Level Set Methods and Sloshing Problems

Geng Tian

A thesis submitted in partial fulfillment of

the requirement of the degree of

Master of Applied Science (in Mathematics and Computing Science)

Saint Mary's University

Halifax, Nova Scotia

September 6, 2005

To My Family

Abstract

Level set methods are powerful numerical techniques for tracking the motion of an interface. Many applications arise in such areas as fluid flow simulations, medical science, and image processing. In fluid flow simulations, tracking the interface between two fluid flow phases is often difficult. Among the mathematical models that can be used to analyze fluid flow are the shallow water equations and Navier-Stokes equations. An important class of fluid flow problems is known as sloshing problems. These problems are concerned with the sloshing of a fluid in a tank, and they arise in the automotive, aerospace, and ship-building industries. In this thesis we consider the modelling of sloshing problems using shallow water equations and Navier-Stokes equations. Whereas the shallow water equations include a function that models the fluid interface, the Navier-Stokes equations do not. In this latter case, however, one can use the level set approach to track the fluid interface. Given the fluid velocity as obtained from the Navier-Stokes equations, one can use it to evolve the interface using the level set approach. We develop a MATLAB based implementation and provide numerical results to demonstrate this approach.

Acknowledgements

It is a pleasure to thank many people who made this thesis possible.

I would like to gratefully acknowledge my senior supervisors, Professor Paul Muir (Saint Mary's University) and Professor Raymond Spiteri (University of Saskatchewan). With their enthusiasm, their inspiration, and their great efforts to explain things clearly and simply, they helped to make mathematics fun for me. Throughout my graduate studies, they provided encouragement, sound advice, good teaching, and lots of good ideas. I would have been lost without them.

I am also grateful to my thesis examining committee, Professor Patrick Keast (Dalhousie University), Professor Walt Finden (Saint Mary's University), and Professor Hai Wang (Saint Mary's University) for their very kind assistance with writing this thesis, giving wise advice, and so on.

And I would like to thank the Faculty of Graduate Studies and Research at Saint Mary's University for their support in the form of scholarships.

I am grateful to the faculty and staff at the department of Mathematics and Computing Science in Saint Mary's University, for helping the department run smoothly and for assisting me in many different ways.

I am indebted to my many student colleagues for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Stuart Crosby, Hui Xu, Rong Wang, Hong Zhao, Zhenyan Sun, Yasushi Akayama, Wei Yu, Fan Luo, Yan Ma, and Arun Yadav.

I would also like to thank all my friends for helping me get through the difficult times, and for all the emotional support, entertainment, and caring they provided. A special thanks to Paul Ransom, Gaetan Lang, and Travis Coady.

I wish to thank my entire extended family for providing a loving environment for me.

Lastly, I would like to thank my parents, Wenmin Tian and Hali An, my sister and her husband, Jing Tian and Elvis Essavi. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis we consider the application of level set methods for the tracking of the fluid interface for a class of fluid flow problems known as sloshing problems.

Liquid in a basin or tank can flow back and forth in standing waves at discrete natural frequencies. This phenomenon is called *sloshing*. Applications arise in, for example, the automotive, aerospace, and ship-building industries. In ships, sloshing loads can cause cracks and weld-line failures in sheet metal compartments. Sloshing is, therefore, a major consideration for oil tankers, cargo ships, and cruise ships. Sloshing of liquids in tanks has received the attention of many researchers over recent decades. The sloshing phenomenon is very important in the design of a liquid-filled tank. The sloshing problem can be critical in a partially filled tank. The sloshing of the liquid can increase the dynamic pressure on the tank sides and bottom; violent sloshing creates impact that can cause serious damage to the tank [15].

The numerical solution of a sloshing problem first requires the development of a mathematical model, i.e., a set of partial differential equations, that describe flow of the fluid within the tank, with possibly an explicit consideration of the motion of the interface. When the motion of the interface is small compared to the depth of the fluid, a family of equations known as the shallow water equations can be used as a mathematical model. For more general contexts, the well-known Navier-Stokes equations, provide a rich mathematical model for treating many fluid flow problems, including sloshing problems. In this latter case, only the velocity of the fluid is explicitly considered by the model. However, as we consider in this thesis, it is possible to use the velocity results from the Navier-Stokes equations to evolve the interface using the level set approach. The interface is represented as the zero level set or contour of a function called the level set function. The level set equation incorporates the fluid velocity to describe the evolution of the zero level set of the level set function, thus allowing us to track the fluid interface in the context of the numerical modelling of a sloshing problem based on the Navier-Stokes equations. To our knowledge, no one has attempted to model sloshing using level set methods.

In this thesis, we consider the numerical solution of Navier-Stokes equations and the numerical solution of level set equations. In Chapter 2 we provide a survey of level set methods, and in Chapter 3, we describe the recently developed level set toolbox that provides a MATLAB [6] based suite of numerical methods for the solution of level set equations [13]. We provide several examples to demonstrate some of the capabilities of this toolbox. Chapter 4 is devoted to a discussion of the Navier-Stokes equations and their numerical solution. Part of the work undertaken in this thesis is a MATLAB based implementation of a collection of numerical methods for the treatment of Navier-Stokes equations. We include the source code in the Appendix. Chapter 4 describes our implementation and provides results for a sample fluid flow problem. In Chapter 5, we begin with a review of the literature on the numerical solution of sloshing problems and related fluid flow problems. We then consider the numerical treatment of a sloshing problem modelled using the shallow water equations. The final part of this chapter considers the modelling of a sloshing problem by the Navier-Stokes equations, with the tracking of the fluid interface handled by the level set approach, the implementation of which is based on the level set toolbox; numerical results are provided. We close in Chapter 6 with our conclusions and suggestions for future work.

Chapter 2

Overview of Level Set Methods

2.1 Introduction

In this chapter we provide an overview of level set methods. This material is largely drawn from [19], (Page 3-Page 67).

In a variety of phenomena, we want to track the motion of an interface. Such phenomena can occur, for example, in fluid mechanics, material science, medical science, combustion, meteorology, control theory, and image processing. An interface (or front) is a boundary between two regions, which we call inside and outside.

In one dimension, suppose we separate the real line into three parts using the points $x = -1$ and $x = 1$ as boundaries. We define $\Omega^- = (-1, 1)$ as the inner region of the domain and $\Omega^+ = (-\infty, -1) \cup (1, +\infty)$ as the outer region of the domain. The points $x = -1$ and $x = 1$ define the interface, $\partial\Omega$, between the two domains. An implicit interface representation is one in

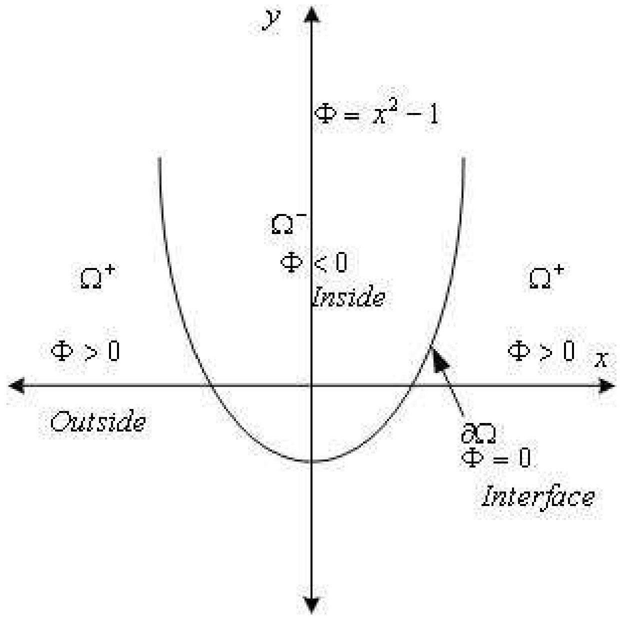

Figure 2.1: Implicit function $\phi(x) = x^2 - 1$ defining the region Ω^- and Ω^+ as well as the boundary $\partial\Omega$, adapted from [19], page 4.

which the points belonging to the interface are implicitly defined by some isocontour function. For example, the zero isocontour of $\phi(x) = x^2 - 1$, the set of zeroes of $\phi(x)$, is precisely $\partial\Omega = \{-1, 1\}$. These definitions are shown graphically in Figure 2.1.

In two dimensions, in order to ensure there are clearly defined interior and exterior regions, we define an interface to be a simple closed curve. For example, consider $\phi(x, y) = x^2 + y^2 - 1$, where the interface defined by the $\phi(x, y) = 0$ isocontour. This is the unit circle defined by $\partial\Omega = \{(x, y) : \sqrt{x^2 + y^2} = 1\}$. The interior region is $\Omega^- = \{(x, y) : \sqrt{x^2 + y^2} < 1\}$, and the exterior region is $\Omega^+ = \{(x, y) : \sqrt{x^2 + y^2} > 1\}$. These regions are shown in Figure 2.2.

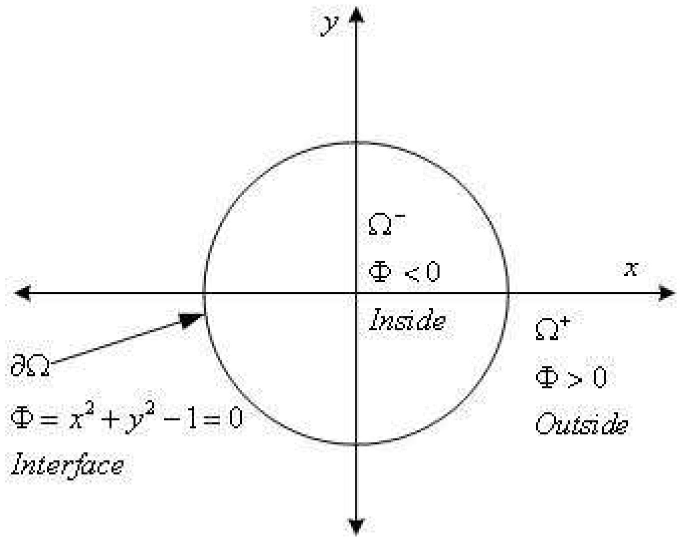

Figure 2.2: Implicit representation of the curve $x^2 + y^2 = 1$, adapted from [19], page 5.

In three dimensions, as an example, consider $\phi(x, y, z) = x^2 + y^2 + z^2 - 1$, where the interface defined by the $\phi(x, y, z) = 0$ isocontour. This is the unit sphere defined by $\partial\Omega = \{(x, y, z) : \sqrt{x^2 + y^2 + z^2} = 1\}$. The interior

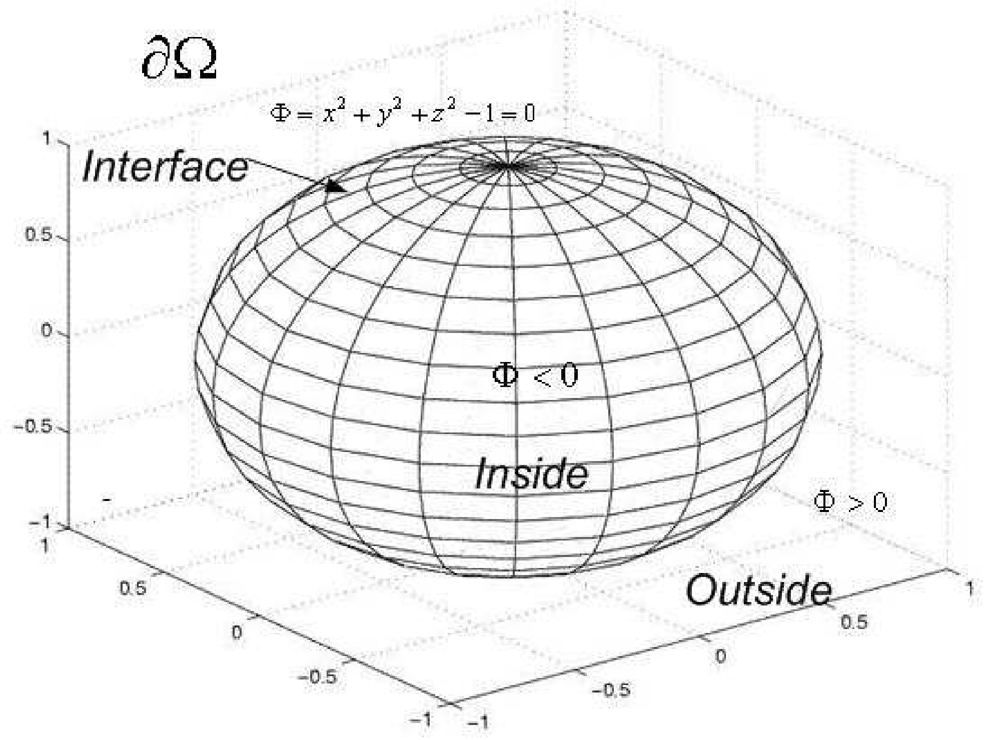

Figure 2.3: Implicit representation of the surface $x^2 + y^2 + z^2 = 1$.

region is $\Omega^- = \{(x, y, z) : \sqrt{x^2 + y^2 + z^2} < 1\}$, and the exterior region is $\Omega^+ = \{(x, y, z) : \sqrt{x^2 + y^2 + z^2} > 1\}$. These regions are shown in Figure 2.3.

Note that the interface itself is not explicitly available. It is given as the isocontour of the implicit function ϕ and thus will have to be interpolated from the set of data points where the implicit function ϕ is defined. Let $\mathbf{x} = (x, y)^T$; in the context of a numerical computation, $\phi(\mathbf{x})$ will be represented by a discrete set of values associated with data points distributed throughout the problem domain. The set of data points is called a *grid*; for example, uniform Cartesian grids are defined as $\{(x_i, y_j) : 1 \leq i \leq m, 1 \leq j \leq n\}$. We assume $x_1 < \cdots < x_i < x_{i+1} < \cdots < x_m$ and $y_1 < \cdots < y_j < y_{j+1} < \cdots < y_n$. We set $\Delta x = x_{i+1} - x_i$, $\Delta y = y_{j+1} - y_j$. By definition, Cartesian grids imply a rectangular domain $D = [x_1, x_m] \times [y_1, y_n]$. Because ϕ is only important near the interface, we can optimize the implicit representation by only storing a subset of a uniform Cartesian grid, discarding grid points that are not sufficiently near the interface. If we do not know the location of any of the points on the interface, that is, data points $\mathbf{x}$ where $\phi(\mathbf{x}) = 0$, interpolation is needed. The isocontour that includes the points $\mathbf{x}$ has to be determined from the interpolant using a contour plotting routine. Such routines are normally available in standard software libraries and in problem solving environments like MATLAB [6].

2.2 Distance Functions and Signed Distance Functions

Let $\mathbf{x} = [x_1, x_2, \cdots, x_n]^T$. We define a distance function $d(\mathbf{x}) = \min(|\mathbf{x} - \mathbf{x}_I|)$ for all $\mathbf{x}_I \in \partial\Omega$. Thus $d(\mathbf{x}) = 0$ on the interface where $\mathbf{x} \in \partial\Omega$. When $\mathbf{x} \notin \partial\Omega$ and $\mathbf{x}_c$ is the closest point on the interface to $\mathbf{x}$, then $d(\mathbf{x}) = \sqrt{(x_1 - x_{c,1})^2 + \cdots + (x_n - x_{c,n})^2}$, and then

$$\nabla d(\mathbf{x}) = \begin{bmatrix} \frac{d}{dx_1} d(\mathbf{x}) \\ \vdots \\ \frac{d}{dx_n} d(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \frac{2(x_1 - x_{c,1})}{\sqrt{(x_1-x_{c,1})^2 + \cdots + (x_n - x_{c,n})^2}} \\ \vdots \\ \frac{1}{2} \frac{2(x_n - x_{c,n})}{\sqrt{(x_1-x_{c,1})^2 + \cdots + (x_n - x_{c,n})^2}} \end{bmatrix}.$$

Then $|\nabla d(\mathbf{x})| = \left(\frac{(x_1 - x_{c,1})^2}{(x_1-x_{c,1})^2 + \cdots + (x_n - x_{c,n})^2} + \cdots + \frac{(x_n - x_{c,n})^2}{(x_1-x_{c,1})^2 + \cdots + (x_n - x_{c,n})^2} \right)^{\frac{1}{2}} = 1$. We define a *signed distance function* $\phi(\mathbf{x})$ such that $\phi(\mathbf{x}) = -d(\mathbf{x})$ in the interior region Ω^-, $\phi(\mathbf{x}) = d(\mathbf{x})$ in the exterior region Ω^+, and $\phi(\mathbf{x}) = 0$ on the boundary $\partial\Omega$. Thus a signed distance function is positive on the exterior, negative on the interior, and zero on the boundary. Since $|\nabla d(\mathbf{x})| = 1$, we also have $|\nabla \phi(\mathbf{x})| = 1$.

Given a point $\mathbf{x}$, $\phi(\mathbf{x})$ is the signed distance to the closest point on the interface. We can therefore trace from $\mathbf{x}$ along the normal to the interface at this closest point in order to find the coordinates of this closest point. That is, the point on the interface closest to $\mathbf{x}$ is given by $\mathbf{x}_c = \mathbf{x} - \phi(\mathbf{x})\mathbf{N}$, where $\mathbf{N}$ is the local unit normal at $\mathbf{x}$; i.e., $\mathbf{N} = \frac{\nabla\phi}{|\nabla\phi|}$ since $\nabla\phi(x)$ points in the direction of the normal to the surface at $\mathbf{x}$. Because $|\nabla\phi(\mathbf{x})| = 1$ for a signed distance function ϕ, we have $\mathbf{N} = \nabla\phi$.

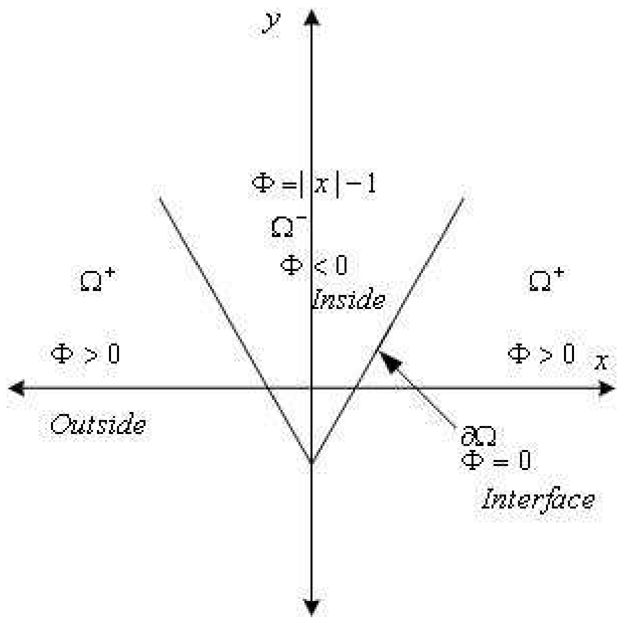

Figure 2.4: Implicit function $\phi(x) = |x| - 1$ defining the region Ω^- and Ω^+ as well as the boundary $\partial\Omega$, adapted from [19], page 20.

Let us now consider an example in one dimension. We previously used $\phi(x) = x^2 - 1$ as an implicit representation of $\partial\Omega = \{-1, 1\}$. A signed distance function representation of these points, $\phi(x) = |x| - 1$, is shown in

Figure 2.4. The signed distance function $\phi(x) = |x| - 1$, gives the same boundary $\partial\Omega$, interior region Ω^-, and exterior region Ω^+.

In two dimensions, the implicit function $\phi(x, y) = x^2 + y^2 - 1$ is replaced by the signed distance function $\phi(x, y) = \sqrt{x^2 + y^2} - 1$, and the unit circle is represented by $\partial\Omega = \{(x, y) : \sqrt{x^2 + y^2} = 1\}$. In three dimensions, the implicit function $\phi(x, y, z) = x^2 + y^2 + z^2 - 1$ is replaced by the signed distance function $\phi(x, y, z) = \sqrt{x^2 + y^2 + z^2} - 1$, and the unit sphere is represented by $\partial\Omega = \{(x, y, z) : \sqrt{x^2 + y^2 + z^2} = 1\}$.

2.3 Level Sets

At a given time t, the *zero level set* of the evolving function, $\phi(\mathbf{x}(t), t)$ is the set of points $\mathbf{x}(t)$, such that

$$\phi(\mathbf{x}(t), t) = 0,$$

where in two dimensions, for example,

$$\mathbf{x}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}.$$

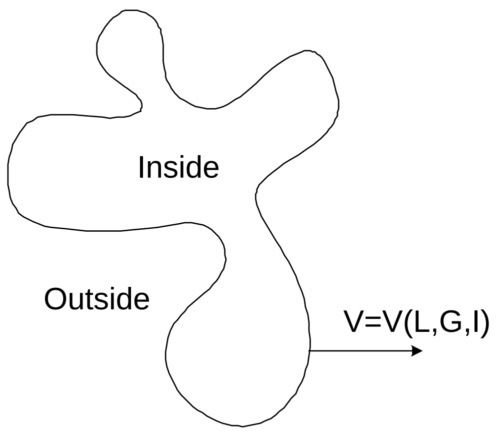

Figure 2.5: Curve propagating with velocity $\mathbf{V}$ in normal direction, adapted from [21].

2.3.1 The Velocity Function

Imagine a closed curve propagating with velocity $\mathbf{V}$ in the direction normal to itself. The velocity function $\mathbf{V}$, which may depend on many factors, can be written as: $\mathbf{V} = \mathbf{V}(L, G, I)$. See Figure 2.5, where L represents local information (e.g., curvature and normal direction), G represents global information (e.g., integrals along the front, heat diffusion), and I represents information that is independent of the shape of the front (e.g., an underlying fluid velocity that transports the front).

There are many ways to define the velocity function. For example, consider a velocity function $\mathbf{V}$ that depends only on the local curvature κ of the curve, that is, $\mathbf{V} = \mathbf{V}(\kappa)$, where the curvature κ is defined to be

$$\kappa = \nabla \cdot \mathbf{N}, \tag{2.1}$$

where $\nabla \cdot$ is the divergence operator, and $\mathbf{N}$ is the normal direction, $\mathbf{N} = \frac{\nabla\phi}{|\nabla\phi|}$, where, for example, in two dimensions, $\nabla\phi = \begin{pmatrix} \phi_x \\ \\ \phi_y \end{pmatrix}$ and $|\nabla\phi| = \sqrt{(\phi_x^2 + \phi_y^2)}$. $\nabla\phi$ is perpendicular to the isocontours of ϕ and points in the direction of increasing ϕ. From (2.1) we get, after simplification,

$$\kappa = \frac{\phi_{xx}\phi_y^2 - 2\phi_{xy}\phi_x\phi_y + \phi_{yy}\phi_x^2}{(\phi_x^2 + \phi_y^2)^{3/2}}. \tag{2.2}$$

2.3.2 The Level Set Equation

Level set methods rely on two central embeddings: the first is the embedding of the interface as the zero level set of a function called the evolving or level set function ϕ. The level set function is evolved in time by the level set equation. The second embedding is the embedding of the interface's velocity within the level set equation.

We can thus link the two central embeddings through a time-dependent initial value PDE. At any time, t, the interface is given by the zero level set of the time-dependent level set function $\phi(\mathbf{x}(t), t)$, where $\phi(\mathbf{x}(t), t)$ evolves according to the equation,

$$\phi_t + \mathbf{V} \cdot \nabla \phi = 0, \tag{2.3}$$

with a given initial condition, $\phi(\mathbf{x}(0), 0)$. This is the level set equation given in [20]. Equation (2.3) describes the time evolution of the level set function ϕ, under the influence of the interface velocity $\mathbf{V}$. Note that the velocity term in equation (2.3) may be dependent on external sources. For example, the interface velocity may be obtained by solving the two-phase Navier-Stokes equations, where the $\phi(\mathbf{x}(t), t) = 0$ isocontour represents the interface between two different phases. Generally, the interface velocity involves both space and time, so we can write $\mathbf{V}$ as $\mathbf{V}(\mathbf{x}(t), t)$.

In the area of combustion dynamics modelling, there is an equation called the G-equation that is of the form

$$G_t + \mathbf{V} \cdot \nabla G = 0,$$

where the $G(\mathbf{x}(t), t) = 0$ isocontour represents the reaction surface of an evolving flame front implicitly. The G-equation is obviously equivalent to the level set equation (2.3), and some researchers have begun to use level set methods to find numerical solutions of combustion problems [19], (page 26).

Now we consider an example of interface motion for a velocity field $\mathbf{V}$

that depends on the level set function ϕ. In two dimensions, the velocity $\mathbf{V}$ can be written as $\mathbf{V} = V_n\mathbf{N} + V_t\mathbf{T}$, where V_n is normal velocity, V_t is tangential velocity, and $\mathbf{T}$ is tangent vector. Since $\mathbf{T} \cdot \nabla\phi = 0$, (recall $\nabla\phi$ is in the direction of the normal; see discussion in Section 2.3.1), the level set equation becomes

$$\phi_t + V_n\mathbf{N} \cdot \nabla\phi = 0.$$

Because $\mathbf{N} \cdot \nabla\phi = \frac{\nabla\phi}{|\nabla\phi|} \cdot \nabla\phi = |\nabla\phi|$, we have

$$\phi_t + V_n|\nabla\phi| = 0. \tag{2.4}$$

Continuing our example from the previous section, if we substitute $V_n = -b\kappa$ for some constant b into (2.4), we get

$$\phi_t = b\kappa|\nabla\phi|. \tag{2.5}$$

Generally the level set equation is a hyperbolic PDE. However, in this example of interface motion because of the dependence of κ on ϕ as defined in equation (2.2), the $b\kappa|\nabla\phi|$ is a *parabolic* term, and $\nabla\phi$ can be discretized with a central (or symmetric) difference, rather than a one-sided or upwind scheme, as is normally required for a *hyperbolic* PDE. We consider this further in Section 2.4.

