The C++ Implementation of Refined Level Set Grid(RLSG) Method

by

Salar Safarkhani

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2015 by the
Graduate Supervisory Committee:

Marcus Herrmann, Chair
Jay Oswald
Konrad Rykczewski

ARIZONA STATE UNIVERSITY

August 2015

ABSTRACT

In this thesis, a FORTRAN code is rewritten in C++ with an object oriented approach. There are several reasons for this purpose. The first reason is to establish the basis of a GPU programming. To write programs that utilize GPU hardware, CUDA or OpenCL is used which only support C and C++. FORTRAN has a feature that lets its programs to call C/C++ functions. FORTRAN sends relevant data to C/C++, which in turn sends that data to OpenCL. Although this approach works, it makes the code messy and bulky and in the end more difficult to deal with. Moreover, there is a slight performance decrease from the additional data copy. This is the motivation to have the code entirely written in C++ to make it more uniform, efficient and clean. The second reason is the object oriented feature of the C++. The "abstraction", "inheritance" and "run-time polymorphism" features of C++ provide some form of classes and objects, the ability to build new abstractions, and some form of run-time binding, respectively. In recent years, some of popular codes has been rewritten in C++ which were initially in FORTRAN. One of these softwares is LAMMPS.

In this code the level set equation is solved by RLSG method to track the interface in two phase flow. In gas/fluid flows, the surface tension is important and only exists at the interface. Therefore, the location and some geometric features of interface need to be evaluated which can be achieved by solving the level set equation.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

1.1 Motivation

In this thesis, a FORTRAN code is rewritten in C++ with object oriented approach. The purpose is to establish the basis of GPU programming. To write programs that utilize GPU hardware, CUDA or OpenCL are used which only support C and C++. We can use a fortran feature that lets FORTRAN programs call C/C++ functions. FORTRAN sends relevant data to to C/C++, which in turn sends that data to OpenCL. This works, but makes the code messy and bulky and more difficult to deal with. Moreover, there is in turn a slight performance decrease from the additional data copy. Thus, that is the motivation to have the code entirely rewritten in C++ to make it more uniform, efficient and clean. The second reason is the object oriented feature of the C++. There are lots of definitions of "object-oriented programming". According to Stroustrup (2013), a language or technique is object-oriented if it supports:

1. Abstarction - Some form of classes and objects provides only essential information for user and hide the details.

2. Inheritence - The ability to build new classes out of existing classes.

3. Run-time polymorphism - The ability to have several forms of a class.

In recent years, some popular codes has been rewritten in C++ which were initially in FORTRAN. One of these softwares is LAMMPS.
By the inheritence feature of C++, other method of interface tracking such as Volume

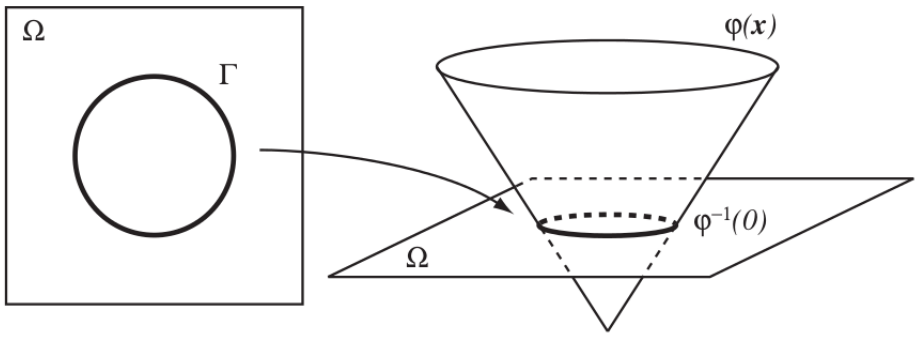


Figure 1.1: Chopp(2012)

of Fluid could be implemented without needing to code from scratch. The data structures that are used in code could be also changed without a need to change a significant part of the code. In this code the level set equation (1.1) is solved to track the interface.

1.2 Level Set Equation and Approaches to Solve It

Basically in this code we solve the initial value level set equation:

$$\frac{\partial \phi}{\partial t} + (\vec{v}.\nabla)\phi = 0 \tag{1.1}$$

The level set method developed by Sethian and Osher is the analyzing of the movement of an interface. Assume that there is a surface which seperates two areas. The level set approach is to reperesent the surface as the zero level set of some higher dimensional function ϕ. It is desirable to define the level set function as a signed distance function. If Γ reperesents the surface in region Ω (figure 1.1) the signed ditance level set function for the domain is:

$$\phi(\mathbf{x}) = sign(-\pi + \int_{\Gamma} arg(\mathbf{y}(s) - \mathbf{x}) \;) \underset{y \in \Gamma}{} min\|\mathbf{x} - \mathbf{y}\| Chopp \text{ (2012)} \tag{1.2}$$

where $y(s)$ is a parameter to define the interface with respect to an origin and x is the position of the other points in the Ω .

The sign of ϕ at time t is defined as equation 1.6

$$\begin{cases} \phi(x,t) > 0 \quad if \quad \mathbf{x} - \mathbf{y} < 0 \\ \phi(x,t) = 0 \quad if \quad \mathbf{x} - \mathbf{y} = 0 \\ \phi(x,t) < 0 \quad if \quad \mathbf{x} - \mathbf{y} > 0 \end{cases} \tag{1.3}$$

To initialize the level set function, the distance from interface is computed at first. To do so, the $y(s)$ is discretized where $\|y(s_{k+1} - y(s_k))\| < \Delta x$ and Δx is the mesh size. Then the grid points near the $y(s_k)$ is marked and the distance is calculated. The smallest value of $\|x_{i,j} - y(s_m)\|$, $(m = k - n, ..., k + n)$ is the distance between the interface and grid point $x_{i,j}$. To calculate the initial level set values of grid points far from interface, the fast marching equation $F\|\nabla\phi\| = 1$ can be solved Sethian (1999a). F is the velocity and can be chosen as 1 for this purpose. However, when the parametric function of initial interface is known such as circle the distance can be computed by the geometric equations. For instance, when the interface is circle with an origin of x_0 and y_0, the level set value for grid point x_i and y_j outside of the interface is $\sqrt{(x_i - x_0)^2 + (y_j - y_0)^2} - R$ and R is the radius of the circle. The evolution of the level set equation in time will be discussed later.

There are a lot of applications for level set method. Our purpose is to track the interface in multiphase flow. In liquid/gas flows, surface tension is important. Computationally, since the surface tension is only active on the interface, it poses singularity. Moreover, as the material propeties change, discontinuity exists at the same location. We will show the the level set values by G rather that ϕ. The interface location is represented by a level set G where $G(x_f, t) = 0$. We define $G(x_f, t) < 0$ for outside of the interface and $G(x_f, t) > 0$ for inside the interface.

As mentioned above, it is preferable that, $G(x_f, t)$ be a signed distance function.

$$|G(x_f, t)| = 1$$

For numerical accuracy, level set values must always be:

$$0 < c < |\nabla G| < C \quad Penget\ al.\ (1999)$$

1.3 Algorithms to Solve The Level Set Equation

In some applications such as image processing, all the level set values are important and we need to have the scalar values of G for entire domain. It is straightforward to implement parallel algorithm in this cases (Sethian (1999b)). However, in multiphase flow application we are only interested in zero level set scalar which reperesents the phase interface. In the next sections, some algorithms will be discussed to solve the level set equations only in a particular distance of the interface rather than the entire domain.

1.3.1 Narrow Band Approach(Chopp (1993))

When one is interested in a specific level set value such as the zero level set (in our case: interface), there are several disadvantages to solve the equation for entire dimain. If the equation is solved for entire grid, each grid point will contain the value of the level set function at that point and all counters will exist. Whereas, only one of them is the zero level set which we are interested(Sethian (1999b)). Thus, if we choose the grids only near the interface it will be more efficient(figure 1.2). There are several reasons for that:

- Speed: The computation cost for a three dimensional grid over the entire domain is $O(N^3)$ where N is the number of grid points in a side. However, if we work only in a neighborhood of the zero level set the computation cost will be $O(kN^2)$.

- Calculating extension variables: The level set approach requires the velocity V. In some cases, the velocity in the entire domain is not known and if the equation

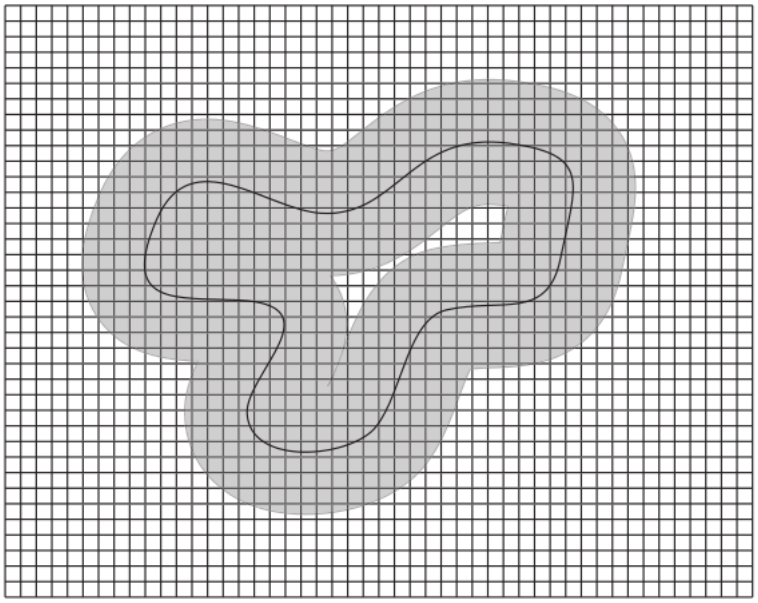


Figure 1.2: Chopp(2012)

is solved for the whole grid, extrapolation will be needed to get the velocity for that points. Extrapolation is difficult and has its errors. However, in narrow band approach the extrapolation will only be for some specific points near the interface.

- Time steps: If the equation is solved for entire domain, it needs a time step that satisfies the CFL condition with respect to maximum velocity of the whole domain. In narrow band approach, since only the grids near the interface is studied, it is more possible(correct) to get a bigger stable time step(Sethian (1999b)).

To implement the idea, all grid points are given two labels depending on their position. The first label represents band name. If for grid point $x_{i,j}$, $|G_{i,j}| \leq k\Delta x$, it is called narrow band point. If for grid point $x_{i,j}$, $k\Delta x < |G_{i,j}| \leq k\Delta X$, it is called barrier band point.

The second label, represents that, whether all the neighbors of a grid point either in

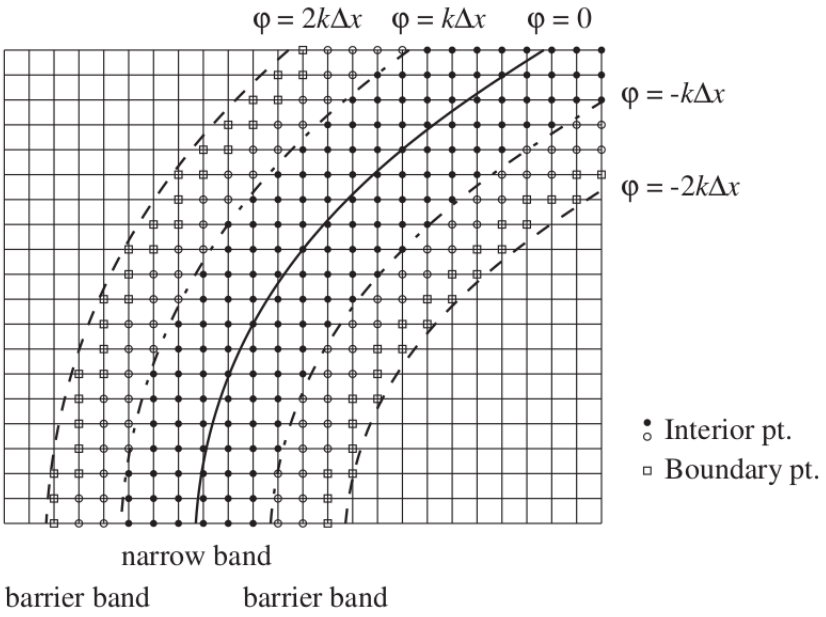


Figure 1.3: Chopp(2012)

the narrow band or barrier band. If so, it is called interiror point, otherwise it is a boundary point. It is necessary to calculate the derivatives.(Chopp (2012))

There are several ways to code this method. For comrarison we assume that each grid point needs to store M variables on a $N \times N$ grid. There are $4kN$ grid points that need to be updated. The normal level set method needs N^2M variables and the computation cost for that is $O(N^2)$. However according to D. Chopp, different data structures can be impelemented for that such as:

1. If there is a two dimensional mesh, the sequential lists include $N^2M + 4kN$ variables. This is greater than the normal level set. However the complexity is $O(N)$.

2. If there is a two dimensional mesh, but the mesh is a mesh of pointers. The storage requirement for this is $N^2 + 4kNM$. The cost is almost $O(N)$.

3. If there is no uderlying mesh, instead, there are only pointers for each spatioal

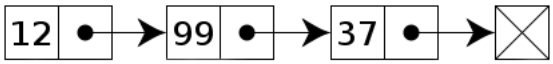


Figure 1.4: The Linked List Forward

grid and they can identify their neighbors. The storage cost is $4kN(M + 8)$ where 8 comes from extra pointers. Since the data is not storaged sequentially, there is some performance loss in this case. The advantages of this method is that, when the interface is advanced and the previous interior and barrier points are deleted and new interior and barrier points are added, it is faster. This is similar to linked list data structure(Chopp (2012)).

1.3.2 Linked List

In linked list data structure objects are arranged in linear order and the order is determined by pointers. Figure 1.4) illustrates a linked list forward. Assume that each item is a "Node" class. This class has two variables, "link" and "data". "link" stores the address of the next node and "info" stroes the relavent information. Therefore, the order of the nodes is determined by the address, called "link". Linked list supports all the properties of search, insert and delete (Cormen *et al.* (2009)). The linked list data structure is fast to remove and add an element. However, it is not appropriate for random access. The following algorithms show how an element can be inserted and deleted in linked list.

Assume that, we want to create a new node called "newNode" with the value of "val" and insert it after node "p". The following algorithm shows how to do this.

```
Node *newNode = new Node;
newNode->info = val;
newNode->link = p->link;
```

```
p->link;
```

The computational cost to insert an element in list is $O(1)$. Now assume that we want to remove an element from list which is located after node "p". The following algorithm illustrates how to do this.

```
p->link = p->link->link;
```

The computational cost to remove an element in worse case is $O(N)$ where N is the number of elements.

1.4 Refined Level Set Grid Method

In this section the RLSG method(Herrmann (2008)) is explained.

