads to a linear system in the form of $A\mathbf{u} = \mathbf{b}$, where

$$A = \begin{bmatrix} I & 0 & 0 & \cdots & 0 \\ -I_x & B_y & -I_x & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -I_x & B_y & -I_x \\ 0 & \cdots & 0 & 0 & I \end{bmatrix}, \tag{3.8}$$

$$\mathbf{u} = \begin{bmatrix} u_{1,1} \\ \vdots \\ u_{1,N_y} \\ u_{2,1} \\ u_{2,2} \\ \vdots \\ u_{N_x-1,N_y-1} \\ u_{N_x-1,N_y} \\ u_{N_x,1} \\ \vdots \\ u_{N_x,N_y} \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} g_{1,1} \\ \vdots \\ g_{1,N_y} \\ g_{2,1} \\ f_{2,2} \\ \vdots \\ f_{N_x-1,N_y-1} \\ g_{N_x-1,N_y} \\ g_{N_x,1} \\ \vdots \\ g_{N_x,N_y} \end{bmatrix}, \tag{3.9}$$

with identity matrix I and B_y and I_x as follows:

$$B_y = \begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ \frac{-1}{\Delta y^2} & 2S_{xy} & \frac{-1}{\Delta y^2} & 0 & & & 0 \\ 0 & \frac{-1}{\Delta y^2} & 2S_{xy} & \frac{-1}{\Delta y^2} & 0 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & & 0 & \frac{-1}{\Delta y^2} & 2S_{xy} & \frac{-1}{\Delta y^2} & 0 \\ 0 & & & 0 & \frac{-1}{\Delta y^2} & 2S_{xy} & \frac{-1}{\Delta y^2} \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix}, \quad I_x = \begin{bmatrix} 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \frac{1}{\Delta x^2} & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & \frac{1}{\Delta x^2} & 0 \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix}, \tag{3.10}$$

where $S_{xy} = \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}$.

Similar to Chapter 2, for numerical purposes, we often rewrite the boundary conditions as

$$\begin{aligned} S_{xy} \cdot u_{1,j} &= S_{xy} \cdot g_{1,j} \quad \text{and} \quad S_{xy} \cdot u_{N_x,j} = S_{xy} \cdot g_{N_x,j}, \quad \text{for } j = 1, 2, ..., N_y \\ S_{xy} \cdot u_{i,1} &= S_{xy} \cdot g_{i,1} \quad \text{and} \quad S_{xy} \cdot u_{i,N_y} = S_{xy} \cdot g_{i,N_y}. \quad \text{for } i = 1, 2, ..., N_x \end{aligned} \tag{3.11}$$

This ensures a lower condition number. Notice that A is a sparse, positive definite and block[1] matrix consisting of square sub-matrices B_y and I_x. I_x is a diagonal matrix, and B_y is a tridiagonal matrix. So matrix A is a positive definite matrix with 5 non-zero diagonals.

3.2 In Three Dimensions

Consider the three dimensional Poisson's equation

$$\begin{cases} -\triangle u(\boldsymbol{x}) = -\dfrac{\partial^2}{\partial x^2}u(\boldsymbol{x}) - \dfrac{\partial^2}{\partial y^2}u(\boldsymbol{x}) - \dfrac{\partial^2}{\partial z^2}u(\boldsymbol{x}) = f(\boldsymbol{x}), & \forall \boldsymbol{x} = (x, y, z) \in \Omega \\ u(\boldsymbol{x}) = g(\boldsymbol{x}), & \forall \boldsymbol{x} \in \partial\Omega \end{cases} \tag{3.12}$$

where $\Omega = (\alpha_0, \alpha_1) \times (\beta_0, \beta_1) \times (\gamma_0, \gamma_1)$.

We choose integers N_x, N_y and N_z and define the grid size as $\Delta x = (\alpha_1 - \alpha_0)/(N_x - 1)$, $\Delta y = (\beta_1 - \beta_0)/(N_y - 1)$ and $\Delta z = (\gamma_1 - \gamma_0)/(N_z - 1)$, which partitions $[\alpha_0, \alpha_1]$, $[\beta_0, \beta_1]$ and $[\gamma_0, \gamma_1]$ into $(N_x - 1)$, $(N_y - 1)$ and $(N_z - 1)$ equal parts of length Δx, Δy and Δz respectively. Define the grid point (x_i, y_j, z_k) by

$$x_i = \alpha_0 + (i - 1)\Delta x, \qquad i = 1, 2, ..., N_x \tag{3.13}$$

$$y_j = \beta_0 + (j - 1)\Delta y, \qquad j = 1, 2, ..., N_y \tag{3.14}$$

and

$$z_k = \gamma_0 + (k - 1)\Delta z, \qquad k = 1, 2, ..., N_z \tag{3.15}$$

and the value $u_{i,j,k}$ by

$$u_{i,j,k} \simeq u(x_i, y_j, z_k). \tag{3.16}$$

[1] A block matrix is a matrix that is defined by partitioning it into smaller matrices.

The centered second order differences are

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j,k} = \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{\Delta x^2} + O(\Delta x^2), \tag{3.17}$$

$$\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j,k} = \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{\Delta y^2} + O(\Delta y^2), \tag{3.18}$$

$$\left(\frac{\partial^2 u}{\partial z^2}\right)_{i,j,k} = \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{\Delta z^2} + O(\Delta z^2). \tag{3.19}$$

The discretized three dimensional Poisson's equation is

$$\begin{cases} -\frac{u_{i,j,k-1}}{\Delta z^2} - \frac{u_{i-1,j,k}}{\Delta x^2} - \frac{u_{i,j-1,k}}{\Delta y^2} + 2\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)u_{i,j,k} \\ \qquad -\frac{u_{i,j+1,k}}{\Delta y^2} - \frac{u_{i+1,j,k}}{\Delta x^2} - \frac{u_{i,j,k+1}}{\Delta z^2} = f_{i,j,k}, \\ \qquad\qquad \text{for } i = 2,3,...,N_x - 1 \\ \qquad\qquad \text{and } j = 2,3,...,N_y - 1 \\ \qquad\qquad \text{and } k = 2,3,...,N_z - 1 \\ u_{1,j,k} = g_{1,j,k} \text{ and } u_{N_x,j,k} = g_{N_x,j,k}, \text{ for } j = 1,2,...,N_y \text{ and } k = 1,2,...,N_z \\ u_{i,1,k} = g_{i,1,k} \text{ and } u_{i,N_y,k} = g_{i,N_y,k}, \text{ for } i = 1,2,...,N_x \text{ and } k = 1,2,...,N_z \\ u_{i,j,1} = g_{i,j,1} \text{ and } u_{i,j,N_z} = g_{i,j,N_z}. \text{ for } i = 1,2,...,N_x \text{ and } j = 1,2,...,N_y \end{cases} \tag{3.20}$$

We can obtain all the necessary equations to solve this problem by expanding (3.20) from back to front and left to right and top to bottom; see Figure 3.2.

This leads to a linear system of the form $A\mathbf{u} = \mathbf{b}$.

$$\mathbf{A} = \begin{bmatrix} I & 0 & 0 & \cdots & 0 \\ -I_z & B_{xy} & -I_z & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -I_z & B_{xy} & -I_z \\ 0 & \cdots & 0 & 0 & I \end{bmatrix}, \tag{3.21}$$

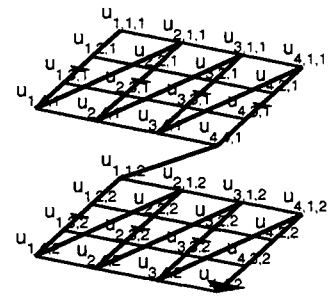

Figure 3.2: Example of reshaping a 3d matrix into a vector.

$$\mathbf{u} = \begin{bmatrix} u_{1,1,1} \\ \vdots \\ u_{N_x,N_y,1} \\ \vdots \\ u_{2,1,2} \\ u_{2,2,2} \\ \vdots \\ u_{N_x-1,N_y-1,N_z-1} \\ u_{N_x,1,N_z-1} \\ \vdots \\ u_{1,1,N_z} \\ \vdots \\ u_{N_x,N_y,N_z} \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} g_{1,1,1} \\ \vdots \\ g_{N_x,N_y,1} \\ \vdots \\ g_{2,1,2} \\ f_{2,2,2} \\ \vdots \\ f_{N_x-1,N_y-1,N_z-1} \\ g_{N_x,1,N_z-1} \\ \vdots \\ g_{1,1,N_z} \\ \vdots \\ g_{N_x,N_y,N_z} \end{bmatrix}, \tag{3.22}$$

with identity matrix I and B_{xy} and I_z as follows:

$$B_{xy} = \begin{bmatrix} I & 0 & 0 & \cdots & 0 \\ -I_x & B_y & -I_x & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -I_x & B_y & -I_x \\ 0 & \cdots & 0 & 0 & I \end{bmatrix}, \quad I_z = \begin{bmatrix} 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \frac{1}{\Delta z^2} & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & \frac{1}{\Delta z^2} & 0 \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix}, \tag{3.23}$$

with B_y and I_x as follows:

$$B_y = \begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ \frac{-1}{\Delta y^2} & 2S_{xyz} & \frac{-1}{\Delta y^2} & 0 & & & 0 \\ 0 & \frac{-1}{\Delta y^2} & 2S_{xyz} & \frac{-1}{\Delta y^2} & 0 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & & 0 & \frac{-1}{\Delta y^2} & 2S_{xyz} & \frac{-1}{\Delta y^2} & 0 \\ 0 & & & 0 & \frac{-1}{\Delta y^2} & 2S_{xyz} & \frac{-1}{\Delta y^2} \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix}, \tag{3.24}$$

$$I_x = \begin{bmatrix} 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \frac{1}{\Delta x^2} & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & 0 & \frac{1}{\Delta x^2} & 0 \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix},$$

where $S_{xyz} = \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}$.

Similarly, for numerical purposes, we often rewrite the boundary conditions as

$$\begin{aligned} S_{xyz} \cdot u_{1,j,k} = S_{xyz} \cdot g_{1,j,k} \text{ and } S_{xyz} \cdot u_{N_x,j,k} = S_{xyz} \cdot g_{N_x,j,k}, \\ \text{for } j = 1,2,...,N_y \text{ and } k = 1,2,...,N_z \\ S_{xyz} \cdot u_{i,1,k} = S_{xyz} \cdot g_{i,1,k} \text{ and } S_{xyz} \cdot u_{i,N_y,k} = S_{xyz} \cdot g_{i,N_y,k}, \\ \text{for } i = 1,2,...,N_x \text{ and } k = 1,2,...,N_z \\ S_{xyz} \cdot u_{i,j,1} = S_{xyz} \cdot g_{i,j,1} \text{ and } S_{xyz} \cdot u_{i,j,N_z} = S_{xyz} \cdot g_{i,j,N_z}. \\ \text{for } i = 1,2,...,N_x \text{ and } j = 1,2,...,N_y \end{aligned} \tag{3.25}$$

This ensures a lower condition number. Notice that matrix A is a block matrix consisting of square sub-matrices B_{xy} and I_z. I_z is a diagonal matrix, and B_{xy} is again a block matrix consisting of square sub-matrices B_x and I_y. I_y is a diagonal matrix, and B_x is a tridiagonal matrix. So this matrix A is positive definite with two levels of block matrix structure, and it has 7 non-zero diagonals.

3.3 In Higher Dimensions

In this section, we use the ideas from Sections 3.1 and 3.2 to generalize to higher dimensions.

In one dimension, matrix A is a tridiagonal positive definite matrix. In two dimensions, matrix A is a positive definite block matrix consisting of tridiagonal and diagonal matrices. Hence, A is a definite matrix with 5 non-zero diagonals. In three dimensions, matrix A is again a positive definite block matrix consisting of another substructure of block matrices and diagonal matrices, and the structure of the inner block matrices are same as the two dimensional case. Matrix A is said to have two levels of blocks. Hence it is a positive definite matrix with 7 non-zero diagonals and two levels of block matrix structure.