From (2.1) and (2.5), we have

$$\phi_t = b\Delta\phi. \tag{2.6}$$

To see this, recall that the level set equation is $\phi_t + V_n|\nabla\phi| = 0$. In our example $V_n = -b\kappa$ where $\kappa = \nabla \cdot \mathbf{N} = \nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}$. Therefore the level set equation becomes

$$\phi_t - \left(b\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}\right)|\nabla\phi| = 0 \Rightarrow \phi_t - b\nabla \cdot \nabla\phi = 0 \Rightarrow \phi_t = b\Delta\phi$$

because $\nabla \cdot \nabla\phi = \nabla \cdot \begin{pmatrix} \phi_x \\ \phi_y \end{pmatrix} = \phi_{xx} + \phi_{yy} = \Delta\phi$.

2.3.3 Reinitialization of the Level Set Function

The level set function is initially a signed distance function. After the advection of the interface using the level set equation, it is uncommon for the level set function to remain a signed distance function. This means that the level set function needs to be reinitialized (i.e., reconstructed so that it again becomes a signed distance function) at regular time intervals. A simple and accurate technique is to calculate how far each grid point is from the zero isocontour of the level set directly. This technique is quite expensive in practice, and as such it cannot be used in real world examples or with schemes that require frequent reinitialization. The crossing time gives the distance for the grid point. If we evolve the interface in both the normal and negative normal direction at the same time, we obtain the following equation, known as the *reinitialization equation*,

$$\phi_t = S(\phi_0)(1 - |\nabla(\phi)|),$$

where $\phi_0 = \phi(\mathbf{x}, 0)$, $S(\phi_0)$ is a smoothed (i.e., not a step function with a discontinuous derivative) sign function that is positive and approximately equal to 1 in Ω^+, negative and approximately equal to -1 in Ω^-, and 0 on the interface. This function has the form $S(\phi_0) = \frac{\phi_0}{\sqrt{\phi_0^2+\epsilon^2}}$, where $0 < \epsilon < 1$. For example, if $\phi_0 = x_1^2 + x_2^2 - 1$, then $S(\phi_0) = \frac{x_1^2+x_2^2-1}{\sqrt{(x_1^2+x_2^2-1)^2+\epsilon^2}}$. As long as ϕ is relatively smooth and the initial data are somewhat balanced across the interface, this method works well. See [19], (page 67), for further details.

2.4 Spatial Derivative Approximations

In general the level set equation (2.3) is a first-order hyperbolic PDE that is related to the well-known hyperbolic conservation laws. Such equations can be difficult to treat computationally; one has to be careful in order to keep the numerical computation from becoming unstable. In this section, we consider techniques for approximating the spatial derivatives arising in (2.3), namely $\nabla\phi(\mathbf{x}(t),t)$. We consider the two-dimensional case; however generalization to three-dimensions is certainly possible. We define $\phi_{i,j}$ to be the discrete grid function, i.e., $\phi_{i,j} = \phi\left(\begin{pmatrix} x_i \\ y_j \end{pmatrix}, t\right)$, $\mathbf{x} = (x,y)^T$, where we have for simplicity suppressed the dependence on t.

In order to treat first derivatives, we could use first-order *forward* difference approximations,

$$\phi_x(\mathbf{x}) \approx \phi_x^+ := \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x} \quad \text{and} \quad \phi_y(\mathbf{x}) \approx \phi_y^+ := \frac{\phi_{i,j+1} - \phi_{i,j}}{\Delta y}.$$

The first-order *backward* difference approximation gives

$$\phi_x(\mathbf{x}) \approx \phi_x^- := \frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x} \quad \text{and} \quad \phi_y(\mathbf{x}) \approx \phi_y^- := \frac{\phi_{i,j} - \phi_{i,j-1}}{\Delta y}.$$

The schemes ϕ_x^-, ϕ_y^-, ϕ_x^+, and ϕ_y^+ are referred to as *upwind* schemes. The second-order central difference approximation gives

$$\phi_x(\mathbf{x}) \approx \phi_x^0 := \frac{\phi_{i+1,j} - \phi_{i-1,j}}{2\Delta x} \quad \text{and} \quad \phi_y(\mathbf{x}) \approx \phi_y^0 := \frac{\phi_{i,j+1} - \phi_{i,j-1}}{2\Delta y}.$$

For the treatment of second derivatives, we have the following second-order finite difference approximations:

$$\phi_{xx}(\mathbf{x}) \approx \phi_{xx}^0 := \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{(\Delta x)^2}, \qquad \phi_{yy}(\mathbf{x}) \approx \phi_{yy}^0 := \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{(\Delta y)^2},$$

and

$$\phi_{xy}(\mathbf{x}) \approx \phi_{xy}^0 := \frac{\phi_{i+1,j+1} - \phi_{i-1,j+1} - \phi_{i+1,j-1} + \phi_{i-1,j-1}}{4\Delta x \Delta y}.$$

The appropriate choice of spatial derivative approximation depends on the type of PDE one is trying to solve. For hyperbolic PDEs, an upwinded scheme is required. For a parabolic PDE, such as (2.6), a central difference scheme is appropriate. Once the spatial derivatives are approximated, the resultant ODE system can be solved using a numerical time-stepping scheme. For certain types of time-stepping schemes and for hyperbolic PDEs, the well known CFL condition provides a restriction on the size of the time-step; we consider this further in the next section. For parabolic PDEs, and for example for (2.6), a standard stability analysis with a forward Euler time integration (see Section 2.5.1) combined with central differencing of $\Delta\phi$ requires $\Delta t \left(\frac{2b}{(\Delta x)^2} + \frac{2b}{(\Delta y)^2} \right) < 1$ [19], (page 44), which gives a restriction on the time step with respect to Δx and Δy. If we use an implicit time stepping method, such as the backward Euler method, there is no stability restriction on the size of Δt.

2.4.1 The CFL Condition

Courant, Friedrichs, and Lewy [18] formulated a necessary condition now known as the CFL condition for the convergence of a difference approximation in terms of its domain of dependence compared to that of the underlying PDE. Consider the simple model problem, $u_t(x,t) + au_x(x,t) = 0$, where a is a constant. Assume an initial condition $u(x,0) = u_0(x)$. Then the solution at a point P can be written as $u(x,t) = u_0(x - at)$.

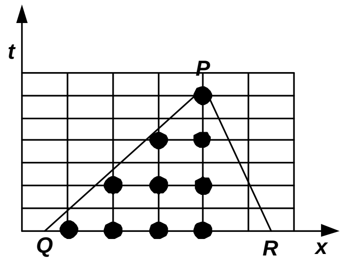

Figure 2.6: An example of a violation of the CFL condition, adapted from [18], page 89.

A plot of the points $x - at = b$ for some constant b gives a line in the (x,t) plane. Along this line, called a *characteristic*, the solution of the PDE is constant.

Suppose we apply a one-sided finite difference scheme for the spatial discretization of u_x; e.g., $u_x = \frac{u_i - u_{i-1}}{\Delta x}$ where $u_i \approx u(x_i)$. Suppose also that Δx

is fixed and that Δt is fixed. Then the triangle of points shown in Figure 2.6 represents the solution approximations from previous time steps upon which the solution approximation at P depends. This triangle is called the *domain of dependence* of the numerical scheme.

Figure 2.6 also illustrates two situations in which the CFL condition is violated. Suppose for two different choices of a in the model equation $u_t + au_x = 0$, we get the characteristic lines PQ and PR. Both of the characteristics PQ and PR lie outside the triangle of points representing the domain of dependence of the numerical scheme. Figure 2.6 shows that the scheme cannot converge for a differential equation for which $a < 0$, because this would give a characteristic like PR. If $a > 0$, and we have a characteristic like PQ, the scheme also does not converge. This characteristic line shows the dependence of the solution of the underlying PDE on the initial condition. *The CFL condition states that the domain of dependence of the numerical scheme must include the domain of dependence of the PDE.* It gives a restriction on the size of the time step, because the condition that the characteristic must lie with in the triangle of dependence of the numerical scheme requires that $|a|\Delta t/\Delta x \leq 1$.

2.4.2 First-Order Upwind Difference Approximations

Consider the one-dimensional level set equation (2.4),

$$\phi_t + \mathbf{V}|\nabla\phi| = 0.$$

The spatial discretization of this equation yields one ODE for each mesh point. The ϕ_t term becomes $\dot{\phi}(x_i, t)$, the $\mathbf{V}$ term becomes $\mathbf{V}_i := \mathbf{V}(x_i, t)$, and the approximation of $|\nabla\phi|$ must be considered carefully. From the discussion in Section 4.1 we see that if $\mathbf{V}_i < 0$, we should find an approximate value of ϕ at time t_{n+1} from the right; otherwise if $\mathbf{V}_i > 0$, we should look to the left to find an approximate value of ϕ at time t_{n+1}. That is, if $\mathbf{V}_i < 0$, we should use ϕ_x^+ to approximate ϕ_x and if $\mathbf{V}_i > 0$, we should use ϕ_x^- to approximate ϕ_x. If we use a forward Euler time discretization (see Section 2.5.1) and an appropriately upwinded spatial discretization, we will have a consistent numerical scheme. Stability comes from the CFL condition which can be written in this case as

$$\Delta t \cdot \max\left(\frac{|\mathbf{V}|}{\Delta x}\right) < 1.$$

2.4.3 Second-Order ENO Upwind Difference Approximations

Hyperbolic PDEs can develop shocks in the solution over the spatial domain. Finite difference schemes of higher order which adaptively avoid such shocks can be developed using an approach that is local, free of problem-dependent parameters, and does not require any characteristic information for hyperbolic conservation laws. The schemes are called essentially non-oscillatory (ENO) schemes [23]. Such a finite difference scheme uses solution information from several points in the solution domain; this set of points is called the *stencil* for the finite difference scheme. ENO methods choose (see below) one stencil out of a number of potential candidates. In one spatial dimension (see Figure 2.7), we suppose we have a first-order finite difference scheme that employs solution approximations at x_i and x_{i+1}. We wish to obtain a higher-order finite difference scheme that is based on additional solution approximations, associated with other nearby mesh points. In the ENO scheme approach, one first considers the neighbouring points, x_{i-1} and x_{i+2}, and for each of these computes certain higher order divided differences (e.g., second-order Newton divided differences; see, e.g., [1]). The neighbouring point to be included in the stencil for the higher order finite difference scheme is the one which yields the smallest value for the Newton divided difference.

This process can be repeated to include further neighbouring points in the stencil, yielding a higher order ENO scheme. Figure 2.7 shows 3 possible sets of neighbouring points, each of which could represent the best stencil for a finite difference scheme of third order. The ENO scheme approach allows us to adaptively choose a finite difference scheme to avoid differencing over a discontinuity in the solution.

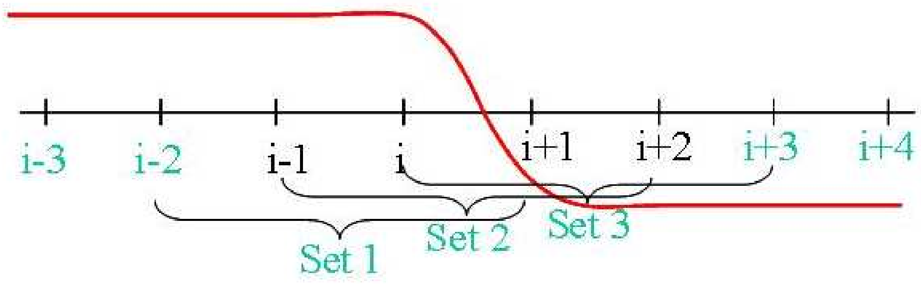

Figure 2.7: The ENO scheme approach chooses the best four points out of a larger set of grid points to improve smoothness.

Let us consider more details for the two-dimensional case [24]. We construct second-order approximations to ϕ_x and ϕ_y of the forms

$$\phi_x \approx \frac{\phi_{i+1/2,j} - \phi_{i-1/2,j}}{\Delta x} \quad \text{and} \quad \phi_y \approx \frac{\phi_{i,j+1/2} - \phi_{i,j-1/2}}{\Delta y},$$

where we will determine the expressions $\phi_{i+1/2}$, $\phi_{i-1/2,j}$, $\phi_{i,j+1/2}$, and $\phi_{i,j-1/2}$ using the ENO approach. See Figure 2.8. Define

$$m(a,b) \equiv \begin{pmatrix} a, & \text{if } |a| \leq |b| \\ \\ b, & \text{otherwise} \end{pmatrix}.$$

Let

$$\phi_L \equiv \phi_{i,j} + \frac{1}{2} m(\phi_{i+1,j} - \phi_{i,j}, \phi_{i,j} - \phi_{i-1,j}), \tag{2.7}$$

$$\phi_R \equiv \phi_{i+1,j} - \frac{1}{2} m(\phi_{i+2,j} - \phi_{i+1,j}, \phi_{i+1,j} - \phi_{i,j}), \tag{2.8}$$

and

$$\phi_M \equiv \frac{1}{2}(\phi_L + \phi_R). \tag{2.9}$$

We then choose

$$\phi_{i+1/2,j} \equiv \begin{cases} \phi_M, & \text{if } \phi_L \leq 0 \text{ and } \phi_R \geq 0, \\ \\ \phi_R, & \text{if } \phi_M \leq 0 \text{ and } \phi_R \leq 0, \\ \\ \phi_L, & \text{if } \phi_M \geq 0 \text{ and } \phi_L \geq 0, \end{cases}$$

where $\phi_L < \phi_M < \phi_R$ (see [23]).

This corresponds to choosing the neighbouring points in order to get a two-dimensional second-order finite difference approximation that has the smallest value for the approximation of the derivative. The idea is to avoid including within the stencil two points which have a discontinuity in the solution between them.

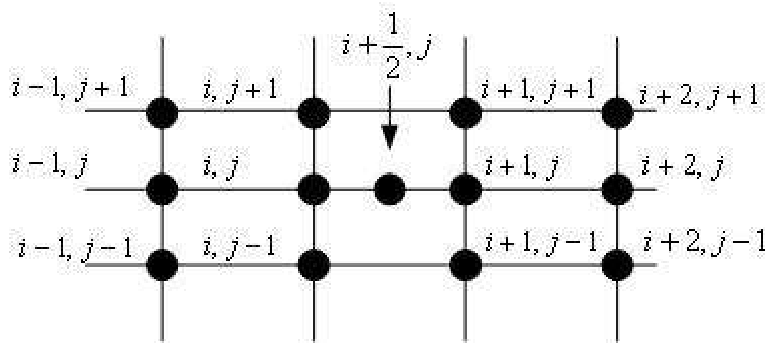

Figure 2.8: ENO Computational Grid

It is possible to construct even better methods by considering weighted combinations of ENO schemes; such schemes are known as weighted ENO (WENO) schemes. See, e.g., [19], for further details.

2.5 Time Stepping

Consider the test equation $\dot{y} = \lambda y$. For a given time stepping method and a time step Δt, the region of absolute stability is the set of points, $\Delta t \cdot \lambda$, in the complex plane such that the method yields an approximate solution satisfying the absolute stability requirement $|y^n| \leq |y^{n-1}|$, $n = 1, 2, \cdots$, where $y^n \approx y(t_n)$, where t_n is the nth step. In Figure 2.9, we see the regions of absolute stability for several popular numerical time-stepping methods called Runge-Kutta methods having orders of accuracy $p = 2, 3$, and 4. (The order of a scheme is p provided the local error is proportional to $(\Delta t)^{p+1}$.) These stability regions provide time step restrictions for the numerical time stepping schemes.

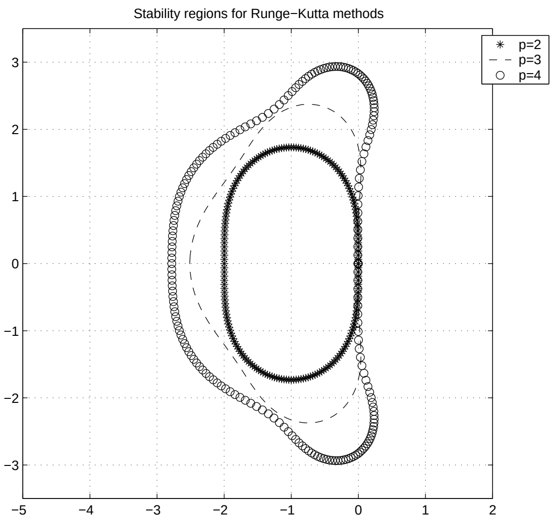

Figure 2.9: Stability regions for some second-, third-, and fourth-order explicit Runge-Kutta methods.

Once the implicit function ϕ and velocity function V are defined at the grid points of our Cartesian grid, we can apply a numerical time stepping method to calculate ϕ forward in time moving the interface across the grid. At some point in time, say time t_n, let $\phi^n = \phi(t_n)$ represent the current values of ϕ at the grid points. Updating ϕ in time consists of finding new values of ϕ at every grid point after some time increment Δt. We denote these new values of ϕ by $\phi^{n+1} = \phi(t_{n+1})$, where $t_{n+1} = t_n + \Delta t$. We next consider several time stepping schemes.

2.5.1 The Forward Euler Method

For the standard ODE, $\dot{y} = f(y, t)$, the forward Euler method has the form

$$y^n = y^{n-1} + \Delta t f(y^{n-1}, t_{n-1}),$$

where y^n approximates $y(t_n)$. We can apply the forward Euler method for the time discretization of equation (2.3) to get the equation

$$\frac{\phi^n - \phi^{n-1}}{\Delta t} + V^{n-1} \cdot \nabla \phi^{n-1} = 0,$$

where $V^n = V(\mathbf{x}, t_n)$, and $\phi^n = \phi(\mathbf{x}, t_n)$. This method is first order in time and has a stability region that is a unit circle, centered at $(-1, 0)$ in the complex plane.

2.5.2 Runge-Kutta Methods

A general s-stage Runge-Kutta method for the ODE system, $\dot{y} = f(y, t)$, is usually written in the form

$$y^n = y^{n-1} + \Delta t \sum_{i=1}^{s} b_i f(Y_i, t_{n-1} + c_i \Delta t), \quad \text{where}$$

$$Y_i = y^{n-1} + \Delta t \sum_{j=1}^{s} a_{ij} f(Y_j, t_{n-1} + c_j \Delta t), \quad 1 \le i \le s.$$

The Runge-Kutta method is explicit iff $a_{ij} = 0$ for $j \ge i$, because then each Y_i is given in terms of known quantities.

One of the most widely used Runge-Kutta methods [1], (page 80), is the following four-stage, fourth-order method which has

$a_{1j} = 0$, $(j = 1, 2, 3, 4)$, $c_1 = 0$, $c_2 = c_3 = \frac{1}{2}$, $c_4 = 1 \Rightarrow Y_1 = y^{n-1}$,

$a_{21} = \frac{1}{2}$, $a_{2j} = 0$, $j = 2, \cdots, 4 \Rightarrow Y_2 = y^{n-1} + \frac{\Delta t}{2} f(Y_1, t_{n-1} + \frac{\Delta t}{2})$,

$a_{31} = 0$, $a_{32} = \frac{1}{2}$, $a_{3j} = 0$, $j = 3, 4 \Rightarrow Y_3 = y^{n-1} + \frac{\Delta t}{2} f(y_2, t_{n-1} + \frac{\Delta t}{2})$,

$a_{41} = a_{42} = 0$, $a_{43} = 1$, $a_{44} = 0 \Rightarrow Y_4 = y^{n-1} + \Delta t f(Y_3, t_{n-1} + \Delta t)$,

and $b_1 = b_4 = \frac{1}{6}$, $\quad b_2 = b_3 = \frac{1}{3}$, giving

$$y^n = y^{n-1} +$$

$$\Delta t \left(\frac{1}{6} f(Y_1, t_{n-1}) + \frac{1}{3} f(Y_2, t_{n-1} + \frac{\Delta t}{2}) + \frac{1}{3} f(Y_3, t_{n-1} + \frac{\Delta t}{2}) + \frac{1}{6} f(Y_4, t_{n-1} + \Delta t) \right).$$

When we apply it to the level set equation (2.3) we get:

$$\Phi_1 = \phi^{n-1},$$

$$\Phi_2 = \phi^{n-1} + \frac{\Delta t}{2} V(\Phi_1, t_{n-1} + \frac{\Delta t}{2}) \nabla \phi(\Phi_1, t_{n-1} + \frac{\Delta t}{2}),$$

$$\Phi_3 = \phi^{n-1} + \frac{\Delta t}{2} V(\Phi_2, t_{n-1} + \frac{\Delta t}{2}) \nabla \phi(\Phi_2, t_{n-1} + \frac{\Delta t}{2}),$$

$$\Phi_4 = \phi^{n-1} + \Delta t V(\Phi_3, t_{n-1} + \Delta t) \nabla \phi(\Phi_3, t_{n-1} + \Delta t),$$

$$\frac{\phi_n - \phi_{n-1}}{\Delta t} =$$

$$(\frac{1}{6} V(\Phi_1, t_{n-1} + \Delta t) \nabla \phi(\Phi_1, t_{n-1} + \Delta t) + \frac{1}{3} V(\Phi_2, t_{n-1} + \frac{\Delta t}{2}) \nabla \phi(\Phi_2, t_{n-1} + \frac{\Delta t}{2}) +$$

$$\frac{1}{3} V(\Phi_3, t_{n-1} + \frac{\Delta t}{2}) \nabla \phi(\Phi_3, t_{n-1} + \frac{\Delta t}{2}) + \frac{1}{6} V(\Phi_4, t_{n-1} + \Delta t) \nabla \phi(\Phi_4, t_{n-1} + \Delta t)).$$

2.5.3 TVD Runge-Kutta Methods

Shu and Osher [22] proposed total variation diminishing (TVD) Runge-Kutta (RK) methods to improve upon the use of the forward Euler method and increase the accuracy of the time stepping scheme. Furthermore, for a TVD scheme, we have the property that $\|y^{n+1}\| \leq \|y^n\|$ provided that, for the given spatial discretization, the forward Euler method also satisfies $\|y^{n+1}\| \leq \|y^n\|$, where $\|.\|$ is a given norm. These TVD RK schemes guarantee that no spurious oscillations are produced as a consequence of the higher-order accurate temporal discretization as long as no spurious oscillations are produced with the forward Euler method for the given spatial discretization. A TVD explicit Runge-Kutta method is built up from a convex combination of forward Euler steps. We next present second and third order TVD Runge-Kutta methods.

A well-known second order TVD Runge-Kutta method [22] has the form,

$$\Phi_1 = \phi^{n-1},$$

$$\Phi_2 = \Phi_1 + \Delta t V(\Phi_1, t_{n-1} + \Delta t)\nabla\phi(\Phi_1, t_{n-1} + \Delta t),$$

$$\Phi_3 = \frac{1}{2}\Phi_1 + \frac{1}{2}\Phi_2 + \frac{1}{2}\Delta t V(\Phi_2, t_{n-1} + \Delta t)\nabla\phi(\Phi_2, t_{n-1} + \Delta t),$$

$$\Phi^n = \Phi_3.$$

A well-known third order TVD Runge-Kutta method [22] has the form,

$$\Phi_1 = \phi^{n-1},$$

$$\Phi_2 = \Phi_1 + \Delta t V(\Phi_1, t_{n-1} + \Delta t)\nabla\phi(\Phi_1, t_{n-1} + \Delta t),$$

$$\Phi_3 = \frac{3}{4}\Phi_1 + \frac{1}{4}\Phi_2 + \frac{1}{4}\Delta t V(\Phi_2, t_{n-1} + \Delta t)\nabla\phi(\Phi_2, t_{n-1} + \Delta t),$$

$$\Phi_4 = \frac{1}{3}\Phi_1 + \frac{2}{3}\Phi_3 + \frac{2}{3}\Delta t V(\Phi_3, t_{n-1} + \Delta t)\nabla\phi(\Phi_3, t_{n-1} + \Delta t),$$

$$\Phi^n = \Phi_4.$$

2.6 Motion Involving Mean Curvature

In this section, we consider several examples involving the evolution of an interface according to the level set approach. In these examples we assume that the velocity of the interface depends only on the curvature, κ, of the interface.

2.6.1 Example I

We consider the level set equation (2.4) ,

$$\phi_t + V|\nabla\phi| = 0.$$

Let ϕ be the height of the propagating function at time t, and $\phi_0(x) = \cos(4\pi x)$, $x \in [0, \pi]$. We consider the two cases $V = 1$ and $V(\kappa) = 1 - \epsilon\kappa$, where $\epsilon \leq 1$, and $\kappa = -\frac{\phi_{xx}}{(1+\phi_x^2)^{\frac{3}{2}}}$ [21], (page 26). Because $|\nabla\phi| = (1 + \phi_x^2)^{\frac{1}{2}}$, for the latter choice of V, the level set equation becomes:

$$\phi_t + \sqrt{1 + \phi_x^2} = -\epsilon\frac{\phi_{xx}}{1 + {\phi_x}^2}. \tag{2.10}$$

We implemented an algorithm for the numerical solution of (2.10) in MATLAB. We use the finite difference methods, $\phi_x = \frac{\phi_{i+1}-\phi_i}{\Delta x}$ and $\phi_{xx} = \frac{\phi_{i+2}-2\phi_{i+1}+\phi_i}{(\Delta x)^2}$, for the spatial derivatives, and the 4th order Runge-Kutta method given in Section 2.5.2 for the time integration, with a time step $\Delta t = .01$. We will let N, the number of mesh points in the x domain, be 150. We consider the case where $\epsilon = 0$ and $\epsilon = 0.1$. The results are shown in Figure 2.10.