1.4.1 The Idea of The RLSG Method

In this code the refined level set grid method(RLSG) is implemented. The level set equations are solved on a seperate equidistant Cartesian grid. Since the G grid is different from flow solver grid, it can be independantly refined. Equation 1.1 and the following two equations are solved all together:

$$\mathbf{n} = \frac{\nabla \mathbf{G}}{|\nabla \mathbf{G}|} \tag{1.4}$$

$$\kappa = \nabla . \mathbf{n} \tag{1.5}$$

To implement this method, firstly the G grid is constructed by super grid cells. The super grid cells that include a part of the interface or are within a specific distance from the interface are called "active"(figure 1.5) These active cells are stored in an array. Each processor stores a copy of i, j, k value of the active super grids if the cell is local for the processor and stores a negative number if the super grid is not local.

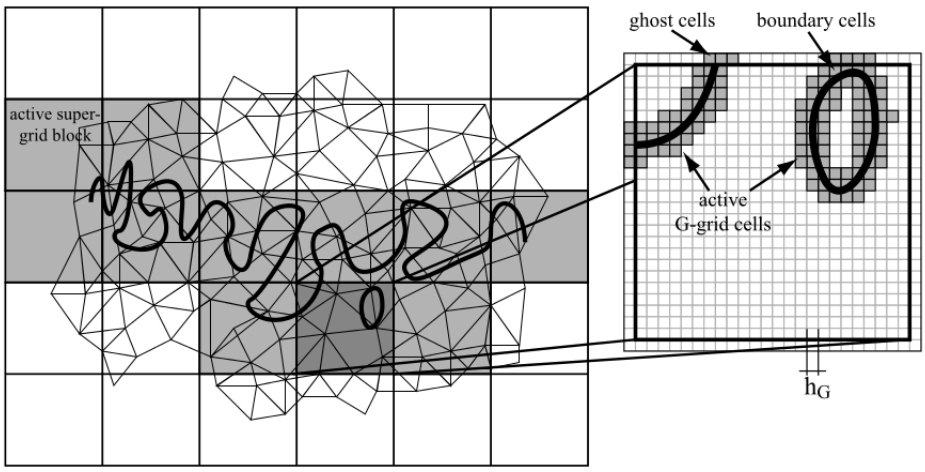


Figure 1.5: Herrmann(2008)

Then each active block is dicretized. The size of new cells are h_G. These cells are called local cells. Agian those cells that are part of the interface or within a specific distance of the interface are called "active" and stored in linked list data structure. In the code each block stores a pointer to the next block such as following:

```
class block_t{
public:
    some variables to store the geometry
    and number of cells in each band;
    block_t *next;
}
```

This approach resembles the narrow band approach. It reduces the computational cost from $O(N^3)$ to $O(N^2)$ where N is the number of cells in each direction. It also decrease the memory usage.

In the the figure 1.6 the algorithm for the "RLSG" method is showed.

for $n = 1, n_G$ **do**
  do level set transport
  if (trigger re-initialization = TRUE) **then**
    do band generation
    do re-initialization
  end if
end do
do load balancing

Figure 1.6: Herrmann(2008)

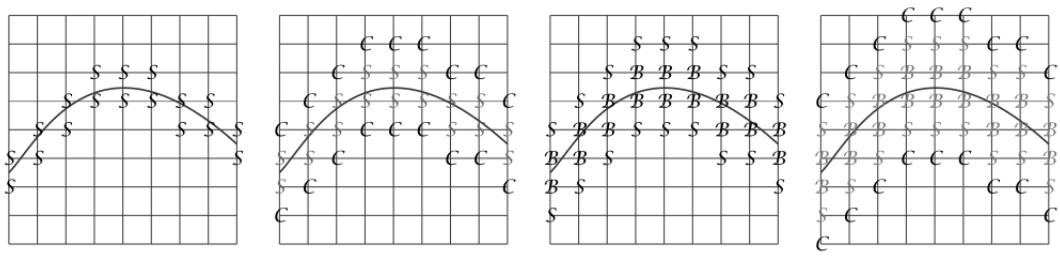


Figure 1.7: Herrmann(2008)

1.4.2 Band Generation

As mentioned before, all the level set equations are solved in a specific band. As the interface moves, the narrow band needs to be updated. It means that, in each time step we need to clean some parts of the previous band and add some new cells to our narrow band. The figures 1.7 and 1.8 illustrate the band generation algorithm.

In this method, all the cells that are part of the narrow band are marked as body and shown by B . The current layer after B is skin and shown by S . The new band layer that is going to be brought to the narrow band is called cloth, shown by C.

As we see in the figure 1.8, all the cells that are intercepted by the interface are marked as S . These are the cells where $G_{i,j,k}G_{i\pm1,j,k} \leq 0$, $G_{i,j,k}G_{i,j\pm1,k} \leq 0$ or $G_{i,j,k}G_{i,j,k\pm1} \leq 0$. All unmarked cells in the neighborhood of the S marked as C. When the method is impelemnted in parallel, since the cloth layer can grow across the supergrid boundary,

for $n = 1, n_i$ **do**
  for all cells marked $\mathcal{S}$ **do**
    mark all unmarked neighbors of $\mathcal{S}$ as $\mathcal{C}$
  end for
  sync ghostnodes with neighboring blocks
  mark all $\mathcal{S}$ cells as $\mathcal{B}$
  mark all $\mathcal{C}$ cells as $\mathcal{S}$
end for

Figure 1.8: Herrmann(2008)

these ghost cells are copied to the neighbor super grid. If that super grid does not exist, it is activated and appended to the active super grid cells. At the end, the skin layers are changed to body layers and cloth layers are changed to skin layers. In this method four different bands are constructed, *T, N, W, X* . *T* band is used for transport, *N* band is the reinitialization band and *W* band is the WENO band. X band is the flow solver band where the volume integration is implemented. In next sections, the transport, re initialization and WENO scheme are discussed.

1.4.3 Level Set Transport

The level set equation is a Hamilton- Jacobi equation. Fifth order WENO(Jiang and Peng (2000)) is used to advance the level set scalar in space. For time integeration TVD-RK3(Shu (1988)) is used. The level set equation is only solved in transport band(*T)*. It is suggested that, V is replaced by

$$V_{cut} = c(G)V$$

with the following condition(Peng *et al.* (1999)):

$$c(G) = \begin{cases} 1 : \alpha \leq -3 \\ \frac{2}{27}\alpha^3 + \frac{1}{3}\alpha^2 : -3 < \alpha \leq 0 \\ 0 : \alpha > 0 \end{cases} \tag{1.6}$$

and

$$\alpha = \frac{|\phi|}{\Delta x} - n_T$$

where n_T is the nomber of cells in T band. Thus, the velocity will be zero at the boundary of the transport and WENO band.

1.4.4 WENO Scheme

For the Hamilton-Jacobi equation(1.7)

$$G_t + H(x, t, G, DG) = 0, \quad G_0(x) = G(x, 0) \tag{1.7}$$

The ENO (Essentially Non-Oscillatory) scheme is (Harten *et al.* (1987)):

$$G^-_{x,i} = \begin{cases} G^{-,0}_{x,i} & |\Delta^-\Delta^+ G_{i-1}| < |\Delta^-\Delta^+ G_i| \quad and \quad |\Delta-\Delta-\Delta+G_{i-1}| < |\Delta^+\Delta-\Delta+G_{i-1}| \\ G^{-,2}_{x,i} & |\Delta^-\Delta^+ G_{i-1}| > |\Delta^-\Delta^+ G_i| \quad and \quad |\Delta-\Delta-\Delta+G_i| > |\Delta^+\Delta-\Delta+G_i| \\ G^{-,1}_{x,i} & otherwise \end{cases} \tag{1.8}$$

where:

$$\begin{cases} G^{-,0}_{x,i} = \frac{1}{3}\frac{\Delta^+ G_{i-3}}{\Delta x} - \frac{7}{6}\frac{\Delta^+ G_{i-2}}{\Delta x} + \frac{11}{6}\frac{\Delta^+ G_{i-1}}{\Delta x} \\ G^{-,1}_{x,i} = -\frac{1}{6}\frac{\Delta^+ G_{i-2}}{\Delta x} + \frac{5}{6}\frac{\Delta^+ G_{i-1}}{\Delta x} + \frac{1}{3}\frac{\Delta^+ G_i}{\Delta x} \\ G^{-,2}_{x,i} = \frac{1}{3}\frac{\Delta^+ G_{i-1}}{\Delta x} + \frac{5}{6}\frac{\Delta^+ G_i}{\Delta x} - \frac{1}{6}\frac{\Delta^+ G_{i+1}}{\Delta x} \end{cases} \tag{1.9}$$

and:

$$\Delta^+\phi_k = \phi_{k+1} - \phi_k, \quad \Delta^-\phi_k = \phi_k - \phi_{k-1} \tag{1.10}$$

The WENO(Weighted Essentially Non-Oscillatory) scheme is a weighted averarage of $G^{-,n}$(n=0,1,2)Jiang and Peng (2000).

The WENO approximation of $G_{x,i}$ on a left-biased stecile is:

$$\begin{aligned} \phi^-_{x,i} = \frac{1}{12}\left(\frac{\Delta^+\phi_{i-2}}{\Delta x} + 7\frac{\Delta^+\phi_{i-1}}{\Delta x} + 7\frac{\Delta^+\phi_i}{\Delta x} - \frac{\Delta^+\phi_{i+1}}{\Delta x}\right) \\ -\Phi^{WENO}\left(\frac{\Delta^-\Delta^+\phi_{i-2}}{\Delta x}, \frac{\Delta^-\Delta^+\phi_{i-1}}{\Delta x}, \frac{\Delta^-\Delta+\phi_i}{\Delta x}, \frac{\Delta^-\Delta^+\phi_{i+1}}{\Delta x}\right) \end{aligned} \tag{1.11}$$

where:

$$\Phi^{WENO}(a,b,c,d) = \frac{1}{3}\omega_0(a-2b+c) + \frac{1}{6}(\omega_2 - \frac{1}{2})(b-2c+d) \tag{1.12}$$

and:

$$\omega_0 = \frac{\alpha_0}{\alpha_0+\alpha_1+\alpha_2}, \quad \omega_2 = \frac{\alpha_2}{\alpha_0+\alpha_1+\alpha_2} \tag{1.13}$$

and the α values are:

$$\alpha_0 = \frac{1}{(\epsilon+IS_0)^2}, \quad \alpha_1 = \frac{6}{(\epsilon+IS_1)^2}, \quad \alpha_2 = \frac{3}{(\epsilon+IS_2)^2} \tag{1.14}$$

where:

$$\begin{aligned} IS_0 &= 13(a-b)^2 + 3(a-3b)^2 \\ IS_1 &= 13(b-c)^2 + 3(b+c)^2 \\ IS_2 &= 13(c-d)^2 + 3(3c-d)^2 \end{aligned} \tag{1.15}$$

ϵ is used to have a nonzero denominator.

1.4.5 TVD-RK3

To solve the $G_t + aG_x = 0$ TVD(Total-Variation-Diminishing) schemes Shu (1988) are used. To solve the equation 1.1, the WENO scheme is combined with 3 step

Runge-Kutta TVD time integration.

The m-step Runge-Kutta TVD time integration for timestep $n+1$ is:

$$\begin{cases} step \quad 0: \quad G_i^{(0)} = G_i^n \\ step \quad 1: \quad G_i^{(1)} = G_i^{(0)} - \alpha_{1,0}\left(a\Delta t \frac{\partial G^{(0)}}{\partial x}\Big|_i^-\right) \\ step \quad 2: \quad G_i^{(2)} = G_i^{(1)} - \alpha_{2,0}\left(a\Delta t \frac{\partial G^{(0)}}{\partial x}\Big|_i^-\right) - \alpha 2,1\left(a\Delta t \frac{\partial G^{(1)}}{\partial x}\Big|_i^-\right) \\ step \quad m: \quad G^{(m)} = G_i^{m-1} - \sum_{r=0}^{m-1} \alpha_{m,r}\left(a\Delta t \frac{\partial G^{(r)}}{\partial x}\Big|_i^-\right) \end{cases} \tag{1.16}$$

where n is the n^{th} time step. For 3 step Runge-Kutta:

$$\alpha_{0,1} = 1, \quad \alpha_{2,0} = -\frac{3}{4}, \alpha_{2,1} = \frac{1}{4}, \quad \alpha_{3,0} = -\frac{1}{12}, \alpha_{3,1} = -\frac{1}{12}, \alpha_{3,2} = \frac{2}{3} \tag{1.17}$$

1.4.6 Reinitialization

While the solution is advanced, an initially smooth solution starts to get unstable and G loses its distance function behaviour. However, if the level set values are reconstructed like initial condition, unstable solution can be prevented Chopp (2012). The other reason is in evaluating the curvature. To calculate the curvature the corerct G values are necessary not only for G_0. For this purposes the reinitialization is used. One of the following conditions can be implemented:

$$|\nabla G| = 1 \tag{1.18}$$

or the PDE reinitialization (Sussman *et al.* (1994)):

$$\frac{\partial G}{\partial t^*} + S_0(|\nabla G| - 1) = 0 \tag{1.19}$$

$$S0 = \frac{G}{\sqrt{G^2 + |\nabla G|^2 h_G^2}} \quad and \quad h_G : grid \quad size \tag{1.20}$$

In the RLSG the PDE reinitialization is used.

Although for the reasons mentioned above the reinitialization is necessary, it is not

mass conservative. In other words, if the reinitialization is done frequently, due to the error in reinitialization equation the G_0 values will change and causes volume and mass change. Thus, the reinitialization is not done in every time step and it is restricted to be done only under the following conditions:

$$min(|\nabla G|) < 10^{-4}, \quad max(|\nabla G|) > 2 \tag{1.21}$$

This condition is checked inside the N band and if it is satisfied the iteration for the PDE reinitialization equation is stoped.

1.4.7 Curvature Calculation in RLSG

The mean curvature is conventially calculated by the following equation Sethian (1999b):

$$\begin{aligned} \kappa = \nabla . \frac{\nabla G}{|\nabla G|} = & \frac{(G_{yy} + G_{zz})G_x^2 + (G_{xx} + Gzz)G_y^2 + (G_{xx} + G_{yy})G_z^2}{G_x^2 + G_y^2 + G_z^2} \\ & -2\frac{G_x G_y G_{xy} + G_x G_z G_{xz} + G_y G_z G_{yz}}{G_x^2 + G_y^2 + G_z^2} \end{aligned} \tag{1.22}$$

1.4.8 Parallel Implementation of RLSG, Domain Decomposition and Load Balancing

For partitioning the domain, at the begining of the simulation all the super grid cells are assigned to a processor. In the super grid look up table a negative value of the processors rank that contain the super grid block is stored. If a band grows to a super grid cell that was not active previously, a new processor is assigned to that super grid. However, this can cause load imbalancing. When the load-imbalance trigger is active, the processor with the most active super grid cells send its data to the processor with smallest active super grid cells. This is how the domain is decomposed for parallel purpose. The *PARMETIS* can be used for domain decomposotion. In next section the *PARMETIS* is discussed briefly.

