University of California
Santa Barbara

A Level-Set Approach for Simulating Dendritic Crystal Growth

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

Master of Science
in
Mechanical Engineering

by

Megan Maria Chang

Committee in charge:

Professor Frederic Gibou, Chair
Professor Paolo Luzzatto-Fegiz
Professor Jeffrey Moehlis

June 2017

The Thesis of Megan Maria Chang is approved.

Professor Paolo Luzzatto-Fegiz

Professor Jeffrey Moehlis

Professor Frederic Gibou, Committee Chair

March 2017

A Level-Set Approach for Simulating Dendritic Crystal Growth

by

Megan Maria Chang

Acknowledgements

First and foremost, I offer my sincerest gratitude to my graduate research advisor, Dr. Frederic Gibou, Professor of Mechanical Engineering UCSB, for his unwavering encouragement and guidance. I thank Raphael Egan for his patience and for taking me under his computational wing. I thank my parents for their wise counsel and sympathetic ear. And I thank my friends who kept me sane throughout the crazy ride that is graduate school.

Abstract

A Level-Set Approach for Simulating Dendritic Crystal Growth

by

Megan Maria Chang

In this thesis, we consider the piecewise constant coefficient Stefan problem, a free boundary problem described by a partial differential equation with an unknown concentration u and an unknown time-dependent irregular domain Ω^-, used primarily to study phase transitions. We present a numerical method for solving the two-dimensional, unsteady, two-phase, diffusion equation on an irregular domain with Dirichlet boundary conditions at the solidification front. Several techniques were implemented to achieve this including: the implicit level-set method to update the location of the interface and keep track of the two phases it separates; the Ghost-Fluid method to impose boundary conditions on an irregular domain and allow for symmetric discretization of our diffusion matrix; a third-order extrapolation method to allow for both accurate interface velocity calculations and implicit discretization by providing valid values at grid points that may be contained in Ω^- in the next time step; a combination of WENO spatial discretization and TVD RK3 time discretization to achieve third-order accurate advection; and finally for diffusion, we implemented the Crank-Nicholson method to achieve second-order accuracy in both space and time with implicit time stepping. Overall, for the Stefan problem, we demonstrate that through robust and computationally efficient methods, it is possible to simulate complex dendritic crystal growth.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Stefan problem is a free boundary problem that describes the motion of a front driven by diffusion, making it highly useful for analyzing phase transitions. For example, the Stefan problem is used in thermodynamics to study crystal growth in a supercooled solution; in materials science to study the mixture of molten metals and elements to create alloys with superior properties; and in biomedicine to study cancer invasion and tumor growth. Its plethora of applications concerning diffusion-dominated phenomena have made it an important model to study.

The governing equations of the Stefan problem are: diffusion, which is the driving force of our phase transition; the Gibbs-Thomson boundary condition, which imposes anisotropic behavior that leads to dendritic growth; and the interface velocity calculation used to move our level-set and update the interface location through advection. These three equations are given by

$$\begin{aligned} \text{Diffusion:} \quad & \frac{\partial u}{\partial t} = D\Delta u, \\ \text{Gibbs-Thomson BC:} \quad & u_\gamma = u + \epsilon\kappa, \\ \text{Interface Velocity:} \quad & \vec{v} = [D\nabla u]_\gamma, \end{aligned} \tag{1.1}$$

where Δ is the Laplace operator (a second-order differential operator), and ∇ is the gradient (a first-order differential operator), u is the concentration, u_γ is the concentration at the front, t is real time, D is the diffusion coefficient, ϵ is the anisotropy strength, κ is the curvature, $\vec{v}$ is the interface velocity, and $[\cdot]_\gamma$ denotes a jump across the interface.

The most common example of dendritic crystal growth is the snowflake, where the dendrites refer to the branches of the tree-like structure. Since there are no existing analytical methods for solving and predicting dendritic crystal growth, we rely on numerical methods to tackle these complicated problems. A successful method should be able to track a moving solid-liquid interface undergoing complex topological changes, and must be computationally efficient since these methods often require high grid resolutions, in order to capture these dendrites, and strict time step restrictions to ensure stability.

Several techniques are used in practice to track a time-evolving boundary, or interface. These methods can be described as either explicit tracking or implicit capturing. Explicit methods such as front tracking are valued for their accuracy, but fall short when it comes to handling topological changes such as materials merging or separating. In the case where a material melts and develops holes, or else experiences crystal growth such that dendrites begin to merge, additional numerical treatment is required. One would need to develop an algorithm that could detect the moment objects merged or separated and construct new parameterizations to describe each newly separated or joined shape, a very challenging task.

Alternatively, implicit methods such as the level-set method represent the interface as an isocontour of a Lipschitz continuous function. This allows for straightforward handling of topological changes, since objects do not have to be parametrized. As we move from two-dimensional objects to three-dimensional objects, this becomes increasingly important. The main drawback of the level-set method is that it is less accurate

in terms of mass conservation. However in [2], Min and Gibou address this problem with adaptive grids and demonstrate its ability to alleviate this problem of mass loss or gain.

Since we are interested in developing a method which can simulate complicated dendritic crystal growth, we employ the level-set method paired with the Ghost-Fluid method to achieve a superior interface capturing scheme.

The level-set advection equation

$$\phi_t + \vec{v} \cdot \nabla \phi = 0 \tag{1.2}$$

is a PDE which uses a higher dimensional level-set function ϕ to describe the motion of a co-dimension 1 shape, under a velocity field $\vec{v}$. In this thesis, we use a three-dimensional level-set function ϕ to keep track of a two-dimensional shape, where the zero-contour represents the interface separating two materials.

Henceforth, in a domain Ω, we can refer to the region where $\phi < 0$ as reacted material, or the subdomain Ω^-; and refer to the region where $\phi > 0$ as unreacted material, or the subdomain Ω^+. These two substances are separated by the interface, where $\phi = 0$, across which unreacted material can be converted into reacted material or vice versa (i.e. ice in water begins to melt into more water, or supercooled water surrounding ice begins to freeze into more ice). The labels reacted vs. unreacted hold no physical implication, they are simply a nominal tool used to distinguish whether we are in the negative ϕ subdomain (i.e. Ω^-) or the positive ϕ subdomain (i.e. Ω^+). That is to say the reacted material could refer to either ice or water, as you prefer.

This is an important fact to keep in mind because throughout the thesis, we will discuss how to solve only in Ω^-. However, this **does not** mean we are ignoring the material in Ω^+. On the contrary, both subdomains are of equal importance in simulating

crystal growth. We simply ignore Ω^+, while we solve in Ω^-, then ignore Ω^- while we solve in Ω^+ using the exact same methods. Decoupling the two solutions is a useful simplification we can make since we are given Dirichlet boundary conditions. Thus, to solve for diffusion in each subdomain, our governing equation becomes

$$\begin{aligned} \text{Diffusion in } \Omega^-: \quad & \frac{\partial u_{\text{in}}}{\partial t} = D_{\text{in}} \ \Delta u_{\text{in}} \ , \\ \text{Diffusion in } \Omega^+: \quad & \frac{\partial u_{\text{out}}}{\partial t} = D_{\text{out}} \Delta u_{\text{out}}, \end{aligned} \tag{1.3}$$

where D_{in} and u_{in} represent values in Ω^-, while D_{out} and u_{out} represent values in Ω^+. And the governing interface velocity calculation becomes

$$\vec{v} = D_{\text{in}} \nabla u_{\text{in}} - D_{\text{out}} \nabla u_{\text{out}} \tag{1.4}$$

to appropriately represent the meeting of the two subdomains at the interface.

To build the most efficient solver requires **symmetric** matrix discretization. This symmetry allows us to use fast matrix inversion methods like the Preconditioned Conjugate Gradient (PCG) method. Since we build a diffusion matrix that is symmetric and positive definite, the Matlab backslash operator can select the Choleski triangular solver to precondition our matrix.

Previous methods, for analyzing the Stefan problem, sacrificed this symmetry to achieve second-order accuracy by implementing higher-order extrapolation to compute ghost values. However, in [1], Gibou and Fedkiw show that using only linear and constant extrapolation to calculate ghost values, rather than quadratic leads to a symmetric discretization that is still second-order accurate in the solution of u, but loses second-order accuracy in the gradient (i.e. ∇u). This means our Stefan solver would become first-order accurate due to the gradient calculations in the interface velocity

computation. However, we choose computational efficiency over second-order accuracy since this non-symmetric discretization becomes computationally prohibitive at higher grid resolutions. Thus, we implement ghost values using linear extrapolation.

Our computational grid is made up of cells of width Δx and height Δy. The cell centers are known as grid points or grid nodes with the ith grid node located at x_i and the jth grid node located at y_j. The value of interest at each cell is concentration, denoted as $u_{i,j}$ at some location (x_i, y_j). Further, we use the superscript n to describe our time at some t^n, such that the initial conditions would be described as $u^0_{i,j}$.

In order to capture dendritic growth, these Stefan problems often require very fine grid resolutions. Since our time step restriction is dependent on the spatial grid size, we want to minimize this dependency by using **implicit** time stepping in our diffusion discretization to allow our time step Δt to be proportional to Δx, as opposed to Δt being proportional to Δx^2, as is the case with explicit time stepping. This way, as we refine our grid from a resolution of 100×100 to 200×200, the time step will shrink by a factor of $\frac{1}{2}$ instead of $\frac{1}{4}$, and as we move from a resolution of 100×100 to 400×400, the time step will shrink by a factor of $\frac{1}{4}$ instead of $\frac{1}{16}$. It is important to note that the implicit level-set method enables us to use implicit discretization, here, as well. Hence, we employ the Crank-Nicholson scheme so that Δt can be equal to Δx. This means our time step will be limited by advection where we will implement a Courant-Friedrichs-Lewy (CFL) number of 0.5, such that

$$\Delta t = \frac{0.5\min(\Delta x, \Delta y)}{|v|_{max}}, \tag{1.5}$$

where Δt represents a real physical time step used in the diffusion and advection steps, and $|v|_{max}$ is the maximum interface velocity. This Δt must be recalculated at every time step since $|v|_{max}$ will be constantly changing.

This implicit Crank-Nicholson scheme may require the use of extrapolated values. Consider the case where between time steps t^n and t^{n+1}, the interface moves so that a point $u_{i,j}$ contained in Ω^+ is now contained in Ω^-. In the Crank-Nicholson scheme, we would need a valid value at that point for $u_{i,j}^n$ and $u_{i,j}^{n+1}$. However, this value for $u_{i,j}^n$ would not exist, since it was not contained in Ω^- at t^n, and as we said before, the solutions in each subdomain are decoupled. For this reason, we need to employ a third-order extrapolation method to extrapolate values of u^n in the layer of grid nodes near the interface.

This third-order extrapolation method is also necessary for computing gradient calculations at the interface to determine the interface velocity (1.1c). Gradient calculations in two-dimensions are given by

$$\nabla u = \Big(u_x, u_y \Big) = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right), \tag{1.6}$$

where ∇ is the symbolic representation of the gradient field. In derivative notation, the subscript denotes the partial differential of a function u with respect to either x or y. This is not to be confused with our $u_{i,j}$ subscript notation to describe spatial location.

For this extrapolation method, as well as for the constant extrapolation and reinitialization methods, a new fictitious time step restriction must be satisfied to ensure numerical stability. This fictitious time step,

$$\Delta\tau = 0.5\text{min}(\Delta x, \Delta y), \tag{1.7}$$

is used to reach a steady state solution in our iterative methods. This will become more clear as we discuss each method in the later sections. However, we mention this, now, because it is important to keep in mind that there are two separate time step

restrictions used in this paper that are not to be confused.

Throughout this thesis, we analyze errors using the standard mathematical *p-norm*

$$||\vec{x}||_p = \left(\sum_{k=1}^{N} |x_k|^p\right)^{\frac{1}{p}}, \tag{1.8}$$

where the scalar x_k refers to the error $u_{\text{exact}} - u_{\text{numerical}}$ at every point in the domain. To be thorough, we consider the L^1 norm and the L^∞ norm, where L^1 is the sum of the absolute values of error multiplied by the grid size area, and L^∞ returns the maximum absolute value of error. Thus, L^1 gives us a good sense of the overall accuracy of our solvers across the entire domain, and L^∞ warns us if there is any one point in the domain that is giving rise to more error such as, for instance, at the front. When we plot these errors against the grid size on a log-log plot, the slope tells us the order of accuracy of our scheme, where the order quantifies the rate of convergence of our numerical approximation to the exact solution. If a method is said to be second-order accurate, one would expect the error to reduce by a factor of 4 if the grid resolution increased from 100×100 to 200×200, and reduce by a factor of 16 as we move from a resolution of 100×100 to 400×400. To provide this data in a more concise form, rather than plots, we can use

$$\text{Order } = \log_2\left(\frac{L^1 \text{ error using } \Delta x}{L^1 \text{ error using } \frac{\Delta x}{2}}\right) \tag{1.9}$$

to calculate the order of accuracy, where the base of $\log_2$ comes from the fact that the grid resolutions double each time we refine the grid. To be specific and consistent, we use grid resolutions 81×81, 161×161, and 321×321 for all of our accuracy testing.

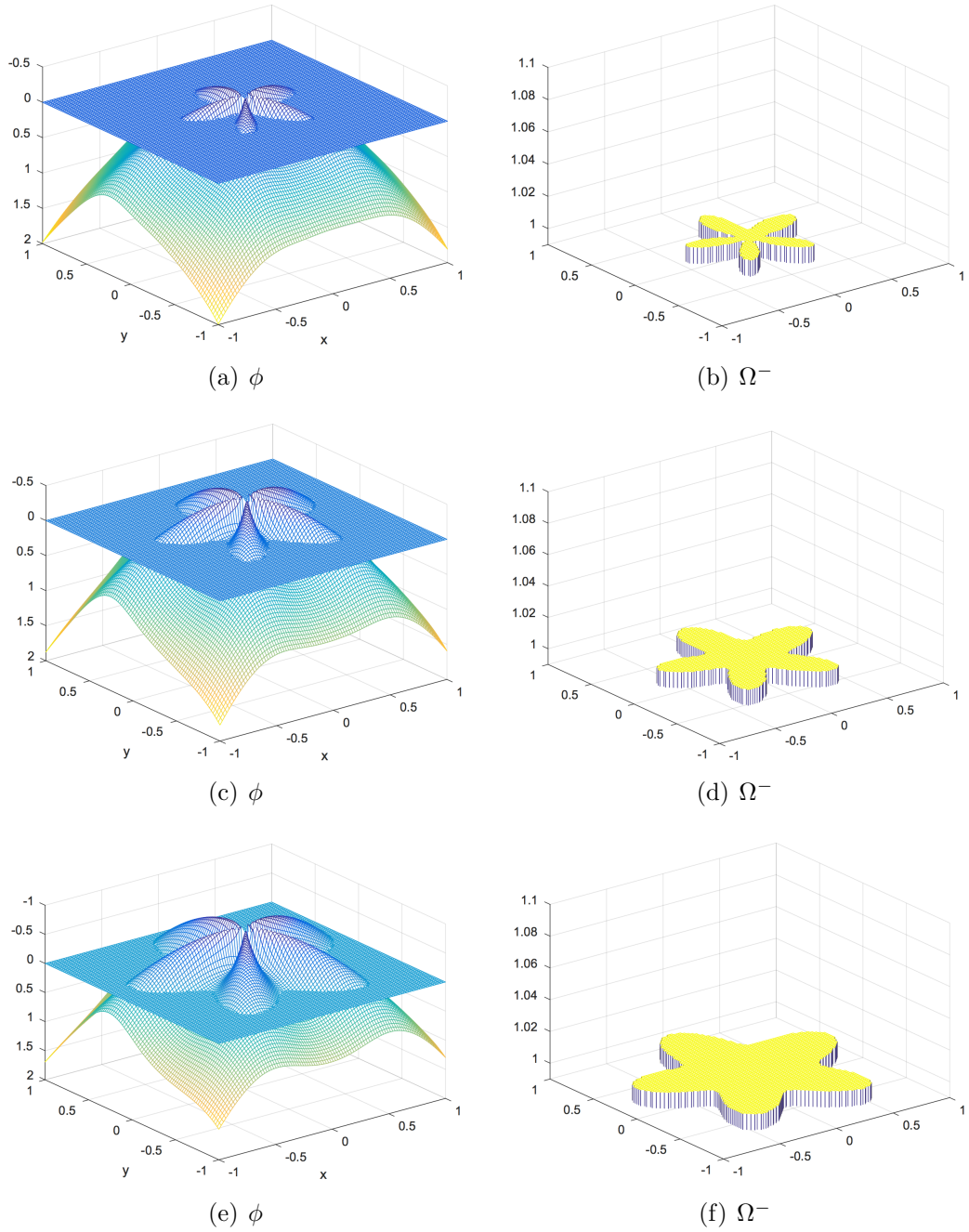
Chapter 2

The Level-Set Method

In Figure 2.1, we demonstrate how the three-dimensional level-set allows us to easily keep track of the interface separating reacted and unreacted material. Notice, we have inverted the z- axis so that ϕ is positive downwards, for better visual representation. Following this, we included Figure 2.2 to demonstrate the real power of the level-set method: its ability to handle complex topological changes such as melting into separate pieces or conversely, pieces growing and merging together.