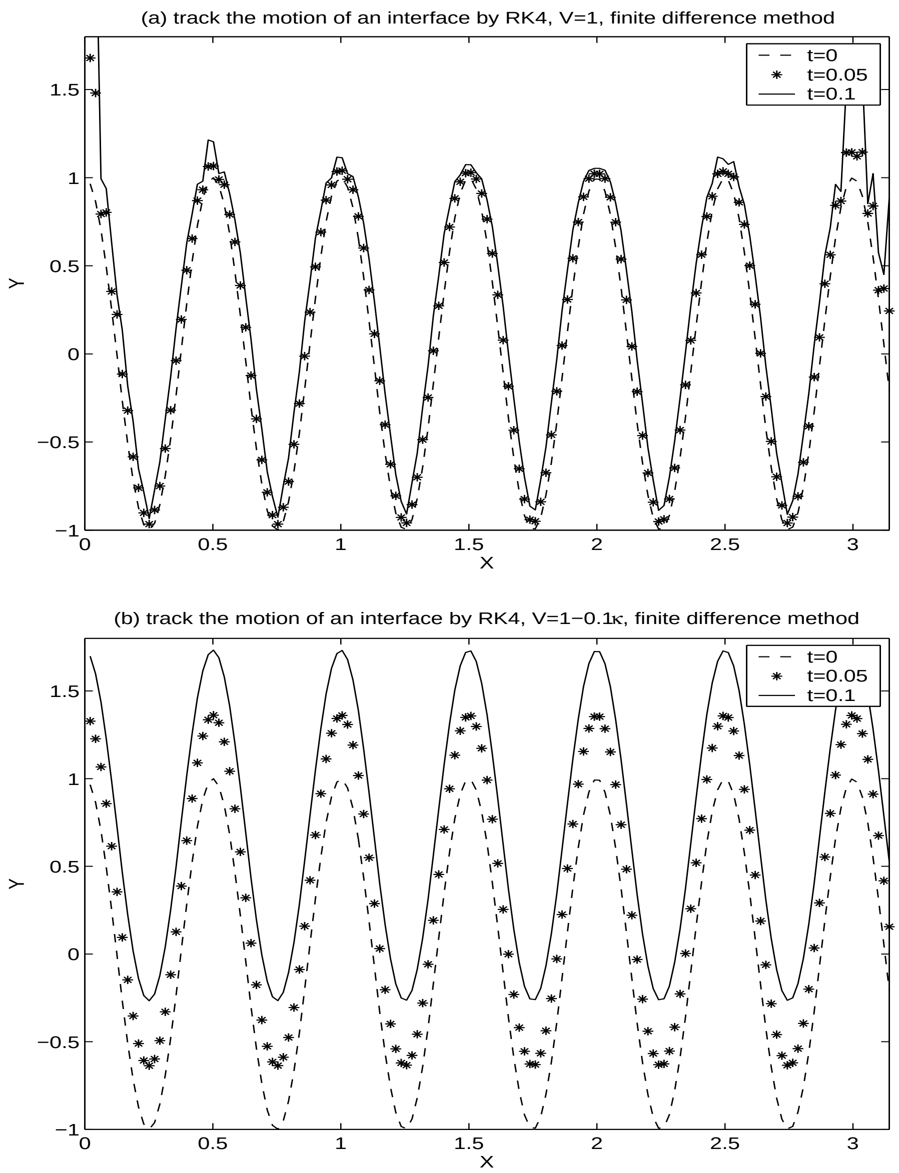

Figure 2.10: (a) $V = 1$, (b) $V = 1 - 0.1\kappa$, with the 4th order Runge-Kutta method.

The velocity function $V = 1$ causes swallowtail behavior [21], (page 21). After adding a little curvature term (viscosity) (i.e., $V = 1 - \epsilon\kappa$), the curve is well-behaved.

2.6.2 Example II

We consider a speed function $\mathbf{V} = -\kappa$, where

$\kappa = \frac{\phi_x^2\phi_{yy} - 2\phi_x\phi_y\phi_{xy} + \phi_y^2\phi_{xx} + \phi_x^2\phi_{zz} - 2\phi_x\phi_z\phi_{xz} + \phi_z^2\phi_{xx} + \phi_y^2\phi_{zz} - 2\phi_y\phi_z\phi_{yz} + \phi_z^2\phi_{yy}}{|\nabla\phi|^3}$ and an initial sphere of radius is 0.4. We use $N = 50$ and a time step $\Delta t = .01$. The evolving surface is a sphere of decreasing radius which eventually disappears. See Figure 2.11. The solution steps are the following:

1. Given initial curve as an input.

2. Build the signed distance function $\phi(x, y, z, t) = 0$.

3. Solve the Level Set Equation, $\phi_t + V|\nabla\phi| = 0$, with $V = -\kappa$, using central differences for the spatial derivatives and forward Euler's method for the time integration. Stability can be achieved by using a much more restrictive CFL condition $\Delta t \sim (\Delta x)^2$; recall that the standard CFL condition from a hyperbolic PDE gives $\Delta t \sim \Delta x$, see Section 2.4.

4. Plot zero contour of ϕ to show the zero level set evolving in time.

We implemented the above algorithm in MATLAB to obtain the results shown in Figure 2.11.

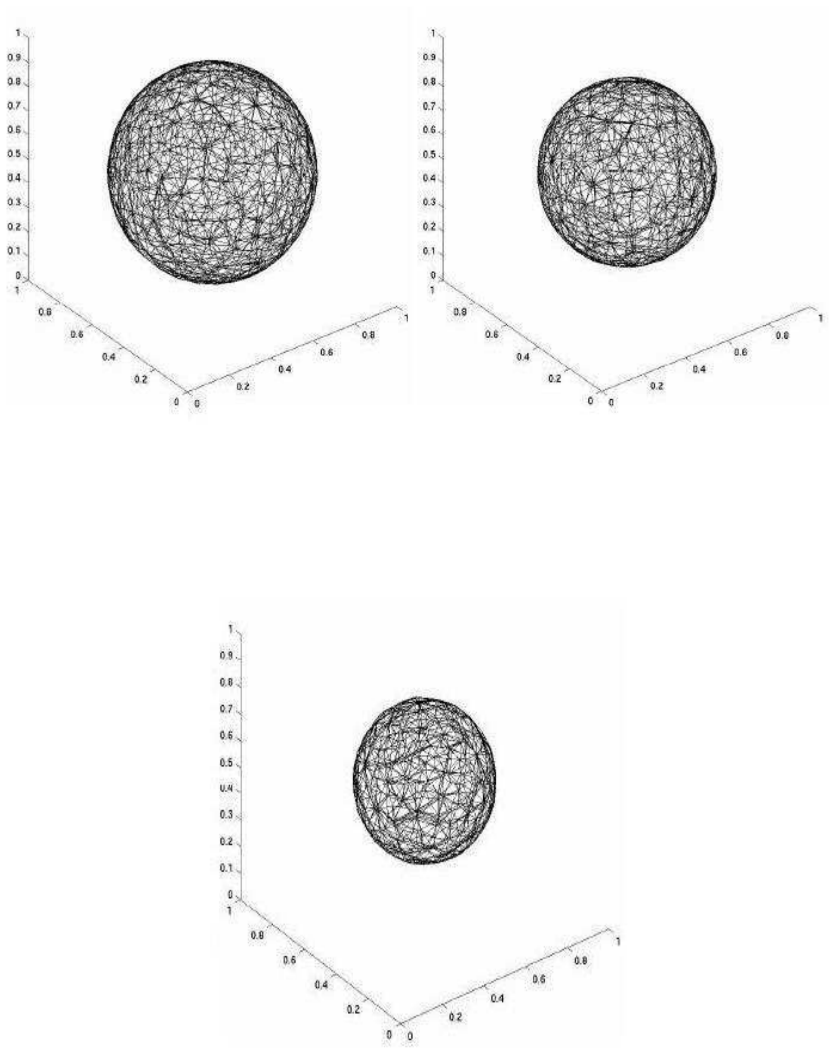

Figure 2.11: $V = -\kappa$; sphere at $t = 0, 0.01, 0.02$, from left to right, from top to bottom.

Chapter 3

Level Set Toolbox

3.1 Introduction

In this chapter, this material is largely drawn from [13]. “A Toolbox of Level Set Methods” is introduced; its source code and documentation are copyrighted by Ian M. Mitchell. Version 1.1 of ”A Toolbox of Level Set Methods” is available at

`http://www.cs.ubc.ca/~mitchell/ToolboxLS`. The purpose of this package of software tools is to provide a resource for those interested in implementing level set methods. This toolbox is a collection of MATLAB routines for working with dynamic implicit surfaces and approximating the solution of Hamilton-Jacobi PDEs; the specific form for the equations will be defined shortly.

The supplied routines work in 1, 2, or 3 dimensions on a fixed Cartesian grid. Routines that implement upwind spatial derivative and explicit temporal derivative approximations of high-order accuracy are included, as well as routines that implement level set methods for more than a dozen examples drawn from the level set literature. Most of the algorithms and examples in this version of the toolbox are taken from [19].

The toolbox is designed to ease the process of exploring the application of level set methods by reducing the total coding, execution, and analysis time. The sophisticated analysis, data manipulation, and visualization capabilites of MATLAB make construction of numerical codes much simpler, when compared to compiled languages like C++ or Fortran. Although [13] is not by itself a tutorial on these methods, the author suggests that for those who are new to the field it would serve as an excellent supplement to either of the textbooks [19], [21].

3.2 Purpose of the Toolbox

Using the routines provided in the toolbox, a user can treat a Hamilton-Jacobi (HJ) PDE of the following form, as defined by Mitchell in [13]:

$$0 = D_t\phi(\mathbf{x}, t) \tag{3.1}$$

$$+v(\mathbf{x}, t) \cdot \nabla\phi(\mathbf{x}, t) \tag{3.2}$$

$$+a(\mathbf{x}, t)\|\nabla\phi(\mathbf{x}, t)\| \tag{3.3}$$

$$-b(\mathbf{x}, t)\kappa(\mathbf{x})\|\nabla\phi(\mathbf{x}, t)\| \tag{3.4}$$

$$+\text{sign}(\phi(\mathbf{x}, 0))(\|\nabla\phi(\mathbf{x}, t)\| - 1) \tag{3.5}$$

$$+H(\mathbf{x}, \nabla\phi), \tag{3.6}$$

with constraints

$$D_t\phi(\mathbf{x}, t) \geq 0, \quad \text{or} \quad D_t\phi(\mathbf{x}, t) \leq 0,$$

where $\mathbf{x} \in \Re^n$ is the state space, ϕ is the level set function, and $\nabla\phi(\mathbf{x}, t)$ is the gradient of ϕ. All terms (3.1)-(3.6) are defined below.

The time derivative appearing in (3.1) is approximated with an explicit TVD RK integration scheme; CFL conditions, determined automatically within the level set toolbox, restrict the size of each timestep. Note that the time derivative (3.1) and at least one term involving a spatial derivative (3.2)-(3.6) must appear, otherwise the equation is not a time-dependent HJ PDE.

The term (3.2) represents motion of the interface subject to a constant velocity field. The velocity field v must be provided by the user, and then the toolbox provides an upwind finite difference scheme to approximate the gradient $\nabla\phi(\mathbf{x}, t)$.

The term (3.3) represents motion of the interface in the normal direction. The user provides the speed of the interface a, and the gradient $\nabla\phi(\mathbf{x}, t)$ is approximated with an upwind finite difference scheme.

The next term (3.4) represents motion by mean curvature. The user provides the speed b, and the mean curvature $\kappa(\mathbf{x})$. We recall from Chapter 2 that for this term, the PDE is parabolic and upwind schemes are not necessary. The spatial derivatives can therefore be approximated by centered second-order finite difference approximations, which are provided by the toolbox.

The next term (3.5) represents the reinitialization equation. It can be used with implicit surface functions so as to change them back into signed distance functions. See Section 2.2 of Chapter 2.

The final term (3.6), allows for the representation of a general Hamilton-Jacobi term, which is first-order, spatially dependent, and continuous in x and t. Such terms arise in optimal control and zero-sum differential games. The user provides the analytic Hamiltonian H, and the gradient $\nabla\phi(\mathbf{x}, t)$ is approximated with an upwind finite difference scheme.

3.3 Level Set Examples

3.3.1 Application of the Toolbox

Below we provide the basic steps that should be followed in order to solve an application problem using the toolbox:

1. Identify a HJ PDE from (3.1)-(3.6).

2. Specify the desired order of accuracy, the CFL condition, and other information relevant to the HJ PDE, such as velocity.

3. Specify the boundary conditions, the initial condition, and the grid.

4. Integrate in time by using a TVD RK method.

We consider shortly an example demonstrating the use of the toolbox in solving a problem.

3.3.2 Getting Started with the Toolbox

To use the toolbox, one needs to run the basic version of MATLAB. (We have performed tests using version 6.5, but earlier 6.* versions may also work.) No additional toolboxes are required. The computing environment we use is Microsoft Windows XP; the machine type is a Dell Dptiplex Gx270 personal computer. One can try out the distribution of the level set toolbox by going to the Examples subdirectory of the level set toolbox. There one will find the file `addPathToKernel.m`, which should be edited to modify the path name contained there so that it contains the absolute path (starting from root) of the Kernel subdirectory of the distribution. Once this modification had been performed, one can start MATLAB, and execute one of the examples in any of the example subdirectories by typing its name at the MATLAB prompt.

Following the lead of [13], we next provide an introduction to the toolbox through a few examples; this will introduce a number of routines from the toolbox. We will then follow the examples with an overview of the full set of routines provided in the toolbox.

3.3.3 Motion by a Constant Velocity Field

This section describes an example involving motion by a constant velocity field (3.2). It is accessed by using the file `Examples/Basic/convectionDemo`. We first discuss this example to see how it works. We will then examine the source code for this example to see how the level set toolbox routines are used to implement it. The MATLAB command to execute this example is `[data, g, data0]= convectionDemo(flowType, accuracy, displayType)`. All of the input parameters are optional; the first input parameter, `flowType`, specifies the type of flow. The options are as given in Table 3.1. The con-

flowType	String to specify type of flow field
'constant'	Constant flow field V = Constant (default)
'linear'	Linear flow field V = Linear=$A\mathbf{x} + b$
'constantRev'	Constant flow field, reverses direction at $t_{half} = \frac{1}{2}t_{max}$
'linearRev'	Linear flow field, reverses direction at $t_{half} = \frac{1}{2}t_{max}$

Table 3.1: Flowtype Options, adapted from [13].

stant flow field demonstrates a spatially independent flow field. The linear flow field demonstrates a spatially dependent flow field. The quantity, t_{max}, gives the end of the time interval.

The second input parameter, `accuracy`, specifies how much accuracy is needed. The choices available for this parameter are as given in Table 3.2, where the time integraters are `odeCFL1`, forward Euler, `odeCFL2`: second-

accuracy	Method used
'low'	Use odeCFL1 and upwindFirstFirst (default)
'medium'	Use odeCFL2 and upwindFirstENO2
'high'	Use odeCFL3 and upwindFirstENO3
'veryHigh'	Use odeCFL3 and upwindFirstWENO5

Table 3.2: Accuracy Options, adapted from [13].

order TVD RK scheme with CFL condition, and `odeCFL3`: a third-order TVD RK scheme with CFL condition, and the spatial discretization schemes are `upwindFirstFirst`: a first-order upwind scheme, `upwindFirstENO2`: a second-order ENO scheme, `upwindFirstENO3`: a third-order ENO scheme, and `upwindFirstWENO5`: a fifth-order WENO scheme. The third input parameter `displayType` specifies how to display results.

The first returned parameter, `data`, is the implicit surface function at t_{max}; the second returned parameter, `data0`, is implicit surface function at t_0, and the third returned parameter, `g`, is the computational grid. These three returned parameters are arrays.

Let us now consider a specific example. Suppose we type the MATLAB command `[data, g, data0]= convectionDemo('constant', 'medium')`. The results we get are shown in Figure 3.1. The initial interface setup by this example is a circle, and the effect of this example is to implement a circle moving with a constant velocity from left to right as in Figure 3.1.

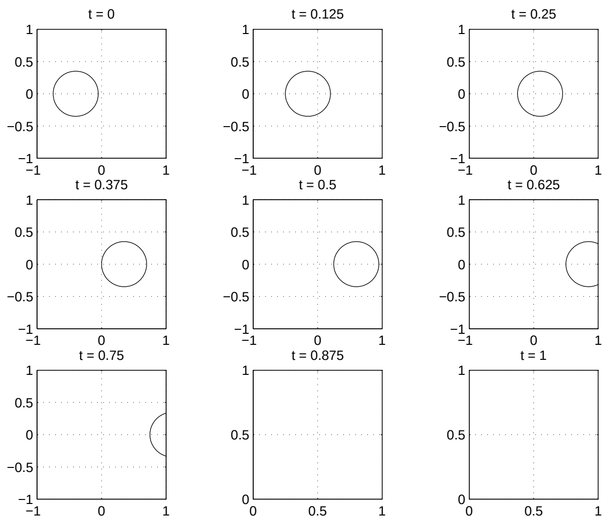

Figure 3.1: Motion of a circle by a constant velocity field.

The results given in Table 3.3 show the execution time for `convectionDemo('constant', accuracy)` with the different choices of accuracy.

Accuracy Parameter	Temporal Accuracy	Spatial Accuracy	Execution Time Seconds
low	1	1	1.168
medium	2	ENO2	5.095
high	3	ENO3	16.864
very high	3	WENO5	20.242

Table 3.3: The execution time for convectionDemo with different choices of accuracy.

Review of the convectionDemo Function

This demo is used as the basis for our implementation of level sets for the treatment of the sloshing problem. We provide the source code for this new function, called `convectionSlosh` in the appendix and refer the reader to it as a reference for this material. A number of values are set (but easily modified) inside this function. The variables `t0`(initial time) and `tMax`(final time) control the length of the simulation. The variable `g.dim` specifies the dimension (one, two, or three); `useSubplots` determines whether to display results in a single figure or in separate subplots. The function `schemeFunc` is the subfunction which describes the spatial approximation schemes. The function `termConvection` that implements a spatial approximation for the term (3.2). The function `integratorFunc` is the subfunction which describes the time integration schemes (the options are: `odeCFL1`, `odeCFL2`, and `odeCFL3`). The flow field information is stored in `schemeData.velocity`. The function `visualizeLevelSet` performs the actual visualization.

3.3.4 Motion in the Normal Direction

In this section we examine motion in the normal direction (3.3) using another example from the level set toolbox, namely the function, `Examples/OsherFedkiw/normalStarDemo`. This function describes motion in the normal direction at speed $a(\mathbf{x}, t)$ of a star-shaped interface, where the user provides the speed $a(\mathbf{x}, t)$.

The initial configuration for the level set in this case is a star; see Figure 3.2. The MATLAB command is

`[data, g, data0]=normalStarDemo(accuracy, reverseFlow, displayType)`.

The parameters `accuracy` and `displayType` are the same as in the previous example. The boolean parameter `reverseFlow` specifies whether to reverse the motion of the flow at half time.

If we type the MATLAB command

```
[data, g, data0]= normalStarDemo('medium', '0'),
```

the results we get are shown in Figure 3.2. We see the evolution of the star shaped interface by motion in the direction normal to the interface. The im-

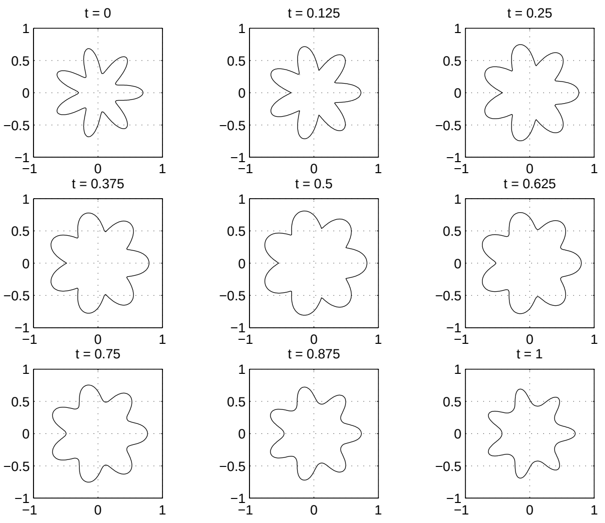

Figure 3.2: Motion of a star shaped interface in the normal direction, adapted from [13].

plementation of the `normalStarDemo` is similar to that of `convectionSlosh`, given in the appendix.

3.3.5 Motion by Mean Curvature

In this section we examine the `curvatureStarDemo` in the directory `Examples/OsherFedkiw/`, and the function `dumbbell1` in the directory

`Examples/Sethian/`. The function `curvatureStarDemo` describes motion of a star shaped interface by mean curvature (4) with speed $b(\mathbf{x}, t)$ which is provided by the user. It uses the subfunction `switchValue`.

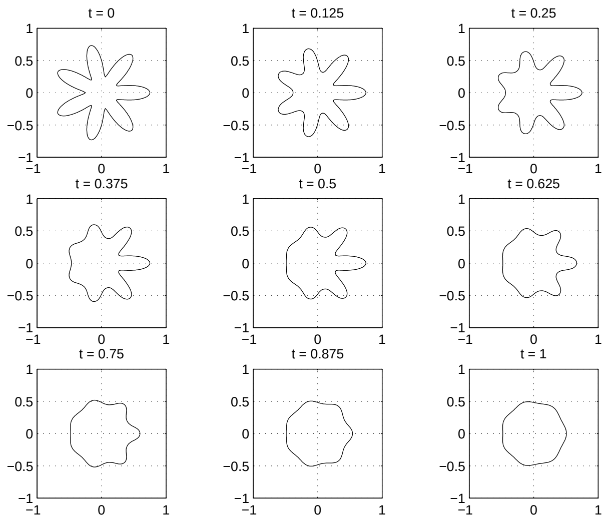

Figure 3.3: Motion by mean curvature of a star shape with mulitplier $b(\mathbf{x}, t)$ varying in time and space, adapted from [13].

The MATLAB command is

`[data, g, data0]=curvatureStarDemo(accuracy, splitFlow, displayType)`.

The parameters, `accuracy` and `displayType` are as before. The boolean parameter `splitFlow` ('0' or '1') specifies whether the multiplier should be constant or varying in time. The returned parameters are the same as in previous examples.

If we type the MATLAB command

`[data, g, data0]= curvatureStarDemo('medium', '1')`, the results we get are shown in Figure 3.3. The figure shows motion of a star shape by mean curvature, with the speed $b(\mathbf{x}, t)$ varying in time and space. For the time and spatially varying case `splitFlow=1`. The source code for the function `curvatureStarDemo` is also similar to the previously described functions.

Review of the dumbbell1 Function

We now consider an example involving the motion of a three-dimensional dumbbell shaped figure by mean curvature. The MATLAB command is `[data, g, data0]= dumbbell1(accuracy)`. The `accuracy` parameter is as the same as before, as are the returned parameters. The dumbbell is created as the zero contour of a ϕ function defined in [13]:

$$\psi_{left}(x) = \sqrt{(x_1 + o)^2 + x_2^2 + x_3^2} - r,$$

$$\psi_{right}(x) = \sqrt{(x_1 - o)^2 + x_2^2 + x_3^2} - r,$$

$$\psi_{center}(x) = \max\left[(|x_1 - o|), \left(\sqrt{x_2^2 + x_3^2} - \omega\right)\right],$$

$$\phi(x, 0) = \min[\psi_{left}(x), \psi_{right}(x), \psi_{center}(x)],$$

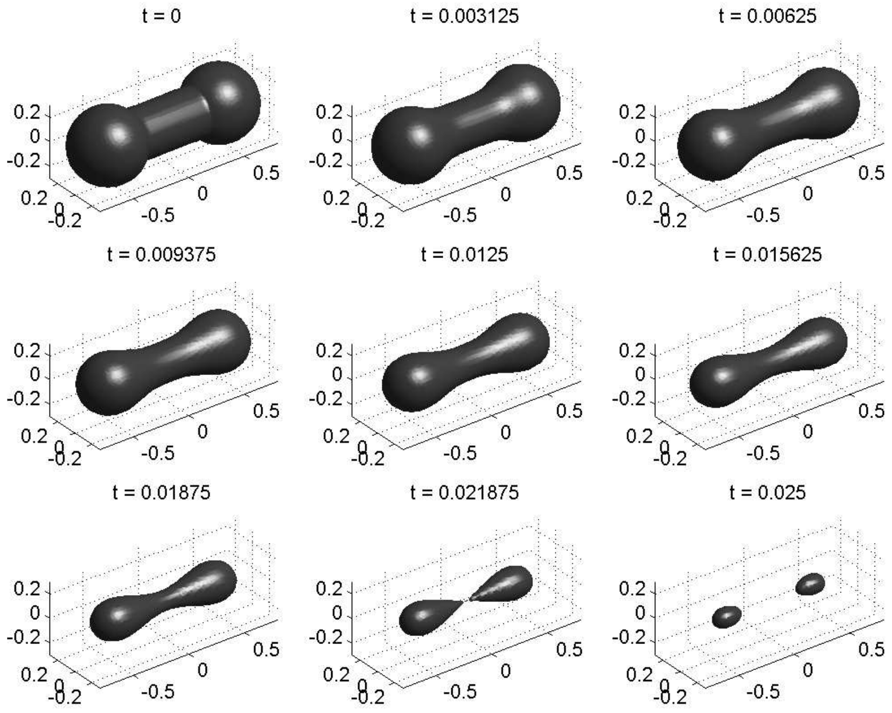

Figure 3.4: Motion by mean curvature of a three dimensional dumbbell, adapted from [13].

where o is the offset of the center of the lobes of the dumbbell, r is the radius of the lobes, and ω is the radius of the center cylinder. The dumbbell is the union of these three implicit surfaces.

Let us now consider a specific example. If we type the MATLAB command
`[data, g, data0]= dumbbell1('medium')`, the results we get are shown in Figure 3.4. The source code for `dumbbell1` is also similar to that of `convectionSlosh`.

3.4 The Toolbox Functions

Here we list the complete set of functions available in the toolbox:

`convectionDemo` demonstrates motion by an external velocity field of a circle.

`reinitDemo` demonstrates the reinitialization equation.

`laxFriedrichsDemo` is an implementation of time independent convective flow using a general HJ solver.