1.4.9 PARMETIS

PARMETIS is a MPI-based library that implements several algorithm for partitioning the graphs. *PARMETIS* is appropriate for parallel numerical simulations with large meshes. It decreases the time in communication by computing mesh decomposition(Chevalier and Pellegrini (2008)).

In this code the "*ParMETIS_V3_PartKway*" function can be used for mesh decomposition. This function is as following:

*int ParMTIS_V3_PartKway(idx_t *vtxdist, idx_t *xadj, idx_t *adjncy, idx_t *vwgt, idx_t *adjwgt, idx_t *wgtflag, idx_t *numflag, idx_t *ncon, idx_t *nparts, real_t *tpwgts, real_t *ubvec, idx_t *options, idx_t *edgecut, idx_t *part, MPI_Comm *comm)*

Chevalier and Pellegrini (2008)

This function uses multilevel *k-way* partitioning algorithm to compute a k-way partitioning of graph on "p" processors. The *k-way* partition devides the graph(mesh) into k smaller parts (figure 1.9).

Parameters:

- vtxdist: The distribution of verrtices of graph among the processor
- xadj, adjncy: These arrays stores the adjancy of graph at each processor
- vwgt, adjwgt: The weight of the vertices
- wgtflag: A flag to check if the graph is weighted
- numflag: Indicates the numbering scheme. It is zero for C style
- ncon: Number of vertex that each vertex has
- nparts: Number of subdomains

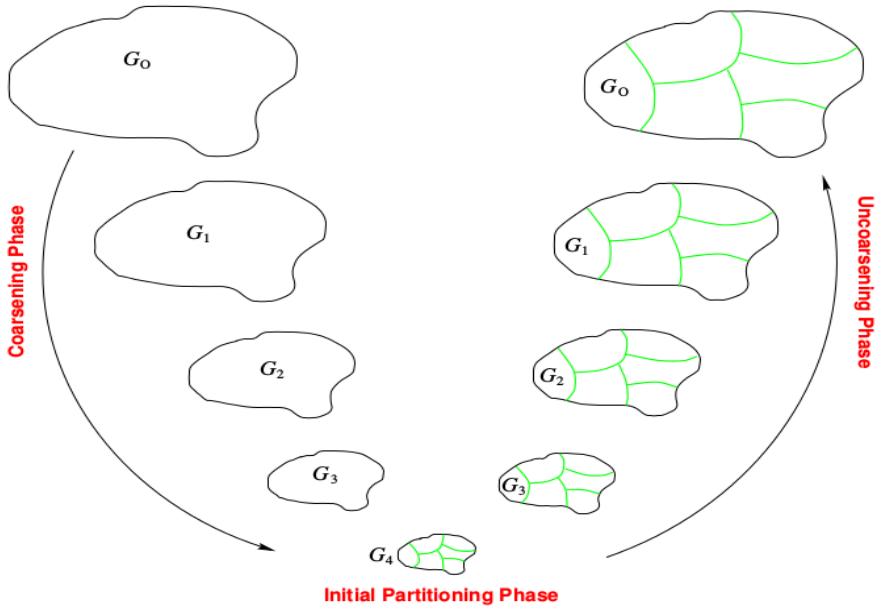


Figure 1.9: "k-way" Partitioning Chevalier and Pellegrini (2008)

- tpwgts: An array of size $ncon * nparts$
- ubvec: Used for imbalance tolerance
- options: Used to pass additional parameters
- edgecut: The number edges that are cut is written to this variable
- part: The partitioned vertices is written to this array
- comm: A pointer to MPI communicator

The "real_t" data type is for storing the double precision and "idx_t" is to store the signed integer. In the finite difference and finite element the vertices is the nodes of the cells and elements. The edges(2D) and faces(3D) are the connections among these nodes.

Chapter 2

THE C++ IMPLEMENTATION

2.1 Introduction

To write the code in C++, first all data in the FORTRAN modules are gathered and classified in specific classes. Almost, 36 classes are used for different purposes in this code. To use the members of classes inside other functions and classes, a unique pointer is defined and allocated where ever a particular member or function of a class is needed. In FORTRAN the data in the modules has static behavior. Thus, to have same behavior, some of the members in some particular classes are defined as static variables. Assume that the following code is a FORTRAN code in "file_m.f90"

```
module mod1
      some variables and types
      .
      .
      .
    contains:
      subroutine S1(the argument list)
      subroutine S2(the argument list)
      subroutine S3(the argument list)
      subroutine S4(the argument list)
end module mod1
```

In C++ the equivalent file of "file_m.f90" is written as "file.h" which containes:

```
class A1{
```

```
public:
   void S1(the argument list);
   void S2(the argument list);
   .
   .
   .
   static some variables;
private:
   some variables;
};
class A2{
public:
   void S3(the argument list);
   void S4(the argument list);
   .
   .
   .
   static some variables;
private:
   some variables;
};
```

where the members in module "mod1" with particular purposes are classified in "class A1" and "class A2". Now, consider the "file1.h" is defined as following:

```
class A{
public:
```

```
    void fun1(the argument list);
    static some objects or variables;
private:
    some objects or variables;
};
```

To use the "calss A" and its members in "class B" which is in an other file, the following procedure is done:

```
void B::function2(the argument list){
    std::unique_ptr<A> upA(new A);
    .
    .
    .
    upA->function1(the argument list);
    .
    .
    .
    upA->(some static variable in class A of file1);
}
```

For one dimensional arrays in C++ the "array" class from standard library is used. One dimensional arrays that are dynamically allocated in FORTRAN during run-time are defined as "vector" class in C++ from standard library. The vector is more efficient than arrays for adding an element to the end of the list. Moreover, deallocation is not required when "vector" is used and it will be automatically deallocated in appropriate place and time. However, for multi-dimensional arrays it is not reasonable to use multi-dimensional vectors. To preserve the pushing efficiency of the "vector",

compilers reserves some free spaces in memory. For instance, "gcc" complier reserve three spaces for this purpose. Assume that, there is an array of $500 \times 500 \times 500$ of doubles which requires $1GB$ of memory. If a three dimensional "vector" is used for this array, since there are three spaces in the back of the each dimension of the vector $1GB \times 9 + 3 \times 8 = 9GB$ amount of memory will be consumed. Therefore, it is not efficient and practical to define multi-dimensional "vector". The other way to have a "vector" for multi-dimensional arrays is to use an algorithm which access to the elements of multi-dimensional array through a one-dimensional vector. Consider array "arr" of dimension $d1 \times d1 \times d3$. If we desire to access the elements of this array to do some operation on them three "for loop" is needed such as following:

```
for(auto i=0; i<d1l i++){
    for(auto j=0; j<d2; j++){
        for(auto k=0; k<d3; k++){
            arr[i][j][k] = some operation here;
        }
    }
}
```

To define a one-dimensional "vector" to get access to all the elements of the arr the following algorithm can be used:

```
vec<double>(d1*d2*d3);
for(auto i=0; i<d1l i++){
    for(auto j=0; j<d2; j++){
        for(auto k=0; k<d3; k++){
            vec[i + j*d1 + k*d1*d2] = some operation here;
        }
```

```
    }
}
```

Although it is more convenient to define all the multi-dimensional arrays as one-dimensional "vector" through this algorithm, it is not efficient. To test it, $500MB$ of memory is used to define a 3-dimensional array and do some operation on each of its elements. The speed is much faster than when a one-dimensional "vector" is defined for this goal using the above algorithm. Thus, for multi-dimensional arrays in FORTRAN the C-type multi-dimensional arrays is used in C++.

One of the other problems with the FORTRAN code is the array bands. In FORTRAN the lower band of arrays can be started from any number such as -7 or 11. Hence, it is required to shift the lower band of this arrays in C++ code. Since it is difficult to identify all the arrays with this behaviour it causes problem and bugs in the C++ code. To solve this problem, a tempelate class in C++ could be defined such as following.

```
template<class T1, int st> class shift{
public:
    t1& operator[](int idx)
};

t1& shift::operator[](int idx){
    return t1[idx-st];
}
```

This method only works for one-dimensional arrays and for multi-dimensional arrays the overloading of "[]" operator is complex. For instance, to implement a 3D array access "a[i][j][k]=sn" operator "[]" has to return a reference to a 2D array which needs

to have its own operator[] that returns a reference to a 1D array which has to have operator[] that returns a reference to the element.

To run the test cases a directory is built and test cases is run in that directory. For this purpose, the "test.h" file is defined as following.

```
namespace lit_tests{
    class tests{
        virtual void setVelocity(){} ;
    };
};
```

Then in "test_case.h" file a "test" function is defined.

```
namespace lit_tests{
    void test01();
    void test02();
    void test03();
    void test04();
    static void run_test(const string &s) {
        auto word split(s);
        bool all = word[1]=="all";
        int myrank;
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        if(myrank==0){
            if(all || "test01") test01();
            if(all || "test02") test02();
            if(all || "test03") test03();
            if(all || "test04") test04();
```

```
        }
    }
};
```

Then inside the other files the "tese01()", "test02()", etc. functions are defined and for each test the "setVelocity" function is different. If in input file "test" is specified the "run_test" function will be run and execute one of the test files.

2.1.1 Files and Classes

In this section some of the most important classes will be introduced in the code. The first class that is used in all other classes is the "global_variable" class defined in the "datag" file. In this class the most common data and variables are collected. This class contains members that are frequently used almost in all parts of the code. The members such as the number of distinct G bands around $G = G0$, size of each band, G values for outermost T band cell, the array that stores either the rank of supergrid cell or the local block number, member to mark the active supergrid cells, and some other members to store number of global blocks, number of local blockes, size of supergrid global structure, size of local blocks, the size of narrow bands, grid coordinates, cell faces, grid sizes, start and end x, y, z coordinates of supergrid cells and minimum and maximum allowed curvature. It also contains variables that are used for dumping the solution and some pointers to other classes. Moreover, several functions are defined within the "global_variable" such as "setBlockPointers". In this function some pointers are set for short hand access to block variables and assign a value to the pointers defined in the class.

Within this file the other classes that are defined are, "block_t", "block_p_t", "Gnode_t", "Gnode_short_t" and "shape_t". "block_t" class stands for local block struc-

tures. It holds some members that store size of block in i, j and k direction, boundary of ghost cell index shift, start and end global index of block, number of G nodes that are transported, reinitialized, used in WENO stenciles and velocity/volume update(in flow solver band). It also hold a variable called "next" which points to the "block_t", necessary in the linked list data structure.

"block_p_t" class only includes a pointer to the "block_t" class.

Class "Gnode_t" holds some members that store level set scalar values, velocity vector and i, j, k coordinates. The other class, "Gnode_short_t" only contains level set scalar values and i, j, k coordinates.

The last class in this file is "shap_t". This class is used for initial interface shape data. It has some members to store the number of real and integer data that are used to define the shape of the initial interface.

The "advection" file has a class called "advection". This class includes some functions such as "lit_advection" which used for advection. Various methods are defined in this function for advection. First order upwind, 5^{th} order upwind centeral(UC-5), 7^{th} order upwind centeral(UC-7), 9^{th} order upwind centeral(UC-9), 11^{th} order upwind centeral(UC-11), WENO-3 and WENO-5 are defined. It also contains some private members such as WENO order for advection and parameters used for TVD-RK3 method.

The next file is "band". This file containes classes relevant to narrow band generation, maintenance and regeneration. It has two classes. The first class in "band" which includes some functions to generate, regenerate, expand and delete the narrow bands. It also contains some private members to store the number of skin cells, cloth cells and total cells in band. The second class in this file is "ghostcloth_bl_t". This class contains members relevant to ghost cells in super grids. It is used to gather data in sending and recieving among the processors when some super grids need to

get active due to band growth.

The "bl" file includes a class called "bl". In this class some functions relevant to supergrid cells are defined. In the "bl_init" function we initialize block by initial level set distribution. The "bl_clear_all_ghosts" is used to clear and free the block's ghost node storage. The "bl_remove_from_list" is used to deallocate a block from the block list. The "bl_activate_new" sets some general data for a newly created block that are required to calculate initial G and determine if block is active.

The "bound" file stands for boundary provides the ghost nodes update routines. It inculdes several functions and variables. Some members are used as marker for what type of boundary condition is used such as Neumann or Periodic. Some other variables are used as number of cells to be sent and recieved through MPI functions. Generally in "*_m_init" functions in the code the static variables of the class are initialized. Through "bound_m_init" function some variables are allocated used in sending and recieving. In "PrepareGhostNodes" function the ghost nodes of the bands or blocks are recognized and prepared. In "UpdateGhostNodes" the ghost nodes are updated due to band grow and regeneration. In the "get_bound_normals" the nodes in neighbourhood of a node are identified.

The next file in the code is "gnodes". In this file several functions such as "enlargenGnodes", "shortenGnodes", "addGnode" and "ensureSizeGnodes" are defined and used to change the "Gnodes" size.

In "init" file the G values of Gnode class are initialized. In this file through the "init_geometry" function the size of the cells, size of the super grid structure are calculated and weights for WENO and upwind centeral schemes used to discretize the space are set. In "G_init_value" the G values are initialized for the different shapes of inteface geometry such as plane, notched circle, circle, column, ring, rod, sphere,

cylinder, bubble, etc.

The routines to read input and dump the solutions are defined in the "io" file. In this file through the "read_input" function the input file is read and parsed and variables such as the start and end points for the grid, number of super grid cells, number of local cells, the shape of the geomtry and its initial position, variables to be written as output, advection sceheme and reinitialization scheme are extracted. In the "dumpSolution" function the procedure of dumping the solution is begun. Ensight Gold format is used to write the binary files. Inside this function the "dumpEnsight" function is called. In the "dumpEnsight" function first, the binary file of the geometry which stores the position of the nodes and number of the elements is written. The "MPI_Write_file" function is used to write this binary file. Then the function "writeEnsightScalar" and "writeEnsightVector" are called. These two functions are used to dump the scalar outputs such as level set values and vector values such as velocity, respectively. To read the file in Paraview, "writeCaseFileHeader" function is used to write a case file which is read by Paraview. The case file is shown in figure 2.1.

The Ensight Gold format for writing the geometry, scalar and vector variables are discussed in next sections.

The nxet file is "litBuffer". In this file some functions are defined to allocate and deallocate multidimensional arrays.

The core file of the code is "lit_coupler". In this file four functions of "lit_initialize", "lit_initialize2", "litRunIteration" and "lit_finalize" are called. In the first function MPI_Init function and some other functions for preparing the files to monitor the running procedure of the code are called. In the "lit_initialize2" function all the prequisites before the advection are initialized. In the "litRunIteration" the advection and reinitialization are done. Finally, through this file, "lit_finalize" function is

```
FORMAT
type:   ensight gold
GEOMETRY
model:   1  sol_lit_******.geo
VARIABLE
scalar per node:   1  G   sol_lit_******_G.scalar
TIME
time set:  1
number of steps:            9
filename start number: 000000
filename increment: 000001
time values:
 0.000000000000000E+000
 1.00000000000000
  2.00000000000000
  3.00000000000000
  4.00000000000000
  5.00000000000000
  6.00000000000000
  7.00000000000000
  8.00000000000000
```

Figure 2.1: Case File That Is Read By Paraview

called which is used as the last function in the "main.cpp". Through this function all the bands are cleaned up and the "MPI_Finalize" is called. In the "param" file some functions are defined to parse the input file. In the "redist" file "lit_redist_pde" routine is defined for redistribution of vectors in the interface normal direction. This function is used in filtering the velocity.