2.1 The Reinitialization Equation

The reinitialization scheme, which transforms an arbitrary level-set function into a signed distance function, has been proven to produce more robust numerical results, improve mass conservation, and improve proficiency of geometrical computations such as calculating interface curvatures. When we reinitialize our level-set, we effectively smooth out the gradient and remove any numerical noise that may build up as a result of our moving boundary. Even a small amount of noise will be greatly amplified when calculating the level-set gradients for our normal vectors. This was one of the

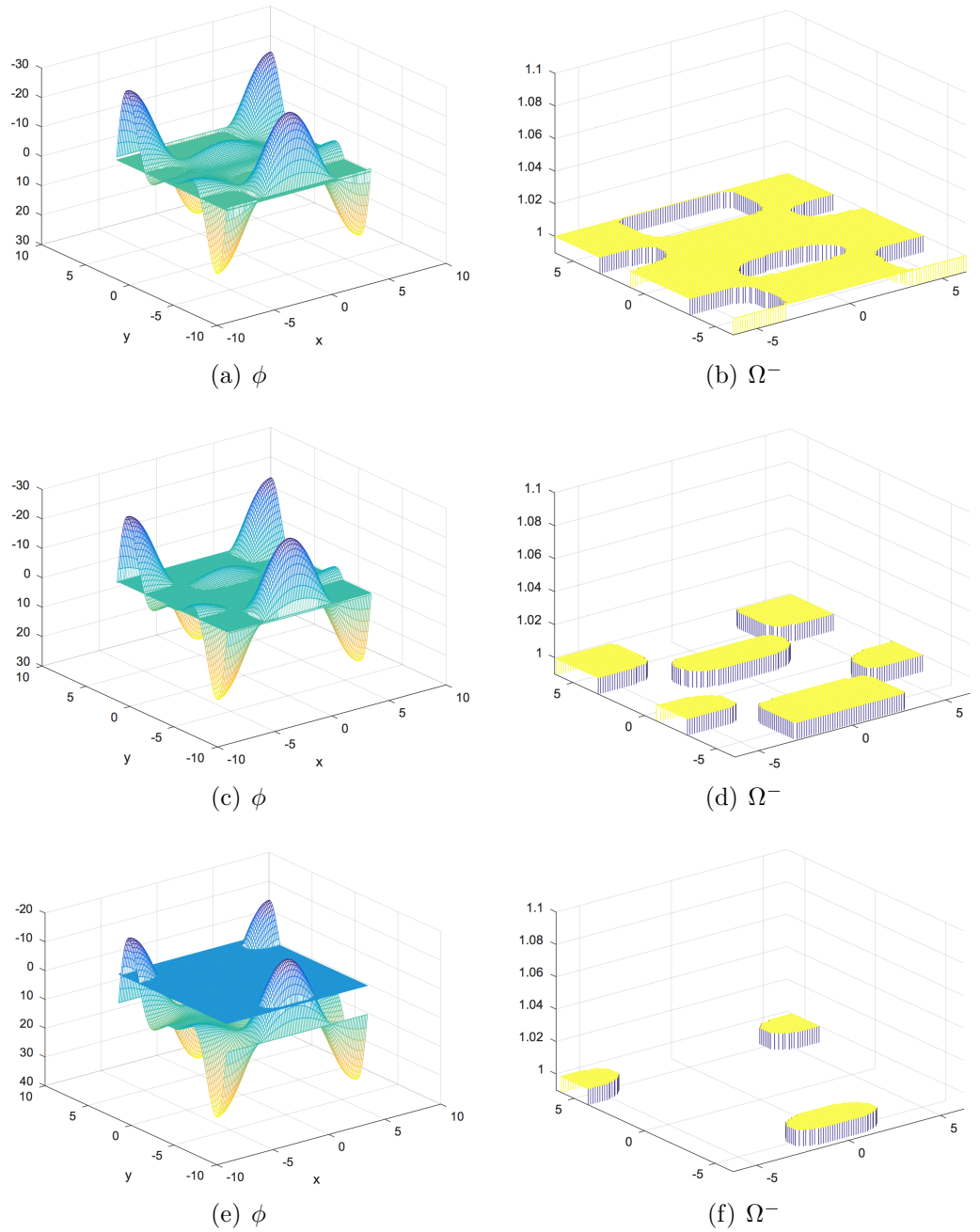


(a) ϕ (b) Ω^-

(c) ϕ (d) Ω^-

(e) ϕ (f) Ω^-

Figure 2.1: In this example, we use the level-set method to model a flower shape growing uniformly in time. One can see how the negative ϕ values in the three-dimensional object on the left create our two-dimensional shape of reacted material on the right, bounded by a border where $\phi = 0$. (The z-axis is inverted so that the positive axis points downward.)



(a) ϕ (b) Ω^-

(c) ϕ (d) Ω^-

(e) ϕ (f) Ω^-

Figure 2.2: In this example, we demonstrate a more complex problem in which reacted material shrinks and separates into individual segments of reacted material. Take for instance a glacier melting and separating into smaller chunks of ice. (The z-axis is inverted so that the positive axis points downward.)

reasons the level-set method previously had poor mass conservation properties. With the reinitialization method, we greatly reduce this problem.

The reinitialization equation reads

$$\phi_\tau + S(\phi^0)(|\nabla\phi| - 1) = 0 \tag{2.1}$$

proposed by Sussman, Smereka and Osher in [4], where $S(\phi^0)$ is a smoothed out sign function, and τ is a fictitious time which allows us to find a steady state solution for ϕ with a smooth gradient of magnitude 1, everywhere, while maintaining the location of the zero level-set. In semi-discretized form, this equation can be represented as

$$\frac{\partial\phi}{\partial\tau} + \mathrm{sgn}(\phi^0)[H_G(D_x^+\phi, D_x^-\phi, D_y^+\phi, D_y^-\phi) - 1] = 0, \tag{2.2}$$

where ϕ^0 is the initial level-set at time $\tau = 0$ and $\mathrm{sgn}(\phi^0)$ will have a value of either -1 or +1, as determined by the sign of ϕ^0. Finally, H_G is the Godunov Hamiltonian defined as

$$H_G(a,b,c,d) = \begin{cases} \sqrt{\max(|a^+|^2, |b^-|^2) + \max(|c^+|^2, |d^-|^2)} & \text{if } \mathrm{sgn}(\phi^0) < 0, \\ \sqrt{\max(|a^-|^2, |b^+|^2) + \max(|c^-|^2, |d^+|^2)} & \text{if } \mathrm{sgn}(\phi^0) > 0, \end{cases} \tag{2.3}$$

where $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$. Further, from (2.2), we know that a, b, c, and d correspond to the directional derivatives $D_x^+\phi, D_x^-\phi, D_y^+\phi$, and $D_y^-\phi$ computed

using first-order accurate one-sided finite differences

$$\begin{aligned} D_x^+\phi_{i,j} &= \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x}, \\ D_x^-\phi_{i,j} &= \frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x}, \\ D_y^+\phi_{i,j} &= \frac{\phi_{i,j+1} - \phi_{i,j}}{\Delta y}, \\ D_y^-\phi_{i,j} &= \frac{\phi_{i,j} - \phi_{i,j-1}}{\Delta y}, \end{aligned} \tag{2.4}$$

where in derivative notation, the subscript denotes a partial derivative with respect to either x or y, and the superscript denotes the upwind direction (either + or -) to indicate whether the points i and $i+1$ are involved, or i and $i-1$, respectively.

Next, we evolve our solution in time using the second-order accurate Total Variation Diminishing Runge-Kutta (TVD RK2) scheme which uses two Euler steps to develop a temporary $\tilde{\phi}^{n+2}$ value

$$\begin{aligned} &\frac{\tilde{\phi}^{n+1} - \phi^n}{\Delta\tau} + \operatorname{sgn}(\phi^0)[H_G(D_x^+\phi^n, D_x^-\phi^n, D_y^+\phi^n, D_y^-\phi^n) - 1] = 0, \\ &\frac{\tilde{\phi}^{n+2} - \tilde{\phi}^{n+1}}{\Delta\tau} + \operatorname{sgn}(\phi^0)[H_G(D_x^+\tilde{\phi}^{n+1}, D_x^-\tilde{\phi}^{n+1}, D_y^+\tilde{\phi}^{n+1}, D_y^-\tilde{\phi}^{n+1}) - 1] = 0, \end{aligned} \tag{2.5}$$

and one averaging step to achieve a second-order accurate value for ϕ^{n+1}

$$\phi^{n+1} = \frac{\phi^n + \tilde{\phi}^{n+2}}{2}. \tag{2.6}$$

Again, τ is a fictitious time which allows us to evolve our solution until steady state is reached by reinserting ϕ^{n+1} into the above equations as our new ϕ^n. For stability purposes, we select a $\Delta\tau = 0.5\ \min(\Delta x, \Delta y)$ for our fictitious time step.

Notice, however, that in (2.5), despite the time step n, we continue to use the sign of the initial level-set (i.e. $\operatorname{sgn}(\phi^0)$). This is because we do not want the interface moving

during reinitialization. That is to say, the sign of $\phi_{i,j}$ should be consistent throughout this process in order to preserve area and uphold conservation of mass.

Also to comply with mass conservation, extra care must be taken at the points neighboring the interface [4]. More often than not, the interface will not fall exactly on a grid point $\phi_{i,j}$. So instead of being represented by a zero value, the interface will be located between two grid nodes that are **changing sign**, such that $\phi_{i,j} \cdot \phi_{i-1,j} < 0$. For these points, it is necessary to calculate the value θ which represents the fractional distance between the interface and the neighboring grid points. We calculate these θ values before entering our iterative solver since this distance θ should remain constant throughout the reinitialization process. These θ values can then be used to maintain the zero level-set and ensure the location of our interface does not change during reinitialization. So, assuming that our point of interest $\phi_{i,j} < 0$, we would then calculate θ as

$$\theta_{i,j}^{L} = \frac{\phi_{i,j}}{\phi_{i,j} - \phi_{i-1,j}} \qquad \text{if } \phi_{i,j} \cdot \phi_{i-1,j} < 0, \tag{2.7}$$

where the superscript L indicates that the interface passes to the left of the point $\phi_{i,j}$. Note, for each grid point, there can be up to four values of θ associated to it since it is possible for the interface to pass a point on more than one side. For instance, if the interface passed to the left and above it, our point $\phi_{i,j}$ would have a $\theta_{i,j}^{L}$ and $\theta_{i,j}^{A}$ associated to it. The other three possible θ values for a point $\phi_{i,j}$ are calculated as

$$\begin{aligned}
\theta_{i,j}^{R} &= \frac{\phi_{i,j}}{\phi_{i,j} - \phi_{i+1,j}} \qquad \text{if } \phi_{i,j} \cdot \phi_{i+1,j} < 0, \\
\theta_{i,j}^{B} &= \frac{\phi_{i,j}}{\phi_{i,j} - \phi_{i,j-1}} \qquad \text{if } \phi_{i,j} \cdot \phi_{i,j-1} < 0, \\
\theta_{i,j}^{A} &= \frac{\phi_{i,j}}{\phi_{i,j} - \phi_{i,j+1}} \qquad \text{if } \phi_{i,j} \cdot \phi_{i,j+1} < 0,
\end{aligned} \tag{2.8}$$

where θ must always be positive. These θ values are then used to recompute the

directional derivatives at that point $\phi_{i,j}$

$$\begin{aligned} D_x^+ \phi_{i,j} &= \frac{\phi_{I,j} - \phi_{i,j}}{\theta_{i,j}^R \Delta x}, \\ D_x^- \phi_{i,j} &= \frac{\phi_{i,j} - \phi_{I,j}}{\theta_{i,j}^L \Delta x}, \end{aligned} \tag{2.9}$$

where $\phi_{I,j}$ is the value of ϕ at the interface, and since we know this to be zero, these equations can simply be written as

$$\begin{aligned} D_x^+ \phi_{i,j} &= \frac{-\phi_{i,j}}{\theta_{i,j}^R \Delta x}, \\ D_x^- \phi_{i,j} &= \frac{\phi_{i,j}}{\theta_{i,j}^L \Delta x}, \end{aligned} \tag{2.10}$$

with similar construction for the directional derivatives with respect to y:

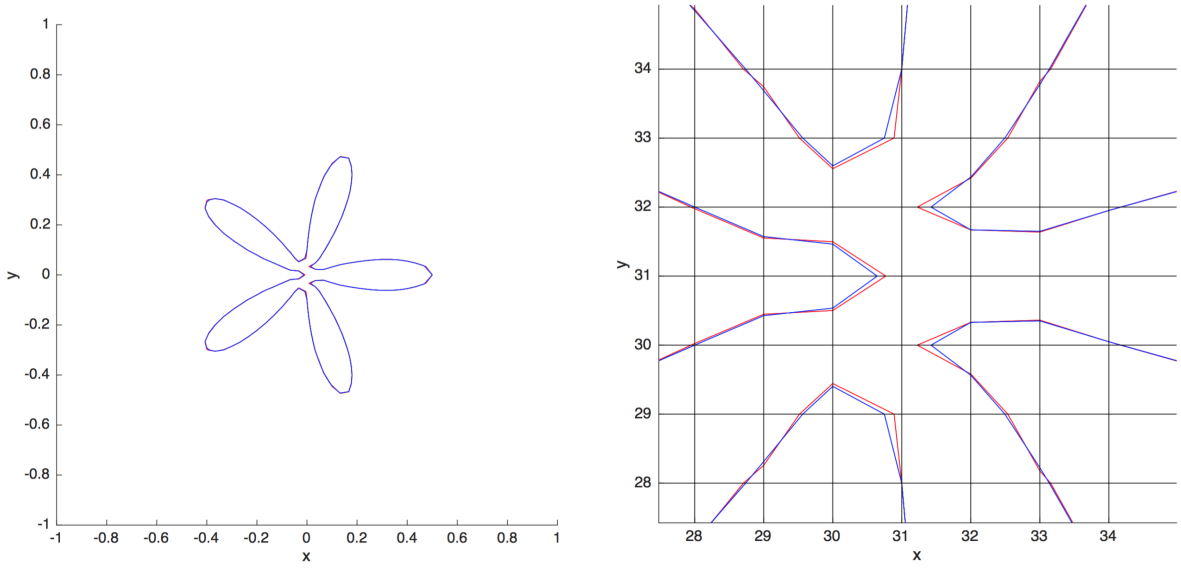
$$\begin{aligned} D_y^+ \phi_{i,j} &= \frac{-\phi_{i,j}}{\theta_{i,j}^A \Delta y}, \\ D_y^- \phi_{i,j} &= \frac{\phi_{i,j}}{\theta_{i,j}^B \Delta y}. \end{aligned} \tag{2.11}$$

For these points $\phi_{i,j}$ that have at least one associated θ value, a new time step stability restriction must be satisfied:

$$\Delta\tau = 0.5\min(\theta^R \Delta x, \theta^L \Delta x, \theta^A \Delta y, \theta^B \Delta y). \tag{2.12}$$

Also, notice that θ is in the denominator of our one-sided finite difference calculations. This means small values of θ can cause numerical instabilities. To avoid producing non-finite data, we say that if θ is too small ($\theta < \Delta x$), we can assume the point is very close to the interface, and set $\phi_{i,j} = 0$, reassigning this point to the interface. This small perturbation in the location of the zero level-set (also known as artificial boundary

perturbation) has a negligible impact on the accuracy of our solver, and guarantees numerical stability. Figure 2.3 shows how effectively the zero level-set is maintained on a complex geometry with sharp kinks. And Figure 2.4 best demonstrates visually what

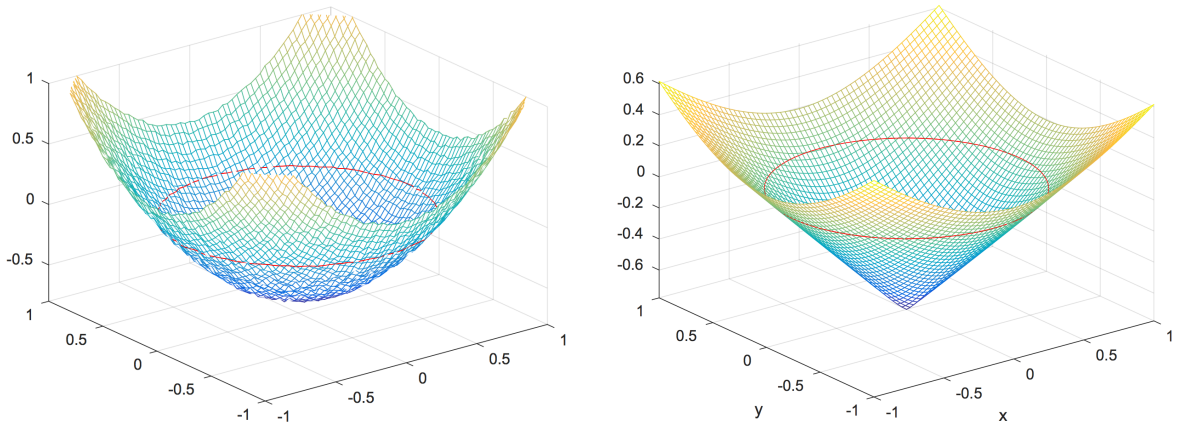


(a) Zero level-set contour

(b) Zoomed in

Figure 2.3: The zero level-set is effectively maintained throughout the reinitizialization scheme. The red line shows the original zero level-set location, while the blue is the new zero level-set location after the function has been reinitialized. The goal is for the blue to completely overlap the red one. In (b), we have zoomed into an area with sharp kinks and where some points have more than one θ value. This is where problems would arise, but our scheme is able to maintain ϕ_0^0 nicely.

is achieved via the reinitialization method, where one will notice our original level-set shaped as a bowl is reinitialized into a cone-like shape. Also notice that the initial level-set is very noisy (i.e. the data does not appear smooth). This noise is removed via reinitialization so that it does not create large numerical errors when calculating $\nabla\phi$. It is important to note that beyond interface tracking, ϕ has no relevant meaning. This means we care only about the zero level-set's position, and the sign of ϕ which tells us which material region we are in (either Ω^- or Ω^+). The actual magnitude of ϕ, however, is irrelevant.



(a) Before (b) After

Figure 2.4: The reinitialization scheme will take a noisy level-set and return as an output, a level-set with a smooth gradient of magnitude 1, everywhere. This makes them produce more robust numerical results, especially when computing normal vectors which rely on ϕ gradients. The red isocontour line represents the zero level-set, which has not moved during the reinitialization process.

Chapter 3

Diffusion

Diffusion describes the net movement of particles from a region of high concentration to a region of lower concentration due to molecular Brownian motion. For example, consider a drop of colored ink in a glass of water. Assume we are dealing with an ink droplet with the same density as water that has been placed in a state of suspension, completely undisturbed by convection or gravitational effects. With time, the ink would naturally spread, so that eventually the cup of water would be uniformly colored with uniform distribution of the ink concentration. This natural phenomenon of homogenization is described by

$$\frac{\partial u}{\partial t} = \nabla \cdot (D \nabla u) + S, \tag{3.1}$$

where u is the concentration, D is the diffusion coefficient, and S is the source term. We use the implicit Crank-Nicholson scheme in semi-discretized form

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}\nabla \cdot (D \nabla u^{n+1}) + \frac{1}{2}\nabla \cdot (D \nabla u^n) + \frac{1}{2}(S^{n+1} + S^n), \tag{3.2}$$

because it allows us to achieve second-order accuracy in space and time using $\Delta t \sim \Delta x$. This way, as we refine our grid, our time step gets proportionally smaller, rather than

quadratically smaller. Further, we know our time step will be limited by advection, rather than diffusion to ensure stability. In discretized form, for a two-dimensional problem with constant D and standard centered finite differences in space, we obtain

$$\begin{aligned}\frac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t} = \frac{1}{2}D\Bigg(&\frac{u_{i-1,j}^{n+1}-2u_{i,j}^{n+1}+u_{i+1,j}^{n+1}}{\Delta x^2}+\frac{u_{i,j-1}^{n+1}-2u_{i,j}^{n+1}+u_{i,j+1}^{n+1}}{\Delta y^2}\\ &+\frac{u_{i-1,j}^{n}-2u_{i,j}^{n}+u_{i+1,j}^{n}}{\Delta x^2}+\frac{u_{i,j-1}^{n}-2u_{i,j}^{n}+u_{i,j+1}^{n}}{\Delta y^2}\Bigg)+\frac{1}{2}(S_{i,j}^{n+1}+S_{i,j}^{n}).\end{aligned} \tag{3.3}$$

And if we rearrange our system of linear equations to resemble $\boldsymbol{A}\vec{u}^{\,n+1}=\vec{f}(\vec{u}^{\,n},BC^{n+1})$, so that all our knowns are on the RHS, we are left with

$$\begin{aligned}&\frac{u_{i,j}^{n+1}}{\Delta t}-\frac{1}{2}D\Bigg(\frac{u_{i-1,j}^{n+1}-2u_{i,j}^{n+1}+u_{i+1,j}^{n+1}}{\Delta x^2}+\frac{u_{i,j-1}^{n+1}-2u_{i,j}^{n+1}+u_{i,j+1}^{n+1}}{\Delta y^2}\Bigg)\\ &=\frac{u_{i,j}^{n}}{\Delta t}+\frac{1}{2}D\Bigg(\frac{u_{i-1,j}^{n}-2u_{i,j}^{n}+u_{i+1,j}^{n}}{\Delta x^2}+\frac{u_{i,j-1}^{n}-2u_{i,j}^{n}+u_{i,j+1}^{n}}{\Delta y^2}\Bigg)+\frac{1}{2}(S_{i,j}^{n+1}+S_{i,j}^{n}),\end{aligned} \tag{3.4}$$

where $\boldsymbol{A}$ is our diffusion matrix and $\vec{u}^{\,n+1}$ is our vector of unknowns at the next time step assuming that our source term does not depend on u.