`curvatureSpiralDemo` demonstrates motion by mean curvature of a two-dimensional wound spiral interface.

`curvatureStarDemo` demonstrates motion of a star by mean curvature with multiplier $b(x)$.

`normalStarDemo` demonstrates motion of the surface of a star in the normal direction at speed $a(\mathbf{x}, t)$.

`spinStarDemo` demonstrates the combination of motion of a star in the normal direction and with convective rotation.

`tripleSine` demonstrates the evolution of a sine-shaped interface under a combination of curvature and normal motion.

`dumbbell1` demonstrates the evolution of a three-dimensional dumbbell under motion by mean curvature.

`burgersLF` demonstrates solution of Burgers' equation.

`figureAir3D` visualizes the three dimensional reachable set, and possibly the initial collision/target set.

`Grids` found in the directory `Kernel/Grids`; a grid is represented by a structure: a fixed rectangular Euclidean mesh.

`BoundaryConditions` is called by the spatial derivative approximation function (`schemeFunc`). There are three boundary conditions from which to choose: Periodic, Dirichlet, and Neumann boundary conditions.

`InitialConditions` can create basic shapes: circles, spheres, cylinders, squares, cubes, rectangles, hyperplanes, and polygons. The functions `termConvection`, `termNormal`, and `termReinit` in the directory `Kernel/ExplicitIntegration/Term/` approximate the terms that implement convection by a velocity field (3.2), motion in the normal direction (3.3), and the reinitialization equation (3.4), respectively.

The functions in the directories `Kernel/ExplicitIntegration/Term/termLaxFriedrichs`, and `Kernel/ExplicitIntegration/Dissipation` approximate general HJ terms.

Chapter 4

Numerical Modelling of the Navier-Stokes Equations

4.1 Introduction

The Navier-Stokes equations are the fundamental equations for the modelling of fluid flow problems and are thus relevant to the modelling of sloshing phenomena. In this chapter we introduce the Navier-Stokes equations and their numerical solution. We also consider the numerical solution of the Navier-Stokes equations for one example.

4.2 The Navier-Stokes Equations

The following material is largely drawn from [8], (page 11-page 35).

The Navier-Stokes equations are derived from a number of assumptions about the attributes of the fluid, such as the density, ρ, etc. Based on the density, the flow can be classified as compressible or incompressible. If the flow is incompressible, the density ρ is constant with respect to both space and time. We consider incompressible flows in this thesis. The dimensionless Navier-Stokes equations take the following form:

the momentum equation is

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \frac{1}{Re}\nabla^2 \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u} + \mathbf{g} \tag{4.1}$$

and the continuity equation is

$$\nabla \cdot \mathbf{u} = 0, \tag{4.2}$$

where $\mathbf{u}$ is the velocity vector, p is pressure, Re is the Reynolds number, one of the basic dimensionless numbers in fluid dynamics, and $\mathbf{g}$ denotes body forces such as gravity.

Because

$$(\mathbf{u} \cdot \nabla)\mathbf{u} = \begin{pmatrix} u\partial u/\partial x + v\partial u/\partial y \\ u\partial v/\partial x + v\partial v/\partial y \end{pmatrix},$$

from (4.2), we have

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \Rightarrow u\frac{\partial u}{\partial x} + u\frac{\partial v}{\partial y} = 0.$$

Also

$$\frac{v\partial u}{\partial x} + \frac{v\partial v}{\partial y} = 0.$$

We next observe that

$$\frac{\partial(u^2)}{\partial x} + \frac{\partial(uv)}{\partial y} = \frac{2u\partial u}{\partial x} + \frac{u\partial v}{\partial y} + \frac{v\partial u}{\partial y} = \frac{u\partial u}{\partial x} + \frac{v\partial u}{\partial y},$$

and

$$\frac{\partial(uv)}{\partial x} + \frac{\partial(v^2)}{\partial y} = \frac{u\partial v}{\partial x} + \frac{v\partial u}{\partial x} + \frac{2v\partial v}{\partial y} = \frac{u\partial v}{\partial x} + \frac{v\partial v}{\partial y}.$$

The momentum equations then become

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} + g_x,$$

and

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} + g_y. \tag{4.3}$$

The continuity equation is

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{4.4}$$

The initial conditions are $u(x, y, 0) = u_0(x, y)$ and $v(x, y, 0) = v_0(x, y)$, and we assume that these functions satisfy (4.4).

To define the boundary conditions, we will employ the following definitions: φ_n will be the component of velocity orthogonal to the boundary, φ_t will be the component of velocity parallel to the boundary, and $\frac{\partial \varphi_n}{\partial n}$, $\frac{\partial \varphi_t}{\partial n}$, will be their derivatives in the normal direction. We assume that the boundaries are parallel to the coordinate axes. On the vertical components of the boundary we have

$$\varphi_n = u, \qquad \varphi_t = v, \qquad \frac{\partial \varphi_n}{\partial n} = \frac{\partial u}{\partial x}, \qquad \frac{\partial \varphi_t}{\partial n} = \frac{\partial v}{\partial x},$$

whereas, on the horizontal segments of the boundary we have

$$\varphi_n = v, \qquad \varphi_t = u, \qquad \frac{\partial \varphi_n}{\partial n} = \frac{\partial v}{\partial y}, \qquad \frac{\partial \varphi_t}{\partial n} = \frac{\partial u}{\partial y}.$$

For fluid on the boundary, one the following sets of boundary condtions are often assumed:

1. No-slip condition: No fluid flows through the boundary; i.e., the velocities in the horizontal and vertical directions at the boundaries must be zero,

$$\varphi_n(x, y) = 0, \qquad \varphi_t(x, y) = 0.$$

2. Free-slip condition: The velocity normal to the boundary is zero, but there is no change in the tangential velocity with respect to the normal (i.e., no frictional losses)

$$\varphi_n(x,y) = 0, \qquad \varphi_t(x,y)/\partial n = 0.$$

3. Inflow condition: The velocities in the horizontal and vertical directions at the boundaries are given; i.e.,

$$\varphi_n(x,y) = \varphi_n^0, \qquad \varphi_t(x,y) = \varphi_t^0, \qquad \text{with} \quad \varphi_n^0, \varphi_t^0 \quad \text{given.}$$

4. Outflow condition: Neither velocity component changes in the direction normal to the boundary; i.e.,

$$\varphi_n(x,y)/\partial n = 0, \qquad \varphi_t(x,y)/\partial n = 0.$$

In general, solutions to the Navier-Stokes equations cannot be obtained analytically. Rather, they must be approximated numerically.

4.3 The Numerical Treatment of the Navier-Stokes Equations

In this section we consider basic methods for the numerical solution of the unsteady incompressible Navier-Stokes equations, (4.3), (4.4). This material is largely drawn from [8], (page 22-page 39).

4.3.1 Spatial Discretization

We consider a finite difference method for the discretization of Navier-Stokes equations on rectangular domains. We will employ what is referred to as a staggered grid. For a given cell of such a grid, the pressure p is associated with the center of the cell, the horizontal velocity u is associated with the midpoints of each vertical cell edge, and the vertical velocity v is associated with the midpoints of each horizontal cell edge. Thus the pressure value $p_{i,j}$ is defined to be at the coordinates $((i-0.5)\Delta x, (j-0.5)\Delta y$, the horizontal velocity value $u_{i,j}$ is definded to be at the coordinates $(i\Delta x, (j-0.5)\Delta y)$, and the vertical velocity value $v_{i,j}$ at the coordinates $((i-0.5)\Delta x, j\Delta y)$, where $i \in \{0, i_{max}\}$ and $j \in \{0, j_{max}\}$.

We can then discretize the spatial derivatives arising in equation (4.3) as follows, [8], (page 29)

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}, \quad \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2},$$

$$\frac{\partial^2 v}{\partial x^2} \approx \frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{\Delta x^2}, \quad \frac{\partial^2 v}{\partial y^2} \approx \frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{\Delta y^2},$$

$$\frac{\partial (u^2)}{\partial x} \approx \frac{1}{\Delta x}\left(\left(\frac{u_{i,j} + u_{i+1,j}}{2}\right)^2 - \left(\frac{u_{i-1,j} + u_{i,j}}{2}\right)^2\right),$$

$$\frac{\partial (v^2)}{\partial y} \approx \frac{1}{\Delta y}\left(\left(\frac{v_{i,j} + v_{i,j+1}}{2}\right)^2 - \left(\frac{v_{i,j-1} + v_{i,j}}{2}\right)^2\right),$$

$$\frac{\partial(uv)}{\partial x} \approx \frac{1}{\Delta x}\left(\frac{(u_{i,j}+u_{i,j+1})}{2}\frac{(v_{i,j}+v_{i+1,j})}{2} - \frac{(u_{i-1,j}+u_{i-1,j+1})}{2}\frac{(v_{i-1,j}+v_{i,j})}{2}\right),$$
$$\frac{\partial(uv)}{\partial y} \approx \frac{1}{\Delta y}\left(\frac{(v_{i,j}+v_{i+1,j})}{2}\frac{(u_{i,j}+u_{i,j+1})}{2} - \frac{(v_{i,j-1}+v_{i+1,j-1})}{2}\frac{(u_{i,j-1}+u_{i,j})}{2}\right),$$

and

$$\frac{\partial p}{\partial x} \approx \frac{p_{i+1,j}-p_{i,j}}{\Delta x}, \quad \frac{\partial p}{\partial y} \approx \frac{p_{i,j+1}-p_{i,j}}{\Delta y}.$$

4.3.2 Boundary Values for the Discrete Equations

For the momentum equations, (4.3), we assume values on the boundary (see Figure 4.1.) That is, we assume values for the solution

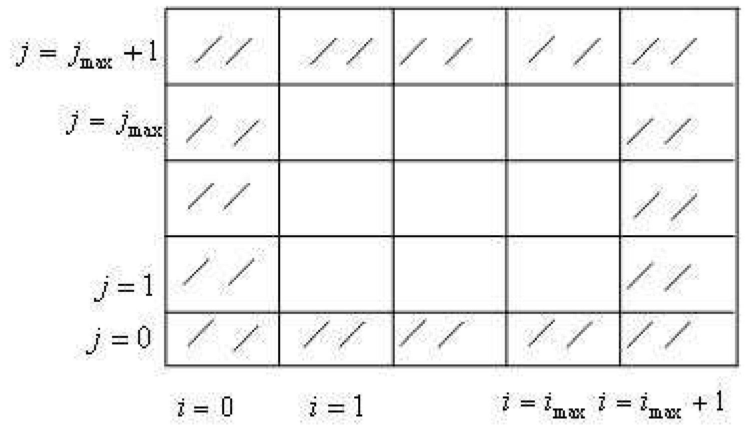

Figure 4.1: Domain with boundary cells; solution values are assumed to be available on these cells, adapted from [8], page 27.

$$u_{0,j}, \qquad u_{i_{max},j}, \qquad j = 1, \cdots, j_{max},$$

$$v_{i,0}, \qquad v_{i,j_{max}}, \qquad i = 1, \cdots, i_{max}.$$

$$u_{i,0}, \qquad u_{i,j_{max+1}}, \qquad i = 1, \cdots, i_{max},$$

$$v_{0,j}, \qquad v_{i_{max+1},j}, \qquad j = 1, \cdots, j_{max},$$

outside the domain Ω. These velocity values are obtained from a discretization of the boundary conditions of the continuous problem, as defined in [8]:

1. No-slip condition: The values of continuous velocities should be zero at the boundary; therefore we set:

$$u_{0,j} = 0, \qquad u_{i_{max},j} = 0, \qquad j = 1, \cdots, j_{max},$$

$$v_{i,0} = 0, \qquad v_{i,j_{max}} = 0, \qquad i = 1, \cdots, i_{max}.$$

The zero boundary value is calculated by averaging the values on either side of the boundary as shown in Figure 4.2. $\mathbf{v}_a$ is the velocity at the midpoint of the cell outside the boundary; $\mathbf{v}_i$ is the velocity at the midpoint of the cell inside the boundary.

$$\mathbf{v}_r := \frac{\mathbf{v}_a + \mathbf{v}_i}{2} = 0 \quad \Longrightarrow \qquad \mathbf{v}_a = -\mathbf{v}_i. \tag{4.5}$$

We thus obtain the conditions

$$u_{i,0} = -u_{i,1}, \qquad u_{i,j_{max+1}} = -u_{i,j_{max}}, \qquad i = 1, \cdots, i_{max},$$

$$v_{0,j} = -v_{1,j}, \qquad v_{i_{max}+1,j} = -v_{i_{max},j}, \qquad j = 1, \cdots, j_{max}.$$

2. Free-slip condition: the values of velocities normal to the boundary should be zero on the boundary; therefore we set:

$$u_{0,j} = 0, \qquad u_{i_{max},j} = 0, \qquad j = 1, \cdots, j_{max},$$

$$v_{i,0} = 0, \qquad v_{i,j_{max}} = 0, \qquad i = 1, \cdots, i_{max}.$$

We will approximate the normal derivative of the tangential velocity v, i.e., $\partial v/\partial n$, at a point Q using a simple divided difference, $(v_i - v_a)/\Delta x$; (see Figure 4.3); so that the requirement $\partial v/\partial n = 0$ leads to the condition $v_a = v_i$. We thus obtain the boundary conditions

$$u_{i,0} = u_{i,1}, \qquad u_{i,j_{max+1}} = u_{i,j_{max}}, \qquad i = 1, \cdots, i_{max},$$

$$v_{0,j} = v_{1,j}, \qquad v_{i_{max+1},j} = v_{i_{max},j}, \qquad j = 1, \cdots, j_{max}.$$

3. Outflow condition: the normal derivatives of u and v are set to zero at the boundary. We set velocity values at the boundary equal to their neighboring velocities inside the domain.

$$u_{0,j} = u_{1,j}, \qquad u_{i_{max},j} = u_{i_{max-1},j},$$

$$v_{0,j} = v_{1,j}, \qquad v_{i_{max+1},j} = v_{i_{max},j}, \qquad j = 1, \cdots, j_{max},$$

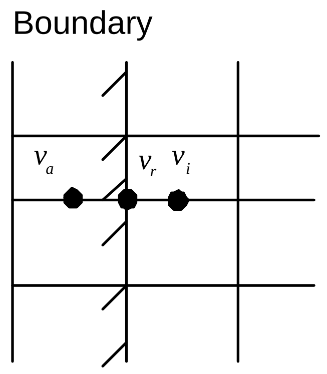

Figure 4.2: No-slip Boundary Condition, adapted from [8], page 30.

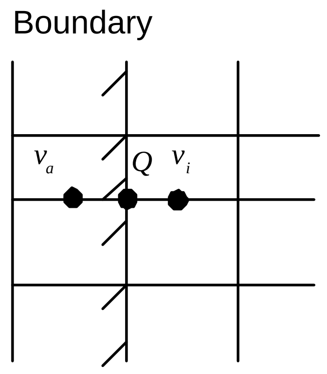

Figure 4.3: Free-slip Boundary Condition, adapted from [8], page 31.

$$u_{i,0} = u_{i,1}, \qquad u_{i,j_{max}+1} = u_{i,j_{max}},$$

$$v_{i,0} = v_{i,1}, \qquad v_{i,j_{max}} = v_{i,j_{max}-1}, \qquad i = 1, \cdots, i_{max}.$$

4. Inflow condition: for the velocity components tangential to the boundary, we average the values on either side of the boundary as in (4.5).

4.3.3 Time Stepping

We subdivide the time interval $[0, t_{end}]$ into equal subintervals $[n\Delta t, (n+1)\Delta t]$, $n = 0, \cdots, t_{end}/\Delta t - 1$. Denote by u^n, v^n, and p^n, the velocity and pressure values defined at times $n\Delta t$. To discretize the time derivatives at time t_{n+1} we use the forward Euler method. The finite difference approximations in time to (4.3) can then be written in the following form:

$$\frac{u^{n+1} - u^n}{\Delta t} = -\frac{\partial p}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x, \tag{4.6}$$

$$\frac{v^{n+1} - v^n}{\Delta t} = -\frac{\partial p}{\partial y} + \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y. \tag{4.7}$$

Letting

$$F := u^n + \Delta t\left(\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x\right),$$

and

$$G := v^n + \Delta t \left(\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \right), \qquad (4.8)$$

we can rewrite (4.6), (4.7) in the form

$$\begin{aligned} u^{n+1} &= F - \Delta t \frac{\partial p}{\partial x}, \\ v^{n+1} &= G - \Delta t \frac{\partial p}{\partial y}. \end{aligned} \qquad (4.9)$$

Differentiating and substituting these equations into (4.4) gives

$$\frac{\partial u^{n+1}}{\partial x} + \frac{\partial v^{n+1}}{\partial y} = \frac{\partial F^n}{\partial x} - \Delta t \frac{\partial^2 p^{n+1}}{\partial x^2} + \frac{\partial G^n}{\partial y} - \Delta t \frac{\partial^2 p^{n+1}}{\partial y^2} = 0.$$

After rearranging terms, this becomes a Poisson equation for the pressure p^{n+1} at time t_{n+1}:

$$\frac{\partial^2 p^{n+1}}{\partial x^2} + \frac{\partial^2 p^{n+1}}{\partial y^2} = \frac{1}{\Delta t} \left(\frac{\partial F^n}{\partial x} + \frac{\partial G^n}{\partial y} \right).$$

We have implemented an algorithm for the numerical treatment of the Navier-Stokes equations using the numerical methods described here. The MATLAB source code is given in the Appendix A.2.

4.3.4 Stream Function

If u and v are the velocities of the flow field, then the stream function $\phi(x, y)$ is defined by

$$\frac{\partial \phi(x, y)}{\partial x} = -v, \quad \frac{\partial \phi(x, y)}{\partial y} = u.$$

For a physical interpretation of the stream function, we first introduce the concept of a streamline. A streamline is a curve whose tangent is parallel to the velocity vector $(u, v)^T$ at each of its points (x, y) at a fixed time t. We show an example involving streamlines in the next subsection that is implemented using MATLAB. We solve the Navier-Stokes equations using a solver we have written based on the description of the numerical methods given previously; we use finite different methods for spatial discretizations in a rectangular domain, and the forward Euler method for the time discretization.

4.3.5 Example Application

As a model problem, we consider the steady flow of an incompressible fluid in a square cavity ($0 \leq x \leq 2, 0 \leq y \leq 2$). The flow is induced by the sliding motion of the top wall ($y = 2$) from left to right; see Figure 4.4. The boundary conditions are no-slip conditions with the exception of the upper boundary, for the upper boundary we have $v = 0$, and $u = 1$. This is referred to as a driven cavity.

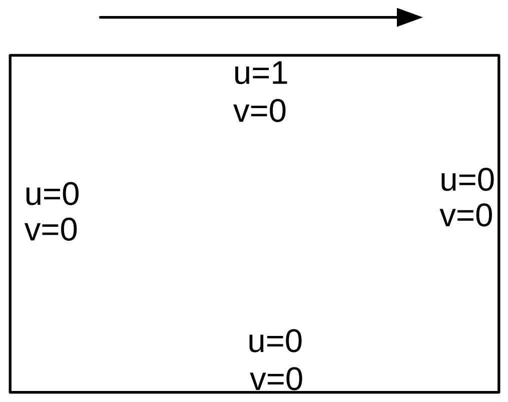

Figure 4.4: Driven Cavity Flow, adapted from [8], page 67.

Figure 4.5 shows the streamlines for $Re = 1000$ at $t = 1, 5, 10, 15$. Figure 4.6 shows the streamlines for $Re = 1, 10, 100, 1000$ at $t = 8$.

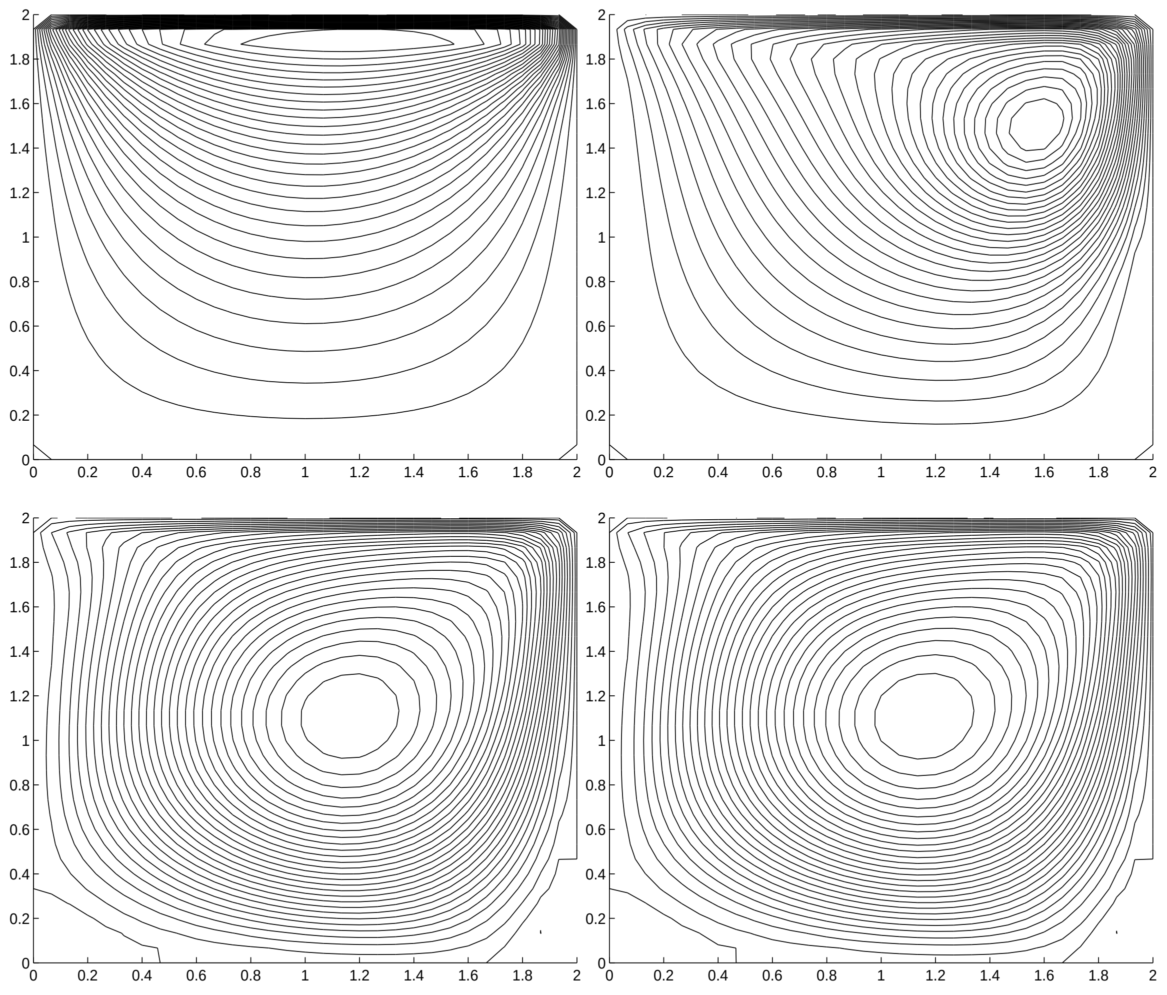

Figure 4.5: Driven cavity; streamlines in x-y plane, at $t = 1, 5, 10, 15$; $Re =$ 1000.

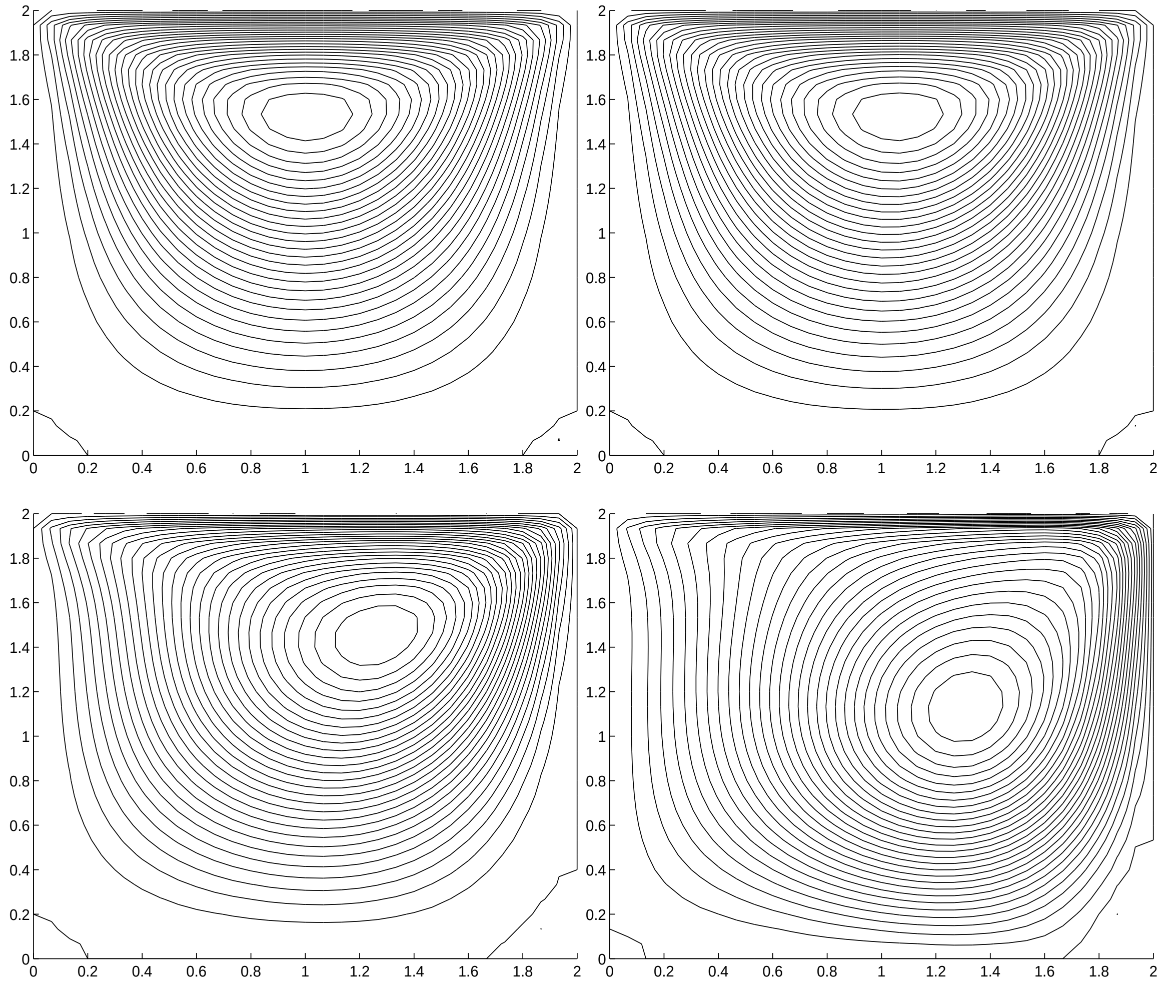

Figure 4.6: Driven cavity; streamlines in x-y plane with Reynolds numbers $1, 10, 100, 1000$ at $t = 8$, (from left to right, top to bottom).

