2015-11-01

# Alternating Direction Implicit Method with Adaptive Grids for Modeling Chemotaxis in Dictyostelium discoideum

Christopher F. Loomis
*Brigham Young University - Provo*

# Alternating Direction Implicit Method with Adaptive Grids for Modeling Chemotaxis in *Dictyostelium discoideum*

Christopher F. Loomis

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mathematics

Brigham Young University

November 2015

## ABSTRACT

Alternating Direction Implicit Method with Adaptive Grids for Modeling Chemotaxis in *Dictyostelium discoideum*

Christopher F. Loomis
Department of Mathematics, BYU
Master of Science

*Dictyostelium discoideum* (Dd) is a model organism, studied for reasons from cell movement to chemotaxis to human disease control. Creating a computer model of the life cycle of Dd has garnered great interest, one part of which is the Aggregation Stage, where thousands of amoeba gather together to form a slug. Chemotaxis is the mechanism through which this is accomplished. This thesis develops two- and three-dimensional alternating direction implicit code which solves the diffusion equation on an adaptive grid. The calculated values for both two and three dimensions are checked against the actual solution and error results are provided. Comparisons are made between the coarse grid with refinement case and a fine grid without refinement case. Also, a non-negativity condition for two dimensions is derived to give a bound on the three major parameters: the diffusion coefficient and the spatial and time discretizations.

Keywords: *Dictyostelium discoideum*, Numerical Methods, ADI, Adaptive Grids, Non-negativity condition

## Acknowledgments

First and foremost, I would like to thank my wonderful wife, Jennie. Her patience, confidence, and support have meant the world to me. Without her, I would not have gotten very far in the program. She has been behind me 100% of the way and continues to inspire me.

I would also like to thank Dr. John Dallon for his tireless patience, help, and encouragement. He was always willing to meet and check my code, while at the same time making sure I did the work of debugging so that I could learn. I am a much better coder today because of his guidance and direction.

Next, I would like to thank Dr. Emily Evans. Through two classes and a year of wading through PDE's, she has been a wonderful teacher. I understand schemes, discretizations, and the math behind my code so much better due to her tutelage.

I am thankful to the Math Department for being so helpful at so many times I needed it. Dr. Michael Dorff was especially helpful to me, and Lonette Stoddard was invaluable.

My favorite editor, Daniel Smyth, was invaluable in helping this thesis to be as readable as it is, as well as encouraging me and keeping me going.

Lastly, I am indebted to my fellow students. As a student returning to math after twelve years, I needed much help getting back in the swing of things. Nick Callor, Lynnae Despain, Andrew Logan, Ryan Sandberg, Ben Schoonmaker, and Skyler Simmons were among the most prominent that I count as true friends and helpers.

# Contents

**Contents** **iv**

**List of Tables** **viii**

**List of Figures** **ix**

**1 Introduction** **1**

**2 Biological Overview** **3**

2.1 Vegetative Stage . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2.2 Aggregation Stage . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.3 Migration Stage . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.4 Culmination Stage . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Numerical Methods** **7**

3.1 Thomas Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.2 Numerical Methods for Solving the Diffusion Equation . . . . . . . . . . . . 9

3.3 Alternating Direction Implicit Method . . . . . . . . . . . . . . . . . . 10

3.4 The Peaceman-Rachford Algorithm . . . . . . . . . . . . . . . . . . . 12

3.5 Three-dimensional Generalized Douglas Scheme . . . . . . . . . . . . . . 14

3.6 Adaptive Grid Methods . . . . . . . . . . . . . . . . . . . . . . . . 15

**4 Two-Dimensional Algorithm** **18**

4.1 Overview of the Program . . . . . . . . . . . . . . . . . . . . . . . 19

4.2 Initialization of Major Variables . . . . . . . . . . . . . . . . . . . . 19

4.3 Initializing a Pulse . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.4 Chemical Diffusion . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.5 Refining the Grid . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.6 Coarsening the grid . . . . . . . . . . 24
4.7 Isolated Single Points . . . . . . . . . . 25
4.8 Isolated Pairs of Points . . . . . . . . . . 27
4.9 Making the Cell Move . . . . . . . . . . 28
4.10 Transitioning to Three Dimensions . . . . . . . . . . 28

**5 Results** **30**

5.1 Error Computations for the Two-Dimensional Case . . . . . . . . . . 30
5.2 Refinement Versus a Finer Grid Mesh . . . . . . . . . . 33
5.3 Error Computations for the Three-Dimensional Case . . . . . . . . . . 34
5.4 Nonnegativity Condition of the Parameters . . . . . . . . . . 36

**6 Description of Code Modules** **40**

6.1 Two-Dimensional Code . . . . . . . . . . 40
6.2 Project . . . . . . . . . . 40
6.3 Pulse . . . . . . . . . . 40
6.4 Move . . . . . . . . . . 40
6.5 Diffuse . . . . . . . . . . 41
6.6 Refine . . . . . . . . . . 41
6.7 Coarsen . . . . . . . . . . 41
6.8 Tridiagonal Matrix Solver . . . . . . . . . . 41
6.9 Isolated Single Point . . . . . . . . . . 42
6.10 Isolated Pair of Points . . . . . . . . . . 42
6.11 Three-Dimensional Code . . . . . . . . . . 42
6.12 Project . . . . . . . . . . 42
6.13 Pulse . . . . . . . . . . 42
6.14 Move . . . . . . . . . . 43
6.15 Diffuse . . . . . . . . . . 43

6.16 Refine . . . 43
6.17 Coarsen . . . 43
6.18 Tridiagonal Matrix Solver . . . 44
6.19 Isolated Single Point . . . 44
6.20 Isolated Pair of Points . . . 44
6.21 First Known Value . . . 44

**Appendices** **45**

**A Two-dimensional Code** **46**

A.1 Project . . . 46
A.2 Pulse . . . 50
A.3 Move . . . 51
A.4 Diffuse . . . 53
A.5 Refine . . . 62
A.6 Coarsen . . . 67
A.7 Thomas Algorithm . . . 69
A.8 Isolated Single Point . . . 70
A.9 Isolated Pair of Points . . . 72

**B Three-Dimensional Code** **75**

B.1 Project . . . 75
B.2 Pulse . . . 85
B.3 Move . . . 86
B.4 Diffuse . . . 88
B.5 Refine . . . 100
B.6 Coarsen . . . 109
B.7 Thomas Algorithm . . . 113
B.8 Isolated Single Point . . . 113

B.9 Isolated Pair of Points . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

B.10 First Known Value . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124

**Bibliography** **129**

# List of Tables

5.1 Two-dimensional Error without Refinement . . . . . . . . . . . . . . . . . . . 32

5.2 Two-dimensional Error with Refinement . . . . . . . . . . . . . . . . . . . . . 33

5.3 Three-dimensional Error without Refinement, $\epsilon = 0.02$ . . . . . . . . . . . . 35

5.4 Three-dimensional Error with Refinement, $\epsilon = 0.02$ . . . . . . . . . . . . . . 35

5.5 Three-dimensional Error without Refinement, $\epsilon = 0.0002$ . . . . . . . . . . . 35

5.6 Three-dimensional Error with Refinement, $\epsilon = 0.0002$ . . . . . . . . . . . . . 36

# List of Figures

2.1 The Life Cycle of *Dictyostelium discoideum* [9] . . . . . . . . . . . . . . . . . . 4

2.2 Aggregating cells [27] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.3 Subdivisions of the multicellular organism [9] . . . . . . . . . . . . . . . . . . . 7

3.1 The Grid Surrounding a Central Point with No Refinement . . . . . . . . . . 17

3.2 The Grid Surrounding a Central Point with Refinement Point 28 . . . . . . . 18

4.1 Labelling of Distances from the Center Point . . . . . . . . . . . . . . . . . . . 21

4.2 Examples of Refining the Grid . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.3 Single Isolated Point . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.4 Single Isolated Point with Shadow Points . . . . . . . . . . . . . . . . . . . . . 26

4.5 A Pair of Isolated Points with Shadow Points . . . . . . . . . . . . . . . . . . 27

5.1 Average Areas without Refinement . . . . . . . . . . . . . . . . . . . . . . . . 31

5.2 Average Areas with Refinement . . . . . . . . . . . . . . . . . . . . . . . . . . 32

5.3 Nonnegativity Check - Within the Bound, No Negative Values . . . . . . . . 38

5.4 Nonnegativity Check - Just Outside the Bound, One Negative Value . . . . . 39

5.5 Nonnegativity Check - Far Outside the Bound, Many Negative Values . . . . 39

# Chapter 1. Introduction

*Dictyostelium discoideum* (Dd) is a commonly studied slime mold; it has been named a model organism by the National Institutes of Health for the large number of things we can learn from it, including cell differentiation, chemotaxis and thermotaxis, programmed cell death, and DNA repair. Its life-cycle is divided into four parts, generally known as the Vegetative Stage, the Aggregation Stage, the Migration Stage, and the Culmination Stage. In the Aggregation Stage, thousands of Dd come together to form a slug, and the major process that guides this aggregation is chemotaxis – orientation or movement of an organism or cell in relation to chemical agents [14].

Chemotaxis is important to study because it helps us understand inflammation, arthritis, asthma, lymphocyte trafficking, and axon guidance in human bodies better. When tissue becomes inflamed, it releases chemotactic factors into the bloodstream. Neutrophils, a type of white blood cell, make contact with endothelial receptors attached to the inflamed tissue, which then cause the neutrophils to exit the bloodstream and enter the tissue. Similarly, in patients with rheumatoid and osteoarthritis, neutrophils are called into action. When the joint becomes inflamed, components of the surrounding synovial fluid attract the neutrophils to the irritated tissue [11]. In a similar way, neutrophils and eosinophils are used to ease inflamed tissue in the lungs of asthma sufferers, and they are guided by chemotaxins to the offending tissue [29].

Information processing work done in the brain is performed by an intricate network of connections between nerve cells, or neurons. The average adult human brain has over a trillion neurons, each connected to around a thousand other cells. This fascinating and essential part of the nervous system starts growing during embryonic development. A neuron will extend an axon to migrate to its synaptic target in the embryonic environment. Observations of this process show that axons approach their target regions in a very conventional and directed manner, with few errors along the way. It is quite amazing, as the distance most axons travel is more than a thousand times their own diameter. The key to the axon hitting its target is chemotaxis, both in the form of chemoattractants and chemorepellants, to make sure it arrives at the target site without hitting something else first [25].

The end goal, of which this is a part, is to create a computer model of the full life cycle of Dd. To accomplish this, code must be assembled for each of the different life stages of the slime mold. To advance the progress of this model, a program is needed to predict chemical concentration around the thousands of Dd as they pulse, in order to find the paths in which the Dd would naturally move. This thesis creates the necessary program, which solves the three-dimensional diffusion equation on an adaptive grid and that will allow us to diffuse throughout the grid the cyclic adenosine monophosphate (cAMP) that is used in the signalling process of chemotaxis.

In working with the numerics of the given physical problem,

$$u_t = b(\nabla^2 u) = b(u_{xx} + u_{yy}), \tag{1.1}$$

there is deserved concern that with such high concentrations of cAMP diffusing over small steps, there may be problems with high derivatives that could affect the accuracy of the numeric solution. One way to relieve this problem is to use an adaptive grid. Then, by altering the discretization of the grid, the calculations can have a predetermined precision and we can increase the efficiency of the work by getting values closer to the true ones. Instead of using the common method of mesh refinement, which involves creating a new mesh to mimic the area where more accuracy is desired, we work with single points being added to the grid. While other common methods create new domains with finer mesh sizes to get better precision, this method allows us to save on required computer memory. We won't be creating multiple grids to accommodate for the many extra grids that would be necessary, and in calculation times, as we would not be making calculations over the new grids.

This thesis is the culmination of creating and testing the desired code. In both two and three dimensions, an adaptive grid of predetermined size was created, over which the diffusion equation can be solved with a given number of point sources. Four different mesh sizes were used to determine the efficiency of the scheme and both the two- and three-dimensional schemes were found to be second-order accurate in space. A non-negativity condition was found for the two-dimensional scheme and checked to make sure it was accurate.

The biological background of Dd and its life cycle, as well as chemotaxis are discussed in Chapter 2. The numerical methods used in the program are laid out in Chapter 3. In Chapter 4 the two- and three-dimensional algorithms are discussed. Results for both algorithms are given in Chapter 5, a brief overview of the separate modules that make up the algorithms are found in Chapter 6, and the code is listed in the Appendices.

# Chapter 2. Biological Overview

*Dictyostelium discoideum* (Dd) is a cellular slime mold and a social amoeba that belongs to a group of life forms that can exist as single cells or multi-cellular organisms. They are found in a wide range of soil habitats containing decaying organic matter. They tend to live in the leaf litter of tropical to temperate forests, where they find bacteria to eat in ample supply [21]. Figure 2.1 shows the life cycle of Dd with approximate times, according to Gaudet, et al [9]. While this cycle could start at any point in the process, the standard practice is to begin with the unicellular amoebae and moves to the fruiting body. Section "A" represents the Vegetative Stage; "B" through "E" shows the Aggregation Stage; "F" and "G" are the Migration Stage; and "H" through "J" is the Culmination Stage. Each of these stages will be discussed in detail over the following four sections.

## 2.1 Vegetative Stage

As long as Dd has an adequate supply of bacteria on which to feed, it can live as undifferentiated, single cells, able to divide indefinitely by mitosis every 4 to 10 hours. The

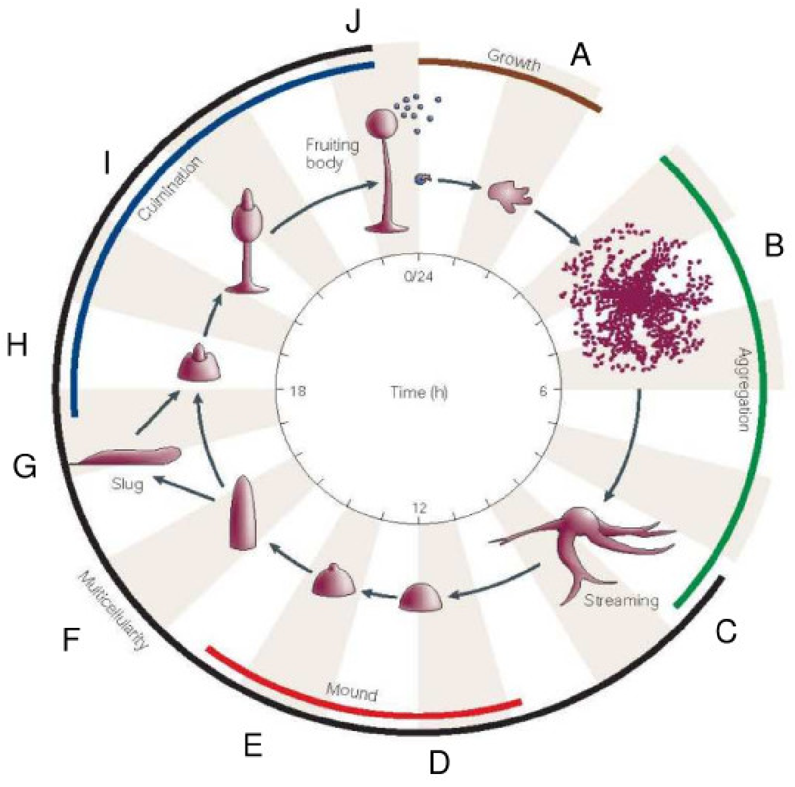

Figure 2.1: The Life Cycle of *Dictyostelium discoideum* [9]

cells intermittently monitor the amount of available food relative to their cell density by emitting a glycoprotein called prestarvation factor (PSF). When the ratio of PSF to food crosses a certain threshold, the cell recognizes its developmental phase of life is over and stops multiplying. The cell then initiates the expression of genes that are required for aggregation [21] [28].

Upon recognizing that their food source is dwindling, the cell will start to produce certain glycoproteins that will lead to aggregation. The prominent players in this group are cAMP-dependent protein kinase (PKA) and conditioned medium factor (CMF). CMF and PSF both stimulate gene expression and both signals make possible cAMP signalling by inducting genes involved in cAMP synthesis and detection. Among the newly activated genes are those that induce production of adenylate cyclase A, which synthesizes cAMP, cAMP receptors (the cARs) which detect extracellular cAMP, and the extracellular cAMP phosphodiesterase which hydrolyzes cAMP. These proteins, along with PKA and an intracellular cAMP phosphodiesterase with response regulator, form a biochemical network that will create the emissions of cAMP [21].

As these proteins begin to function inside the cells, one or more starving cells will emit a pulse of cAMP. The cAMP signal is relayed by cells nearby and the mold transitions to the Aggregation Stage.

## 2.2 Aggregation Stage

As the waves of cAMP begin rolling through the environment, individual cells react to it. They move toward the pulse, and the signal is amplified as they emit cAMP pulses themselves. The cluster of cells which spontaneously produce cAMP in a periodic fashion will become the "aggregation center"; chemotaxis and the relaying of the signal cause the rapid aggregation of up to 100,000 cells [21] [28]. Each of the cells has transmembrane cAMP receptors which exist in one of two states: active or inactive, determining whether the receptors can bind with extracellular cAMP, which stimulates the synthesis of cAMP in the interior of the cell, or not. The internal cAMP is secreted by the cell, and thus is carried back to the external cAMP receptor and stimulates it again, as well as surrounding cells [28]. After repetition of this process and extended exposure to high cAMP concentrations, the receptor will switch from active to inactive. This slows down the production of internal cAMP and ends the pulse, and the external cAMP begins to decay as it is not being replenished [18].

Dd movement toward the aggregation center is guided by the pulses of cAMP each cell detects [18]. The cells react to the pulses by moving toward the highest gradient of chemical. Cells around them do the same thing, so, naturally, there are a large number of cells moving in the same direction. As pulses continue, the Dd form streams and then rivers as they move toward the aggregation center as can be seen in Figure 2.2. Analysis has indicated that the collection into streams is reinforced and stabilized by the relaying of the cAMP signal [5].

The amoeba meet near the aggregation center and form a mound.

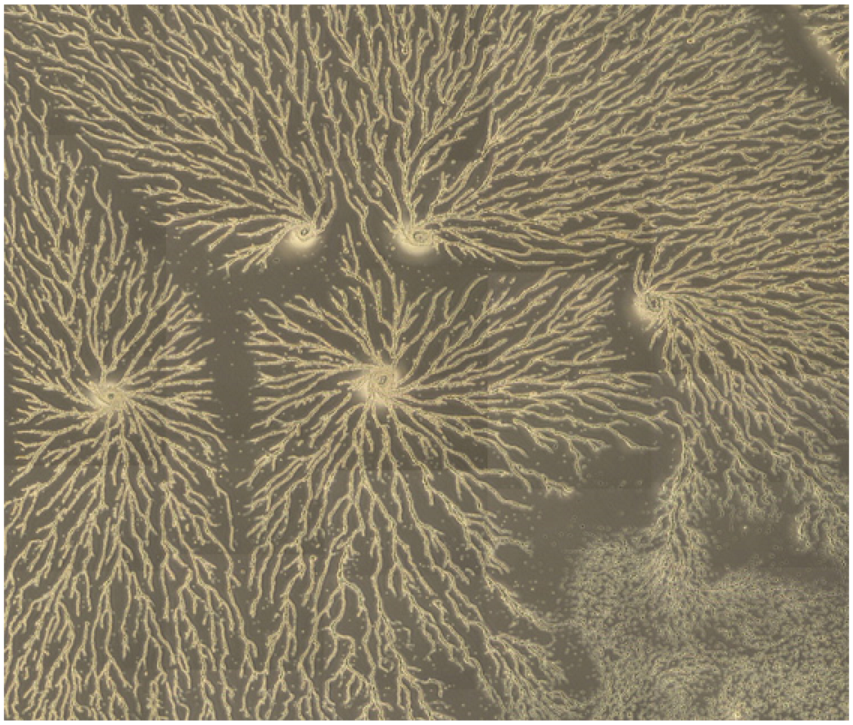

Figure 2.2: Aggregating cells [27]

## 2.3 Migration Stage

Research has shown that as the mound forms, the cells begin to differentiate themselves [21]. In Figure 2.3, the prestalk and prespore zones are recognizable from the tipped mound stage. This diagram represents the different subdivisions of the multicellular organism at the migratory slug stage. The subdivisions remain in the same relative positions and proportions until culmination.

Cell differentiation is controlled by extracellular cAMP acting on cARs to induce prespore differentiation in aggregates and by intracellular cAMP acting on PKA. On the other hand, a secreted polyketide called differentiation inducing factor I (DIF-1) has been shown to induce prestalk differentiation and stalk maturation in the presence of a PKA activator.

As the cells aggregate, they begin to form multicellular mounds. The upper part of the mound continues to send out pulses of cAMP and is pushed upward by the movement of cells

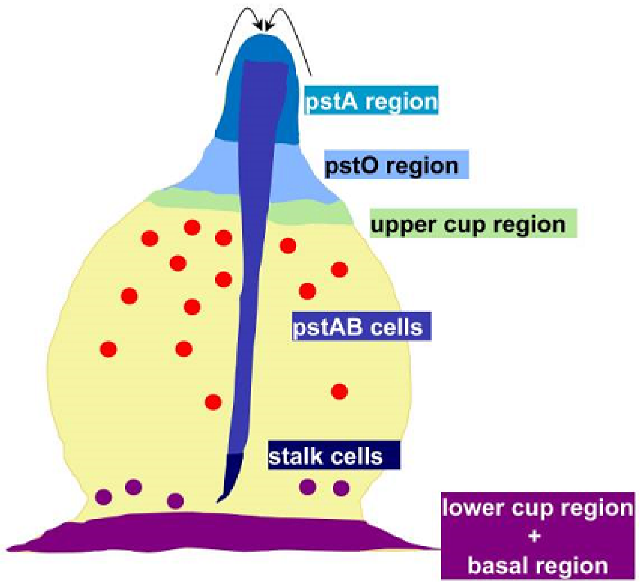

Figure 2.3: Subdivisions of the multicellular organism [9]

underneath it. Eventually the mound of cells topples over to become a pseudoplasmodium, more commonly termed a slug. Guided by light and heat, the slug begins its migration, generally toward the top layer of the soil [21], where it will find ample amounts of bacteria.

## 2.4 Culmination Stage

As the slug finds a new food source and this phase nears its end, cells near the tip of the slug begin to produce large quantities of cellulose that aid the slug in standing erect [15], and a “Mexican hat” forms. The anterior prestalk cells will slide down the middle of the cellulose tube, eventually forming the stalk. In the meantime, posterior prespore cells slide up the outside of the tube. After a six- to eight-hour process, the cells have created the desired fruiting body. Once the spores are mature, they will be dispersed, and the whole life cycle begins anew.

# Chapter 3. Numerical Methods

This chapter will give a brief overview of the numerical methods used in this thesis, as well as how we used and/ or adapted them to the study of the diffusion equation.

## 3.1 Thomas Algorithm

The Thomas algorithm is an efficient way to solve equations with a tridiagonal matrices. Developed by British mathematician Llewellyn Thomas around 1949, the algorithm is a version of Gaussian elimination used on a tridiagonal matrix. The method is to take a tridiagonal matrix equation,

$$\begin{bmatrix} b_1 & c_1 & 0 & ... & ... & 0 \\ a_2 & b_2 & c_2 & 0 & ... & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & 0 & ... & 0 & a_n & b_n \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ ... \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ ... \\ d_{n-1} \\ d_n \end{bmatrix},$$

and use the first equation to solve for $x_1$ in terms of $x_2$, that is $x_1 = \dfrac{d_1 - c_1 x_2}{b_1}$. Then substitute that expression for $x_1$ into the second equation and solve for $x_2$ in terms of $x_3$, and so on, down to the last equation. Next, substitute an expression for $x_{n-1}$ in terms of $x_n$ into the equation; at that point, there will be one equation with one variable and thus one can solve for $x_n$.

Turning around, there is now an equation for $x_{n-1}$ in terms of $x_n$, and since $x_n$ is known, so one can find $x_{n-1}$. In this way, it is possible move back up the list of equations for $x_i$ in terms of $x_{i+1}$ and solve each one for $x_i$.

There are some restrictions in using the Thomas algorithm. For each equation, there is division by the lead coefficient, so obviously, there cannot be a zero there. Also, in order for the algorithm to be stable, it is necessary for $|b_i| < |a_i| + |c_i|$ for all $i$. With the discretizations of the domain used in this thesis, these two requirements are met and so the Thomas algorithm is used freely.

## 3.2 Numerical Methods for Solving the Diffusion Equation

Considering Equation 1.1, there are numerous methods of solving the partial differential equation (PDE) over the desired domain. For instance, one could use the forward-time central-space method, where $u(x, y, n) = u_{xy}^n$ refers to the value at entry $[x, y]$ at time step $n$:

$$\frac{u_{xy}^{n+1} - u_{xy}^n}{k} = b\frac{u_{x-1y}^n - 2u_{xy}^n + u_{x+1y}^n}{h^2} + b\frac{u_{xy-1}^n - 2u_{xy}^n + u_{xy+1}^n}{h^2}, \tag{3.1}$$

where $k$ and $h$ are the time and space steps, respectively, and $b$ is the diffusion coefficient for all directions. For an $m \times m$ matrix and using Dirichlet boundary conditions, we let both $x$ and $y$ run from 2 to $m-1$ so as to leave the boundary values equal to zero. If we let $r = \dfrac{k}{h^2}$, then this equation simplifies to:

$$u_{xy}^{n+1} = (1 - 4br)u_{xy}^n + br(u_{x-1y}^n + u_{x+1y}^n + u_{xy-1}^n + u_{xy+1}^n), \tag{3.2}$$

which is first-order accurate in time and conditionally stable when solving the diffusion equation [20]. The difficulty with this method is that its conditional stability might produce issues when applied to an adaptive grid, as the stability ratio could grow too big, inducing instability[1].

Another common method of solving the diffusion equation is the Crank-Nicolson method, developed by John Crank and Phyllis Nicolson circa 1947. For the diffusion equation specifically, it is numerically stable. Based on the trapezoidal rule, it gives second-order convergence in time. Its discretized form, with diffusion coefficient $b$, $r = \dfrac{k}{2h^2}$, and both sides centered

[1]In mathematics, numerical stability addresses the stability of solutions of differential equations and of trajectories of dynamical systems under small perturbations of initial conditions. In other words, numerical stability ensures that small perturbations in the input, i.e., round-off or truncation errors, cause only small perturbations in the output. This tells us that, although we see error as compared to the actual solution, these errors do not grow as time progresses, but shrink. Stability is a key component in assuring that a scheme converges properly. Unconditional stability means that no matter the discretizations in time and space, the scheme is stable. The schemes used in this thesis have been shown to be unconditionally stable [4] [24].

on time step $n + \dfrac{1}{2}$, is:

$$(1+4br)u_{xy}^{n+1} - u_{x+1y}^{n+1} - u_{x-1y}^{n+1} - u_{xy+1}^{n+1} - u_{xy-1}^{n+1} = (1-4br)u_{xy}^{n} + u_{x+1y}^{n} + u_{x-1y}^{n} + u_{xy+1}^{n} + u_{xy-1}^{n}$$

The problem with Crank-Nicolson comes when we move to higher dimensions. The system of band-diagonal equations in higher dimensions, as compared to the tridiagonal ones in one dimension, would be very costly to solve via direct methods, making this method undesirable for use with Equation 1.1.

The alternating-direction implicit method (ADI) is a finite difference method of solving PDEs in higher dimensions, be they parabolic, hyperbolic, or elliptical. The ADI method was developed in the 1950's to simplify the issues involved with these two methods. The ADI method is also more successful in higher dimensions because it is an operator-splitting method, meaning that it takes a multi-dimensional operator and "splits" it into one-dimensional components. We detail the development of ADI here to demonstrate these points.

## 3.3 Alternating Direction Implicit Method

We consider Equation 1.1 on a square grid. Let $A_1$ and $A_2$ be linear operators such that $A_1 u = u_{xx}$ and $A_2 u = u_{yy}$. There is a nice substitution into Equation 1.1 to make $u_t = bA_1 u + bA_2 u$, which we can solve using the methods for simple, one-dimensional problems.