3.1 Building an Efficient Sparse Matrix

Since our matrix will contain mostly zero elements, we can significantly minimize memory usage by creating a sparse matrix $\boldsymbol{A}$. So, rather than populate a full matrix with a few non-zero elements $A_{i,j}$, we simply keep track of the non-zero $\boldsymbol{A}$ element's value and location in three separate vectors: $\vec{A}, \vec{i}$, and $\vec{j}$. Then, we use Matlab's built-in sparse function to generate a sparse matrix. For example, the following matrix

$$A = \begin{bmatrix} 15 & 0 & 0 & 0 \\ 0 & 25 & 0 & 0 \\ 0 & 17 & 35 & 0 \\ 0 & 0 & 0 & 45 \end{bmatrix}$$

would instead be represented as

$$\begin{aligned} \vec{A} &= \begin{bmatrix} 15 & 25 & 17 & 35 & 45 \end{bmatrix}, \\ \vec{i} &= \begin{bmatrix} 1 & 2 & 3 & 3 & 4 \end{bmatrix}, \\ \vec{j} &= \begin{bmatrix} 1 & 2 & 2 & 3 & 4 \end{bmatrix}, \end{aligned}$$

where $\vec{A}$ contains the values of the non-zero elements, and $\vec{i}$ and $\vec{j}$ contain their row and column locations. While this may not appear useful for a 4×4 matrix, the sparse indexing method becomes highly advantageous as $\boldsymbol{A}$ grows. Take, for example, an identity matrix of size 1,000×1,000. In full matrix storage mode, this matrix requires 8 megabytes of memory. In sparse mode, it requires only 0.024 megabytes (that equates to a 99.7% reduction in memory space). Since each row of our matrix can have at most five non-zero elements regardless of the grid resolution, the sparse matrix is highly effective for our purposes.

This greatly reduces memory usage, but requires some additional preparation. In order to preallocate the size of our RHS vector, we count the number of points contained in Ω^-. To preallocate the size of our sparse vectors $\vec{A}, \vec{i}$, and $\vec{j}$, we must count all of the points in Ω^-, and additionally count the nupmber of neighbors also encompassed within Ω^- using the product test (e.g. $\phi_{i,j}^{n+1} \cdot \phi_{i+1,j}^{n+1} > 0$).

Now, we are ready to fill our sparse vector $\vec{A}$. We take the coefficients directly from the discretized diffusion equation (3.4):

$$\text{Above} = -\frac{1}{2}D\frac{\Delta t}{\Delta y^2},$$

$$\text{Left} = -\frac{1}{2}D\frac{\Delta t}{\Delta x^2}, \qquad \text{Center} = 1 + \frac{1}{2}D\left(\frac{2\Delta t}{\Delta x^2} + \frac{2\Delta t}{\Delta y^2}\right), \qquad \text{Right} = -\frac{1}{2}D\frac{\Delta t}{\Delta x^2},$$

$$\text{Below} = -\frac{1}{2}D\frac{\Delta t}{\Delta y^2}. \tag{3.5}$$

Each time we add a value to $\vec{A}$, we also add its position to our $\vec{i}$ and $\vec{j}$ vectors. For instance, for a point $\phi_{i,j}^{n+1}$ contained in Ω^-, we add five elements to each of our vectors ($\vec{A}$, $\vec{i}$, and $\vec{j}$) as shown in Table 3.1. To understand this table, we must introduce two new functions: the column-stacking function p and the tagging function.

Our system of linear equations $\boldsymbol{A}\vec{u}^{\,n+1} = \vec{f}\,(\vec{u}^{\,n}, BC^{n+1})$ requires u to be stored in vector form. Thus, we store our elements of $u_{i,j}^{n+1}$ in a vector $\vec{u}_p^{\,n+1}$, where

$$p = (j-1)m + i. \tag{3.6}$$

This function allows us to stack our $u_{i,j}$ values, column by column, where m corresponds to the number of elements in a column for an $m \times n$ grid, i and j correspond to the location of $u_{i,j}$ in the grid, and p is a number from 1 to $m \cdot n$ corresponding to the index of u in the vector $\vec{u}_p^{\,n+1}$.

Further, we must define our tag function, a system which was implemented in order to remove any trivial equations such as discretizations at grid points outside of Ω^-, where the solution was invalid and unused (i.e. $u_p^{n+1} = 0$). Skipping the creation of these trivial equations greatly improved computational efficiency, but changed the location of points in our vector $\vec{u}_p^{\,n+1}$. Consider, for example, if the first linear equation were to be removed. Every element in the vector $\vec{u}_p^{\,n+1}$ would now be located at $p-1$

(e.g. u_2^{n+1} would now be u_1^{n+1}). This tagging system filters through each grid point and *tags* any nontrivial points so that we can properly locate them in our nontrivial vector $\vec{u}_p^{\,n+1}$.

Table 3.1: Sparse Matrix Location Vectors

$\vec{A}$	$\vec{i}$	$\vec{j}$
Center	$\text{tag}(p(i,j,m))$	$\text{tag}(p(i,j,m))$
Left	$\text{tag}(p(i-1,j,m))$	$\text{tag}(p(i,j,m))$
Right	$\text{tag}(p(i+1,j,m))$	$\text{tag}(p(i,j,m))$
Below	$\text{tag}(p(i,j,m))$	$\text{tag}(p(i,j-1,m))$
Above	$\text{tag}(p(i,j,m))$	$\text{tag}(p(i,j+1,m))$

Table 3.1 describes how we would handle a point in which all the neighboring points are also encompassed in the reacted region Ω^-. If this is not the case - that is to say, if a neighboring point was in Ω^+, we would use a different treatment. In this case, we would not include a term in our $\boldsymbol{A}$ matrix. Instead, we must introduce ghost points.

3.2 Building a Symmetric Matrix to Treat Diffusion on Irregular Domains (The Ghost Fluid Method)

In constructing our diffusion matrix $\boldsymbol{A}\vec{u}^{\,n+1} = \vec{f}\,(\vec{u}^{\,n}, BC^{n+1})$, we consider only our unknown values of interest, such that $\vec{u}^{\,n+1}$ contains only values within the Ω^- domain. To reiterate, we can ignore any points that fall exactly on the interface ϕ_0^{n+1}, where Dirichlet boundary conditions are given, as well as any points in Ω^+.

Now, consider the example shown in Figure 3.1. Consider a point neighboring the interface such as $u_{2,2}^{n+1}$. To account for diffusion from $u_{2,3}^{n+1}$ and $u_{3,2}^{n+1}$, we implement the process developed in the previous section. However, we know that $u_{2,1}^{n+1} = u_\gamma$ since

it falls on the interface, and $u_{1,2}^{n+1}$ is contained in Ω^+. Since both of these points are excluded from our vector $\vec{u}^{\,n+1}$, but are needed in the discretization at $u_{2,2}^{n+1}$, we must define **ghost nodes** as proposed in [1] by Gibou and Fedkiw.

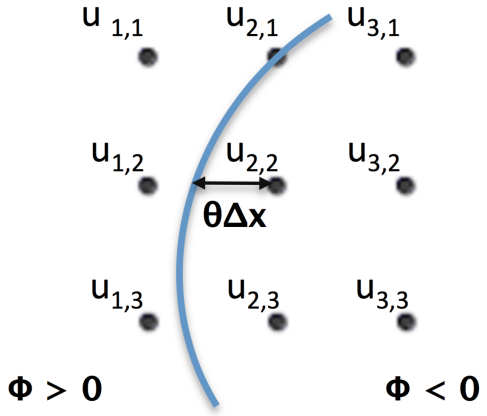


Figure 3.1: Assume the blue arc represents the interface, so that left of the arc $\phi^{n+1} > 0$, and right of the arc $\phi^{n+1} < 0$. Ghost nodes are used to handle any points neighboring the interface. The distance between neighboring points is Δx, whereas the distance between $u_{2,2}^{n+1}$ and the interface is $\theta \Delta x$.

Our diffusion equation (3.4) for $u_{2,2}^{n+1}$ is shown below, with the problem points bolded:

$$\begin{aligned} &\frac{u_{2,2}^{n+1}}{\Delta t} - \frac{1}{2}D\left(\frac{\boldsymbol{u_{1,2}^{n+1}} - 2u_{2,2}^{n+1} + u_{3,2}^{n+1}}{\Delta x^2} + \frac{\boldsymbol{u_{2,1}^{n+1}} - 2u_{2,2}^{n+1} + u_{2,3}^{n+1}}{\Delta y^2}\right) \\ &= \frac{u_{2,2}^{n}}{\Delta t} + \frac{1}{2}D\left(\frac{\boldsymbol{u_{1,2}^{n}} - 2u_{2,2}^{n} + u_{3,2}^{n}}{\Delta x^2} + \frac{\boldsymbol{u_{2,1}^{n}} - 2u_{2,2}^{n} + u_{2,3}^{n}}{\Delta y^2}\right) + \frac{1}{2}(S_{2,2}^{n+1} + S_{2,2}^{n}). \end{aligned} \tag{3.7}$$

These bolded values must be replaced with ghost values, u^G, found using linear and constant extrapolation, respectively. For example, for $u_{1,2}^{n+1}$, we would have the following two options:

$$\begin{aligned} u_{i-1,j}^{G} &= \frac{u_\gamma + (\theta - 1)u_{i,j}}{\theta}, \\ u_{i-1,j}^{G} &= u_\gamma, \end{aligned} \tag{3.8}$$

where u_γ is the concentration at the interface given by Dirichlet boundary conditions,

and $\theta \in [0,1]$ is the fractional distance between $u_{i,j}^{n+1}$ and the interface, as calculated previously in (2.7) and (2.8).

Wherever possible, we will use linear extrapolation (3.8a) to calculate our ghost value. However, one will notice that (3.8a) behaves poorly for small θ. Thus, when $\theta < \Delta x$, we need to use constant extrapolation (3.8b) to ensure stability. To do so, we would reassign the interface to this point, so that $\phi = 0$. So, if the interface did not fall exactly on the grid point $u_{2,1}^{n+1}$ in Figure 3.1, θ would be considered small enough to use (3.8b). This second-order-accurate perturbation of the interface location (also known as artificial boundary perturbation) does not degrade the overall second-order accuracy of the solution. Note, that reassigning a point to an interface location makes it a *known value*. Thus, we have to filter through every point to determine if θ is small enough to be considered on the interface and make this reassignment **before** we initiate preallocate the size of $\vec{A}$.

Finally, for the case shown in Figure 3.1, we end up with the following linear equation where the ghost values have been included:

$$\begin{aligned}
&\frac{u_{2,2}^{n+1}}{\Delta t} - \frac{1}{2}D\left(\frac{\frac{(\mathbf{1-\theta})}{\boldsymbol{\theta}}\boldsymbol{u_{2,2}^{n+1}} - 2u_{2,2}^{n+1} + u_{3,2}^{n+1}}{\Delta x^2} + \frac{-2u_{2,2}^{n+1} + u_{2,3}^{n+1}}{\Delta y^2}\right) \\
&\quad = \frac{u_{2,2}^{n}}{\Delta t} + \frac{1}{2}D\left(\frac{\frac{(\mathbf{1-\theta})}{\boldsymbol{\theta}}\boldsymbol{u_{2,2}^{n}} - 2u_{2,2}^{n} + u_{3,2}^{n}}{\Delta x^2} + \frac{-2u_{2,2}^{n} + u_{2,3}^{n}}{\Delta y^2}\right) \\
&\qquad + \frac{1}{2}(S_{2,2}^{n+1} + S_{2,2}^{n}) + \frac{1}{2}D\left(\frac{\frac{\boldsymbol{u_{\gamma}^{n+1}} + \boldsymbol{u_{\gamma}^{n}}}{\boldsymbol{\theta}}}{\Delta x^2} + \frac{\boldsymbol{u_{\gamma}^{n+1}} + \boldsymbol{u_{\gamma}^{n}}}{\Delta y^2}\right).
\end{aligned} \tag{3.9}$$

For the ghost value replacing $u_{1,2}^{n+1}$, we must alter two $\vec{A}$ values (3.5) on the LHS:

$$\begin{aligned}
\text{Left} &= 0, \\
\text{Center} &= 1 + \frac{1}{2}D\left(\frac{2\Delta t}{\Delta x^2} + \frac{2\Delta t}{\Delta y^2} + \frac{\frac{\theta-1}{\theta}\Delta t}{\Delta x^2}\right),
\end{aligned} \tag{3.10}$$

and add $\frac{1}{2}D\left(\frac{u_\gamma^{n+1}}{\theta\Delta x^2}\right)$ to the RHS. We follow the same procedure when we replace $u_{1,2}^n$ with its corresponding ghost value.

Since we use constant extrapolation to replace $u_{2,1}^{n+1}$, only one $\vec{A}$ value (3.5) changes:

$$\text{Above} = 0, \tag{3.11}$$

and we add $\frac{1}{2}D\left(\frac{u_\gamma^{n+1}}{\Delta y^2}\right)$ to the RHS. We use the same method to replace $u_{2,1}^n$ with its corresponding ghost value.

When linearly extrapolated ghost values are applied to create a sharp interface, we achieve **symmetric discretization**. This refers to the consistent representation of diffusion between the same two grid nodes. For instance, consider the diffusion between neighboring points $u_{2,2}^{n+1}$ and $u_{3,2}^{n+1}$ in Figure 3.1. The diffusion between the two points are described by the **Right** coefficient (3.5) in the discretization at $u_{2,2}^{n+1}$ and the **Left** coefficient (3.5) in the discretization at $u_{3,2}^{n+1}$. In the way we have implemented our ghost values, both coefficients would equal $-\frac{1}{2}D\frac{\Delta t}{\Delta x^2}$, hence the symmetric discretization.

Non-symmetric discretization, on the other hand, occurs when one uses quadratic extrapolation to determine ghost values:

$$u_{i-1,j}^G = \frac{2u_\gamma^{n+1} + (2\theta^2 - 2)u_{i,j} + (-\theta^2 + 1)u_{i+1,j}}{\theta^2 + \theta}. \tag{3.12}$$

In this case, the discretization at $u_{3,2}^{n+1}$ has not changed (i.e. Left $= -\frac{1}{2}D\frac{\Delta t}{\Delta x^2}$), but the discretization at $u_{2,2}^{n+1}$ now includes some factor θ in the Right coefficient $\left(\text{i.e. Right} = -\frac{1}{2}D\frac{\Delta t}{\Delta x^2}\left(1 + \frac{-\theta^2+1}{\theta^2+\theta}\right)\right)$. This unequal representation of diffusion between the same two points leads to a non-symmetric discretization, which is computationally more expensive to solve. For this reason, we use linearly extrapolated ghost values.

3.3 Accuracy of the Diffusion Solver

Figure 3.2 illustrates our diffusion program working with several different level-sets described by the equations in Table 3.2 on a domain $\Omega = [-1, 1]^2$, with a source term

$$S(x, y, t) = \cos(x)\sin(x)e^y\cos(t) - D(-2\sin(2x)e^y\sin(t) + \cos(x)\sin(x)e^y\sin(t)),$$

where we use a diffusion coefficient $D = 2$, and ran the simulation to a final time $t = 0.1$. The reason for this somewhat convoluted source term is because we are trying to impose a desired outcome $u_{\text{numerical}}$. To do this, we plug a test function

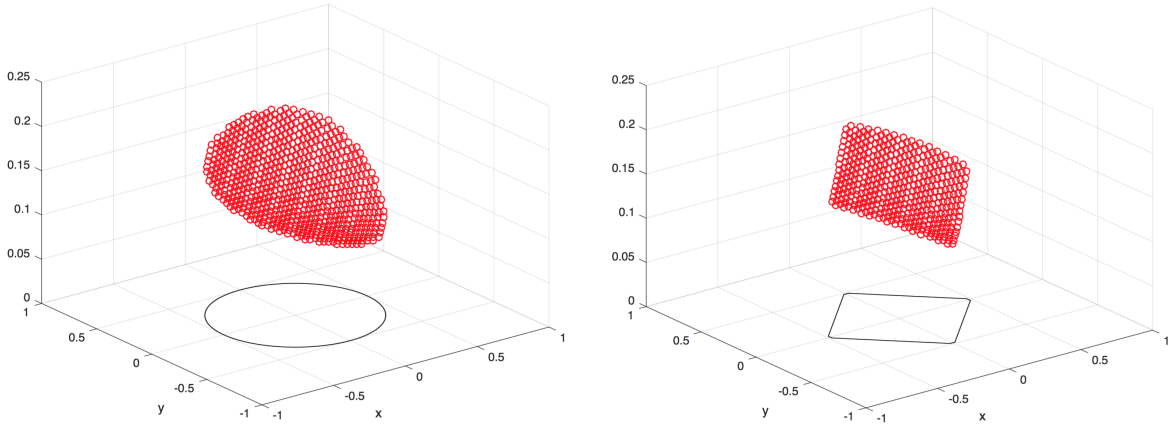
$$u_{\text{exact}}(x, y, t) = \cos(x)\sin(x)e^y\sin(t) \tag{3.13}$$

into the diffusion equation (3.4) and solve analytically for S. We can then use u_{exact} to define initial conditions and Dirichlet boundary conditions, and use the source term to compute a numerical solution, $u_{\text{numerical}}$, that we can compare to u_{exact} for error analysis.

Tables 3.3 - 3.6 demonstrate that our diffusion solver has second-order accuracy for all four level-set shapes, where we set $\Delta t = \Delta x$ to ensure that the stringent time step restriction has been alleviated through our implicit method.