In d-dimensions, we could also expand the discretized d-dimensional Poisson's equation into a linear system in the form of $A\mathbf{u} = \mathbf{b}$; this can be done by reshaping this d-dimensional domain into a vector. Matrix A will have $(d-1)$ levels of block, and its structure will be a positive definite sparse matrix with $(2d+1)$ non-zero diagonals, and with size equal to the product of the number of grid points in every directions. Notice that, as the number of dimensions increases, the size of the matrix will increase, and consequently, many known methods will fail to compute in a sufficient period of time. This is the difficulty in solving high dimensional problems.

Chapter 4

Hybrid Monte-Carlo Finite Difference Method

The main advantage for using a hybrid Monte-Carlo finite difference method is the ability to solve a larger main problem collectively by solving smaller problems multiple times. The method thereby resolves features that a single smaller problem may not.

The motivation of the hybrid Monte-Carlo finite difference method is from the stochastic version; it can be thought as using the traditional Monte-Carlo method. We will motivate this hybrid Monte-Carlo finite difference method by an example. Consider Figure 4.1, the goal is to obtain the solution at the "*star*". By traditional Monte-Carlo method, we take some Brownian paths from the *star* until they reach the boundaries, and the solution is given as an average of a functional over those Brownian paths. For the hybrid Monte-Carlo finite difference method, unlike the traditional Monte-Carlo method, which take some Brownian paths from the *star* to the boundaries, we take a smaller scale Brownian paths from the *star* to the "*circles*"; this is same as applying the Monte-Carlo method to the *star* on the fine grid. For each *circle*, we take a larger scale Brownian paths until they reach the boundaries; this is same as applying the Monte-Carlo method from those *circles* on the coarse grid. Notice that, for those *circles* that lie on the coarse grid (see Figure 4.2), we can directly take Brownian paths from those points. For those *circles* that do not lie on the coarse grid, we cannot directly take Brownian paths from those points; we need to shift the coarse grid first, then take Brownian paths (see Figure 4.3). Recall from Chapter 1; from the finite difference method, the solution is obtained by solving a system of linear equations. From the

Monte-Carlo method, the solution at each point is given as an average of a functional over the Brownian paths. The previous two statements are the PDE version and the stochastic version of the solution of Poisson's equation; hence they are equivalent. We replace each of the Monte-Carlo method by the finite difference method. This hybrid Monte-Carlo finite difference method can be thought of as a special kind of finite difference method, which motivate by the traditional Monte-Carlo method.

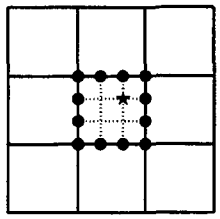

Figure 4.1: Take Brownian paths from "star" until it reaches the coarse grid.

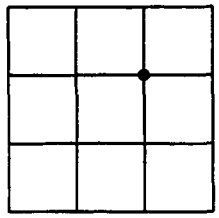

Figure 4.2: Example for a point that lies on the coarse grid.

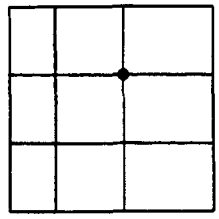

Figure 4.3: Example for a point that does not lies on the coarse grid.

The idea behind the hybrid Monte-Carlo finite difference method is that it can break the problem into two steps; the first step consists of shifting the coarse grid in different directions, the shifted grids representing different Brownian paths taken; the second step consists of solving smaller systems using a regular finite difference method multiple times collectively to solve a larger system.

4.1 Algorithm

The algorithm is as follows:

1. *Initialize*: Define two sets of grids, namely the coarse grid and the fine grid. Let the coarse grid have n_c points, with grid size DX, and the fine grid have n_f points, with grid size dx. Each space between consecutive coarse grid points contains a fine grid. This is equivalent of having a refined grid with n_{ref} points, where $n_{ref} = (n_c - 1)(n_f - 1) + 1$. In general, we choose $n_f = n_c$.

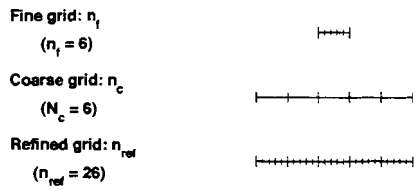

Figure 4.4: Example of a coarse grid, fine grid and refined grid in 1d.

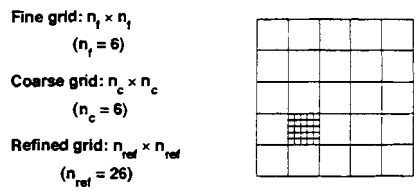

Figure 4.5: Example of a coarse grid, fine grid and refined grid in 2d.

2. *First step*: Solve the problem on coarse grid using the finite difference method. Then shift the coarse grid such that it lies on the boundary of the fine grid, and solve this "shifted problem"; keep shifting the coarse grid and solving the shifted problem until we obtain all the boundary points for the fine grid. An illustration of this shifting procedure will be made for the two dimensional case in Section 4.4.
(Special case: in one dimension, do not need to shift the coarse grid.)

3. *Second step*: Solve the problem on each fine grid using the finite difference method.

4. *Interpolation (Optional)*: Interpolate the refined solution onto an interpolated grid. The purpose of interpolation is to compute error for convergence analysis.

4.2 Implementation

Recall that n_c denotes the number of points on the coarse grid, n_f denotes the number of points on the fine grid between two consecutive coarse grid points. Since $n_f = n_c$, we denote $n = n_f = n_c$, and n_{ref} as the number of points on the refined grid, so that $n_{ref} = (n_c - 1)(n_f - 1) + 1$. Let n_i denotes the number of points on the interpolated grid, with $n_{ref} < n_i$. We are assuming the number of points is the same in every direction for all grids.

There are a few difficulties in the implementation:

1. *Setting up the problem on the non-uniform coarse grid*: As we mentioned in Chapter 1, a standard technique for solving Poisson's equation is by using the finite difference method, which is essentially equivalent to solving a large system of linear equations in the form of $A\mathbf{u} = \mathbf{b}$, with the size of A depending on the dimensionality. In d-dimension, A is an $(n_c)^d \times (n_c)^d$ matrix, with $(2d+1)$ non-zero diagonals. It turns out that setting up the system of equations is efficient, even with nested for-loops, since $(2d+1) << (n_c)^d$. The following is a part of the MATLAB script for setting up the system of equations in one dimension:

```
for i = 2:Nx-1
  DXL = xm(i)-xm(i-1); DXR = xm(i+1)-xm(i);     % Grid sizes
  bX = 0.5*(DXL*DXR^2 + DXL^2*DXR);             % Denominator
  emid(i) = (DXL+DXR)/bX;                       % Diagonal
  edxL(i-1) = -DXR/bX;                          % Subdiagonal
  edxR(i+1) = -DXL/bX;                          % Superdiagonal
end
A = spdiags([edxL, emid, edxR],[-1, 0, 1], Nx, Nx);
```

2. *Solving the system of equations*: This is a bottleneck of the hybrid Monte-Carlo finite difference method. For this thesis, for one and two dimensions, we chose to use the direct method, such as MATLAB's built-in function backslash ("\"); for three and four dimensions, we chose to use the iterative methods, such as BiConjugate gradients method (MATLAB's built-in function *bicg*). The iterative methods generally work better than direct methods in terms of efficiency, provided the solution does not diverge. The direct methods generally work better than iterative methods in terms of robustness, provided the computer does not run out of memory. Details are provided in Appendix A. The following is the MATLAB script for solving the system of equations:

```
% For low dimensions
u = A\b;

% For high dimensions
u = bicg(A,b);
```

3. *Interpolating the numerical solution*: This is also a bottleneck of the hybrid Monte-Carlo finite difference method. We used MATLAB's built-in function *interp1* to perform the interpolation, which is expensive and inefficient as the number of dimensions increases.

Since each shifting is an independent calculation, we are able to perform the hybrid Monte-Carlo finite difference method in a parallel computing system, and the CPU time can be reduced by approximately a factor of the number of processors available; see Figure 4.6. For example, for a two dimensional Poisson's equation with two processors; we assign one processor to perform shifting and solving the shifted problem in the x-direction, and the other for the y-direction. We then split the domain into two pieces, and assign each processor to solve the fine grid problem in their assigned domain. Therefore, the CPU time can be reduced by approximately a factor of 2.

Two types of test cases are investigated in one, two and three dimensions: the "smooth" test case and the "spiky" test case.

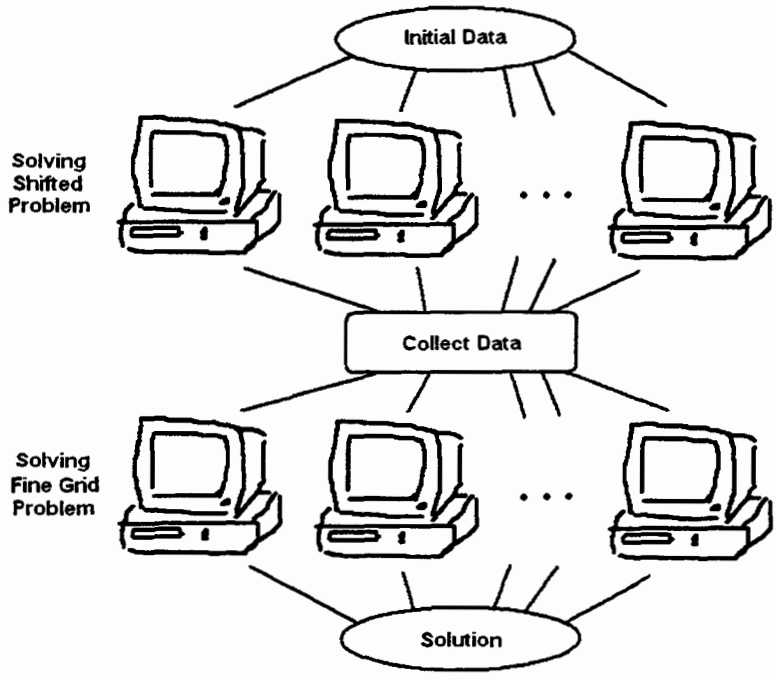

Figure 4.6: Schematic of hybrid Monte-Carlo finite difference method on parallel computing system.

4.3 In One Dimension

The one dimensional hybrid Monte-Carlo finite difference method is a special case, since we do not need to shift the coarse grid; the coarse grid already lies on the boundary for each fine grid. The steps are as follows:

Step 1: Define the coarse grid and fine grid

Step 2: Solve the problem on the coarse grid (*circles* in Figure 4.7)

Step 3: Solve the problem inside each fine grid (*triangles* in Figure 4.7)

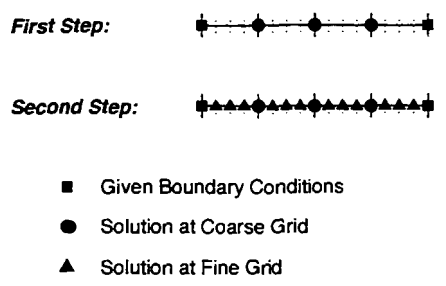

Figure 4.7: 1d: Coarse grid and fine grid.

Consider the following smooth test case (1d smooth test case):

$$\begin{cases} -u''(x) = 8, & \forall x \in (0,1) \\ u(0) = 0, \quad u(1) = 0, \end{cases} \tag{4.1}$$

which has an exact solution,

$$u(x) = 4x(1-x). \tag{4.2}$$

The numerical results are presented in Figures 4.8, 4.9 and 4.10. Figure 4.8 is the exact solution. Figure 4.9 is the numerical solution solved by the regular finite difference method. Figure 4.10 is the numerical solution solved by the hybrid Monte-Carlo finite difference method. From these figures, the numerical solutions by those methods are almost identical; later in Chapter 5, we will show that the hybrid Monte-Carlo finite difference method is better than the regular finite difference method.