Adopting an idea taken from the Crank-Nicolson scheme, that of centering the difference scheme about $t = (n + \frac{1}{2})k$, we can examine the Taylor series for $u_t = bA_1 u + bA_2 u$:

$$\frac{u^{n+1} - u^n}{k} = \frac{b}{2}(A_1 u^{n+1} + A_1 u^n) + \frac{b}{2}(A_2 u^{n+1} + A_2 u^n) + O(k^2). \tag{3.3}$$

Solving for the $u^{n+1}$ terms, we get

$$(I - \frac{bk}{2}A_1 - \frac{bk}{2}A_2)u^{n+1} = (I + \frac{bk}{2}A_1 + \frac{bk}{2}A_2)u^n + O(k^3). \tag{3.4}$$

The matrix on the left-hand side of this equation would generally be costly to invert, and thus the Crank-Nicolson method would be less effective. We note that:

$$(1 \pm x)(1 \pm y) = 1 \pm x \pm y + xy. \tag{3.5}$$

We use this fact to work with both sides of Equation 3.4. Altering Equation 3.4 by adding $\frac{(bk)^2}{4} A_1 A_2 u^{n+1}$ to both sides and adding and subtracting $\frac{(bk)^2}{4} A_1 A_2 u^n$ on the right side in order to use Equation 3.5. This will complete the square on both sides. The result is

$$\begin{aligned}(I - \frac{bk}{2}A_1 - \frac{bk}{2}A_2 + \frac{(bk)^2}{4}A_1A_2)u^{n+1} =&(I + \frac{bk}{2}A_1 + \frac{bk}{2}A_2 + \frac{(bk)^2}{4}A_1A_2)u^n \\ &+ \frac{(bk)^2}{4}A_1A_2(u^{n+1} - u^n) + O(k^3).\end{aligned}$$

Now, we can factor the two sides, which results in:

$$\begin{aligned}(I - \frac{bk}{2}A_1)(I - \frac{bk}{2}A_2)u^{n+1} =&(I + \frac{bk}{2}A_1)(I + \frac{bk}{2}A_2)u^n \\ &+ \frac{(bk)^2}{4}A_1A_2(u^{n+1} - u^n) + O(k^3).\end{aligned}$$

Since we know $u^{n+1} = u^n + O(k)$, the $k^2$ factor makes the second term on the right side on the order of $k^3$, the order of existing errors on that side, and thus the equation becomes:

$$(I - \frac{bk}{2}A_1)(I - \frac{bk}{2}A_2)u^{n+1} =(I + \frac{bk}{2}A_1)(I + \frac{bk}{2}A_2)u^n + O(k^3).$$

Discretizing this equation creates a convenient method for solving successive time steps. In the case where $A_1 = u_{xx}$ and $A_2 = u_{yy}$, the matrices of the form $I - \frac{k}{2}A_i$ will be tridiagonal matrices, conveniently solved with the Thomas algorithm (Section 3.1). Let $A_{1h}$ and $A_{2h}$ be second-order approximations to $A_1$ and $A_2$, respectively. Then:

$$(I - \frac{bk}{2}A_{1h})(I - \frac{bk}{2}A_{2h})u^{n+1} =(I + \frac{bk}{2}A_{1h})(I + \frac{bk}{2}A_{2h})u^n + O(k^3) + O(kh^2).$$

It is from this equation that the general two-dimensional discretized ADI scheme for the diffusion equation is derived:

$$(I - \frac{bk}{2}A_{1h})(I - \frac{bk}{2}A_{2h})v^{n+1} = (I + \frac{bk}{2}A_{1h})(I + \frac{bk}{2}A_{2h})v^n. \tag{3.6}$$

## 3.4 The Peaceman-Rachford Algorithm

Now that we have Equation 3.6, we would like to split this two-dimensional problem into two one-dimensional problems that could be solved by moving implicitly in each direction or dimension. This is the origin of the name *alternating direction implicit method.* There are a number of ADI methods from which to choose, all using $A_{ih}$ to represent the second order difference over the given point in a specific direction, $i = 1$ for the $x$-direction and $i = 2$ for $y$. Thus, for the point $(x, y)$, $A_{1h} = u_{x+1y} - 2u_{xy} + u_{x-1y}$ will be used in approximating the second derivative over the point in the $x$-direction. Using $r = \frac{k}{h^2}$ where $k$ is the time discretization and $h$ is the spatial discretization, a method attributed to Jim Douglas, Jr., circa 1956 was:

$$\left[1 - \frac{bk}{2}(1 - \frac{1}{6r})\big(A_{1h} + A_{2h}\big)\right]v_{n+1} = \left[1 + \frac{bk}{2}(1 + \frac{1}{6r})\big(A_{1h} + A_{2h}\big) + \frac{(bk)^2}{6r}A_{1h}A_{2h}\right]v_n,$$

which may be solved by a sequence of intermediate solutions $\beta^{(1)}_{n+1}, \beta^{(2)}_{n+1} \equiv v_{n+1}$ using:

$$\left[1 - \frac{bk}{2}(1 - \frac{1}{6r})A_{1h}\right]\beta^{(1)}_{n+1} = \left[1 + \frac{bk}{2}(1 + \frac{1}{6r})A_{1h} + bkA_{2h}\big) + \frac{(bk)^2}{6r}A_{1h}A_{2h}\right]v_n,$$

and

$$\left[1 - \frac{bk}{2}(1 - \frac{1}{6r})A_{2h}\right]\beta^{(2)}_{n+1} = \beta^{(1)}_{n+1} - \frac{bk}{2}(1 - \frac{1}{6r})A_{2h}v_n.$$

This takes advantage of solving only tridiagonal systems of equations [8].

Another method developed by Douglas and Henry H. Rachford, Jr., around 1955 made use of an intermediate solution, $\hat{v}^{n+\frac{1}{2}}$ [7]:

$$A_{1h}\hat{v}^{n+\frac{1}{2}} + A_{2h}v^n = \frac{\hat{v}^{n+\frac{1}{2}} - v^n}{k},$$

and

$$A_{1h} v^{n+1} = A_{2h} v^n + \frac{v^{n+1} - \hat{v}^{n+\frac{1}{2}}}{k}.$$

The method chosen for the two-dimensional code used in this thesis was developed by the aforementioned Rachford and Donald W. Peaceman, in the mid-1950's [17]:

$$\big(I - \frac{bk}{2} A_{1h}\big) \tilde{v}^{n+\frac{1}{2}} = \big(I + \frac{bk}{2} A_{2h}\big) v^n, \tag{3.7}$$

$$\big(I - \frac{bk}{2} A_{2h}\big) v^{n+1} = \big(I + \frac{bk}{2} A_{1h}\big) \tilde{v}^{n+\frac{1}{2}}. \tag{3.8}$$

To show equivalence to Equation 3.6, we start with the left-hand side of Equation 3.6 and use Equations 3.8 and then 3.7, with $A_{1h}$ and $A_{2h}$ the opposite-directional second-order approximations, to see:

$$\begin{aligned} \big(I - \frac{bk}{2} A_{1h}\big)\big(I - \frac{bk}{2} A_{2h}\big) v^{n+1} &= \big(I - \frac{bk}{2} A_{1h}\big)\big(I + \frac{bk}{2} A_{1h}\big) \tilde{v}^{n+\frac{1}{2}} \\ &= \big(I + \frac{bk}{2} A_{1h}\big)\big(I - \frac{bk}{2} A_{1h}\big) \tilde{v}^{n+\frac{1}{2}} \\ &= \big(I + \frac{bk}{2} A_{1h}\big)\big(I + \frac{bk}{2} A_{2h}\big) v^n. \end{aligned}$$

Here, note that the variable $\tilde{v}^{n+\frac{1}{2}}$ should not be thought of as an actual approximation of $u(x, y, n)$ at any time, $n$, but as a temporary variable in the calculation of $v^{n+1}$. Also, note that the Peaceman-Rachford method is unconditionally stable [24].

Implementing the Peaceman-Rachford algorithm means utilizing Equations 3.7 and 3.8 to calculate $v^{n+1}$ from $v^n$. The general difference equations corresponding to Equations 3.7 and 3.8 are, respectively,

$$-\frac{br}{2} \tilde{v}_{x-1y}^{n+\frac{1}{2}} + (1 + br) \tilde{v}_{xy}^{n+\frac{1}{2}} - \frac{br}{2} \tilde{v}_{x+1y}^{n+\frac{1}{2}} = \frac{br}{2} v_{xy-1}^n + (1 - br) v_{xy}^n + \frac{br}{2} v_{xy+1}^n, \tag{3.9}$$

$$-\frac{br}{2} v_{xy-1}^{n+1} + (1 + br) v_{xy}^{n+1} - \frac{br}{2} v_{xy+1}^{n+1} = \frac{br}{2} \tilde{v}_{x-1y}^{n+\frac{1}{2}} + (1 - br) \tilde{v}_{xy}^{n+\frac{1}{2}} + \frac{br}{2} \tilde{v}_{x+1y}^{n+\frac{1}{2}}. \tag{3.10}$$

for $x = 2, ..., m-1$, and for $y = 2, ..., m-1$ with diffusion coefficient $b$ and the constant $r = \frac{k}{h^2}$. The system of equations for Equation 3.9 consists of $m - 2$ tridiagonal systems of equations, one for each value of $y$. We use the Thomas algorithm to solve them. Having calculated $\tilde{v}^{n+\frac{1}{2}}$, we move to the next stage. Similarly, the system of equations for Equation 3.10 consists of $m-2$ tridiagonal systems of equations, one for each value of $x$, and again, we use the Thomas algorithm to solve them and construct $v^{n+1}$. To do this, two-dimensional arrays for $v$ and $\tilde{v}^{n+\frac{1}{2}}$ are needed, as well as vectors for $a$, $b$, $c$, and the $d$ values. These vectors are the input for the Thomas algorithm; $a$, $b$, and $c$ are the coefficients of the $x-1$, $x$, and $x+1$ terms of $\tilde{v}^{n+\frac{1}{2}}$ on the left-hand side of Equation 3.9, respectively, while $d$ is generated by the known values on the right-hand side. The arrays hold all of the coefficients as $x$ runs from 2 to $m-1$. The Thomas algorithm takes these vectors and returns a row or column for $\tilde{v}^{n+\frac{1}{2}}$ or $v^{n+1}$, whichever is being solved.

## 3.5 Three-dimensional Generalized Douglas Scheme

Peaceman, Rachford, and Douglas, were the driving force in working on numerically solving two-dimensional PDE's in the 1950's. These methods were all extended to three-dimensional PDE's later that decade. In [26] we find a valuable comparison of nine of the most popular methods of solving PDE's in three-dimensions, of which Jules finds the Douglas method the best by the criteria used. They were compared with regards to relative accuracy, ease of programming, computation time, and core storage requirements [31]. One major reason for switching from Peaceman-Rachford is the fact that it is only conditionally stable for $n > 2$ dimensions, while the Douglas method is unconditionally stable. Specifically, the modified Douglas method developed by [4] Weizhong Dai is unconditionally stable and second order accurate in space if we choose our parameters carefully.

The involved equations use the following designations:

1) $\left(A_{1h} + A_{2h} + A_{3h}\right)$ is used to approximate $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} + \dfrac{\partial^2 u}{\partial z^2}$ where:

2) $A_{1h}u_{xyz} = \dfrac{1}{h^2}\left(u_{x+1yz} - 2u_{xyz} + u_{x-1yz}\right)$,

3) $A_{2h}u_{xyz} = \dfrac{1}{h^2}\left(u_{xy+1z} - 2u_{xyz} + u_{xy-1z}\right)$,

4) $A_{3h}u_{xyz} = \dfrac{1}{h^2}\left(u_{xyz+1} - 2u_{xyz} + u_{xyz-1}\right)$.

Similar to the two-dimensional case, where we split the algorithm into half-time steps, we divide the algorithm into third-time steps here, one for each dimension. As referenced in the paper and using the above designations, Li and Hong [4] wrote the Douglas alternating-direction implicit scheme as follows:

$$\big(1-\frac{br}{2}A_{1h}\big)u_{xyz}^{n+\frac{1}{3}}-u_{xyz}^{n}=br\big(A_{1h}+A_{2h}+A_{3h}\big)u_{xyz}^{n}, \tag{3.11}$$
$$u_{xyz}^{n+\frac{2}{3}}-u_{xyz}^{n+\frac{1}{3}}=\frac{br}{2}A_{2h}\big(u_{xyz}^{n+\frac{2}{3}}-u_{xyz}^{n}\big), \tag{3.12}$$
$$u_{xyz}^{n+1}-u_{xyz}^{n+\frac{2}{3}}=\frac{br}{2}A_{3h}\big(u_{xyz}^{n+1}-u_{xyz}^{n}\big), \tag{3.13}$$

where $r=\frac{k}{h^2}$ and $b$ is the diffusion coefficient. As written, the scheme has slow convergence when $r$ is large. Since a refinement will be necessary, we would like to be able to reduce $h$ without losing speed of convergence. In [4] a modified Douglas scheme for three dimensions using a new parameter $\epsilon$ is introduced. When $\epsilon=0$, the scheme becomes the original Douglas 3-D scheme. With $r=\frac{k}{h^2}$ as before and diffusion coefficient $b$, the three-dimensional scheme, with $\epsilon$ introduced, is:

$$\big(1-\frac{br-br\epsilon}{2}A_{1h}\big)\big(u_{xyz}^{n+\frac{1}{3}}-u_{xyz}^{n}\big)=br\big(A_{1h}+A_{2h}+A_{3h}\big)u_{xyz}^{n}+b^3(6\epsilon+2\epsilon^3)(A_{1h}A_{2h}A_{3h})u_{xyz}^{n} \tag{3.14}$$
$$u_{xyz}^{n+\frac{2}{3}}-u_{xyz}^{n+\frac{1}{3}}=\frac{br-br\epsilon}{2}A_{2h}\big(u_{xyz}^{n+\frac{2}{3}}-u_{xyz}^{n}\big) \tag{3.15}$$
$$u_{xyz}^{n+1}-u_{xyz}^{n+\frac{2}{3}}=\frac{br-br\epsilon}{2}A_{3h}\big(u_{xyz}^{n+1}-u_{xyz}^{n}\big) \tag{3.16}$$

The general discretized equation for Equation 3.14 is quite tedious to expand because of the last term, and thus the three equations will be omitted in this part. They can be found in the code in the FirstKnown module. For a brief overview, see Section 6.21, or find the full code in Appendix B.10. We do note that the left-hand side of each equation is identical-looking when solved for the next third time step, and the previous iteration is used to update the next step.

## 3.6 Adaptive Grid Methods

Adaptive grid methods have been around since the 1970's and used in conjunction with different types of problems [1]. Adaptive techniques have been used in problems of adaptive quadrature routines [6], variable-step variable-order ODE integrators [22], or the general process of picking a mesh size, solving on the mesh, adapting the mesh, resolving, and then repeating [12]. One of the first to use adaptive mesh refinement using finite difference was Joseph Oliger in 1979 [16], and his work was continued and expanded by Martha Berger and others in the early 1980's [1].

Programmers found out early on that refining the whole grid at once was costly in wall-clock time and memory. Localized mesh refinement advanced to creating a separate mesh that would mimic part of the original mesh, only with a finer discretization in both time and space. This process of creating new, finer meshes could be repeated any arbitrary number of times until the desired accuracy was achieved. The procedure for solving over each mesh would be systematic. If the refined mesh's discretization were half of the original mesh's discretization, then the algorithm would run twice as many times on the finer mesh using half of the time step, so that the time over which it was run would be equal on the coarser and finer meshes. The points on the coarser meshes would be updated with the values calculated on the finer meshes to provide better accuracy. Once a mesh was no longer needed, it could be discarded easily; in this way, computer memory would only be filled with required information. The number of calculations would be minimized to those necessary to achieve the desired accuracy.

Other ways of developing optimal grids were developed in the 1980's and 1990's, including variational grid generation techniques [2] [3] [23]. These techniques seek to improve on the Euler-Lagrange formulation of making an optimum grid on an irregular geometry. The adaptive direct variational grid generation approach seeks to control three properties of the grid: grid spacing, grid cell areas, and grid orthogonality. These properties are derived from the discrete geometry. Using a reference grid, one which is more simple than the physical grid but has the same properties, one can use the discrete geometry on the reference grid to create a grid with the desired properties on the irregular geometry.

In this thesis, the general procedure of mesh refinement has been used and modified so that specific points are added into the current mesh to improve accuracy around troublesome areas with high derivatives (see Section 4.5). The mesh was created as a grid with active and inactive points; the active ones are points initially used, while the inactive ones are places that can be used in refinement. Therefore, the matrix used was created with this idea in mind. The plan was to start off with a sparse matrix where every fourth point would be considered active, while the three in between would be unused unless it were necessary to refine there. It would require little extra memory and many fewer calculations than if extra matrices had been created to hold additional meshes. We use a pointer matrix that shows at any given point what the adjacent point is in any direction, to allowing points to be inserted relatively easily. A value is given to the new point by interpolating over the two nearby points, and we continue with the same algorithm as before. In Theorem 2.1 [13] of their paper, Mai and Wu showed that the accuracy when using such refinement is of first order in space. Thus, the accuracy of the scheme described here is of first order in space instead of second.

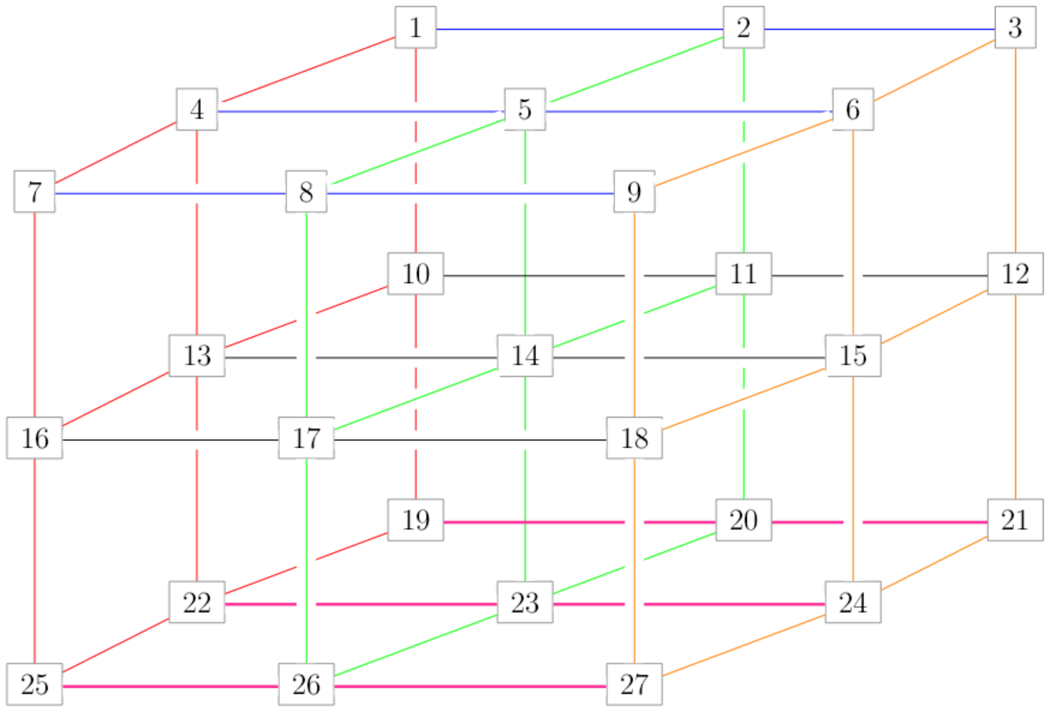

Figure 3.1: The Grid Surrounding a Central Point with No Refinement

For the adaptive grid, we had to make a decision about the term $(A_1A_2A_3)u^n_{xyz}$ in Equation 3.14. The operator utilizes the given point and its closest neighbor in all twenty-six directions, i.e., up and down, left and right, forward and backward, and all of the possible combinations. Think of it as three $3 \times 3$ squares of nine points as shown in the Figure 3.1, with the the point $[x, y, z] = 14$ in the center. On a static grid, this operator would give us little problem; imagine taking three movements – a positive movement in the $x$-direction toward 15, in the $y$-direction toward 11, and in the $z$-direction toward 5. No matter in which order they were done, one would always end up at

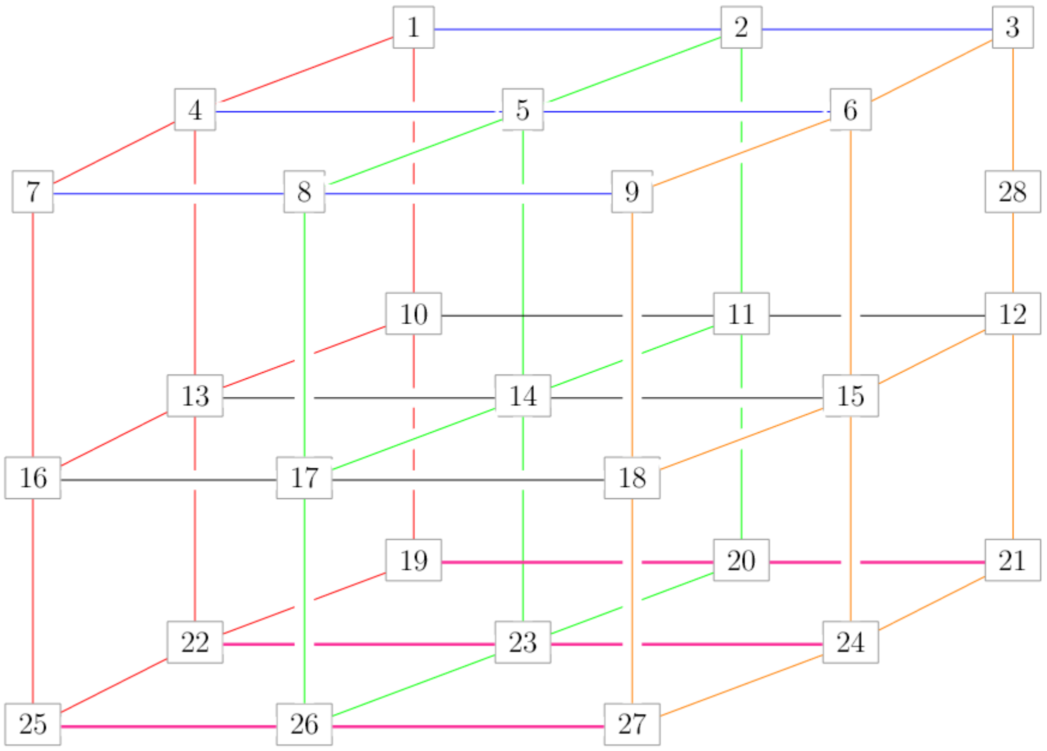

Figure 3.2: The Grid Surrounding a Central Point with Refinement Point 28

the point labelled 3.

But on a refined grid, the order in which we discretized might affect the outcome when acting on a given point, depending on whether there were refined points nearby. Imagine there were a refined point (28) between 3 and 12, as in Figure 3.2. Because of the refinement, starting at 14, moving positively in the $x$-direction towards 15, then positively in the $y$-direction towards 12, and lastly positively in the $z$-direction would land one on point 28. But doing the same three things in a different order ends differently. Again starting at 14, moving positively in the $y$-direction towards 11, then positively in the $z$-direction towards 2 and positively in the $x$-direction would end up on 3. The refined point there makes a difference, but it only becomes a problem based on the order of the movements, which comes from the order of discretization. We chose to discretize it in the order listed, by $z$, then $y$, and then $x$. Any order of discretization would present the same problems.

# Chapter 4. Two-Dimensional Algorithm

This chapter will give an overview of the two-dimensional code and summarize what each function inside the program does. For actual code, see Section A. We then briefly discuss transitioning from two dimensions to three.

## 4.1 Overview of the Program

The two-dimensional program consists of six basic parts: initializing variables, pulsing, moving, diffusing, refining, and coarsening. Diffusion is done using the Peaceman-Rachford method. At times within the diffusion function, situations arise that do not fit well with the P-R method, that require extra work to diffuse correctly. At those times, we call on two other functions that deal with individual points, or pairs of points that are isolated from other active points.

## 4.2 Initialization of Major Variables

The program (*Project*) starts by initializing some important values, including the time and space discretizations, as well as the diffusion coefficient. It takes as an input, $n$, the number of initial active points along one side of the matrix. It calculates the matrix size $m$ (for an $m \times m$ matrix) by adding three points in between each point, which will be potential refinement points, if necessary. Then the standard square matrix $v$ is created, initialized with zeros. When working with the matrix, it was easiest to treat is as Cartesian coordinates.

Next, the pointer matrix $PT$ is initialized by setting six values for each initial grid point:

1) the x-coordinate of the point to the right,

2) the distance to that point,

3) the x-coordinate of the point to the left,

4) the y-coordinate of the point above,

5) the distance to that point,

6) the y-coordinate of the point below.

Values are also set for one to three for the left and right boundary points, and values four through six for the lower and upper boundary points. Thus, along any row or column, when refined grid points are activated, they will always be between two defined points.

Lastly, the starting location(s) for the cell(s), or point sources, inside the matrix are initialized in the $p \times 2$ matrix $Loc$, where $p$ is the number of point sources.

The solution algorithm proceeds via the following series of steps: the cell moves, pulses at appropriate intervals, and repeats. In between, the cAMP is diffused, refinement is done, if necessary, and coarsening is also performed at intervals, if necessary.

## 4.3 Initializing a Pulse

A point source with value one is allowed to diffuse for twenty time steps using the actual analytical solution,

$$u(x, y, t) = \frac{1}{4\pi bkt} \exp\left( - \frac{\left((x - x_0)^2 + (y - y_0)^2\right)}{h^2} \cdot \frac{1}{4bkt} \right), \tag{4.1}$$

where $[x_0, y_0]$ is the location of the pulse, $b$ is the diffusion coefficient, $h$ is the spatial discretization, and $k$ is the time discretization. Then, as the initial conditions for the pulse, this analytic solution is inserted the values in the matrix around the location of the point source, truncated past the radius $\frac{1.1}{h}$. If pulsing successive times, the pulses after the first would add to the chemical at the points around the point source. This allows us to measure the accuracy of the program by initializing a pulse, diffusing the chemical for a number of time steps using our scheme, and checking the values we calculated against the actual solution at the proper time. Accurate error calculations can then be made by checking the difference between calculated and actual values point by point throughout the matrix.

## 4.4 Chemical Diffusion

The equation used to diffuse cAMP is

$$u_t = b(u_{xx} + u_{yy}) + g(t), \tag{4.2}$$

where $u_t$ is the derivative with respect to time, $b$ is the diffusion coefficient, $u_{xx}$ and $u_{yy}$ are the second derivatives in the $x$ and $y$ directions, respectively, and $g(t)$ is the time-dependent pulse function.

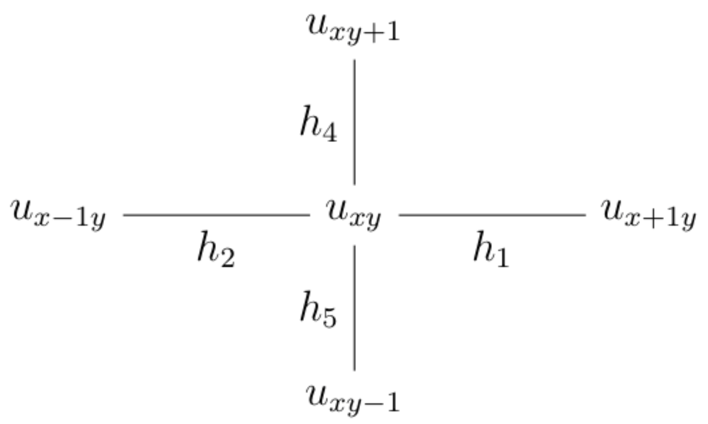

Figure 4.1: Labelling of Distances from the Center Point

We discretize the equation thusly, with $k$ as the time step and $h_i$ as the distance between the two points whose values being subtracted, as shown in Diagram 4.1. For $h_3$ and $h_6$, the averages of $h_1$ and $h_2$, and $h_4$ and $h_5$, respectively, are used:

$$u_t = b(u_{xx} + u_{yy}) + g(t), \tag{4.3}$$

$$\frac{v_{xy}^{n+1} - v_{xy}^n}{k} = b\left(\frac{\frac{v_{x+1y}^n - v_{xy}^n}{h_1} - \frac{v_{xy}^n - v_{x-1y}^n}{h_2}}{h_3} + \frac{\frac{v_{xy+1}^n - v_{xy}^n}{h_4} - \frac{v_{xy}^n - v_{xy-1}^n}{h_5}}{h_6}\right) + g(t), \tag{4.4}$$

$$v_{xy}^{n+1} = v_{xy}^n + bk\left(\frac{\frac{v_{x+1y}^n - v_{xy}^n}{h_1} - \frac{v_{xy}^n - v_{x-1y}^n}{h_2}}{h_3} + \frac{\frac{v_{xy+1}^n - v_{xy}^n}{h_4} - \frac{v_{xy}^n - v_{xy-1}^n}{h_5}}{h_6}\right) + kg(t). \tag{4.5}$$

We take as input the $v$ matrix and use the Peaceman-Rachford ADI method to solve for the next time step. We note that $r = \frac{k}{h^2}$ for Equations 4.6 and 4.7 and see that:

$$-\frac{br}{2}\tilde{v}_{x-1y}^{n+\frac{1}{2}} + (1+br)\tilde{v}_{xy}^{n+\frac{1}{2}} - \frac{br}{2}\tilde{v}_{x+1y}^{n+\frac{1}{2}} = \frac{br}{2}v_{xy-1}^n + (1-br)v_{xy}^n + \frac{br}{2}v_{xy+1}^n + kg(t), \tag{4.6}$$

$$-\frac{br}{2}v_{xy-1}^{n+1} + (1+d\mu)v_{xy}^{n+1} - \frac{br}{2}v_{xy+1}^{n+1} = \frac{br}{2}\tilde{v}_{x-1y}^{n+\frac{1}{2}} + (1-br)\tilde{v}_{xy}^{n+\frac{1}{2}} + \frac{br}{2}\tilde{v}_{x+1y}^{n+\frac{1}{2}}. \tag{4.7}$$

These are the equations generated. We let $x$ and $y$ run from 2 through $m-1$. We've had to modify Equations 4.6 and 4.7 slightly to account for the adaptive grid, the main difference being that $r$ is not constant but based on the two points over which the derivative is being calculated. The modified equations are below:

$$\begin{aligned}
-\frac{bk}{2h_2h_3}\tilde{v}^{n+\frac{1}{2}}_{x-1y}+&(1+\frac{bk(h_1+h_2)}{h_1h_2h_3})\tilde{v}^{n+\frac{1}{2}}_{xy}-\frac{bk}{2h_1h_3}\tilde{v}^{n+\frac{1}{2}}_{x+1y}\\
&=\frac{bk}{2h_5h_6}v^{n}_{xy-1}+(1-\frac{bk(h_4+h_5)}{h_4h_5h_6})v^{n}_{xy}+\frac{bk}{2h_4h_6}v^{n}_{xy+1}+kg(t), \qquad (4.8)\\
-\frac{bk}{2h_5h_6}v^{n+1}_{xy-1}+&(1+\frac{bk(h_4+h_5)}{h_4h_5h_6})v^{n+1}_{xy}-\frac{bk}{2h_4h_6}v^{n+1}_{xy+1}\\
&=\frac{bk}{2h_2h_3}\tilde{v}^{n+\frac{1}{2}}_{x-1y}+(1-\frac{bk(h_1+h_2)}{h_1h_2h_3})\tilde{v}^{n+\frac{1}{2}}_{xy}+\frac{bk}{2h_1h_3}\tilde{v}^{n+\frac{1}{2}}_{x+1y}. \qquad (4.9)
\end{aligned}$$

Here, the intermediate matrix $\tilde{v}^{n+\frac{1}{2}}$ is where we store the calculations after the first half time-step, which is used to solve for $v$ at the next time step. These equations lend naturally to setting up a tridiagonal matrix on the left-hand side that can be used with the Thomas algorithm to solve the system of equations. This algorithm can be used both when calculating $\tilde{v}$ from the current $v$, and also when calculating the new $v$ from $\tilde{v}$.