In the "reinit" file, the routines for reinitialization of the level set scalar values are defined. Through this file, the pde reinitialization and the routine for calculation of the sign are defined.

The "sg" file contains functions used for the super grid global structure. In this file the super grid global structure are built up. We also control the band regeneration and identify the active super grid cells. The functions to balance the load are also defined in this file. Two load balancing function are defined. One of them is a function that balance the load manually and the other one uses Parmetis to balance the

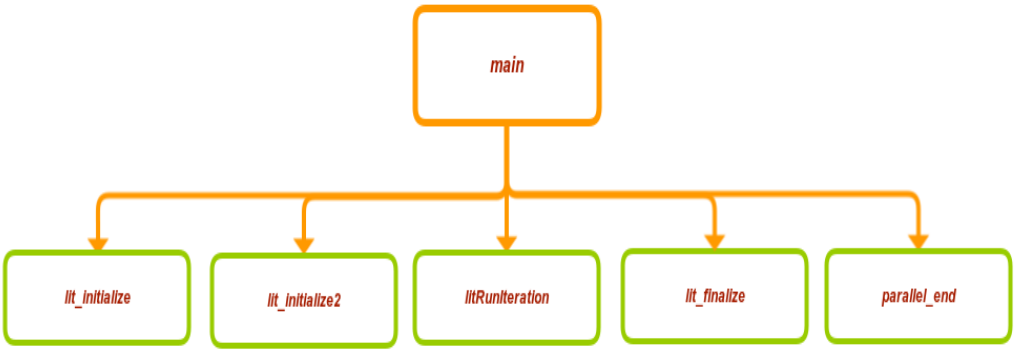


Figure 2.2: The Main Branch of The Code

load among the processors.

There are some other files that are used in the code. In the "timeStep" file the stable time step is calculated. In the "toolbox" file several functions are used such as heavyside, curvature and normal which are used in different parts of the code to solve the level set equation and calculate the curvature and surface tension. In the "weno" file the WENO scheme is defined. In the "string" file some tools are defined to compare two strings, convert the string to uppercase or lowecase and trim the string.

2.2 The Map of The Code

In this section we explain how the code works. The figure 2.2 shows how the functions are called through the main function. Inside the main function, "lit_initialize" is called first. The figure 2.3 illustrate how this function works. In this function the

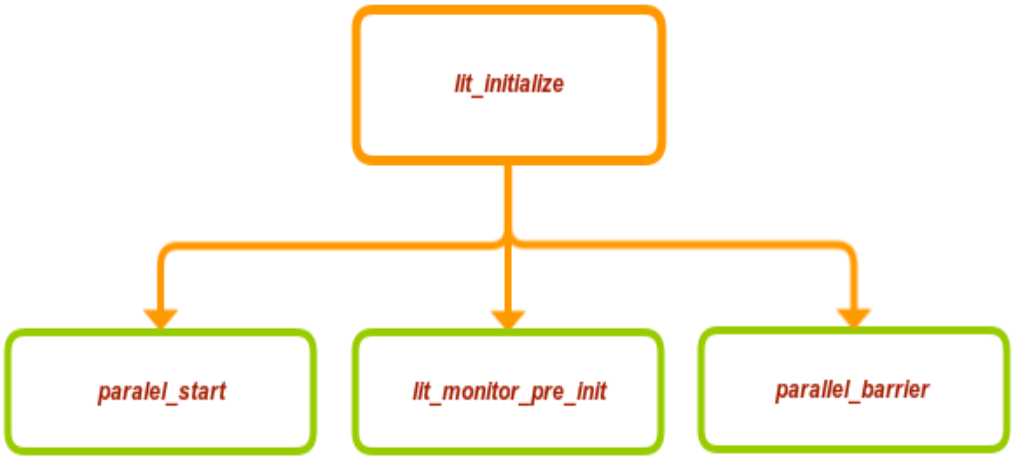


Figure 2.3: The "lit_initialize" Function

MPI in initialized by calling MPI_Init. The dumping directory is also created for the solution. Then MPI_Barrier is called to allow all processors to reach this point of the code.

Then the "lit_initialize2" function is called. Figure 2.5 shows how the functions are called in this function. Through this function, first the time and date of running the code are set to monitor them in the monitor files. Then "io_init" function is called. In this function the input file is read. Input file format is shown in figure 2.4. In first and second lines the start and end points of the grid are read. At third and fourth lines the number of supergrid cells and local block cells are read in x, y and z direction, respectively. Then the minimum and maximum value of gradient of G are read to use in the reinitialization. In next line the initial geometry of the interface and the parameters that is needed to define it are read. In the last line of the input file, the output format and the variables that will be written as output to be visualized in paraview, are read.

Then by calling the "init_geometry" fuction, size of Cartesion grid cell in each direction, volume of cells, minimum grid size, minimum and maximum value of allowed

```
LIT.XYZS_SG = 0.0  0.0   0.0
LIT.XYZE_SG = 1.0  1.0   0.01
LIT.IJKM_GL = 100  100    1
LIT.IJKM_BL = 50  50   1
LIT.REINIT_TRIGGER_MIN = 1.0e-4
LIT.REINIT_TRIGGER_MAX = 2.0
LIT.REINIT_STEPS_MIN = 1
LIT.NVELFILTER = 0
LIT.INIT_SHAPE notched_circle 0.5 0.75 0.0 0.15 0.05 0.25
WRITE_LIT_STEP ENSIGHT zalesak 157 T-BAND G V
```

Figure 2.4: The Input File Format

curvature, size of super grid structure, size of global structure, size of local blocks, grid coordinates and cell faces are initialized. Then the "band_m_init" function is called. Through this function static variables in the band class such as band sizes(number of cells in each band) for A, T, N, W and X band are allocated and initialized. Then the G values are set for the first and last band cell that need to be advected, maximum number of subsycles before reinitialization, band node start and end indices and ghost nodes start and end indices(each node has 26 point at its neighbourhood). The next step is calling the "reinit_m_init" function. Through this function the static variables in the "reinit" class are initialized. The time step, maximum number of iteration to solve the reinitialization equation 1.19 and the α values for RK method are initialized. The next functions that are called in the "lit_initialize2" are "redist_m_init" function, "timestep" function, "toolbox_m_init" function and "sg_m_init" function. Then by calling the "sg_init" function, the initial super grid structure is built up. In this function the array for super grid structure is allocated and all active blocks inside the domain are identified. At the end of this function by calling "sg_calc_active" function the active supergrid cells are calculated. Then by calling the "band_init"

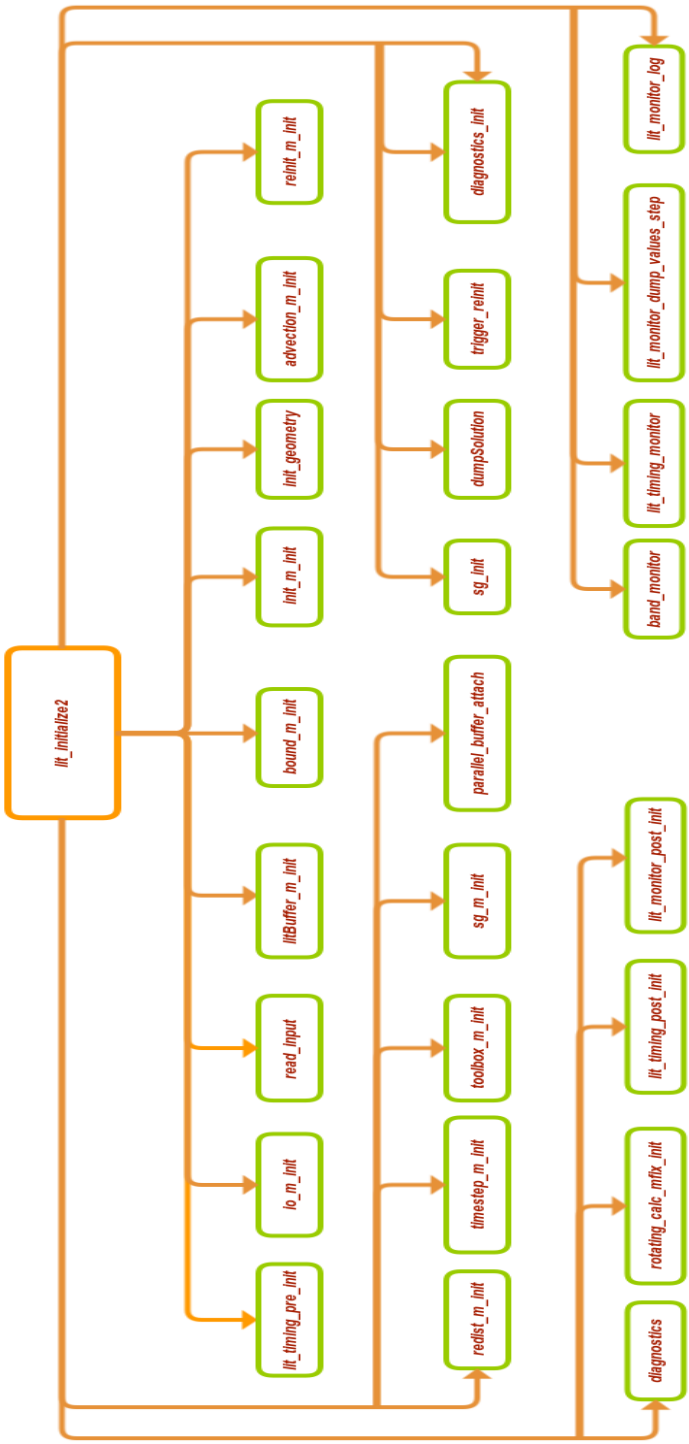


Figure 2.5: The "lit_initialize2" Function

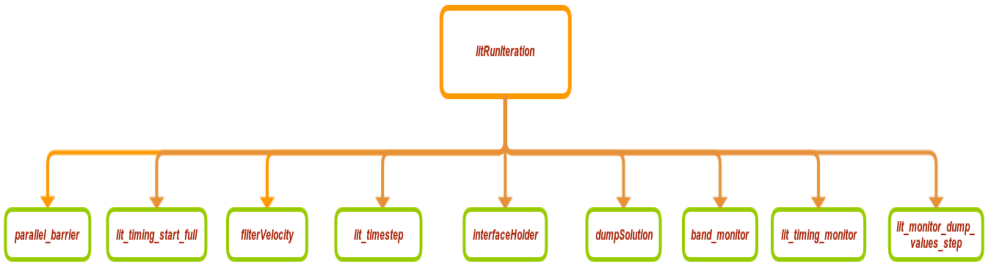


Figure 2.6: The "litRunIteration" Function

function the band structures from the intial settinig function of G are generated and initialized. At the end, "sg_load_balance" is called to decompose the domain. The next step is dumping the solutions. This function is called two times. By calling it for first time inside the "lit_initialize2" function, the case file is created and solution is written at $time = 0$. After the "lit_initialize2" function, the "setVelocity" function is called to initialize the velocity field at $time = 0$. Then inside a time loop, for each time step the velocity is set and the function "litRunIteration" is called. Figure 2.6 shows how this function works. Through this function, by calling "lit_timestep" the advection(marching in time) and reinitialization (if "trigger_reinitializatuion" is active) are done. This function also computes the stable time step using CFL in narrow bands. At the end by calling "dumpSolution" function for some specific time steps the binary file for the output variables such as level set scalar values and velocity vectors are written.

2.3 "Ensight Gold" Format

To visualize the results with Paraview, the solution is written in Ensight Gold format. Ensight Gold data includes the following files:

- case: Includes all other required files including model geometry and variables.
- sources of Ensight Gold format data: All the files including the geometry and variables wich can be written either in binary or ASCII.

The case file is an ASCII format file that includes all the files and names for model, variable and time. It contains five sections of, "FORMAT", "GEOMETRY", "VARIABLE", "TIME" and "FILE". Figure 2.1 is a case file in Ensight Gold format the is used in the code.

Figure 2.7 illustrates the binary format for geometry in Ensight Gold. where:

- # = A part number
- nn = Total number of nodes in part
- ne = Number of elements of a given type
- np = Number of nodes per element for a given element type
- nf = Number of faces per element
- id_* = Node or element id number
- x_* = x component
- y_* = y component
- z_* = z component

```
C Binary                                                    80 chars
description line 1                                          80 chars
description line 2                                          80 chars
node id <off/given/assign/ignore>                           80 chars
element id <off/given/assign/ignore>                        80 chars
[extents                                                    80 chars
xmin xmax ymin ymax zmin zmax]                               6 floats
part                                                        80 chars
#                                                            1 int
description line                                            80 chars
coordinates                                                 80 chars
nn                                                           1 int
[id_n1 id_n2 ... id_nn]                                     nn ints
x_n1 x_n2 ... x_nn                                          nn floats
y_n1 y_n2 ... y_nn                                          nn floats
z_n1 z_n2 ... z_nn                                          nn floats
element type                                                80 chars
ne                                                           1 int
[id_e1 id_e2 ... id_ne]                                     ne ints
n1_e1 n2_e1 ... np_e1
n1_e2 n2_e2 ... np_e2
 .
 .
n1_ne n2_ne ... np_ne                                       ne*np ints
element type                                                80 chars
 .
 .
part                                                        80 chars
 .
 .
part                                                        80 chars
#                                                            1 int
description line                                            80 chars
block [iblanked] [with_ghost] [range]                       80 chars
i j k            # nn = i*j*k, ne = (i-1)*(j-1)*(k-1)        3 ints
[imin imax jmin jmax kmin kmax]  # if range used:            6 ints
                                  nn = (imax-imin+1)*(jmax-jmin+1)* (kmax-kmin+1)
                                       ne = (imax-imin)*(jmax-jmin)*(kmax-kmin)
x_n1 x_n2 ... x_nn                                          nn floats
y_n1 y_n2 ... y_nn                                          nn floats
z_n1 z_n2 ... z_nn                                          nn floats
[ib_n1 ib_n2 ... ib_nn]                                     nn ints
[ghost_flags]                                               80 chars
[gf_e1 gf_e2 ... gf_ne]                                     ne ints
[node_ids]                                                  80 chars
[id_n1 id_n2 ... id_nn]                                     nn ints
[element_ids]                                               80 chars
[id_e1 id_e2 ... id_ne]                                     ne ints
part                                                        80 chars
#                                                            1 int
description line                                            80 chars
block rectilinear [iblanked] [with_ghost] [range]           80 chars
i j k            # nn = i*j*k, ne = (i-1)*(j-1)*(k-1)        3 ints
[imin imax jmin jmax kmin kmax]  # if range used:            6 ints
                                  nn = (imax-imin+1)*(jmax-jmin+1)* (kmax-kmin+1)
                                       ne = (imax-imin)*(jmax-jmin)*(kmax-kmin)
x_1 x_2 ... x_i                                              i floats
y_1 y_2 ... y_j                                              j floats

z_1 z_2 ... z_k                                              k floats
[ib_n1 ib_n2 ... ib_nn]                                     nn ints
[ghost_flags]                                               80 chars
[gf_e1 gf_e2 ... gf_ne]                                     ne ints
[node_ids]                                                  80 chars
[id_n1 id_n2 ... id_nn]                                     nn ints
[element_ids]                                               80 chars
[id_e1 id_e2 ... id_ne]                                     ne ints
part                                                        80 chars
#                                                            1 int
description line                                            80 chars
block uniform [iblanked] [with_ghost] [range]               80 chars
i j k            # nn = i*j*k, ne = (i-1)*(j-1)*(k-1)        3 ints
[imin imax jmin jmax kmin kmax]  # if range used:            6 ints
                                  nn = (imax-imin+1)*(jmax-jmin+1)*(kmax-kmin+1)
                                       ne = (imax-imin)*(jmax-jmin)*(kmax-kmin)
x_origin y_origin z_origin                                   3 floats
x_delta y_delta z_delta                                      3 floats
[ib_n1 ib_n2 ... ib_nn]                                     nn ints
[ghost_flags]                                               80 chars
[gf_e1 gf_e2 ... gf_ne]                                     ne ints
[node_ids]                                                  80 chars
[id_n1 id_n2 ... id_nn]                                     nn ints
[element_ids]                                               80 chars
[id_e1 id_e2 ... id_ne]                                     ne ints
```