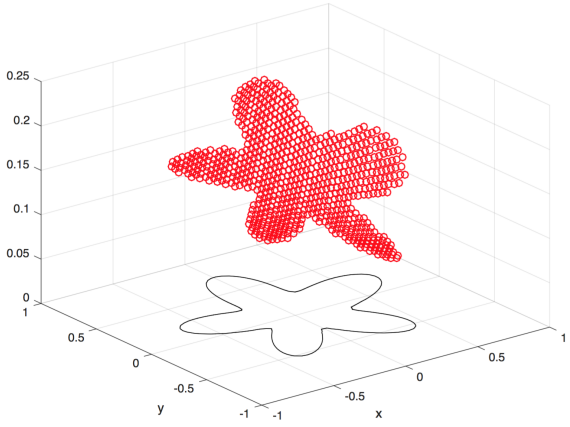
Table 3.2: Level-Set Functions

Circle	$\phi(x, y) = \sqrt{x^2 - y^2} - 0.5$
Diamond	$\phi(x, y) = \lvert x\rvert + \lvert y\rvert - 0.5$
Fat Flower	$\phi(x, y) = (x - 0.02\sqrt{5})^2 + (y - 0.02\sqrt{5})^2 - (0.5 + 0.2\ \sin(5\,\theta(x, y)))^2$
Thin Flower	$\phi(x, y) = \sqrt{x^2 + y^2} - (0.35 + 0.28\ \cos(9\,\theta(x, y)))$

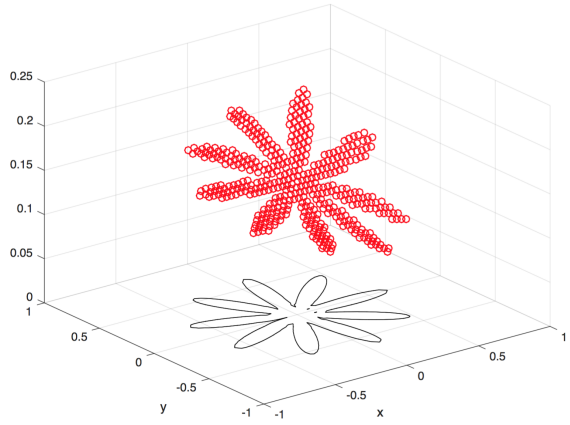


(a) Circle

(b) Diamond



(c) Fat-Petal Flower



(d) Thin-Petal Flower

Figure 3.2: Two-dimensional solutions of the heat equation with Dirichlet boundary conditions solved in only the Ω^- subdomain. We use the same source term for each level-set shape (described by the equations in Table 3.2) on a 60×60 grid resolution to impose our desired outcome $u_{\text{numerical}}$ for error analysis.

Table 3.3: Circle 2D Heat Equation - Crank-Nicholson - $\Delta t = \Delta x$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	1.616×10^{-6}	-	1.068×10^{-5}	-
161×161	4.327×10^{-7}	1.90	2.869×10^{-6}	1.90
321×321	1.091×10^{-7}	1.99	7.671×10^{-7}	1.90

Table 3.4: Diamond 2D Heat Equation - Crank-Nicholson - $\Delta t = \Delta x$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	3.408×10^{-8}	-	1.716×10^{-7}	-
161×161	8.319×10^{-9}	2.03	4.225×10^{-8}	2.02
321×321	2.075×10^{-9}	2.00	1.060×10^{-8}	2.00

Table 3.5: Fat-Petal Flower 2D Heat Equation - Crank-Nicholson - $\Delta t = \Delta x$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	2.291×10^{-6}	-	2.517×10^{-5}	-
161×161	4.704×10^{-7}	2.28	6.856×10^{-6}	1.88
321×321	1.037×10^{-7}	2.18	2.099×10^{-6}	1.71

Table 3.6: Thin-Petal Flower 2D Heat Equation - Crank-Nicholson - $\Delta t = \Delta x$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	2.061×10^{-5}	-	2.933×10^{-4}	-
161×161	5.284×10^{-6}	1.96	7.207×10^{-5}	2.02
321×321	1.296×10^{-6}	2.03	1.303×10^{-5}	2.47

Chapter 4

Calculating the Interface Velocity

The Stefan problem is a free boundary problem for which we solve for both a changing concentration u and changing domain shape Ω^-. Here, we discuss how we account for the changing Ω^- by using the level-set advection equation's (1.2) interface velocity

$$\vec{v} = [D\nabla u]_\gamma \tag{4.1}$$

to find our new ϕ^{n+1}, where D is the diffusion coefficient, ∇u is the gradient of concentration, and $[\cdot]_\gamma$ denotes a jump across the interface. In a more explicit representation, we are looking for

$$\begin{aligned} v_x &= D_{\text{in}}\left(\frac{\partial u_{\text{in}}}{\partial x}\right) - D_{\text{out}}\left(\frac{\partial u_{\text{out}}}{\partial x}\right), \\ v_y &= D_{\text{in}}\left(\frac{\partial u_{\text{in}}}{\partial y}\right) - D_{\text{out}}\left(\frac{\partial u_{\text{out}}}{\partial y}\right), \end{aligned} \tag{4.2}$$

where v_x and v_y represent the x and y components of the interface velocity, respectively. Further, D_{in} and u_{in} refer to values in the subdomain Ω^-, whereas D_{out} and u_{out} refer to values in the subdomain Ω^+.

To compute the gradient of ϕ, we use centered finite differencing in space

$$\nabla\phi = \Big(\phi_x, \phi_y\Big) = \left(\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2\Delta x}, \frac{\phi_{i,j+1} - \phi_{i,j-1}}{2\Delta y}\right). \tag{4.3}$$

We use central differencing again to find the gradient of u

$$\nabla u = \Big(u_x, u_y\Big) = \left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}, \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y}\right). \tag{4.4}$$

It is in this step that our Stefan solver becomes first-order accurate. This becomes evident when one considers the gradient of u with the orders included. Remember that our solution for u is limited by the second-order accurate diffusion solver so that each u has an associated error $O(\Delta x^2)$. Central differencing is second-order accurate as well, so if we consider these orders of error, our equation appears as:

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} + O(\Delta x^2) - u_{i-1,j} + O(\Delta x^2)}{2\Delta x} + O(\Delta x^2), \tag{4.5}$$

which reduces to a first-order accurate derivative since

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + O(\Delta x) + O(\Delta x^2), \tag{4.6}$$

simply becomes

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + O(\Delta x). \tag{4.7}$$

This occurs because the smaller magnitude of error $O(\Delta x^2)$ will be dominated by the first-order error $O(\Delta x)$. As one can see, central differencing will lower the order of accuracy by one since Δx is in the denominator. This means at best, with a second-order accurate solution, one can achieve first-order accuracy. And since our interface velocity is first-order, our Stefan solver is first-order accurate.

Seeing that we need to find gradients at the interface in both the reacted Ω^- and unreacted Ω^+ domains, we implement a third-order extrapolation method to extrapolate values of u in the layer of grid nodes near the interface. As we discussed before, our Dirichlet boundary conditions allow us to decouple the solutions across each region. That is to say, we can store the two different region concentrations u_{in} and u_{out}, separately. For u_{in}, we extrapolate from the Ω^- domain a small layer into the Ω^+ domain; and for u_{out}, we extrapolate from the Ω^+ domain a small layer into the Ω^- domain. Thus, we will create a small band, or layer of points, around the interface for which we have valid extrapolated values for both u_{in} and u_{out}.

Now, we can calculate the gradients at the interface and calculate the interface velocity. Although this velocity is only needed at the interface, in practice it is easier to simply compute the interface velocity in the entire domain.

4.1 Third-Order Extrapolation Method

We implement the third-order extrapolation method presented by Aslam in [5]. This higher-order extrapolation method is necessary for two reasons. First, we require accurate gradient values ∇u at the interface to calculate the interface velocity $\vec{v}$. Second, as the phase boundary moves to include new grid nodes, we need to have valid values of u_{in} at these new grid nodes in order to employ the Crank-Nicholson method for our diffusion solver. Therefore, we need to extrapolate u quadratically in a band about the interface. To achieve this, we first calculate the second normal directional derivative of u in our domain Ω^-

$$u_{nn} = \vec{n} \cdot \nabla(\vec{n} \cdot \nabla u), \tag{4.8}$$

where ∇u is defined by (4.4) and $\vec{n}$ is the local unit normal to the interface ($\vec{n} = \frac{\nabla\phi}{|\nabla\phi|}$), or in more explicit representation

$$\vec{n} \ = \Big(n_1, n_2\Big) = \ \Big(\frac{\phi_x}{\sqrt{\phi_x^2 + \phi_y^2}} \ , \ \frac{\phi_y}{\sqrt{\phi_x^2 + \phi_y^2}} \Big). \tag{4.9}$$

Then, we solve the following partial differential equations for u_{nn}, u_n, and u, respectively, using an iterative solver

$$\begin{aligned} &\frac{\partial u_{nn}}{\partial\tau} + H(\phi, u_{nn})(\vec{n}\cdot\nabla u_{nn} \qquad\quad\ ) = 0, \\ &\frac{\partial u_{n}}{\partial\tau} + H(\phi, u_{n}\ )(\vec{n}\cdot\nabla u_{n}\ - u_{nn}) = 0, \\ &\frac{\partial u}{\partial\tau} + H(\phi, u\ \ )(\vec{n}\cdot\nabla u\ \ - u_{n}\ ) = 0, \end{aligned} \tag{4.10}$$

where the Heaviside functions H are defined below, and τ is a fictitious time. Only a few iterations are required to propagate the solution out in the layer of grid points near the interface. Notice, that the three equations follow the same format whereby we impose our desired normal derivative by subtracting it from the parenthesized term $(\vec{n}\cdot\nabla u)$. Hence, to impose extrapolation in the normal direction, we begin by subtracting zero from $(\vec{n}\cdot\nabla u_{nn})$. Now, we have our second directional derivative defined and can use it to compute the first directional derivative u_n by solving our second PDE. And finally, we compute u in our third PDE with the imposed first directional derivative u_n. In semi-discretized form, we have

$$\begin{aligned} &\frac{d}{d\tau}u_{nn} + H(\phi, u_{nn})(n_x^+ D_x^- u_{nn} + n_x^- D_x^+ u_{nn} + n_y^+ D_y^- u_{nn} + n_y^- D_y^+ u_{nn}) = 0, \\ &\frac{d}{d\tau}u_{n}\ + H(\phi, u_{n)}\ (n_x^+ D_x^- u_{n}\ + n_x^- D_x^+ u_{n}\ + n_y^+ D_y^- u_{n}\ + n_y^- D_y^+ u_{n}\ ) = H(\phi, u_n) u_{nn}, \\ &\frac{d}{d\tau}u\ \ + H(\phi, u\ \ )(n_x^+ D_x^- u\ \ + n_x^- D_x^+ u\ \ + n_y^+ D_y^- u\ \ + n_y^- D_y^+ u\ \ ) = H(\phi, u) u_n\ , \end{aligned} \tag{4.11}$$

where $n_x^+ = \max(n_x, 0)$, $n_x^- = \min(n_x, 0)$, $n_y^+ = \max(n_y, 0)$, and $n_y^- = \min(n_y, 0)$. Also, we use the first-order one-sided finite differences (2.4) to calculate our directional derivatives.

The Heaviside functions $H(\phi, x)(v_j)$ label each point as having a known or unknown value for x at a point v_j, where zero corresponds to a known and one corresponds to an unknown. For example, since we know u at all points in Ω^-, $H(\phi, u)(v_j) = 0$ at all points v_j where $\phi < 0$, while $H(\phi, u)(v_j) = 1$ for all points v_j where $\phi \geq 0$. (Recall that since we reassign values of ϕ to zero when θ is too small, it is better to consider these u values to be unknown, so that we can extrapolate values at these points where $\phi = 0$. We found this produced more robust and accurate results.)

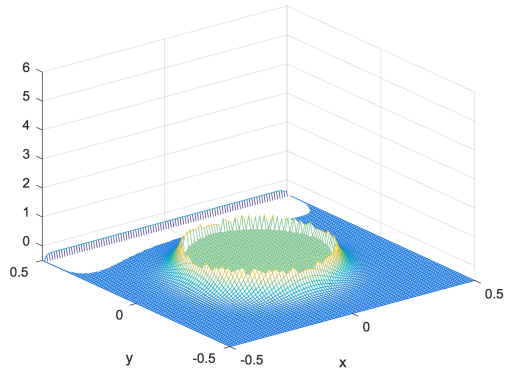
To calculate the derivatives u_x and u_y using central differencing, we need all four neighboring points v_i to be contained in Ω^-. In other words, we need $H(\phi, u)(v_i) = 0$ for all four neighbors v_i of v_j to set $H(\phi, u_n)(v_j) = 0$. Likewise, we need all four neighboring points v_i to have known first derivatives in order to use central differencing to find the second derivatives (u_{xx} and u_{yy}). To summarize, the Heaviside functions are defined as

$$H(\phi, u_{nn})(v_i) = \begin{cases} 0 & \text{if } H(\phi, u_n)(v_j) = 0 \text{ for all } v_j \in \text{ngbd}(v_i), \\ 1 & \text{otherwise,} \end{cases}$$

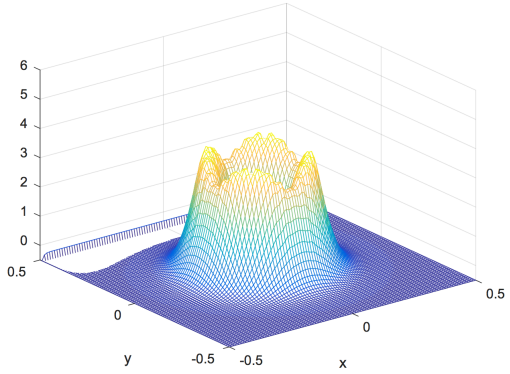
$$H(\phi, u_n\ )(v_i) = \begin{cases} 0 & \text{if } H(\phi, u)(v_j) = 0 \text{ for all } v_j \in \text{ngbd}(v_i), \\ 1 & \text{otherwise,} \end{cases} \tag{4.12}$$

$$H(\phi, u\quad)(v_i) = \begin{cases} 0 & \text{if } \phi(v_i) < 0, \\ 1 & \text{otherwise,} \end{cases}$$

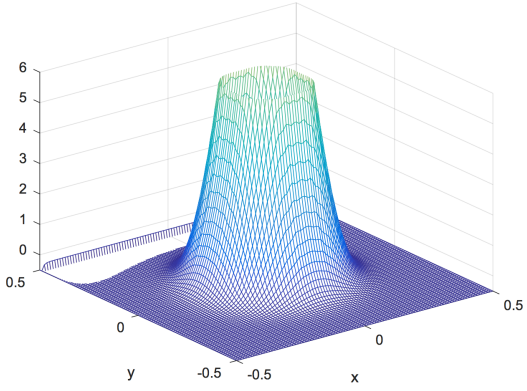
Figure 4.1 demonstrates visually what this third-order extrapolation method achieves.



(a) Initial Data



(b) 15 iterations



(c) 30 iterations

Figure 4.1: We begin with u defined everywhere except inside the circle at the center. As we execute more iterations, the extrapolated solution gets propagated further. In addition, it becomes more accurate so that the points shown in (b) are now more accurate in (c).

4.2 Accuracy Analysis

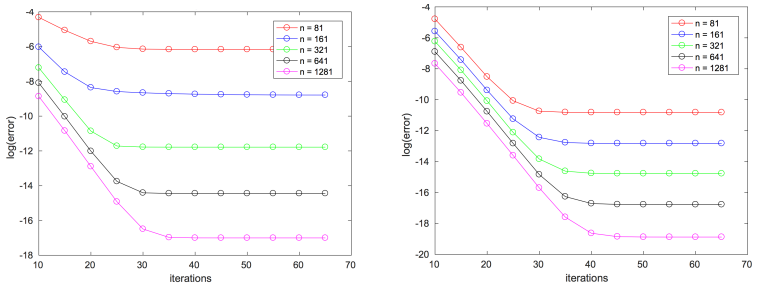
Since we are only interested in the points near the interface, we compare the exact solution to our extrapolated solution for only points in Ω^+ where $|\phi| < 2\Delta x$. Furthermore, we investigated the necessary amount of iterations needed for our solution to converge in this band, with the goal of maximizing the order of accuracy, while minimizing the amount of iterations to save computational effort. Figure 4.2 plots the error analysis in L^1 and L^∞ against the number of iterations. We perform this error analysis for the four level-sets described in Table 3.2 to demonstrate the third-order extrapolation method's convergence dependency on both grid resolution and level-set smoothness. Also, note that we had to reinitialize the level-sets before third-order accurate extrapolation could be achieved.

As one can see in Figure 4.2, the accuracy continues to improve for a given number of iterations until steady state is reached. The necessary amount of iterations to achieve this maximum accuracy increases with the number of grid points.

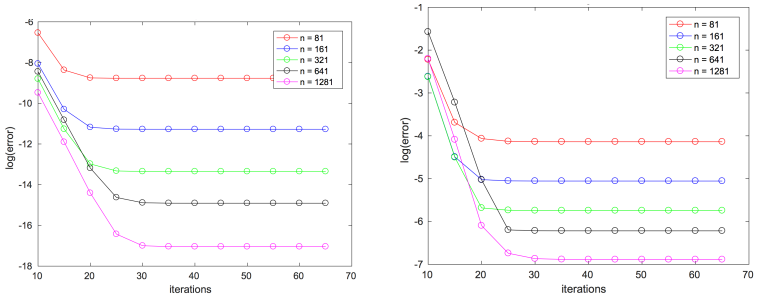
The level-sets that displayed sharp features or kinks reached convergence faster than the smooth circle, but had much larger magnitudes of error at convergence. In Figure 4.3, one can see that this larger error is caused by the sharp corners. However, we found that for all level-set cases, 50 iterations was enough for convergence and so the following tables show orders of accuracy after 50 iterations, which we expect to have at least third-order convergence in L^1, and first-order convergence in L^∞.

Table 4.1: Circle Third-Order Extrapolation

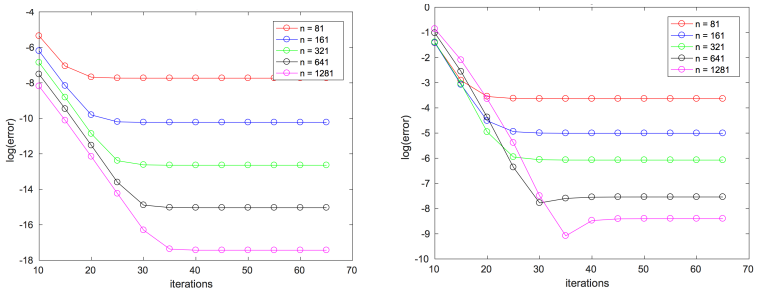
Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	2.139×10^{-5}	-	2.757×10^{-4}	-
161×161	2.972×10^{-6}	2.84	9.393×10^{-5}	1.55
321×321	4.102×10^{-7}	2.86	2.707×10^{-5}	1.79



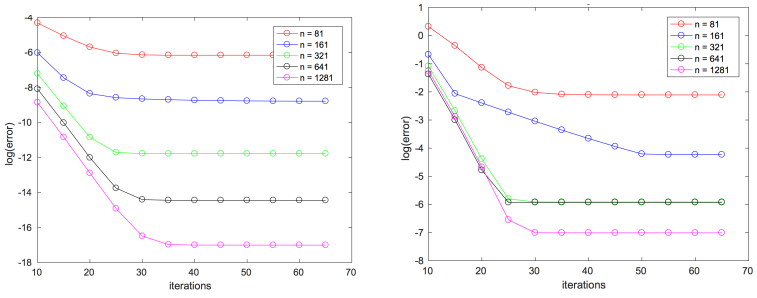
(a) Circle L^1 (b) Circle L^∞



(c) Diamond L^1 (d) Diamond L^∞

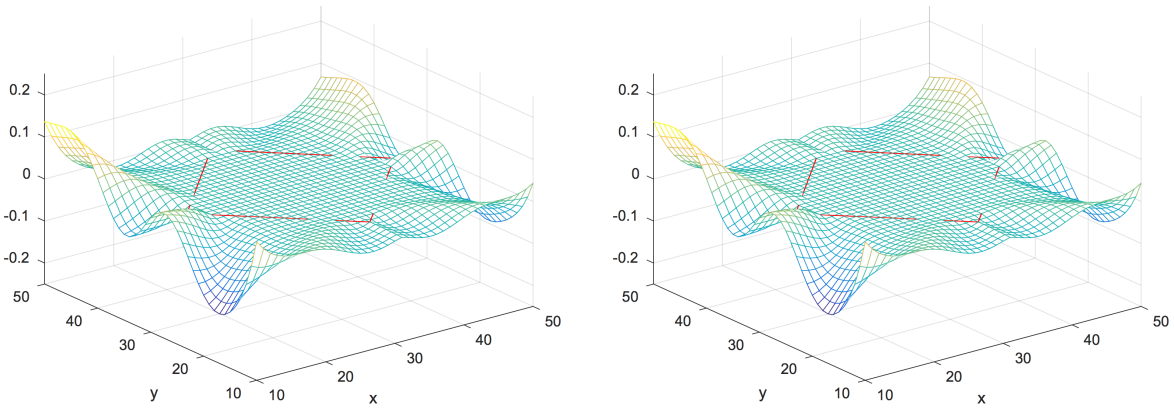


(e) Fat-Petal Flower L^1 (f) Fat-Petal Flower L^∞



(g) Thin-Petal Flower L^1 (h) Thin-Petal Flower L^∞

Figure 4.2: Iterations required to reach convergence in the band of space $2\Delta x$ outside the interface for the four level-set shapes described by in Table 3.2.