Now, how one time step looks in terms of the *Diffuse* function is shown. Let us assume we are working in a row, i.e., in the $x-$direction (although the algorithm works the same for the $y$-direction). We calculate first derivatives along the whole row. The forward derivative is calculated by finding the difference between the adjacent point forward along the row and the current point and then dividing by the distance between the two points. The backward derivative is similarly found, taking the current point value minus the value of the next point backward along the row and dividing by the distance between them.

Next, we set up the four vectors $(a, b, c, know)$ used by the Thomas algorithm. The first three arrays are set up with the coefficients from Equation 4.8 or 4.9, depending on whether we're solving for $\tilde{v}$ or $v$. The $a$ and $c$ matrices are the coefficients of the first and third terms on the left-hand side of the equation, respectively, and $b$ is assigned the coefficient of the middle term. The *know* vector is the calculated right-hand side. To set these up, let $x$ and $y$ run from 2 through $m-1$. The second derivative at each point is calculated as the difference of the two first derivatives, divided by the average of the distances used in the first derivatives. The second derivative is used as part of the *know* vector, added to the current concentration at that point. We set up the coefficients in each matrix to coordinate with each second derivative calculated.

Then the Thomas algorithm is used to solve the system of equations generated, and reinsert the calculated values of the solution into the correct matrix, be it $\tilde{v}$ or $v$, whichever is being solved for. This is repeated along all rows of $v^n$ first to find $\tilde{v}$, then along the columns of $\tilde{v}$ to find $v^{n+1}$ for the following time step. This is one iteration of the *Diffuse* function.

## 4.5 Refining the Grid

After each pulse or each time step the chemical is diffused as above, and a check is done to see if the grid needs refinement. To do so, run through each row and column of the matrix to check to see if the derivative between any two consecutive points in use is above a defined tolerance. If it is, the unused point half-way in between the two points is activated, as shown in Figure 4.2 in columns 5 and 93.

$$\begin{array}{ccccccccc}
v_{i,1} & v_{i,5} & \cdot & \cdot & v_{i,49} & \cdot & \cdot & v_{i,93} & v_{i,97} \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & v_{i+2,5} & \cdot & \cdot & \cdot & \cdot & \cdot & v_{i+2,93} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
v_{i+4,1} & v_{i+4,5} & \cdot & \cdot & v_{i+4,49} & \cdot & \cdot & v_{i+4,93} & v_{i+4,97}
\end{array}$$

Figure 4.2: Examples of Refining the Grid

There is a restriction placed on this process. Obviously, if the two points are right next to each other, maximal refinement has been reached and a new point cannot be activated between them. But two points that are “too far” away from each other should not be considered, as are $[i+2, 5]$ and $[i+2, 93]$ in Figure 4.2, points that are farther away from each other than the original discretization ($h$) between two active points in the matrix. This prohibits an increase in the error of the scheme by introducing a large spatial discretization.

So refinement is done by travelling along a row or column and checking the absolute value of the derivative between each pair of consecutive points that meet the above distance requirement. If the derivative is above the predetermined tolerance, the following actions are performed. Assume for this example that the movement is down column 5, and the points are where the above occurs, namely $[i, 5]$ and $[i+4, 5]$. First, a point halfway between the two points is established and initialized in the pointer matrix. For its chemical value in the matrix, it is assigned the average of the two identified points, so $v_{i+2,5} = \frac{v_{i,5} + v_{i+4,5}}{2}$. Next, the new pointers are established for the three points within the column, the two identified points, and the new point. Also, the point is inserted into row $i+2$ of the pointer matrix, finding the closest points to the right and left of it, and inserting it between them. In this case, they would both be the boundary points, as the refinement in column 93 would not yet have occurred and there are no other active points in the row. Last, all of the distances for the five points involved are adjusted - the four distances for the center point and the forward or backward distance for the other four points according to their location with respect to the center point. After finishing, the point after the refined point in the pointer matrix would be checked, in this case $[i + 4, 5]$, and it would be checked with the next point down the column.

## 4.6 Coarsening the grid

After every 10 time steps, the matrix is checked to see if coarsening is needed. As refining adds points to the grid, coarsening removes refined points that have become unnecessary as the simulation develops across the grid.

As with methods of refining a mesh, there are numerous ways of coarsening, also. One could look at the squares created by four of the original active points. If the removal of a refined point inside a square didn't change the average value of all points in the square, it could be removed. This idea could also be extended to refined points inside of smaller squares, or to a norm over the whole domain.

In this program, the grid is combed one row or column at a time and checked to see if the current point were a refined point. Then, if the derivatives in all directions from the refined point are below a defined tolerance, the refined point is removed. In doing so, the value in the matrix at the refined point is set to 0 and the pointer matrix adjusted to skip over the removed point in both the $x$ and $y$ directions.

## 4.7 Isolated Single Points

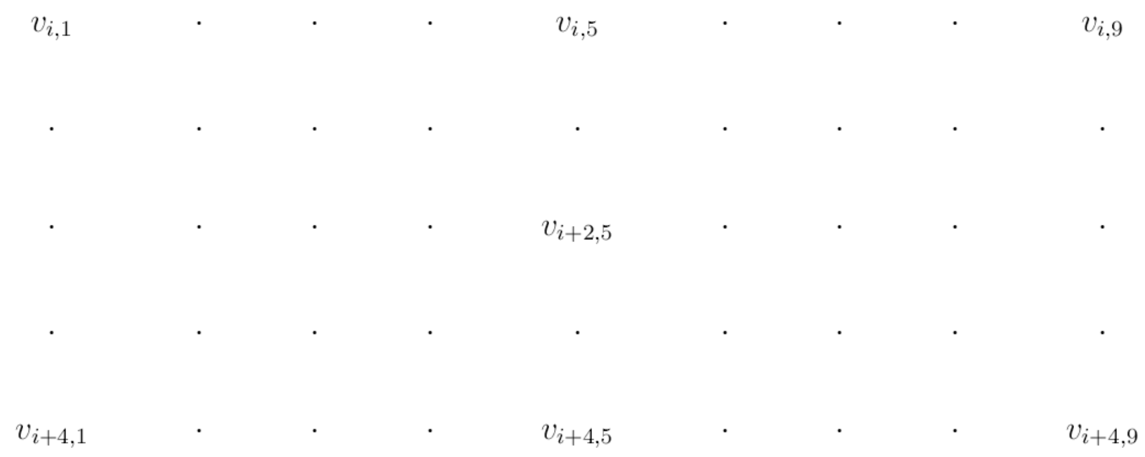

Figure 4.3: Single Isolated Point

Because we are refining by adding single points, a situation is sometimes created where in a row (or column) designated for refined points, there only appears one refined point in use, so the adjacent points according to the pointer matrix are the boundaries in one direction, and the two points that caused it to be created in the other. Another situation is where we have other points in the row, but they are too far away, i.e., farther away from each other than the original discretization of two points, to calculate together. The above diagram shows what either situation might look like, the point in question being $[i+2,5]$. Here, the six exterior points are original points. In row $i+2$, there is only one point; this might be because it is the only one in the row, or there might be another that is too far away to use in the calculation. Diffusing across the points in column 5 is no problem, but the other direction is troublesome. The ADI scheme is to be used to calculate values for the three points in row $i+2$ centered around the point in column 5, using the three points seen in column 5 ($i, i+2$, and $i+4$). The issue is that there are not three active points nearby, in the row, in which to insert the calculated values. This is because the boundary values are not changing, nor are new points being created around the isolated one.

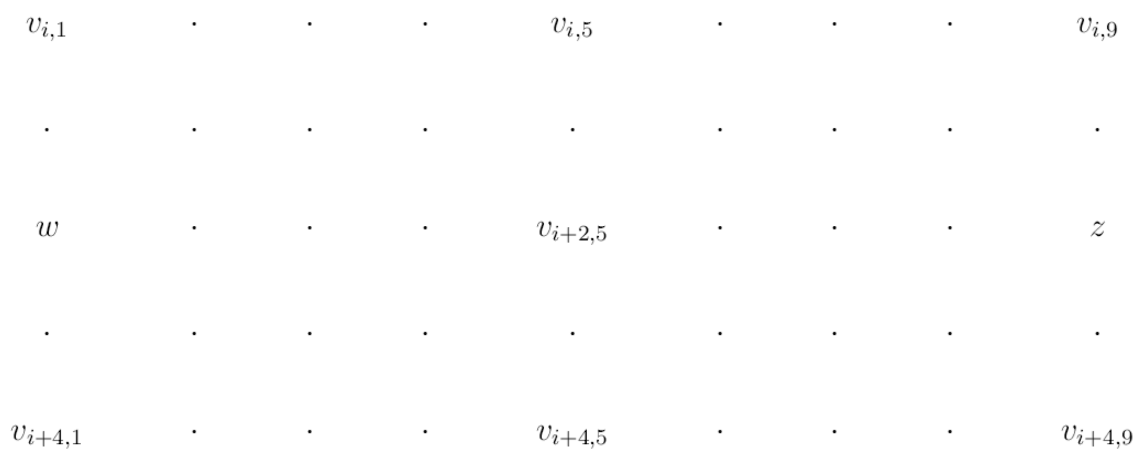

Figure 4.4: Single Isolated Point with Shadow Points

Thus, we modify our approach so that we are only solving for the one point in use, namely $\tilde{v}_{xy}^{n+\frac{1}{2}}$. To do this, we take $a = c = -\dfrac{d\mu}{2}$ and $b^{\pm} = (1 \pm d\mu)$ with subscripts differentiating direction to get:

$$a_1\tilde{v}_{x-1y}^{n+\frac{1}{2}} + b_1^+\tilde{v}_{xy}^{n+\frac{1}{2}} + c_1\tilde{v}_{x+1y}^{n+\frac{1}{2}} = a_0v_{xy-1}^n + b_0^-v_{xy}^n + c_0v_{xy+1}^n, \tag{4.10}$$

and solve it for $\tilde{v}_{xy}^{n+\frac{1}{2}}$. To do so, we calculate the right side as we normally do. We have the distances between those points, so we use them to calculate the coefficients of the right side. Next, we multiply the coefficients by the values at those points to get the known value. Then we estimate $w = \tilde{v}_{x-1y}^{n+\frac{1}{2}}$ and $z = \tilde{v}_{x+1y}^{n+\frac{1}{2}}$ using the closest rows and columns around the current point in the $v$ matrix. In Figure 4.3, we would take $w = \frac{1}{2}\left(v_{i,1}^n + v_{i+4,1}^n\right)$ and $z = \frac{1}{2}\left(v_{i,9}^n + v_{i+4,9}^n\right)$, creating a shadow point in the front and back. Then we multiply the shadow points by the correct coefficients, and move them to the other side. Dividing by $b_1^+$, we have:

$$\tilde{v}_{xy}^{n+\frac{1}{2}} = \frac{a_0v_{xy-1}^n + b_0^-v_{xy}^n + c_0v_{xy+1}^n - a_1w - c_1z}{b_1^+}. \tag{4.11}$$

In this way we can figure out the value of a single, isolated point at the next half-time step.

## 4.8 Isolated Pairs of Points

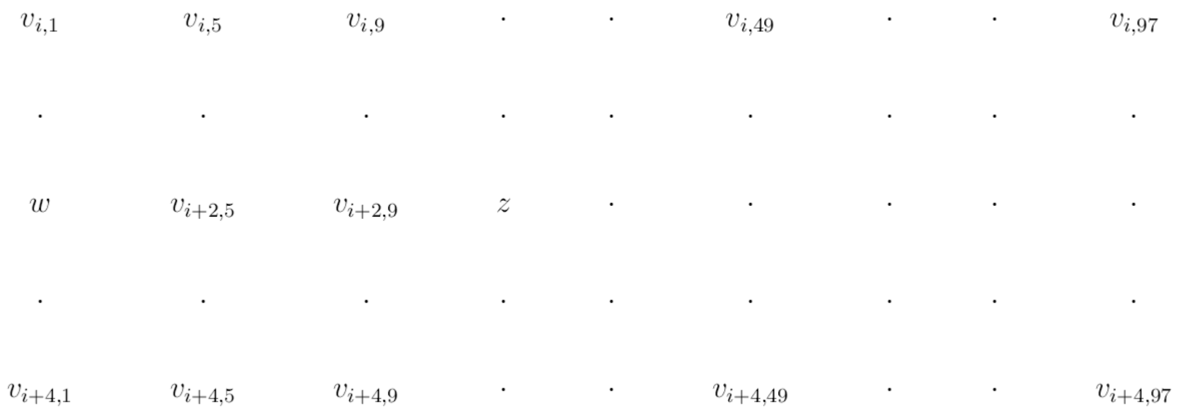

Figure 4.5: A Pair of Isolated Points with Shadow Points

As above, there are also times when refining the grid creates the problem of having two points ($v_{i+2,5}$ and $v_{i+2,9}$), close enough together to pair up but isolated from any others by distance, or they may be the only ones in that row as in Figure 4.5. As in Section 4.7, we will create shadow points – one ($z$) ahead of the front point ($v_{i+2,9}$), and one ($w$) behind the trailing point ($v_{i+2,5}$). We will differentiate between the two with superscript 1 for the new point, and 0 for the current point. In this situation, we are solving for two points, so we use the system of equations:

$$\begin{aligned} a_1 w^1 + b_1 v^1_{i+2,5} + c_1 v^1_{i+2,9} &= a_1 v^0_{i,5} + b_1 v^0_{i+2,5} + c_1 v^0_{i+4,5}, \\ &= a_1 v^0_{i,5} + b_1 x^0 + c_1 v^0_{i+4,5}, \\ a_2 v^1_{i+2,5} + b_2 v^1_{i+2,9} + c_2 z^1 &= a_2 v^0_{i,9} + b_2 v^0_{i+2,9} + c_2 v^0_{i+4,9}, \\ &= a_2 v^0_{i,9} + b_2 y^0 + c_2 v^0_{i+4,9}. \end{aligned}$$

So, $w^1 = \frac{1}{2}\left(v^0_{i,1} + v^0_{i+4,1}\right)$ and $z^1 = \frac{1}{2}\left(v^0_{i,13} + v^0_{i+4,13}\right)$ are approximated and moved over to the right-hand side to make:

$$\begin{aligned} b_1 v^1_{i+2,5} + c_1 v^1_{i+2,9} &= a_1 v^0_{i,5} + b_1 x^0 + c_1 v^0_{i+4,5} - a_1 w^1, \\ a_2 v^1_{i+2,5} + b_2 v^1_{i+2,9} &= a_2 v^0_{i,9} + b_2 y^0 + c_2 v^0_{i+4,9} - c_2 z^1. \end{aligned}$$

Here elimination can be used by multiplying the first equation by $-a_2$ and the second by $b_1$. This yields:

$$\begin{aligned}
-a_2 b_1 v^1_{i+2,5} - a_2 c_1 v^1_{i+2,9} &= -a_2\big(a_1 v^0_{i,5} + b_1 v^0_{i+2,5} + c_1 v^0_{i+4,5} - a_1 w^1\big),\\
a_2 b_1 v^1_{i+2,5} + b_1 b_2 v^1_{i+2,9} &= b_1\big(a_2 v^0_{i,9} + b_2 v^0_{i+2,9} + c_2 v^0_{i+4,9} - c_2 z^1\big).
\end{aligned}$$

Adding them together makes:

$$\begin{aligned}
\big(b_1 b_2 - a_2 c_1\big) y^1 &= -a_2\big(a_1 v^0_{i,5} + b_1 x^0 + c_1 v^0_{i+4,5} - a_1 w^1\big) + b_1\big(a_2 v^0_{i,9} + b_2 y^0 + c_2 v^0_{i+4,9} - c_2 z^1\big),\\
y^1 &= \frac{-a_2\big(a_1 v^0_{i,5} + b_1 x^0 + c_1 v^0_{i+4,5} - a_1 w^1\big) + b_1\big(a_2 v^0_{i,9} + b_2 y^0 + c_2 v^0_{i+4,9} - c_2 z^1\big)}{b_1 b_2 - a_2 c_1},
\end{aligned}$$

leaving an expression for $y^1$ using the current values and shadow points. Using this new value for $y^1$, one can back-substitute and solve for $x^1$. This is how an isolated pair of points is treated.

## 4.9 Making the Cell Move

The goal was to make an adaptive model for diffusion, so simplistic cell motion rules were used to test the scheme. A random number was used to pick a positive or negative direction, although the chance that 0 shows up has measure zero. Then a different randomly generated number is used as the number of spaces moved in the direction previously chosen, but with two stipulations that are checked before moving. It is checked to make sure that the cells are not moving onto the border of the matrix, and, if so, it is stoped on the row or column before the border. Also, it is checked to make sure that the cells are on a valid point, i.e., a point in use in the matrix. If it is set to move to an unused point, it is shifted to the closest active point. This process is repeated for each cell in the matrix, with newly generated random numbers each time the function is called.

## 4.10 Transitioning to Three Dimensions

Getting to a three-dimensional adaptive algorithm was a lot of work. It started in a class, working with two-dimensional ADI in a homework problem. From there, having chosen to do this as a thesis project, we started into three-dimensional ADI on a static grid.

The lengthy task of discretizing the three equations from Dai's [4] scheme was done. Over the course of a month or so, code was created that put into practice the discretization. Visualizing in three dimensions how the diffusion scheme worked took some time, and the result was thinking about it in standard $x-$, $y-$, and $z-$coordinates.

After the code was suitably accurate for the task given, the need was for a way to keep track of points in the adaptive part. Starting in one dimension, points could be easily added and the points behind and in front stored in a $3 \times 1$ array. Two dimensions became more difficult because the easiest method was to keep a nice square array; so adding points in the middle of what started as a square array would cause lopsidedness, or cause an increase in the dimensions of the array and move values around with every point added. The decision came to use a sparse matrix, considering every fourth point to be active initially and the three in-between to be points used for refinement. Using a pointer matrix, a point could just be "activated" by inserting it in the pointer matrix and giving it a value in the chemical matrix.

In the two-dimensional version, there were situations that one naturally doesn't see in one dimension. Here, the first one encountered was the problem of isolated single points or pairs of points. Adjustments had to be made to check and account for these anomalies. Once the code was running smoothly and accurately, three dimensions were attempted.

One of the major decisions in moving to three dimensions was choosing matrix sizes. In two dimensions, sizes could be chosen for the domain with little thought for computer limitations, despite the sometimes very large three-dimensional pointer matrix. In three dimensions, considerations had to be made about the size of the matrix and the size of the now four-dimensional pointer matrix. Using the same matrix sizes from the two-dimensional code would not work. For the larger ones, the computer would not even initialize the pointer matrix. So a second look had to be taken at our domain size and a reconsideration of how large it should be.

The other major decision came with the $(A_1A_2A_3)u_{xyz}$ term and deciding how to work with it. With no reason any specific order would work better than another, it was decided to do it in the order presented. It became a new module for the three dimensional code.

In terms of modules, there wasn't much difference between the two-dimensional and three-dimensional programs. The new equation to solve is

$$u_t = b(u_{xx} + u_{yy} + u_{zz}) + g(t). \tag{4.12}$$

The *Pulse* function required the three-dimensional analytical solution. Movement required two more random numbers to determine the direction and distance moved in the $z$-direction. There was the obvious necessity to change the *Diffuse* module as the scheme was changing, as shown in Section 3.5. The troublesome situations with isolated single or pairs of points occur in three-dimensions, also, so the functions were expanded to accommodate those situations. Again, the code is listed in Appendix B.

# Chapter 5. Results

In this chapter, the results of checking the calculations of the two- and three-dimensional codes against the analytic solutions for the appropriate number of dimensions are presented. Next, the non-negativity condition that depends on the diffusion coefficient and the space and time discretizations is explored for the two-dimensional ADI scheme used.

## 5.1 Error Computations for the Two-Dimensional Case

We considered the solution of Equation 1.1 both computationally and analytically. Initial conditions for the computational solution were set by considering the analytic solution at time $t = 20$ seconds within the radius $\frac{1.1}{h}$ ($h$ = spatial discretization) from the point source $[x_0, y_0]$, while outside the radius all values were set to zero. The initial value at any point $[x, y]$ within the radius would be:

$$u(x, y, 20) = \frac{1}{4\pi bk(20)} \exp\left( - \frac{\left((x-x_0)^2 + (y-y_0)^2\right)}{h^2} \frac{1}{4bk(20)} \right), \tag{5.1}$$

with diffusion coefficient $b = 0.003$ and time and spatial discretizations $k = 0.0001$ and $h$, respectively. The computational solution was run for 36 time steps and compared again to the actual solution at $t = 56$ and at each active point in the matrix.

One way to measure the amount of chemical in the domain is to look at the integral of the concentration. Without refinement, our method to estimate this integral over the square grid was to take the concentration at each point and multiply it by the area of the average rectangle around it. The area of the rectangle is found by taking the average horizontal and vertical distances to neighboring points and finding their product, as in Figure 5.1 below. We start with a concentration of one for each trial.

Figure 5.1 shows the matrix in blue. The active points are the gray dots; since there is no refinement, they are equally spaced. The colored squares represent the average rectangle around each point. These are found by taking the average horizontal and vertical distances to adjacent points and creating a rectangle centered on the point. The value at the grey point is the average concentration for the square.

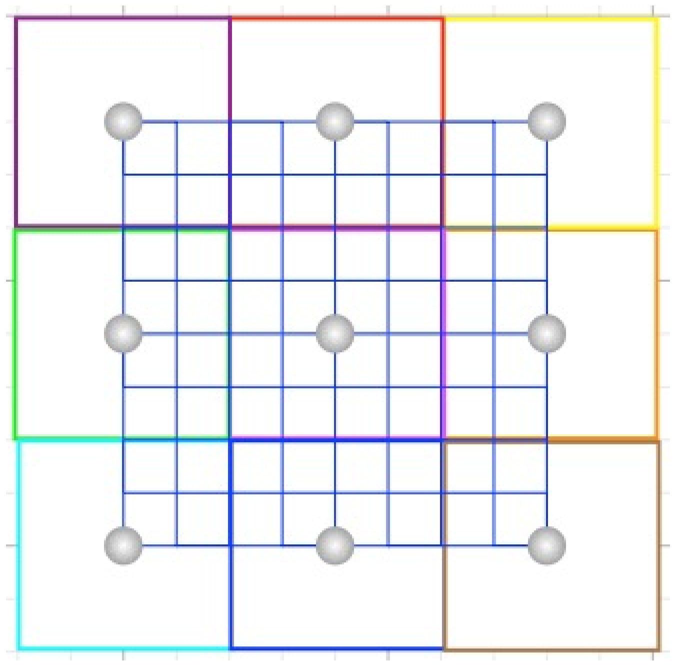
Figure 5.1: Average Areas without Refinement

For each table, $h$ is the spatial discretization; $m$ is the number of points along the edge of the matrix, where every fourth one starts as active and the three in-between are for refinement; the error is calculated here by moving through the matrix and summing over the area of the average rectangle times difference between the analytic solution ($act$) at that point and the calculated value ($v$) at that point, i.e., we find $h^2\ \Sigma_{x,y}|act(x,y)-v(x,y)|$; the next two columns are found by taking similar sums over the matrix of calculated solutions, and the sum over a matrix with the analytic solutions, both of which should be close to one for time small enough; the last column displays the ratio between the consecutive calculated errors. A scheme that is second-order accurate in space should have ratios of at least four.

| h | m | Error after $t = 36$ | $h^2 \times \Sigma(\text{v})$ | $h^2 \times \Sigma(\text{act})$ | Error Ratio |
|---|---|---|---|---|---|
| 0.00125 | 6401 | 0.004931584 | 1.000000000 | 1.000000000 | 4.066137 |
| 0.00250 | 3201 | 0.020052495 | 1.000000000 | 1.000000000 | 4.371305 |
| 0.00500 | 1601 | 0.087655581 | 1.000307066 | 1.000000000 | 5.603045 |
| 0.01000 | 801 | 0.491138184 | 1.409817206 | 1.005275068 | N/A |

Table 5.1: Two-dimensional Error without Refinement

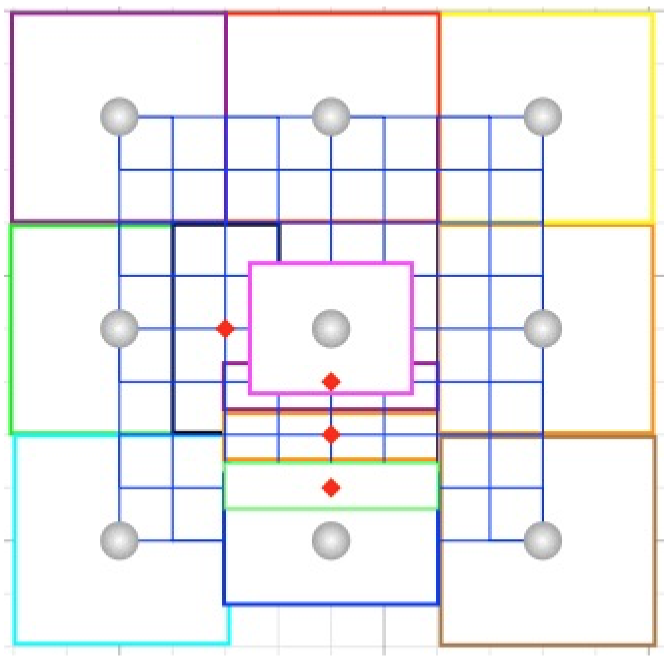

Figure 5.2: Average Areas with Refinement

Working with refinement, it was more difficult to analyze the accuracy of the scheme using the previous method where we looked at average rectangles to ascertain the integral over a given area. In Figure 5.1, we see that there is no trouble with an average rectangle around the grey points in use. With refinement, in Figure 5.2, the locations of the refined points (red diamonds) make this difficult. There are overlapping rectangles and places where no rectangle covers the space, making it nearly impossible to create the average rectangles whose areas sum to the area of the domain. Without extensive and expensive calculations, we cannot create the rectangles without adding extra area with the overlapping portions of the rectangles. And so instead of looking at an average area, we treat each point as if it were refined, so that the area around each point is $\left(\frac{h}{4}\right)^2$. The results are given in Table 5.2. Again, the third column used the difference between the calculated and actual solutions at each point. The fourth column used only the calculated solution, and the fifth the actual solution. The ratios in the sixth column are found the same way as in the first table. The last column shows how many refined points were activated during the process.

| h | m | Error after $t = 36$ | $(\frac{h}{4})^2 \times \Sigma(\text{v})$ | $(\frac{h}{4})^2 \times \Sigma(\text{act})$ | Error Ratio | Points Added |
|---|---|---|---|---|---|---|
| 0.00125 | 6401 | 0.004644024 | 0.995103761 | 0.998002053 | 3.567933 | 13982 |
| 0.00250 | 3201 | 0.016569566 | 0.985011465 | 0.993624912 | 3.967418 | 3372 |
| 0.00500 | 1601 | 0.065738400 | 0.984324385 | 0.998403694 | 5.447710 | 852 |
| 0.01000 | 801 | 0.358123717 | 1.355111014 | 0.997111294 | N/A | 216 |

Table 5.2: Two-dimensional Error with Refinement

The negative consequence of refinement is demonstrated in the Error Ratio column, where we see that second order accuracy in space is no longer guaranteed as we move from one value of $h$ to another.