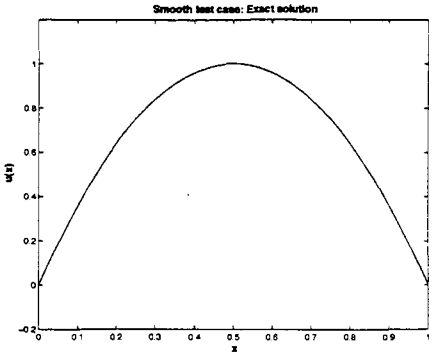

Figure 4.8: 1d smooth test case: Exact solution with $n_i = 3500$.

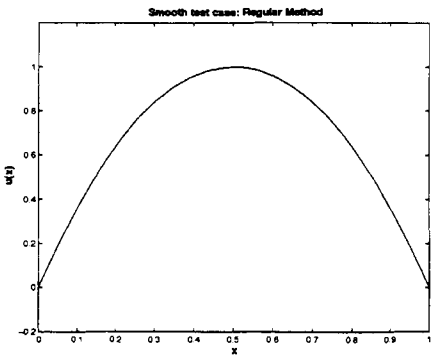

Figure 4.9: 1d smooth test case: By regular finite difference method with $n_c = 24$ and $n_i = 3500$.

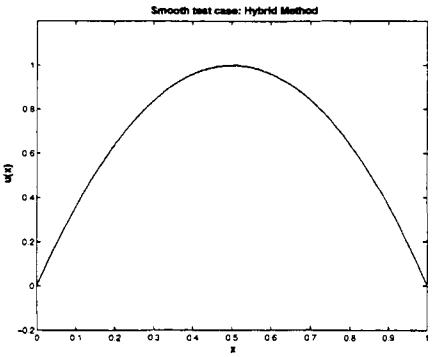

Figure 4.10: 1d smooth test case: By hybrid Monte-Carlo finite difference method with $n_c = n_f = 24$ and $n_i = 3500$.

Consider the following spiky test case (1d spiky test case): $\sigma = \frac{1}{600}$, where σ is the standard deviation of a gaussian.

$$\begin{cases} -u''(x) = 8 + \frac{1}{\sigma\sqrt{2\pi}}\Big(\frac{1}{\sigma^2} - \frac{(x-0.5)^2}{\sigma^4}\Big)exp\Big(-\frac{(x-0.5)^2}{2\sigma^2}\Big), & \forall x \in (0,1) \\ u(0) = 0, \quad u(1) = 0, \end{cases} \tag{4.3}$$

which has an exact solution,

$$u(x) = 4x(1-x) + \frac{1}{\sigma\sqrt{2\pi}}exp\Big(-\frac{(x-0.5)^2}{2\sigma^2}\Big). \tag{4.4}$$

The numerical results are presented in Figures 4.11, 4.12 and 4.13. Figure 4.11 is the exact solution. Figure 4.12 is the numerical solution solved by the regular finite difference method. Figure 4.13 is the numerical solution solved by the hybrid Monte-Carlo finite difference method. From these figures, we can see that the regular finite difference method failed to resolve the spike, whereas the hybrid Monte-Carlo finite difference method resolved the spike.

4.4 In Two Dimensions

In two dimensions, we need to shift the coarse grid in two directions, the x and y directions. The boundary conditions on the fine grid are along the four edges. The implementation of the shifting is straightforward: first fix x and vary y; this will give us the left and right edges. Then fix y and vary x, and this will give us the top and bottom edges. An example of the procedure of combining the solution of the coarse grid problem and the shifted problem is given in Figure 4.14.

The steps are as follows:

Step 1: Define the coarse grid and fine grid

Step 2: Solve the problem on the coarse grid (*circles* in Figure 4.15)

Step 3: Shift the coarse grid and solve the shifted problem (*stars* in Figure 4.15)

Step 4: Solve the problem inside each fine grid (*triangles* in Figure 4.15)

Consider the following smooth test case (2d smooth test case):

$$\begin{cases} -\triangle u(\boldsymbol{x}) = 32x(1-x) + 32y(1-y), & \forall \boldsymbol{x} = (x,y) \in (0,1)^2 = \Omega \\ u(\boldsymbol{x}) = 0, & \forall \boldsymbol{x} \in \partial\Omega \end{cases} \tag{4.5}$$

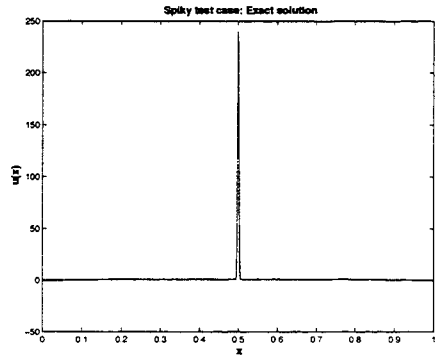

Figure 4.11: 1d spiky test case: Exact solution with $n_i = 3500$.

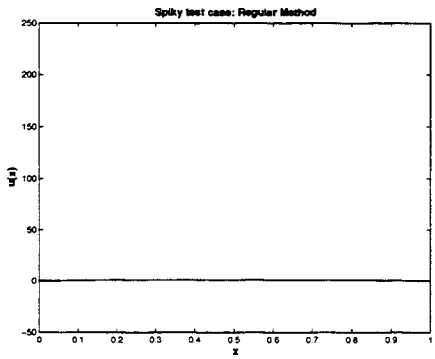

Figure 4.12: 1d spiky test case: By regular finite difference method with $n_c = 24$ and $n_i = 3500$.

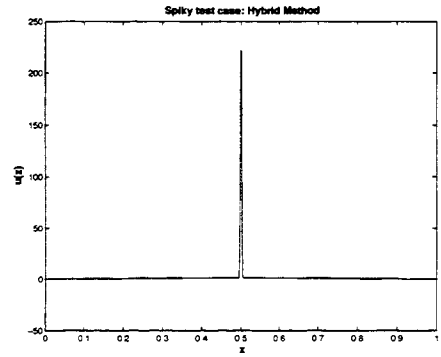

Figure 4.13: 1d spiky test case: By hybrid Monte-Carlo finite difference method with $n_c = n_f = 24$ and $n_i = 3500$.

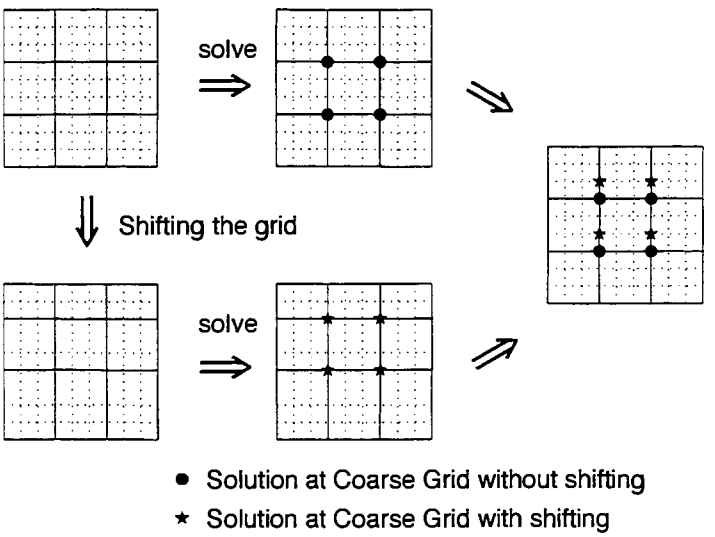

Figure 4.14: 2d: Example of the procedure of combining the solution of the coarse grid problem and the shifted problem.

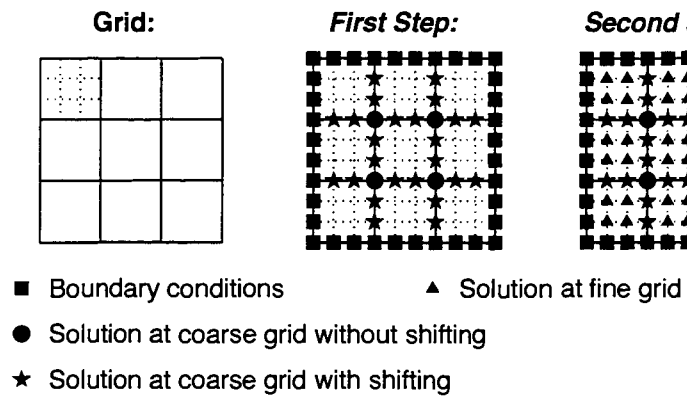

Figure 4.15: 2d: Coarse grid and fine grid.

which has an exact solution,

$$u(\boldsymbol{x}) = 16xy(1-x)(1-y). \tag{4.6}$$

The numerical results are presented in Figures 4.16, 4.17 and 4.18. Figure 4.16 is the exact solution. Figure 4.17 is the numerical solution solved by the regular finite difference method. Figure 4.18 is the numerical solution solved by the hybrid Monte-Carlo finite difference method. Similar to one dimension, from these figures, the numerical solutions by those methods are almost identical; later in Chapter 5, we will show that the hybrid Monte-Carlo finite difference method is better than the regular finite difference method.

Consider the following spiky test case (2d spiky test case): $\sigma = \frac{1}{600}$

$$\begin{cases} -\triangle u(\boldsymbol{x}) = 32x(1-x) + 32y(1-y) \\ \quad + \frac{1}{\sigma\sqrt{2\pi}}\Big(\frac{2}{\sigma^2} - \frac{(x-0.5)^2}{\sigma^4} - \frac{(y-0.5)^2}{\sigma^4}\Big) exp\Big(-\frac{(x-0.5)^2}{2\sigma^2} - \frac{(y-0.5)^2}{2\sigma^2}\Big), \\ \qquad\qquad\qquad\qquad \forall \boldsymbol{x} = (x,y) \in (0,1)^2 = \Omega \\ u(\boldsymbol{x}) = 0, \qquad\qquad \forall \boldsymbol{x} \in \partial\Omega \end{cases} \tag{4.7}$$

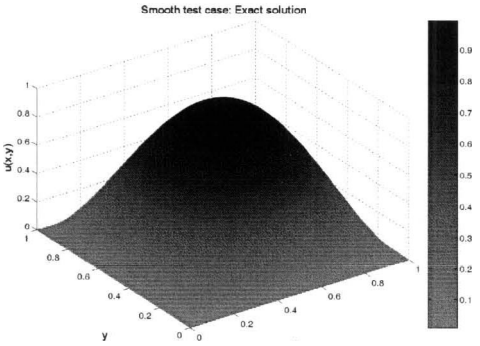

Figure 4.16: 2d smooth test case: Exact solution with $n_i = 1500$.

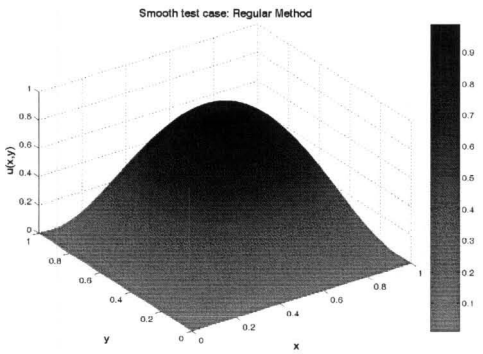

Figure 4.17: 2d smooth test case: By regular finite difference method with $n_c = 36$ and $n_i = 1500$.