(a) Diamond L^1

(b) Diamond L^∞

Figure 4.3: Notice the error arises at the sharp corners of the diamond level-set. Likewise, with the flower level-sets (not pictured), we observed the greatest error at the sharp corners where the petals meet at the base.

Table 4.2: Diamond Third-Order Extrapolation

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	2.459×10^{-4}	-	1.470×10^{-2}	-
161×161	3.157×10^{-5}	2.96	7.500×10^{-3}	0.97
321×321	4.014×10^{-6}	2.98	3.800×10^{-3}	0.98

Table 4.3: Fat-Petal Flower Third-Order Extrapolation

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	2.654×10^{-4}	-	1.410×10^{-2}	-
161×161	1.262×10^{-5}	4.39	4.900×10^{-3}	1.52
321×321	1.147×10^{-6}	3.46	4.999×10^{-4}	3.30

Table 4.4: Thin-Petal Flower Third-Order Extrapolation

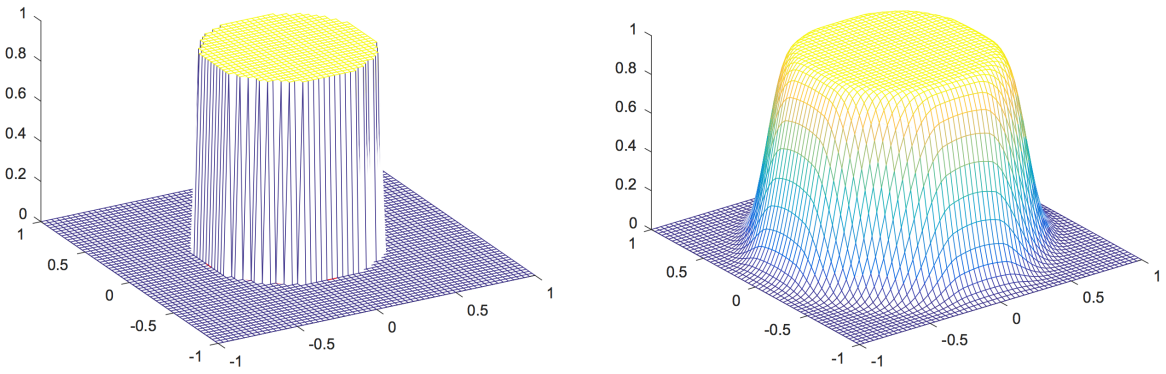
Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	1.370×10^{-2}	-	1.793×10^{-1}	-
161×161	6.558×10^{-4}	4.38	5.070×10^{-2}	1.82
321×321	1.590×10^{-5}	5.37	7.000×10^{-3}	2.85

4.3 Constant Extrapolation of the Interface Velocity

Although we are interested only in the level-set velocity at the interface $\phi = 0$, we compute the level-set velocity on our entire grid. This means different parts of our level-set will be displaced at different rates which could lead to a poorly behaved level-set. However, there is a simple preconditioning tool we can use to make our level-set behave better and allow the reinitialization equation to converge faster. We implement constant extrapolation of the interface velocity. This is a first-order extrapolation method, and thus uses only one PDE with an imposed first derivative of 0

$$\begin{aligned} \frac{\partial v_x}{\partial \tau} + H(\phi, v_x)(\vec{n} \cdot \nabla v_x) = 0, \\ \frac{\partial v_y}{\partial \tau} + H(\phi, v_y)(\vec{n} \cdot \nabla v_y) = 0. \end{aligned} \tag{4.13}$$

We call this function using ϕ to constantly extrapolate the velocities from the interface into Ω^+, then repeat the process using $-\phi$ to extrapolate the velocities from the interface into Ω^-. In this way, the values at the interface will be extrapolated in the normal direction throughout the grid space. Figure 4.4 demonstrates this constant extrapolation.



(a) Velocity in the subdomain Ω^- (b) Velocity extrapolated beyond the interface

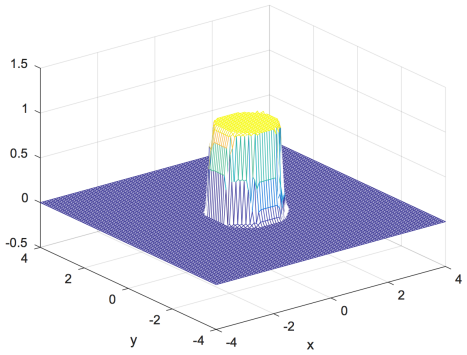
Figure 4.4: Constant normal extrapolation of the values in the Ω^- region out into the Ω^+ region.

Chapter 5

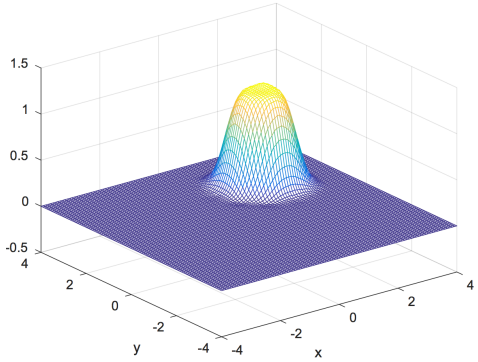
Advection

We use the level-set equation (1.2) to keep track of the moving interface. This equation can be described as advection, or the transport of material via bulk motion (see Figure 5.1). Advection is described by a linear hyperbolic PDE, which is not trivial to solve numerically. Because it contains strictly first derivatives and a causality principle, unique numerical challenges arise in its discretization. For instance, using standard central differencing techniques can create instability. Upwind schemes are a popular numerical technique used to tackle such problems because they consider the direction of propagation of information. However, we must be careful to select the appropriate time step to minimize both even-order spatial derivative related errors (i.e. numerical dissipation, or damping); and odd-order spatial derivative related errors (i.e. numerical dispersion, or the incorrect propagation velocity, which can also produce dispersive waves left in the wake). Choosing the appropriate CFL (Courant-Friedrichs-Lewy) number will reduce these errors as well as ensure numerical stability. In practice, we use a CFL number of 0.5.

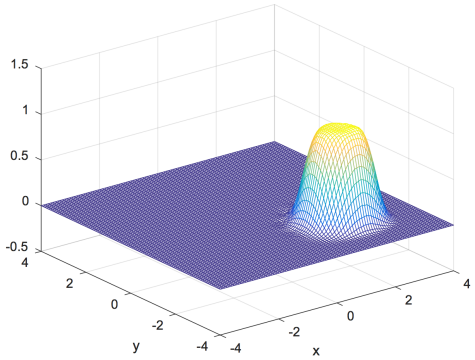
Thus far, we have used first-order accurate upwind schemes which are impractical in this application. But, higher-order methods like the Lax-Wendroff and Beam-Warming



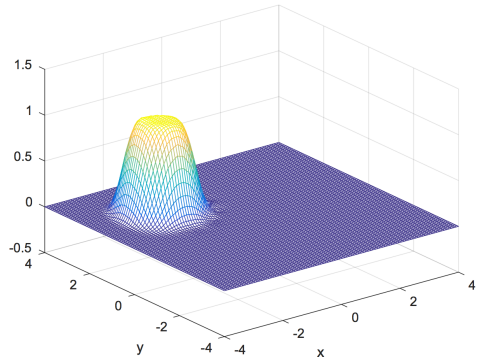
(a) Initial Data



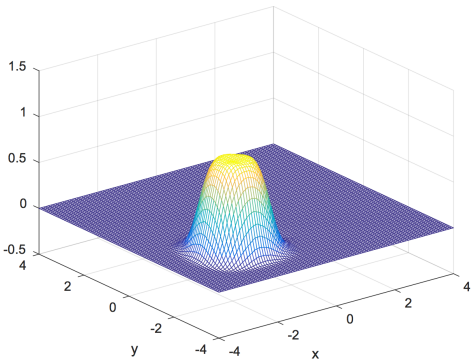
(b) $u = +1, v = +1$



(c) $u = +1, v = -1$



(d) $u = -1, v = +1$



(e) $u = -1, v = -1$

Figure 5.1: Advection in different directions.

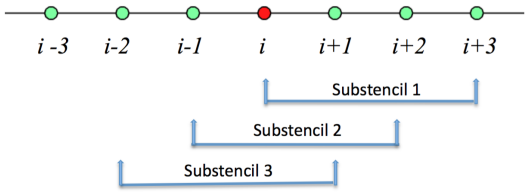
methods produce oscillatory results near discontinuities. Thus, we develop a higher-order spatial discretization scheme known as WENO, or the Weighted Essentially Non-Oscillatory scheme, first introduced by Liu, Osher, and Chan in [6]. We combine this

with a TVD RK3 time discretization to compute advection.

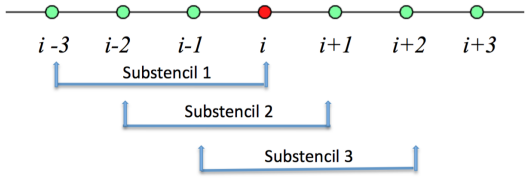
5.1 WENO Schemes

The WENO scheme was developed for solving one-sided finite difference problems with higher-order accuracy in smooth regions, while simultaneously improving behavior near discontinuities. This is achieved by considering three different stencils (depending on the upwind direction), and determining for each grid point which stencil or convex weighted combination of stencils will produce the smoothest results.

Figure 5.2(a) shows the three substencil choices for the upwind direction D_x^+, so called because it involves the points $i+1$ and i, such that information propagates from $i+1$ to i and $v_x < 0$. Figure 5.2(b) shows the three substencil choices for the upwind direction D_x^-, so called because it involves the points $i-1$ and i, such that information propagates from $i-1$ to i and $v_x > 0$.



(a) Possible stencil choices for computing D_x^+



(b) Possible stencil choices for computing D_x^-

Figure 5.2: WENO substencil choices for computing one-sided finite differences.

Figure 5.3 shows an example of how we would choose the smoothest possible stencil.

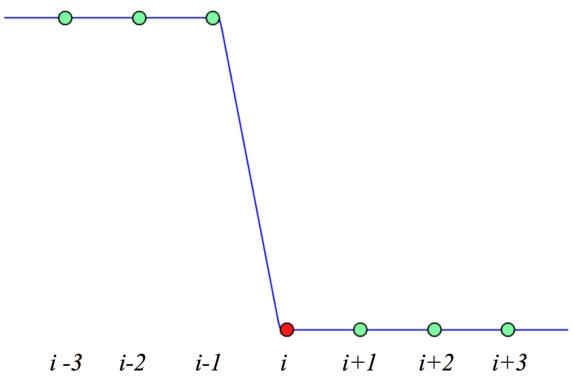


Figure 5.3: Suppose for the stencil you see here, v_x is negative, so that information is traveling from $i+1$ to i. We would then make our substencil selection from the options for D_x^+ in Figure 5.2(a). It is clear from inspection that substencils 2 and 3 introduce a new maximum which would trigger oscillations in the solution. Thus, we would ultimately choose substencil 1 because it is the smoothest.

This smoothness can be computed by determining the divided differences which measure the level of discontinuity in a solution. The three possible ENO approximations of $D_x^- u$ are

$$\begin{aligned} u_x^1 &= \frac{d_1}{3} - \frac{7d_2}{6} + \frac{11d_3}{6}, \\ u_x^2 &= -\frac{d_2}{6} + \frac{5d_3}{6} + \frac{d_4}{3}, \\ u_x^3 &= \frac{d_3}{3} + \frac{5d_4}{6} - \frac{d_5}{6}, \end{aligned}$$

where d_k is defined as the simple finite differences:

$$\begin{aligned} d_1 &= \frac{u_{i-2} - u_{i-3}}{\Delta x}, \\ d_2 &= \frac{u_{i-1} - u_{i-2}}{\Delta x}, \\ d_3 &= \frac{u_i \quad - u_{i-1}}{\Delta x}, \\ d_4 &= \frac{u_{i+1} - u_i}{\Delta x}, \\ d_5 &= \frac{u_{i+2} - u_{i+1}}{\Delta x}. \end{aligned}$$

The WENO approximation of $D_x^- u$ is

$$D_x^- u = \omega_1 u_x^1 + \omega_2 u_x^2 + \omega_3 u_x^3, \tag{5.1}$$

where the stencil weights ω create a convex combination (i.e. $\omega_1 + \omega_2 + \omega_3 = 1$), chosen to create fifth-order accuracy in regions of smoothness, and reduce oscillatory behavior near discontinuities. To begin, we calculate the smoothness S of each stencil

$$\begin{aligned} S_1 &= \frac{13}{12}(d_1 - 2d_2 + d_3)^2 + \frac{1}{4}(d_1 - 4d_2 + 3d_3)^2, \\ S_2 &= \frac{13}{12}(d_2 - 2d_3 + d_4)^2 + \frac{1}{4}(d_2 - d_4)^2, \\ S_3 &= \frac{13}{12}(d_3 - 2d_4 + d_5)^2 + \frac{1}{4}(3d_3 - 4d_4 + d_5)^2. \end{aligned}$$

Then, we define the coefficients α_k as

$$\alpha_1 = \frac{0.1}{(S_1 + \epsilon)^2}, \quad \alpha_2 = \frac{0.6}{(S_2 + \epsilon)^2}, \quad \alpha_3 = \frac{0.3}{(S_3 + \epsilon)^2},$$

where $\epsilon = 10^{-6} \times \max(d_1^2, d_2^2, d_3^2, d_4^2, d_5^2) + 10^{-99}$. Finally, we can compute ω_k as

$$\omega_1 = \frac{\alpha_1}{\alpha_1 + \alpha_2 + \alpha_3}, \quad \omega_2 = \frac{\alpha_2}{\alpha_1 + \alpha_2 + \alpha_3}, \quad \omega_3 = \frac{\alpha_3}{\alpha_1 + \alpha_2 + \alpha_3}.$$

The construction of $D_x^+ u$ uses the same process, except that our simple finite differences

d_k change to

$$\begin{aligned} d_1 &= \frac{u_{i+3} - u_{i+2}}{\Delta x}, \\ d_2 &= \frac{u_{i+2} - u_{i+1}}{\Delta x}, \\ d_3 &= \frac{u_{i+1} - u_i}{\Delta x}, \\ d_4 &= \frac{u_i - u_{i-1}}{\Delta x}, \\ d_5 &= \frac{u_{i-1} - u_{i-2}}{\Delta x}. \end{aligned}$$

Further, to extend these schemes to two-dimensions, we simply execute the above dimension by dimension. That is to say, we begin with column $j = 1$, and complete the spatial discretization for this column before moving to the next column, $j = 2$. And to construct $D_y^+ u$ and $D_y^- u$, we simply replace the i index with j.

5.2 Accuracy of WENO Schemes for Computing First Derivatives

To test for the accuracy of our WENO schemes, we first used a continuous test function

$$\phi(x, y) = \cos(x)\sin(y), \tag{5.2}$$

and solved analytically for the partial derivatives with respect to x and y

$$\begin{aligned} \phi_x(x, y) &= -\sin(x)\sin(y), \\ \phi_y(x, y) &= \cos(x)\cos(y), \end{aligned} \tag{5.3}$$

in the domain $\Omega = [0,2\pi]^2$ so that periodic boundary conditions could be applied (for the sake of error analysis). We compare ϕ_x to the resulting numerical solution found using our D_x^+ and D_x^- schemes; and compare ϕ_y to the numerical solution found using D_y^+ and D_y^-. For a continuous function like this, we expect fifth-order accuracy, and from Table 5.1, we can see that this was indeed achieved. The L^1 and L^∞ error, as well as the orders were the same for all four upwind directions.

Table 5.1: 2D WENO Scheme for All Upwind Directions

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	5.502×10^{-6}	-	3.489×10^{-7}	-
161×161	1.738×10^{-7}	4.98	1.090×10^{-8}	5.00
321×321	5.426×10^{-9}	5.00	3.405×10^{-10}	5.00

Perhaps, of more interest is what happens when we use a discontinuous function. For this second test, we used the piecewise function shown in Figure 5.4 and defined by

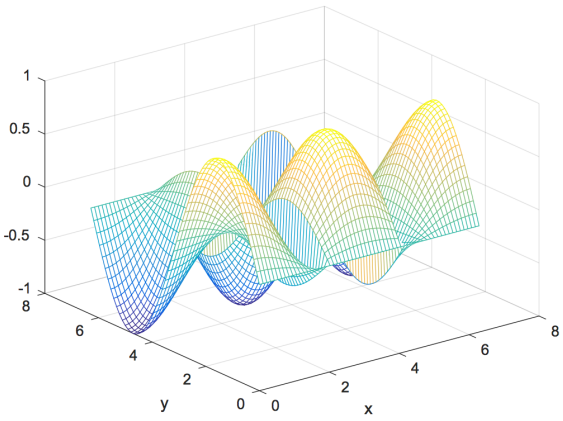
$$\phi(x,y) = \begin{cases} \cos(x)\sin(y) & \text{for} \quad 0 \le x < N/3, \\ -\cos(x)\sin(y) & \text{for} \quad N/3 \le x \le 2N/3, \\ \cos(x)\sin(y) & \text{for} \quad 2N/3 < x \le N, \end{cases} \tag{5.4}$$

where N is the number of points in the grid, so that the function is the same in the initial and final third, but negative in the middle third. By beginning and ending with the same function, we can impose periodic boundary conditions which was necessary for accuracy testing. The above equation is discontinuous in the x-direction, and so was used to test our D_x^+ and D_x^- programs against the exact analytical solution ϕ_x in (5.3). Again, noting that for the middle third, we use $-\phi_x$ (5.3). We use the same process to test the accuracy of D_y^+ and D_y^-, except that we make our function discontinuous along

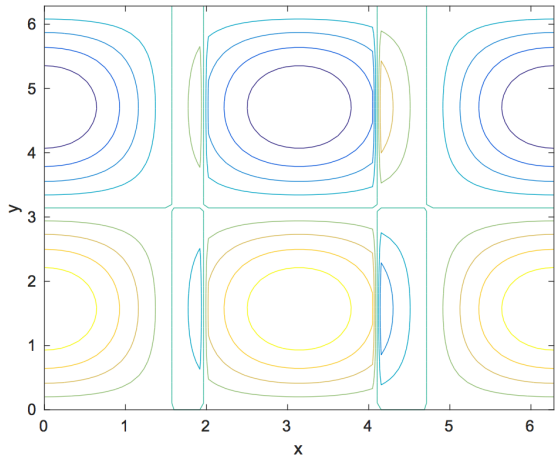
y, instead:

$$\phi(x,y) = \begin{cases} \cos(x)\sin(y) & \text{for} \quad 0 \le y < N/3, \\ -\cos(x)\sin(y) & \text{for} \quad N/3 \le y \le 2N/3, \\ \cos(x)\sin(y) & \text{for} \quad 2N/3 < y \le N. \end{cases} \tag{5.5}$$

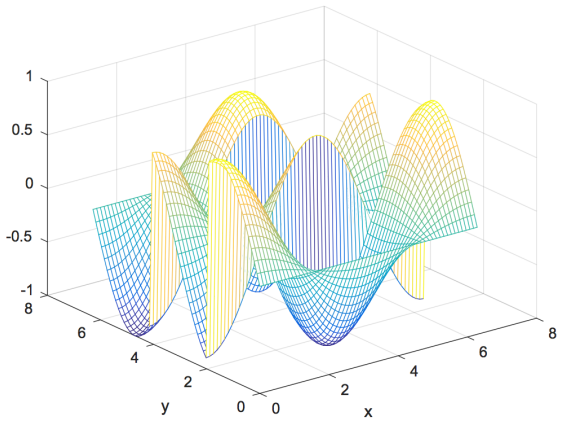
For a discontinuous function, the greatest error in our WENO scheme accumulates near the discontinuity with a peak of error along the row or column at which the discontinuity actually occurs. Neglecting this one row of error which is impossible to avoid, we would expect at least third-order convergence for our piecewise function. Tables 5.2 - 5.5 show that for each of the four one-sided finite difference WENO scheme, third-order accuracy is achieved.