Figure 2.7: The "GEOMETRY" File Format in Ensight Gold

```
description line 1                                    80 chars
part                                                  80 chars
#                                                      1 int
coordinates                                           80 chars
s_n1 s_n2 ... s_nn                                    nn floats
part                                                  80 chars
 .
 .
part                                                  80 chars
#                                                      1 int
block                    # nn = i*j*k                 80 chars
s_n1 s_n2 ... s_nn                                    nn floats
```

Figure 2.8: The "SCALAR" File Format in Ensight Gold

- n*_e* = Node number for element
- f*_e* = Face number for an element
- [] = Optional portion

Figure 2.8 shows the procedure of writing the scalar such as level set scalar values(G) in Ensight Gold format. Figure 2.9 shows the procesure of writing a vector such as velocity in Ensight Gold format.

```
description line 1                                    80 chars
part                                                  80 chars
#                                                      1 int
coordinates                                           80 chars
s_n1 s_n2 ... s_nn                                    nn floats
part                                                  80 chars
 .
 .
part                                                  80 chars
#                                                      1 int
block                    # nn = i*j*k                 80 chars
s_n1 s_n2 ... s_nn                                    nn floats
```

Figure 2.9: The "VECTOR" File Format in Ensight Gold

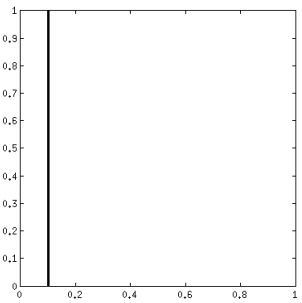
Chapter 3

RESULTS

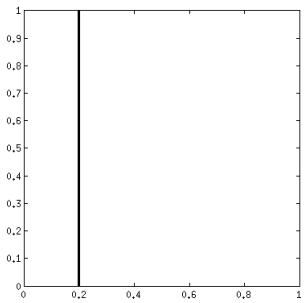
In this chapter, the results for some test cases are shown. The results are shown for interface (zero level set), "T" band and "X" band.

3.1 Vertical Plane

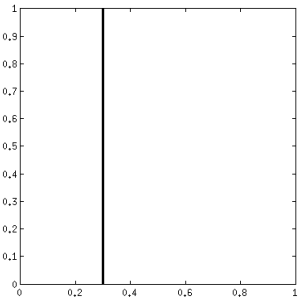
In this test, the code is run for a vertical plane initially located at $x = 0.1$. The velocity of the field is $\mathbf{u}(\mathbf{x}, t) = (1.0, 0.0)$ and $h_G = 1/256$.
Figures 3.1, 3.2 and 3.3 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/256$.



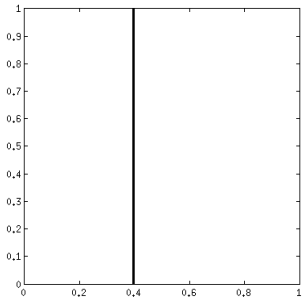
(a) $t = 0.0s$



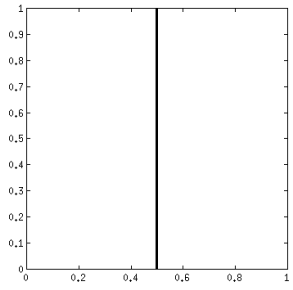
(b) $t = 0.1s$



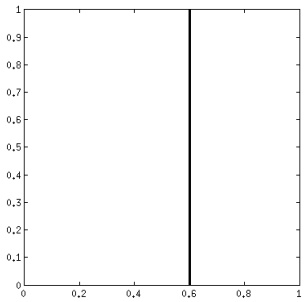
(c) $t = 0.2s$



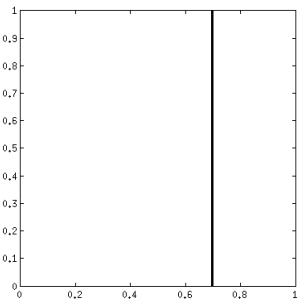
(d) $t = 0.3s$



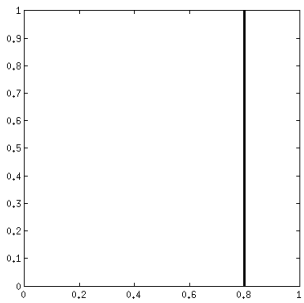
(e) $t = 0.4s$



(f) $t = 0.5s$

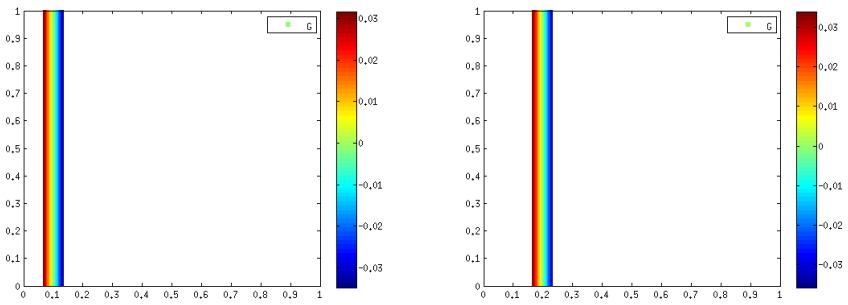


(g) $t = 0.6s$

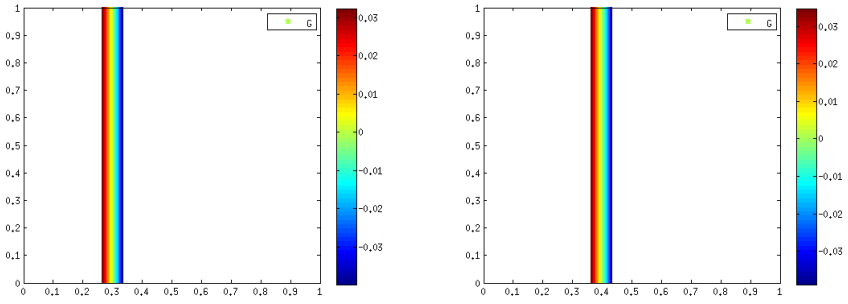


(h) $t = 0.7s$

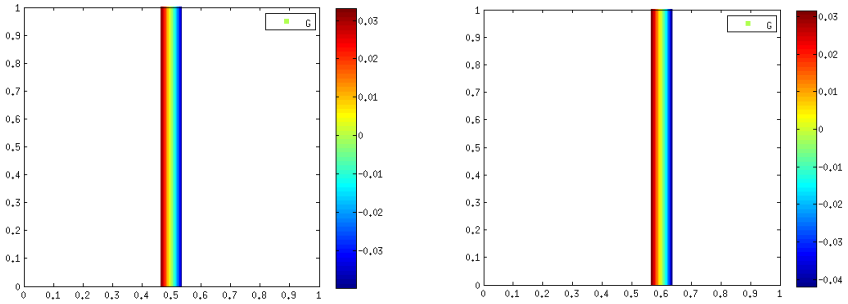
Figure 3.1: Zero Level Set for Vertical Plane



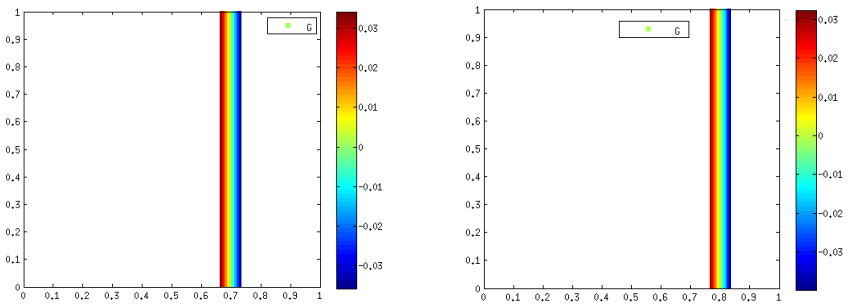
(a) $t = 0.0s$ (b) $t = 0.1s$



(c) $t = 0.2s$ (d) $t = 0.3s$

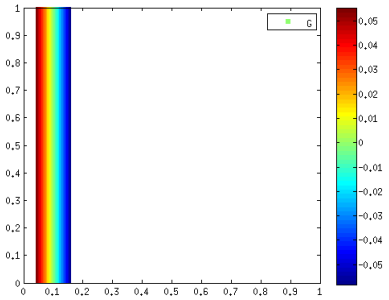


(e) $t = 0.4s$ (f) $t = 0.5s$

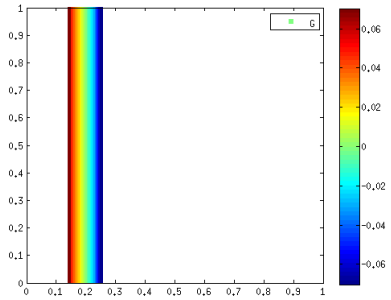


(g) $t = 0.6s$ (h) $t = 0.7s$

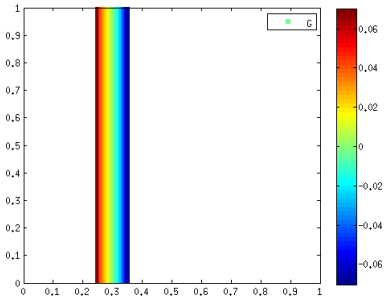
Figure 3.2: Level Set Values in "T" Band for Vertical Plane



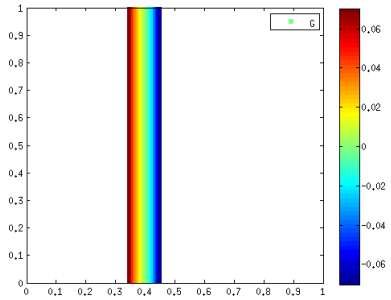
(a) $t = 0.0s$



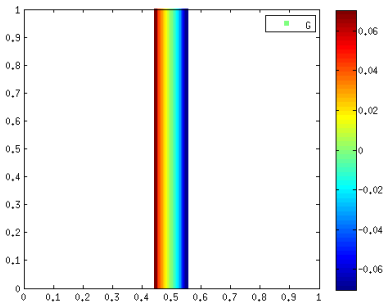
(b) $t = 0.1s$



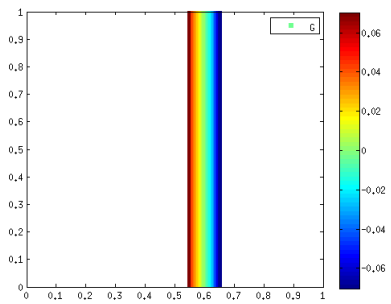
(c) $t = 0.2s$



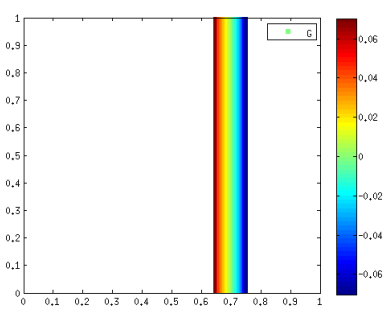
(d) $t = 0.3s$



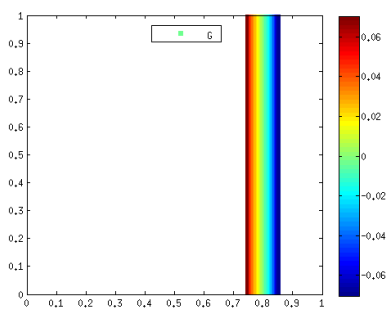
(e) $t = 0.4s$



(f) $t = 0.5s$



(g) $t = 0.6s$

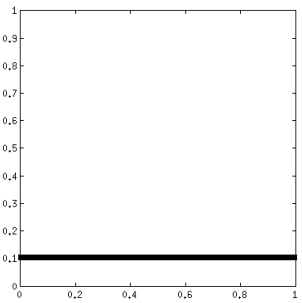


(h) $t = 0.7s$

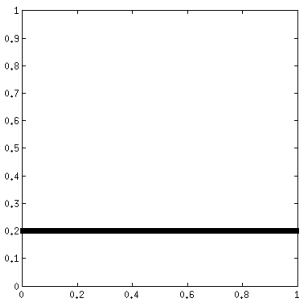
Figure 3.3: Level Set Values in "X" Band for Vertical Plane

3.2 Horizontal Plane

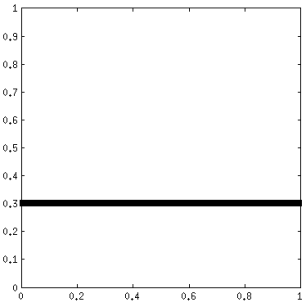
In this test, the code is run for a horizontal plane initially located at $y = 0.1$. The velocity of the field is $\mathbf{u}(\mathbf{x}, t) = (0.0, 1.0)$ and $h_G = 1/256$. Figures 3.4, 3.5 and 3.6 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/256$.