## 5.2 Refinement Versus a Finer Grid Mesh

One of the things tested using the two-dimensional code was whether the refinement was worth it or not. Can refinement on a coarser grid adequately replace using a finer mesh? Because refinement is used at the expense of some accuracy, the finer mesh is more accurate. In run time, though, refinement is immensely faster.

Error was calculated by looking at the difference between each point of the refined grid and the corresponding point from the finer mesh. The difference was multiplied by the area of the average rectangle of a refined point on the coarser grid, then we summed over all of the points.

Five point sources were used, each pulsing at different times throughout the run-through. The finer grid had spatial discretization $h = 0.00125$, while the coarser grid had $h = 0.01$. In one run, indicative of the others performed, there were 1080 refined points created during the 36 time steps taken. The overall error for each point source used was about 0.514. The time comparison was 18.137 seconds versus 670.901 seconds, or a little over 11 minutes, just a hair under 37 times longer to run.

## 5.3 Error Computations for the Three-Dimensional Case

We considered the solution of Equation 4.12 both computationally and analytically. Initial conditions for the computational solution were set by considering the analytic solution at time $t = 100$ seconds within the radius $40/h$ from the point source $[x_0, y_0, z_0]$, while outside the radius all values were set to zero. Then the initial value at any point $[x, y, z]$ within the radius would be

$$u(x,y,z,100) = \frac{1}{(4\pi bk(100))^{\frac{3}{2}}} \exp\left( - \frac{\left((x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2\right)}{h^2} \frac{1}{4bk(100)} \right), \qquad (5.2)$$

with diffusion coefficient $b$ and time and spatial discretizations $k$ and $h$, respectively. The computational solution was then run for 36 time steps and compared again to the actual solution at $t = 136$ and at each active point in the matrix.

The errors are calculated in a similar manner to the two-dimensional code, moving through the matrix and finding the volume of the average rectangular prism and multiplying by the value at the point.

| h | m | Error after t=36 | $h^3 \times \Sigma$(v) | $h^3 \times \Sigma$(act) | Error Ratio |
|---|---|---|---|---|---|
| 0.00125 | 401 | 0.00622260 | 0.999999999 | 1.000000000 | 3.937277 |
| 0.00250 | 201 | 0.02450010 | 1.000000000 | 1.000000000 | 4.210928 |
| 0.00500 | 101 | 0.10316816 | 1.000460634 | 1.000000000 | 6.429091 |
| 0.00943 | 53 | 0.66327744 | 1.480521019 | 1.003485248 | N/A |

Table 5.3: Three-dimensional Error without Refinement, $\epsilon = 0.02$

| h | m | Error after $t = 36$ | $(\frac{h}{4})^3 \times \Sigma$(v) | $(\frac{h}{4})^3 \times \Sigma$(act) | Error Ratio | Points Added |
|---|---|---|---|---|---|---|
| 0.00125 | 401 | 0.00611102 | 0.999814362 | 0.999330003 | 3.990098 | 2440376 |
| 0.00250 | 201 | 0.02438359 | 0.999522421 | 0.992771031 | 3.876924 | 278752 |
| 0.00500 | 101 | 0.09453331 | 0.997534297 | 1.000486572 | 4.768217 | 40035 |
| 0.00943 | 53 | 0.45075535 | 1.450327378 | 0.999572033 | N/A | 6683 |

Table 5.4: Three-dimensional Error with Refinement, $\epsilon = 0.02$

These first two tables show the difference between the scheme with and without refinement and $\epsilon = 0.02$. The next two give error measurements with $\epsilon = 0.0002$. The parameter $\epsilon$ was varied to see if the paper [4] was right in claiming that there was improvement in the code if $\epsilon$ were on the order of $k$. When not using refinement, this was found to be true.

| h | m | Error after $t = 36$ | $h^3 \times \Sigma$(v) | $h^3 \times \Sigma$(act) | Error Ratio |
|---|---|---|---|---|---|
| 0.00125 | 401 | 0.00605348 | 0.999999999 | 1.000000000 | 4.022268 |
| 0.00250 | 201 | 0.02434871 | 1.000000000 | 1.000000000 | 4.232553 |
| 0.00500 | 101 | 0.10305719 | 1.000460634 | 1.000000000 | 6.435854 |
| 0.00943 | 53 | 0.66326104 | 1.480521019 | 1.003485248 | N/A |

Table 5.5: Three-dimensional Error without Refinement, $\epsilon = 0.0002$

| h | m | Error after $t = 36$ | $(\frac{h}{4})^3 \times \Sigma(\text{v})$ | $(\frac{h}{4})^3 \times \Sigma(\text{act})$ | Error Ratio | Points Added |
|---|---|---|---|---|---|---|
| 0.00125 | 401 | 0.00624376 | 0.999810825 | 0.999278030 | 3.925679 | 2440000 |
| 0.00250 | 201 | 0.02451100 | 0.999520162 | 0.992771041 | 3.860080 | 278459 |
| 0.00500 | 101 | 0.09461440 | 0.997528669 | 1.000489038 | 4.764135 | 39971 |
| 0.00943 | 53 | 0.45075575 | 1.450327768 | 0.999572021 | N/A | 6678 |

Table 5.6: Three-dimensional Error with Refinement, $\epsilon = 0.0002$

## 5.4 Nonnegativity Condition of the Parameters

If the initial conditions are non-negative, the solution to the diffusion equation should remain non-negative for all time $t$. Although the scheme is unconditionally stable, there is a non-negativity condition that must be met and the desire was to find the bound. A bound on the parameters used in the computation, namely the diffusion coefficient, the time discretization, and the space discretization was calculated. To do this, combinations of them were sought that would force the matrix to have negative values in future time steps.

To start, let $r = \frac{k}{2h^2}$. Then the coefficient matrix for the left-hand side of the system of equations generated by the ADI equations looks like this:

$$\begin{bmatrix} 1+2br & -br & 0 & ... & ... & 0 \\ -br & 1+2br & -br & 0 & ... & 0 \\ 0 & -br & 1+2br & -br & 0 & ... \\ 0 & ... & ... & ... & ... & 0 \\ 0 & ... & 0 & -br & 1+2br & -br \\ 0 & 0 & ... & 0 & -br & 1+2br \end{bmatrix}.$$

The initial conditions are a nonnegative chemical value in the middle of the domain, represented by a seven-by-seven matrix, the nonzero value $A$ in the middle. Then a bound on $br$ is calculated using the Peaceman-Rachford Algorithm. After the first half-time step, moving along the columns first, the matrix goes from:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & A & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \text{ to } \frac{A}{2(br)^4+16(br)^3+20(br)^2+8br+1} \times$$

$$\begin{bmatrix} 0 & 0 & (br)^4 & -2(br)^4+(br)^3 & (br)^4 & 0 & 0 \\ 0 & 0 & 2(br)^4+(br)^3 & -4(br)^4+(br)^2 & 2(br)^4+(br)^3 & 0 & 0 \\ 0 & 0 & 3(br)^4+4(br)^3+(br)^2 & -6(br)^4-5(br)^3+2(br)^2+br & 3(br)^4+4(br)^3+(br)^2 & 0 & 0 \\ 0 & 0 & 4(br)^4+10(br)^3+6(br)^2+br & -8(br)^4-16(br)^3-2(br)^2+4br+1 & 4(br)^4+10(br)^3+6(br)^2+br & 0 & 0 \\ 0 & 0 & 3(br)^4+4(br)^3+(br)^2 & -6(br)^4-5(br)^3+2(br)^2+br & 3(br)^4+4(br)^3+(br)^2 & 0 & 0 \\ 0 & 0 & 2(br)^4+(br)^3 & -4(br)^4+(br)^2 & 2(br)^4+(br)^3 & 0 & 0 \\ 0 & 0 & (br)^4 & -2(br)^4+(br)^3 & (br)^4 & 0 & 0 \end{bmatrix}.$$

Because of the symmetry in the above matrix, we note here that the second step of the algorithm produces a matrix that is symmetric over the middle row and the middle column. Values calculated along the first row will be the same as those of the last row, and the values calculated from the front of the first row will be the same as those of the end of the first row, although they would be in the opposite order. So when looking at the nonnegativity condition, we need only look at the $4 \times 4$ matrix in the upper left-hand corner. Thus, we calculate along the rows of the matrix above and consider the upper corner. For each of the fractions not in the last row or column of this $4 \times 4$ matrix, $r > 0$ is all that is necessary for the values to remain nonnegative. It is only in the numerators of the fractions in the last row and column that an upper bound is required. These four distinct values are all multiplied by $\dfrac{A}{(2(br)^4+16(br)^3+20(br)^2+8br+1)^2}$, which is a positive number, and they are: $-2(br)^3(2(br)^4+8(br)^3-4br-1), -2(br)^2(2br+1)(2(br)^4+8(br)^3-4br-1), -2br(2(br)^4+8(br)^3-4br-1)(3(br)^2+4br+1)$, and $(2(br)^4+8(br)^3-4(br)-1)^2$. Note here that the factors $2br+1$ and $3(br)^2+4br+1$ are always positive when $br > 0$, so the concern comes from the other factor they all share: $2(br)^4+8(br)^3-4(br)-1$. One factor has this polynomial squared, while the other three also have a $-2$ on the front. So we consider where this polynomial is actually negative, while keeping $br > 0$. The nonnegativity bound on the polynomial is $0 < br < -1 + \sqrt{\frac{3}{2}} + \sqrt{\frac{1}{2}(3-\sqrt{6})} \approx 0.74939249$.

The bound has been checked using the following parameters: $A = 1$, $b = 0.003$, $k = 0.0001$ and $h = .01$ with $d = 1.48$, $d = 1.5$, and then $d = 15$, with results shown in Figures 5.3, 5.4, and 5.5, respectively. The lowest value in the matrix is plotted after each time step, with the values recorded on the $y$-axis and time along the $x$-axis. The reasoning behind the first two choices of $d$ was to check $br = .74$ and then $br = .75$, values just on either side of the bound. The first gave no negative values, while the second gave a negative value on the first time step, but none thereafter. With $d = 15$, making $br = 7.5$, it was easy to see that the negative values would manifest for more time steps. To check that the scheme was stable, it was important to let it run sufficiently long to check that the negativity disappeared. This was done and worked correctly. Even though we started with all nonnegative values in all three cases, the second and third show the necessity for "sufficiently smooth" [10] [19] [30] conditions at the outset, given any parameter set.

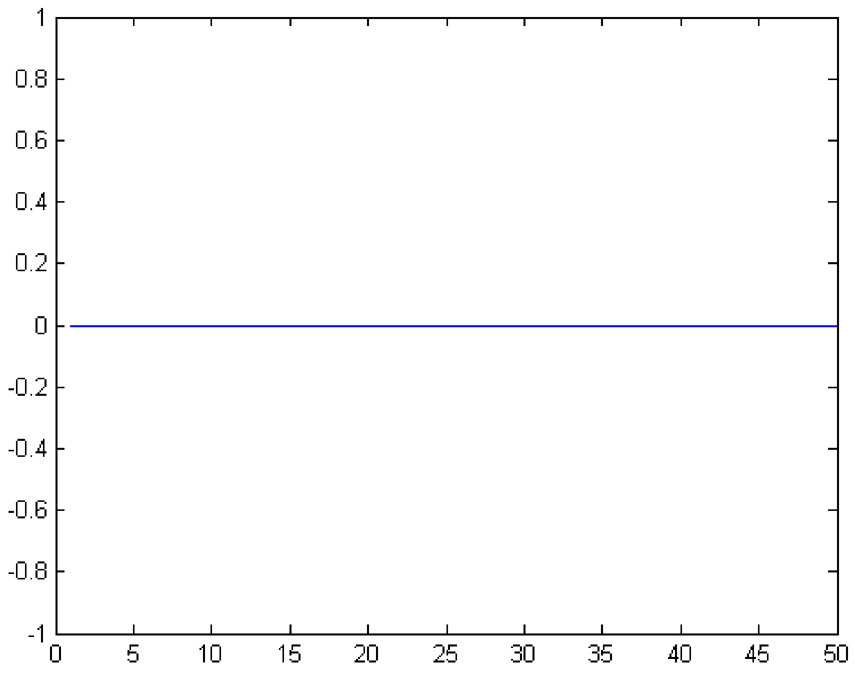

Figure 5.3: Nonnegativity Check - Within the Bound, No Negative Values

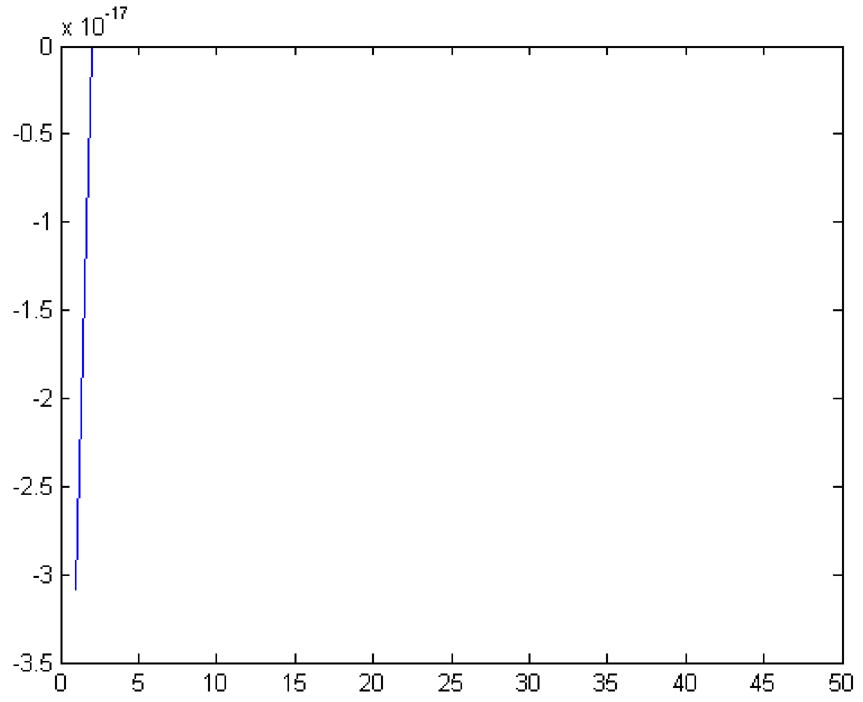

Figure 5.4: Nonnegativity Check - Just Outside the Bound, One Negative Value

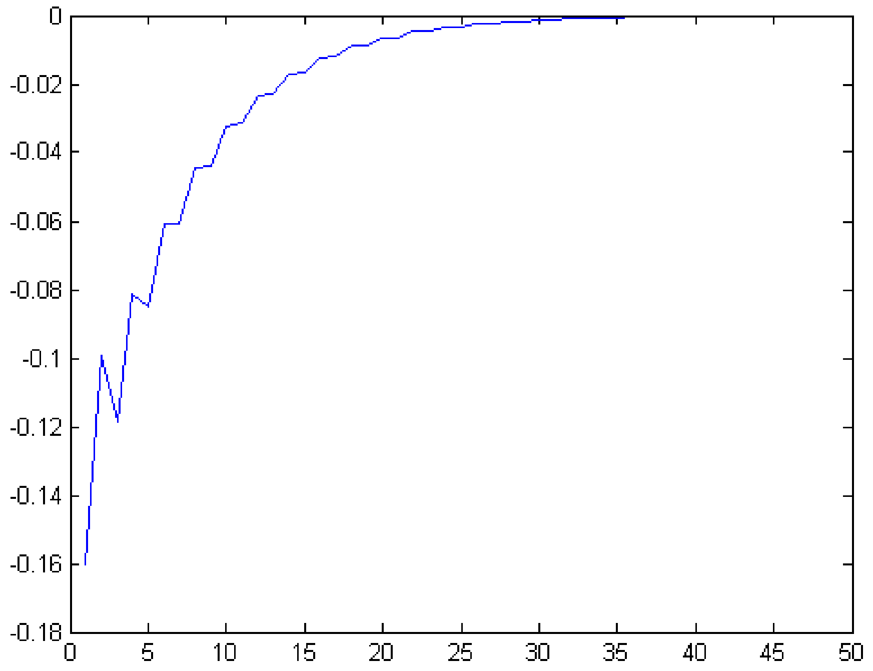

Figure 5.5: Nonnegativity Check - Far Outside the Bound, Many Negative Values

# Chapter 6. Description of Code Modules

In this chapter, we will give a brief overview of each of the modules used in the two- and three-dimensional codes. We will include what is sent in as input to the module and what comes out as output, as well as a short description of what the module accomplishes.

## 6.1 Two-Dimensional Code

## 6.2 Project

The *Project* module is the main part of the program. Here we initialize all of the variables and the main matrices. After the initial pulse, for each time step, we move, pulse (if there is to be continuous pulsing), and diffuse the chemical. We also refine and coarsen at intervals. After the time steps are finished, we calculate the errors in the matrix and output them.

## 6.3 Pulse

The *Pulse* module distributes the pulse around the point source(s). As input, it takes the chemical matrix, the matrix size, the location matrix, how many point sources there are, and the diffusion coefficient, and the time and space discretizations, hereafter referred to as the three main constants. For each point source, it calculates the analytic solution at time $t = 20$ within the radius $\frac{1.1}{h}$, given by

$$u(x, y, 20) = \frac{1}{4\pi bk(20)} \exp\left( - \frac{\left((x - x_0)^2 + (y - y_0)^2\right)}{h^2} \cdot \frac{1}{4bk(20)} \right), \tag{6.1}$$

and adds it to the points in the chemical matrix. As output, it returns the updated chemical matrix.

## 6.4 Move

As input, the *Move* module takes the chemical matrix, the time $t$, the location matrix, the size of the chemical matrix, and the number of point source(s). It uses $t$ to generate some numbers used in the directions the point sources move. As output, the updated location matrix is returned.

## 6.5 Diffuse

*Diffuse* takes in the chemical matrix, the pointer matrix, the chemical matrix size, and the three main constants. It utilizes the ADI method and the Thomas algorithm, running along the columns of $v^n$ to create $\tilde{v}^{n+\frac{1}{2}}$, then along the rows of $\tilde{v}^{n+\frac{1}{2}}$ to find $v^{n+1}$. It returns the updated chemical matrix.

## 6.6 Refine

*Refine* is where we comb through the matrix to check to see if we need to add points. We take in the chemical and pointer matrices, the size of the chemical matrix and the spatial discretization. Looking at the derivative across consecutive points in the matrix that are close together, we refine when the value is above the tolerance. As output, we return the updated chemical and pointer matrices, as well as two variables used in verifying whether there was a need for refinement or not, and then how many new points were created.

## 6.7 Coarsen

In *Coarsen*, we again take in the chemical and pointer matrices and the size of the chemical matrix. With them, we check to see if any of the refined points are no longer necessary. As output, we return the updated chemical and pointer matrices, as well as two variables used in verifying whether there was a need for coarsening or not, and then how many refined points were removed.

## 6.8 Tridiagonal Matrix Solver

We utilize the Thomas algorithm to solver the systems of equations generated by the ADI method. As input, the module takes in the four arrays created in the *Diffuse* module and solves the system of equations. It returns an array that will become the new row or column in the next half-time step.

## 6.9 Isolated Single Point

Here we take in the chemical and pointer matrices, the location of the isolated point, the spacial discretization, the diffusion coefficients for the $x$ and $y$ directions, and whether we are moving along a row or column. We calculate the new half-time step value at the isolated point using shadow points. This value is returned.

## 6.10 Isolated Pair of Points

Here we also take in the chemical and pointer matrices, the location of the pair of points, the spacial discretization, the diffusion coefficients for the $x$ and $y$ directions, and whether we are moving along a row or column. We calculate the new half-time step value for the isolated pair using shadow points. The new values of the two points are returned as output.

## 6.11 Three-Dimensional Code

## 6.12 Project

The *Project* module is the main part of the program. Here we initialize all of the variables and the main matrices; because of the added dimension, there is significantly more to initialize. After the initial pulse, for each time step, we move, pulse (if there is to be continuous pulsing), and diffuse the chemical. We also refine and coarsen at intervals. After the time steps are finished, we calculate the errors in the matrix and output them.

## 6.13 Pulse

The *Pulse* module distributes the pulse around the point source(s). As input, it takes the chemical matrix, the matrix size, the location matrix, how many point sources there are, and the three main constants. For each point source, it calculates the actual solution within the radius $\dfrac{40}{h}$ at time

$t = 100$, given by

$$u(x, y, z, 100) = \frac{1}{(4\pi bk(100))^{\frac{3}{2}}} \exp\left( - \frac{\left((x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2\right)}{h^2} \cdot \frac{1}{4bk(100)} \right), \tag{6.2}$$

and adds it to the points in the chemical matrix. As output, it returns the updated chemical matrix.

## 6.14 Move

The *Move* module is similar to the two-dimensional version in its input and output; the only change is the added random variable for movement in the $z$-direction.

## 6.15 Diffuse

*Diffuse* takes in the chemical matrix, the pointer matrix, the chemical matrix size, and the three main constants. The parameter $\epsilon$ is initialized here for use in the module. We utilize the discretized Dai scheme and the Thomas algorithm, first moving through $v^n$ in the $x$-direction where we use the $FirstKnown$ module to generate $\tilde{v}^{n+\frac{1}{3}}$. Then we move in the $y$-direction to create $\tilde{v}^{n+\frac{2}{3}}$, and finally in the $z$-direction to find $v^{n+1}$. It returns the updated chemical matrix.

## 6.16 Refine

*Refine* for three dimensions is also similar to its two-dimensional counterpart; we now check if refinement is necessary in the $z$-direction.

## 6.17 Coarsen

In the three-dimensional version of *Coarsen*, we now verify the derivatives are below tolerance in all six directions.

## 6.18 Tridiagonal Matrix Solver

This module did not change.

## 6.19 Isolated Single Point

Here we take in two chemical matrices since we need the matrix with which we are updating; the pointer matrix; the location of the isolated point; the spacial discretization; two coefficients for the scheme, $rc = \frac{bk}{2}(1+\epsilon)$ and $eps = \frac{(bk)^3}{8}(6\epsilon + 2\epsilon^3)$; the diffusion coefficients for the three directions; and which direction we are moving. We calculate the new third-time step value at the isolated point using shadow points. This value is returned.

## 6.20 Isolated Pair of Points

Here we also take in two chemical matrices since we need the matrix with which we are updating; the pointer matrix; the location of the first of the pair of points; the spacial discretization; the diffusion coefficients for the $x$, $y$, and $z$ directions; two coefficients for the scheme, $rc = \frac{bk}{2}(1+\epsilon)$ and $eps = \frac{(bk)^3}{8}(6\epsilon + 2\epsilon^3)$; and which direction we are moving. We calculate the new third-time step value for the isolated pair using shadow points. The new values of the two points are returned as output.

## 6.21 First Known Value

*FirstKnown* is where we calculate the $(A_1A_2A_3)v_{xyz}$ term. As input, it receives the chemical matrix currently in use, the pointer matrix, the center point to use, the diffusion coefficient for each direction, and the two coefficients for the scheme, $rc = \frac{bk}{2}(1+\epsilon)$ and $eps = \frac{(bk)^3}{8}(6\epsilon + 2\epsilon^3)$. It utilizes the center point and the twenty-six points around it to calculate the desired value, which is returned as output.

# Appendices

This appendix contains the actual code used in the two parts of this thesis. The two-dimensional code is listed first, followed by the three-dimensional code.

# Appendix A. Two-dimensional Code

We have separated all of the code into sections, one for each major part of the code.

## A.1 Project

```
% In the making of this mesh, it is most useful for me to think of the mesh
% on a  Cartesian set of axes x and y. In that light, the first entry in the
% arrays is  the x-coordinate, and the second is y.
clf; close all
p = 1;                          % Number of point sources
d = 3*10^(-3);                  % Diffusion coefficient
k = .0001;                      % How time is discretized
Loc = zeros(p, 2);              % Holds point source location(s)
error = zeros(4,1);             % Holds the total error using 'average area'
hi = error;                     % Holds the total error using refined points
m = [6401 3201 1601 801];
h = [.00125 .0025 .005 .01];
for z=1:4
    refined = 0;               % Keeps count of the number of refined points
    coarsened = 0;             % Keeps count of the number of coarsened points
    v = zeros(m(z),m(z));      % This will hold the concentration of chemical
    act = v;                   % This will hold the actual solutions
    Chriserror = v;            % This will hold errors at each point
    PT = zeros(m(z), m(z), 6); % Holds the number order of cells and
                               % distances between
% Set up pointers
for x = 1:m(z)
    for y = 1:m(z) % Set the fixed grid points' pointer, x and y
```

```
        if mod(x,4)==1 && mod(y,4)==1
            PT(x,y,1) = x+4;  % What the next node is in the x-direction
            PT(x,y,2) = h(z); % The distance to the next node in
                              % the x-direction
            PT(x,y,3) = x-4;  % Which node points to it in the x-direction
            PT(x,y,4) = y+4;  % What the next node is in the y-direction
            PT(x,y,5) = h(z); % The distance to the next node in
                              % the y-direction
            PT(x,y,6) = y-4;  % Which node points to it in the y-direction
        end
    end
    % Set the boundary points' pointers
    if mod(x,4) ~= 1
        % Set left/bottom boundary pointers
        PT(1,x,1)=m(z);
        PT(1,x,2)=1;
        PT(1,x,3)=-3;
        PT(x,1,4)=m(z);
        PT(x,1,5)=1;
        PT(x,1,6)=-3;
        % Set right/top boundary pointers
        PT(m(z),x,3)=1;
        PT(m(z),x,2)=h(z);
        PT(m(z),x,1)=m(z)+4;
        PT(x,m(z),6)=1;
        PT(x,m(z),5)=h(z);
        PT(x,m(z),4)=m(z)+4;
    end
end
Loc(1,1) = (m(z)+1)/2;
Loc(1,2) = (m(z)+1)/2;
[v] = Adaptive_Pulse_2_Dex(v, m(z), Loc, p, d, k, h(z));
cref = 0;
```

```
ccoa = 0;
checkcoar = 0;
for t = 1:20
    [Loc] = Adaptive_Move_2_D(v, t, Loc, m(z), p);
    [v] = Adaptive_Diffuse_2_D(v, PT, k, h(z), m(z), d);
        [v, PT, checkref, countr] = Adaptive_Refine_2_D(v, PT, m(z), h(z));
        refined = refined + countr;
        countc = 0;
        if mod(t,10) == 0
            [v, PT, checkcoar, countc] = Adaptive_Coarsen_2_D(v, PT, m(z));
        end % Coarsen
        coarsened = coarsened + countc;
        if checkref
            cref = 1;
        end
        if checkcoar
            ccoa = 1;
        end
end
% Calculate actual solution
num=4*d*k*(t+20);
% 'ptct' keeps track of how many points are used to find the average
ptct = 0;
% 'actsum' keeps track of the sum of the actual points
actsum = 0;
% 'vsum' keeps track of the sum of the points in 'v'
vsum = 0;
voldsum = 0;
ct = 0;
for x=5:4:m(z)-4
    ptr = PT(1,x,1);
    while ptr < m(z)
        act(ptr,x)=1/(num*pi)*exp(-(h(z)/4)^2*(((m(z)+1)/2-x)^2+
                ((m(z)+1)/2-ptr)^2)/num);
        % Find average horizontal distance to next point, h as default
        if ptr - PT(ptr,x,3) < 5
            hd1 = PT(PT(ptr,x,3),x,2);
        else hd1 = h(z);
        end
        if PT(ptr,x,1) - ptr < 5
            hd2 = PT(ptr,x,2);
        else hd2 = h(z);
        end
        hd=(hd1+hd2)/2;
        % Find average vertical distance to next point, h as default
        if ptr - PT(ptr,x,6) < 5
            vd1 = PT(ptr,PT(ptr,x,6),5);
        else vd1 = h(z);
        end
        if PT(ptr,x,4) - ptr < 5
            vd2 = PT(ptr,x,5);
        else vd2 = h(z);
        end
        vd=(vd1+vd2)/2;
        % Keep track of the sum of the actual differences at each point
        ct = ct + abs(act(ptr,x)-v(ptr,x));
        % Count the chemical concentration at each point
        actsum = actsum + act(ptr,x);
        vsum = vsum + v(ptr,x);
        % Find the error using the actual discretization
        Chriserror(y,x)=abs(h(z)^2*v(y,x)-h(1)^2*act(y,x));
        % Used in the error using an 'average area'
        act(ptr,x)=act(ptr,x)*hd*vd;
        v(ptr,x) = v(ptr,x)*hd*vd;
        % Advance the pointer
        ptr=PT(ptr,x,1);
```

```
    end % While
end
format long
% Sum the chemical concentration for the actual and calculated solutions
sumact = sum(sum(act))
sumv = sum(sum(tempv))
% Treat all points as the most refined points
intv = vsum * (h(z)/4)^2
intave = actsum * (h(z))^2;
% Calculate errors
error(z) = sum(sum(Chriserror));
hi(z) = ct*(h(z)/4)^2;
% Display if we refined and coarsened
if cref
    disp('refined');
    refined
end
if ccoa
    disp('coarsened');
    coarsened
end
end % Big loop
% Display the errors
error
hi
```