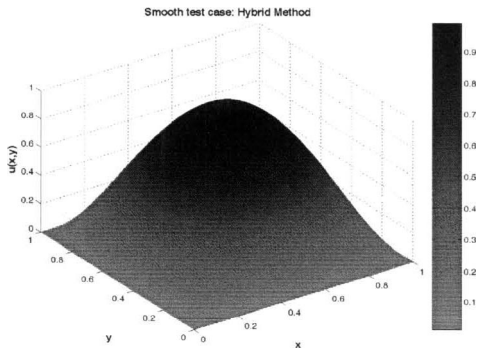

Figure 4.18: 2d smooth test case: By hybrid Monte-Carlo finite difference method with $n_c = n_f = 36$ and $n_i = 1500$.

which has an exact solution,

$$u(\boldsymbol{x}) = 16xy(1-x)(1-y) + \frac{1}{\sigma\sqrt{2\pi}} exp\Big(- \frac{(x-0.5)^2}{2\sigma^2} - \frac{(y-0.5)^2}{2\sigma^2} \Big). \tag{4.8}$$

The numerical results are presented in Figures 4.19, 4.20 and 4.21. Figure 4.19 is the exact solution. Figure 4.20 is the numerical solution solved by the regular finite difference method. Figure 4.21 is the numerical solution solved by the hybrid Monte-Carlo finite difference method. Similar to one dimension, from these figures, we can see that the regular finite difference method failed to resolve the spike, whereas the hybrid Monte-Carlo finite difference method resolved the spike.

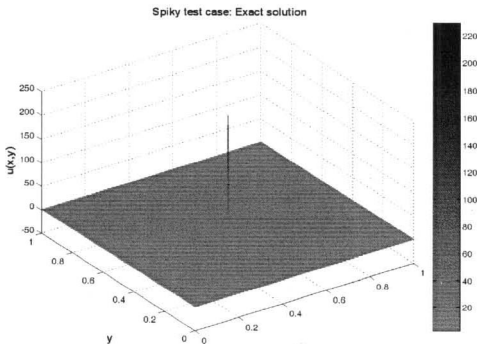

Figure 4.19: 2d spiky test case: Exact solution with $n_i = 1500$.

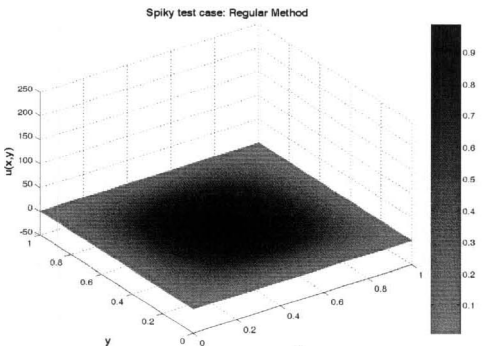

Figure 4.20: 2d spiky test case: By regular finite difference method with $n_c = 36$ and $n_i = 1500$.

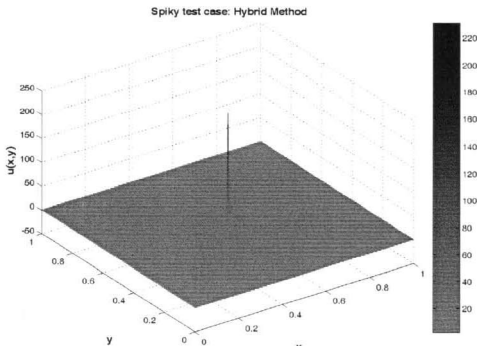

Figure 4.21: 2d spiky test case: By hybrid Monte-Carlo finite difference method with $n_c = n_f = 36$ and $n_i = 1500$.

4.5 In Three Dimensions

In three dimensions, we need to shift the coarse grid in three directions, the x, y and z directions. For the shifting, the idea is similar as two dimensional case. First we fix x and vary y and z; this will give us the left and right planes. Then we fix y and vary x and z; which will give us the front and back planes. Finally we fix z and vary x and y, this will give us the top and bottom planes. We then have all 6 planes for boundary conditions. The steps are as follows:

Step 1: Define the coarse grid and fine grid

Step 2: Solve the problem on the coarse grid

Step 3: Shift the coarse grid and solve the shifted problem

Step 4: Solve the problem inside each fine grid

Consider the following smooth test case (3d smooth test case):

$$\begin{cases} -\triangle u(\boldsymbol{x}) = 128xy(1-x)(1-y) + 128xz(1-x)(1-z) + 128yz(1-y)(1-z), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall \boldsymbol{x} = (x,y,z) \in (0,1)^3 = \Omega \\ \qquad\qquad u(\boldsymbol{x}) = 0, \qquad\qquad \forall \boldsymbol{x} \in \partial\Omega \end{cases} \tag{4.9}$$

which has an exact solution,

$$u(\boldsymbol{x}) = 64xyz(1-x)(1-y)(1-z). \tag{4.10}$$

The numerical results for the 0.5-isosurface are presented in Figures 4.22, 4.23 and 4.24. Figure 4.22 is the exact solution. Figure 4.23 is the numerical solution solved by the regular finite difference method. Figure 4.24 is the numerical solution solved by the hybrid Monte-Carlo finite difference method. From these figures, we can see that the numerical solution solved by the hybrid Monte-Carlo finite difference method give a smoother solution than the regular finite difference method.

Consider the following spiky test case (3d spiky test case): $\sigma = \frac{1}{180}$

$$\begin{cases} -\triangle u(\boldsymbol{x}) = 128xy(1-x)(1-y) + 128xz(1-x)(1-z) + 128yz(1-y)(1-z) \\ \qquad\qquad +A * exp\Big(-\dfrac{(x-0.5)^2}{2\sigma^2} - \dfrac{(y-0.5)^2}{2\sigma^2} - \dfrac{(z-0.5)^2}{2\sigma^2} \Big), \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall \boldsymbol{x} = (x,y) \in (0,1)^2 = \Omega \\ \qquad\qquad u(\boldsymbol{x}) = 0, \qquad\qquad \forall \boldsymbol{x} \in \partial\Omega \\ \text{where } A = \dfrac{1}{\sigma\sqrt{2\pi}}\Big(\dfrac{3}{\sigma^2} - \dfrac{(x-0.5)^2}{\sigma^4} - \dfrac{(y-0.5)^2}{\sigma^4} - \dfrac{(z-0.5)^2}{\sigma^4} \Big) \end{cases} \tag{4.11}$$

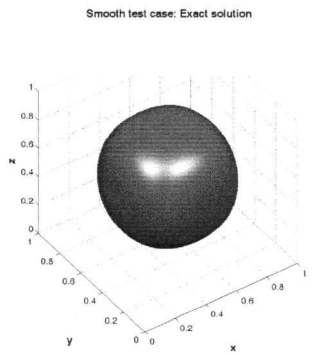

Figure 4.22: 3d smooth test case: Exact solution with $n_i = 150$.

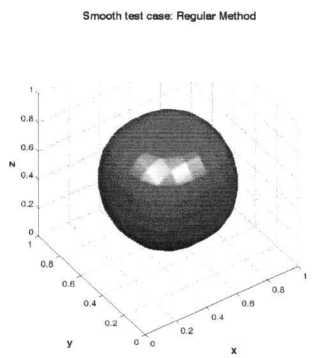

Figure 4.23: 3d smooth test case: By regular finite difference method with $n_c = 12$ and $n_i = 150$.

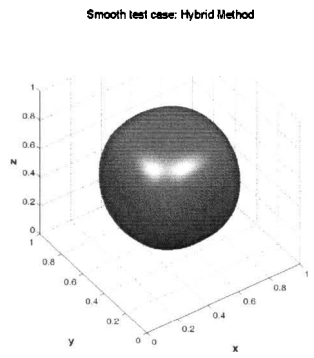

Figure 4.24: 3d smooth test case: By hybrid Monte-Carlo finite difference method with $n_c = n_f = 12$ and $n_i = 150$.

which has an exact solution,

$$u(\boldsymbol{x}) = 64xyz(1-x)(1-y)(1-z)+\frac{1}{\sigma\sqrt{2\pi}}exp\Big(-\frac{(x-0.5)^2}{2\sigma^2}-\frac{(y-0.5)^2}{2\sigma^2}-\frac{(z-0.5)^2}{2\sigma^2}\Big). \quad (4.12)$$

The numerical results for the 1.5-isosurface are presented in Figures 4.25, 4.26 and 4.27. Figure 4.25 is the exact solution. Figure 4.26 is the numerical solution solved by the regular finite difference method. Figure 4.27 is the numerical solution solved by the hybrid Monte-Carlo finite difference method. In these figures, we can see that the regular finite difference method failed to resolve the sharp feature, whereas the hybrid Monte-Carlo finite difference method resolved the spike.

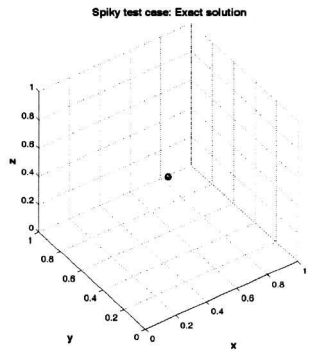

Figure 4.25: 3d spiky test case: Exact solution with $n_i = 150$.

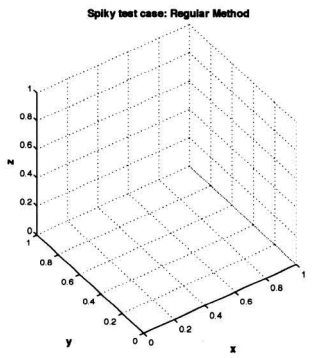

Figure 4.26: 3d spiky test case: By regular finite difference method with $n_c = 12$ and $n_i = 150$.

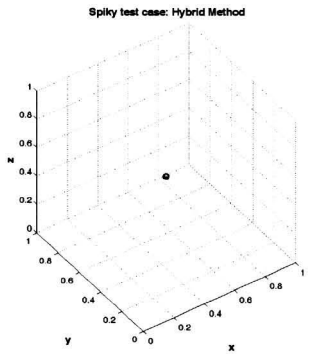

Figure 4.27: 3d spiky test case: By hybrid Monte-Carlo finite difference method with $n_c = n_f = 12$ and $n_i = 150$.

4.6 In Higher Dimensions

In this section, we will generalize the idea from previous sections into higher dimensions. For four dimensions, we need to shift the coarse grid in four directions, and the corresponding boundary conditions on the fine grid are eight cubes. For the shifting, the idea is to fixed one direction, and vary the other three. After fixing all four directions, then we will have all eight cubes for boundary conditions as desired.

Consider the following test case (4d test case):

$$\begin{cases} -\triangle u(\boldsymbol{x}) = 57\pi^2 \sin(\pi x_1)\sin(4\pi x_2)\sin(2\pi x_3)\sin(6\pi x_4), \\ \qquad\qquad\qquad\qquad \forall \boldsymbol{x} = (x_1, x_2, x_3, x_4) \in (0,1)^4 = \Omega \\ \qquad u(\boldsymbol{x}) = 0, \qquad\qquad \forall \boldsymbol{x} \in \partial\Omega \end{cases} \tag{4.13}$$

which has an exact solution,

$$u(\boldsymbol{x}) = \sin(\pi x_1)\sin(4\pi x_2)\sin(2\pi x_3)\sin(6\pi x_4). \tag{4.14}$$

The numerical results for the 0.05-isosurface at $x_4 = \frac{24}{49}$ are presented in Figures 4.28, 4.29 and 4.30. Figure 4.28 is the exact solution. Figure 4.29 is the numerical solution solved by the regular finite difference method. Figure 4.30 is the numerical solution solved by the hybrid Monte-Carlo finite difference method. Similar to three dimensions, from those figures, we can see that numerical solution solved by the hybrid Monte-Carlo finite difference method give a smoother solution than the regular finite difference method.

In general, for a d-dimensional problem, we need to shift the coarse grid in d different directions. For the shifting, we need to fix one direction and vary the other $(d-1)$ directions. We continue until all d directions have been shifted. Then we solve the problem for each fine grid cell.