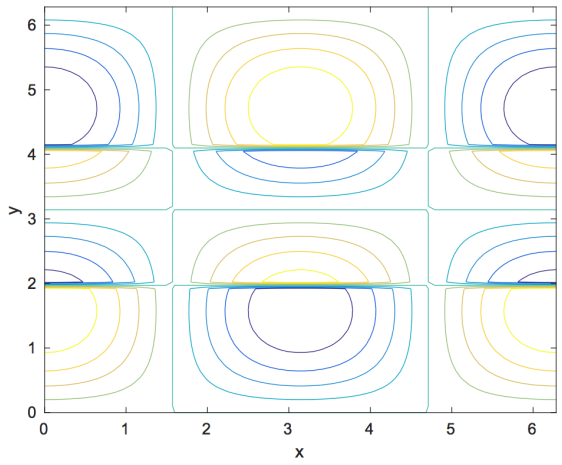
(a) Mesh: Discontinuous function ϕ about x



(b) Contour: Discontinuous function ϕ about x

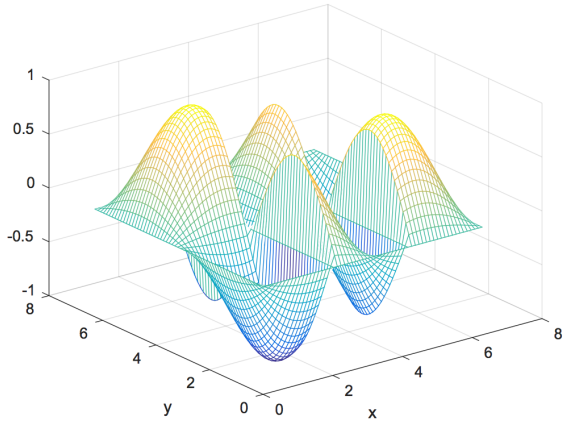


(c) Mesh: Discontinuous function ϕ about y

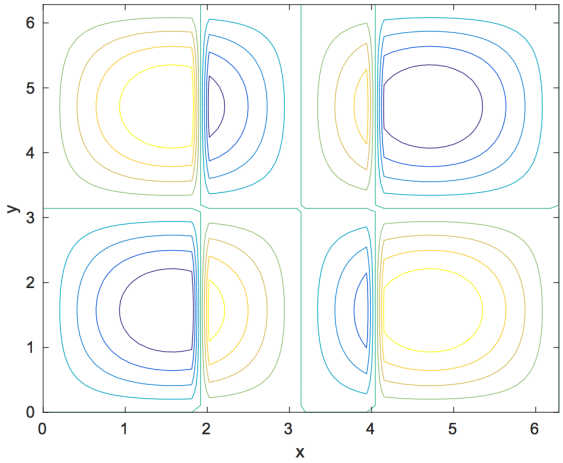


(d) Contour: Discontinuous function ϕ about y

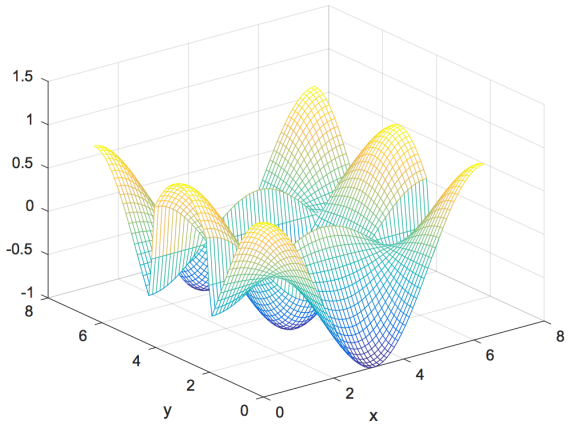
Figure 5.4: Discontinuous function separated into three parts where the first and final third are defined by the equation $f(x, y) = \cos(x)\sin(y)$, whereas the middle is defined by the negative of that function $f(x, y) = -\cos(x)\sin(y)$.



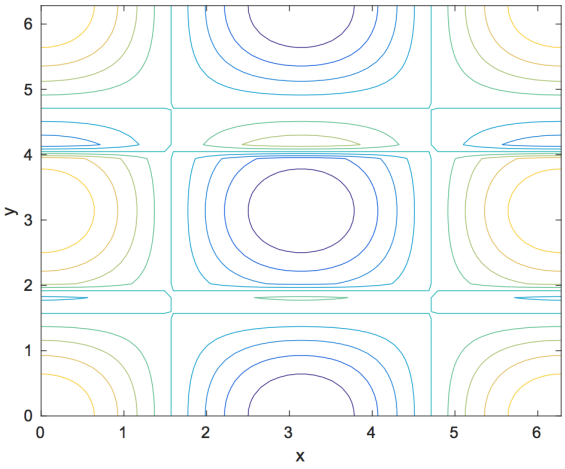
(a) Mesh: ϕ_x



(b) Contour: ϕ_x



(c) Mesh: ϕ_y



(d) Contour: ϕ_y

Figure 5.5: The derivative ϕ_x was calculated using the function ϕ discontinuous about x, and the derivative ϕ_y was calculated using the function ϕ discontinuous about y to demonstrate the WENO scheme's ability to produce non-oscillatory results for discontinuous functions. In other schemes, treatment at discontinuities creates oscillatory results in the first derivative calculations. Note that these plots do not include the one row or column of error that does peak due to the discontinuity, which is unavoidable.

Table 5.2: 2D WENO Scheme for Upwind Direction $D_x^+\phi$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	6.368×10^{-5}	-	6.486×10^{-5}	-
161×161	3.848×10^{-6}	4.05	8.010×10^{-6}	3.02
321×321	2.398×10^{-7}	4.00	9.634×10^{-7}	3.06

Table 5.3: 2D WENO Scheme for Upwind Direction $D_x^-\phi$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	6.318×10^{-5}	-	8.004×10^{-5}	-
161×161	3.844×10^{-6}	4.04	8.508×10^{-6}	3.24
321×321	2.397×10^{-7}	4.00	1.027×10^{-6}	3.05

Table 5.4: 2D WENO Scheme for Upwind Direction $D_y^+\phi$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	1.114×10^{-4}	-	1.087×10^{-4}	-
161×161	6.800×10^{-6}	4.03	1.354×10^{-5}	3.00
321×321	4.164×10^{-7}	4.03	1.655×10^{-6}	3.03

Table 5.5: 2D WENO Scheme for Upwind Direction $D_y^-\phi$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	1.105×10^{-4}	-	1.157×10^{-4}	-
161×161	6.792×10^{-6}	4.02	1.380×10^{-5}	3.07
321×321	4.162×10^{-7}	4.03	1.690×10^{-6}	3.03

5.3 Total Variation Diminishing Runge-Kutta (TVD RK3) Method

Now that we have developed a higher-order accurate method to approximate ϕ_x, we can implement the TVD RK3 method to compute advection using a combination of

three Euler steps and two averaging steps

$$\begin{aligned}
\frac{\tilde{\phi}^{n+1} - \phi^n}{\Delta t} + u^n \phi_x^n + v^n \phi_y^n &= 0, \\
\frac{\tilde{\phi}^{n+2} - \tilde{\phi}^{n+1}}{\Delta t} + u^{n+1}\tilde{\phi}_x^{n+1} + v^{n+1}\tilde{\phi}_y^{n+1} &= 0, \\
\tilde{\phi}^{n+\frac{1}{2}} = \frac{3}{4}\phi^n + \frac{1}{4}\tilde{\phi}^{n+2}&, \\
\frac{\tilde{\phi}^{n+\frac{3}{2}} - \tilde{\phi}^{n+\frac{1}{2}}}{\Delta t} + u^{n+\frac{1}{2}}\tilde{\phi}_x^{n+\frac{1}{2}} + v^{n+\frac{1}{2}}\tilde{\phi}_y^{n+\frac{1}{2}} &= 0, \\
\phi^{n+1} = \frac{1}{3}\phi^n + \frac{2}{3}\tilde{\phi}^{n+\frac{3}{2}}&,
\end{aligned} \tag{5.6}$$

where v_x and v_y are the interface velocity components. We find ϕ_x and ϕ_y using our WENO schemes so that, if $v_x > 0$, we would use D_x^- and if $v_x < 0$, we would use D_x^+. Similarly, if $v_y > 0$, we would use D_y^- and if $v_y < 0$, we would use D_y^+. Since this is a third-order accurate method, it will be the limiting factor in our advection accuracy (remembering that WENO was up to fifth-order accurate and at minimum, third-order).

5.4 Accuracy of the Advection Scheme

To check the order of accuracy for advection, we use two tests. The first, is the vortex test where we begin with a circular level-set

$$\phi = (x - 0.5)^2 + (y - 0.75)^2 - 0.15^2, \tag{5.7}$$

on a domain $\Omega = [0,1]^2$ and deform it for 1 second using a vortex, defined by the conservative velocity field:

$$\begin{aligned} v_x(x,y) &= -\sin^2(\pi x)\sin(2\pi y), \\ v_y(x,y) &= \quad \sin^2(\pi y)\sin(2\pi x), \end{aligned} \tag{5.8}$$

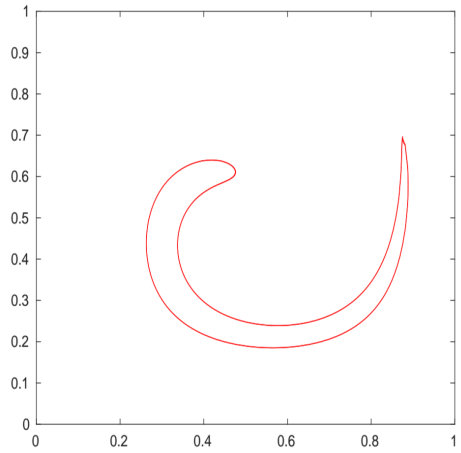
so that it takes the shape shown in Figure 5.6(a). We then run our numerical simulation for the same amount of time (1 second) with the reverse velocities $-v_x$ and $-v_y$ so that our zero level-set returns to its original position and shape. Then, we can compare the initial and final zero level-sets to ensure that our numerical interface is capable of undergoing dramatic deformations while still upholding the law of conservation of mass. In Figure 5.6(a), there is likely some mass lost at the tail of the level-set deformed by the vortex as it is stretched to thinner degrees. In Figure 5.6(b), we plot the initial and final zero level-sets to show that there is, indeed, mass loss. However, as we refine our grid, this mass loss becomes negligible as shown in Figure 5.7.

Our numerical scheme for advection maintains between third and fifth-order accuracy as one can see in Table 5.6. This is to be expected since we used a third-order accurate method for our time discretization and fifth-order accurate method for our spatial discretization.

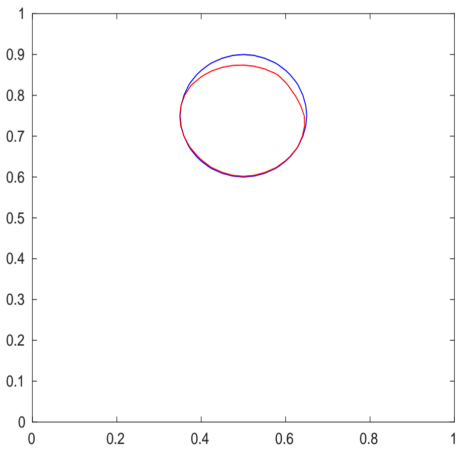
Table 5.6: Advection Vortex Test w/ $\Delta t = 0.5 \min(\Delta x, \Delta y)/|v|_{max}$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	1.352×10^{-4}	-	1.100×10^{-3}	-
161×161	5.966×10^{-6}	4.50	1.319×10^{-4}	3.04
321×321	2.307×10^{-7}	4.69	1.090×10^{-5}	3.60

To be thorough in our testing, we implemented a second test. Here, we advect a



(a) Level-set disfigured after vortex has acted on it for 1 second.



(b) Initial and final level-set after vortex has deformed it in one direction then deformed it in reverse direction.

Figure 5.6: We test for mass conservation by distorting the level-set using vortices acting in one direction, then reversing the vortices and comparing the difference between the initial and final zero level-sets. One can clearly see that mass has indeed been lost. We chose a low grid resolution of 41×41 to demonstrate this, but in fact, our program is very good at conserving mass as can be seen in Figure 5.7.

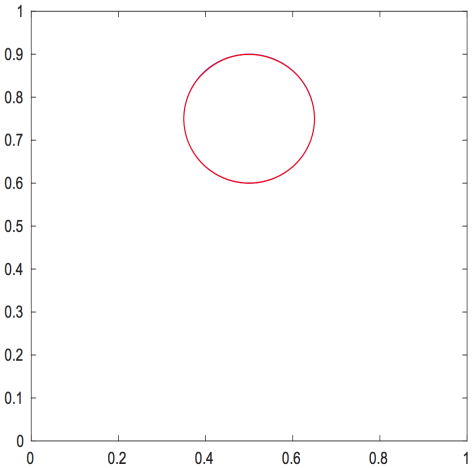


Figure 5.7: Here, we show the same plot as in Figure 5.6(b), at a higher grid resolution of 101×101. At this grid resolution, one cannot even distinguish the difference between the initial and final zero level-sets, proving that the mass loss is negligible for refined grids.

periodic level-set given by

$$\phi(x, y) = \cos(x - x_c)\sin(y - y_c), \tag{5.9}$$

in a domain $\Omega = [-\pi,\pi]^2$. We use velocities $v_x = 1$ and $v_y = 1$ and a final time $t_{\text{final}} = 0$, so that we know if the solution (x_c, y_c) is initially at (0,0), the final analytical solution will be at (1,1). This test proves our advection solver achieves third-order accuracy as can be seen in Table 5.7.

Table 5.7: Advection Test w/ $\Delta t = 0.5\min(\Delta x, \Delta y)/|v|_{max}$

Grid Resolution	L^1 - error	Order	L^∞ - error	Order
81×81	5.204×10^{-4}	-	2.054×10^{-5}	-
161×161	6.426×10^{-5}	3.05	2.536×10^{-6}	3.05
321×321	7.961×10^{-6}	3.03	3.151×10^{-7}	3.02

Chapter 6

Stefan Solver

Our first-order accurate Stefan solver calls upon all of the previously discussed functions to simulate crystal growth where the conserved variable u of interest is temperature. Thus, in terms of the heat equation, this diffusion can be thought of as conduction. For example, if you were to hold a metal rod so that one end was in a fire, eventually, through the diffusion of temperature in the metal rod, or the conduction of heat, the rod would burn your hand even if your hand was never near the fire (assuming the diffusion coefficient of the metal was large enough).

We consider an initial level-set describing ice in supercooled water, where supercooled means the water is below freezing point without solidification. We begin by setting the parameters: grid resolution $N \times N$, diffusion coefficients D_{in} and D_{out}, boundary conditions u_γ, initial temperature conditions T^0_{in} and T^0_{out}, and initial level-set ϕ^0. Since our temperature T will be discontinuous about the interface, initially, we must smooth them out using the diffusion function until a steady state is reached whereby T^0_{in} and T^0_{out} match at the interface and form a continuous temperature field. We can set $\Delta t = \Delta x$, since our Crank-Nicholson scheme allows it. To be safe, we execute our diffusion code for the same number of iterations as our iterative methods to ensure convergence.

Once, we have defined satisfactory initial conditions for temperature, we can enter our while loop that runs until the final time is reached. The pseudo code below shows the algorithm used to solve our Stefan problem.

```
while t < tfinal

    Third Order Extrapolation of T_in^n from Ω⁻ into Ω⁺
    Third Order Extrapolation of T_out^n from Ω⁺ into Ω⁻

    Compute velocities v_x and v_y

    In normal direction at interface:
       Constant Extrapolation of v_x into Ω⁺
       Constant Extrapolation of v_x into Ω⁻
       Constant Extrapolation of v_y into Ω⁺
       Constant Extrapolation of v_y into Ω⁻

    Compute new time step Δt = 0.5min(Δx, Δy)/ |v|_max;

    Find φ^{n+1} by advecting φ^n with interface velocity

    Compute anisotropic boundary conditions

    Diffusion of T_in^{n+1} using φ^{n+1}
    Diffusion of T_out^{n+1} using −φ^{n+1}

    Reinitialize φ^{n+1}

    t = t+dt;

end
```

For each time step, we begin by using our third-order extrapolation method to extrapolate the band of points T_{in} and T_{out} about the interface. Next, we are ready to calculate the interface velocity $\vec{v}$ using the gradients of temperature at the interface. We use constant extrapolation to make the interface velocity constant in the normal direction to the interface to ensure better behavior of the level-set. For stability purposes, we

recalculate the new time step, here, using $\Delta t = 0.5\min(\Delta x, \Delta y)/|v|_{\max}$. Next, we compute the new level-set ϕ^{n+1} by advecting the front with the computed interface velocity. Now, we can calculate the Gibbs-Thomson boundary conditions (6.1) (discussed later) to impose anisotropic conditions. Then, we call on our diffusion solver to solve in each domain Ω^- and Ω^+. We use the current extrapolated temperature field T^n_{in}, ϕ^{n+1}, D_{in} and our boundary conditions to compute T^{n+1}_{in} in Ω^-, and use the current extrapolated temperature field T^n_{out}, $-\phi^{n+1}$, D_{out} and our boundary conditions to compute T^{n+1}_{out} in Ω^+. Finally, we reinitialize our level-set. To reach the final time, we would repeat this process for the necessary amount of time steps.