## A.2 Pulse

```
function [T] = Adaptive_Pulse_2_D(T, m, Loc, p, d, k, h)
% For the pulse function, we distribute the chemical pulse like a tent
% over the grid.
firstcheck = 0;
if isempty(find(T>0,1)) == 1
    firstcheck = 1;
```

```
end
t = 20;
num=4*d*k*t;
r = 1.1/h;
for w = 1:p
    x0 = Loc(w,1);
    y0 = Loc(w,2);
    for x=2:m-1
        for y=2:m-1
            if firstcheck
                if mod(x,4)==1 && mod(y,4)==1
                    if sqrt((x0-x)^2+(y0-y)^2) < r
                        T(x,y) = 1/(num*pi)*exp(-(h/4)^2
                                    *((x0-x)^2+(y0-y)^2)/num);
                    end
                end
            else
                if T(x,y)>0
                    if sqrt((x0-x)^2+(y0-y)^2) < r
                        T(x,y) = T(x,y) + 1/(num*pi)*exp(-(h/4)^2
                                    *((x0-x)^2+(y0-y)^2)/num);
                    end
                end
            end
        end
    end
end
end % Adaptive_Pulse_2_D
```

## A.3 Move

```
function [Loc] = Adaptive_Move_2_D(v, t, Loc, m, p)
% Direction will take the time t and use it to come up with a coordinate
% pair (x, y) with x, y in {-1, 0, 1}. The cell will then move in the
% given direction, e.g., (-1, 1) tells it move up diagonally left.
x = round(cos(t));  % Use t to pick -1, 0, or 1 for the two directions
y = round(sin(x));
% Set the new location(s)
for w = 1:p
    % Set x direction
    numx = randi(2,1);
    if Loc(w,1) + numx*x < 2        % Check for left boundary problems
        Loc(w,1) = 2;
    else
        if Loc(w,1) + numx*x > m-1  % Check for right boundary problems
            Loc(w,1) = m-1;
        else
            Loc(w,1) = Loc(w,1) + numx*x;
        end % Right boundary check
    end % Left boundary check
    % Set y direction
    numy = randi(2,1);
    if Loc(w,2) + numy*y < 2        % Check for lower boundary problems
        Loc(w,2) = 2;
    else
        if Loc(w,2) + numy*y > m-1  % Check for upper boundary problems
            Loc(w,2) = m-1;
        else
            Loc(w,2) = Loc(w,2) + numy*y;
        end % Upper boundary check
    end % Lower boundary check
    % Check to see if we are on a valid point
    if v(Loc(w,1), Loc(w,2)) == 0
        if mod(Loc(w,1),4) ~= 1
            if Loc(w,1) < 8 % Check how far left
                Loc(w,1) = 5;
            else
```

```
                if Loc(w,1) > m-6 % Check how far right
                    Loc(w,1) = m-4;
                else Loc(w,1) = floor(Loc(w,1)/4)*4+1;
                end
            end
        end
        if mod(Loc(w,2),4) ~= 1
            if Loc(w,2) < 8 % Check how far down
                Loc(w,2) = 5;
            else
                if Loc(w,2) > m-6 % Check how far up
                    Loc(w,2) = m-4;
                else Loc(w,2) = floor(Loc(w,2)/4)*4+1;
                end
            end
        end
    end % If on used point loop
end % Cell loop
end % Adaptive_Move_2_D
```

## A.4 Diffuse

```
function [T] = Adaptive_Diffuse_2_D(v, PT, k, h, m, d)
% Pulse will calculate the chemical concentration for each point in the
% matrix, based on the previous time step. The new matrix will inherently
% be different because we'll have sources providing more chemical. It uses
% the difference equations from Peaceman and Rachford to solve for the
% next time step.
% As a function, it takes as input the matrix in the current time step and
% solves for the next. It also takes in the space and time discretization
% lengths, and the size of the matrix.
% As output, it returns the matrix after the next time step.
halfkdx = k*d/2;
halfkdy = k*d/2;
```

```
T = zeros(m,m);              % To store the new values
vHat = T;                    % To store the intermediate values
% Calculate vHat along columns of v
for col=2:m-1
    % Check to see if we need to do this column by looking at it and
    % adjacent columns for nonzero values
    if mod(col,4) == 1
        nzb = find(v(col-4,:)>0); % Find nonzero terms in column behind
        nzc = find(v(col,:)>0);   % Find nonzero terms in column
        nzf = find(v(col+4,:)>0); % Find nonzero terms in column forward
        go = isempty(nzb)==0 | isempty(nzc)==0 | isempty(nzf)==0;
    else
        nzc = find(v(col,:)>0);        % Find nonzero terms in column
        go = isempty(nzc)==0;
    end
    if go   % Check to make sure we need to do this column
        sder = zeros(m,1);  % Store the second derivatives
        % Check to see if we are on an original column
        if mod(col,4) == 1
            % Start where necessary
            ptr=PT(col,1,4);
            % Stop at the end of the column
            max=m;
            start = ptr;
            % Calculate second derivatives
            y=0;
            while ptr < max
                y=y+1;
                forder = (v(PT(col,ptr,1),ptr)-v(col,ptr))
                                    /PT(col,ptr,2);
                backder = (v(col,ptr)-v(PT(col,ptr,3),ptr))
                        /PT(PT(col,ptr,3),ptr,2);
                sder(y) = (forder-backder)/((PT(col,ptr,2)
                        +PT(PT(col,ptr,3),ptr,2))/2);
                ptr = PT(col,ptr,4);
            end % While for second derivatives
            % Store the known values for the TDM Solver
            know = zeros(y,1);
            % Store the coefficient matrices a,b,c for the TDM Solver
            a = know;
            b = a;
            c = b;
            ptr = start;
            y = 0;
            while ptr < max
                y=y+1;
                % Set up distances
                h1 = PT(col,ptr,5);
                h2 = PT(col,PT(col,ptr,6),5);
                h3 = (h1+h2)/2;
                % Set up a, b, c, know
                a(y)=-halfkdx/(h2*h3);
                b(y)=1+halfkdx*(h1+h2)/(h1*h2*h3);
                c(y)=-halfkdx/(h1*h3);
                know(y)=v(col,ptr)+halfkdy*sder(y);
                ptr = PT(col,ptr,4);
            end % While for second derivative
            % Calculate the values to enter into vHat
            temp = Adaptive_TDMsolver_2_D(a,b,c,know);
            ptr = start;
            for y = 1:length(temp)
                vHat(col,ptr) = temp(y);
                ptr = PT(col,ptr,4);
            end % For loop input
        else % We are not on a grid column
            ptr = nzc(1);
```

```
            max = nzc(length(nzc))+1;
            while ptr < max
                % 'Counter' keeps track of how many consecutive points are
                % close enough to use together
                counter = 1;
                % 'On' is a logical variable that tells when we can't use
                % the next point with the previous ones
                on = 1;
                % 'Start' holds where the consecutive points started
                start = ptr;
                while on
                    if PT(col,ptr,4) - ptr < 5 && PT(col,ptr,4) < max
                        counter = counter + 1;
                        ptr = PT(col,ptr,4);
                    else
                        on = 0;
                    end
                end
                if counter < 3
                    if counter == 1
                        [vHat(col,ptr)] =
Adaptive_Single_Point_2_D(v,PT,col,ptr,h,halfkdx,halfkdy,'c');
                        ptr = PT(col,ptr,4);
                    else
                        [vHat(col,start),vHat(col,ptr)] =
Adaptive_Two_Points_2_D(v,PT,col,start,ptr,h,halfkdx,halfkdy,'c');
                        ptr = PT(col,ptr,4);
                    end
                else
                    max1 = ptr+1;
                    ptr = start;
                    % Calculate second derivatives
                    y=0;
```

```
while ptr < max1
    y=y+1;
    forder = (v(PT(col,ptr,1),ptr)-v(col,ptr))/
                        PT(col,ptr,2);
    backder = (v(col,ptr)-v(PT(col,ptr,3),ptr))
                        /PT(PT(col,ptr,3),ptr,2);
    sder(y) = (forder-backder)
    /((PT(col,ptr,2)+PT(PT(col,ptr,3),ptr,2))/2);
    ptr = PT(col,ptr,4);
end % While for second derivatives
% Store the known values for the TDM Solver
know = zeros(y,1);
% Store the coefficient matrices a,b,c for
% the TDM Solver
a = know;
b = a;
c = b;
ptr = start;
y = 0;
while ptr < max1
    y=y+1;
    % Set up distances
    h1 = PT(col,ptr,5);
    h2 = PT(col,PT(col,ptr,6),5);
    h3 = (h1+h2)/2;
    % Set up a, b, c, know
    a(y)=-halfkdx/(h2*h3);
    b(y)=1+halfkdx*(h1+h2)/(h1*h2*h3);
    c(y)=-halfkdx/(h1*h3);
    know(y)=v(col,ptr)+halfkdy*sder(y);
    ptr = PT(col,ptr,4);
end % While for second derivative
% Calculate the values to enter into vHat
```

```
                    temp = Adaptive_TDMsolver_2_D(a,b,c,know);
                    ptr = start;
                    for y = 1:length(temp)
                        vHat(col,ptr) = temp(y);
                        ptr = PT(col,ptr,4);
                    end % For loop input
                end % Counter loop
            end % While loop on ptr, max
        end % Mod = 1
    end % isempty check
end % Columns
% Calculate the new v along rows of vHat
for row=2:m-1
    % Check to see if we need to do this row by looking at it and
    % adjacent rows for nonzero values
    if mod(row,4) == 1
        nzb = find(vHat(:,row-4)>0);      % Find nonzero terms in row below
        nzr = find(vHat(:,row)>0);        % Find nonzero terms in row
        nza = find(vHat(:,row+4)>0);      % Find nonzero terms in row above
        go = isempty(nzb)==0 | isempty(nzr)==0 | isempty(nza)==0;
    else
        nzr = find(vHat(:,row)>0);        % Find nonzero terms in row
        go = isempty(nzr) == 0;
    end
    if go   % Check to make sure we need to do this row
        sder = zeros(m,1);  % Store the second derivatives
        % Check to see if we are on an original column
        if mod(row,4) == 1
            % Start where necessary
            ptr=PT(1,row,1);
            % Stop at the end of the row
            max = m;
            start = ptr;
            % Calculate second derivatives
            y=0;
            while ptr < max
                y=y+1;
                forder = (vHat(ptr,PT(ptr,row,4))-vHat(ptr,row))/
                                        PT(ptr,row,5);
                backder = (vHat(ptr,row)-vHat(ptr,PT(ptr,row,6)))
                                        /PT(ptr,PT(ptr,row,6),5);
                sder(y) = (forder-backder)/
((PT(ptr,row,5)+PT(ptr,PT(ptr,row,6),5))/2);
                ptr = PT(ptr,row,1);
            end % While for second derivatives
            % Store the known values for the TDM Solver
            know = zeros(y,1);
            % Store the coefficient matrices a,b,c for the TDM Solver
            a = know;
            b = a;
            c = b;
            ptr = start;
            y = 0;
            while ptr < max
                y=y+1;
                % Set up distances
                h1 = PT(ptr,row,2);
                h2 = PT(PT(ptr,row,3),row,2);
                h3 = (h1+h2)/2;
                % Set up a, b, c, know
                a(y)=-halfkdy/(h2*h3);
                b(y)=1+halfkdy*(h1+h2)/(h1*h2*h3);
                c(y)=-halfkdy/(h1*h3);
                know(y)=vHat(ptr,row)+halfkdx*sder(y);
                ptr = PT(ptr,row,1);
            end % While for second derivative
            temp = Adaptive_TDMsolver_2_D(a,b,c,know);
            ptr = start;
            for y = 1:length(temp)
                T(ptr,row) = temp(y);
                ptr = PT(ptr,row,1);
            end % For loop input
        else % We are not on a grid row
            % Start at the first nonzero node
            ptr = nzr(1);
            max = nzr(length(nzr))+1;
            while ptr < max
                % 'Counter' keeps track of how many consecutive points are
                % close enough to use together
                counter = 1;
                % 'On' is a logical variable that tells when we can't use
                % the next point with the previous ones
                on = 1;
                % 'Start' holds where the consecutive points started
                start = ptr;
                while on
                    if PT(ptr,row,1) - ptr < 5 && PT(ptr,row,1) < max
                        counter = counter + 1;
                        ptr = PT(ptr,row,1);
                    else
                        on = 0;
                    end
                end
                if counter < 3
                    if counter == 1
                        [T(ptr,row)]=
Adaptive_Single_Point_2_D(vHat,PT,row,ptr,h,halfkdx,halfkdy,'r');
                        ptr = PT(ptr,row,1);
                    else
                    [T(start,row),T(ptr,row)] =
Adaptive_Two_Points_2_D(vHat,PT,row,start,ptr,h,halfkdx,halfkdy,'r');
                    ptr = PT(ptr,row,1);
                end
            else
                max1 = ptr+1;
                ptr = start;
                % Calculate second derivatives
                y=0;
                while ptr < max1
                    y=y+1;
                    forder =
(vHat(ptr,PT(ptr,row,4))-vHat(ptr,row))/PT(ptr,row,5);
                    backder = (vHat(ptr,row)-vHat(ptr,PT(ptr,row,6)))
                            /PT(ptr,PT(ptr,row,6),5);
                    sder(y) = (forder-backder)
/((PT(ptr,row,5)+PT(ptr,PT(ptr,row,6),5))/2);
                    ptr = PT(ptr,row,1);
                end % While for second derivatives
                % Store the known values for the TDM Solver
                know = zeros(y,1);
                % Store the coefficient matrices a,b,c for
                % the TDM Solver
                a = know;
                b = a;
                c = b;
                ptr = start;
                y = 0;
                while ptr < max1
                    y=y+1;
                    % Set up distances
                    h1 = PT(ptr,row,2);
                    h2 = PT(PT(ptr,row,3),row,2);
                    h3 = (h1+h2)/2;
                    % Set up a, b, c, know
                    a(y)=-halfkdy/(h2*h3);
                    b(y)=1+halfkdy*(h1+h2)/(h1*h2*h3);
                    c(y)=-halfkdy/(h1*h3);
                    know(y)=vHat(ptr,row)+halfkdx*sder(y);
                    ptr = PT(ptr,row,1);
                end % While for second derivative
                % Calculate the values to enter into vHat
                temp = Adaptive_TDMsolver_2_D(a,b,c,know);
                ptr = start;
                for y = 1:length(temp)
                    T(ptr,row) = temp(y);
                    ptr = PT(ptr,row,1);
                end % For loop input
            end % Counter loop
        end % While loop on ptr, max
    end % Mod = 1
end % isempty check
end % Rows
end % Adaptive_Diffuse_2_D
```

## A.5 Refine

```
function [T, PT, checker, count] = Adaptive_Refine_2_D(T, PT, m, h)
% We loop through the scheme to find areas that need refinement. We search
% for areas where the secant line through two consecutive points has slope
% higher than some tolerance. If we find one, we add a point between them.
% We find its value by the midpoint of the two values it divides. As we are
% searching using all rows and columns, we do check to make sure the two
% points are within the normal discretization of the grid.
tol = 750;      % Tolerance for the secant slope
% 'Checker' keeps track of whether or not we refined this time through the
% matrix
```

```
checker = 0;
count = 0;
for col = 2:m-1 % Loop along the interior columns of v to check refinement
    fp = find(T(col,:)>0);   % Find nonzero terms (fp for find positive)
    if isempty(fp) == 0      % Check to make sure there is one
        ptr = PT(col,fp(1),6);         % At which node we start the loop
        % Loop on the number of points in use currently for this column
        while ptr < fp(length(fp))+1
            % Check to see if there's a close point above to use
            % for refinement
            above = PT(col,ptr,4)-ptr<5 && PT(col,ptr,4)-ptr>1;
            % Check slope of secant above
            if abs(T(col,PT(col,ptr,4))-T(col,ptr))/PT(col,ptr,5) > tol
                    && above
                checker = 1;
                count = count + 1;
                if mod(PT(col,ptr,4)+ptr,2) == 1
                    if mod(ptr,4) == 2
                        L = ptr + 1;
                    else L = ptr + 2;
                    end
                else
                    L = (PT(col,ptr,4)+ptr)/2;
                end
                % Set the value in the matrix for the refined point
                T(col,L)=T(col,ptr)+(L-ptr)*(T(col,PT(col,ptr,4))-
                        T(col,ptr))/(PT(col,ptr,4)-ptr);
                % Set the vertical values for the pointer matrix
                PT(col,L,4) = PT(col,ptr,4);
                PT(col,L,5) = h/4*(PT(col,L,4)-L);
                PT(col,L,6) = ptr;
                % Insert the new point vertically
                PT(col,ptr,4) = L;
                PT(col,ptr,5) = h/4*(L-ptr);
                PT(col,PT(col,L,4),6) = L;
                % Set all of the horizontal pointers for the new point
                left = PT(1,L,1);
                if left < col % Check for a nonboundary point to the left
                    while PT(left,L,1) < col
                        left = PT(left,L,1);
                    end
                    % Set forward point
                    PT(col,L,1) = PT(left,L,1);
                    % Set new backward point for that point
                    PT(PT(col,L,1),L,3) = col;
                    % Set backwards point
                    PT(col,L,3) = left;
                    % Set new forward point and distance for that point
                    PT(left,L,1) = col;
                    PT(left,L,2) = h/4*(col-left);
                    PT(col,L,2) = h/4*(PT(col,L,1) - col);
                else
                    PT(col,L,1) = left;
                    PT(col,L,2) = h/4*(left-col);
                    PT(col,L,3) = 1;
                    PT(left,L,3) = col;
                    PT(1,L,1) = col;
                    PT(1,L,2) = h/4*(col-1);
                end % There exists a nonboundary point to the left
                % Advance the pointer
                ptr = PT(col,L,4);
            else
                % Advance the pointer
                ptr = PT(col,ptr,4);
            end % If secant slope loop
        end % While ptr loop
        end % If isempty loop
end % y loop
for row = 2:m-1 % Loop along the interior rows of v to check for refinement
    fp = find(T(:,row)>0);    % Find nonzero terms
    if isempty(fp) == 0       % Check to make sure there is one
        ptr = PT(fp(1),row,3);         % At which node we start the loop
        % Loop on the number of points in use currently for this column
        while ptr < fp(length(fp))+1
            % Check to see if there's a point on the right to use for
            % refinement
            right = PT(ptr,row,1)-ptr<5 && PT(ptr,row,1)-ptr>1;
            % Check slope of secant to the right
            if abs(T(PT(ptr,row,1),row)-T(ptr,row))/PT(ptr,row,2) > tol
                    && right
                checker = 1;
                count = count + 1;
                if mod(PT(ptr,row,1)+ptr,2) == 1
                    if mod(ptr,4) == 2
                        L = ptr + 1;
                    else L = ptr + 2;
                    end
                else
                    L = (PT(ptr,row,1)+ptr)/2;
                end
                % Set the value in the matrix for the refined point
                T(L,row) =T(ptr,row)+(L-ptr)*(T(PT(ptr,row,1),row)-
                        T(ptr,row))/(PT(ptr,row,1)-ptr);
                % Set the horizontal values for the pointer matrix
                PT(L,row,1) = PT(ptr,row,1);
                PT(L,row,2) = h/4*(PT(L,row,1)-L);
                PT(L,row,3) = ptr;
                % Insert the new point horizontally
                PT(ptr,row,1) = L;
                PT(ptr,row,2) = h/4*(L-ptr);
                PT(PT(L,row,1),row,3) = L;
                below = PT(L,1,4);
                if below < row % Check for a nonboundary point below
                    while PT(L,below,4)<row
                        below = PT(L,below,4);
                    end
                    % Set point above
                    PT(L,row,4) = PT(L,below,4);
                    % Set new point below for that point
                    PT(L,PT(L,row,4),6) = row;
                    % Set backwards point
                    PT(L,row,6) = below;
                    % Set new forward point and distance for that point
                    PT(L,below,4) = row;
                    PT(L,below,5) = h/4*(row-below);
                    PT(L,row,5) = h/4*(PT(L,row,4) - row);
                else
                    PT(L,row,4) = below;
                    PT(L,row,5) = h/4*(below-row);
                    PT(L,row,6) = 1;
                    PT(L,below,6) = row;
                    PT(L,1,4) = row;
                    PT(L,1,5) = h/4*(row-1);
                end
                % Advance the pointer
                ptr = PT(L,row,1);
            else
                % Advance the pointer
                ptr = PT(ptr,row,1);
            end % If secant slope loop
        end % while ptr loop
end% If isempty loop
```

```
end % y loop

end % Adaptive_Refine_2_D
```

## A.6 Coarsen

```
function [T, PT, checker, ct] = Adaptive_Coarsen_2_D(T, PT, m)

% The coarsen function will be executed every five time steps to see if

% any of the added points can now be removed because they are no longer

% relevant in terms of calculating derivatives. We loop through the added

% points and check to see if the difference between them and the points

% around them fall below a given tolerance level. If so, the points are

% removed, and the pointer array (PT) is modified.

tol = 1;            % Tolerance for removing points

checker = 0;

ct = 0;

% Coarsen over the whole matrix along the columns

for col = 2:m-1   % Check all original interior columns

    nonzero = find(T(col,:)>0);      % Find nonzero terms

    if isempty(nonzero)==0

        ptr = nonzero(1);

        while ptr < m

            if mod(col,4)==1 && mod(ptr,4)==1 % Check if it's a grid point

                ptr = PT(col,ptr,4);

            else % If it's not a grid point

                % Check the derivative above

                if PT(col,ptr,4) - ptr < 5

                    top = abs(T(col,PT(col,ptr,4))-T(col,ptr))/

                            PT(col,ptr,5)<tol;

                else

                    top = 1;

                end

                % Check the derivative below

                if ptr - PT(col,ptr,6) < 5

                    bot = abs(T(col,PT(col,ptr,6))-T(col,ptr))
```

```
                  /PT(col,PT(col,ptr,6),5)<tol;
else
    bot = 1;
end
% Check the derivative to the right
if PT(col,ptr,1) - ptr < 5
    right = abs(T(PT(col,ptr,1),ptr)-T(col,ptr))/
            PT(col,ptr,2)<tol;
else
    right = 1;
end
% Check the derivative to the left
if ptr - PT(col,ptr,3) < 5
    left = abs(T(PT(col,ptr,3),ptr)-T(col,ptr))
                  /PT(PT(col,ptr,3),ptr,2)<tol;
else
    left = 1;
end
if top && bot && left && right % If it's unneeded
    checker = 1;
    ct = ct + 1;
    % Set the point to zero
    T(col,ptr) = 0;
    % Remove the point horizontally
    PT(PT(col,ptr,3),ptr,1) = PT(col,ptr,1);
    PT(PT(col,ptr,3),ptr,2) = PT(PT(col,ptr,3),ptr,2) +
            PT(col,ptr,2);
    PT(PT(col,ptr,1),ptr,3) = PT(col,ptr,3);
    %Remove the point vertically
    PT(col,PT(col,ptr,6),4) = PT(col,ptr,4);
    PT(col,PT(col,ptr,6),5) = PT(col,PT(col,ptr,6),5)+
            PT(col,ptr,5);
    PT(col,PT(col,ptr,4),6) = PT(col,ptr,6);
```

```
                    % Erase the information in the pointer matrix
                    keep = PT(col,ptr,4);
                    PT(col,ptr,:) = 0;
                    ptr = keep;
                else % If it's still needed
                    ptr = PT(col,ptr,4);
                end % If unneeded
            end % If grid point
        end
    end
end % Loop on columns
end % Adaptive_Coarsen_2_D
```

## A.7 Thomas Algorithm

```
function x = Adaptive_TDMsolver_2_D(a,b,c,d)
% a, b, c are the column vectors for the compressed tridiagonal matrix,
% d is the right vector
n = length(d);       % n is the number of elements in the array
x = zeros(n,1);
% Modify the first-row coefficients
c(1) = c(1)/ b(1);    % Division by zero risk.
d(1) = d(1) / b(1);
for i = 2:n-1
    temp = b(i) - a(i) * c(i-1);
    c(i) = c(i) / temp;
    d(i) = (d(i) - a(i) * d(i-1))/temp;
end % For loop adjusting coefficients
d(n) = (d(n) - a(n) * d(n-1))/(b(n) - a(n) * c(n-1));
% Now back substitute.
x(n) = d(n);
for i = n-1:-1:1
    x(i) = d(i) - c(i) * x(i+1);
end % For loop back substitution
end % function ADI_TDMsolver_2_D
```

## A.8 Isolated Single Point

```
function [ans] = Adaptive_Single_Point_2_D(T,PT,rowcol,ptr,h,hkdx,hkdy,rc)
% We use this function to calculate the new value at a point when that
% point is isolated on a refined row or column. We cannot use the Thomas
% algorithm due to lack of support. The input matrix may be 'v' or 'vHat',
% so we'll use 'T' to represent the matrix temporarily in use. We also use
% the pointer matrix, a row or column (depending on the direction we were
% moving in 'Diffuse', a pointer, the current discretization 'h', and some
% way to tell which way we were moving.
if rc == 'c' % If we were diffusing up a column
    % Set the columns before and after the current one
    before = floor((rowcol-1)/4)*4 + 1;
    after = floor((rowcol+2)/4)*4 + 1;
    % Set the shadow rows above and below
    if mod(ptr,4) == 1
        below = ptr - 4;
        above = ptr + 4;
    else
        below = floor((ptr-1)/4)*4 + 1;
        above = floor((ptr+2)/4)*4 + 1;
    end
    % Calculate the second derivative across the current point
    rdist = h/4*(PT(rowcol,ptr,1) - rowcol);
    rder = (T(PT(rowcol,ptr,1),ptr)-T(rowcol,ptr))/rdist;
    ldist = h/4*(rowcol - PT(rowcol,ptr,3));
    lder = (T(rowcol,ptr)-T(PT(rowcol,ptr,3),ptr))/ldist;
    sd = (rder-lder)/((rdist+ldist)/2);
    % Set the shadow point above and its distance from the point
    temab = (T(before,above)+T(after,above))/2;
    d1 = h/4*(above-ptr);
    % Set the shadow point below and its distance from the point
```

```
        tembel = (T(before,below)+T(after,below))/2;
        d2 = h/4*(ptr-below);
        d3 = (d1+d2)/2;
        % Set the coefficients for the calculation
        a = -hkdx/(d2*d3);
        b = 1+hkdx*(d1+d2)/(d1*d2*d3);
        c = -hkdx/(d1*d3);
        ans = (T(rowcol,ptr)+hkdy*sd - c*temab - a*tembel)/b;
    else % We were diffusing along a row
        % Set the shadow rows above and below
        below = floor((rowcol-1)/4)*4 + 1;
        above = floor((rowcol+2)/4)*4 + 1;
        % Set the columns before and after the current one
        if mod(ptr,4) == 1
            before = ptr - 4;
            after = ptr + 4;
        else
            before = floor((ptr-1)/4)*4 + 1;
            after = floor((ptr+2)/4)*4 + 1;
        end
        % Calculate the second derivative across the current point
        udist = h/4*(PT(ptr,rowcol,4) - rowcol);
        uder = (T(ptr,PT(ptr,rowcol,4))-T(ptr,rowcol))/udist;
        ddist = h/4*(rowcol - PT(ptr,rowcol,6));
        dder = (T(ptr,rowcol)-T(ptr,PT(ptr,rowcol,6)))/ddist;
        sd = (uder-dder)/((udist+ddist)/2);
        % Set the shadow point after and its distance from the point
        temaf = (T(after,below)+T(after,above))/2;
        d1 = h/4*(after-ptr);
        % Set the shadow point before and its distance from the point
        tembef = (T(before,below)+T(before,above))/2;
        d2 = h/4*(ptr-before);
        d3 = (d1+d2)/2;
        % Set the coefficients for the calculation
        a = -hkdy/(d2*d3);
        b = 1+hkdy*(d1+d2)/(d1*d2*d3);
        c = -hkdy/(d1*d3);
        ans = (T(ptr,rowcol)+hkdx*sd - c*temaf - a*tembef)/b;
    end
    end % function Adaptive_Single_Point_2_D
```