Chapter 5

Numerical Solution of Sloshing Problems

5.1 Introduction

In this chapter, we first review sloshing problems and the substantial literature concerned with the numerical modelling of these problems. We then describe a sloshing problem model based on the shallow water equations and provide some results on the numerical solution of this model. We next examine the numerical solution of a second sloshing problem model and in particular the application of level set methods in the determination of the fluid motion as described by Navier-Stokes equations.

5.2 Sloshing Problems

Because of the complexities of modelling the fluid motion associated with sloshing, an overall analysis using an appropriate numerical method is necessary. However, difficulties arise from the fact that the domain of interest has an unknown boundary or free surface; i.e., the surface of the fluid should be determined as a part of the solution. To treat a free surface numerically, it is necessary to employ an accurate and efficient numerical scheme that can resolve the free surface as it moves continuously with time. This scheme should be able to model a variety of free surface configurations. To describe these flows mathematically, the governing equations and surface conditions must be given. There are a number of mathematical models that can describe a sloshing problem including ones based on the shallow water equations and the Navier-Stokes equations.

Early simulations of sloshing problems were mostly performed with waves of small height. The sloshing height was assumed to be sufficiently small so that the nonlinear effects could be neglected. Many analytical and experimental studies on sloshing were performed in the 1950's and 1960's for tank design of space vehicles [16]. Graham and Rodriguez [7] gave a very thorough analysis of the impulsive and convective pressures in a rectangular container. The most commonly applied idealization for estimating liquid response in rectangular and cylindrical tanks was formulated by Housner [10]. He divided hydrodynamic pressures of the contained liquid into two components: the impulsive pressure caused by the portion of the liquid accelerating with the tank and the convective pressure caused by the portion of the liquid sloshing in the tank. A method associated with the simulation of free surface fluid flow called the Marker and Cell (MAC) method was published by Harlow and Welch [9]. They introduced massless markers that move with the fluid and a novel finite difference algorithm for the velocity field. The massless markers are used to define the location and track the movement of the free surface. Disadvantages of this approach include the additional storage required for locating the marker particles, and the additional programming complexity required to locate the cells containing the free surfaces [25].

In the 1970's and early 1980's, the sloshing problem became an important issue in the design of the liquified natural gas carriers. Several numerical approaches were considered during this time. Nakayama and Washizu [14] modeled nonlinear sloshing by finite element and boundary element methods. They carried out numerical simulations of a two-dimensional liquid under horizontal and pitching periodic ground motions. Mikelis and Journee [12] conducted experimental and numerical simulations of sloshing behaviour in liquid cargo tanks and its effect on ship motions. They used a two-dimensional finite-difference transient solution based on the MAC approach, adapted for the prediction of liquid motions and induced pressures in partially filled ship tanks. Essentially the Navier-Stokes equations are solved for each cell of the computational mesh in conjunction with the appropriate boundary conditions and ancillary equations. The solution is advanced through time in a way that enables viscous transient fluid flow problems to be considered.

In the 1990's, many attempts were made to develop methods for the simulation of sloshing phenomena. In 1992, Hwang [11] employed the *panel method*, which was based on the boundary element method, to investigate the three-dimensional sloshing problem. Szymczk [25] used codes designed to model compressible flow in the study of free surface problems for incompressible fluids. However, these methods are not particularly well suited for treating the long-time motion of nearly incompressible fluids like water. Free surface velocity boundary conditions also play a critical role in the simulation of fluid flow problems involving free surfaces; accurate procedures for the application of free surface velocity boundary conditions are presented in Chen [3].

There has been recent work related to but not specifically about sloshing problems. Sussman [23] developed a level set approach for computing solutions to incompressible two-phase flow. A level set technique is coupled with a projection method. The *projection method* was introduced in 1968 by Chorin [23] as a way of efficiently computing solutions of Navier-Stokes equations for incompressible flow. The method uses backward Euler in time and centered-differencing in space and assumes periodic boundary conditions. The advantage of this approach is that the numerical boundary layers are explicitly characterized. The disadvantage, however, is that it requires far more regularity of the exact solution than is necessary, or realistically expected. An important feature of this method is that it maintains the level set function as a distance function for all time, without reconstructing the interface.

Chen [4] presented a new method for simulation of two-dimensional, incompressible, free surface fluid flow that was called the surface marker and micro cell (SMMC) method. The new method is therefore a marker and cell method, but it differs from the MAC method in one essential way. Surface markers, rather than markers distributed throughout the fluid, are used in the SMMC method. The evolution of the fluid free surface is accomplished by moving the surface markers to new locations. This method is validated by comparison of simulation and experimental results for water sloshing in a tank. The comparison shows good agreement between the shapes and locations of the simulated and experimental free surfaces.

Chang [2] derived a level set formulation for incompressible, immiscible multi-fluid flow. A second-order projection method or velocity-based method can be used to discretize the fluid equations in the level set formulation. There is no explicit tracking of the fluid interface. The fluid interface is recovered at the end of the calculation by locating the zero level set of a smooth function. The effects of discontinuous density, discontinuous viscosity, and surface tension can all be taken into account naturally. The method is robust, efficient, and capable of handling topological change in the fluid interfaces. It can be generalized to three-dimensional problems fairly easily.

In Kim [17], a numerical method is applied for the simulation of fluid flows in two- and three-dimensional tanks. The method has been applied to several models.

In important recent work, Neilson [15] investigated free surface modeling and the sloshing problem in his Ph.D. thesis. The free surface is described by a Volume of Fluid Method (VOF). In the VOF method, the free surface is represented on fixed grids using a fractional fluid volume in each cell. A cell with a volume fraction value of 0 is empty, and a volume fraction value of 1 implies a full cell. A cell with a volume fraction value between 0 and 1, implies that the cell contains a free surface. Initially, all cells are given a volume fraction value corresponding to the initial fluid surface, and at each time step a transport equation is solved to find the distribution of fluid at the new time step. The overall solution algorithm is simple and efficient, and it can also handle complex geometry. However, the VOF-based method has difficulties in determining a free surface location on fixed grids. Numerical smearing on the free surface was not handled effectively, and three-dimensional sloshing problems could not be simulated accurately. Neilson also reports an experimental testing of sloshing in a tank. The free surface from the numerical methods was compared to the one observed in the experiments, and it was found that the computation gave a very good prediction for the sloshing flow.

Sussman [24] investigated a numerical method called the coupled level set and volume of fluid (CLSVOF) method to represent the free surface for several fluid flow problems, but not the sloshing problem. The position of the free surface is updated via the level set equation $\phi_t + \mathbf{V} \cdot \nabla\phi = 0$. The volume fractions are used to express the interfacial curvature to second-order accuracy.

Kim [16] considered a new free surface tracking algorithm based on the VOF method. The novel features of the proposed algorithm are characterized by two numerical tools: the orientation vector used to represent the free surface orientation in each cell and the baby-cell used to determine the fluid volume flux at each cell boundary. The proposed algorithm can be easily implemented on any irregular non-uniform grid, such as usually encountered in the finite element method. Most of the analysis was limited to two-dimensional problems; however the proposed algorithm can be extended and applied to the three-dimensional free surface flow problems without additional efforts. The robustness of the proposed numerical algorithm was also demonstrated.

5.3 Modelling of Sloshing Problems with the Shallow Water Equations

5.3.1 One-dimensional and Two-dimensional Models

If the degree of sloshing is small compared to the depth of the fluid then we can use the shallow water equations to describe the motion of a fluid in a tank. The one-dimensional shallow water equations are as follows.

The mass conservation equation is

$$\frac{\partial H}{\partial t}(x,t) + \frac{\partial (HV)}{\partial x}(x,t) = 0, \tag{5.1}$$

and the momentum conservation equation is

$$\frac{\partial V}{\partial t}(x,t) + \frac{\partial}{\partial x}\left(gH + \frac{V^2}{2}\right)(x,t) = 0, \tag{5.2}$$

where

$x \in [0, L]$ is the spatial coordinate attached to the tank of length L,

$t \in [0, T]$ is the time coordinate, $T > 0$,

g is the gravity constant,

$H(x,t)$ denotes the height of the liquid,

$V(x,t)$ denotes the horizontal speed of the fluid in the coordinates attached to the tank.

The initial conditions are:

$$H(x,0) = H_0(x) \quad \text{and} \quad V(x,0) = V_0(x),$$

where we model a solitary wave by choosing

$$H_0(x) = H\text{sech}(x^2),$$

and

$$V_0(x) = -2\frac{\pi}{gL}\cos\left(\frac{\pi x}{L}\right)\sin\left(\frac{\pi x}{L}\right).$$

The boundary conditions for V are given by, for all $t \in [0,T]$,

$$V(0,t) = 0 \quad \text{and} \quad V(L,t) = 0.$$

The boundary conditions for H are derived from requiring $\frac{d^3(H)}{dx^3} = 0$ at each boundary; a one-sided high-order finite difference approximation gives

$$H(x_1,t) = 3H(x_2,t) - 3H(x_3,t) + H(x_4,t),$$

$$H(x_N,t) = 3H(x_{N-1},t) - 3H(x_{N-2},t) + H(x_{N-3},t).$$

where $x_1,\ x_2,\ \cdots,\ x_N$ are uniformly spaced over the x domain. The two-dimensional shallow water equations are as follows.

The mass conservation equation is

$$\frac{\partial H}{\partial t} + \frac{\partial (HV)}{\partial x} + \frac{\partial (HU)}{\partial y} = 0, \tag{5.3}$$

where H, V are as before, and U denotes the speed of the liquid in the coordinate Y attached to the tank. All functions H, V, U depend on t, x, and y. For the boundary conditions we set $H(x_n, t) = 0$ and the velocity component normal to the boundary is set to zero.

The momentum conservation equations are

$$\frac{\partial (HV)}{\partial t} + \frac{\partial}{\partial x}\left(V^2 H + \frac{gH^2}{2}\right) + \frac{\partial}{\partial y}(HVU) = 0, \tag{5.4}$$

and

$$\frac{\partial (HU)}{\partial t} + \frac{\partial}{\partial x}(HUV) + \frac{\partial}{\partial y}\left(U^2 H + \frac{gH^2}{2}\right) = 0. \tag{5.5}$$

5.3.2 Numerical Treatment of One-dimensional Shallow Water Equations

The system of shallow water equations in one dimension can be written as

$$\frac{\partial}{\partial t}\begin{pmatrix} H(x,t) \\ V(x,t) \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} H(x,t)V(x,t) \\ gH(x,t) + \frac{V^2(x,t)}{2} \end{pmatrix} = 0. \tag{5.6}$$

In (5.6) we can explicitly apply differentiation with respect to x; the two components of the second term of (5.6) become

$$\frac{\partial(H(x,t)V(x,t))}{\partial x} = H_x(x,t)V(x,t) + H(x,t)V_x(x,t), \tag{5.7}$$

$$\frac{\partial}{\partial x}\left(gH(x,t) + \frac{V^2(x,t)}{2}\right) = gH_x(x,t) + V(x,t)V_x(x,t). \tag{5.8}$$

We next rewrite (5.6) by substituting (5.7) and (5.8); we have

$$\frac{\partial}{\partial t}H(x,t) = -\left(H_x(x,t)V(x,t) + H(x,t)V_x(x,t)\right), \tag{5.9}$$

and

$$\frac{\partial}{\partial t}V(x,t) = -\left(gH_x(x,t) + V(x,t)V_x(x,t)\right). \tag{5.10}$$

Applying the forward Euler method gives

$$H(x,t_n) = H(x,t_{n-1}) - \Delta t\left(H_x(x,t_{n-1})V(x,t_{n-1}) + H(x,t_{n-1})V_x(x,t_{n-1})\right), \tag{5.11}$$

$$V(x,t_n) = V(x,t_{n-1}) - \Delta t\left(gH_x(x,t_{n-1}) + V(x,t_{n-1})V_x(x,t_{n-1})\right). \tag{5.12}$$

We discretize the spatial derivatives using the usual central second-order finite difference scheme. This gives

$$H_x(x_i,t) = \frac{H(x_{i+1},t) - H(x_{i-1},t)}{2\Delta x},$$

and

$$V_x(x_i,t) = \frac{V(x_{i+1},t) - V(x_{i-1},t)}{2\Delta x}.$$

5.3.3 Use of a Linear Approximation in the Solution of the One-dimensional Shallow Water Equations

We can rewrite system (5.1), (5.2), in a matrix-vector form [5]

$$\frac{\partial}{\partial t}\begin{pmatrix} H(x,t) \\ V(x,t) \end{pmatrix} + A(H(x,t),V(x,t))\frac{\partial}{\partial x}\begin{pmatrix} H(x,t) \\ V(x,t) \end{pmatrix} = 0, \qquad (5.13)$$

with the characteristic matrix

$$A(H(x,t),V(x,t)) = \begin{pmatrix} V(x,t) & H(x,t) \\ g & V(x,t) \end{pmatrix}.$$

Following the development in [5] we see that the eigenvalues of this matrix are

$$\lambda_1(H,V) = V - \sqrt{gH} \quad \text{and} \quad \lambda_2(H,V) = V + \sqrt{gH}.$$

We can then employ a change of coordinates; let

$$\epsilon_1 = V - V_0 - 2(\sqrt{gH} - \sqrt{gH_0}) \quad \text{and} \quad \epsilon_2 = V - V_0 + 2(\sqrt{gH} - \sqrt{gH_0}). \qquad (5.14)$$

We can then rewrite the system (5.13) in the form:

$$\frac{\partial}{\partial t}\begin{pmatrix} \epsilon_1 \\ \epsilon_2 \end{pmatrix} + \Lambda(\epsilon_1,\epsilon_2)\frac{\partial}{\partial x}\begin{pmatrix} \epsilon_1 \\ \epsilon_2 \end{pmatrix} = 0, \qquad (5.15)$$

where

$$\Lambda(\epsilon_1,\epsilon_2) = \begin{pmatrix} \frac{1}{4}\epsilon_2 + \frac{3}{4}\epsilon_1 + V_0 - \sqrt{gH_0} & 0 \\ 0 & \frac{3}{4}\epsilon_2 + \frac{1}{4}\epsilon_1 + V_0 - \sqrt{gH_0} \end{pmatrix}. \qquad (5.16)$$

Thus H, V can be expressed in terms of ϵ_1 and ϵ_2 as

$$H = \frac{(\epsilon_2 - \epsilon_1 + 4\sqrt{gH_0})^2}{16g} \quad \text{and} \quad V = \frac{\epsilon_2 + \epsilon_1}{2} + V_0. \tag{5.17}$$

Rather than solve the system (5.15) directly, de Halleux [5] indicates that he employs a linear approximation for ϵ_1 and ϵ_2 in (5.17) to compute H and V. However, he dose not give explicit details for this approximation. Here we employ our own linear approximation for ϵ_1, ϵ_2. From equation (5.14), we have: $\epsilon_1 = V - V_0 - \Delta\epsilon$ and $\epsilon_2 = V - V_0 + \Delta\epsilon$, where $\Delta\epsilon = 2(\sqrt{gH} - \sqrt{gH_0})$. Since g and H_0 are known, our approximation for $\Delta\epsilon$ (and thus ϵ_1 and ϵ_2) is obtained once we have an approximation for H at the current time t. A linear approximation for $H(t)$ comes at no cost from $H(t - \Delta t)$, the H value at the previous time step. A simple Taylor series argument shows that $H(t - \Delta t)$ is an $O(\Delta t)$ approximation to $H(t)$. This approximation for H is employed in the above expression for $\Delta\epsilon$ which is then used to get approximations for ϵ_1, ϵ_2 and from (5.17), we then get linear approximations for H and V at the current time.

5.3.4 Use of a Linear Approximation in the Solution of the Two-dimensional Shallow Water Equations

We can rewrite the system (5.3), (5.4), and (5.5) in a matrix-vector form [5] as

$$\frac{\partial}{\partial t}\begin{pmatrix} H \\ HV \\ HU \end{pmatrix} + A\frac{\partial}{\partial x}\begin{pmatrix} H \\ HV \\ HU \end{pmatrix} + B\frac{\partial}{\partial y}\begin{pmatrix} H \\ HV \\ HU \end{pmatrix} = 0, \tag{5.18}$$

with the characteristic matrices, [5]

$$A = \begin{pmatrix} 0 & 1 & 0 \\ -V^2 + gH & 2V & 0 \\ -VU & U & V \end{pmatrix},$$

and

$$B = \begin{pmatrix} 0 & 0 & 1 \\ -VU & U & V \\ -U^2 + gH & 0 & 2U \end{pmatrix}.$$

The eigenvalues of the matrix A are

$$\lambda_1 = V - \sqrt{gH}, \lambda_2 = V + \sqrt{gH} \quad \text{and} \quad \lambda_3 = V.$$

The eigenvalues of the matrix B are

$$\gamma_1 = U - \sqrt{gH}, \gamma_2 = U + \sqrt{gH} \quad \text{and} \quad \gamma_3 = U.$$

We can then employ the following change of coordinates; let

$$\epsilon_1 = V - V_0 - (\sqrt{gH} - \sqrt{gH_0}) \quad \text{and} \quad \epsilon_2 = V - V_0 + (\sqrt{gH} - \sqrt{gH_0}). \quad (5.19)$$

$$\eta_1 = U - U_0 - (\sqrt{gH} - \sqrt{gH_0}) \quad \text{and} \quad \eta_2 = U - U_0 + (\sqrt{gH} - \sqrt{gH_0}). \quad (5.20)$$

We can then rewrite the system (5.18) in the form

$$\frac{\partial}{\partial t}\begin{pmatrix} \epsilon_1 \\ \\ \epsilon_2 \end{pmatrix} + \Lambda(\epsilon_1, \epsilon_2)\frac{\partial}{\partial x}\begin{pmatrix} \epsilon_1 \\ \\ \epsilon_2 \end{pmatrix} = 0, \quad (5.21)$$

where $\Lambda(\epsilon_1, \epsilon_2)$ is the same as in (5.16) and

$$\frac{\partial}{\partial t}\begin{pmatrix} \eta_1 \\ \\ \eta_2 \end{pmatrix} + \Gamma(\eta_1, \eta_2)\frac{\partial}{\partial y}\begin{pmatrix} \eta_1 \\ \\ \eta_2 \end{pmatrix} = 0, \quad (5.22)$$

where

$$\Gamma(\eta_1, \eta_2) = \begin{pmatrix} \frac{1}{4}\eta_2 + \frac{3}{4}\eta_1 + U_0 - \sqrt{gH_0} & 0 \\ \\ 0 & \frac{3}{4}\eta_2 + \frac{1}{4}\eta_1 + U_0 - \sqrt{gH_0} \end{pmatrix}. \quad (5.23)$$

H, V, and U can then be expressed in terms of ϵ_1, ϵ_2, η_1, and η_2 as

$$H = \frac{(\epsilon_2 - \epsilon_1 + \eta_2 - \eta_1 + 4\sqrt{gH_0})^2}{16g}, \quad (5.24)$$

$$V = \frac{\epsilon_2 + \epsilon_1}{2} + V_0, \quad \text{and} \quad U = \frac{\eta_2 + \eta_1}{2} + U_0. \quad (5.25)$$

As in the one-dimensional case we consider linear approximations for ϵ_1, ϵ_2, η_1, η_2 obtained as follows. From (5.19) and (5.20) we observe that

$$\epsilon_1 = V - V_0 - \Delta\epsilon \quad \text{and} \quad \epsilon_2 = V - V_0 + \Delta\epsilon,$$

$$\eta_1 = U - U_0 - \Delta\eta \quad \text{and} \quad \eta_2 = U - U_0 + \Delta\eta,$$

where $\Delta\epsilon = \Delta\eta = (\sqrt{gH} - \sqrt{gH_0})$. A similar analysis therefore holds and linear approximations for H, V and U are obtained as in (5.24), (5.25).

5.3.5 Numerical Results for One-dimensional Shallow Water Equations

In this section we provide numerical results associated with the solution of the nonlinear shallow water equations, in discretized form (5.6). The computing environment is MATLAB 6.1 running under Microsoft Windows XP; the machine type is a Dell Dptiplex Gx270 personal computer, and we have $L = 1$, $T = 4$, $\Delta t = 0.0004$, and $\Delta x = 0.005$. Figures generated from the numerical results are shown in Figures 5.1 and 5.2.

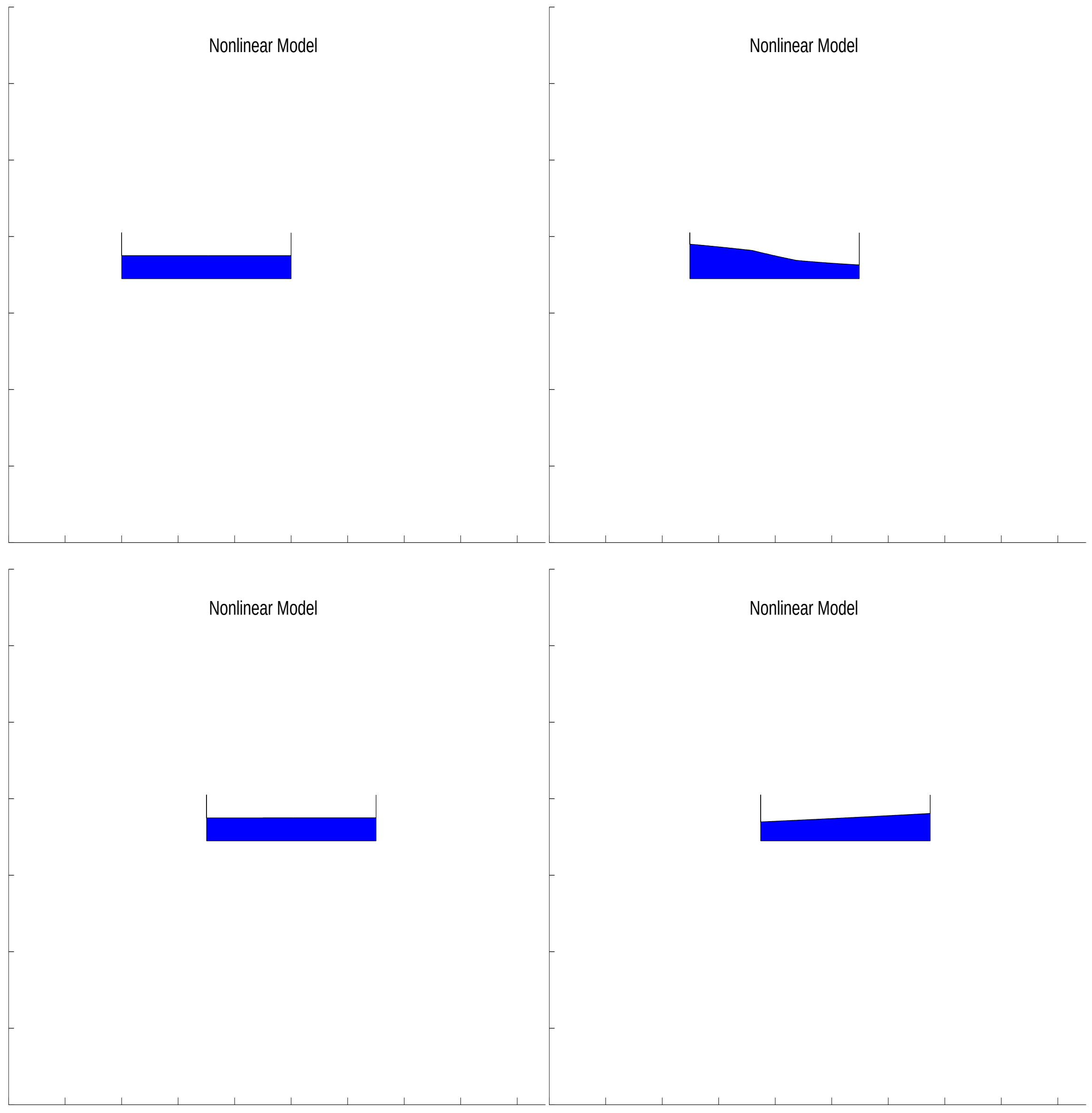

Figure 5.1: Shallow water equations in the x-y plane; nonlinear one-dimensional model of sloshing at $t = 0.0, 1.0, 1.5, 2.0$, from left to right, top to bottom.