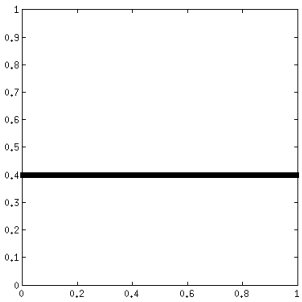
(a) $t = 0.0s$



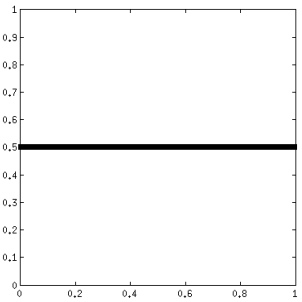
(b) $t = 0.1s$



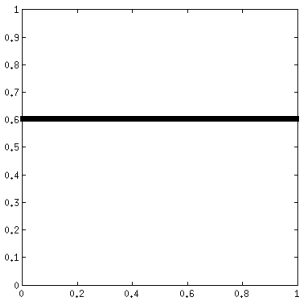
(c) $t = 0.2s$



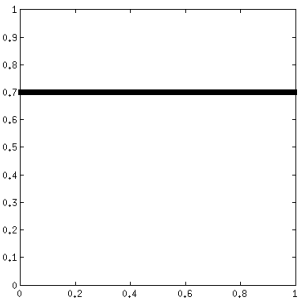
(d) $t = 0.3s$



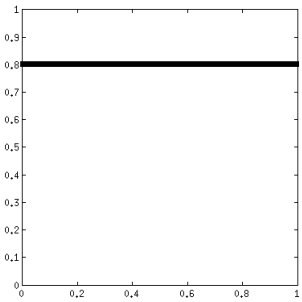
(e) $t = 0.4s$



(f) $t = 0.5s$

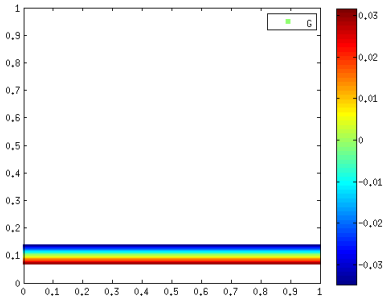


(g) $t = 0.6s$

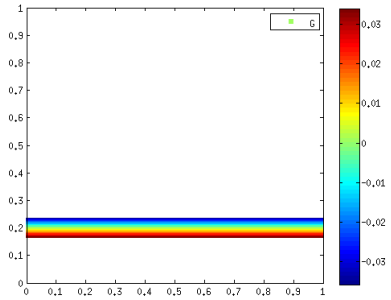


(h) $t = 0.7s$

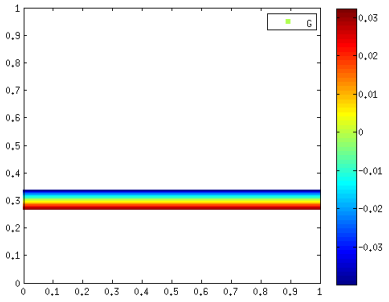
Figure 3.4: Zero Level Set for Horizontal Plane



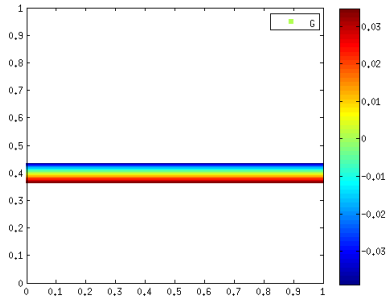
(a) $t = 0.0s$



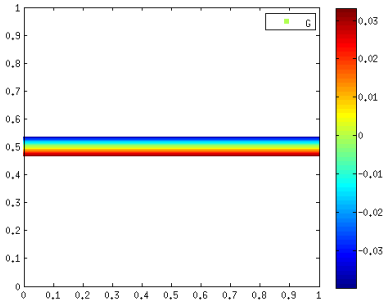
(b) $t = 0.1s$



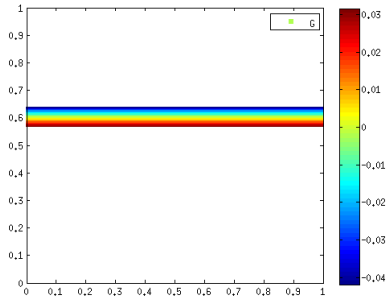
(c) $t = 0.2s$



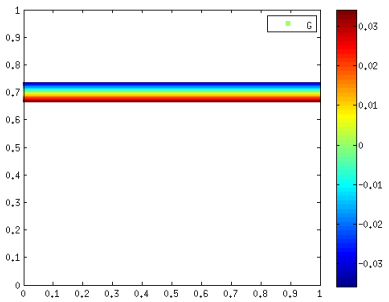
(d) $t = 0.3s$



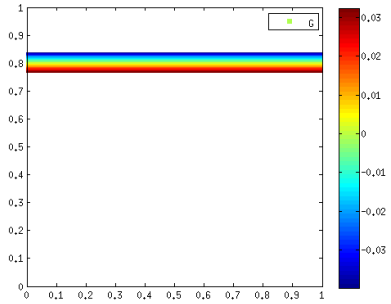
(e) $t = 0.4s$



(f) $t = 0.5s$

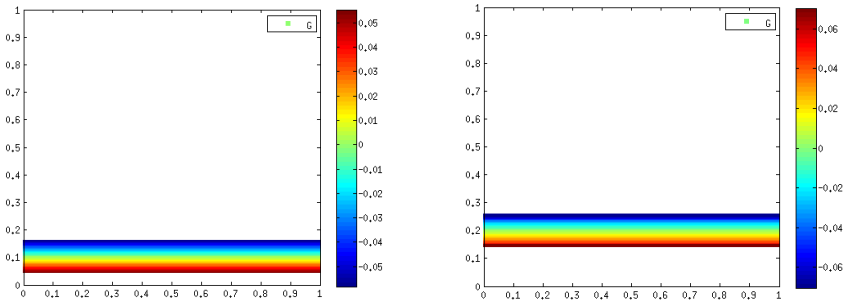


(g) $t = 0.6s$

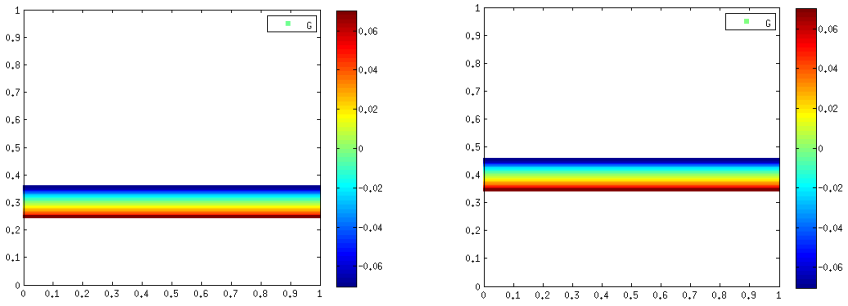


(h) $t = 0.7s$

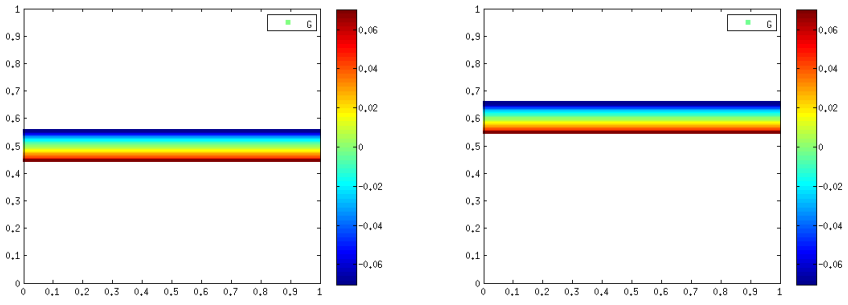
Figure 3.5: Level Set Values in "T" Band for Horizontal Plane



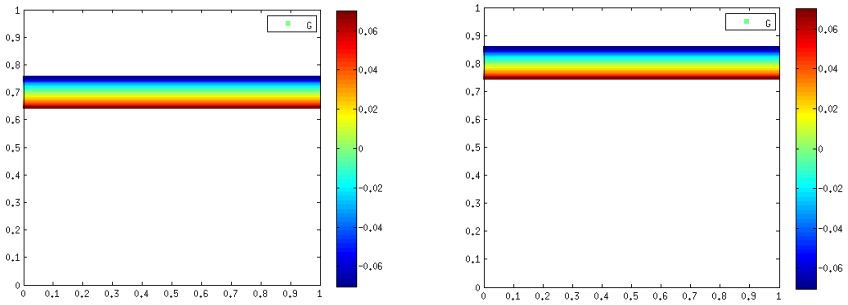
(a) $t = 0.0s$ (b) $t = 0.1s$



(c) $t = 0.2s$ (d) $t = 0.3s$



(e) $t = 0.4s$ (f) $t = 0.5s$



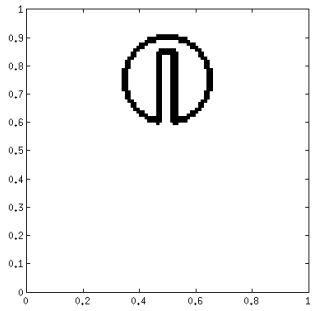
(g) $t = 0.6s$ (h) $t = 0.7s$

Figure 3.6: Level Set Values in "X" Band for Horizontal Plane

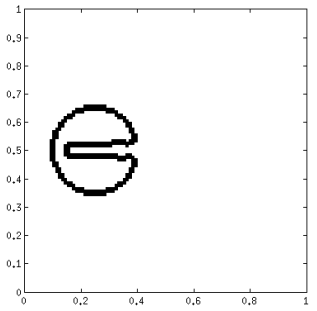
3.3 Zalesak's Disk

In this section the code is teseted for solid body rotation of a notched circle, also known as Zalesak's disk. A disk of redius 0.15 notch width 0.15 and notch height "0.25" is placed in a 1×1 box. The velocity field is $\mathbf{u}(\mathbf{x}, t) = (0.5 - y, x - 0.5)$. The time step is $\Delta t = 2\pi/628$.

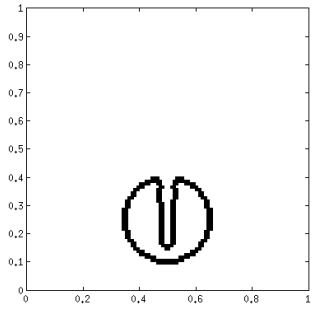
Figures 3.7, 3.8 and 3.9 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/100$.



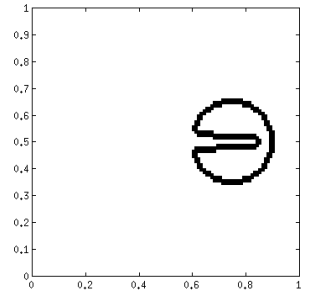
(a) $t = 0.0s$



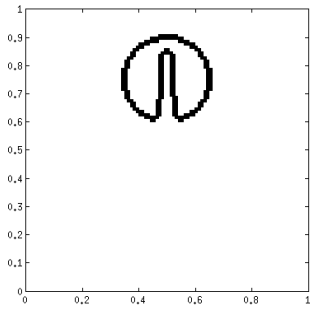
(b) $t = \pi/2s$



(c) $t = \pi s$

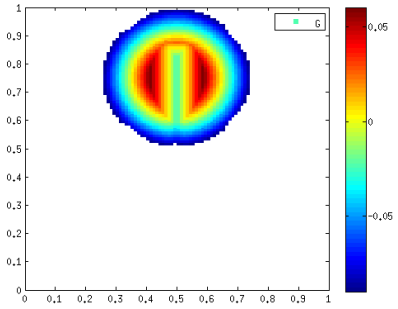


(d) $t = 3\pi/2s$

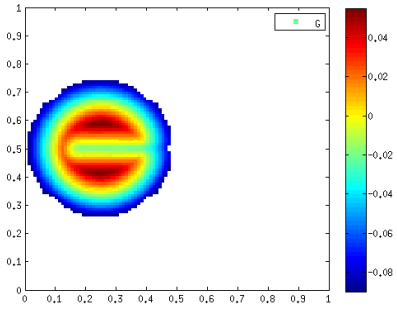


(e) $t = 2\pi s$

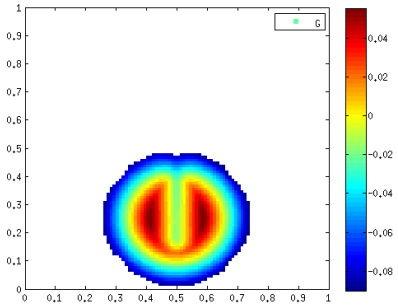
Figure 3.7: Zero Level Set for Zalesak's Disk With $h_G = 1/100$



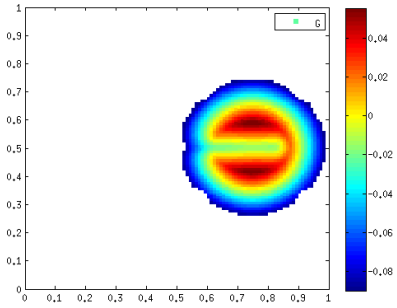
(a) $t = 0.0s$



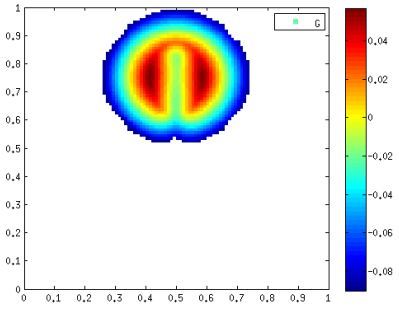
(b) $t = \pi/2s$



(c) $t = \pi s$

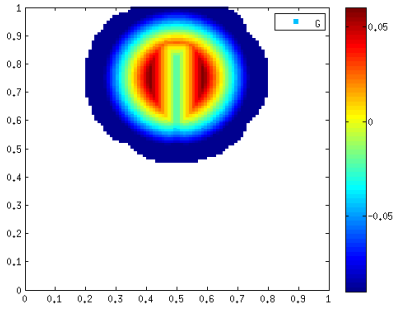


(d) $t = 3\pi/2s$

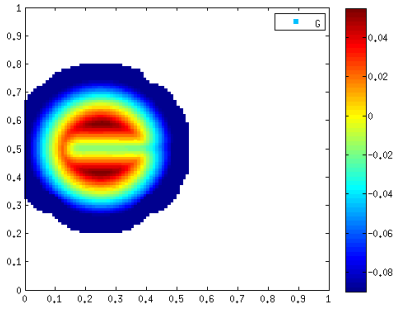


(e) $t = 2\pi s$

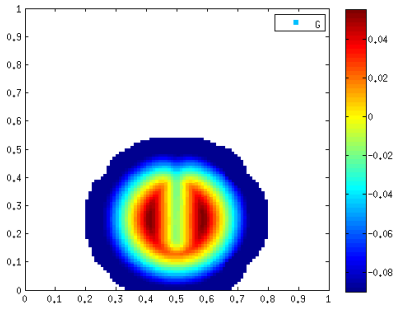
Figure 3.8: Level Set Values for Zalesak's Disk in "T" Band With $h_G = 1/00$



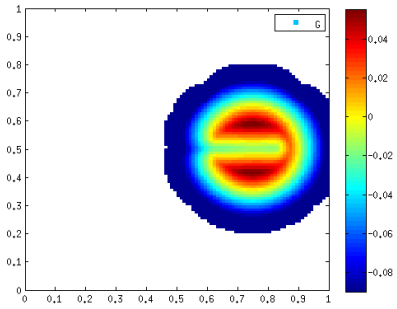
(a) $t = 0.0s$



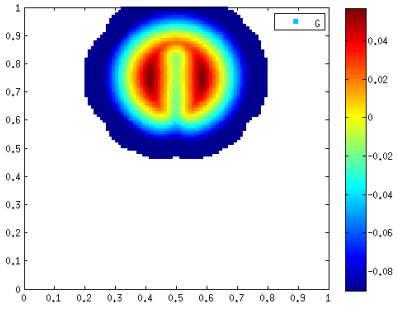
(b) $t = \pi/2s$



(c) $t = \pi s$



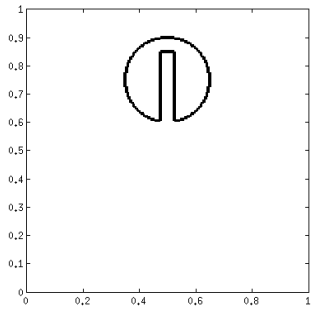
(d) $t = 3\pi/2s$



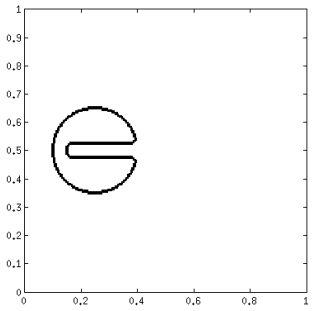
(e) $t = 2\pi s$

Figure 3.9: Level Set Values for Zalesak's Disk in "X" Band With $h_G = 1/100$

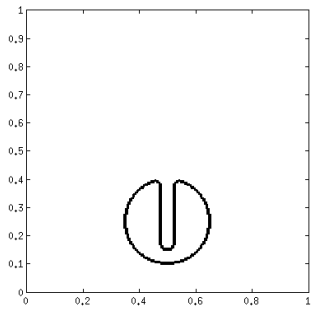
Figures 3.10, 3.11 and 3.12 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/200$.