## A.9 Isolated Pair of Points

```
function [ans1, ans2] = Adaptive_Two_Points_2_D
                                (T, PT, rowcol, n1, n2, h, hkdx, hkdy, rc)
% This function calculates new half-points for two isolated points on a row
% or column. We take an input matrix, be it 'v' or 'vHat', the pointer
% matrix, a row or column, the two isolated points on that row or column,
% the discretization, and a character input to tell whether we are on a row
% or column.
if rc == 'c' % We are on a column
    % Set the columns before and after the current one
    before = floor((rowcol-1)/4)*4 + 1;
    after = floor((rowcol+2)/4)*4 + 1;
    x = T(rowcol,n1);
    y = T(rowcol,n2);
    % Set the shadow rows above and below
    if mod(n1,4) == 1
        below = n1 - 4;
    else
        below = floor((n1-1)/4)*4 + 1;
    end
    if mod(n2,4) == 1
        above = n2 + 4;
    else
        above = floor((n2+2)/4)*4 + 1;
    end
```

```
% Calculate the two second derivatives
rdist1 = h/4*(PT(rowcol,n1,1) - rowcol);
rder1 = (T(PT(rowcol,n1,1),n1) - x)/rdist1;
ldist1 = h/4*(rowcol - PT(rowcol,n1,3));
lder1 = (x - T(PT(rowcol,n1,3),n1))/ldist1;
sd1 = (rder1-lder1)/((rdist1+ldist1)/2);
rdist2 = h/4*(PT(rowcol,n2,1) - rowcol);
rder2 = (T(PT(rowcol,n2,1),n2) - y)/rdist2;
ldist2 = h/4*(rowcol - PT(rowcol,n2,3));
lder2 = (y - T(PT(rowcol,n2,3),n2))/ldist2;
sd2 = (rder2-lder2)/((rdist2+ldist2)/2);
% Set the shadow point above and its distance from the point
temab = (T(before,above)+T(after,above))/2;
% We'll use t for the shadow point on top
t1 = h/4*(above-n2);
t2 = PT(rowcol,n1,5);
t3 = (t1+t2)/2;
% Set the shadow point below and its distance from the point
tembel = (T(before,below)+T(after,below))/2;
% We use d for the shadow point down below
d1 = t2;
d2 = h/4*(n1-below);
d3 = (d1+d2)/2;
% Set the coefficients for the calculation
% We use a 1 for the coefficients that would be in
% the first row, 2 for the second
a1 = -hkdx/(d2*d3);
b1 = 1+hkdx*(d1+d2)/(d1*d2*d3);
c1 = -hkdx/(d1*d3);
a2 = -hkdx/(t2*t3);
b2 = 1+hkdx*(t1+t2)/(t1*t2*t3);
c2 = -hkdx/(t1*t3);
ans2 = (a1*a2*tembel-a2*x-a2*hkdy*sd1+b1*y+b1*hkdy*sd2-b1*c2*temab);
```

```
    ans2 = ans2/(b1*b2-a2*c1);
    ans1 = (x - c1*ans2+hkdy*sd1-a1*tembel)/b1;
else % We are using a row
    % Set the rows below and above the current one
    below = floor((rowcol-1)/4)*4 + 1;
    above = floor((rowcol+2)/4)*4 + 1;
    x = T(n1,rowcol);
    y = T(n2,rowcol);
    % Set the shadow columns before and after
    if mod(n1,4) == 1
        before = n1 - 4;
    else
        before = floor((n1-1)/4)*4 + 1;
    end
    if mod(n2,4) == 1
        after = n2 + 4;
    else
        after = floor((n2+2)/4)*4 + 1;
    end
    % Calculate the two second derivatives
    udist1 = h/4*(PT(n1,rowcol,4) - rowcol);
    uder1 = (T(n1,PT(n1,rowcol,4)) - x)/udist1;
    ddist1 = h/4*(rowcol - PT(n1,rowcol,6));
    dder1 = (x - T(n1,PT(n1,rowcol,6)))/ddist1;
    sd1 = (uder1-dder1)/((udist1+ddist1)/2);
    udist2 = h/4*(PT(n2,rowcol,4) - rowcol);
    uder2 = (T(n2,PT(n2,rowcol,4)) - y)/udist2;
    ddist2 = h/4*(rowcol - PT(n2,rowcol,6));
    dder2 = (y - T(n2,PT(n2,rowcol,6)))/ddist2;
    sd2 = (uder2-dder2)/((udist2+ddist2)/2);
    % Set the shadow point after and its distance from the point
    temaf = (T(after,below)+T(after,above))/2;
    % We'll use t for the shadow point on top
    t1 = h/4*(after-n2);
    t2 = PT(n1,rowcol,2);
    t3 = (t1+t2)/2;
    % Set the shadow point before and its distance from the point
    tembef = (T(before,below)+T(before,above))/2;
    % We use d for the shadow point down below
    d1 = t2;
    d2 = h/4*(n1-before);
    d3 = (d1+d2)/2;
    % Set the coefficients for the calculation
    % We use a 1 for the coefficients that would be in
    % the first row of the LHS matrix, 2 for the second row
    a1 = -hkdy/(d2*d3);
    b1 = 1+hkdy*(d1+d2)/(d1*d2*d3);
    c1 = -hkdy/(d1*d3);
    a2 = -hkdy/(t2*t3);
    b2 = 1+hkdy*(t1+t2)/(t1*t2*t3);
    c2 = -hkdy/(t1*t3);
    ans2 = (a1*a2*tembef-a2*x-a2*hkdx*sd1+b1*y+b1*hkdx*sd2-b1*c2*temaf);
    ans2 = ans2/(b1*b2-a2*c1);
    ans1 = (x - c1*ans2+hkdx*sd1-a1*tembef)/b1;
end % function Adaptive_Two_Points_2_D
```

# Appendix B. Three-Dimensional Code

The three-dimensional code is also separated into sections, one for each major part of the program.

## B.1 Project

```
% In the making of this mesh, it is most useful for me to think of the mesh
% on a Cartesian set of axes in x, y, and z. In that light, the first entry
% in the arrays is the x-coordinate, the second is y, and the third is z.
```

```
clf; close all
p = 1;                          % Number of point sources
d = 3*10^(-3);
k = .0001;                      % How time is discretized
Loc = zeros(p, 3);              % Holds point source location(s)
error = zeros(4,1);
m = [401 201 101 53];
h = [.00125 .0025 .005 .0096153846153846177];
for a=1:4
    refined = 0;
    coarsened = 0;
    v = zeros(m(a),m(a),m(a)); % This will hold the concentration of chemical
    tempv = v;
    Chriserror = v;
    act = v;
    PT = zeros(m(a),m(a),m(a),9); % Holds the number order of cells
                                  % and distances between
% Set up pointers
for x = 1:m(a)
    for y = 1:m(a) % Set the fixed grid points' pointer, x, y, and z
        for z = 1:m(a)
            if mod(x,4)==1 && mod(y,4)==1 && mod(z,4)==1
                PT(x,y,z,1) = x+4;  % What the next node is in
                                    % the x-direction
                PT(x,y,z,2) = h(a); % The distance to the next node
                                    % in the x-direction
                PT(x,y,z,3) = x-4;  % Which node points to it in
                                    % the x-direction
                PT(x,y,z,4) = y+4;  % What the next node is in
                                    % the y-direction
                PT(x,y,z,5) = h(a); % The distance to the next node
                                    % in the y-direction
                PT(x,y,z,6) = y-4;  % Which node points to it in
```

```
                                % the y-direction
            PT(x,y,z,7) = z+4;  % What the next node is in
                                % the z-direction
            PT(x,y,z,8) = h(a); % The distance to the next node
                                % in the z-direction
            PT(x,y,z,9) = z-4;  % Which node points to it in
                                % the z-direction
        end
    end
    % Set nonboundary, nongrid point pointers
    if x>1 && (mod(x,4) ~= 1 || mod(y,4) ~= 1)
        % In x-y plane
        PT(x,y,1,1) = x+1;
        PT(x,y,1,2) = h(a)/4;
        PT(x,y,1,3) = x-1;
        PT(x,y,1,4) = y+1;
        PT(x,y,1,5) = h(a)/4;
        PT(x,y,1,6) = y-1;
        PT(x,y,1,7) = m(a);
        PT(x,y,1,8) = h(a)/4*(m(a)-1);
        PT(x,y,1,9) = -3;
        PT(x,y,m(a),1) = x+1;
        PT(x,y,m(a),2) = h(a)/4;
        PT(x,y,m(a),3) = x-1;
        PT(x,y,m(a),4) = y+1;
        PT(x,y,m(a),5) = h(a)/4;
        PT(x,y,m(a),6) = y-1;
        PT(x,y,m(a),7) = m(a)+4;
        PT(x,y,m(a),8) = h(a);
        PT(x,y,m(a),9) = 1;
        % In x-z plane
        PT(x,1,y,1) = x+1;
        PT(x,1,y,2) = h(a)/4;
```

```
PT(x,1,y,3) = x-1;
PT(x,1,y,4) = m(a);
PT(x,1,y,5) = h(a)/4*(m(a)-1);
PT(x,1,y,6) = -3;
PT(x,1,y,7) = y+1;
PT(x,1,y,8) = h(a)/4;
PT(x,1,y,9) = y-1;
PT(x,m(a),y,1) = x+1;
PT(x,m(a),y,2) = h(a)/4;
PT(x,m(a),y,3) = x-1;
PT(x,m(a),y,4) = m(a)+4;
PT(x,m(a),y,5) = h(a);
PT(x,m(a),y,6) = 1;
PT(x,m(a),y,7) = y+1;
PT(x,m(a),y,8) = h(a)/4;
PT(x,m(a),y,9) = y-1;
% In y-z plane
PT(1,x,y,1) = m(a);
PT(1,x,y,2) = h(a)/4*(m(a)-1);
PT(1,x,y,3) = -3;
PT(1,x,y,4) = x+1;
PT(1,x,y,5) = h(a)/4;
PT(1,x,y,6) = x-1;
PT(1,x,y,7) = y+1;
PT(1,x,y,8) = h(a)/4;
PT(1,x,y,9) = y-1;
PT(m(a),x,y,1) = m(a)+4;
PT(m(a),x,y,2) = h(a);
PT(m(a),x,y,3) = 1;
PT(m(a),x,y,4) = x+1;
PT(m(a),x,y,5) = h(a)/4;
PT(m(a),x,y,6) = x-1;
PT(m(a),x,y,7) = y+1;
```

```
        PT(m(a),x,y,8) = h(a)/4;
        PT(m(a),x,y,9) = y-1;
    end
end
% Set the boundary points' pointers
if mod(x,4) ~= 1
    % Along x-axis
    PT(x,1,1,4)=m(a);
    PT(x,1,1,5)=1;
    PT(x,1,1,6)=-3;
    PT(x,1,1,7)=m(a);
    PT(x,1,1,8)=1;
    PT(x,1,1,9)=-3;
    % Along y-axis
    PT(1,x,1,1)=m(a);
    PT(1,x,1,2)=1;
    PT(1,x,1,3)=-3;
    PT(1,x,1,7)=m(a);
    PT(1,x,1,8)=1;
    PT(1,x,1,9)=-3;
    % Along z-axis
    PT(1,1,x,1)=m(a);
    PT(1,1,x,2)=1;
    PT(1,1,x,3)=-3;
    PT(1,1,x,4)=m(a);
    PT(1,1,x,5)=1;
    PT(1,1,x,6)=-3;
    % Parallel to x-axis, y=1, z=max
    PT(x,1,m(a),4)=m(a);
    PT(x,1,m(a),5)=1;
    PT(x,1,m(a),6)=-3;
    PT(x,1,m(a),7)=m(a)+4;
    PT(x,1,m(a),8)=h(a);
PT(x,1,m(a),9)=1;
% Parallel to x-axis, y=max, z=1
PT(x,m(a),1,4)=m(a)+4;
PT(x,m(a),1,5)=h(a);
PT(x,m(a),1,6)=1;
PT(x,m(a),1,7)=m(a);
PT(x,m(a),1,8)=1;
PT(x,m(a),1,9)=-3;
% Parallel to x-axis, y=max, z=max
PT(x,m(a),m(a),4)=m(a)+4;
PT(x,m(a),m(a),5)=h(a);
PT(x,m(a),m(a),6)=1;
PT(x,m(a),m(a),7)=m(a)+4;
PT(x,m(a),m(a),8)=h(a);
PT(x,m(a),m(a),9)=1;
% Parallel to y-axis, x=1, z=max
PT(1,x,m(a),1)=m(a);
PT(1,x,m(a),2)=1;
PT(1,x,m(a),3)=-3;
PT(1,x,m(a),7)=m(a)+4;
PT(1,x,m(a),8)=h(a);
PT(1,x,m(a),9)=1;
% Parallel to y-axis, x=max, z=1
PT(m(a),x,1,1)=m(a)+4;
PT(m(a),x,1,2)=h(a);
PT(m(a),x,1,3)=1;
PT(m(a),x,1,7)=m(a);
PT(m(a),x,1,8)=1;
PT(m(a),x,1,9)=-3;
% Parallel to y-axis, x=max, z=max
PT(m(a),x,m(a),1)=m(a)+4;
PT(m(a),x,m(a),2)=h(a);
PT(m(a),x,m(a),3)=1;
```

```
            PT(m(a),x,m(a),7)=m(a)+4;
            PT(m(a),x,m(a),8)=h(a);
            PT(m(a),x,m(a),9)=1;
            % Parallel to z-axis, x=1, y=max
            PT(1,m(a),x,1)=m(a);
            PT(1,m(a),x,2)=1;
            PT(1,m(a),x,3)=-3;
            PT(1,m(a),x,4)=m(a)+4;
            PT(1,m(a),x,5)=h(a);
            PT(1,m(a),x,6)=1;
            % Parallel to z-axis, x=max, y=1
            PT(m(a),1,x,1)=m(a)+4;
            PT(m(a),1,x,2)=h(a);
            PT(m(a),1,x,3)=1;
            PT(m(a),1,x,4)=m(a);
            PT(m(a),1,x,5)=1;
            PT(m(a),1,x,6)=-3;
            % Parallel to z-axis, x=max, y=max
            PT(m(a),m(a),x,1)=m(a)+4;
            PT(m(a),m(a),x,2)=h(a);
            PT(m(a),m(a),x,3)=1;
            PT(m(a),m(a),x,4)=m(a)+4;
            PT(m(a),m(a),x,5)=h(a);
            PT(m(a),m(a),x,6)=1;
        end
    end
    Loc(1,1) = (m(a)+1)/2-2;
    Loc(1,2) = (m(a)+1)/2-2;
    Loc(1,3) = (m(a)+1)/2-2;
    [v] = ADI_Adap_3_D_Pulse(v, m(a), Loc, p, d, k, h(a));
    cref = 0;
    ccoa = 0;
    checkcoar = 0;
```

```
first = 0;
for t = 1:24
    [Loc] = ADI_Adap_3_D_Move(v, t, Loc, m(a), p);
    [v] = ADI_Adap_3_D_Diffuse(v, PT, k, h(a), m(a), d);
    [v, PT, checkref, countr] = ADI_Adap_3_D_Refine(v, PT, m(a), h(a));
    refined = refined + countr;
    countc = 0;
    if mod(t,10) == 0
        [v, PT, checkcoar, countc] = ADI_Adap_3_D_Coarsen(v, PT, m(a));
    end % Coarsen
    coarsened = coarsened + countc;
    if checkref
        cref = 1;
    end
    if checkcoar
        ccoa = 1;
    end
end
% Calculate actual solution
num2=4*d*k*(t+100);
num1=(num2)^(-3/2);
pinum=pi^(-3/2);
x0 = (m(a)+1)/2-2;
y0 = (m(a)+1)/2-2;
z0 = (m(a)+1)/2-2;
% 'ptct' keeps track of how many points are used to find the average
ptct = 0;
% 'actsum' keeps track of the sum of the actual points
actsum = 0;
% 'vsum' keeps track of the sum of the points in 'v'
vsum = 0;
ct = 0;
for x=5:4:m(a)-4
    for y=5:4:m(a)-4
        ptr=PT(x,y,1,7);
        while ptr < m(a)
            act(x,y,ptr)=(num1*pinum)*exp(-(h(a)/4)^2*
                                  ((x0-x)^2+(y0-y)^2+(z0-ptr)^2)/num2);
            % Find average distance to next point in the x-direction,
            % h as default
            if PT(x,y,ptr,1) - x < 5
                h1 = PT(x,y,ptr,2);
            else h1 = h(a);
            end
            if x - PT(x,y,ptr,3) < 5
                h2 = PT(PT(x,y,ptr,3),y,ptr,2);
            else h2 = h(a);
            end
            h3=(h1+h2)/2;
            % Find average distance to next point in the x-direction,
            % h as default
            if PT(x,y,ptr,4) - y < 5
                d1 = PT(x,y,ptr,5);
            else d1 = h(a);
            end
            if y - PT(x,y,ptr,6) < 5
                d2 = PT(x,PT(x,y,ptr,6),ptr,5);
            else d2 = h(a);
            end
            d3=(d1+d2)/2;
            % Find average distance to next point in the z-direction,
            % h as default
            if PT(x,y,ptr,7) - ptr < 5
                m1 = PT(x,y,ptr,8);
            else m1 = h(a);
            end
```

```
            if ptr - PT(x,y,ptr,9) < 5
                m2 = PT(x,y,PT(x,y,ptr,9),8);
            else m2 = h(a);
            end
            m3=(m1+m2)/2;
            ptct = ptct + 1;
            ct = ct + abs(act(x,y,ptr)-v(x,y,ptr));
            actsum = actsum + act(x,y,ptr);
            vsum = vsum + v(x,y,ptr);
            tempv(x,y,ptr) = v(x,y,ptr)*h3*d3*m3;
            act(x,y,ptr)=act(x,y,ptr)*h3*d3*m3;
            Chriserror(x,y,ptr)=abs(tempv(x,y,ptr)-act(x,y,ptr));
            ptr=PT(x,y,ptr,7);
        end % While
    end
end
format long
sumact = sum(sum(sum(act)))
sumv = sum(sum(sum(tempv)))
vave = vsum * h(a)^3
actave = actsum * h(a)^3
error(a) = h(a)^3*(sum(sum(sum(Chriserror))));
hi = h(a)^3*ct
if cref
    disp('refined');
    refined
end
if ccoa
    disp('coarsened');
    coarsened
end
end % Big loop
% error
```

## B.2 Pulse

```
function [T] = ADI_Adap_3_D_Pulse(T, m, Loc, p, d, k, h)
% For the pulse function, we distribute the chemical pulse using the actual
% solution over the grid.
firstcheck = 0;
if isempty(find(T>0,1)) == 1
    firstcheck = 1;
end
t = 100;
num2=4*d*k*t;
num1=(num2)^(-3/2);
pinum=pi^(-3/2);
r = 40/h;
for w = 1:p
    x0 = Loc(w,1);
    y0 = Loc(w,2);
    z0 = Loc(w,3);
    for x=2:m-1
        for y=2:m-1
            for z=2:m-1
                if firstcheck
                    if mod(x,4)==1 && mod(y,4)==1 && mod(z,4)==1
                        if sqrt((x0-x)^2+(y0-y)^2+(z0-z)^2) < r
                            T(x,y,z) = (num1*pinum)*
exp(-(h/4)^2*((x0-x)^2+(y0-y)^2+(z0-z)^2)/num2);
                        end
                    end
                else
                    if T(x,y,z)>0
                        if sqrt((x0-x)^2+(y0-y)^2+(z0-z)^2) < r
                            T(x,y,z) = T(x,y,z) +
(num1*pinum)*exp(-(h/4)^2*((x0-x)^2+(y0-y)^2+(z0-z)^2)/num2);
```

```
                              end
                          end
                      end
                  end
              end
          end
      end
end % ADI_Adap_3_D_Pulse
```

## B.3 Move

```
function [Loc] = ADI_Adap_3_D_Move(v, t, Loc, m, p)
% Direction will take the time t and use it to come up with a coordinate
% pair (x, y) with x, y in {-1, 0, 1}. The cell will then move in the
% given direction, e.g., (-1, 1) tells it move up diagonally left.
x = round(cos(t));      % Use t to pick -1, 0, or 1 for the two directions
y = round(sin(x));
z = round(cos(y));
% Set the new location(s)
for a = 1:p
    % Set x direction
    numx = randi(5,1);
    if Loc(a,1) + numx*x < 2          % Check for left boundary problems
        Loc(a,1) = 2;
    else
        if Loc(a,1) + numx*x > m-1   % Check for right boundary problems
            Loc(a,1) = m-1;
        else
            Loc(a,1) = Loc(a,1) + numx*x;
        end % Right boundary check
    end % Left boundary check
    % Set y direction
    numy = randi(5,1);
    if Loc(a,2) + numy*y < 2          % Check for front boundary problems
        Loc(a,2) = 2;
    else
        if Loc(a,2) + numy*y > m-1  % Check for back boundary problems
            Loc(a,2) = m-1;
        else
            Loc(a,2) = Loc(a,2) + numy*y;
        end % Front boundary check
    end % Back boundary check
    % Set z direction
    numz = randi(5,1);
    if Loc(a,3) + numz*z < 2        % Check for upper boundary problems
        Loc(a,3) = 2;
    else
        if Loc(a,3) + numz*z > m-1  % Check for lower boundary problems
            Loc(a,3) = m-1;
        else
            Loc(a,3) = Loc(a,3) + numz*z;
        end % Upper boundary check
    end % Lower boundary check
    % Check to see if we are on a valid point
    if v(Loc(a,1), Loc(a,2), Loc(a,3)) == 0
        if mod(Loc(a,1),4) ~= 1
            if Loc(a,1) < 8 % Check how far left
                Loc(a,1) = 5;
            else
                if Loc(a,1) > m-6 % Check how far right
                    Loc(a,1) = m-4;
                else Loc(a,1) = floor(Loc(a,1)/4)*4+1;
                end
            end
        end
        if mod(Loc(a,2),4) ~= 1
            if Loc(a,2) < 8 % Check how far front
```

```
                Loc(a,2) = 5;
            else
                if Loc(a,2) > m-6 % Check how far back
                    Loc(a,2) = m-4;
                else Loc(a,2) = floor(Loc(a,2)/4)*4+1;
                end
            end
        end
        if mod(Loc(a,3),4) ~= 1
            if Loc(a,3) < 8 % Check how far up
                Loc(a,3) = 5;
            else
                if Loc(a,3) > m-6 % Check how far down
                    Loc(a,3) = m-4;
                else Loc(a,3) = floor(Loc(a,3)/4)*4+1;
                end
            end
        end
    end % If on used point loop
end % Cell loop
end % ADI_Adap_3_D_Move
```