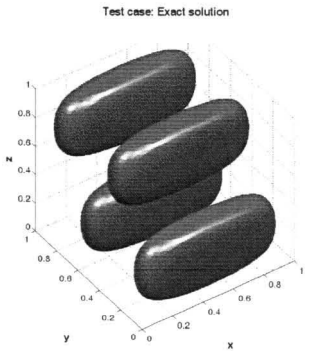

Figure 4.28: 4d test case: Exact solution with $n_i = 50$.

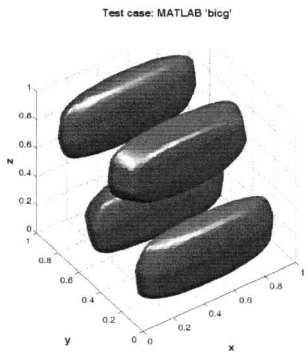

Figure 4.29: 4d test case: By regular finite difference method with $n_c = 6$ and $n_i = 50$.

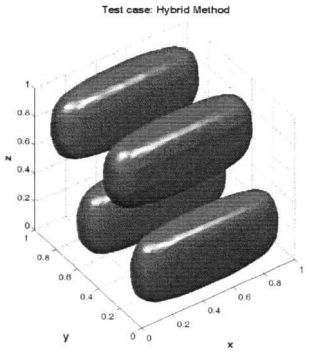

Figure 4.30: 4d test case: By hybrid Monte-Carlo finite difference method with $n_c = n_f = 6$ and $n_i = 50$.

Chapter 5

Convergence Analysis

As was explained in Chapters 2 and 3, the hybrid Monte-Carlo finite difference method is second order accurate. In this chapter, we will verify this accuracy in one, two and three dimensions. The test cases are the same as those introduced in Chapter 4.

Recall that the d-dimensional domain is $[0,1]^d$, n is the number of points on coarse grid and fine grid. DX is the size of the coarse grid, and dx is the size of the fine grid, which satisfy,

$$DX = \frac{1}{n-1}, dx = \frac{DX}{n-1}.$$

This implies

$$O(dx) = O((DX)^2). \tag{5.1}$$

5.1 In One Dimension

The l_2-error and l_∞-error are computed by first solving the problem using a regular finite difference method and the hybrid Monte-Carlo finite difference method, interpolating these numerical solutions onto a grid with 3500 points, then computing the errors on this interpolated grid. For the smooth test case, Figures 5.1 and 5.2 illustrate that when the problem is solved by the regular finite difference method, both the l_2-error and l_∞-error are second order accurate with respect to DX, which implies they are first order accurate with respect to dx. When the same problem is solved by the hybrid Monte-Carlo finite difference method, both the l_2-error and l_∞-error are fourth order accurate with respect to DX, which implies they are second order accurate with respect to dx.

Table 5.1: 1d smooth test case: Comparison of the l_2-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	24	68	8.1661182e-02	1.5436896e-04
28	28	80	5.9257566e-02	8.1286098e-05
32	32	92	4.4951889e-02	4.6776159e-05
36	36	104	3.5264298e-02	2.8787182e-05
40	40	116	2.8401555e-02	1.8672949e-05
44	44	128	2.3363313e-02	1.2635648e-05
48	48	140	1.9555801e-02	8.8527849e-06

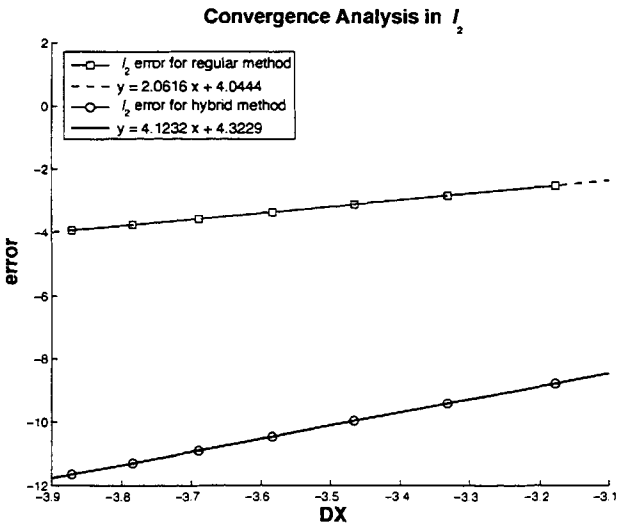

Figure 5.1: 1d smooth test case: Log-log plot of the l_2-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Table 5.2: 1d smooth test case: Comparison of the l_∞-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	24	68	1.8903590e-03	3.5734575e-06
28	28	80	1.3717420e-03	1.8816763e-06
32	32	92	1.0405826e-03	1.0828123e-06
36	36	104	8.1632646e-04	6.6638896e-07
40	40	116	6.5746214e-04	4.3225651e-07
44	44	128	5.4083284e-04	2.9250018e-07
48	48	140	4.5269349e-04	2.0493142e-07

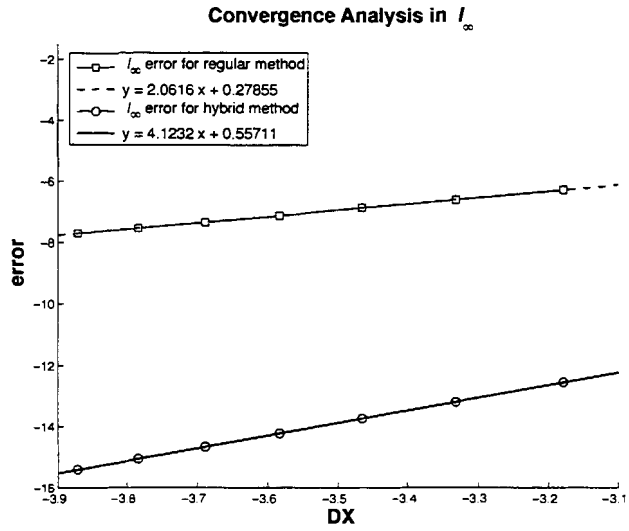

Figure 5.2: 1d smooth test case: Log-log plot of the l_∞-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

For the spiky test case, the conclusion is the same as for the smooth test case; Figures 5.3 and 5.4 illustrate that when the same problem is solved by the hybrid Monte-Carlo finite difference method, both the l_2-error and l_∞-error are fourth order accurate with respect to DX, which implies they are second order accurate with respect to dx.

Table 5.3: 1d spiky test case: Comparison of the l_2-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	24	68	7.6957323e+02	4.7390319e+01
28	28	80	7.6957087e+02	2.1361744e+01
32	32	92	7.6956937e+02	9.8091310e+00
36	36	104	7.6956835e+02	7.1882416e+00
40	40	116	7.6956767e+02	3.8807904e+00
44	44	128	7.6957281e+02	2.1143355e+00
48	48	140	7.6979926e+02	2.4478220e+00

Table 5.4: 1d spiky test case: Comparison of the l_∞-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	24	68	2.3848907e+02	1.8259998e+01
28	28	80	2.3848855e+02	1.0258321e+01
32	32	92	2.3848822e+02	3.4652194e+00
36	36	104	2.3848800e+02	3.9194268e+00
40	40	116	2.3848785e+02	1.5358952e+00
44	44	128	2.3848898e+02	8.0260636e-01
48	48	140	2.3853834e+02	6.8900509e-01

We should expect first order accuracy with respect to dx for the regular finite difference method since we discretized the PDE using a $O(DX^2)$ scheme, by (5.1), this is a $O(dx)$ scheme. Whereas for the hybrid Monte-Carlo finite difference method, each of the fine grid problem is solved by a $O(dx^2)$ scheme. Notice that at the boundaries of the fine grid, the

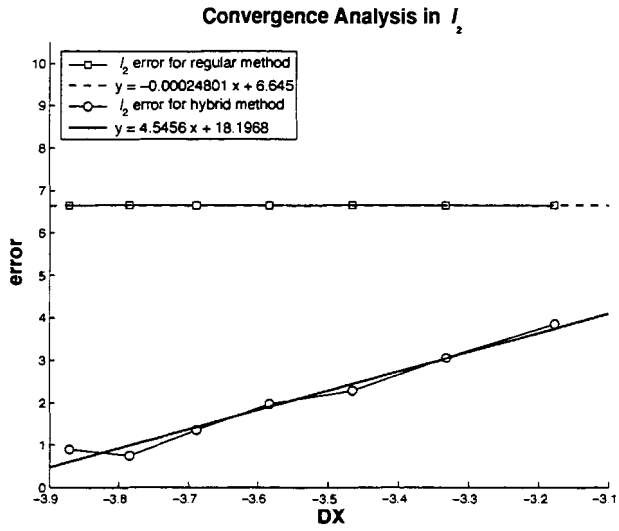

Figure 5.3: 1d spiky test case: Log-log plot of the l_2-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

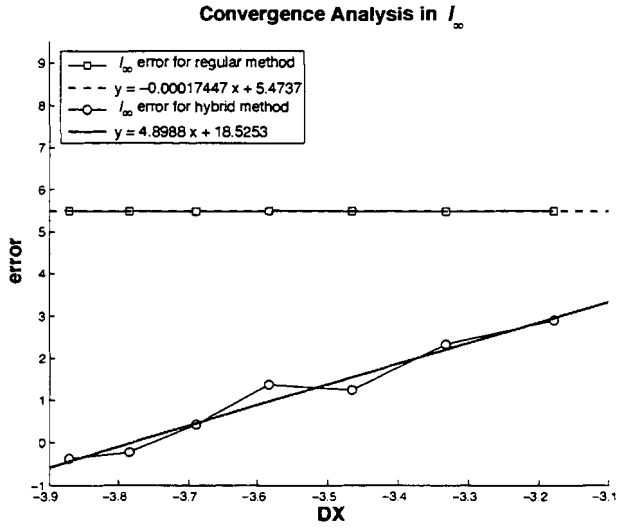

Figure 5.4: 1d spiky test case: Log-log plot of the l_∞-error versus DX for the regular linear solver and the hybrid Monte-Carlo finite difference method.

errors could be $O(dx)$ in the worst case, we suspect the affect of the interpolation average out the $O(dx)$ to $O(dx^2)$.

5.2 In Two Dimensions

Similar to one dimension, for two dimensions, the l_2-error and l_∞-error are computed by first solving the problem using a regular finite difference method and the hybrid Monte-Carlo finite difference method, interpolating these numerical solutions onto a 2500 $\times$ 2500 grid, then computing the errors on this interpolated grid. For the smooth test case, Figures 5.5 and 5.6 illustrate that when the problem is solved by the regular finite difference method, both the l_2-error and l_∞-error are first order accurate with respect to dx. When the same problem is solved by the hybrid Monte-Carlo finite difference method, both the l_2-error and l_∞-error are second order accurate with respect to dx.

Table 5.5: 2d smooth test case: Comparison of the l_2-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	576	2512	4.6344882e+00	8.7676346e-03
28	784	3488	3.3637355e+00	4.6168425e-03
32	1024	4624	2.5520268e+00	2.6570487e-03
36	1296	5920	2.0022162e+00	1.6347700e-03
40	1600	7376	1.6126798e+00	1.0605781e-03
44	1936	8992	1.3266642e+00	7.1766278e-04
48	2304	10768	1.1104984e+00	5.0280891e-04

For the spiky test case, the conclusion is the same as for the smooth test case; Figures 5.7 and 5.8 illustrate that both the l_2-error and l_∞-error are second order accurate with respect to dx, when the problem solved by the hybrid Monte-Carlo finite difference method.