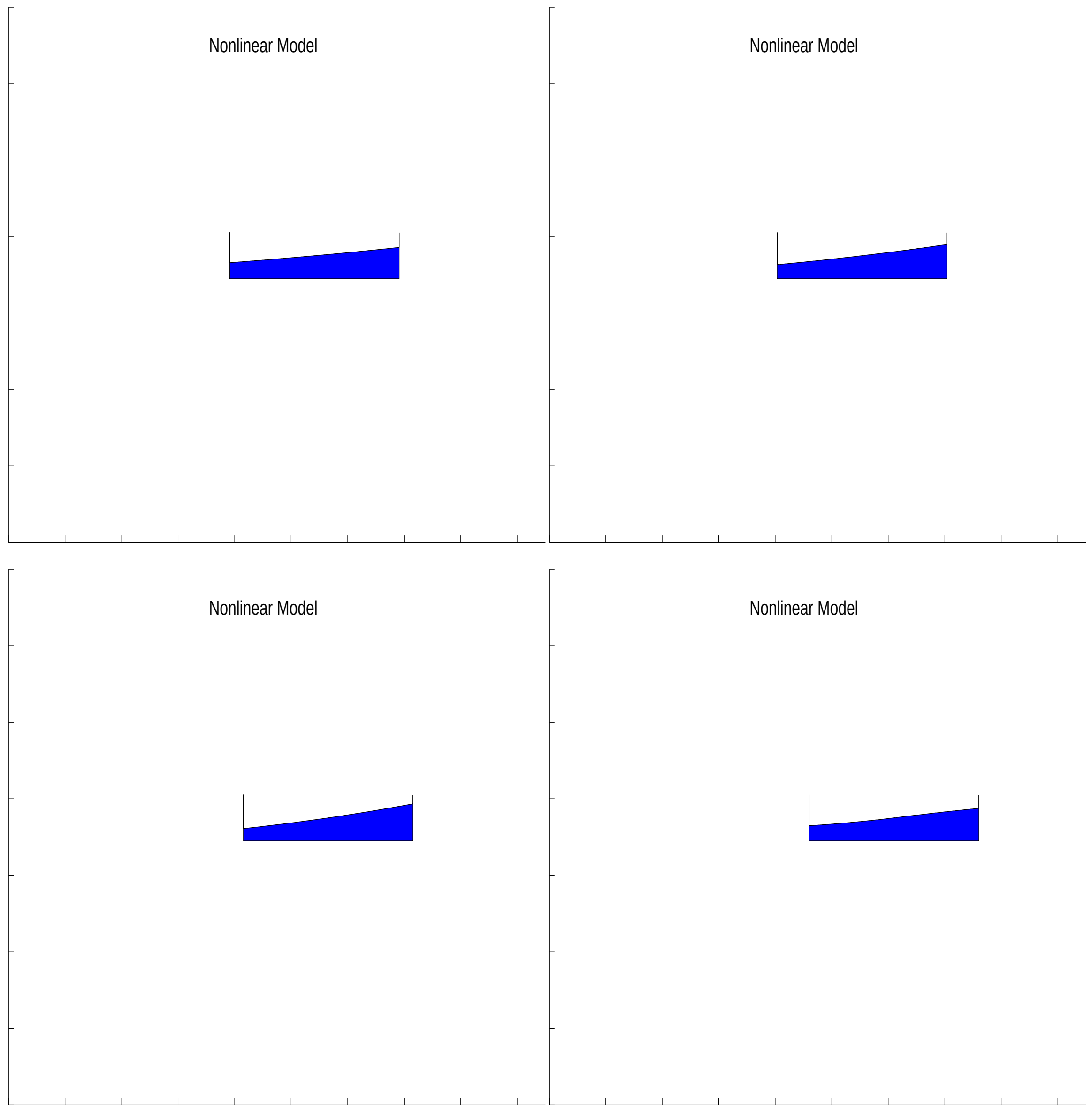

Figure 5.2: Shallow water equations in the x-y plane; nonlinear one-dimensional model of sloshing at $t = 2.5, 3.0, 3.5, 4.0$, from left to right, top to bottom.

5.3.6 Numerical Results for a Linear Approximation in the Solution of the One-dimensional Shallow Water Equations

In this section we present numerical results based on a linear approximation in the one-dimensional shallow water equations (5.6). The same computing environment and parameter values were employed. The results are shown in Figures 5.3 and 5.4. We note that while the implementation involving the linear approximation is simpler, the results of the linear case do differ somewhat from those obtained from the direct treatment of the numerical model.

5.3.7 Numerical Results for a Linear Approximation in the Solution of the Two-dimensional Shallow Water Equations

In this section we present numerical results based on a linear approximation in the two-dimensional shallow water equations (5.18). The same computing environment and parameter values were employed. The results are shown in Figures 5.5 and 5.6.

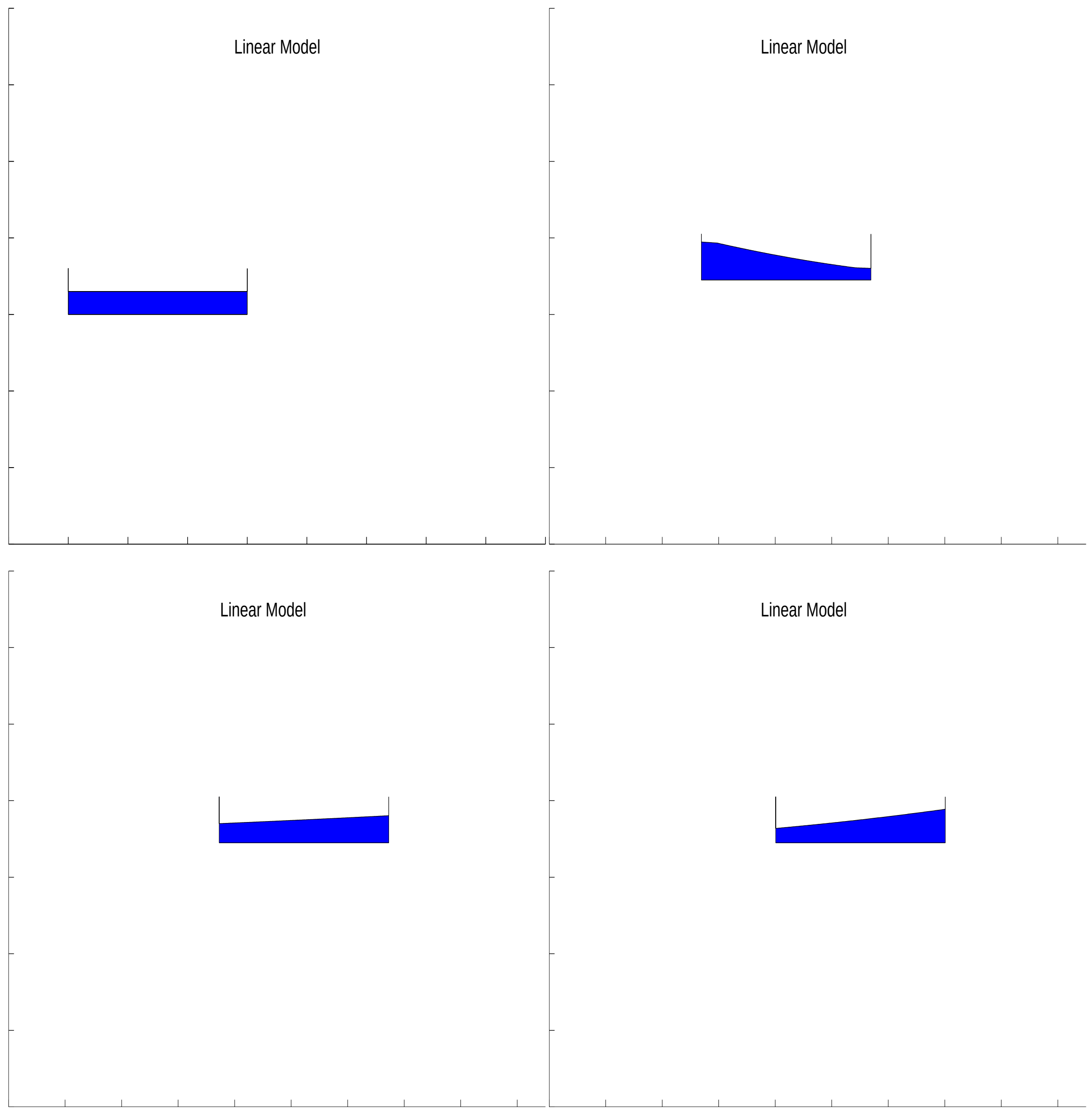

Figure 5.3: Shallow water equations in the x-y plane; solution based on a linear approximation in the one-dimensional model of sloshing at $t = 0.0, 1.0, 1.5, 2.0$, from left to right, top to bottom.

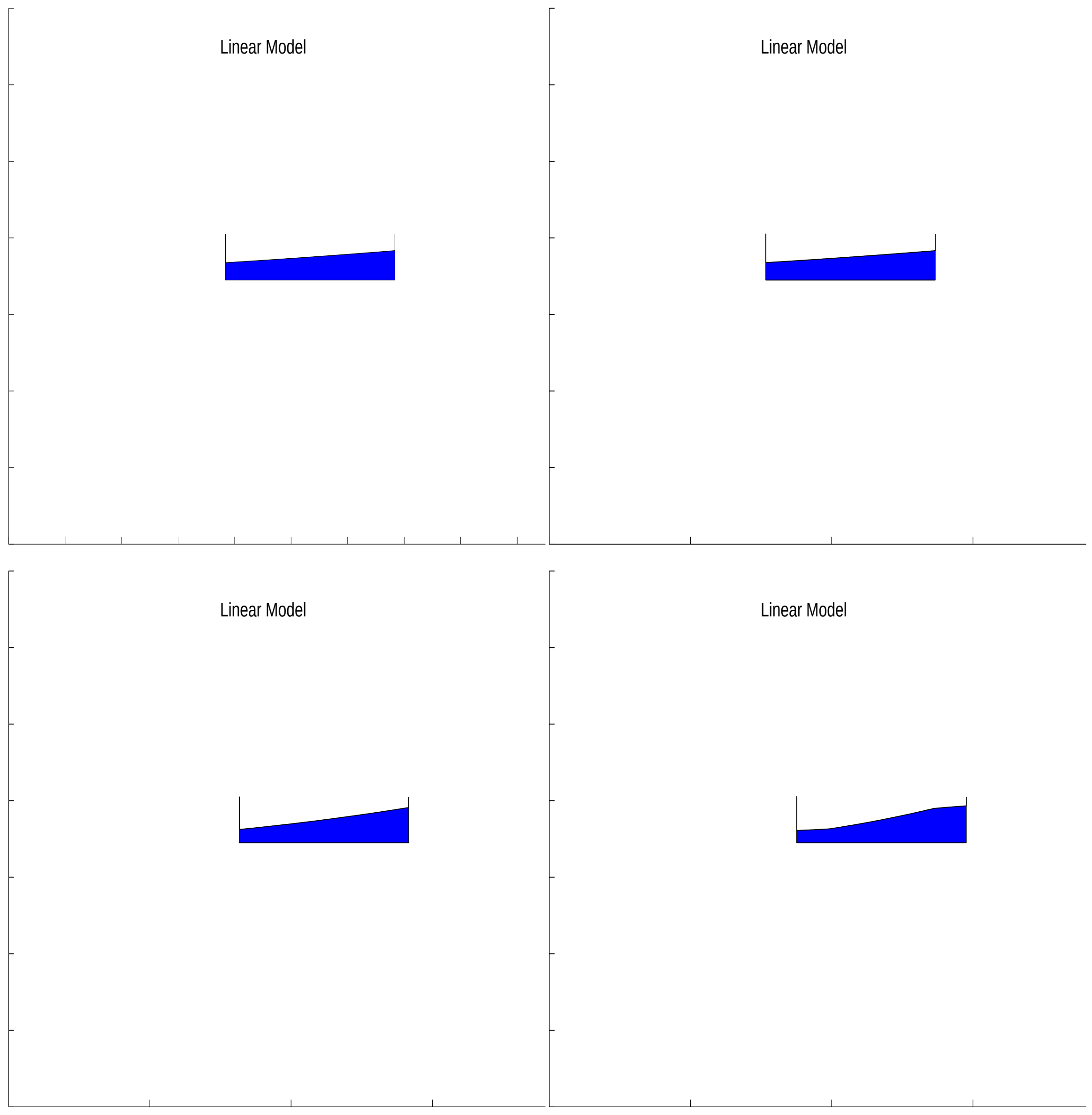

Figure 5.4: Shallow water equations in the x-y plane; solution based on a linear approximation in the one-dimensional model of sloshing at $t = 2.5, 3.0, 3.5, 4.0$, from left to right, top to bottom.

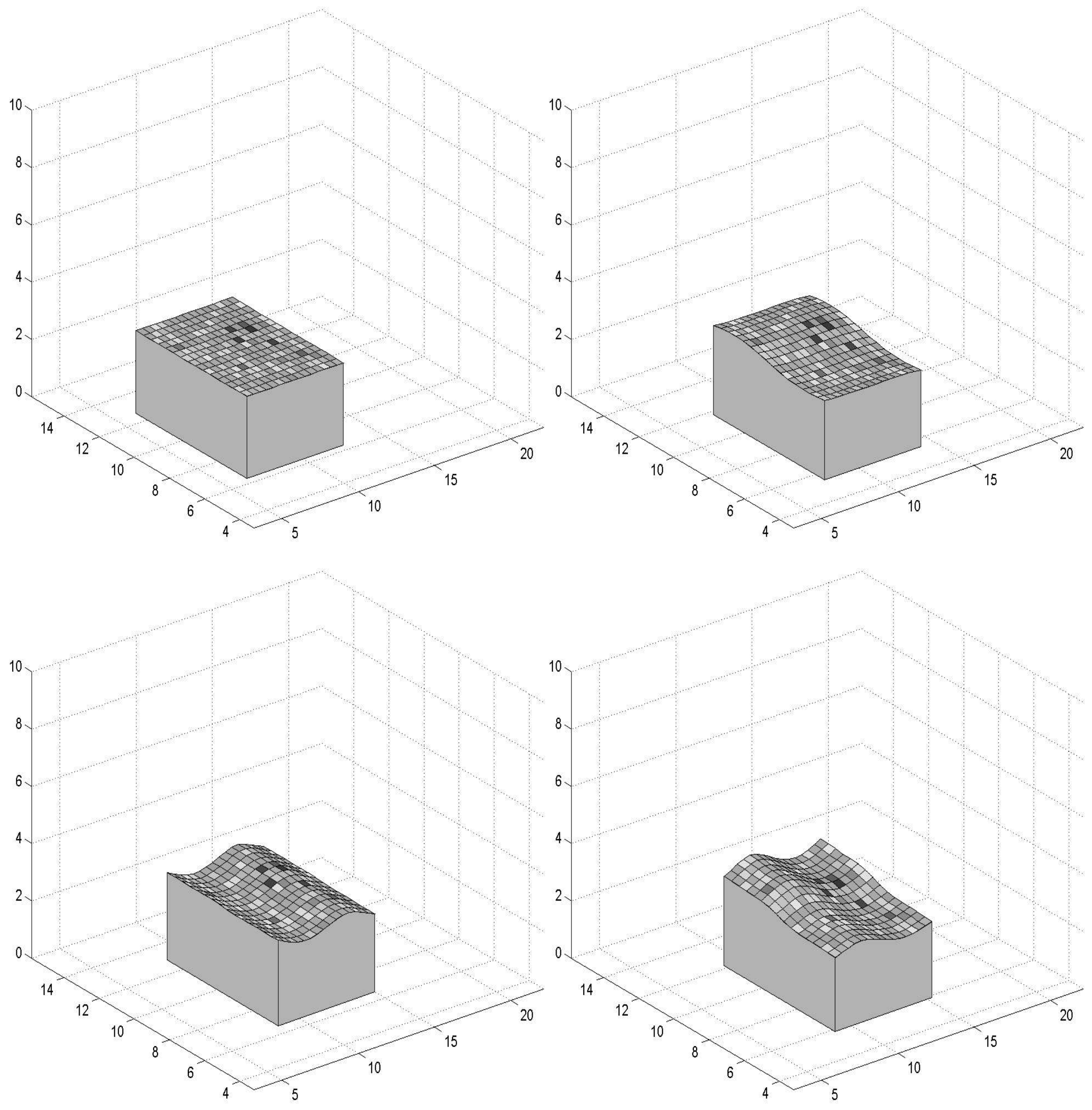

Figure 5.5: Shallow water equations in x-y-z space; solution based on a linear approximation in the two-dimensional model of sloshing at $t = 0.0, 1.0, 1.5, 2.0$, from left to right, top to bottom.

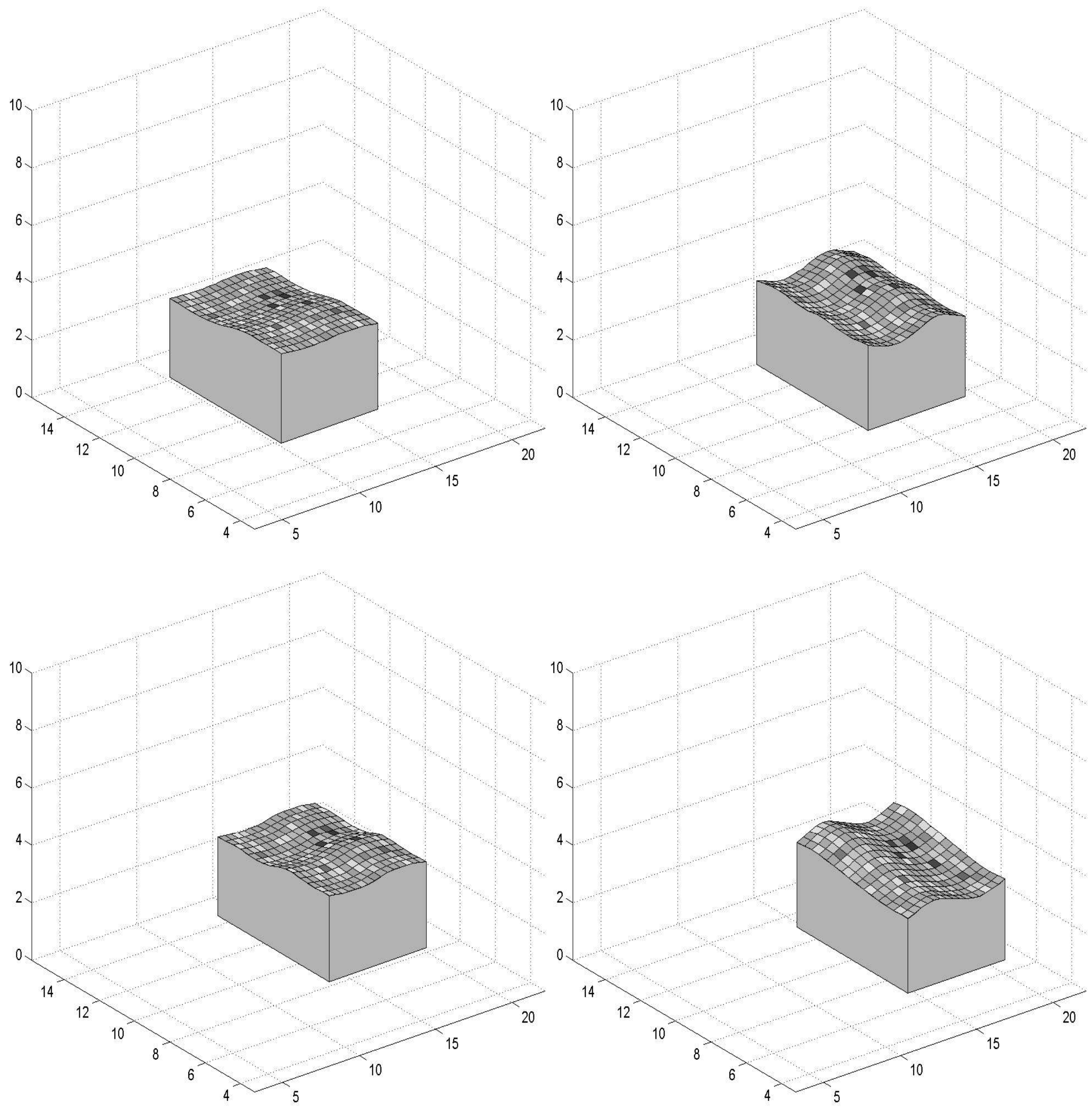

Figure 5.6: Shallow water equations in x-y-z space; solution based on a linear approximation in the two-dimensional model of sloshing at $t = 2.5, 3.0, 3.5, 4.0$, from left to right, top to bottom.

5.4 Level Set Methods Coupled with Navier-Stokes Equations

In the remainder of this chapter, we present a preliminary investigation in which we attempt to couple the level set method with Navier-Stokes equations for the modelling of sloshing problems. We implement the level set component of this simulation using the level set toolbox described in Chapter 3 and implement the Navier-Stokes component using the Navier-Stokes solver described in Chapter 4.

5.4.1 Governing Equations

We recall the dimensionless incompressible Navier-Stokes equations: the momentum equation is

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \frac{1}{Re}\nabla^2 \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u} + \mathbf{g} \tag{5.26}$$

and the continuity equation is

$$\nabla \cdot \mathbf{u} = 0, \tag{5.27}$$

where $\mathbf{u}$ is the velocity vector, p is the pressure, Re is the Reynolds number, (we choose $Re = 100$), and $\mathbf{g}$ denotes body forces such as gravity; we initially consider $\mathbf{g} = (9.8, 0)^T$ corresponding to the presence of the force of gravity in the vertical direction. For two-dimensional incompressible flows, $\mathbf{u} = (u, v)^T$, $\mathbf{x} = (x, y)^T$, $\mathbf{g} = (g_x, g_y)^T$, and $\mathbf{u}$ is dependent on $\mathbf{x}$; i.e., $\mathbf{u} = \mathbf{u}(\mathbf{x}, t)$. We take the domain to be the box $[-1, 1] \times [-1, 1]$. We discretize these equations as described in the previous chapters and use a MATLAB implementation to compute the horizontal and vertical fluid velocities by solving the discretized Navier-Stokes equations. The MATLAB source code is shown in the Appendix.

Boundary conditions along the lower and upper horizontal segments of our domain are no-slip conditions; boundary conditions along the left and right vertical segments are no-slip conditions as well. The discretized form of the boundary conditions is therefore

$$u_{0,j} = 0, \qquad u_{i_{max},j} = 0, \qquad j = 1, \cdots, j_{max},$$

$$v_{0,j} = -v_{1,j}, \qquad v_{i_{max}+1,j} = -v_{i_{max},j}, \qquad j = 1, \cdots, j_{max},$$

$$u_{i,0} = -u_{i,1}, \qquad u_{i,j_{max}+1} = -u_{i,j_{max}}, \qquad i = 1, \cdots, i_{max}.$$

$$v_{i,0} = 0, \qquad v_{i,j_{max}} = 0, \qquad i = 1, \cdots, i_{max}.$$

The initial conditions we use are $u(\mathbf{x}, 0) = u_0(\mathbf{x}) = 2$, and $v(\mathbf{x}, 0) = v_0(\mathbf{x}) = 0$. These are the same conditions as used in [8] for a similar problem.

We observe that while the Navier-Stokes equations allow us to compute the velocity as it changes in time, the motion of the surface of the fluid is not considered. Thus in our approach the position of the free surface is updated via the level set equation

$$\phi_t + \mathbf{u} \cdot \nabla\phi = 0,$$

where ϕ is the level set function whose zero contour will give the surface of the fluid and $\mathbf{u}$ is the fluid velocity vector as in equations (5.26) and (5.27). We choose the initial value for the level set function to be $\phi(\mathbf{x}, 0) = 0.5\sin(0.76\pi x + 0.5) + 0.24\pi y$. See Figure 5.8, $t = 0$, to see the initial fluid surface.

5.4.2 Implementation

We modified `convectionDemo`, which was introduced in Chapter 3, to compute level sets based on the velocity results of the Navier-Stokes solver. The level set toolbox and the Navier-Stokes solver pass information back and forth. The source code is given in the Appendix A.1.

The level set toolbox includes four spatial derivative approximations, which were introduced in Section 3.3. We used the default method `upwindFirstFirst` (First-order upwind scheme). We also have the option of using the methods:

`upwindFirstENO2`, a second-order ENO scheme, `upwindFirstENO3`, a third-order ENO scheme, and `upwindFirstWENO5`, a fifth order WENO.

Three time derivative approximations are provided in the level set toolbox introduced in Section 3.3. We used the default method `odeCFL1` (forward Euler method). We also have the option of using the methods: `odeCFL2`, second-order TVD RK scheme, `odeCFL3`, a third-order TVD RK scheme, are available as well. These functions handle the CFL condition automatically.

We choose the boundary conditions by calling `addGhostNeumann` (default for the level set toolbox); the boundary cells will have a constant specified derivative normal to the boundary; i.e., Neumann boundary conditions are specified and the derivative is set to 1.

5.4.3 Simulation Results

The results are shown in Figure 5.7. This model describes the sloshing of a fluid in a stationary tank, with the initial fluid surface configuration shown in the upper left hand corner of Figure 5.7. This is a restricted version of the sloshing problem since the tank is stationary. We see the settling down of a fluid surface starting from the shape shown in Fig 5.7 for $t = 0$.