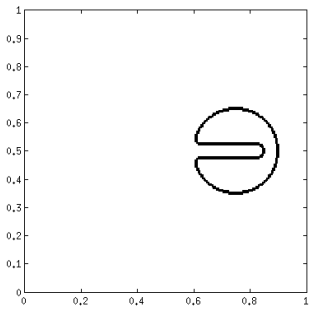
(a) $t = 0.0s$



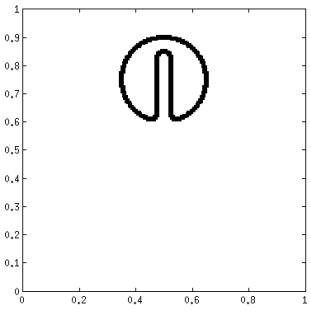
(b) $t = \pi/2s$



(c) $t = \pi s$

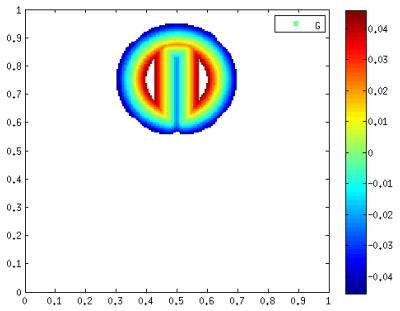


(d) $t = 3\pi/2s$

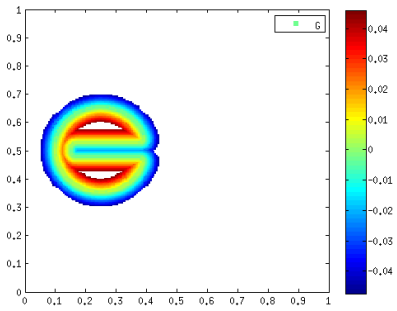


(e) $t = 2\pi s$

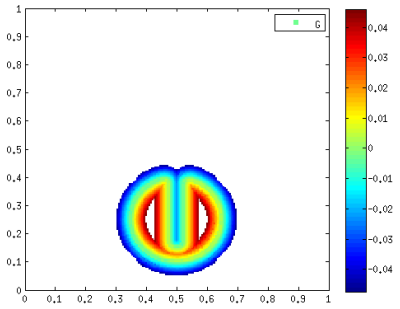
Figure 3.10: Zero Level Set for Zalesak's Disk With $h_G = 1/200$



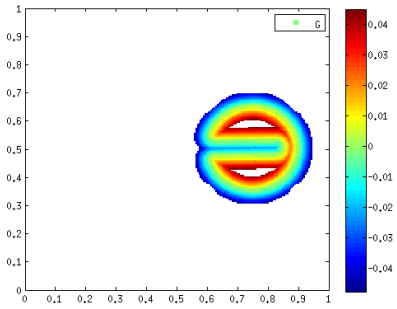
(a) $t = 0.0s$



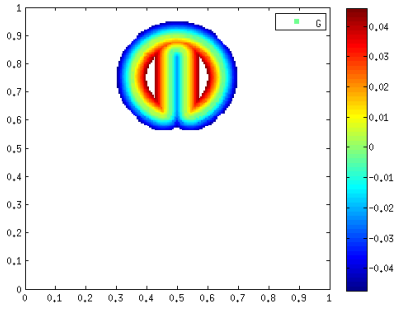
(b) $t = \pi/2s$



(c) $t = \pi s$

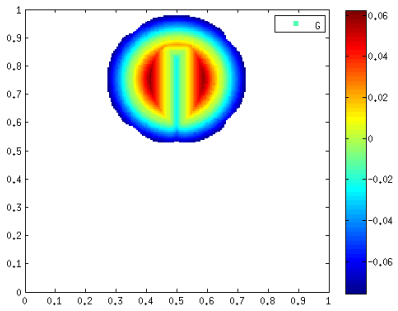


(d) $t = 3\pi/2s$

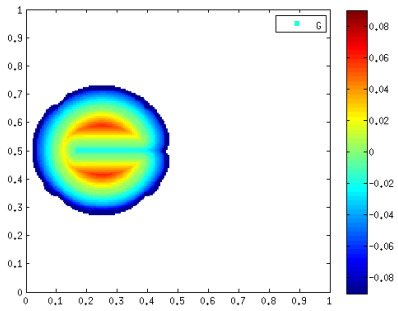


(e) $t = 2\pi s$

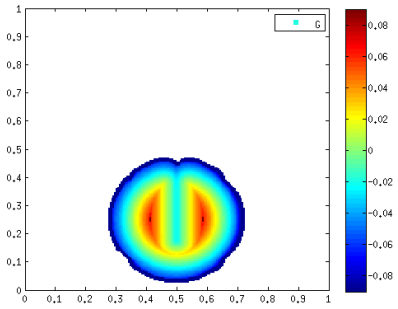
Figure 3.11: Level Set Values for Zalesak's Disk in "T" Band With $h_G = 1/200$



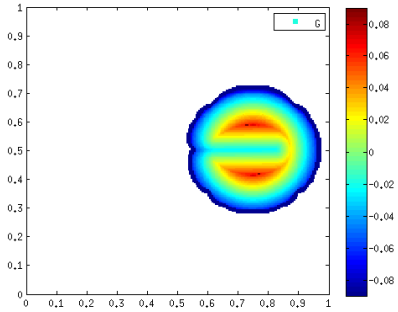
(a) $t = 0.0s$



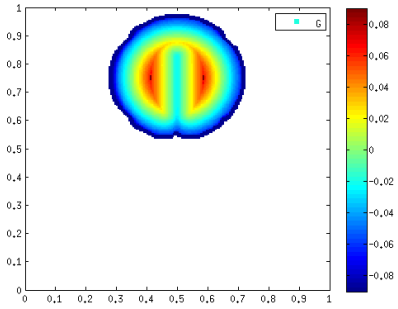
(b) $t = \pi/2s$



(c) $t = \pi s$



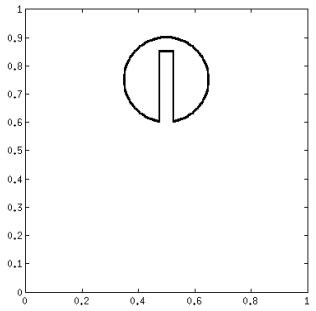
(d) $t = 3\pi/2s$



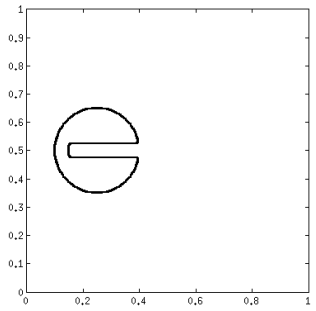
(e) $t = 2\pi s$

Figure 3.12: Level Set Values for Zalesak's Disk in "X" Band With $h_G = 1/200$

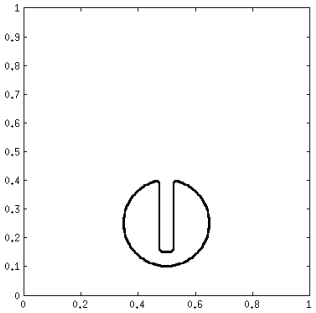
Figures 3.13, 3.14 and 3.15 show the results for interface(zero levelset), "T" band and "X" band, respectively with $h_G = 1/400$.



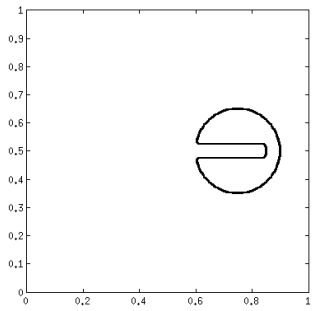
(a) $t = 0.0s$



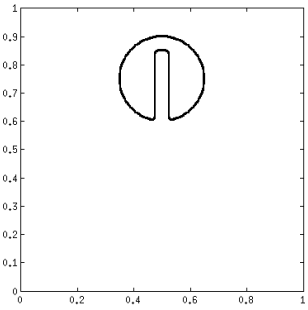
(b) $t = \pi/2s$



(c) $t = \pi s$

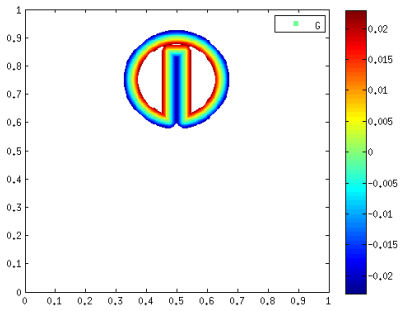


(d) $t = 3\pi/2s$

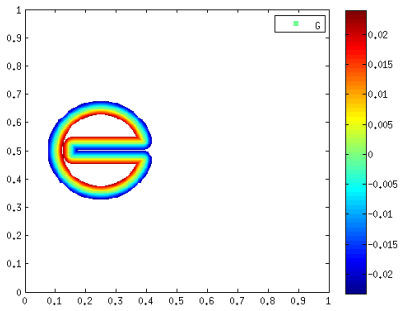


(e) $t = 2\pi s$

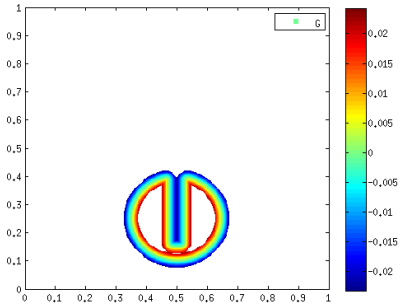
Figure 3.13: Zero Level Set for Zalesak's Disk With $h_G = 1/400$



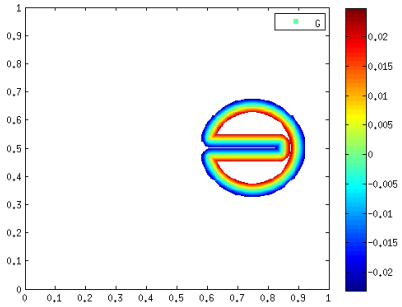
(a) $t = 0.0s$



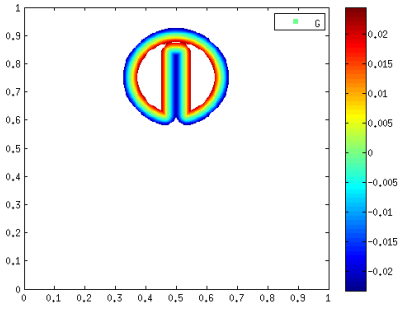
(b) $t = \pi/2s$



(c) $t = \pi s$

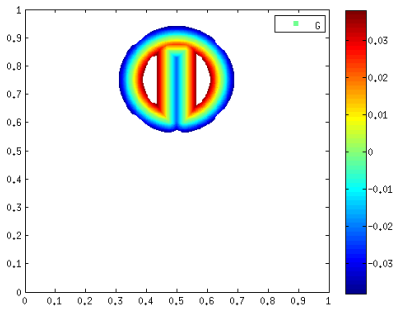


(d) $t = 3\pi/2s$

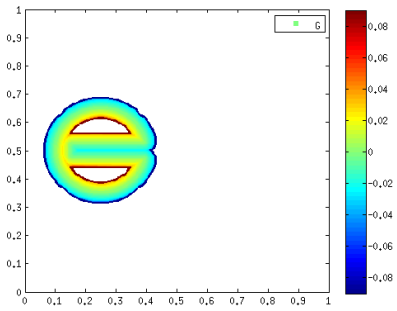


(e) $t = 2\pi s$

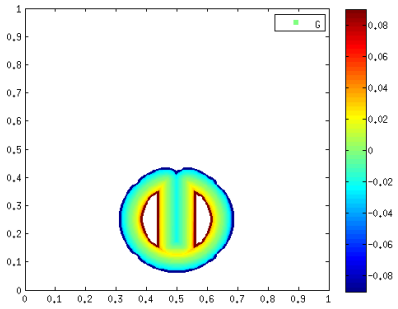
Figure 3.14: Level Set Values for Zalesak's Disk in "T" Band With $h_G = 1/400$



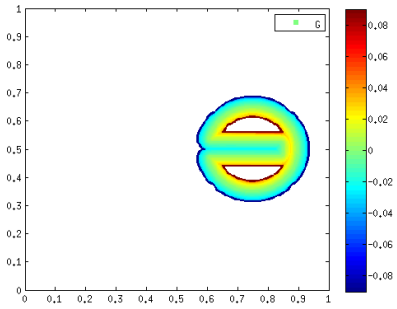
(a) $t = 0.0s$



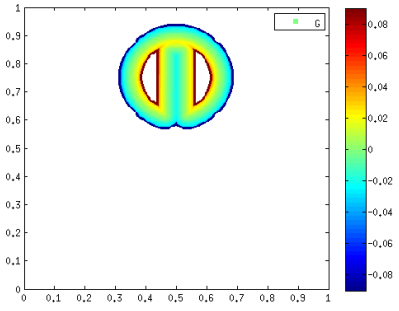
(b) $t = \pi/2s$



(c) $t = \pi s$



(d) $t = 3\pi/2s$



(e) $t = 2\pi s$

Figure 3.15: Level Set Values for Zalesak's Disk in "X" Band With $h_G = 1/400$

3.4 Circle in a Deformation Field

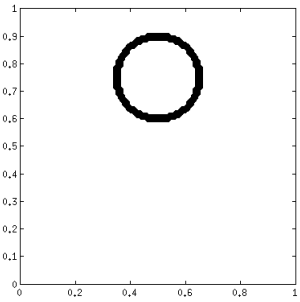
In this section the code is teseted for a circle in a deformation field. A circle of redius 0.15 and center $(x, y) = (0.5, 0.75)$ is placed inside a 1×1 box. The velocity field is :

$$\mathbf{u}(\mathbf{x}, t) = -2.0 sin^2(\pi x) sin(\pi y) cos(\pi y) cos(\pi t/T)$$

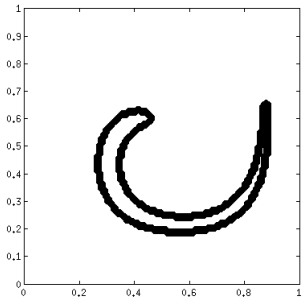
$$\mathbf{v}(\mathbf{x}, t) = -2.0 sin^2(\pi y) sin(\pi x) cos(\pi x) cos(\pi t/T)$$

The time step is $\Delta t = 1/256$.