5.3 In Three Dimensions

Similar to previous sections, the l_2-error and l_∞-error are computed by first solving the problem using a regular finite difference method and the hybrid Monte-Carlo finite difference

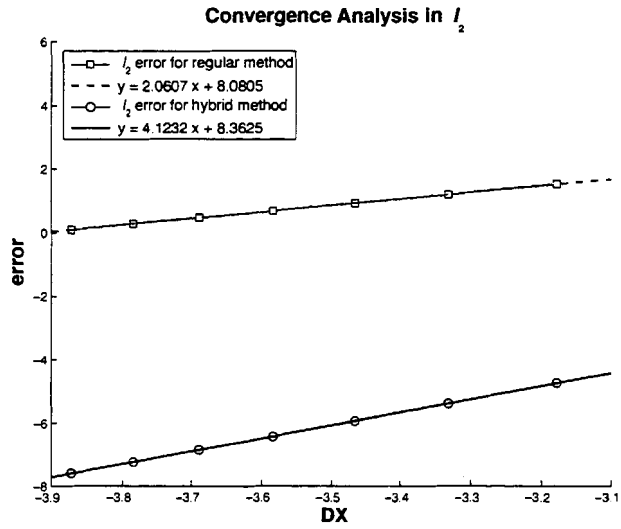

Figure 5.5: 2d smooth test case: Log-log plot of the l_2-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Table 5.6: 2d smooth test case: Comparison of the l_∞-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	576	2512	3.7768246e-03	7.1430280e-06
28	784	3488	2.7412823e-03	3.7566760e-06
32	1024	4624	2.0797624e-03	2.1530844e-06
36	1296	5920	1.6316664e-03	1.3317530e-06
40	1600	7376	1.3141719e-03	8.6443844e-07
44	1936	8992	1.0810530e-03	5.8462773e-07
48	2304	10768	9.0486187e-04	4.0970043e-07

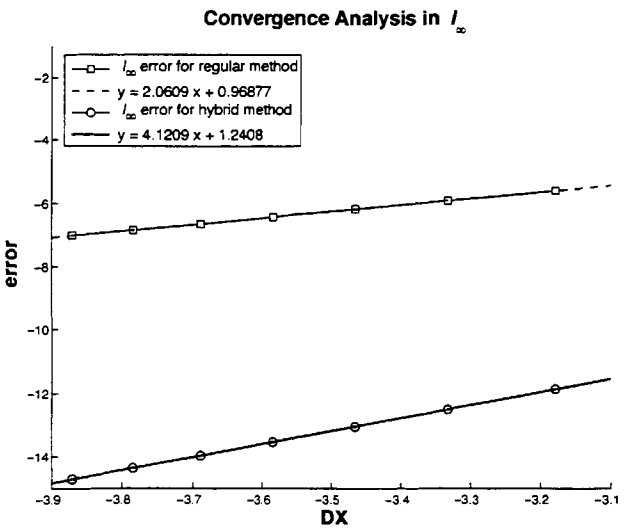

Figure 5.6: 2d smooth test case: Log-log plot of the l_∞-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Table 5.7: 2d spiky test case: Comparison of the l_2-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	576	2512	1.7671214e+03	1.6818751e+02
28	784	3488	1.7671032e+03	7.6616214e+01
32	1024	4624	1.7670921e+03	4.5133328e+01
36	1296	5920	1.7670847e+03	2.2140317e+01
40	1600	7376	1.7670797e+03	1.6099298e+01
44	1936	8992	1.7670760e+03	1.0599295e+01
48	2304	10768	1.7670732e+03	6.4038401e+00

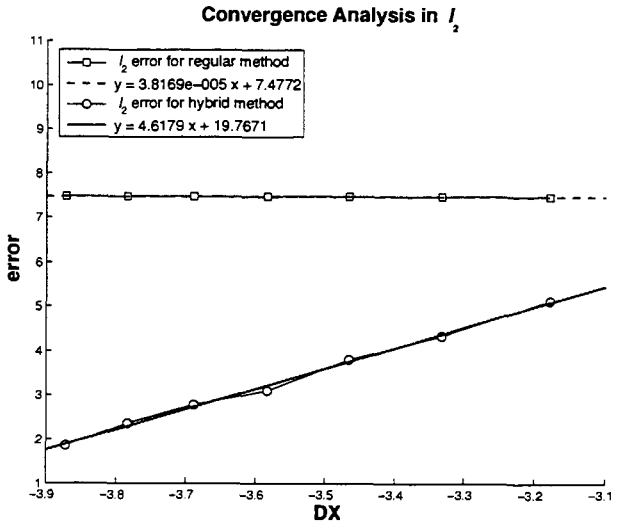

Figure 5.7: 2d spiky test case: Log-log plot of the l_2-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Table 5.8: 2d spiky test case: Comparison of the l_∞-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
24	576	2512	2.3594426e+02	4.1709136e+01
28	784	3488	2.3594323e+02	1.8391503e+01
32	1024	4624	2.3594257e+02	8.7205089e+00
36	1296	5920	2.3594212e+02	3.9318849e+00
40	1600	7376	2.3594180e+02	3.1998599e+00
44	1936	8992	2.3594157e+02	2.3306440e+00
48	2304	10768	2.3594139e+02	1.1973687e+00

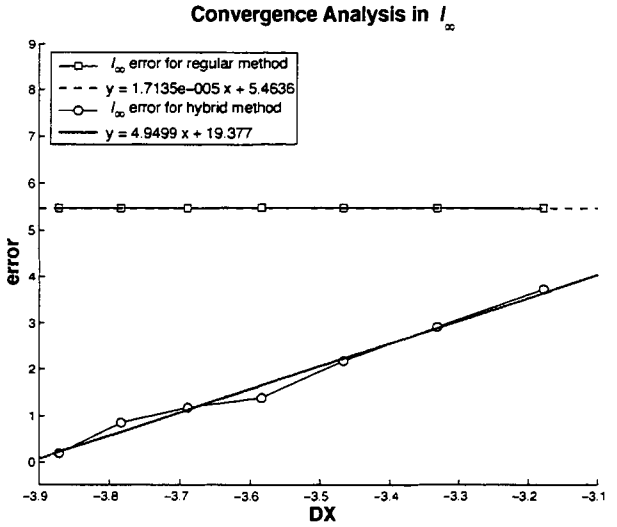

Figure 5.8: 2d spiky test case: Log-log plot of the l_∞-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

method, interpolating these numerical solutions onto a $450 \times 450 \times 450$ grid, then computing the errors on this interpolated grid. For the smooth test case, Figures 5.9 and 5.10 illustrate that when the problem is solved by a regular finite difference method, both the l_2-error and l_∞-error are first order accurate with respect to dx. When the same problem is solved by the hybrid Monte-Carlo finite difference method, both the l_2-error and l_∞-error are second order accurate with respect to dx.

Table 5.9: 3d smooth test case: Comparison of the l_2-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
12	1728	7728	8.1431117e+01	6.7756377e-01
14	2744	13112	5.8416310e+01	3.4736397e-01
16	4096	20560	4.3934907e+01	2.0171980e-01
18	5832	30408	3.4229340e+01	1.1894284e-01

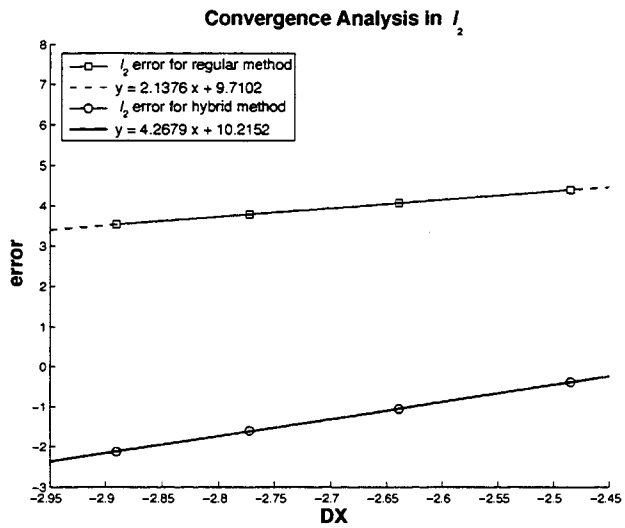

Figure 5.9: 3d smooth test case: Log-log plot of the l_2-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Table 5.10: 3d smooth test case: Comparison of the l_∞-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
12	1728	7728	2.4574169e-02	2.0414153e-04
14	2744	13112	1.7631753e-02	1.0366868e-04
16	4096	20560	1.3259284e-02	4.7750951e-05
18	5832	30408	1.0329876e-02	4.4105264e-05

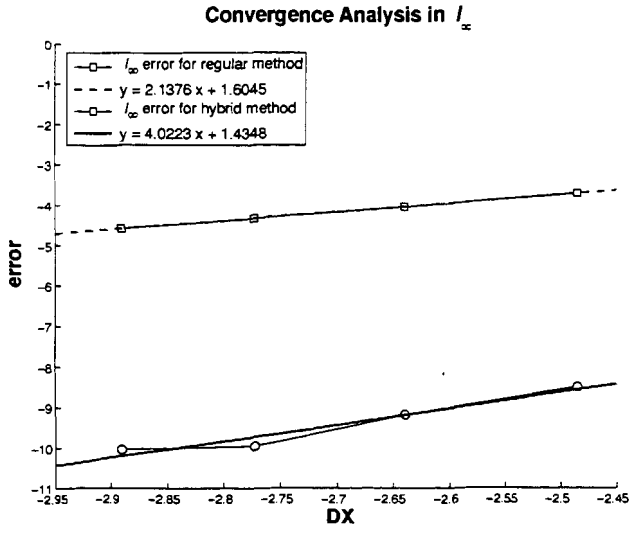

Figure 5.10: 3d smooth test case: Log-log plot of the l_∞-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

For the spiky test case, the conclusion is the same as for the smooth test case; Figures 5.11 and 5.12 illustrate that when the problem is solved by the hybrid Monte-Carlo finite difference method, both the l_2-error and l_∞-error are second order accurate with respect to dx, .

Table 5.11: 3d spiky test case: Comparison of the l_2-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
12	1728	7728	6.7316365e+02	2.0864511e+02
14	2744	13112	6.7058706e+02	7.4289823e+01
16	4096	20560	6.6936635e+02	3.9009422e+01
18	5832	30408	6.6872254e+02	2.2621176e+01

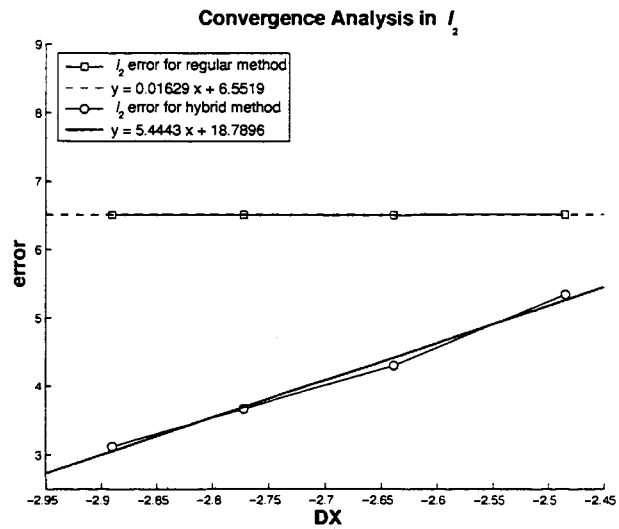

Figure 5.11: 3d spiky test case: Log-log plot of the l_2-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Table 5.12: 3d spiky test case: Comparison of the l_∞-error when solved by the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Number of grid points, n	Size of A, N	Number of non-zero elements of A	Regular finite difference method	Hybrid Monte-Carlo finite difference method
12	1728	7728	6.7634227e+01	3.4418625e+01
14	2744	13112	6.7627284e+01	1.4429372e+01
16	4096	20560	6.7622912e+01	6.2010211e+00
18	5832	30408	6.7619982e+01	2.2174662e+00

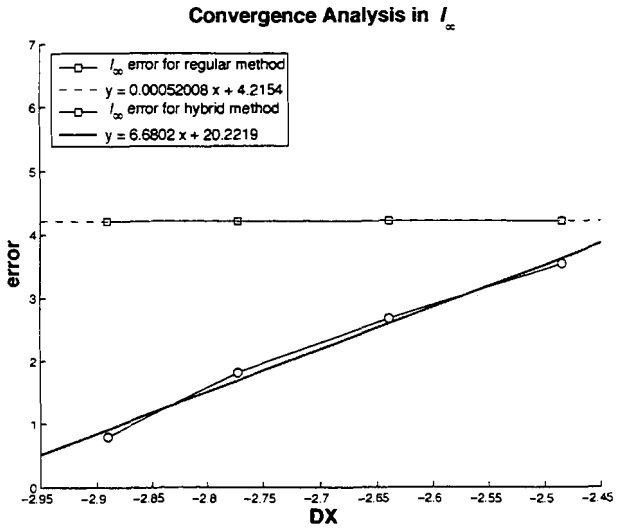

Figure 5.12: 3d spiky test case: Log-log plot of the l_∞-error versus DX for the regular finite difference method and the hybrid Monte-Carlo finite difference method.