For our second set of results, we will consider introducing a horizontal

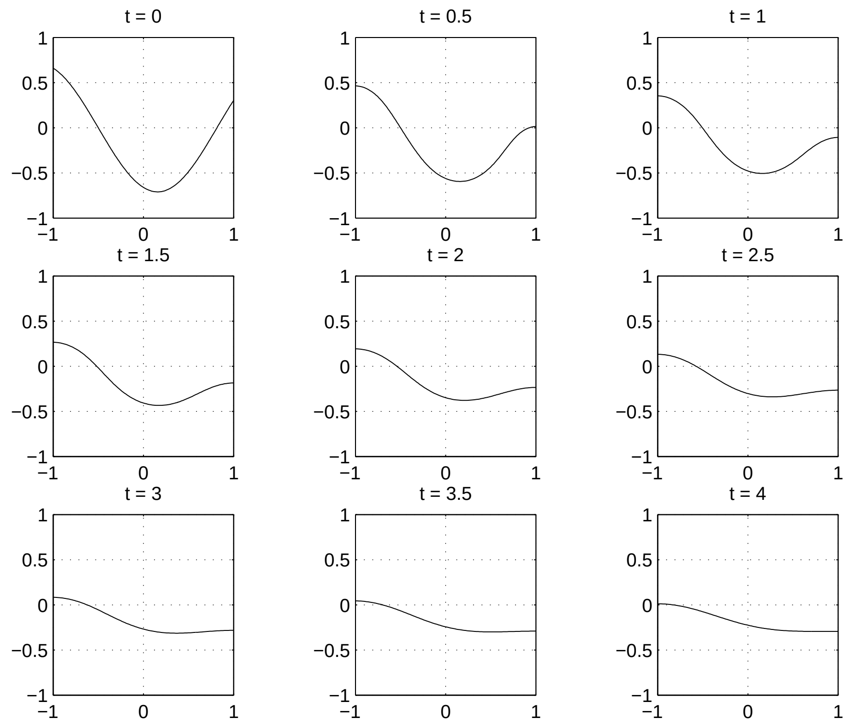

Figure 5.7: Simulation using the Navier-Stokes equations and level set equations, $t \in [0, 4]$, $Re = 100$, stationary tank.

force which simulates the motion of the tank. In our first set of results the forcing term was $\mathbf{g} = (9.8, 0)^T$ representing the vertical force of gravitiy and the absence of any horizontal force. Here we choose $\mathbf{g} = (9.8, 5)^T$ to represent a horizontal motion. The results are shown in Figure 5.8.

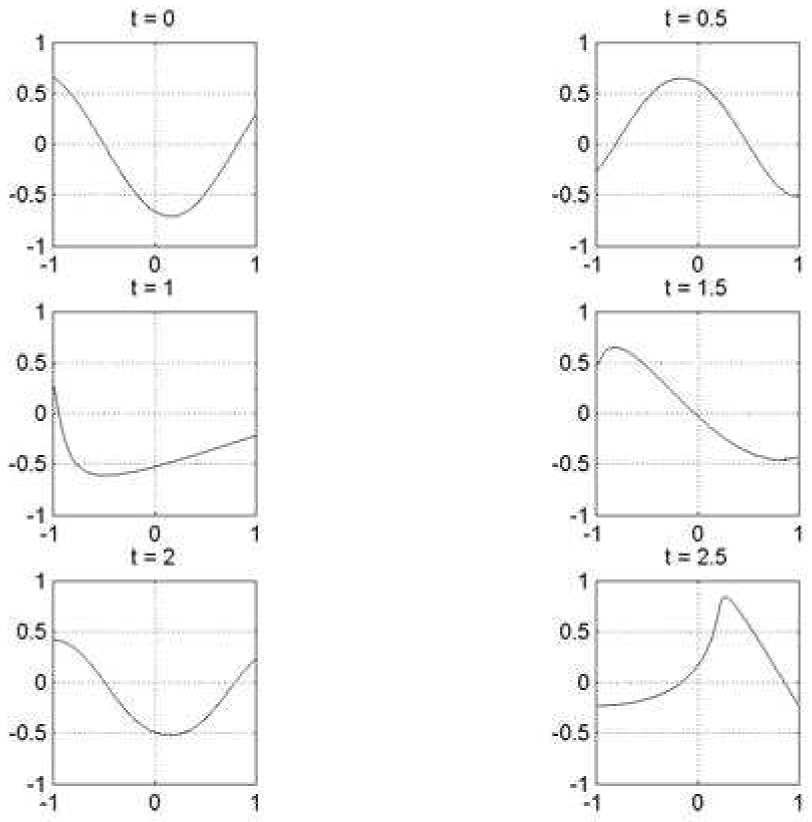

Figure 5.8: Simulation using the Navier-Stokes equations and level set equations, $t \in [0, 2.5]$, $Re = 100$, horizontal tank motion.

Chapter 6

Conclusions and Future Work

In this thesis, we have provided a survey of level set methods for the implicit tracking of interfaces in an elegant manner. We have also provided a review of the MATLAB based level set toolbox that provides many useful tools for the implementation of level set methods [13]. The Navier-Stokes equation and models for their numerical solution have been considered as well. This thesis has considered mathematical models and numerical methods for an important class of fluid flow problems known as sloshing problems, which have applications in areas such as the automotive, aerospace, and ship-building industries. In particular, we have applied both the shallow water equations and the Navier-Stokes equations to model sloshing problems. In the latter case, the fluid interface is not treated explicitly by the model, and we have explored the coupling of the level set approach for the tracking of the interface.

One possible area for further work following from this thesis is the development and numerical solution of the three-dimensional sloshing problem. Although we have considered this briefly within the shallow water equations model, it would be interesting to examine a three-dimensional Navier-Stokes model, coupled with a three-dimensional level set approach, for tracking the interface surface. Other possibilities include consideration of related fluid flows problems or implementations in standard programming language environment such as C++ or Fortran where we could expect the implementation to be much more efficient.

Appendix A

Appendix

A.1 Level Set Method - convectionSlosh.m

```
%Modification of convectionDemo.m to

%implement use of level set methods

%for the sloshing problem.


function [ data, g, data0 ]

 = convectionSlosh(accuracy,initial,displayType)

% [ data, g, data0 ]

%= convectionSlosh(accuracy, initial, displayType)
```

```
% sloshDemo: demonstrate sloshing problem.

% This function was originally designed as a script file,
%so most of the options can only be modified by editing the
% file.
% For example, edit the file to change the grid dimension,
% boundary conditions, flow field parameters, etc.
% Parameters:
%   accuracy     Controls the order of approximation.
% Note that the spatial approximation is always second order.
%                  'low'         Use odeCFL1.
%                  'medium'      Use odeCFL2 (default).
%                  'high'        Use odeCFL3.
%   displayType  Sinuoidal function.
%   data         Implicit surface function at t_max.
%   g            Grid structure on which data was computed.
%   data0        Implicit surface function at t_0.
%-------------------------------------------------------------
% You will see many executable lines that are commented out.
% These are included to show some of the options available;

% modify the commenting to modify the behavior.

%---------------------------------------------------------------

% Make sure we can see the kernel m-files.

run('../addPathToKernel');

%---------------------------------------------------------------

%Call function nsvelv in order to

%get velocity from Navier-Stokes solver.

 v = @nsvelV;

% Reinitialize to get signed distance function

% at the beginning.

reinitToStart = 0;

%---------------------------------------------------------------

% Time integration parameters.

tMax = 4;        % End time.

plotSteps = 9;  % How many intermediate plots to produce?

t0 = 0;          % Start time.

singleStep = 0;   % Plot at each timestep (overrides tPlot).

% Period at which intermediate plots should be produced.
```

```
tPlot = (tMax - t0) / (plotSteps - 1);

% How close (relative) do we need to get to

% tMax to be considered finished?

small = 1000 * eps;

%---------------------------------------------------------------

% What level set should we view?

level = 0;

% Pause after each plot?

pauseAfterPlot = 0;

% Delete previous plot before showing next?

deleteLastPlot = 0;

% Plot in separate subplots

%(set deleteLastPlot = 0 in this case.

useSubplots = 1;

%---------------------------------------------------------------

% Use periodic boundary conditions?

periodic = 0;

% Create the grid.

g.dim = 2;
```

```
g.min = -1;
g.dx = 1 / 50;
if(periodic)
  g.max = (1 - g.dx);
  g.bdry = @addGhostPeriodic;
else
  g.max = +1;
  g.bdry = @addGhostNeumann;
end
g = processGrid(g);


%---------------------------------------------------------------
% What kind of display?
if(nargin < 3)
  switch(g.dim)
   case 1
    displayType = 'plot';
   case 2
    displayType = 'contour';
   case 3
    displayType = 'surface';
   otherwise
error('Default display type undefined for dimension %d',g.dim);
  end
end
%---------------------------------------------------------------
% Create initial conditions (sinusoidal function).
data = 0.5*sin(0.76* pi * (g.xs{1} + 0.5)) + 2 * g.xs{2};
data0 = data;
% Reinitialize if requested (with same level
%of accuracy as main
% computation).
if(reinitToStart)
% The maximum travel of the reinitialization wavefront should
% only be about a quarter of the grid size.
  tMaxReinit = 0.25 * norm(g.max - g.min);
  % We're willing to quit early if the results look good.
  errorMax = 0.01;
```

```
 data=
signedDistanceIterative(g,data,tMaxReinit,errorMax,accuracy);
end
%-------------------------------------------------------------
% Set up fluid motion with velocity.
schemeFunc = @termConvection;
schemeData.grid = g;
schemeData.b = v;
%-------------------------------------------------------------
if(nargin < 1)
  accuracy = 'low';
end
% Set up time approximation scheme.
integratorOptions = odeCFLset('factorCFL',0.5,'stats','on');
%matlab integrator routine
% Choose approximations at appropriate level of accuracy.
switch(accuracy)
 case 'low'
  integratorFunc = @odeCFL1;
 case 'medium'

  integratorFunc = @odeCFL2;

 case 'high'

  integratorFunc = @odeCFL3;

 otherwise

  error('Unknown accuracy level %s', accuracy);

end

if(singleStep)

integratorOptions=odeCFLset(integratorOptions,'singleStep','on');

end

%-----------------------------------------------------------------

% Initialize Display

f = figure;

% Set up subplot parameters if necessary.

if(useSubplots)

  rows = ceil(sqrt(plotSteps));

  cols = ceil(plotSteps / rows);

  plotNum = 1;

  subplot(rows, cols, plotNum);
```

```
end

h=

visualizeLevelSet(g,data,displayType,level,['t='num2str(t0) ]);

hold on;

if(g.dim > 1)

  axis(g.axis);

  daspect([ 1 1 1 ]);

end

%---------------------------------------------------------------

% Loop until tMax (subject to a little roundoff).

tNow = t0;

startTime = cputime;

while(tMax - tNow > small * tMax)

% Reshape data array into column vector for ode solver call.

  y0 = data(:);

% How far to step?

  tSpan = [ tNow, min(tMax, tNow + tPlot) ];

% Take a timestep.

  [ t y ] = feval(integratorFunc, schemeFunc, tSpan, y0,...
                  integratorOptions, schemeData);
  tNow = t(end);
  % Get back the correctly shaped data array
  data = reshape(y, g.shape);
  if(pauseAfterPlot)
  % Wait for last plot to be digested.
    pause;
  end
  % Get correct figure, and remember its current view.
  figure(f);
  figureView = view;
  % Delete last visualization if necessary.
  if(deleteLastPlot)
    delete(h);
  end
 % Move to next subplot if necessary.
  if(useSubplots)
    plotNum = plotNum + 1;
    subplot(rows, cols, plotNum);
```

```
  end
 % Create new visualization.
 h=
visualizeLevelSet(g,data,displayType,level,['t='num2str(tNow)]);
 % Restore view.
  view(figureView);
 end
 endTime = cputime;
fprintf('Total execution time %g seconds', endTime - startTime);
%  out = nsvelV(t, data, schemeData)
% Parameters:
%   t            Current time.
%   data         Level set function.
%   schemeData   Structure.
function out = nsvelV(t, data, schemeData)
global levelset_t levelset_vel;
% call the Navier-Stokes solver
  ns1;
  levelset_t = t; %initialize the end time.
  out=levelset_vel; %retrieve the velocity.

end
```