## B.4 Diffuse

```
function [T] = ADI_Adap_3_D_Diffuse(v, PT, k, h, m, d)
% Pulse will calculate the chemical concentration for each point in the
% matrix, based on the previous time step. The new matrix will inherently
% be different because we'll have sources providing more chemical. It uses
% the difference equations from Peaceman and Rachford to solve for the next
% time step.
% As a function, it takes as input the matrix in the current time step and
% solves for the next. It also takes in the space and time discretization
% lengths, and the size of the matrix.
% As output, it returns the matrix after the next time step.
```

```
epsilon = 0.02;
rx = k*d;
ry = k*d;
rz = k*d;
halfkdx = rx/2;
halfkdy = ry/2;
halfkdz = rz/2;
rCx = halfkdx * (1 + epsilon);
rCy = halfkdy * (1 + epsilon);
rCz = halfkdz * (1 + epsilon);
epsC = (6 * epsilon + 2 * epsilon^3) * rx*ry*rz/8;
vHat1 = zeros(m,m,m); % To store the intermediate values in the first third
vHat2 = vHat1;        % To store the intermediate values in the second third
T = vHat2;            % To store the values in the next time step
% Calculate vHat1 from v for the first third
for z = 2:m-1
    for y = 2:m-1
        % Check to see if we need to do this row by looking at it and
        % the 26 possible points around it
        if mod(z,4) == 1 && mod(y,4) == 1
            go = 1;
        else
            nzx = find(v(:,y,z)>0); % Find nonzero terms in x-direction
            go = isempty(nzx)==0;
        end
        if go   % Check to make sure we need to do this row
            % 'know1' is the RHS vector for the first third
            know1 = zeros(m,1);
            % Check to see if we are on an original row
            if mod(y,4) == 1 && mod(z,4) == 1
                % Start where necessary and hold that place in 'start'
                ptr=PT(1,y,z,1);
                start = ptr;
                % Stop at the end of the column
                max=m;
                % Store the coefficient matrices a,b,c for the TDM Solver
                a = zeros(m,1);
                b = a;
                c = b;
                % Calculate second derivatives
                ctr=0;
                while ptr < max
                    ctr=ctr+1;
                    % Set up distances
                    h1 = PT(ptr,y,z,2);
                    h2 = PT(PT(ptr,y,z,3),y,z,2);
                    h3 = (h1+h2)/2;
                    % Set up a, b, c, know1
                    a(ctr)=-rCx/(h2*h3);
                    b(ctr)=1+rCx*(h1+h2)/(h1*h2*h3);
                    c(ctr)=-rCx/(h1*h3);
                    know1(ctr) =
ADI_Adap_3_D_FirstKnown(v,PT,ptr,y,z,rx,ry,rz,rCx,epsC);
                    ptr = PT(ptr,y,z,1);
                end % While for know1
                a = a(1:ctr);
                b = b(1:ctr);
                c = c(1:ctr);
                know1 = know1(1:ctr);
                % Calculate the values to enter into vHat
                [temp] = ADI_Adap_3_D_TDMsolver(a,b,c,know1);
                ptr = start;
                for ctr = 1:length(temp)
                    vHat1(ptr,y,z) = temp(ctr);
                    ptr = PT(ptr,y,z,1);
                end % For loop input
```

```
            else % We are not on a grid column
                ptr = nzx(1);
                max = nzx(length(nzx))+1;
                while ptr < max
                    % 'Counter' keeps track of how many consecutive points
                    % are close enough to use together
                    counter = 1;
                    % 'On' is a logical variable that tells when we can't
                    % use the next point with the previous ones. If 'on'
                    % is true, we keep going.
                    on = 1;
                    % 'Start' holds where the consecutive points started
                    start = ptr;
                    while on
                        if PT(ptr,y,z,1) - ptr < 5 && PT(ptr,y,z,1) < max
                            counter = counter + 1;
                            ptr = PT(ptr,y,z,1);
                        else
                            on = 0;
                        end
                    end
                    if counter < 3
                        if counter == 1
                            [vHat1(ptr,y,z)] =
ADI_Adap_3_D_Single_Point(v,v,PT,ptr,y,z,h,rCx,epsC,rx,ry,rz,'x');
                            ptr = PT(ptr,y,z,1);
                        else
                            [vHat1(start,y,z),vHat1(ptr,y,z)] =
ADI_Adap_3_D_Two_Points(v,v,PT,start,y,z,h,rx,ry,rz,rCx,epsC,'x');
                            ptr = PT(ptr,y,z,1);
                        end
                    else
                        max1 = ptr+1;
                        ptr = start;
                        % Store the coefficient matrices a,b,c for
                        % the TDM Solver
                        a = zeros(m,1);
                        b = a;
                        c = b;
                        % Calculate second derivatives
                        ctr=0;
                        while ptr < max1
                            ctr=ctr+1;
                            % Set up distances
                            h1 = PT(ptr,y,z,2);
                            h2 = PT(PT(ptr,y,z,3),y,z,2);
                            h3 = (h1+h2)/2;
                            % Set up a, b, c, know1
                            a(ctr)=-rCx/(h2*h3);
                            b(ctr)=1+rCx*(h1+h2)/(h1*h2*h3);
                            c(ctr)=-rCx/(h1*h3);
                            know1(ctr) =
ADI_Adap_3_D_FirstKnown(v,PT,ptr,y,z,rx,ry,rz,rCx,epsC);
                            ptr = PT(ptr,y,z,1);
                        end % While for second derivatives
                        a = a(1:ctr);
                        b = b(1:ctr);
                        c = c(1:ctr);
                        know1 = know1(1:ctr);
                        % Calculate the values to enter into vHat1
                        [temp] = ADI_Adap_3_D_TDMsolver(a,b,c,know1);
                        ptr = start;
                        for ctr = 1:length(temp)
                            vHat1(ptr,y,z) = temp(ctr);
                            ptr = PT(ptr,y,z,1);
                        end % For loop input
```

```
                    end % Counter loop
                end % While loop on ptr, max
            end % Mod = 1
        end % isempty check
    end % y loop
end % z loop
% Calculate vHat2 from vHat1 for the second third
for z = 2:m-1
    for x = 2:m-1
        % Check to see if we need to do this row by looking at it and
        % the 26 possible points around it
        if mod(x,4) == 1 && mod(z,4) == 1
            go = 1;
        else
            nzy = find(vHat1(x,:,z)>0);         % Find nonzero terms
                                                % in x-direction
            go = isempty(nzy)==0;
        end
        if go % Check to make sure we need to do this row
            % 'know2' is the RHS vector for the second third
            know2 = zeros(m,1);
            % Check to see if we are on an original row
            if mod(x,4) == 1 && mod(z,4) == 1
                % Start where necessary and hold that place in 'start'
                ptr=PT(x,1,z,4);
                start = ptr;
                % Stop at the end of the column
                max=m;
                % Store the coefficient matrices a,b,c for the TDM Solver
                a = zeros(m,1);
                b = a;
                c = b;
                % Calculate a,b,c,know2
                ctr=0;
                while ptr < max
                    ctr=ctr+1;
                    % Set up distances
                    d1 = PT(x,ptr,z,5);
                    d2 = PT(x,PT(x,ptr,z,6),z,5);
                    d3 = (d1+d2)/2;
                    % Set up a, b, c, know2
                    a(ctr)=-rCy/(d2*d3);
                    b(ctr)=1+rCy*(d1+d2)/(d1*d2*d3);
                    c(ctr)=-rCy/(d1*d3);
                    know2(ctr) = vHat1(x,ptr,z)+c(ctr)*v(x,PT(x,ptr,z,4),z)
+...(b(ctr)-1)*v(x,ptr,z)+a(ctr)*v(x,PT(x,ptr,z,6),z);
                    ptr = PT(x,ptr,z,4);
                end % While for second derivatives
                a = a(1:ctr);
                b = b(1:ctr);
                c = c(1:ctr);
                know2 = know2(1:ctr);
                % Calculate the values to enter into vHat
                [temp] = ADI_Adap_3_D_TDMsolver(a,b,c,know2);
                ptr = start;
                for ctr = 1:length(temp)
                    vHat2(x,ptr,z) = temp(ctr);
                    ptr = PT(x,ptr,z,4);
                end % For loop input
            else % We are not on a grid column
                ptr = nzy(1);
                max = nzy(length(nzy))+1;
                while ptr < max
                    % 'Counter' keeps track of how many consecutive points
                    % are close enough to use together
                    counter = 1;
```

```
% 'On' is a logical variable that tells when we can't
% use the next point with the previous ones. If 'on'
% is true, we keep going.
on = 1;
% 'Start' holds where the consecutive points started
start = ptr;
while on
    if PT(x,ptr,z,4) - ptr < 5 && PT(x,ptr,z,4) < max
        counter = counter + 1;
        ptr = PT(x,ptr,z,4);
    else
        on = 0;
    end
end
if counter < 3
    if counter == 1
        [vHat2(x,ptr,z)] = ADI_Adap_3_D_Single_Point
      (vHat1,v,PT,x,ptr,z,h,rCy,epsC,rx,ry,rz,'y');
        ptr = PT(x,ptr,z,4);
    else
        [vHat2(x,start,z),vHat2(x,ptr,z)] =
    ADI_Adap_3_D_Two_Points
    (vHat1,v,PT,x,start,z,h,rx,ry,rz,rCy,epsC,'y');
        ptr = PT(x,ptr,z,4);
    end
else
    max1 = ptr+1;
    ptr = start;
    % Store the coefficient matrices a,b,c for
    % the TDM Solver
    a = zeros(m,1);
    b = a;
    c = b;
```

```
                    % Calculate second derivatives
                    ctr=0;
                    while ptr < max1
                        ctr=ctr+1;
                        % Set up distances
                        d1 = PT(x,ptr,z,5);
                        d2 = PT(x,PT(x,ptr,z,6),z,5);
                        d3 = (d1+d2)/2;
                        % Set up a, b, c, know2
                        a(ctr)=-rCy/(d2*d3);
                        b(ctr)=1+rCy*(d1+d2)/(d1*d2*d3);
                        c(ctr)=-rCy/(d1*d3);
                        know2(ctr) = vHat1(x,ptr,z)+...
                        c(ctr)*v(x,PT(x,ptr,z,4),z)+...
        (b(ctr)-1)*v(x,ptr,z)+a(ctr)*v(x,PT(x,ptr,z,6),z);
                        ptr = PT(x,ptr,z,4);
                    end % While for second derivatives
                    a = a(1:ctr);
                    b = b(1:ctr);
                    c = c(1:ctr);
                    know2 = know2(1:ctr);
                    % Calculate the values to enter into vHat2
                    [temp] = ADI_Adap_3_D_TDMsolver(a,b,c,know2);
                    ptr = start;
                    for ctr = 1:length(temp)
                        vHat2(x,ptr,z) = temp(ctr);
                        ptr = PT(x,ptr,z,4);
                    end % For loop input
                end % Counter loop
            end % While loop on ptr, max
        end % Mod = 1
    end % isempty check
end % x loop
end % z loop
% Calculate T from vHat2 for the last third
for x = 2:m-1
    for y = 2:m-1
        % Check to see if we need to do this row by looking at it and
        % the 26 possible points around it
        if mod(x,4) == 1 && mod(y,4) == 1
            go = 1;
        else
            nzz = find(vHat2(x,y,:)>0);% Find nonzero terms in x-direction
            go = isempty(nzz)==0;
        end
        if go % Check to make sure we need to do this row
            % 'know3' is the RHS vector for the last third
            know3 = zeros(m,1);
            % Check to see if we are on an original row
            if mod(x,4) == 1 && mod(y,4) == 1
                % Start where necessary and hold that place in 'start'
                ptr=PT(x,y,1,7);
                start = ptr;
                % Stop at the end of the column
                max=m;
                % Store the coefficient matrices a,b,c for the TDM Solver
                a = zeros(m,1);
                b = a;
                c = b;
                % Calculate second derivatives
                ctr=0;
                while ptr < max
                    ctr=ctr+1;
                    % Set up distances
                    m1 = PT(x,y,ptr,8);
                    m2 = PT(x,y,PT(x,y,ptr,9),8);
                    m3 = (m1+m2)/2;
                    % Set up a, b, c, know3
                    a(ctr)=-rCz/(m2*m3);
                    b(ctr)=1+rCz*(m1+m2)/(m1*m2*m3);
                    c(ctr)=-rCz/(m1*m3);
                    know3(ctr)=vHat2(x,y,ptr)+c(ctr)*v(x,y,PT(x,y,ptr,7))+...
(b(ctr)-1)*v(x,y,ptr)+a(ctr)*v(x,y,PT(x,y,ptr,9));
                    ptr = PT(x,y,ptr,7);
                end % While for second derivatives
                a = a(1:ctr);
                b = b(1:ctr);
                c = c(1:ctr);
                know3 = know3(1:ctr);
                % Calculate the values to enter into T
                [temp] = ADI_Adap_3_D_TDMsolver(a,b,c,know3);
                ptr = start;
                for ctr = 1:length(temp)
                    T(x,y,ptr) = temp(ctr);
                    ptr = PT(x,y,ptr,7);
                end % For loop input
            else % We are not on a grid column
                ptr = nzz(1);
                max = nzz(length(nzz))+1;
                while ptr < max
                    % 'Counter' keeps track of how many consecutive points
                    % are close enough to use together
                    counter = 1;
                    % 'On' is a logical variable that tells when we can't
                    % use the next point with the previous ones. If 'on'
                    % is true, we keep going.
                    on = 1;
                    % 'Start' holds where the consecutive points started
                    start = ptr;
```

```
while on
    if PT(x,y,ptr,7) - ptr < 5 && PT(x,y,ptr,7) < max
        counter = counter + 1;
        ptr = PT(x,y,ptr,7);
    else
        on = 0;
    end
end
if counter < 3
    if counter == 1
        [T(x,y,ptr)]=ADI_Adap_3_D_Single_Point
    (vHat2,v,PT,x,y,ptr,h,rCz,epsC,rx,ry,rz,'z');
        ptr = PT(x,y,ptr,7);
    else
  [T(x,y,start),T(x,y,ptr)]=ADI_Adap_3_D_Two_Points
    (vHat2,v,PT,x,y,start,h,rx,ry,rz,rCz,epsC,'z');
        ptr = PT(x,y,ptr,7);
    end
else
    max1 = ptr+1;
    ptr = start;
    % Calculate second derivatives
    ctr=0;
    % Store the coefficient matrices a,b,c for
    % the TDM Solver
    a = zeros(m,1);
    b = a;
    c = b;
    while ptr < max1
        ctr=ctr+1;
        % Set up distances
        m1 = PT(x,y,ptr,8);
        m2 = PT(x,y,PT(x,y,ptr,9),8);
```

```
                            m3 = (m1+m2)/2;
                            % Set up a, b, c, know
                            a(ctr)=-rCz/(m2*m3);
                            b(ctr)=1+rCz*(m1+m2)/(m1*m2*m3);
                            c(ctr)=-rCz/(m1*m3);
                            know3(ctr) = vHat2(x,y,ptr)+...
                            c(ctr)*v(x,y,PT(x,y,ptr,7))+...
        (b(ctr)-1)*v(x,y,ptr)+a(ctr)*v(x,y,PT(x,y,ptr,9));
                            ptr = PT(x,y,ptr,7);
                        end % While for second derivatives
                        a = a(1:ctr);
                        b = b(1:ctr);
                        c = c(1:ctr);
                        know3 = know3(1:ctr);
                        % Calculate the values to enter into vHat
                        [temp] = ADI_Adap_3_D_TDMsolver(a,b,c,know3);
                        ptr = start;
                        for ctr = 1:length(temp)
                            T(x,y,ptr) = temp(ctr);
                            ptr = PT(x,y,ptr,7);
                        end % For loop input
                    end % Counter loop
                end % While loop on ptr, max
            end % Mod = 1
        end % isempty check
    end % y loop
end % x loop
end % ADI_Adap_3_D_Diffuse
```

## B.5 Refine

```
function [T, PT, checker, count] = ADI_Adap_3_D_Refine(T, PT, m, h)
% We loop through the scheme to find areas that need refinement. We search
% for areas where the secant line through two consecutive points has slope
% higher than some tolerance. If we find one, we add a point between them.
% We find its value by the midpoint of the two values it divides. As we are
% searching using all rows and columns, we do check to make sure the two
% points are within the normal discretization of the grid.
tol = 750;       % Tolerance for the secant slope
% 'Checker' keeps track of whether or not we refined this time through the
% matrix
checker = 0;
count = 0;
% Refine in the x-direction first
for y = 2:m-1    % Loop along the interior of v to check refinement
    for z = 2:m-1
        fp = find(T(:,y,z)>0);    % Find nonzero terms (fp for find positive)
        if isempty(fp) == 0       % Check to make sure there is one
            ptr = PT(fp(1),y,z,3);         % At which node we start the loop
            % Loop on the number of points in use currently for this row
            while ptr < fp(length(fp))+1
                % Check to see if there's a close point above to use
                % for refinement
                next = PT(ptr,y,z,1)-ptr<5 && PT(ptr,y,z,1)-ptr>1;
                % Check slope of secant above
                if abs(T(PT(ptr,y,z,1),y,z)-T(ptr,y,z))/PT(ptr,y,z,2) > tol
                                && next
                    checker = 1;
                    count = count + 1;
                    if mod(PT(ptr,y,z,1)+ptr,2) == 1
                        if mod(ptr,4) == 2
                            L = ptr + 1;
                        else L = ptr + 2;
                        end
                    else
                        L = (PT(ptr,y,z,1)+ptr)/2;
                    end
```

```
% Set the value in the matrix for the refined point
T(L,y,z) =T(ptr,y,z)+(L-ptr)*(T(PT(ptr,y,z,1),y,z)-
          T(ptr,y,z))/(PT(ptr,y,z,1)-ptr);
% Set the horizontal values for the pointer matrix
PT(L,y,z,1) = PT(ptr,y,z,1);
PT(L,y,z,2) = h/4*(PT(L,y,z,1)-L);
PT(L,y,z,3) = ptr;
% Insert the new point horizontally
PT(ptr,y,z,1) = L;
PT(ptr,y,z,2) = h/4*(L-ptr);
PT(PT(L,y,z,1),y,z,3) = L;
% Set all of the pointers in the y-direction
left = PT(L,1,z,4);
if left < y % Check for a nonboundary point in
            % the y-direction
    while PT(L,left,z,4) < y
        left = PT(L,left,z,4);
    end
    % Set forward point
    PT(L,y,z,4) = PT(L,left,z,4);
    % Set new backward point for that point
    PT(L,PT(L,left,z,4),z,6) = y;
    % Set backwards point
    PT(L,y,z,6) = left;
    % Set new forward point and distance for that point
    PT(L,left,z,4) = y;
    PT(L,left,z,5) = h/4*(y-left);
    PT(L,y,z,5) = h/4*(PT(L,y,z,4) - y);
else
    PT(L,y,z,4) = left;
    PT(L,y,z,5) = h/4*(left-y);
    PT(L,y,z,6) = 1;
    PT(L,left,z,6) = y;
```

```
            PT(L,1,z,4) = y;
            PT(L,1,z,5) = h/4*(y-1);
        end % There exists a nonboundary point to the left
        % Set all of the pointers in the z-direction
        below = PT(L,y,1,7);
        if below < z % Check for a nonboundary point in
                     % the z-direction
            while PT(L,y,below,7) < z
                below = PT(L,y,below,7);
            end
            % Set point up
            PT(L,y,z,7) = PT(L,y,below,7);
            % Set new downward point for that point
            PT(L,y,PT(L,y,below,7),9) = z;
            % Set downward point
            PT(L,y,z,9) = below;
            % Set new upward point and distance for that point
            PT(L,y,below,7) = z;
            PT(L,y,below,8) = h/4*(z-below);
            PT(L,y,z,8) = h/4*(PT(L,y,z,7) - z);
        else
            PT(L,y,z,7) = below;
            PT(L,y,z,8) = h/4*(below-z);
            PT(L,y,z,9) = 1;
            PT(L,y,below,9) = z;
            PT(L,y,1,7) = z;
            PT(L,y,1,8) = h/4*(z-1);
        end % There exists a nonboundary point below
        % Advance the pointer
        ptr = PT(L,y,z,1);
    else
        % Advance the pointer
        ptr = PT(ptr,y,z,1);
```

```
                end % If secant slope loop
            end % While ptr loop
        end % If isempty loop
    end % z loop
end % y loop
% Refine in the y-direction next
for x = 2:m-1   % Loop along the interior of v to check refinement
    for z = 2:m-1
        fp = find(T(x,:,z)>0);    % Find nonzero terms (fp for find positive)
        if isempty(fp) == 0       % Check to make sure there is one
            ptr = PT(x,fp(1),z,6);         % At which node we start the loop
            % Loop on the number of points in use currently for this row
            while ptr < fp(length(fp))+1
                % Check to see if there's a close point above to use
                % for refinement
                next = PT(x,ptr,z,4)-ptr<5 && PT(x,ptr,z,4)-ptr>1;
                % Check slope of secant above
                if abs(T(x,PT(x,ptr,z,4),z)-T(x,ptr,z))/PT(x,ptr,z,5) > tol
                            && next
                    checker = 1;
                    count = count + 1;
                    if mod(PT(x,ptr,z,4)+ptr,2) == 1
                        if mod(ptr,4) == 2
                            L = ptr + 1;
                        else L = ptr + 2;
                        end
                    else
                        L = (PT(x,ptr,z,4)+ptr)/2;
                    end
                    % Set the value in the matrix for the refined point
                    T(x,L,z) =T(x,ptr,z)+(L-ptr)*(T(x,PT(x,ptr,z,4),z)-
                    T(x,ptr,z))/(PT(x,ptr,z,4)-ptr);
                    % Set the vertical values for the pointer matrix
```

```
PT(x,L,z,4) = PT(x,ptr,z,4);
PT(x,L,z,5) = h/4*(PT(x,L,z,4)-L);
PT(x,L,z,6) = ptr;
% Insert the new point vertically
PT(x,ptr,z,4) = L;
PT(x,ptr,z,5) = h/4*(L-ptr);
PT(x,PT(x,L,z,4),z,6) = L;
% Set all of the pointers in the x-direction
left = PT(1,L,z,1);
if left < x % Check for a nonboundary point in
            % the x-direction
    while PT(left,L,z,1) < x
        left = PT(left,L,z,1);
    end
    % Set forward point
    PT(x,L,z,1) = PT(left,L,z,1);
    % Set new backward point for that point
    PT(PT(left,L,z,1),L,z,3) = x;
    % Set backwards point
    PT(x,L,z,3) = left;
    % Set new forward point and distance for that point
    PT(left,L,z,1) = x;
    PT(left,L,z,2) = h/4*(x-left);
    PT(x,L,z,2) = h/4*(PT(x,L,z,1) - x);
else
    PT(x,L,z,1) = left;
    PT(x,L,z,2) = h/4*(left-x);
    PT(x,L,z,3) = 1;
    PT(left,L,z,3) = x;
    PT(1,L,z,1) = x;
    PT(1,L,z,2) = h/4*(x-1);
end % There exists a nonboundary point
% Set all of the pointers in the z-direction
```

```
                    below = PT(x,L,1,7);
                    if below < z % Check for a nonboundary point below
                        while PT(x,L,below,7) < z
                            below = PT(x,L,below,7);
                        end
                        % Set point up
                        PT(x,L,z,7) = PT(x,L,below,7);
                        % Set new downward point for that point
                        PT(x,L,PT(x,L,below,7),9) = z;
                        % Set downward point
                        PT(x,L,z,9) = below;
                        % Set new upward point and distance for that point
                        PT(x,L,below,7) = z;
                        PT(x,L,below,8) = h/4*(z-below);
                        PT(x,L,z,8) = h/4*(PT(x,L,z,7) - z);
                    else
                        PT(x,L,z,7) = below;
                        PT(x,L,z,8) = h/4*(below-z);
                        PT(x,L,z,9) = 1;
                        PT(x,L,below,9) = z;
                        PT(x,L,1,7) = z;
                        PT(x,L,1,8) = h/4*(z-1);
                    end % There exists a nonboundary point below
                    % Advance the pointer
                    ptr = PT(x,L,z,4);
                else
                    % Advance the pointer
                    ptr = PT(x,ptr,z,4);
                end % If secant slope loop
            end % While ptr loop
        end % If isempty loop
    end % z loop
end % x loop
```

```
% Refine in the z-direction last
for x = 2:m-1   % Loop along the interior of v to check refinement
    for y = 2:m-1
        fp = find(T(x,y,:)>0); % Find nonzero terms (fp for find positive)
        if isempty(fp) == 0 % Check to make sure there is one
            ptr = PT(x,y,fp(1),9); % At which node we start the loop
            % Loop on the number of points in use currently for this row
            while ptr < fp(length(fp))+1
                % Check to see if there’s a close point above to use
                % for refinement
                next = PT(x,y,ptr,7)-ptr<5 && PT(x,y,ptr,7)-ptr>1;
                % Check slope of secant above
                if abs(T(x,y,PT(x,y,ptr,7))-T(x,y,ptr))/PT(x,y,ptr,8)>tol
                                && next
                    checker = 1;
                    count = count + 1;
                    if mod(PT(x,y,ptr,7)+ptr,2) == 1
                        if mod(ptr,4) == 2
                            L = ptr + 1;
                        else L = ptr + 2;
                        end
                    else
                        L = (PT(x,y,ptr,7)+ptr)/2;
                    end
                    % Set the value in the matrix for the refined point
                    T(x,y,L) =T(x,y,ptr)+(L-ptr)*(T(x,y,PT(x,y,ptr,7))-
                    T(x,y,ptr))/(PT(x,y,ptr,7)-ptr);
                    % Set the altitudinal values for the pointer matrix
                    PT(x,y,L,7) = PT(x,y,ptr,7);
                    PT(x,y,L,8) = h/4*(PT(x,y,L,7)-L);
                    PT(x,y,L,9) = ptr;
                    % Insert the new point altitudinally
                    PT(x,y,ptr,7) = L;
                    PT(x,y,ptr,8) = h/4*(L-ptr);
                    PT(x,y,PT(x,y,L,7),9) = L;
                    % Set all of the pointers in the x-direction
                    left = PT(1,y,L,1);
                    if left < x % Check for a nonboundary point in
                                % the x-direction
                        while PT(left,y,L,1) < x
                            left = PT(left,y,L,1);
                        end
                        % Set forward point
                        PT(x,y,L,1) = PT(left,y,L,1);
                        % Set new backward point for that point
                        PT(PT(left,y,L,1),y,L,3) = x;
                        % Set backwards point
                        PT(x,y,L,3) = left;
                        % Set new forward point and distance for that point
                        PT(left,y,L,1) = x;
                        PT(left,y,L,2) = h/4*(x-left);
                        PT(x,y,L,2) = h/4*(PT(x,y,L,1) - x);
                    else
                        PT(x,y,L,1) = left;
                        PT(x,y,L,2) = h/4*(left-x);
                        PT(x,y,L,3) = 1;
                        PT(left,y,L,3) = x;
                        PT(1,y,L,1) = x;
                        PT(1,y,L,2) = h/4*(x-1);
                    end % There exists a nonboundary point
                    % Set all of the pointers in the y-direction
                    below = PT(x,1,L,4);
                    if below < y % Check for a nonboundary point in
                                 % the y-direction
                        while PT(x,below,L,4) < y
                            below = PT(x,below,L,4);
```

```
                        end
                        % Set point up
                        PT(x,y,L,4) = PT(x,below,L,4);
                        % Set new downward point for that point
                        PT(x,PT(x,below,L,4),L,6) = y;
                        % Set downward point
                        PT(x,y,L,6) = below;
                        % Set new upward point and distance for that point
                        PT(x,below,L,4) = y;
                        PT(x,below,L,5) = h/4*(y-below);
                        PT(x,y,L,5) = h/4*(PT(x,y,L,4) - y);
                    else
                        PT(x,y,L,4) = below;
                        PT(x,y,L,5) = h/4*(below-y);
                        PT(x,y,L,6) = 1;
                        PT(x,below,L,6) = y;
                        PT(x,1,L,4) = y;
                        PT(x,1,L,5) = h/4*(y-1);
                    end % There exists a nonboundary point below
                    % Advance the pointer
                    ptr = PT(x,y,L,7);
                else
                    % Advance the pointer
                    ptr = PT(x,y,ptr,7);
                end % If secant slope loop
            end % While ptr loop
        end % If isempty loop
    end % z loop
end % y loop
end % ADI_Adap_3_D_Refine
```

## B.6 Coarsen

```
function [T, PT, checker, ct] = ADI_Adap_3_D_Coarsen(T, PT, m)
```

```
% The coarsen function will be executed every five time steps to see if any
% of the added points can now be removed because they are no longer
% relevant in terms of calculating derivatives. We loop through the added
% points and check to see if the difference between them and the points
% around them fall below a given tolerance level. If so, the points are
% removed, and the pointer array (PT) is modified.
tol = 1;            % Tolerance for removing points
checker = 0;
ct = 0;
% Coarsen over the whole matrix along the columns
for x = 2:m-1   % Check all original interior columns
    for y = 2:m-1
        nonzero = find(T(x,y,:)>0);      % Find nonzero terms
        if isempty(nonzero)==0
            ptr = nonzero(1);
            while ptr < m
                % Check if it's a grid point
                if mod(x,4)==1 && mod(y,4)==1 && mod(ptr,4)==1
                    ptr = PT(x,y,ptr,7);
                else % If it's not a grid point
                    % Check the derivatives in the x-direction
                    % Forward
                    if PT(x,y,ptr,1) - ptr < 5
                        forw = abs(T(PT(x,y,ptr,1),y,ptr)-T(x,y,ptr))
                                                   /PT(x,y,ptr,2)<tol;
                    else
                        forw = 1;
                    end
                    % Backward
                    if ptr - PT(x,y,ptr,3)  < 5
                        back = abs(T(PT(x,y,ptr,3),y,ptr)-T(x,y,ptr))
                                          /PT(PT(x,y,ptr,3),y,ptr,2)<tol;
                    else
                        back = 1;
                    end
                    % Check the derivatives in the y-direction
                    % Front
                    if PT(x,y,ptr,4) - ptr < 5
                        front = abs(T(x,PT(x,y,ptr,4),ptr)-T(x,y,ptr))
                                                   /PT(x,y,ptr,5)<tol;
                    else
                        front = 1;
                    end
                    % Behind
                    if ptr - PT(x,y,ptr,6) < 5
                        behind = abs(T(x,PT(x,y,ptr,6),ptr)-T(x,y,ptr))
                                        /PT(x,PT(x,y,ptr,6),ptr,5)<tol;
                    else
                        behind = 1;
                    end
                    % Check the derivatives in the x-direction
                    % Upward
                    if PT(x,y,ptr,7) - ptr < 5
                        up = abs(T(x,y,PT(x,y,ptr,7))-T(x,y,ptr))
                                               /PT(x,y,ptr,8)<tol;
                    else
                        up = 1;
                    end
                    % Downward
                    if ptr - PT(x,y,ptr,9) < 5
                        down = abs(T(x,y,PT(x,y,ptr,9))-T(x,y,ptr))
                                         /PT(x,y,PT(x,y,ptr,9),8)<tol;
                    else
                        down = 1;
                    end
                    if forw && back && front && behind && up && down
```

```
                    % If it's unneeded
                        checker = 1;
                        ct = ct + 1;
                        % Set the point to zero
                        T(x,y,ptr) = 0;
                        % Remove the point in the x-direction
                        PT(PT(x,y,ptr,3),y,ptr,1) = PT(x,y,ptr,1);
                        PT(PT(x,y,ptr,3),y,ptr,2) =
                            PT(PT(x,y,ptr,3),y,ptr,2) + PT(x,y,ptr,2);
                        PT(PT(x,y,ptr,1),y,ptr,3) = PT(x,y,ptr,3);
                        % Remove the point in the y-direction
                        PT(x,PT(x,y,ptr,6),ptr,4) = PT(x,y,ptr,4);
                        PT(x,PT(x,y,ptr,6),ptr,5) =
                            PT(x,PT(x,y,ptr,6),ptr,5)+PT(x,y,ptr,5);
                        PT(x,PT(x,y,ptr,4),ptr,6) = PT(x,y,ptr,6);
                        % Remove the point in the z-direction
                        PT(x,y,PT(x,y,ptr,9),7) = PT(x,y,ptr,7);
                        PT(x,y,PT(x,y,ptr,9),8) =
                            PT(x,y,PT(x,y,ptr,9),8)+PT(x,y,ptr,8);
                        PT(x,y,PT(x,y,ptr,7),9) = PT(x,y,ptr,9);
                        % Erase the information in the pointer matrix
                        keep = PT(x,y,ptr,7);
                        PT(x,y,ptr,:) = 0;
                        ptr = keep;
                    else % If it's still needed
                        ptr = PT(x,y,ptr,7);
                    end % If unneeded
                end % If grid point
            end % While loop up the z's
        end % If nonzero
    end % y loop
end % x loop
end % ADI_Adap_3_D_Coarsen
```

## B.7 Thomas Algorithm

```
function [x] = ADI_Adap_3_D_TDMsolver(a,b,c,d)
% a, b, c are the column vectors for the compressed tridiagonal matrix,
% d is the right vector
n = length(d);       % n is the number of elements in the array
x = zeros(n,1);
% Modify the first-row coefficients
c(1) = c(1) / b(1);    % Division by zero risk.
d(1) = d(1) / b(1);
for i = 2:n-1
    temp = b(i) - a(i) * c(i-1);
    c(i) = c(i) / temp;
    d(i) = (d(i) - a(i) * d(i-1))/temp;
end % For loop adjusting coefficients
d(n) = (d(n) - a(n) * d(n-1))/(b(n) - a(n) * c(n-1));
% Now back substitute.
x(n) = d(n);
for i = n-1:-1:1
    x(i) = d(i) - c(i) * x(i+1);
end % For loop back substitution
end % ADI_Adap_3_D_TDMsolver
```