Chapter 6

Conclusions

6.1 Summary

In this thesis, we presented a hybrid Monte-Carlo finite difference method for approximating the solution of Poisson's equation, with numerical results presented in one, two, three and four dimensions. This method combined the idea from the Monte-Carlo method and the finite difference method. From the test cases, we have successfully shown that the hybrid Monte-Carlo finite difference method is second order accurate on generic problems, and on problems with sharp features, such as spike.

This hybrid method solves smaller problems multiple times to collectively solve a larger main problem. A bottleneck of this method is the solvable size of the linear solver, which could be direct or iterative method. In general, direct methods require larger memory and more work, but are more robust, while iterative methods require less memory and less work, but are also less robust. For the purpose of this thesis, we chose to use the direct method for one and two dimensions, and the BiConjugate gradient method for three and four dimensions.

We were able to perform the hybrid Monte-Carlo finite difference method in a parallel computing system, and the CPU time can be reduced by approximately a factor of the number of processors available.

6.2 Future Work

It is worthwhile to investigate different ways of shifting the coarse grid to improve accuracy, see Figure 6.1 for example.

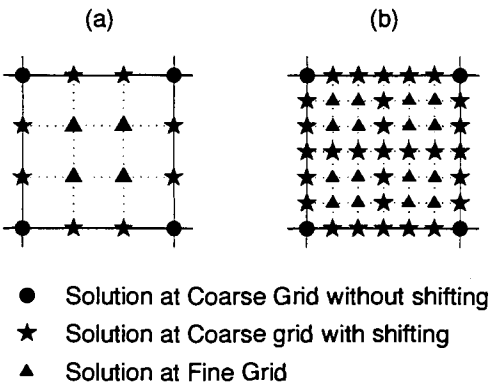

Figure 6.1: (a) Shifting used in this thesis. (b) Different way of shifting.

The investigation of fast linear solver is also worthy of future research, as the hybrid Monte-Carlo finite difference method can be performed on a parallel computing system. The next step is to use a higher order finite difference scheme to discretize the PDE instead of using a second order scheme. The final goal is to generalize this higher order hybrid Monte-Carlo finite difference method to solve nonlinear elliptic PDEs in high dimensions.

Appendix A

Computational Complexity

As stated in Chapter 4, a bottleneck of the hybrid Monte-Carlo finite difference method is the linear solver. We showed that the matrix representation A is always sparse, non-symmetric[1] and positive definite. For an $N \times N$ matrix A, the commonly used direct algorithm for solving $A\mathbf{u} = \mathbf{b}$ for $\mathbf{u}$ is as follows: first apply the LU factorization, then use backward and forward substitutions; this is exactly the algorithm in MATLAB's "\" when non-symmetric matrices are detected, see [6]. Theoretically, for some commonly used iterative methods, such as the BiConjugate gradients method (BiCG) or Generalized Minimum Residual (GMRES) method, it will converge in N iterations; however, this N can be very large in practice, and with the presence of rounding errors, this does not guarantee convergence after N iterations. Sometimes one needs to pick a *good* initial value, or use preconditioners to accelerate convergence speed, which depend on the structure of the matrix, see [1] for details. Tables A.1, A.2, A.3, A.4 and A.5 give an idea of the CPU time it takes to solve a sparse non-symmetric positive definite matrix using above methods in different dimensions.

Tables A.1 and A.2 showed that both GMRES and BiCG are slower than MATLAB's "\" for one and two dimensional problems. We can avoid divergence by increasing the maximum number of iterations for the iterative methods, as the default is 20 iterations, but consequently this will also increase the CPU time. Therefore the preferable linear solver for low dimensions is MATLAB's "\".

As the number of dimensions gets to three or higher, the iterative methods start to dominate over MATLAB's "\". Eventually MATLAB's "\" runs out of memory, while

[1] A is non-symmetric if $A \neq A^T$, the transpose of A.

BiCG is still capable of computing the solution within seconds, see Tables A.3, A.4 and A.5. Therefore the preferable linear solver for three and higher dimensional problem is BiCG. Notice that GMRES seems to diverge for high dimensions; we could either start with different initial values, use preconditioners, or increase the maximum number of iterations. Studies had been done on the choice of initial values and preconditioners in [1] and [15]. Unfortunately, there are no rules governing the choice of the maximum number of iterations. If it is too small, the iterative method may fail to converge while if it is too big, the CPU time will increase. For this thesis, the maximum number of iterations for 3d and 4d is 50.

Table A.1: 1d: Comparison of CPU (in s) with different MATLAB direct and iterative methods.

Number of grid point, n	Size of A, N	Number of non-zero elements of A	"\"	BiCG	GMRES
8	8	20	6.2826390e-05	1.4277552e-03	2.9469297e-03
32	32	92	7.5697699e-05	3.9552423e-03	8.2514993e-03
64	64	188	1.0510069e-04	8.0481086e-03	2.0347290e-02
256	256	764	3.3165373e-04	4.0565146e-02	2.8997676e-01

Table A.2: 2d: Comparison of CPU (in s) with different MATLAB direct and iterative methods.

Number of grid point, n	Size of A, N	Number of non-zero elements of A	"\"	BiCG	GMRES
4	16	32	1.0166873e-01	1.6219542e-01	1.1474611e-01
8	64	208	3.7619237e-02	2.5056189e-03	5.2415389e-03
16	256	1040	3.7399917e-03	9.4747756e-03	diverges
32	1024	4624	1.0810978e-02	6.6301227e-02	diverges
64	4096	19472	4.4514257e-02	7.1515917e-01	diverges

Table A.3: 3d: Comparison of CPU (in s) with different MATLAB direct and iterative methods.

Number of grid point, n	Size of A, N	Number of non-zero elements of A	"\"	BiCG	GMRES
8	512	1808	5.9455005e-03	5.7127436e-03	1.3564642e-02
12	1728	7728	2.8584450e-02	4.1951017e-02	diverges
22	10648	58648	7.5399983e-01	6.4005761e-01	diverges
36	46656	282480	1.7687460e+01	4.4085357e+00	diverges

Table A.4: 4d: Comparison of CPU (in s) with different MATLAB direct and iterative methods.

Number of grid point, n	Size of A, N	Number of non-zero elements of A	"\"	BiCG	GMRES
8	4096	14464	1.1719691e-01	8.9810176e-02	1.4953934e-01
12	20736	100736	5.2448255e+00	8.1901567e-01	diverges
16	65536	372864	9.0207953e+01	3.1677558e+00	diverges
20	160000	999808	out of memory	5.3657419e+00	diverges

Table A.5: 5d: Comparison of CPU (in s) with different MATLAB direct and iterative methods.

Number of grid point, n	Size of A, N	Number of non-zero elements of A	"\"	BiCG	GMRES
4	1024	1344	2.5706150e-03	8.3475234e-03	1.1699926e-02
6	7776	18016	1.4277831e-01	8.0995105e-02	1.6461228e-01
8	32768	110528	9.3720557e+00	7.0985322e-01	9.2218488e-01
10	100000	427680	out of memory	5.2838733e+00	diverges

Appendix B

MATLAB Codes for Solving 2d Poisson's Equation

bc.m

```
function [top,right,bottom,left] = bc(xm,ym)
% Boundary conditions
%
% by Wilson Au   2006
%

[Ny,Nx] = size(xm);

top = 0*ones(Ny,Nx);
bottom = 0*ones(Ny,Nx);
left = 0*ones(Ny,Nx);
right = 0*ones(Ny,Nx);
```

driverPoisson.m

```
%
% Solving 2d Poisson's equation in a unit square
%    -u'' = f       for  0 < x,y < 1
```

```
%       u = g       at x,y = 0,1
%
% required: shiftGrid.m, hybridMethod.m, solvePoisson.m,
%           f.m, bc.m, exactSol.m
%
% by Wilson Au   2006
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Choice of Method
%  1: Regular Finite Difference Method
%  2: Hybrid Monte-Carlo Finite Difference Method
choiceMethod = 1;

close all;

% Number of point on coarse grid
Nx = 24; Ny = 24;

if choiceMethod == 1        % Regular Finite Difference Method
    nx = 0; ny = 0;
    nNx = Nx; nNy = Ny;
else                % Hybrid Monte-Carlo Finite Difference Method
    % Number of point on fine grid
    nx = Nx; ny = Ny;
    nNx = (nx-2)*(Nx-1)+Nx; nNy = (ny-2)*(Ny-1)+Ny;
end

% Interpolation
Nxi = 550; Nyi = 550;

% Solving Poisson's equation
[p,t] = hybridMethod(Nx,Ny,nx,ny,[0 1],[0 1]);
```

```
% Interpolating the numerical solution
Xc = linspace(0,1,nNx); Yc = linspace(0,1,nNy);
[ymc,xmc] = ndgrid(Yc,Xc);
Xi = linspace(0,1,Nxi); Yi = linspace(0,1,Nyi);
[ymi,xmi] = ndgrid(Yi,Xi);
ui = interp2(xmc,ymc,p,xmi,ymi);

% Exact solution
exacti = exactSol(xmi,ymi);

% Calculating the error
di = abs(exacti-ui);
l_2_errori = sqrt(sum(sum(di.^2)));
l_inf_errori = max(max(di));

% Display the error
disp([' ']);
disp(['l-2 error = ',num2str(l_2_errori)]);
disp(['l-inf error = ',num2str(l_inf_errori)]);
disp(['CPU time = ',num2str(t), ' second(s)']);

% Plotting
figure(1), clf;
mesh(xmi,ymi,exacti); colorbar;
xlabel('x'); ylabel('y'); zlabel('u(x,y)');
title('Exact solution');
axis([0 1 0 1 0 1]);
shading interp;
view(-35,40)

figure(2), clf;
mesh(xmi,ymi,ui); colorbar;
```

```
xlabel('x'); ylabel('y'); zlabel('u(x,y)');
title('Numerical solution');
axis([0 1 0 1 0 1]);
shading interp;
view(-35,40)
```

exactSol.m

```
function sol = exactSol(xm,ym)
% Compute the exact solution
%
% by Wilson Au   2006
%

sol = 4*4*xm.*ym.*(1-xm).*(1-ym);
```

f.m

```
function sol = f(xm,ym)
% Forcing term of the Poisson's equation
%
% by Wilson Au   2006
%

sol = 32*xm.*(1-xm) + 32*ym.*(1-ym);
```