A.2 Navier-Stokes Solver- NS1.m

```
%NS1

% Numerical solution of 2-D incompressible

% Navier-Stokes equations in rectangular domain.

% Functions called: problem_specification, grid_generation,

% viscous_matrix, grad_and_div, initial_condition,

% inertia_matrix, streamlineplot,

time0 = clock;

global geval n u0 v0 J K yu yv yseglen levelset_t levelset_vel

% .......................................Input..............

geval = 1; % Enter 1 for horizontal flow to the right

%  or 2 for driven cavity

problem_specification


% ................................End of input.............
```

```
grid_generation

viscous_matrix

 % Generate viscous matrices Bu, Bv and Masku, Maskv

grad_and_div% Generate pressure gradient matrices

% Pu, Pv and initial_condition

% Generate initial u0, v0, p0

 [Lp,Up] = lu([Du Dv]*[Pu;Pv]);

% LU decomposition of pressure correction matrix

nstep = floor(tend/dt); % Number of time steps

t = 0; n = 0;

u1 = u0; v1 = v0;

for n = 1:nstep

  t = n*dt;

  u0 = u1; v0 = v1;

inertia_matrix % Generate inertia matrices

% Navier-Stokes, predictor:

  rum = ru'; rum = rum(:); rvm = rv'; rvm = rvm(:);

  if n == 1, tic, end

  u1 = Bu\(dt*(rum - Pu*p0) + Masku*u0);
  v1 = Bv\(dt*(rvm - Pv*p0) + Maskv*v0);

  End of Stokes

  dp = Up\(Lp\(Du*u1 + Dv*v1)); p0 = p0 + dp/dt;

  u1 = u1 - Pu*dp; v1 = v1 - Pv*dp;

end

streamlineplot

% Plot of streamlines and velocity vectors

% PROBLEM SPECIFICATION

% Specification of parameters, grid and boundary conditions

global Re levelset_t

tic

if geval == 1 % Horizontal Poiseuille flow to the right

  dt = 0.125; % Time step

  tend = levelset_t; % End time

  Re = 1500;% Reynolds number based

 % on unit length and unit velocity

  umax = 0.75;% Estimates of max.

  % velocity components required for stability

  vmax = 0;
  xseglen = [1,1];% Enter segment lengths of horizontal
            %boundary in order of increasing x. Number
            %of segments is arbitrary.
% These segments are used for grid generation
            %and boundary conditions.
  yseglen = [1,1]; % Similar to xseglen in y-direction.
  nx = [4,4]; % Number of cells along x-segments.
  ny = [4,4]; % Number of cells along y-segments.
  % Type of boundary condition: 1: no-slip
  % 2: inflow
  % 3: outflow
  % Give type of boundary condition along lower (first row)
  % and upper (second row) horizontal segments in order of
  % increasing x:
  xbc = [1,1; 1,1];
  % Give type of boundary condition along left (first row)
  % and right (second row) vertical segments in order of
  % increasing y:
  ybc = [1,1; 1,1];
  elseif geval == 2 % Driven cavity

  dt = 0.05; % Time step

  tend = 2;% End time

  Re = 1000;% Reynolds number based on unit length and

  % unit velocity

  umax = 1; % Estimates of max. velocity components required

  % for stability

  vmax = 0;


  xseglen = [1,1];% Enter segment lengths of horizontal

            %boundary in order of increasing x.

            %Number of segments is arbitrary.

% These segments are used for grid

            %generation and boundary conditions.

  yseglen = [1,1]; % Similar to xseglen in y-direction.

  nx = [15,15]; % Number of cells along x-segments.

  ny = [15,15]; % Number of cells along y-segments.

  % Type of boundary condition: 1: no-slip

  % 2: inflow
  % 3: outflow
  % Give type of boundary condition along lower (first row)
  % and upper (second row) horizontal segments in order of
  % increasing x:
  xbc = [1,1; 2,2];
  % Give type of boundary condition along left (first row)
  % and right (second row) vertical segments in order of
  % increasing y:
  ybc = [1,1; 1,1];
  else
  error('Wrong value in input for parameter geval')
end
if (size(xbc(1,:))~=size(nx))|(size(ybc(1,:))~=size(nx))
error('Wrong correspondence in problem_specification')
end
% GRID_GENERATION
global xu yu xv yv dx dy
tic
dx = []; % Size of primary (i.e. pressure) cell
```

```
for s = 1:length(nx)
  dx = [dx, xseglen(s)*ones(1,nx(s))/nx(s)];
end
dy = []; % Size of primary (i.e. pressure) cell
for s = 1:length(ny)
  dy = [dy, yseglen(s)*ones(1,ny(s))/ny(s)];
end
[DX,DY] = meshgrid(dx,dy);
% Sizes of primary cells
J= length(dx); K = length(dy);
xu = [0,cumsum(dx)];
% x-coordinates of u-nodes
xv = 0.5*(xu(1:end-1)+xu(2:end));
% x-coordinates of v-nodes
yv = [0,cumsum(dy)];
% y-coordinates of v-nodes
yu = 0.5*(yv(1:end-1)+yv(2:end));
% y-coordinates of u-nodes
[XU,YU] = meshgrid(xu,yu);
```

```
[XV,YV] = meshgrid(xv,yv);

[XP,YP] = meshgrid(xv,yu);% Coordinates of pressure nodes


DXU = XU; % Size of finite volumes for u; preallocation.

DXU(:,1) = DX(:,1)/2;

DXU(:,2:J) = (DX(:,1:J-1) + DX(:,2:J))/2;

DXU(:,J+1) = DX(:,J)/2;

DYU = [DY,dy'];

DYV = XV;

% Size of finite volumes for v; preallocation.

DYV(1,:) = DY(1,:)/2;

DYV(2:K,:) = (DY(1:K-1,:) + DY(2:K,:))/2;

DYV(K+1,:) = DY(K,:)/2; DXV = [DX;dx];

volu = DXU.*DYU; vol = 1./volu';  n = (J+1)*K;

invvolu = spdiags(vol(:), 0, n, n); % (u-volume)^(-1)

volv = DXV.*DYV; vol = 1./volv';  n = J*(K+1);

invvolv = spdiags(vol(:), 0, n, n);

% (v-volume)^(-1)

jseg = [0,cumsum(nx)];
```

```
% j-indices of horizontal segment boundaries

kseg = [0,cumsum(ny)];

% k-indices of vertical segment boundaries

hx = min(dx);% Required for stability estimate for

hy = min(dy);

clear vol

% GRAD_AND_DIV

% Generates pressure gradient matrices Pu, Pv and divergence

% matrix D Vectorized matrix generation by evaluation of stencil

% coefficients

% Output: Pu, Pv, Du, Dv

J= length(dx); K = length(dy);


%................Pressure gradient matrix Pu ...............

%           |    0    |

% Stencil: [Pu] = |p1 p2  0 |

%           |    0    |

% Indexing convention in staggered grid:

%  +--k+1--+
%  |       |
%  j  jk  j+1
%  |       |
%  +---k---+
tic
p2 = 1./DXU(1,:)'; p1 = zeros(size(p2)); p1(1:J) = -p2(2:J+1);
Puu = spdiags([p1  p2],[-1;0], J+1, J); Pu = kron(speye(K),Puu);
%.....................Boundary corrections......................
for seg = 1:length(ybc(1,:))
  if (ybc(1,seg) == 1)|(ybc(1,seg) == 2)
% No-slip or inflow at left boundary
    k = 1+kseg(seg):kseg(seg+1); Pu(1+(k-1)*(J+1),:) = 0;
  end
  if (ybc(2,seg) == 1)|(ybc(2,seg) == 2)
% No-slip or inflow at right boundary
    k = 1+kseg(seg):kseg(seg+1);  Pu(k*(J+1),:) = 0;
  end
end
tijd = toc; disp(['Breakdown of grad_and_div time'])
```

```
disp(['  Pu time = ',num2str(tijd)])

%................Pressure gradient matrix Pv ..............

%           |     0   |
% Stencil: [Pv] = |0   p2  0|
%           |    p1   |
tic
pp1 = zeros(size(XV)); pp2 = pp1;
% Diagonals of Pv
pp1(2:K+1,:) = -1./DYV(2:K+1,:);  pp2 = -pp1;
pp2(1,:) = 1./DYV(1,:);  pp2(K+1,:) = 0;

%.....................Boundary corrections.................
for seg = 1:length(xbc(1,:))
  if (xbc(1,seg) == 1)|(xbc(1,seg) == 2)
% No-slip or inflow at lower boundary
     j = 1+jseg(seg):jseg(seg+1);
     pp1(1,j) = 0; pp2(1,j) = 0;
  end

  if (xbc(2,seg) == 1)|(xbc(2,seg) == 2)

% No-slip or inflow at upper boundary

     j = 1+jseg(seg):jseg(seg+1);

     pp1(K+1,j) = 0; pp2(K+1,j) = 0;

  end

end

n = J*(K+1);   p1 = reshape(pp1',n,1);

p2 = reshape(pp2',n,1);

p1 = [p1(J+1:n); zeros(J,1)];

% Shift to accomodate spdiags

Pv = spdiags([p1  p2],[-J;0], n,n-J);

tijd = toc; disp(['  Pv time = ',num2str(tijd)])


%................u divergence matrix Du ...................


%   |   0     |

% Stencil: [Du] = |0  p1  p2|

%   |   0     |
tic

Duu = spdiags([-ones(J,1) ones(J,1)], [0;1], J,J+1);

Du = kron(spdiags(dy',0,K,K),Duu);

tijd = toc; disp(['  Du time = ',num2str(tijd)])


%................v divergence matrix Dv ...................


%   |   p2   |

% Stencil: [Dv] = |0  p1  0|

%   |   0    |


tic

Dvv = spdiags([-ones(K,1) ones(K,1)], [0;1], K,K+1);

Dv = kron(Dvv,spdiags(dx',0,J,J));

tijd = toc; disp(['  Dv time = ',num2str(tijd)])


clear  pp1 pp2 p1 p2 Puu Duu Dvv

% Save storage
% INERTIA_MATRIX

% Discretization matrices for inertia term

% Indexing convention in staggered grid:

%  +--k+1--+

%  |       |

%  j  jk  j+1

%  |       |

%  +---k---+


if n< 2, tic, end


au1 = zeros(K,J+1); au2=au1; au3=au1;

au4=au1; au5=au1;

% Diagonals of Cu

av1 = zeros(K+1,J); av2=av1; av3=av1; av4=av1; av5=av1;

% Diagonals of Cv

if central == 1

% Central scheme for inertia term

  up = reshape(u0,size(XU')); up = up';
```

```
% Two-index ordering of u0

  vp = reshape(v0,size(XV')); vp = vp';

  upv = [up;up(K,:)];

  upv(2:K,:) = (upv(1:K-1,:).*DYU(1:K-1,:) ...

  + upv(2:K,:).*DYU(2:K,:))...

  ./(DYU(1:K-1,:) + DYU(2:K,:));

  vpv = [vp,vp(:,J)];

  % vpv: old v in cell vertices

  vpv(:,2:J) = (vpv(:,1:J-1).*DXV(:,1:J-1)...

  + vpv(:,2:J).*DXV(:,2:J))...

  ./(DXV(:,1:J-1) + DXV(:,2:J));

  au1(2:K,:)   = - vpv(2:K,:)./(2*DYU(2:K,:));

  au2(:,2:J+1) = - up(:,1:J)./(2*DXU(:,2:J+1));

  au4(:,1:J)   =   up(:,2:J+1)./(2*DXU(:,1:J));

  au5(1:K-1,:) =   vpv(2:K,:)./(2*DYU(1:K-1,:));

  au3(:,1)     = - up(:,1)./(2*DXU(:,1));

  au3(:,J+1)   =   up(:,J+1)./(2*DXU(:,J+1));

  au3(1:K-1,:) =   au3(1:K-1,:) + au5(1:K-1,:);

  au3(2:K,:)   =   au3(2:K,:)   + au1(2:K,:);
```

```
% Further contributions to au3(1,:) and au3(K,:)
% to be added below depending on boundary conditions

  av1(2:K+1,:) = - vp(1:K,:)./(2*DYV(2:K+1,:));
  av2(:,2:J)   = - upv(:,2:J)./(2*DXV(:,2:J));
  av4(:,1:J-1) =   upv(:,2:J)./(2*DXV(:,1:J-1));
  av5(1:K,:)   =   vp(2:K+1,:)./(2*DYV(1:K,:));
  av3(1,:)     = - vp(1,:)./(2*DYV(1,:));
  av3(K+1,:)   =   vp(K+1,:)./(2*DYV(K+1,:));
  av3(:,2:J)   =   av3(:,2:J)   + av2(:,2:J);
  av3(:,1:J-1) =   av3(:,1:J-1) + av4(:,1:J-1);
% Further contributions to av3(:,1) and av3(:,J)
% to be added below depending on boundary conditions.

else % Upwind scheme for inertia term
  up = (u0 + abs(u0))/2; um = (u0 - abs(u0))/2;
  up = reshape(up,size(XU'));
  up = up'; um = reshape(um,size(XU')); um = um';
  vp = (v0 + abs(v0))/2; vm = (v0 - abs(v0))/2;
  vp = reshape(vp,size(XV'));
  vp = vp'; vm = reshape(vm,size(XV')); vm = vm';
  vpv = [vp,vp(:,J)];
% Approximation of v+|v|, v-|v|
  vmv = [vm,vm(:,J)]; %   in cell vertices
  vpv(:,2:J) = (vpv(:,1:J-1).*DXV(:,1:J-1)...
 + vpv(:,2:J).*DXV(:,2:J))...
  ./(DXV(:,1:J-1) + DXV(:,2:J));
  vmv(:,2:J) = (vmv(:,1:J-1).*DXV(:,1:J-1) ...
+ vmv(:,2:J).*DXV(:,2:J))...
  ./(DXV(:,1:J-1) + DXV(:,2:J));
  upv = [up;up(K,:)]; % Approximation of u+|u|, u-|u|
  umv = [um;um(K,:)]; %   in cell vertices
  upv(2:K,:) = (upv(1:K-1,:).*DYU(1:K-1,:) ...
  + upv(2:K,:).*DYU(2:K,:))...
  ./(DYU(1:K-1,:) + DYU(2:K,:));
  umv(2:K,:) = (umv(1:K-1,:).*DYU(1:K-1,:) ...
  + umv(2:K,:).*DYU(2:K,:))...
  ./(DYU(1:K-1,:) + DYU(2:K,:));
```

```
au1(2:K,:)   = - vpv(2:K,:)./DYU(2:K,:);

au2(:,2:J+1) = - up(:,1:J)./DXU(:,2:J+1);

au4(:,1:J)   =   um(:,2:J+1)./DXU(:,1:J);

au5(1:K-1,:) =   vmv(2:K,:)./DYU(1:K-1,:);

au3(:,1)     = - um(:,1)./DXU(:,1);

au3(:,J+1)   =   up(:,J+1)./DXU(:,J+1);

au3(:,2:J)  =   (up(:,2:J) - um(:,2:J))./DXU(:,2:J);

au3(1,:)= au3(1,:)     + (vpv(2,:) -  vmv(1,:))./DYU(1,:);

au3(K,:)= au3(K,:)   + (vpv(K+1,:) - vmv(K,:))./DYU(K,:);

au3(2:K-1,:)= au3(2:K-1,:) + (vpv(3:K,:) ...

- vmv(2:K-1,:))./DYU(2:K-1,:);

av1(2:K+1,:) = - vp(1:K,:)./DYV(2:K+1,:);

av2(:,2:J)   = - upv(:,2:J)./DXV(:,2:J);

av4(:,1:J-1) =   umv(:,2:J)./DXV(:,1:J-1);

av5(1:K,:)   =   vm(2:K+1,:)./DYV(1:K,:);

av3(:,1)     =   (upv(:,2) - umv(:,1))./DXV(:,1);

av3(:,J)     =   (upv(:,J+1) - umv(:,J))./DXV(:,J);

av3(:,2:J-1) =   (upv(:,3:J) - umv(:,2:J-1))./DXV(:,2:J-1);
```

```
  av3(1,:)     =   av3(1,:) - vm(1,:)./DYV(1,:);
  av3(K+1,:)   =   av3(K+1,:) + vp(K+1,:)./DYV(K+1,:);
  av3(2:K,:)   =   av3(2:K,:) + (vp(2:K,:) ...
  - vm(2:K,:))./DYV(2:K,:);
end

% ................Boundary corrections.....................

rvm = zeros(size(XV)); % Contribution to right-hand side
tm = t + (omega-1)*dt;
for seg = 1:length(ybc(1,:))
  if (ybc(1,seg) == 1)|(ybc(1,seg) == 2)
% No-slip or inflow
% at left boundary
    k = 1+kseg(seg);
     rvm(k,1)
     = rvm(k,1) + 0.5*up(k,1).*vbd(tm,1,k,'left')*DY(k,1);
    k = 2+kseg(seg):kseg(seg+1);
     rvm(k,1) =
      rvm(k,1) + upv(k,1).*vbd(tm,1,k,'left').*DYV(k,1);

    k = kseg(seg+1)+1;

     rvm(k,1) =

     rvm(k,1) + 0.5*up(k-1,1).*vbd(tm,1,k,'left')*DY(k-1,1);


  elseif ybc(1,seg) == 3

% Outflow at left boundary

    if central == 1

      k = 1+kseg(seg);

      av3(k,1) =

      av3(k,1) - 0.5*up(k,1)*DY(k,1)/(DYV(k,1)*DXV(k,1));

      k = 2+kseg(seg):kseg(seg+1);

      av3(k,1) = av3(k,1) - upv(k,1)./DXV(k,1);

      k = kseg(seg+1)+1;

      av3(k,1) =

      av3(k,1) - 0.5*up(k-1,1)*DY(k-1,1)/(DYV(k,1)*DXV(k,1));

    else

      % Do nothing

    end
```

```
 else

   error('Wrong ybc')

 end


 if (ybc(2,seg) == 1)|(ybc(2,seg) == 2)

 % No-slip or inflow at right boundary

   k = 1+kseg(seg);

    rvm(k,J) =

    rvm(k,J) - 0.5*up(k,J+1).*vbd(tm,J,k,'right')*DY(k,J);

   k = 2+kseg(seg):kseg(seg+1);

    rvm(k,J) =

    rvm(k,J) - upv(k,J+1).*vbd(tm,J,k,'right').*DYV(k,J);

   k = kseg(seg+1)+1;

 rvm(k,J) =

 rvm(k,J)-0.5*up(k-1,J+1).*vbd(tm,J,k,'right')*DY(k-1,J);

 elseif ybc(2,seg) == 3

% Outflow at right boundary

   if central == 1

     k = 1+kseg(seg);
```

```
  av3(k,J) =
  av3(k,J) + 0.5*up(k,J+1)*DY(k,J)/(DYV(k,J)*DXV(k,J));
      k = 2+kseg(seg):kseg(seg+1);
      av3(k,J) = av3(k,J) + upv(k,J+1)./DXV(k,J);
      k = kseg(seg+1)+1;
 av3(k,J) =
 av3(k,J) + 0.5*up(k-1,J+1)*DY(k-1,J)/(DYV(k,J)*DXV(k,J));
    else
      % Do nothing
    end
  else
    error('Wrong ybc')
  end
end


rum = zeros(size(XU));
 % Contribution to right-hand side
for seg = 1:length(xbc(1,:))
  if (xbc(1,seg) == 1)|(xbc(1,seg) == 2)
```

```
% No-slip or inflow at lower boundary
    j = 1+jseg(seg):jseg(seg+1);  av3(1,j) = 0;
    av1(1,j) = 0; av2(1,j) = 0; av4(1,j) = 0; av5(1,j) = 0;
    j = 1+jseg(seg);
     rum(1,j) =
     rum(1,j) + 0.5*vp(1,j).*ubd(tm,j,1,'lower')*DX(1,j);
    j = 2+jseg(seg):jseg(seg+1);
    rum(1,j) = rum(1,j) + vpv(1,j).*ubd(tm,j,1,'lower').*DXU(1,j);
    j = jseg(seg+1)+1;
  rum(1,j) =
  rum(1,j) + 0.5*vp(1,j-1).*ubd(tm,j,1,'lower')*DX(1,j-1);
  elseif xbc(1,seg) == 3
  % Outflow at lower boundary
    if central == 1
    j = 1+jseg(seg);
   au3(1,j) =
  au3(1,j) - 0.5*vp(1,j)*DX(1,j)/(DXU(1,j)*DYU(1,j));
      j = 2+jseg(seg):jseg(seg+1);
      au3(1,j) =
  au3(1,j) - vpv(1,j)./DYU(1,j);
      j = jseg(seg+1)+1;
   au3(1,j) =
  au3(1,j) - 0.5*vp(1,j-1)*DX(1,j-1)/(DXU(1,j)*DYU(1,j));
    else
      % Do nothing
    end
  else
    error('Wrong xbc')
  end
  if (xbc(2,seg) == 1)|(xbc(2,seg) == 2)
  % No-slip or inflow at upper boundary
    j = 1+jseg(seg):jseg(seg+1);
    av3(K+1,j) = 0;
     av1(K+1,j) = 0; av2(K+1,j) = 0;
     av4(K+1,j) = 0; av5(K+1,j) = 0;
    j = 1+jseg(seg);
     rum(K,j) =
     rum(K,j) - 0.5*vp(K+1,j).*ubd(tm,j,K,'upper')*DX(K,j);
    j = 2+jseg(seg):jseg(seg+1);
     rum(K,j) =
     rum(K,j) - vpv(K+1,j).*ubd(tm,j,K,'upper').*DXU(K,j);
    j = jseg(seg+1)+1;
  rum(K,j) =
  rum(K,j) - 0.5*vp(K+1,j-1).*ubd(tm,j,K,'upper')*DX(K,j-1);
  elseif xbc(2,seg) == 3
  % Outflow at upper boundary
    if central == 1
      j = 1+jseg(seg);
      au3(K,j) =
      au3(K,j) + 0.5*vp(K+1,j)*DX(K,j)/(DXU(K,j)*DYU(K,j));
      j = 2+jseg(seg):jseg(seg+1);
      au3(K,j) =  au3(K,j) + vpv(K+1,j)./DYU(K,j);
      j = jseg(seg+1)+1;
     au3(K,j) =
     au3(K,j) + 0.5*vp(K+1,j-1)*DX(K,j-1)/(DXU(K,j)*DYU(K,j));
    else
      % Do nothing
    end
  else
    error('Wrong xbc')
  end
end
for seg = 1:length(ybc(1,:))
  if (ybc(1,seg) == 1)|(ybc(1,seg) == 2)
% No-slip or inflow at left boundary
    k = 1+kseg(seg):kseg(seg+1);  rum(k,1) = 0;  au3(k,1) = 0;
    au1(k,1) = 0; au2(k,1) = 0; au4(k,1) = 0; au5(k,1) = 0;
  end
  if (ybc(2,seg) == 1)|(ybc(2,seg) == 2)
% No-slip or inflow at right boundary
     k = 1+kseg(seg):kseg(seg+1);
     rum(k,J+1) = 0;   au3(k,J+1) = 0;
     au1(k,J+1) = 0; au2(k,J+1) = 0;
     au4(k,J+1) = 0; au5(k,J+1) = 0;
  end
end
```

```
for seg = 1:length(xbc(1,:))

  if (xbc(1,seg) == 1)|(xbc(1,seg) == 2)

% No-slip or inflow at lower boundary

    j = 1+jseg(seg):jseg(seg+1);  rvm(1,j) = 0;

  end

  if (xbc(2,seg) == 1)|(xbc(2,seg) == 2)

% No-slip or inflow at upper boundary

    j = 1+jseg(seg):jseg(seg+1); rvm(K,j) = 0;

  end

end

rum = rum./volu; rvm = rvm./volv;

% Contributions to right-hand side

 au1 = au1'; au1 = au1(:); au2 = au2'; au2 = au2(:);

au3 = au3'; au3 = au3(:); au4 = au4'; au4 = au4(:);

au5 = au5'; au5 = au5(:);

%[au1  au2  au3  au4  au5];% Display stencil on screen

nn = (J+1)*K;

au1 = [au1(J+2:nn); zeros(J+1,1)];

% Shifts to accomodate spdiags
```

```
au2 = [au2(2:nn); 0];au4 = [0; au4(1:nn-1)];

au5 = [zeros(J+1,1); au5(1:nn-J-1)];

d = [-J-1;  -1;  0;  1;  J+1];

Cu = spdiags([au1  au2  au3  au4  au5], d, nn, nn);


av1 = av1'; av1 = av1(:); av2 = av2'; av2 = av2(:);

av3 = av3'; av3 = av3(:); av4 = av4'; av4 = av4(:);

av5 = av5'; av5 = av5(:);

% [av1  av2  av3  av4  av5]

% Display stencil on screen

nn = J*(K+1);

av1 = [av1(J+1:nn); zeros(J,1)];

% Shifts to accomodate spdiags

av2 = [av2(2:nn); 0]; av4 = [0; av4(1:nn-1)];

av5 = [zeros(J,1); av5(1:nn-J)];

d = [-J;  -1;  0;  1;  J];

Cv = spdiags([av1  av2  av3  av4  av5], d, nn, nn);

% clear au1 au2 au3 au4 au5 av1 av2 av3 av4 av5 d nn

% VISCOUS_MATRIX
% Vectorized matrix generation

% by evaluation of stencil coefficients

% Output: Bu, Bv

%       |   b5    |

% [B] = |b2 b3  b4|

%       |   b1    |

% Indexing convention in staggered grid:

%  +--k+1--+

%  |       |

%  j  jk  j+1

%  |       |

%  +---k---+

tic

r = 1/Re;

% Mask for old u, v in Dirichlet boundary

% points in time-stepping:

masku = ones(size(XU)); maskv = ones(size(XV));


%................Diagonals of viscous matrix Bu for u.....
```

```
au1 = zeros(K,J+1); au2=au1; au3=au1; au4=au1; au5=au1;

% Diagonals of Bu

au1(2:K,:)   = -2*r.*DXU(2:K,:)./(DYU(1:K-1,:) + DYU(2:K,:));

au2(:,2:J+1) =  -r*DY./DX;

au4(:,1:J) = au2(:,2:J+1);   au5(1:K-1,:) = au1(2:K,:);

au3          = - au1 - au2 - au4 - au5;


%................Diagonals of viscous matrix Bv for v......

av1 = zeros(K+1,J); av2=av1; av3=av1; av4=av1; av5=av1;

% Diagonals of Bv

av1(2:K+1,:)   = -r*DX./DY;

av2(:,2:J)   = -2*r*DYV(:,2:J)./(DXV(:,1:J-1)+DXV(:,2:J));

av4(:,1:J-1) = av2(:,2:J); av5(1:K,:) = av1(2:K+1,:);

av3          = - av1 - av2 - av4 - av5;


% ................Boundary corrections......................

jseg = [0,cumsum(nx)];

% j-indices of horizontal segment boundaries

for seg = 1:length(xbc(1,:))
```

```
  if (xbc(1,seg) == 1)|(xbc(1,seg) == 2)
 % No-slip or inflow at lower boundary
     j = 1+jseg(seg);
      au3(1,j) = au3(1,j) + r*DX(1,j)/DY(1,j);
     j = 2+jseg(seg):jseg(seg+1);
     au3(1,j) = au3(1,j) + 2*r*DXU(1,j)./DYU(1,j);
     j = 1+jseg(seg+1);
     au3(1,j) = au3(1,j) + r*DX(1,j-1)/DY(1,j-1);

 elseif xbc(1,seg) == 3     % Outflow at lower boundary
    % Do nothing
  else
    error('Wrong xbc')
  end
  if (xbc(2,seg) == 1)|(xbc(2,seg) == 2)
% No-slip or inflow at upper boundary
     j = 1+jseg(seg);
     au3(K,j) = au3(K,j) + r*DX(K,j)/DY(K,j);
     j = 2+jseg(seg):jseg(seg+1);
     au3(K,j) = au3(K,j) + 2*r*DXU(K,j)./DYU(K,j);

     j = 1+jseg(seg+1);

     au3(K,j) = au3(K,j) + r*DX(K,j-1)/DY(K,j-1);


  elseif xbc(2,seg) == 3 % Outflow at upper boundary

    % Do nothing

  else

    error('Wrong xbc')

  end

end

kseg = [0,cumsum(ny)];

% k-indices of vertical segment boundaries

for seg = 1:length(ybc(1,:))

  if (ybc(1,seg) == 1)|(ybc(1,seg) == 2)

% No-slip or inflow at left boundary

    k = 1+kseg(seg):kseg(seg+1);

    au3(k,1) = 0;  masku(k,1) = 0;

     au1(k,1) = 0; au2(k,1) = 0;

     au4(k,1) = 0; au5(k,1) = 0;
```

```
      k = 1+kseg(seg);
      av3(k,1) = av3(k,1) + r*DY(k,1)/DX(k,1);
      k = 2+kseg(seg):kseg(seg+1);
      av3(k,1) = av3(k,1) + 2*r*DYV(k,1)./DXV(k,1);
      k = 1+kseg(seg+1);
      av3(k,1) = av3(k,1) + r*DY(k-1,1)/DX(k-1,1);

   elseif ybc(1,seg) == 3
 % Outflow at left boundary
     % Do nothing
   else
     error('Wrong ybc')
   end

   if (ybc(2,seg) == 1)|(ybc(2,seg) == 2)
  % No-slip or inflow at right boundary
     k = 1+kseg(seg):kseg(seg+1);
     au3(k,J+1) = 0;  masku(k,J+1) = 0;
      au1(k,J+1) = 0; au2(k,J+1) = 0; au4(k,J+1) = 0;
      au5(k,J+1) = 0;

      k = 1+kseg(seg);

      av3(k,J) = av3(k,J) + r*DY(k,J)/DX(k,J);

      k = 2+kseg(seg):kseg(seg+1);

      av3(k,J) = av3(k,J) + 2*r*DYV(k,J)./DXV(k,J);

      k = 1+kseg(seg+1);

      av3(k,J) = av3(k,J) + r*DY(k-1,J)/DX(k-1,J);


   elseif ybc(2,seg) == 3

 % Outflow at left boundary

     % Do nothing

   else

     error('Wrong ybc')

   end

 end

 for seg = 1:length(xbc(1,:))

   if (xbc(1,seg) == 1)|(xbc(1,seg) == 2)

  % No-slip or inflow at lower boundary

     j = 1+jseg(seg):jseg(seg+1);
     av3(1,j) = 0; maskv(1,j) = 0;
     av1(1,j) = 0; av2(1,j) = 0; av4(1,j) = 0; av5(1,j) = 0;

   elseif xbc(1,seg) == 3
  % Outflow at lower boundary
     % Do nothing
   else
     error('Wrong xbc')
   end
   if (xbc(2,seg) == 1)|(xbc(2,seg) == 2)
  % No-slip or inflow at upper boundary
     j = 1+jseg(seg):jseg(seg+1);
     av3(K+1,j) = 0; maskv(K+1,j) = 0;
      av1(K+1,j) = 0; av2(K+1,j) = 0;
      av4(K+1,j) = 0; av5(K+1,j) = 0;

   elseif xbc(2,seg) == 3
  % Outflow at upper boundary
     % Do nothing
   else

     error('Wrong xbc')

   end

 end


 au1 = au1'; au1 = au1(:);

 au2 = au2'; au2 = au2(:);

 au3 = au3'; au3 = au3(:);

 au4 = au4'; au4 = au4(:);

 au5 = au5'; au5 = au5(:);

 % [au1  au2  au3  au4  au5];

 % Display stencil on screen

 n = (J+1)*K;

 au1 = [au1(J+2:n); zeros(J+1,1)];

 % Shifts to accomodate spdiags

 au2 = [au2(2:n); 0]; au4 = [0; au4(1:n-1)];

  au5 = [zeros(J+1,1); au5(1:n-J-1)];

 d = [-J-1;  -1;  0;  1;  J+1];

 Bu = invvolu*spdiags([au1  au2  au3  au4  au5], d, n, n);
```

```
av1 = av1'; av1 = av1(:); av2 = av2'; av2 = av2(:);

av3 = av3'; av3 = av3(:); av4 = av4'; av4 = av4(:);

av5 = av5'; av5 = av5(:);

%[av1  av2  av3  av4  av5]

% Display stencil on screen

n = J*(K+1);

av1 = [av1(J+1:n); zeros(J,1)];

% Shifts to accomodate spdiags

av2 = [av2(2:n); 0]; av4 = [0; av4(1:n-1)];

av5 = [zeros(J,1); av5(1:n-J)];

d = [-J;  -1;  0;  1;  J];

Bv = invvolv*spdiags([av1  av2  av3  av4  av5], d, n, n);

masku = masku'; masku = masku(:);

Masku = spdiags(masku,0,length(masku),length(masku));

maskv = maskv'; maskv = maskv(:);

Maskv = spdiags(maskv,0,length(maskv),length(maskv));


clear au1 au2 au3 au4 au5 av1 av2 av3 av4 av5 d masku maskv
```

```
tijd = toc; disp(['viscous_matrix time = ',num2str(tijd)])

% INITIAL_CONDITION

% Generates initial conditions for u,v and p

% Functions called: ubd, vbd

tic


if geval == 1

% Horizontal Poiseuille flow to the right

    u0 = 5*ones(Size(XU));

    v0 = zeros(size(XV));

    p0 = (xu(end) - XP)*12/(Re*(yv(end)-yv(1))^2);


elseif geval == 4 % Driven cavity

     u0 = zeros(size(XU)); v0 = zeros(size(XV));

     p0 = zeros(size(XP));

else

  error('Wrong value for geval in INITIAL CONDITION')

end

u0 = u0'; u0 = u0(:);
```

```
 v0 = v0'; v0 = v0(:); p0 = p0'; p0 = p0(:);

 disp([u0]);

function ye = ubd(t, j, k, side)

% UBD Prescribes u at inflow boundaries

% Possible values for side:

%'lower', 'upper',  'left', 'right'


global geval u0 J K yu yv yseglen alpha


ye = 0;

if geval == 1

% Horizontal Poiseuille flow to the right

  if strcmp(side, 'left') == 1

     ye = u0(k*(J+1));

     % Inflow profile = outflow profile

  else

    ye = 0;

  end

elseif geval == 4 % Driven cavity
```

```
  if strcmp(side, 'upper') == 1
    ye = 1;
  else
    ye = 0;
  end
else
  error('Wrong value in input for parameter geval')
end
function ye = vbd(t, j, k, side)
% VBD Prescribes v at inflow boundaries
% Possible values for side: 'lower',
%'upper', 'left', 'right'

global geval v0 J K alpha

ye = 0;
if geval == 1 % Horizontal flow
  ye = 0;
elseif geval == 4 % Driven cavity
```

```
  ye = 0;

else

  error('Wrong value in input for parameter geval')

end

% STREAMLINEPLOT

% Screen plot of streamlines


figure(3), clf

title('Streamlines','FontSize',16)

hold on

uq = reshape(u1,size(XU')); uq = uq';

vq = reshape(v1,size(XV')); vq = vq';

sf = zeros(size(X)); % Streamfunction

sf(1,:) = [0, - cumsum(DXV(1,:).*vq(1,:))];

for k = 2:K+1

  sf(k,:) = sf(k-1,:) + uq(k-1,:).*DYU(k-1,:);

end

cvals=[linspace(min(min(sf)),max(max(sf)),30),

0,0.995*max(max(sf))];
```

```
contour(xu,yv,sf,cvals,'k')

uq = (uq(:,1:J) + uq(:,2:J+1))/2;

vq = (vq(1:K,:) + vq(2:K+1,:))/2;

quiver(xv,yu,uq,vq,0.9,'k')

hold off
```

Bibliography

[1] Ascher, U.M. and Petzold, L.R., 1998, *Computer Methods for Ordinary Differential Equations and Differential Algebraic Equations*, Society for Industrial and Applied Mathematics, Philadelphia.

[2] Chang, Y.C., Hou, T.Y., Merrian, B., and Osher, S., 1996, A Level Set Formulation of Eulerican Interface Capturing Methods for Incompressible Fluid Flows, *Journal of Computational Physics*, 124, 449-464.

[3] Chen, S., Johnson, D.B., and Raad, P. E., 1994, Velocity Boundary Conditions for The Simulation of Free Surface Fluid Flow, *Journal of Computational Physics*, 116, 262-276.

[4] Chen, S., Johnson, D.B., Raad, P.E. and Fadda, D., 1997, The Surface Marker and Micro Cell Method, *International Journal for Numerical Methods in Fluids*, 25, 749-778.

[5] De Halleux, J., 2003, *Boundary Control of Quasi-Linear Hyperbolic Initial Boundary-Value Problems*, Ph.D. Thesis, Universite Catholique de Louvain, Belgium.

[6] Hanselman, D. and Littlefield, B., 2001, *Mastering MATLAB 6.0.* Prentice Hall, New Jersey.

[7] Graham, E.W. and Rodriguez, A.M., 1952, The Characteristics of Fuel Motion Which Affect Airplane Dynamics, *Journal of Applied Mechanics*, 19, 381-388.

[8] Griebel, M., Dornseifer, T. and Neuhoeffer, T., 1998, *Numerical simulation in fluid dynamics:a practical introduction*, Society for Industrial and Applied Mathematics, Philadelphia.

[9] Harlow, F.H. and Welch, J.E., 1965, Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface, *Physics Fluid*, 8, 2182-2189.

[10] Housner, G., 1957, Dynamic Pressure on Accelerated Fluid Containers, *Bulletin of the Seismological Society of America*, 47, 15-35.

[11] Hwang, J.H., Kim., I.S., Seol and Y.S., Lee, S.C., 1992, Numerical Simulation of Liquid Sloshing in 3-Dimensional Tanks, *Journal of Computers and Structures*, 44, 339-342.

[12] Mikelis, N.E. and Journee, J.M., 1984, Experimental and Numerical Simulations of Sloshing Behaviour in Liquid Tanks and its Effect on Ship Motions, *National Conference on Numerical Methods for Transient and Coupled Problems*, Venice, Italy, 1-11.

[13] Mitchell, I., 2004, *A Toolbox of Level Set Methods*, UBC CS TR-2004-09, Department of Computer Science, University of British Columbia, Canada. Version 1.1 of the Toolbox was released on March, 2005. See http://www.cs.ubc.ca/ mitchell/ToolboxLS/

[14] Nakayama, T. and Washizu, K., 1981, The Boundary Element Method Applied to The Analysis of Two-Dimensional Nonlinear Sloshing Problems, *International Journal For Numerical Methods in Engineering*, 17, 1631-1646.

[15] Nielsen, K. B., 2003, *Numerical Prediction of Green Water Loads on Ships*, Ph.D. Thesis, Department of Mechanical Engineering, Technical University of Denmark, Denmark.

[16] Kim, M.S., 2002, Numerical Analysis of Sloshing Problem, *15th International Workshop on Water Waves and Floating Bodies*, Dan Caesarea, Israel, 1-4.

[17] Kim, M.S. and Lee, W., 2003, A New VOF-based Numerical Scheme for The Simulation of Fluid Flow with Free Surface. Part II: New Free Surface-Tracking Algorithm and its Verification, *Journal of Numerical Methods in Fluids*, 42, 765-790.

[18] Morton, K. William and Mayers, David F., 1993, *Numerical Solution of Partial Differential Equations*, Cambridge University Press, UK.

[19] Osher, S. and Fedkiw, R., 2002. *Level Set Methods and Dynamic Implicit Surfaces*, Springer, New York.

[20] Osher, S. and Sethian, J.A., 1988, Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton-Jacobi Equations. *Journal of Computational Physics.*, 79, 24-25.

[21] Sethian, J.A., 1999, *Level Set Methods and Fast Marching Methods*, Cambridge University Press, UK.

[22] Shu, C. and Osher, S., 1989, Efficient Implementation of Essentially Non-Oscillatory Shock-Capturing Schemes II, *Journal of Computational Physics*, 83, 32-78.

[23] Sussman, M., Smereka, P. and Osher, S., 1994, A Level Set Approach for Computing Solutions to Incompressible Two-phase Flow, *Journal of Computational Physics*, 114, 146-159.

[24] Sussman, M., 2003, A Second Order Coupled Level Set and Volume-of-fluid Method for Computing Growth and Collapse of Vapor Bubbles, *Journal of Computational Physics*, 187, 110-136.

[25] Szymczak, W.G., Rogers, Joel C.W., and Solomon, J.M., and Berger, A.E., 1992, A Numerical Algorithm for Hydrodynamic Free Boundary Problems, *Journal of Computational Physics*, 106, 319-336.