To demonstrate some interesting examples of crystal growth, we refer to Gibou's paper [3] on numerically simulating dendritic solidification by imposing anisotropic diffusion. Anisotropy is the property of being directionally dependent, and when applied to simulate the surface tension at the interface, produces dendritic growth. We impose Gibbs-Thomson boundary conditions at the interface defined as

$$u_\gamma = u + \epsilon\kappa. \tag{6.1}$$

Here, κ is the curvature of the interface ($\kappa = \nabla \cdot \vec{n}$), and ϵ_c is the surface tension coefficient

$$\epsilon_c = d_0(1 - 15\epsilon\cos(4\alpha)), \tag{6.2}$$

where d_0 is the anisotropy coefficient, ϵ is the anisotropy strength, and α is the angle between the normal at the interface and the x-axis. This formula represents the standard four-fold anisotropy and allows us to create interesting geometries such as those shown in Figures 6.1 - 6.3. For each snowflake, we begin with an initial square zero level-set

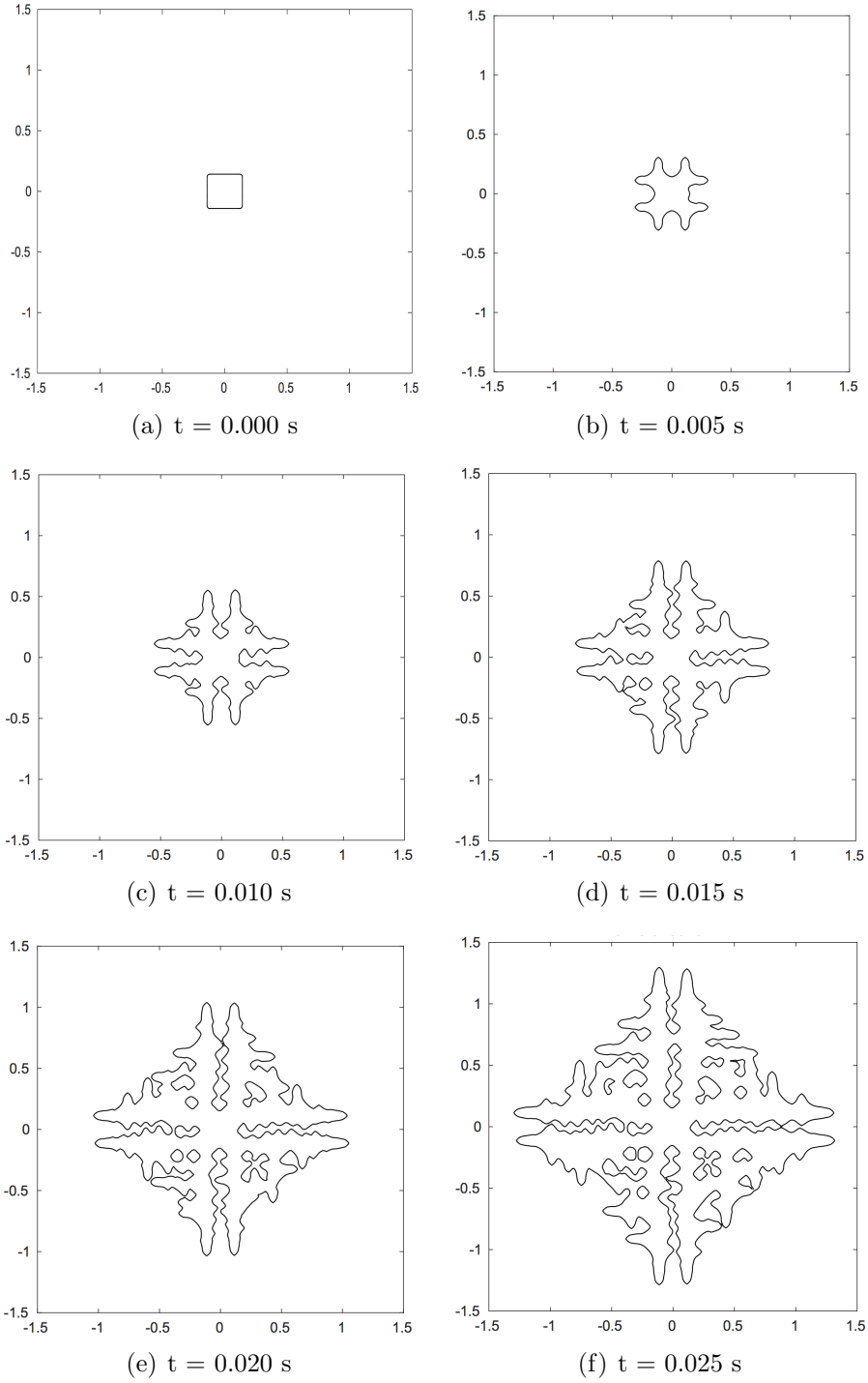
defined by

$$\phi = \max(|x|, |y|) - 0.1\sqrt{2} \tag{6.3}$$

in a domain $\Omega = [-1.5, 1.5]^2$ with grid resolution 200×200. For our time step, we use the restriction $\Delta t = 0.5\min(\Delta x, \Delta y)/|v|_{\max}$. We define both diffusion coefficients $D_{\text{in}} = D_{\text{out}} = 0.5$, and initial temperatures $u^0_{\text{in}} = 0$, and $u^0_{\text{out}} = -0.5$. Finally, to induce dendritic growth, we set the anisotropic feature $d_0 = 0.005$ and vary ϵ to create the three different snowflakes.

One will notice that as the dendrites grow, some merge into each other, which is why the final snowflakes in Figures 6.1(f), 6.2(f), and 6.3(f) have developed holes and gaps in the ice. As we discussed earlier, the level-set method was designed to handle these types of complex topological changes, and indeed, we are able to demonstrate its capabilities in these examples. We include Figure 6.4 to help observe this phenomenon and better visualize how these dendrites grow along preferred directions. One should also notice the lack of symmetry in the actual snowflake growth. That is to say, despite the symmetry of the grid and initial conditions, the dendrite growths vary on different sides of the initial square crystal. This is due to the anisotropic effects and the highly nonlinear unstable nature of this problem.

In Figure 6.5, we show the effects of varying the diffusion coefficient on snowflake growth. One will notice that a greater diffusion coefficient will increase the interface velocity so that the snowflake will grow faster.

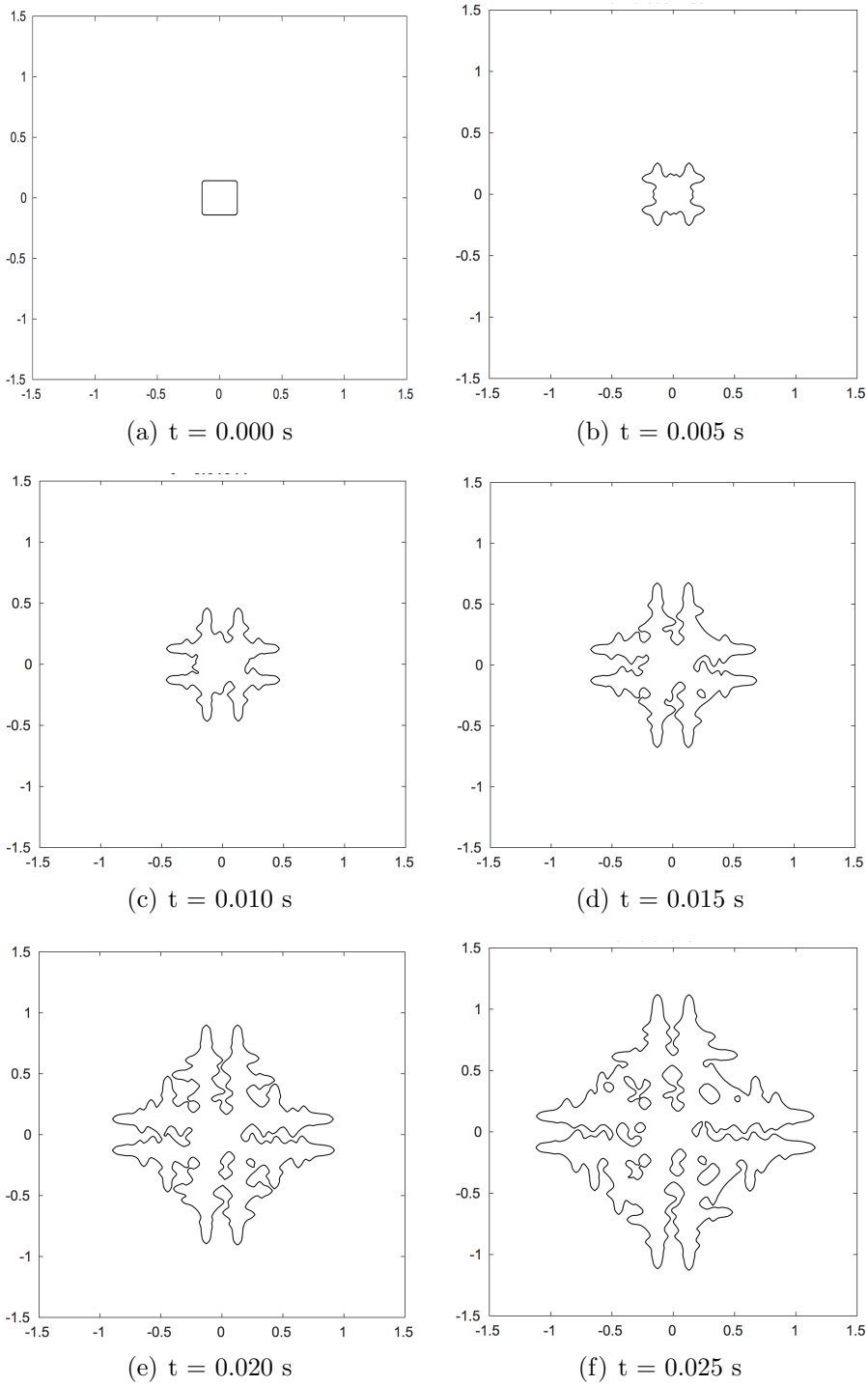


(a) t = 0.000 s (b) t = 0.005 s

(c) t = 0.010 s (d) t = 0.015 s

(e) t = 0.020 s (f) t = 0.025 s

Figure 6.1: Time lapse of snowflake growth on 200×200 grid with $\epsilon = 0.7$.

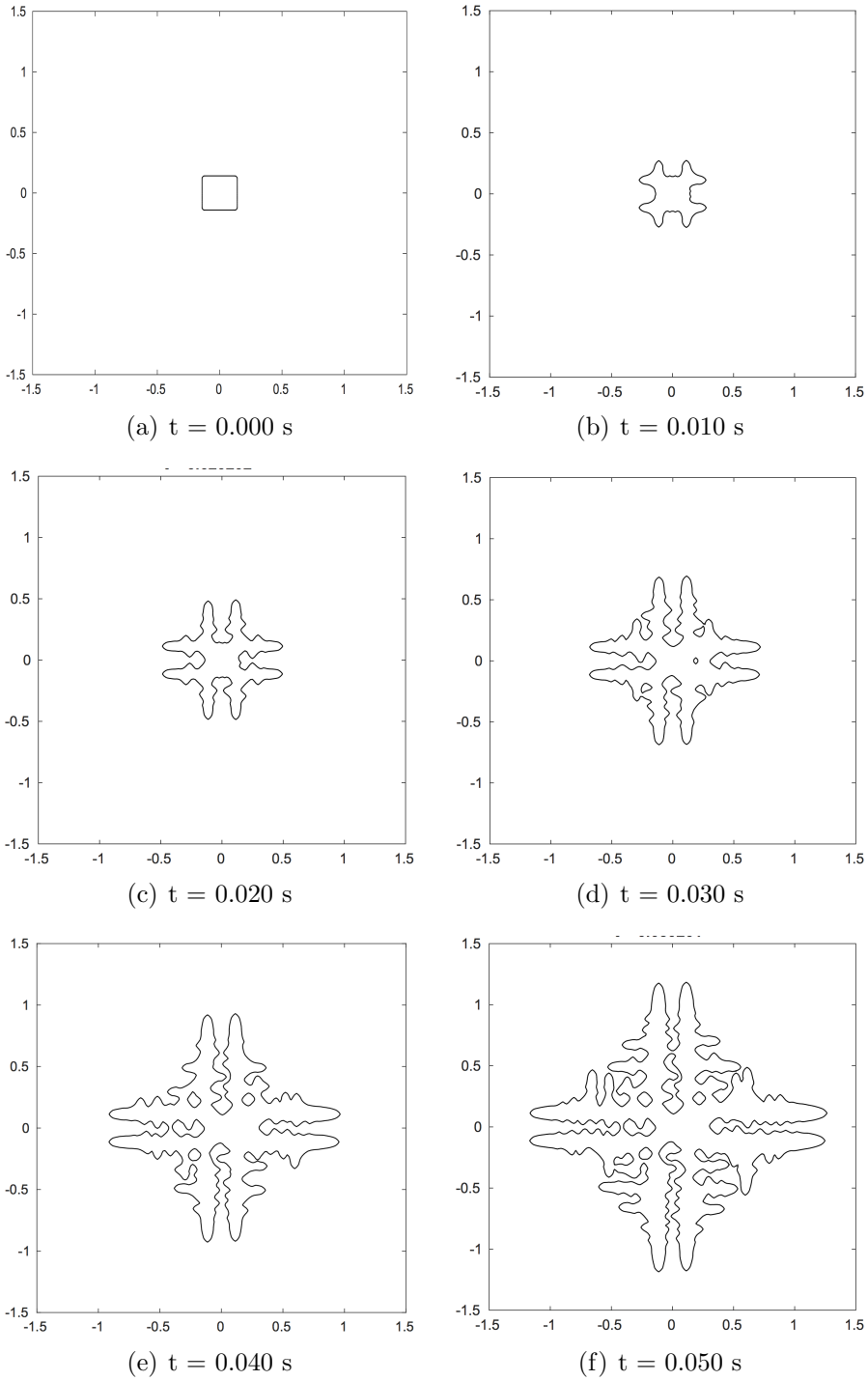


(a) t = 0.000 s (b) t = 0.005 s

(c) t = 0.010 s (d) t = 0.015 s

(e) t = 0.020 s (f) t = 0.025 s

Figure 6.2: Time lapse of snowflake growth on 200×200 grid with $\epsilon = 0.6$.



(a) t = 0.000 s (b) t = 0.010 s

(c) t = 0.020 s (d) t = 0.030 s

(e) t = 0.040 s (f) t = 0.050 s

Figure 6.3: Time lapse of snowflake growth on 200×200 with $\epsilon = 0.3$.

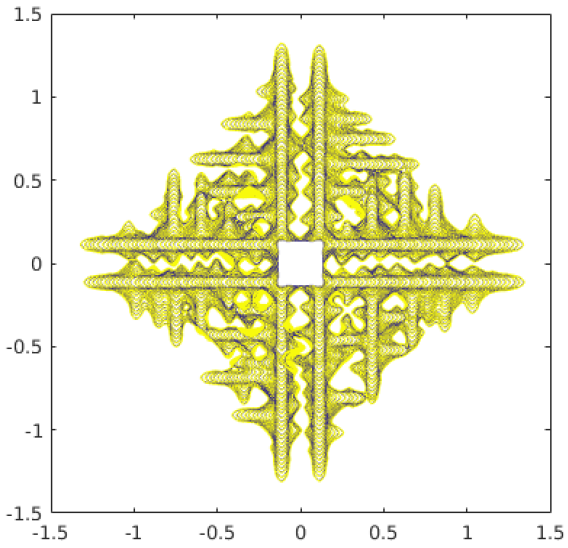
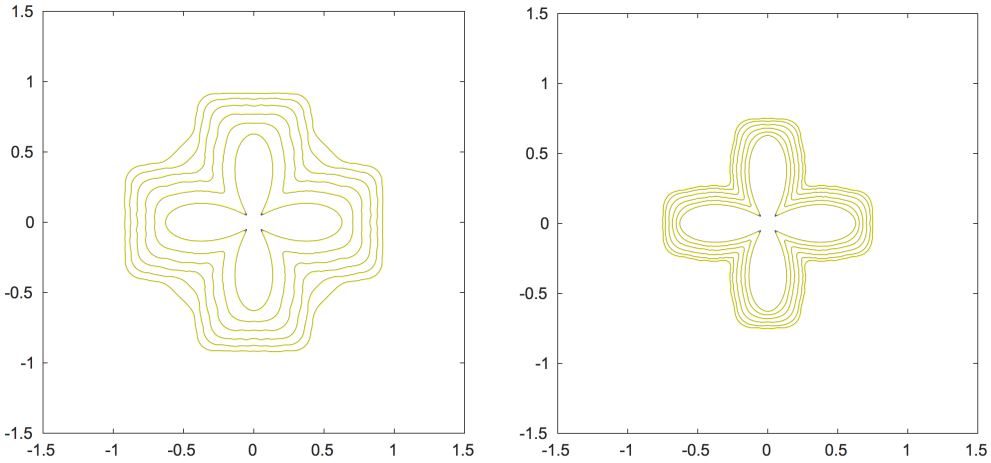
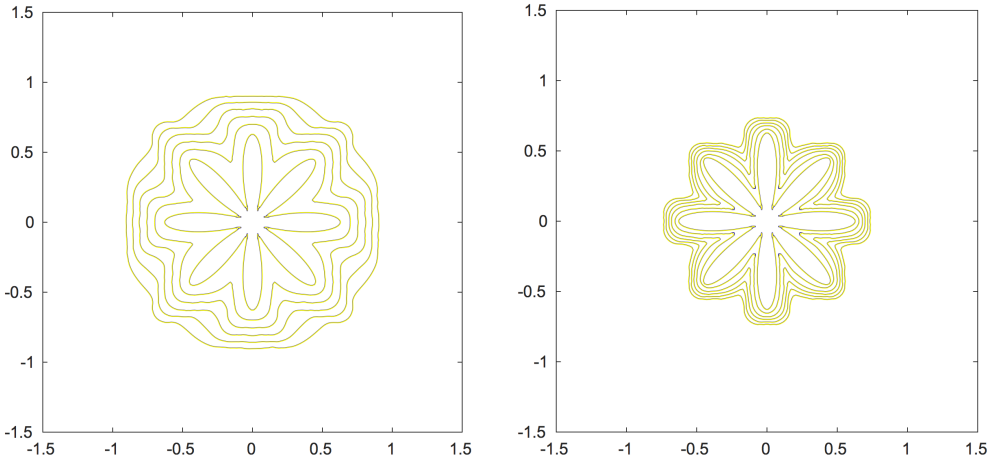


Figure 6.4: Here, we include many more time step renderings all plotted on the same figure. In this figure, it is highly apparent that the dendrites grow along preferred directions due to the anisotropic boundary conditions. One can clearly see how the holes begin to develop from dendrites growing and merging into each other. This is the same snowflake as the one shown in Figure 6.1 (i.e. Snowflake Growth A).