## B.8 Isolated Single Point

```
function [answ] = ADI_Adap_3_D_Single_Point
                        (T1, T2, PT, x, y, z, h, rc, eps, rx, ry, rz, xyz)
% We use this function to calculate the new value at a point when that
% point is isolated on a refined row. We cannot use the Thomas
% algorithm due to lack of support. The input matrix may be 'v', 'vHat1'
% or 'vHat2', so we'll use 'T' to represent the matrix temporarily in use,
% while 'T2' (v) is the other matrix we may need to calculate the new point.
% We also use the pointer matrix, the x, y, and z-coordinates, the current
% discretization 'h', the 'rCoef', the 'epsCoef', 'kd'=k*d (for FirstKnown),
% and some way to tell which way we were moving.
if xyz == 'x' % If we were diffusing in the x-direction
    % Set the x-coordinates before and after the current one
    if mod(x,4) == 1
        before = x - 4;
        after = x + 4;
    else
        before = floor((x-1)/4)*4 + 1;
        after = floor((x+2)/4)*4 + 1;
    end
    % Set the shadow y- and z-coordinates
    if mod(y,4) == 1
        yback = y - 4;
        yup = y + 4;
    else
        yback = floor((y-1)/4)*4 + 1;
        yup = floor((y+2)/4)*4 + 1;
    end
    if mod(z,4) == 1
        zback = z - 4;
        zup = z + 4;
    else
        zback = floor((z-1)/4)*4 + 1;
        zup = floor((z+2)/4)*4 + 1;
    end
    % Calculate the RHS for the current point
    rhs = ADI_Adap_3_D_FirstKnown(T1, PT, x, y, z, rx, ry, rz, rc, eps);
    % Set the shadow point forward and its distance from the point
    temfor = (T1(after,yup,zback)+T1(after,yup,zup)+T1(after,yback,zback)
                                  +T1(after,yback,zup))/4;
    h1 = h/4*(after-x);
    % Set the shadow point below and its distance from the point
    tembeh = (T1(before,yup,zback)+T1(before,yup,zup)+
            T1(before,yback,zback)+T1(before,yback,zup))/4;
    h2 = h/4*(x-before);
    h3 = (h1+h2)/2;
    % Set the coefficients for the calculation
    a = -rc/(h2*h3);
    b = 1+rc*(h1+h2)/(h1*h2*h3);
    c = -rc/(h1*h3);
    answ = (rhs - c*temfor - a*tembeh)/b;
else % We were diffusing along a row
    if xyz == 'y' % If we were diffusing in the y-direction
        % Set the y-coordinates before and after the current one
        if mod(y,4) == 1
            before = y - 4;
            after = y + 4;
        else
            before = floor((y-1)/4)*4 + 1;
            after = floor((y+2)/4)*4 + 1;
        end
        % Set the shadow x- and z-coordinates
        if mod(x,4) == 1
            xback = x - 4;
            xup = x + 4;
        else
            xback = floor((x-1)/4)*4 + 1;
            xup = floor((x+2)/4)*4 + 1;
        end
        if mod(z,4) == 1
            zback = z - 4;
            zup = z + 4;
        else
            zback = floor((z-1)/4)*4 + 1;
            zup = floor((z+2)/4)*4 + 1;
        end
        % Calculate the RHS for the current point
        d1 = PT(x,y,z,5);
        d2 = PT(x,PT(x,y,z,6),z,5);
        d3 = (d1+d2)/2;
        rhs = T1(x,y,z)-rc/(d1*d3)*T2(x,PT(x,y,z,4),z)+rc*(d1+d2)/
            (d1*d2*d3)*T2(x,y,z)-rc/(d2*d3)*T2(x,PT(x,y,z,6),z);
        % Set the shadow point forward and its distance from the point
        temfor = (T1(xup,after,zback)+T1(xup,after,zup)+
                T1(xback,after,zback)+T1(xback,after,zup))/4;
        d1 = h/4*(after-y);
        % Set the shadow point below and its distance from the point
        tembeh = (T1(xup,before,zback)+T1(xup,before,zup)+
                T1(xback,before,zback)+T1(xback,before,zup))/4;
        d2 = h/4*(y-before);
        d3 = (d1+d2)/2;
        % Set the coefficients for the calculation
        a = -rc/(d2*d3);
        b = 1+rc*(d1+d2)/(d1*d2*d3);
        c = -rc/(d1*d3);
        answ = (rhs - c*temfor - a*tembeh)/b;
    else % We are diffusing in the z-direction
        % Set the z-coordinates before and after the current one
        if mod(z,4) == 1
            before = z - 4;
            after = z + 4;
        else
            before = floor((z-1)/4)*4 + 1;
            after = floor((z+2)/4)*4 + 1;
        end
        % Set the shadow x- and y-coordinates
        if mod(x,4) == 1
            xback = x - 4;
            xup = x + 4;
        else
            xback = floor((x-1)/4)*4 + 1;
            xup = floor((x+2)/4)*4 + 1;
        end
        if mod(y,4) == 1
            yback = y - 4;
            yup = y + 4;
        else
            yback = floor((y-1)/4)*4 + 1;
            yup = floor((y+2)/4)*4 + 1;
        end
        % Calculate the RHS for the current point
        m1 = PT(x,y,z,8);
        m2 = PT(x,y,PT(x,y,z,9),8);
        m3 = (m1+m2)/2;
        rhs = T1(x,y,z)-rc/(m1*m3)*T2(x,y,PT(x,y,z,7))+rc*(m1+m2)/
            (m1*m2*m3)*T2(x,y,z)-rc/(m2*m3)*T2(x,y,PT(x,y,z,9));
        % Set the shadow point forward and its distance from the point
        temfor = (T1(xup,yup,after)+T1(xback,yup,after)+
                T1(xup,yback,after)+T1(xback,yback,after))/4;
        m1 = h/4*(after-z);
        % Set the shadow point below and its distance from the point
        tembeh = (T1(xup,yup,before)+T1(xback,yup,before)+
                T1(xup,yback,before)+T1(xback,yback,before))/4;
        m2 = h/4*(z-before);
        m3 = (m1+m2)/2;
        % Set the coefficients for the calculation
        a = -rc/(m2*m3);
        b = 1+rc*(m1+m2)/(m1*m2*m3);
        c = -rc/(m1*m3);
        answ = (rhs - c*temfor - a*tembeh)/b;
```

```
    end
end
end % ADI_Adap_3_D_Single_Point
```

## B.9 Isolated Pair of Points

```
function [ans1, ans2] = ADI_Adap_3_D_Two_Points
                    (T1, T2, PT, x, y, z, h, rx, ry, rz, rc, eps, xyz)
% This function calculates new third-points for two isolated points on a
% row. We take an input matrix, be it 'v', 'vHat1', or 'vHat2', and a
% second matrix if necessary, the pointer matrix, the coordinates of the
% first of the two points, the current discretization, 'kd'=k*d (for
% FirstKnown), the r-coefficient, the epsilon coefficient, and a character
% input to tell whether we are diffusing along the x-, y-, or z-direction.
if xyz == 'x' % We are diffusing in the x-direction
    % Set the x-coordinates before and after the two points
    if mod(x,4) == 1
        before = x-4;
    else
        before = floor((x-1)/4)*4 + 1;
    end
    num = PT(x,y,z,1);
    if mod(num,4) == 1
        after = num+4;
    else
        after = floor((num+2)/4)*4 + 1;
    end
    % Set the shadow y- and z-coordinates
    if mod(y,4) == 1
        yback = y - 4;
        yup = y + 4;
    else
        yback = floor((y-1)/4)*4 + 1;
        yup = floor((y+2)/4)*4 + 1;
```

```
end
if mod(z,4) == 1
    zback = z - 4;
    zup = z + 4;
else
    zback = floor((z-1)/4)*4 + 1;
    zup = floor((z+2)/4)*4 + 1;
end
% Calculate the right-hand side
rhs1 = ADI_Adap_3_D_FirstKnown(T1, PT, x, y, z, rx, ry, rz, rc, eps);
rhs2 = ADI_Adap_3_D_FirstKnown(T1, PT, num, y, z, rx, ry, rz, rc, eps);
% Set the shadow point above and its distance from the point
temab = (T1(after,yup,zup)+T1(after,yup,zback)+T1(after,yback,zup)
                                        +T1(after,yback,zback))/4;
% We'll use t for the shadow point on top
t1 = h/4*(after-num);
t2 = PT(x,y,z,2);
t3 = (t1+t2)/2;
% Set the shadow point below and its distance from the point
tembel = (T1(before,yup,zup)+T1(before,yup,zback)+T1(before,yback,zup)
                                           +T1(before,yback,zback))/4;
% We use d for the shadow point down below
d1 = t2;
d2 = h/4*(x-before);
d3 = (d1+d2)/2;
% Set the coefficients for the calculation
% We use a 1 for the coefficients that would be in
% the first row, 2 for the second
a1 = -rc/(d2*d3);
b1 = 1+rc*(d1+d2)/(d1*d2*d3);
c1 = -rc/(d1*d3);
a2 = -rc/(t2*t3);
b2 = 1+rc*(t1+t2)/(t1*t2*t3);
```

```
    c2 = -rc/(t1*t3)
    ans2 = (a1*a2*tembel-a2*rhs1+b1*rhs2-b1*c2*temab);
    ans2 = ans2/(b1*b2-a2*c1);
    ans1 = (rhs1 - c1*ans2-a1*tembel)/b1;
else
    if xyz == 'y' % We are diffusing in the y-direction
        % Set the y-coordinates before and after the two points
        if mod(y,4) == 1
            before = y-4;
        else
            before = floor((y-1)/4)*4 + 1;
        end
        num = PT(x,y,z,4);
        if mod(num,4) == 1
            after = num+4;
        else
            after = floor((num+2)/4)*4 + 1;
        end
        % Set the shadow x- and z-coordinates
        if mod(x,4) == 1
            xback = x - 4;
            xup = x + 4;
        else
            xback = floor((x-1)/4)*4 + 1;
            xup = floor((x+2)/4)*4 + 1;
        end
        if mod(z,4) == 1
            zback = z - 4;
            zup = z + 4;
        else
            zback = floor((z-1)/4)*4 + 1;
            zup = floor((z+2)/4)*4 + 1;
        end
```

```
% Calculate the right-hand side
t1 = PT(x,num,z,5);
t2 = PT(x,y,z,5);
t3 = (t1+t2)/2;
d1 = t2;
d2 = PT(x,PT(x,y,z,6),z,5);
d3 = (d1+d2)/2;
rhs1 = T1(x,y,z)-rc/(d1*d3)*T2(x,num,z)+rc*(d1+d2)/(d1*d2*d3)*
        T2(x,y,z)-rc/(d2*d3)*T2(x,PT(x,y,z,6),z);
rhs2 = T1(x,num,z)-rc/(t1*t3)*T2(x,PT(x,num,z,4),z)+
        rc*(t1+t2)/(t1*t2*t3)*T2(x,num,z)-rc/(t2*t3)*T2(x,y,z);
% Set the shadow point above and its distance from the point
temab = (T1(xup,after,zup)+T1(xup,after,zback)+
        T1(xback,after,zup)+T1(xback,after,zback))/4;
% We'll use t for the shadow point on top
t1 = h/4*(after-num);
t2 = PT(x,y,z,5);
t3 = (t1+t2)/2;
% Set the shadow point below and its distance from the point
tembel = (T1(xup,before,zup)+T1(xup,before,zback)+
        T1(xback,before,zup)+T1(xback,before,zback))/4;
% We use d for the shadow point down below
d1 = t2;
d2 = h/4*(y-before);
d3 = (d1+d2)/2;
% Set the coefficients for the calculation
% We use a 1 for the coefficients that would be in
% the first row, 2 for the second
a1 = -rc/(d2*d3);
b1 = 1+rc*(d1+d2)/(d1*d2*d3);
c1 = -rc/(d1*d3);
a2 = -rc/(t2*t3);
b2 = 1+rc*(t1+t2)/(t1*t2*t3);
```

```
        c2 = -rc/(t1*t3);
        ans2 = (a1*a2*tembel-a2*rhs1+b1*rhs2-b1*c2*temab);
        ans2 = ans2/(b1*b2-a2*c1);
        ans1 = (rhs1 - c1*ans2-a1*tembel)/b1;
    else % Diffusing in the z-direction
        % Set the z-coordinates before and after the two points
        if mod(z,4) == 1
            before = z-4;
        else
            before = floor((z-1)/4)*4 + 1;
        end
        num = PT(x,y,z,7);
        if mod(num,4) == 1
            after = num+4;
        else
            after = floor((num+2)/4)*4 + 1;
        end
        % Set the shadow x- and y-coordinates
        if mod(x,4) == 1
            xback = x - 4;
            xup = x + 4;
        else
            xback = floor((x-1)/4)*4 + 1;
            xup = floor((x+2)/4)*4 + 1;
        end
        if mod(y,4) == 1
            yback = y - 4;
            yup = y + 4;
        else
            yback = floor((y-1)/4)*4 + 1;
            yup = floor((y+2)/4)*4 + 1;
        end
        % Calculate the right-hand side
```

```
t1 = PT(x,y,num,8);
t2 = PT(x,y,z,8);
t3 = (t1+t2)/2;
d1 = t2;
d2 = PT(x,y,PT(x,y,z,9),8);
d3 = (d1+d2)/2;
rhs1 = T1(x,y,z)-rc/(d1*d3)*T2(x,y,PT(x,y,z,7))+rc*(d1+d2)/
        (d1*d2*d3)*T2(x,y,z)-rc/(d2*d3)*T2(x,y,PT(x,y,z,9));
rhs2 = T1(x,y,z)-rc/(t1*t3)*T2(x,y,PT(x,y,z,7))+rc*(t1+t2)/
        (t1*t2*t3)*T2(x,y,z)-rc/(t2*t3)*T2(x,y,PT(x,y,z,9));
% Set the shadow point above and its distance from the point
temab = (T1(xup,yup,after)+T1(xback,yup,after)+
        T1(xup,yback,after)+T1(xback,yback,after))/4;
% We'll use t for the shadow point on top
t1 = h/4*(after-num);
t2 = PT(x,y,z,2);
t3 = (t1+t2)/2;
% Set the shadow point below and its distance from the point
tembel = (T1(xup,yup,before)+T1(xback,yup,before)+
        T1(xup,yback,before)+T1(xback,yback,before))/4;
% We use d for the shadow point down below
d1 = t2;
d2 = h/4*(z-before);
d3 = (d1+d2)/2;
% Set the coefficients for the calculation
% We use a 1 for the coefficients that would be in
% the first row, 2 for the second
a1 = -rc/(d2*d3);
b1 = 1+rc*(d1+d2)/(d1*d2*d3);
c1 = -rc/(d1*d3);
a2 = -rc/(t2*t3);
b2 = 1+rc*(t1+t2)/(t1*t2*t3);
c2 = -rc/(t1*t3);
```

```
        ans2 = (a1*a2*tembel-a2*rhs1+b1*rhs2-b1*c2*temab);
        ans2 = ans2/(b1*b2-a2*c1);
        ans1 = (rhs1 - c1*ans2-a1*tembel)/b1;
    end
end % ADI_Adap_3_D_Two_Points
```

## B.10 First Known Value

```
function [a] = ADI_Adap_3_D_FirstKnown
                (T, PT, x, y, z, rx, ry, rz, rCx, epsC)
% This function determines the known values for the first "third-time step"
% in the algorithm. I set it as a separate function because of all of the
% calculations that need to be made.

% This is the list of distances dictated by the input point. Distances with
% an 'h' are distances in the x-direction; distances with a 'd' are in the
% y-direction; and distances with an 'm' are in the z-direction. Distances
% where the letter is followed by a '1' are forward distances; if followed
% by a '2', it is a backward distance; and the '3' signifies the average of
% the '1' and '2' distances.
% Distances around the input point
h1 = PT(x,y,z,2);
h2 = PT(PT(x,y,z,3),y,z,2);
h3 = (h1 + h2)/2;
d1 = PT(x,y,z,5);
d2 = PT(x,PT(x,y,z,6),z,5);
d3 = (d1 + d2)/2;
m1 = PT(x,y,z,8);
m2 = PT(x,y,PT(x,y,z,9),8);
m3 = (m1 + m2)/2;
% Distances once we move off of the input point.
% For the refined points, we need to make sure that we move in the proper
% direction, then find distances around those points. These next variables
% will have multiple numbers after them.
% These numbers come from the subscripts in the discretization, with only
% as many as necessary listed. For subscripts that would normally be a -1,
% we will replace it with N1. All other 1's are to be considered positive.
% For example, if we are looking at the distance between u_(i+1,j,k-1) and
% u_(i,j,k-1), we would list it as h10N1, it being a forward distance in
% the x-direction, where the addend to 'j' is zero, and the addend to 'k'
% is negative one.
% Distances for the operator A_2
d11 = PT(x,y,PT(x,y,z,7),5);
d21 = PT(x,PT(x,y,PT(x,y,z,7),6),PT(x,y,z,7),5);
d31 = (d11 + d21)/2;
d10 = PT(x,y,z,5);
d20 = PT(x,PT(x,y,z,6),z,5);
d30 = (d10 + d20)/2;
d1N1 = PT(x,y,PT(x,y,z,9),5);
d2N1 = PT(x,PT(x,y,PT(x,y,z,9),6),PT(x,y,z,9),5);
d3N1 = (d1N1 + d2N1)/2;
% Distances for the operator A_1
h111 = PT(x,PT(x,y,PT(x,y,z,7),4),PT(x,y,z,7),2);
h211 = PT(PT(x,PT(x,y,PT(x,y,z,7),4),PT(x,y,z,7),3),
            PT(x,y,PT(x,y,z,7),4),PT(x,y,z,7),2);
h311 = (h111 + h211)/2;
h101 = PT(x,y,PT(x,y,z,7),2);
h201 = PT(PT(x,y,PT(x,y,z,7),3),y,PT(x,y,z,7),2);
h301 = (h101 + h201)/2;
h1N11 = PT(x,PT(x,y,PT(x,y,z,7),6),PT(x,y,z,7),2);
h2N11 = PT(PT(x,PT(x,y,PT(x,y,z,7),6),PT(x,y,z,7),3),
            PT(x,y,PT(x,y,z,7),6),PT(x,y,z,7),2);
h3N11 = (h1N11 + h2N11)/2;
h110 = PT(x,PT(x,y,z,4),z,2);
h210 = PT(PT(x,PT(x,y,z,4),z,3),PT(x,y,z,4),z,2);
h310 = (h110 + h210)/2;
h100 = PT(x,y,z,2);
```

```
h200 = PT(PT(x,y,z,3),y,z,2);

h300 = (h100 + h200)/2;

h1N10 = PT(x,PT(x,y,z,6),z,2);

h2N10 = PT(PT(x,PT(x,y,z,6),z,3),PT(x,y,z,6),z,2);

h3N10 = (h1N10 + h2N10)/2;

h11N1 = PT(x,PT(x,y,PT(x,y,z,9),4),PT(x,y,z,9),2);

h21N1 = PT(PT(x,PT(x,y,PT(x,y,z,9),4),PT(x,y,z,9),3),
            PT(x,y,PT(x,y,z,9),4),PT(x,y,z,9),2);

h31N1 = (h11N1 + h21N1)/2;

h10N1 = PT(x,y,PT(x,y,z,9),2);

h20N1 = PT(PT(x,y,PT(x,y,z,9),3),y,PT(x,y,z,9),2);

h30N1 = (h10N1 + h20N1)/2;

h1N1N1 = PT(x,PT(x,y,PT(x,y,z,9),6),PT(x,y,z,9),2);

h2N1N1 = PT(PT(x,PT(x,y,PT(x,y,z,9),6),PT(x,y,z,9),3),
            PT(x,y,PT(x,y,z,9),6),PT(x,y,z,9),2);

h3N1N1 = (h1N1N1 + h2N1N1)/2;

u000 = T(x,y,z)*(1+(rCx-rx)*(h1+h2)/(h1*h2*h3)-ry*(d1+d2)/(d1*d2*d3)-...
    rz*(m1+m2)/(m1*m2*m3)-(h100+h200)*(d10+d20)*(m1+m2)*epsC/...
    (h100*h200*h300*d10*d20*d30*m1*m2*m3));

u001 = T(x,y,PT(x,y,z,7))*(rz/(m1*m3)+(h101+h201)*(d11+d21)*epsC/
    (h101*h201*h301*d11*d21*d31*m1*m3));

u00N1 = T(x,y,PT(x,y,z,9))*(rz/(m2*m3)+(h10N1+h20N1)*(d1N1+d2N1)*epsC/
    (h10N1*h20N1*h30N1*d1N1*d2N1*d3N1*m2*m3));

u010 = T(x,PT(x,y,z,4),z)*(ry/(d1*d3)+(h110+h210)*(m1+m2)*epsC/
    (h110*h210*h310*d10*d30*m1*m2*m3));

u011 = -T(x,PT(x,y,PT(x,y,z,7),4),PT(x,y,z,7))*(h111+h211)*epsC/
    (h111*h211*h311*d11*d31*m1*m3);

u01N1 = -T(x,PT(x,y,PT(x,y,z,9),4),PT(x,y,z,9))*(h11N1+h21N1)*epsC/
    (h11N1*h21N1*h31N1*d1N1*d3N1*m2*m3);

u0N10 = T(x,PT(x,y,z,6),z)*(ry/(d2*d3)+(h1N10+h2N10)*(m1+m2)*epsC/
    (h1N10*h2N10*h3N10*d20*d30*m1*m2*m3));

u0N11 = -T(x,PT(x,y,PT(x,y,z,7),6),PT(x,y,z,7))*(h1N11+h2N11)*epsC/
    (h1N11*h2N11*h3N11*d21*d31*m1*m3);
```

```
u0N1N1 = -T(x,PT(x,y,PT(x,y,z,9),6),PT(x,y,z,9))*(h1N1N1+h2N1N1)*epsC/
    (h1N1N1*h2N1N1*h3N1N1*d2N1*d3N1*m2*m3);
u100 = T(PT(x,y,z,1),y,z)*((-rCx+rx)/(h1*h3)+(m1+m2)*(d10+d20)*epsC/
    (h100*h300*d10*d20*d30*m1*m2*m3));
u101 = -T(PT(x,y,PT(x,y,z,7),1),y,PT(x,y,z,7))*(d11+d21)*epsC/
    (h101*h301*d11*d21*d31*m1*m3);
u10N1 = -T(PT(x,y,PT(x,y,z,9),1),y,PT(x,y,z,9))*(d1N1+d2N1)*epsC/
    (h10N1*h30N1*d1N1*d2N1*d3N1*m2*m3);
u110 = -T(PT(x,PT(x,y,z,4),z,1),PT(x,y,z,4),z)*(m1+m2)*epsC/
    (h110*h310*d10*d30*m1*m2*m3);
u111 = T(PT(x,PT(x,y,PT(x,y,z,7),4),PT(x,y,z,7),1),
    PT(x,y,PT(x,y,z,7),4),PT(x,y,z,7))*epsC/(h111*h311*d11*d31*m1*m3);
u11N1 = T(PT(x,PT(x,y,PT(x,y,z,9),4),PT(x,y,z,9),1),
    PT(x,y,PT(x,y,z,9),4),PT(x,y,z,9))*epsC/(h11N1*h31N1*d1N1*d3N1*m2*m3);
u1N10 = -T(PT(x,PT(x,y,z,6),z,1),PT(x,y,z,6),z)*(m1+m2)*epsC/
    (h1N10*h3N10*d20*d30*m1*m2*m3);
u1N11 = T(PT(x,PT(x,y,PT(x,y,z,7),6),PT(x,y,z,7),1),PT(x,y,PT(x,y,z,7),6),
    PT(x,y,z,7))*epsC/(h1N11*h3N11*d21*d31*m1*m3);
u1N1N1 = T(PT(x,PT(x,y,PT(x,y,z,9),6),PT(x,y,z,9),1),PT(x,y,PT(x,y,z,9),6),
    PT(x,y,z,9))*epsC/(h1N1N1*h3N1N1*d2N1*d3N1*m2*m3);
uN100 = T(PT(x,y,z,3),y,z)*((-rCx+rx)/(h2*h3)+(m1+m2)*(d10+d20)
    *epsC/(h200*h300*d10*d20*d30*m1*m2*m3));
uN101 = -T(PT(x,y,PT(x,y,z,7),3),y,PT(x,y,z,7))*(d11+d21)
    *epsC/(h201*h301*d11*d21*d31*m1*m3);
uN10N1 = -T(PT(x,y,PT(x,y,z,9),3),y,PT(x,y,z,9))*(d1N1+d2N1)
    *epsC/(h20N1*h30N1*d1N1*d2N1*d3N1*m2*m3);
uN110 = -T(PT(x,PT(x,y,z,4),z,3),PT(x,y,z,4),z)*(m1+m2)
    *epsC/(h210*h310*d10*d30*m1*m2*m3);
uN111 = T(PT(x,PT(x,y,PT(x,y,z,7),4),PT(x,y,z,7),3),PT(x,y,PT(x,y,z,7),4),
    PT(x,y,z,7))*epsC/(h211*h311*d11*d31*m1*m3);
uN11N1 = T(PT(x,PT(x,y,PT(x,y,z,9),4),PT(x,y,z,9),3),PT(x,y,PT(x,y,z,9),4),
    PT(x,y,z,9))*epsC/(h21N1*h31N1*d1N1*d3N1*m2*m3);
uN1N10 = -T(PT(x,PT(x,y,z,6),z,3),PT(x,y,z,6),z)*(m1+m2)
```

```
    *epsC/(h2N10*h3N10*d20*d30*m1*m2*m3);
uN1N11 = T(PT(x,PT(x,y,PT(x,y,z,7),6),PT(x,y,z,7),3),PT(x,y,PT(x,y,z,7),6),
    PT(x,y,z,7))*epsC/(h2N11*h3N11*d21*d31*m1*m3);
uN1N1N1 = T(PT(x,PT(x,y,PT(x,y,z,9),6),PT(x,y,z,9),3),PT(x,y,PT(x,y,z,9),6),
    PT(x,y,z,9))*epsC/(h2N1N1*h3N1N1*d2N1*d3N1*m2*m3);
a = u000+u001+u00N1 + u010+u011+u01N1 +
    u0N10+u0N11+u0N1N1 + u100+u101+u10N1 +
    u110+u111+u11N1 + u1N10+u1N11+u1N1N1 +
    uN100+uN101+uN10N1 + uN110+uN111+uN11N1 +
    uN1N10+uN1N11+uN1N1N1;
end % ADI_Adap_3_D_FirstKnown
```

# Bibliography

[1] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512, 1984.

[2] Jeremiah U. Brackbill and Jeff S. Saltzman. Adaptive zoning for singular problems in two dimensions. *Journal of Computational Physics*, 46(3):342–368, 1982.

[3] Jose E. Castillo. An adaptive direct variational grid generation method. *Computers Mathematics with Applications*, 21(5):57 – 64, 1991.

[4] Weizhong Dai. A new ADI scheme for solving three-dimensional parabolic differential equations. *Journal of Scientific Computing*, 12(4):361–369, 1997.

[5] John C. Dallon, Brittany Dalton, and Chelsea Malani. Understanding streaming in *dictyostelium discoideum*: theory versus experiments. *Bulletin of Mathematical Biology*, 73(7):1603–1626, 2011.

[6] Carl de Boor. Cadre: Cautious adaptive romberg extrapolation. In John Rice, editor, *Mathematical Software*, pages 430 – 438. Academic Press, 1971.

[7] Jim Douglas, Jr. and Henry H. Rachford, Jr. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American Mathematical Society*, 82:421–439, 1956.

[8] Graeme Fairweather. A note on a generalisation of a method of Douglas. *Mathematics of Computation*, 23:407–409, 1969.

[9] Pascale Gaudet, Jeffrey G. Williams, Petra Fey, and Rex L. Chisholm. An anatomy ontology to represent biological knowledge in dictyostelium discoideum. *BMC Genomics*, 9(130), 2008.

[10] Willem H. Hundsdorfer and Jan G. Verwer. Stability and convergence of the Peaceman-Rachford ADI method for initial-boundary value problems. *Mathematics of Computation*, 53(187):81–101, 1989.

[11] Ahsan Iqbal. Synovial joints. Available at `http://www.mananatomy.com/basic-anatomy/synovial-joints`, accessed July 2015.

[12] Barbro Kreiss and Heinz-Otto Kreiss. Numerical methods for singular perturbation problems. *SIAM Journal on Numerical Analysis*, 18(2):262–276, 1981.

[13] Tsun-Zee Mai and Leina Wu. Multi-layer grid refinement method. *International Journal of Engineering Science and Technology*, 4(7):3521–3530, 2012.

[14] George and Charles Merriam and Noah Webster. Chemotaxis. Available at `http://www.merriam-webster.com/dictionary/chemotaxis`, accessed July 2015.

[15] World of Microbiology and Immunology. Dictyostelium. Available at `http://www.encyclopedia.com/doc/1G2-3409800167.html`, accessed July 2015.

[16] Joseph Oliger. Approximate methods for atmospheric and oceanographic circulation problems. In *Computing methods in applied sciences and engineering (Proc. Third Internat. Sympos., Versailles, 1977), II*, volume 91 of *Lecture Notes in Phys.*, pages 171–184. Springer, Berlin-New York, 1979.

[17] Donald W. Peaceman and Henry H. Rachford, Jr. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society for Industrial and Applied Mathematics*, 3:28–41, 1955.

[18] Miguel Pineda, Cornelius J. Weijer, and Raluca Eftimie. Modelling cell movement, cell differentiation, cell sorting and proportion regulation in dictyostelium discoideum aggregations. *Journal of Theoretical Biology*, 370:135 – 150, 2014.

[19] Laura Portero and Juan Carlos Jorge. A generalization of peaceman-rachford fractional step method. *Journal of Computational and Applied Mathematics*, 189(12):676 – 688, 2006. Proceedings of The 11th International Congress on Computational and Applied Mathematics The 11th International Congress on Computational and Applied Mathematics.

[20] Luciano Rezzola. Numerical methods for the solution of partial differential equations. Lecture notes for the COMPSTAR School on Computational Astrophysics, 8-13/02/10, Caen, France, available at `http://www.aei.mpg.de/~rezzolla/lnotes/Evolution_Pdes/evolution_pdes_lnotes.pdf`, accessed September 2015.

[21] Pauline Schaap. Evolutionary crossroads in developmental biology: Dictyostelium discoideum. *Development*, 138:387–396, 2011.

[22] L. F. Shampine and M. K. Gordon. *Computer solution of ordinary differential equations*. W. H. Freeman and Co., San Francisco, Calif., 1975.

[23] Stanly Steinberg and Patrick J. Roache. Variational grid generation. *Numerical Methods for Partial Differential Equations. An International Journal*, 2(1):71–96, 1986.

[24] John C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations, Second Edition*. Society for Industrial and Applied Mathematics, 2004.

[25] Marc Tessier-Lavigne and Corey S. Goodman. The molecular biology of axon guidance. *Science*, 274:1123–1133, 1996.

[26] Jules Thibault. Comparison of nine three-dimensional numerical methods for the solution of the heat diffusion equation. *Numerical Heat Transfer*, 8:281–298, 1985.

[27] Douwe Veltman. Phase constrast image of a large field of aggregating dictyostelium cells. Available at `http://www.formedium.com/us/products/dictyostelium-discoideum.html?____store=us`, accessed October 2015.

[28] Nikita Warner. *Transcriptional Regulation of the Glycogen Phosphorylase-2 Gene in Dictyostelium discoideum*. PhD thesis, Virginia Polytechnic Institute and State University, 1999.

[29] Ruud A.J. Warringa, Hein J.J. Mengelers, Jan A.M. Raaijmakers, Piet L.B. Bruijnzeel, and Leo Koenderman. Upregulation of formyl-peptide and interleukin-8-induced eosinophil chemotaxis in patients with allergic asthma. *Journal of Allergy and Clinical Immunology*, 91(6):1198 – 1205, 1993.

[30] Olof B. Widlund. On the effects of scaling of the Peaceman-Rachford method. *Mathematics of Computation*, 25:33–41, 1971.

[31] Winifred Elizabeth Williams. *A structural model of heat transfer due to blood vessels in living tissue*. PhD thesis, The University of Arizona, 1990.