hybridMethod.m

```
function [p,t] = hybridMethod(Nx,Ny,nx,ny,Xd,Yd)
%
% Description:
%  Solving Poisson's equation with dirichlet b.c.
%  if nx == 0,
```

```
%       then perform Regular Finite Difference Method
%  if nx ~= 0,
%       then perform Hybrid Monte-Carlo Finite Difference Method
%
% Wilson Au     2006
%

tic;

% Coarse Grid
DX_c = (Xd(2)-Xd(1))/(Nx-1); DY_c = (Yd(2)-Yd(1))/(Ny-1);
X_c = [Xd(1):DX_c:Xd(2)]; Y_c = [Yd(1):DY_c:Yd(2)];
[ym_c,xm_c] = ndgrid(Y_c,X_c);
xms_c = xm_c; yms_c = ym_c;
xms_C = xms_c; yms_C = yms_c;

% Forcing term
f_c = f(xms_c,yms_c);
% Boundary condition
[gt,gr,gb,gl] = bc(xms_c,yms_c);

% Solving Poisson's equation on coarse grid
p_c = solvePoisson(f_c,Nx,Ny,gt,gr,gb,gl,xms_c,yms_c);

if nx == 0
    p = p_c;
    t = toc;
    return
end

% Store data
for i = 1:Nx
    for j = 1:Ny
```

```
        p((j-1)*(ny-1)+1,(i-1)*(nx-1)+1) = p_c(j,i);
    end
end

% Fixed DY_c, vary DX_c
for k = [-(nx-2):-1,1:(nx-2)]
    % Shift the coarse grid in x-direction
    dx_c = -k*DX_c/(nx-1);
    dy_c = 0*DY_c/(ny-1);
    [xms_c,yms_c] = shiftGrid(dx_c,dy_c,xm_c,ym_c);

    % Forcing term
    f_c = f(xms_c,yms_c);
    % Boundary conditions
    [gt,gr,gb,gl] = bc(xms_c,yms_c);

    % Solving Poisson's equation
    p_c = solvePoisson(f_c,Nx,Ny,gt,gr,gb,gl,xms_c,yms_c);

    % Store data
    for i = 2:Nx-1
        for j = 1:Ny
            p((j-1)*(ny-1)+1,(i-1)*(nx-1)+1+k) = p_c(j,i);
        end
    end
end

% Fixed DX_c, vary DY_c
for k = [-(ny-2):-1,1:(ny-2)]
    % Shift the coarse grid in y-direction
    dx_c = 0*DX_c/(nx-1);
    dy_c = -k*DY_c/(ny-1);
    [xms_c,yms_c] = shiftGrid(dx_c,dy_c,xm_c,ym_c);
```

```
    % Forcing term
    f_c = f(xms_c,yms_c);
    % Boundary conditions
    [gt,gr,gb,gl] = bc(xms_c,yms_c);

    % Solving the Poisson equation
    p_c = solvePoisson(f_c,Nx,Ny,gt,gr,gb,gl,xms_c,yms_c);

    % Store data
    for i = 1:Nx
        for j = 2:Ny-1
            p((j-1)*(ny-1)+1+k,(i-1)*(nx-1)+1) = p_c(j,i);
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For fine grid
for i = 1:Nx-1
    for j = 1:Ny-1
        % Fine grid
        dx_f = (xms_C(j,i+1)-xms_C(j,i))/(nx-1);
        dy_f = (yms_C(j+1,i)-yms_C(j,i))/(ny-1);
        X_f = [xms_C(j,i):dx_f:xms_C(j,i+1)];
        Y_f = [yms_C(j,i):dy_f:yms_C(j+1,i)];
        [ym_f,xm_f] = ndgrid(Y_f,X_f);

        % Forcing term
        f_f = f(xm_f,ym_f);

        % Initialize the boundary conditions
        top = zeros(ny,nx);
```

```
        bottom = zeros(ny,nx);
        left = zeros(ny,nx);
        right = zeros(ny,nx);

        % Boundary conditions
        top(1,1:nx) = p((j-1)*(ny-1)+1,(i-1)*(nx-1)+(1:nx));
        bottom(ny,1:nx) = p(j*(ny-1)+1,(i-1)*(nx-1)+(1:nx));
        left(1:ny,1) = p((j-1)*(ny-1)+(1:ny),(i-1)*(nx-1)+1);
        right(1:ny,nx) = p((j-1)*(ny-1)+(1:ny),i*(nx-1)+1);

        % Solving Poisson's equation on fine grid
        p_f = solvePoisson(f_f,nx,ny,top,right,bottom,left,xm_f,ym_f);

        % Store data
        p((j-1)*(ny-1)+(2:ny-1),(i-1)*(nx-1)+(2:nx-1)) = ...
                                                    p_f(2:ny-1,2:nx-1);
    end
end
t = toc;
```

shiftGrid.m

```
function [xms,yms] = shiftGrid(dx,dy,xm,ym)
%
% Description:
%  Perform shifting on the coarse grid
%
% Wilson Au     2006
%

xms = xm; yms = ym;

xms(:,2:end-1) = xm(:,2:end-1)-dx;
```

```
yms(2:end-1,:) = ym(2:end-1,:)-dy;
```

solvePoisson.m

```
function p = solvePoisson(f,Nx,Ny,top,right,bottom,left,xm,ym)
%
% Description:
%  Solving -u'' = f with dirichlet boundary conditions
%  by MATLAB '\'
%
% Wilson Au     2006
%

NxNy = Nx*Ny;

DXL = xm(floor(Ny/2),floor(Nx/2))-xm(floor(Ny/2),floor(Nx/2)-1);
DXR = xm(floor(Ny/2),floor(Nx/2)+1)-xm(floor(Ny/2),floor(Nx/2));
DYT = ym(floor(Ny/2),floor(Nx/2))-ym(floor(Ny/2)-1,floor(Nx/2));
DYB = ym(floor(Ny/2)+1,floor(Nx/2))-ym(floor(Ny/2),floor(Nx/2));
bX = 0.5*DXL*DXR^2 + 0.5*DXL^2*DXR;
bY = 0.5*DYT*DYB^2 + 0.5*DYT^2*DYB;
em = (DXL+DXR)/bX + (DYT+DYB)/bY;

f_vector = reshape(f, NxNy, 1);

to_vec = reshape(top,NxNy,1); bo_vec = reshape(bottom,NxNy,1);
le_vec = reshape(left,NxNy,1); ri_vec = reshape(right,NxNy,1);

f_vector(1:Ny:Ny*(Nx-1)+1) = em*to_vec(1:Ny:Ny*(Nx-1)+1);
f_vector(Ny:Ny:NxNy) = em*bo_vec(Ny:Ny:NxNy);
f_vector(1:Ny) = em*le_vec(1:Ny);
f_vector(Ny*(Nx-1)+1:NxNy) = em*ri_vec(Ny*(Nx-1)+1:NxNy);
```

```
emid = zeros(NxNy,1);
edyR = zeros(NxNy,1); edyL = zeros(NxNy,1);
edxR = zeros(NxNy,1); edxL = zeros(NxNy,1);

% Top-Left Corner
emid(1) = em; edyR(2) = 0; edxR(Ny+1) = 0;
% Bottom-Left Corner
emid(Ny) = em; edyL(Ny-1) =0; edyR(Ny+1) = 0; edxR(2*Ny) = 0;
% Top-Right Corner
emid(Ny*Nx-Ny+1) = em; edyL(Ny*Nx-Ny) = 0; edyR(Ny*Nx-Ny+2) = 0;
edx2L(Ny*Nx-2*Ny+1) = 0;
% Bottom-Right Corner
emid(NxNy) = em; edyL(NxNy-1) = 0; edxL(NxNy-Ny) = 0;

% Left boundary
for k = 2:Ny-1
    emid(k) = em;
    edyL(k-1) = 0; edyR(k+1) = 0; edxR(k+Ny) = 0;
end
% Right boundary
for k = Ny*Nx-Ny+2:NxNy-1
    emid(k) = em;
    edyL(k-1) = 0; edyR(k+1) = 0; edxL(k-Ny) = 0;
end
% Top boundary
for k = Ny+1:Ny:Ny*Nx-2*Ny+1
    emid(k) = em;
    edyL(k-1) = 0; edyR(k+1) = 0;
    edxL(k-Ny) = 0; edxR(k+Ny) = 0;
end
% Bottom boundary
for k = 2*Ny:Ny:Nx*Ny-Ny
    emid(k) = em;
```

```
    edyL(k-1) = 0; edyR(k+1) = 0;
    edxL(k-Ny) = 0; edxR(k+Ny) = 0;
end

for i = 2:Nx-1
    for j = 2:Ny-1
        DXL = xm(j,i)-xm(j,i-1); DXR = xm(j,i+1)-xm(j,i);
        DYT = ym(j,i)-ym(j-1,i); DYB = ym(j+1,i)-ym(j,i);
        bX = 0.5*DXL*DXR^2 + 0.5*DXL^2*DXR;
        bY = 0.5*DYT*DYB^2 + 0.5*DYT^2*DYB;
        emid((i-1)*Ny+j) = (DXL+DXR)/bX + (DYT+DYB)/bY;
        edyL((i-1)*Ny+j-1) = -DYB/bY; edyR((i-1)*Ny+j+1) = -DYT/bY;
        edxL((i-2)*Ny+j) = -DXR/bX; edxR(i*Ny+j) = -DXL/bX;
    end
end

A = spdiags([edxL, edyL, emid, edyR, edxR],...
                                [-Ny, -1, 0, 1, Ny], NxNy, NxNy);

sol_vector = A\f_vector; p = reshape(sol_vector,Ny,Nx);
```

Bibliography

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. 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*, SIAM, Philadelpia, 1994.

[2] R. L. Burden and J. D. Faires, *Numerical Analysis*, Brooks/Cole, Pacific Grove, seventh edition, 2001.

[3] K. L. Chung and Z. Zhao, *From Brownian Motion to Schrödinger's Equation*, Springer-Verlag, Berlin, 1995.

[4] R. Durrett, *Stochastic Calculus: A Practical Introduction*, CRC Press, Florida, 1996.

[5] M. Freidlin, *Functional Integration and Partial Differential Equations*, Princeton University Press, Princetion, 1985.

[6] J. R. Gilbert, C. Moler, and R. Schreiber, Sparse Matrices in MATLAB: Design and Implementation, *SIAM Journal on Matrix Analysis and Applications*, **13**, 1992, 333-256.

[7] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, Springer-Verlag, New York, 2004.

[8] K. Hardy, *Linear algebra for engineers and scientists using MATLAB*, Pearson Prentice Hall, New Jersey, 2005.

[9] C. O. Hwang, M. Mascagni and J. A . Given, A Feynman-Kac path-integral implementation for Poisson's equation using an h-conditioned Green's function, *Mathematics and Comuters in Simulation*, **62**, 1994, 347-355.

[10] I. Karatzas and S. E. Shreve, *Brownian Motion and Stochastic Calculus*, Springer-Verlag, New York, second edition, 1991.

[11] Y. D. Lyuu, *Financial Engineering and Computation: Principles, Mathematics, Algorithms*, Cambridge University Press, New York, 2002.

[12] M. J. Miranda and P. L. Fackler, *Applied Computational Economics and Finance*, The MIT Press, Massachusetts, 2002.

[13] B. Øksendal, *Stochastic Differential Equations: An Introduction with Applications*, Springer-Verlag, Berlin, fourth edition, 1995.

[14] D. L. Powers, *Boundary Value Problems*, Academic Press, San Diego, fourth edition, 1995.

[15] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelpia, second edition, 2003.

[16] W. A. Strauss, *Partial Differential Equations: An Introduction*, John Wiley & Sons, New York, 1992.

[17] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*, SIAM, Philadelpia, 1997.

[18] U. Trottenberg, C.W. Oosterlee and A. Schüller, *Multigrid*, Academic Press, London, 2001.

[19] W. Y. Yang, W. Cao, T. S. Chung and J. Morris, *Applied Numerical Methods Using MATLAB*, Wiley-Interscience, New Jersey, 2005.