(a) D = 3 (b) D = 1



(c) D = 3 (d) D = 1

Figure 6.5: Effects of diffusion coefficient on snowflake growth. Here, we show the growth every 0.001 seconds for a final time of 0.05 seconds on a grid size 200×200 with the same supercooled system as the snowflakes and no anisotropic effects. We vary the diffusion coefficient from 1 to 3 for the two different level-set shapes to demonstrate how the interface velocity increases with the diffusion coefficient.

Chapter 7

Efficient Matlab Implementation

We have already discussed a few strategies used to make our code more powerful such as using sparse matrices to reduce memory storage. Now, we will discuss some of the steps taken to improve computational efficiency. There were two areas where significant improvements were made: the diffusion solver and the iterative methods. (Please note that in the following subsections, a 64-bit Linux Ubuntu OS was used with 62.8 GiB of memory and an Intel Xeon(R) CPU e5-1650 v4 @ 3.60GHz $\times$ 12 processor.

7.1 Building a Compact Diffusion Matrix

The original diffusion function created a linear equation $\boldsymbol{A}\vec{u}^{n+1} = \vec{f}\,(\vec{u}^{\,n}, BC^{n+1})$ for each grid point in the entire domain Ω, which equates to a diffusion matrix size of 40,000$\times$40,000 for a grid resolution of 200$\times$200. This means the program was creating identity equations at grid points that fell on the interface where the solution was known (i.e. $u_{i,j}^{n+1} = BC_{i,j}^{n+1}$), and at grid points in Ω^+ where the solution was invalid and, therefore, unused (i.e. $u_{i,j}^{n+1} = 0$). We used this method initially because it was easy to implement and seemed reasonably efficient for smaller grid resolutions. However, as we

began to refine our grid, this method was no longer effective.

To demonstrate the gain in efficiency, we include Table 7.1, which executes the diffusion program a total of 275 times to simulate 275 time steps. We use this number for comparison throughout the efficiency study so that we can make a reasonable comparison of the comprehensive Stefan solvers at the end. We used a circular level-set with a radius of 0.5 in a domain $[-1, 1]^2$ using various grid resolutions to show the impact of our improved efficiency. (However, keep in mind that as the grid resolution increases, the time step shrinks. Thus, as one refines the grid, many more time steps would be required to reach the same final time.) We included the time elapsed (in seconds) to execute each program, as well as the percent time reduction, which is calculated as (1 - New Time/Old Time). Notice, as the grid size increases, percent time reduction increases, as well, so that for a grid size 100×100, the new diffusion solver is 25.5 times faster, and with a grid resolution of 500×500, the new diffusion solver is 495 times faster than the original scheme. This increase in efficiency has to do with the growing amount of identity equations which our old solver had to compute.

Table 7.1: Diffusion Program Efficiency for Various Grid Resolutions

Grid Resolution	Old Time(s)	New Time(s)	% Time Reduction
100×100	30	1.16	96.08%
200×200	378	5.22	98.62%
300×300	2,025	12.05	99.41%
400×400	6,909	21.56	99.69%
500×500	17,552	35.42	99.80%

To avoid creating thousands of identity equations, we made $\vec{u}^{n+1}$ contain only points in Ω^- in our new diffusion solver. This required use of the ghost value method. Consider, for example, Figure 7.1.

For such a case, our old diffusion solver would create an implicit system of linear

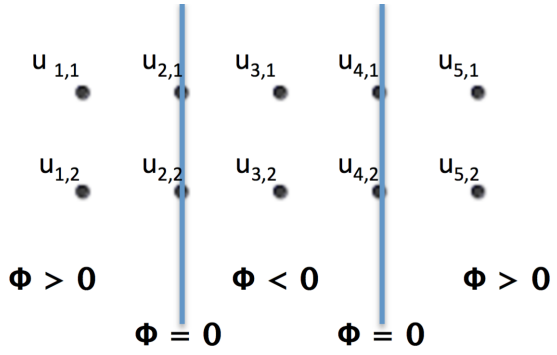


Figure 7.1: Consider this grid with ten points in a 2×5 arrangement. For this system, we have four points in Ω^+ that we do not want to compute, four points at the interface with known boundary condition values, and only two points that need to be solved for.

equations $\boldsymbol{A}\vec{u}^{n+1} = \vec{f}\,(\vec{u}^{\,n}, BC^{n+1})$ shown in matrix form below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{L} & \mathbf{C} & \mathbf{R} & 0 & 0 & 0 & \mathbf{B} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{A} & 0 & 0 & 0 & \mathbf{L} & \mathbf{C} & \mathbf{R} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,1}^{n+1} \\ u_{2,1}^{n+1} \\ u_{3,1}^{n+1} \\ u_{4,1}^{n+1} \\ u_{5,1}^{n+1} \\ u_{1,2}^{n+1} \\ u_{2,2}^{n+1} \\ u_{3,2}^{n+1} \\ u_{4,2}^{n+1} \\ u_{5,2}^{n+1} \end{bmatrix} = \begin{bmatrix} 0 \\ BC_{2,1}^{n+1} \\ f(u^n)_{3,1} \\ BC_{4,1}^{n+1} \\ 0 \\ 0 \\ BC_{2,2}^{n+1} \\ f(u^n)_{3,2} \\ BC_{4,2}^{n+1} \\ 0 \end{bmatrix}, \tag{7.1}$$

where we filled our RHS vector with zeroes for points outside of Ω^-, Dirichlet boundary conditions where applicable, and $f(u^n)$ to represent some **known** function of u^n. The non-trivial values have been highlighted so one can quickly identify the equations that are non-trivial. In this example, eight of the ten rows are identity equations. The new diffusion solver removes all these trivial rows so that we are left solving only for the two

unknowns:

$$\begin{bmatrix} \mathbf{C} & \mathbf{B} \\ \mathbf{A} & \mathbf{C} \end{bmatrix} \begin{bmatrix} u_{3,1}^{n+1} \\ u_{3,2}^{n+1} \end{bmatrix} = \begin{bmatrix} f(u^n)_{3,1} - \mathbf{L}\, BC_{2,1}^{n+1} - \mathbf{R}\, BC_{4,1}^{n+1} \\ f(u^n)_{3,2} - \mathbf{L}\, BC_{2,2}^{n+1} - \mathbf{R}\, BC_{4,2}^{n+1} \end{bmatrix}. \tag{7.2}$$

This matrix is stripped of all lines associated with the known values in the previously used $\vec{u}^{\,n+1}$. One can imagine this by considering the first line of the matrix. We do not want to compute $u_{1,1}^{n+1}$, so we strike the top row and leftmost column from the matrix as shown in Figure 7.2. This does not affect any other equation since all elements off the diagonal are zero. Next, we move to the second value in our vector: $u_{2,1}^{n+1}$. Here, we notice there is a non-zero element off the diagonal. When this occurs, we must implement the ghost value method so that our linear equation is still valid. Thus, we would move the value $\mathrm{L}u_{2,1}^{n+1}$ to the RHS vector by subtracting it.

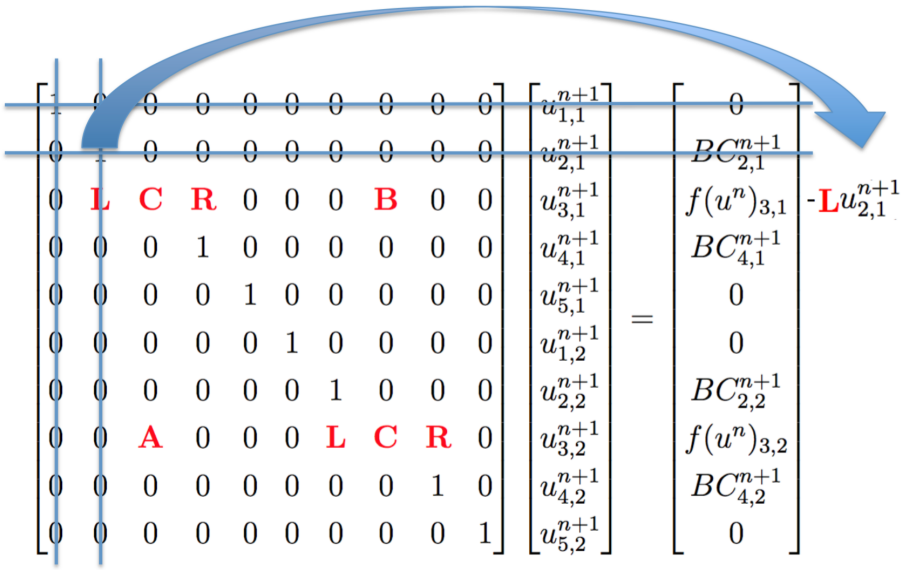


Figure 7.2: Ghost Point Added to the RHS to account for the element L no longer included in the matrix.

Another way to consider this is by looking at the equation itself outside of matrix

form. For the third line, we would end up with the linear equation:

$$\mathbf{L}\, BC_{2,1}^{n+1} + \mathbf{C}\, u_{3,1}^{n+1} + \mathbf{R}\, BC_{4,1}^{n+1} + \mathbf{B}\, u_{3,2}^{n+1} = f(u^n)_{3,1}. \tag{7.3}$$

Moving all our knowns to the RHS, we acquire this rearranged equation:

$$\mathbf{C}\, u_{3,1}^{n+1} + \mathbf{B}\, u_{3,2}^{n+1} = f(u^n)_{3,1} - \mathbf{L}\, BC_{2,1}^{n+1} - \mathbf{R}\, BC_{4,1}^{n+1}. \tag{7.4}$$

Note, Figure 7.2 is simply provided for a visual understanding of how one can transform the original 10×10 matrix into the new 2×2 matrix. However, in its actual implementation, the new diffusion program skips the generation of the identity equations, so that it only ever builds the concise compact matrix you see in the end, adding the ghost values to the RHS on the fly. This is how our new diffusion solver gains its efficiency over the old diffusion solver.

7.2 Computing First-Order Directional Derivatives

In this thesis, we develop higher-order accurate one-sided finite difference schemes called the WENO schemes. These sacrifice computational efficiency for accuracy, making them highly advantageous for a process like advection where they are only called upon six times, but become cumbersome in iterative methods.

In our Stefan solver, we use three iterative methods, which each require a minimum number of iterations to propagate the solution out sufficiently far. During these iterations, the solution will continue to converge to comparable accuracies, regardless of the accuracy of the directional derivatives. For all three snowflakes shown in Figures 6.1 - 6.3, we used 50 iterations of reinitialization; 160 iterations of constant extrapolation (i.e. 40 iterations to extrapolate v_x and v_y in both normal directions); and 80

iterations of third-order extrapolation (i.e. 40 iterations to extrapolate T_{in} and T_{out} in a band around the interface). Recall that these iterations occur at every time step, and with each iteration, we calculate all four directional derivatives. Thus, over the course of 275 time steps, we call upon our directional derivatives a grand total of 55,000 times for reinitialization, 176,000 times for constant extrapolation, and 88,000 times for third-order extrapolation. Thus, we require a cheaper method for computing directional derivatives which we developed in equation (2.4). Tables 7.2 - 7.4 demonstrate the efficiency gained when we use first-order schemes in place of the WENO schemes.

Table 7.2: Reinitialization Efficiency for Various Grid Resolutions

Grid Resolution	Old Time(s)	New Time(s)	% Time Reduction
100×100	635	9.39	98.52%
200×200	2,551	36.60	98.56%
300×300	5,764	79.64	98.62%
400×400	9,708	139.79	98.56%
500×500	15,156	221.99	98.54%

Table 7.3: Constant Extrapolation Efficiency for Various Grid Resolutions

Grid Resolution	Old Time(s)	New Time(s)	% Time Reduction
100×100	2,957	21.44	99.28%
200×200	11,462	80.49	99.30%
300×300	25,818	177.79	99.31%
400×400	49,207	321.36	99.35%
500×500	75,783	515.61	99.32%

Table 7.4: Third-Order Accuracy Efficiency for Various Grid Resolutions

Grid Resolution	Old Time(s)	New Time(s)	% Time Reduction
100×100	2,893	32.41	98.88%
200×200	11,497	123.50	98.93%
300×300	27,699	274.32	99.01%
400×400	46,730	484.58	98.98%
500×500	77,003	760.39	99.01%

Table 7.5 shows the time it would take for the diffusion and iterative methods of

the Stefan solver to execute 275 time steps for a circular level-set with radius 0.5 in a 200×200 grid to demonstrate the efficiency we have gained.

Table 7.5: Time to Make the Snowflake

Method	Old Time(s)	New Time(s)	% Time Reduction
Diffusion	378	5.22	98.62%
Reinitialization	2,551	36.60	98.56%
Constant Extrapolation	11,462	80.49	99.30%
Third-Order Extrapolation	11,497	123.50	98.93%
Total Time	25,888	245.81	99.05%

I have also included a simulation run at a higher grid resolution of 500×500 to demonstrate the significance of this speed enhancement. With the new methods, it takes 35 minutes to execute 275 time steps, whereas with the old methods, it takes three days.

Table 7.6: Time for 275 time steps at Higher Grid Resolution

Method	Old Time(s)	New Time(s)	% Time Reduction
Diffusion	17,552	35.42	99.80%
Reinitialization	15,156	221.99	98.54%
Constant Extrapolation	75,783	515.61	99.32%
Third-Order Extrapolation	77,003	760.39	99.01%
Total Time	254,444	2,055.23	99.19%

Chapter 8

Closing Thoughts

The Stefan problem is a complicated free boundary problem with no analytical solutions. Therefore, we developed a numerical method to allow us to analyze dendritic crystal growth. As we discussed in Section 4, our Stefan solver is first-order accurate overall because of the interface velocity calculation. Using quadratic extrapolation of the ghost values gives us a second-order accurate solution **and** second-order accurate derivative. In numerical analysis, this is called super convergence because it converges faster than one would expect. When we use linear extrapolation instead, we maintain the second-order accurate solution, but lose one order of accuracy in our gradient. However, this allows us to have symmetric discretization, which in turn enables us to use much faster solvers like PCG, whereas with non-symmetric discretization, we are forced to use slow solvers like Gauss-Seidel. With numerical methods, the coder must sometimes choose what to prioritize. In this thesis, we chose to prioritize solver efficiency over one order of accuracy because the non-symmetric discretization is simply too computationally expensive. As we refined the grid, it would have become computationally prohibitive, and grid refinement is unavoidable to capture dendritic structures. Thus, we chose to build a simpler, but faster model that would allow us to execute many more studies in

a shorter amount of time, as well as study higher grid resolutions.

We develop higher-order tools like third-order extrapolation and up to fifth-order WENO methods so that error does not compound throughout the solver and further reduce our precision. Further, our solver was built with higher-order dimensions in mind. Tools like the level-set can easily be extended to a three-dimensional problem by simply adding an extra index. Additionally, the entire code can readily be used with adaptive grid meshing as shown by Chen, Min, and Gibou in [2]. Thus, one advantage of the solver presented in this thesis, is that it provides a solid foundation for modeling the Stefan problem, that can be built upon easily.

We touched upon some reasons why the Stefan problem is a significant problem to study, earlier, and expand upon it here, as well as how our code could be used as a simple model for some of these important applications.

The word dendrites comes from the Greek word for tree, *dendron*. Metallurgists began using the term to describe alloy growth after observing tree-like branch structures in the freshly cast metal mixtures. An alloy is a mixture of metals like brass (copper and zinc) or mixture of metal and elements like steel or stainless steel (iron and carbon). Under a powerful microscope, an alloy appears to be made up of millions of tiny metallic snowflakes growing around and into each other. This is because as molten metal freezes, crystals grow faster along energetically favorable crystallographic directions.

Metallurgists discovered that this dendritic crystal growth played an enormous role in determining the alloy properties, including: softness, malleability, elasticity, load-carrying ability, heat and electricity conductance, and how easily it can be welded to another piece of metal - to name a few. All of these properties directly correlated to the shape, size, and speed of the dendritic growth. Thus, numerical simulations are a valuable tool for predicting how different conditions will affect these growth factors and give rise to the desired attributes [8]. For instance, smaller dendrites, which are

typically produced by rapid cooling cycles, generally lead to higher ductility. Whereas, long dendrites can provide a ready path for corrosive fluid to penetrate. Thus, one might use the numerical simulations to identify the ideal conditions that would minimize dendrite length.

One example of how this is useful is that it allows us to improve the recipe for materials like superalloys - high strength materials that can perform at extremely high temperatures. They are used in the construction of turbine engines, rocket engines, and power plants because the rule in energy efficiency is that the hotter you can burn fuel, the more energy you get out of it, and thus, the less fuel you need to use. This means superalloys increase energy efficiency and reduce pollution. Currently, superalloys can perform at temperatures over 650 degrees Celcius, and each year this number gets a little higher as they improve the recipe using simulations like the one presented in this thesis to determine the ideal conditions for producing superalloys.

Another important application for analyzing the Stefan problem is in medicine. The diffusion-reaction model has long been used to study cancer invasion [9] whereby we can observe the phase transition of healthy cells into cancer cells via the diffusion of oxygen and other nutrients. Diffusion-reaction models have allowed scientists and doctors to better understand the underlying mechanisms that govern the destruction of normal tissues by metastatic cancer, and map the interactions between tumor cells and normal cells at the tumor-host interface which significantly influences the progression of invasive cancer. Thus, numerical simulations can give us valuable insight into cancer cell growth and what can be done to slow down and treat metastatic cancer invasion. Note, this model does not necessarily display dendritic growth behavior. However, since we know our solver is capable of modeling the more complicated dendritic growth, we know that it is well equipped to study most diffusion-dominated problems which are characterized by simpler growth behaviors.

Chapter 9

References

[1] F. Gibou, R. Fedkiw, L.Cheng, and M. Kang, A second-order-accurate symmetric discretization of the poisson equation on irregular domains, *Journal of Computational Physics*, 205-227 (2000).

[2] C. Min and F. Gibou, A second-order-accurate level-set method on non-graded adaptive cartesian grids, *Journal of Computational Physics*, 200-321 (2006).

[3] F. Gibou, R. Fedkiw, R. Caflisch, and S. Osher, A level-set approach for the numerical simulation of dendritic growth, *Journal of Scientific Computing*, 183-199 (2002).

[4] M. Sussman, P. Smereka, and S.Osher, A level set approach for computing solutions in incompressible two-phase flow, *Journal of Computational Physics*, 146-159 (1994).

[5] T. Aslam, A partial differential equation approach to multidimensional extrapolation, *Journal of Computational Physics*, 349-355 (2004).

[6] X. Liu, S. Osher, T. Chan, Weighted essentially non-oscillatory schemes, *Journal of Computational Physics*, 200-212 (1994).

[7] H. Chen, C. Min, F. Gibou, A numerical scheme for the Stefan problem on adaptive Cartesian grids with supralinear convergence rate, *Journal of Computational Physics*, 5803-5818 (2009).

[8] W.L. George and J.A. Warren, A parallel 3D dendritic growth simulator using the phase-field method, *Journal of Computational Physics*, 264-283 (2002).

[9] R.A. Getenby and E.T. Gawlinski, A reaction-diffusion model of cancer invasion, *Journal of Cancer Research*, 5745-5753 (1996).