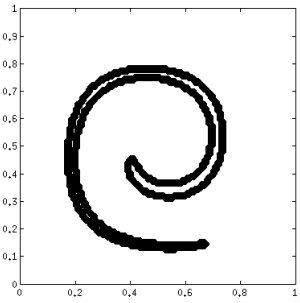
Figures 3.16, 3.17 and 3.18 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/128$.



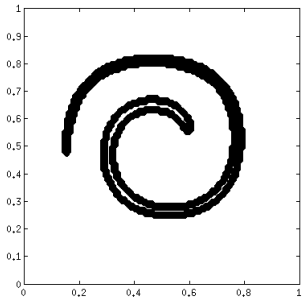
(a) $t = 0.0s$



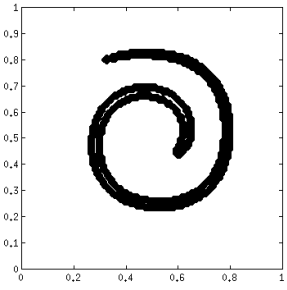
(b) $t = 1.0s$



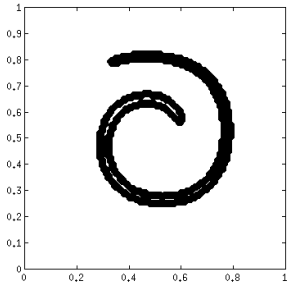
(c) $t = 2.0s$



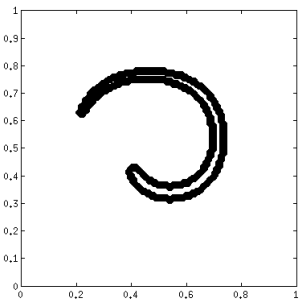
(d) $t = 3.0s$



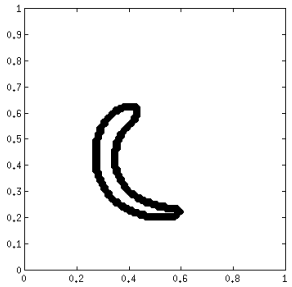
(e) $t = 4.0s$



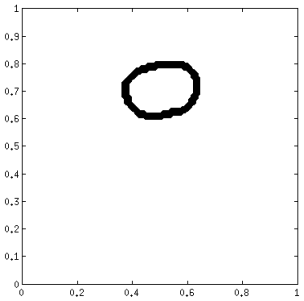
(f) $t = 5.0s$



(g) $t = 6.0s$

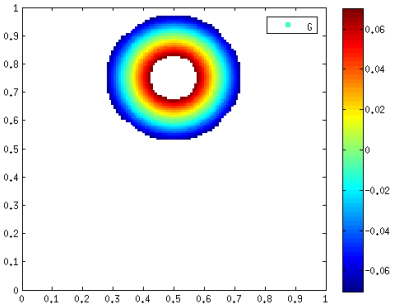


(h) $t = 7.0s$

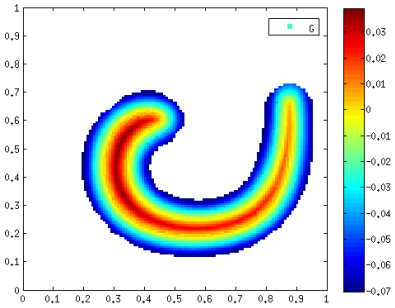


(i) $t = 8.0s$

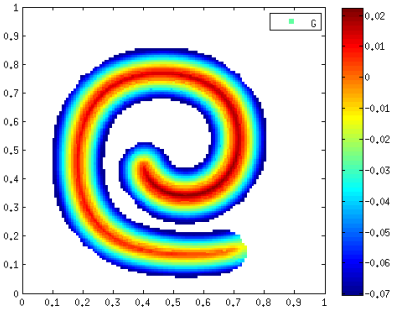
Figure 3.16: Zero Level Set for Circle in Deformation Field With $h_G = 1/128$



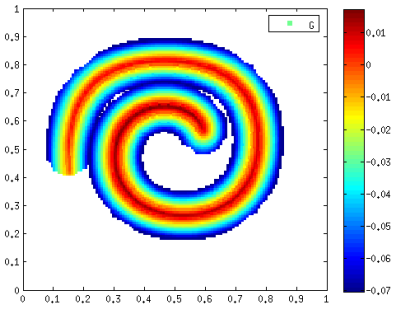
(a) $t = 0.0s$



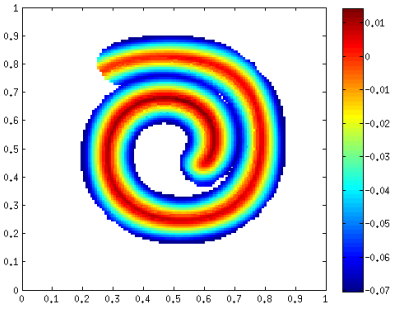
(b) $t = 1.0s$



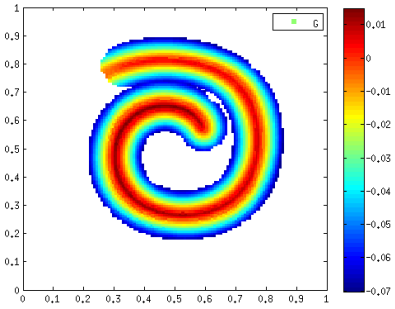
(c) $t = 2.0s$



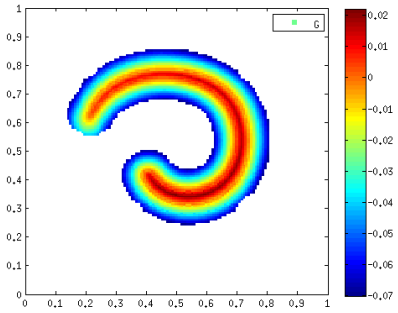
(d) $t = 3.0s$



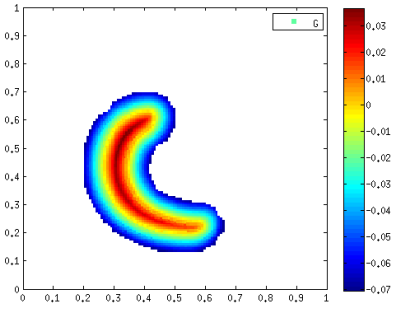
(e) $t = 4.0s$



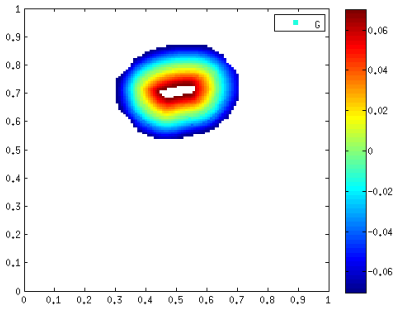
(f) $t = 5.0s$



(g) $t = 6.0s$

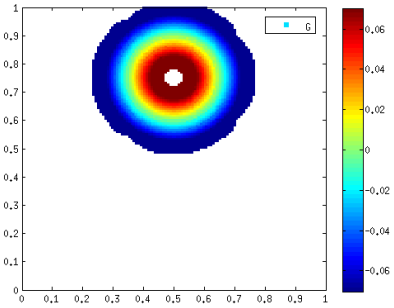


(h) $t = 7.0s$

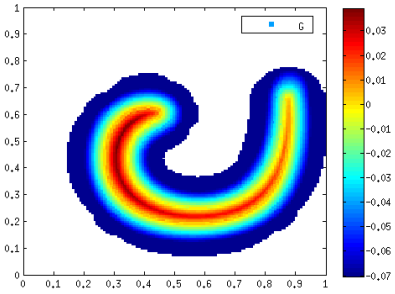


(i) $t = 8.0s$

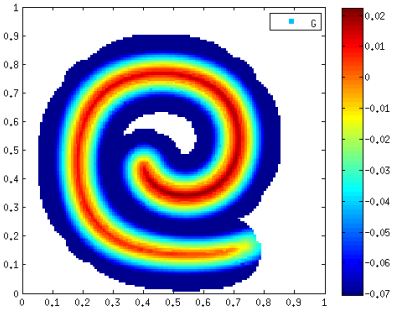
Figure 3.17: Level Set Values in "T" Band for Circle in Deformation Field With $h_G = 1/128$



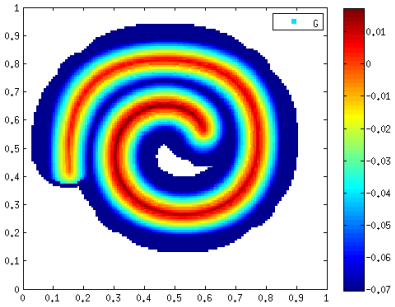
(a) $t = 0.0s$



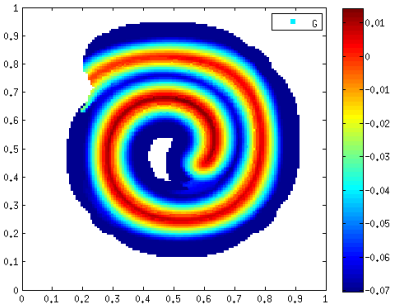
(b) $t = 1.0s$



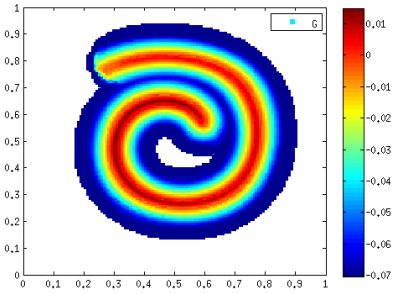
(c) $t = 2.0s$



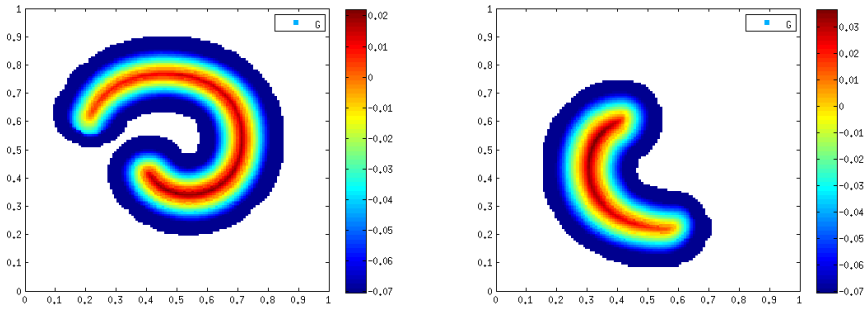
(d) $t = 3.0s$



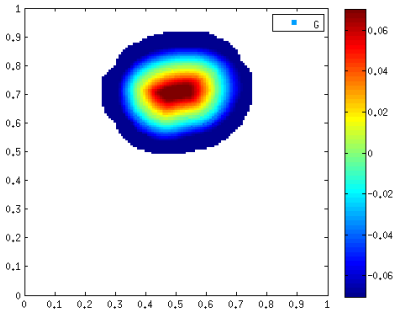
(e) $t = 4.0s$



(f) $t = 5.0s$



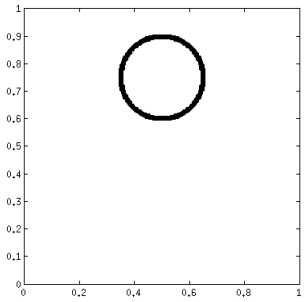
(g) $t = 6.0s$ (h) $t = 7.0s$



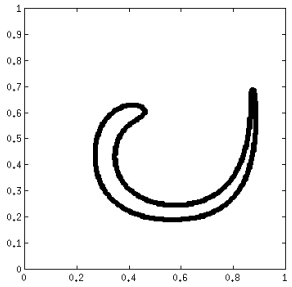
(i) $t = 8.0s$

Figure 3.18: Level Set Values in "X" Band for Circle in Deformation Field With $h_G = 1/128$

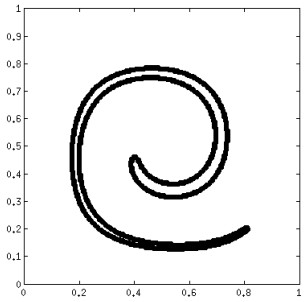
Figures 3.19, 3.20 and 3.21 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/256$.



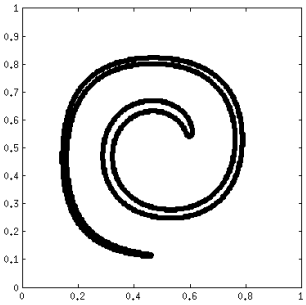
(a) $t = 0.0s$



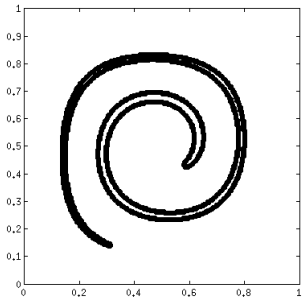
(b) $t = 1.0s$



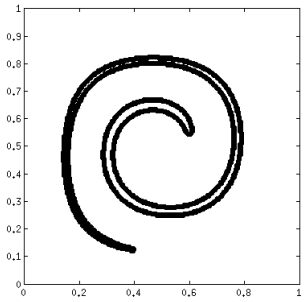
(c) $t = 2.0s$



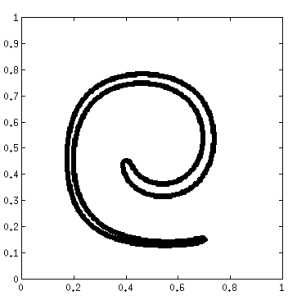
(d) $t = 3.0s$



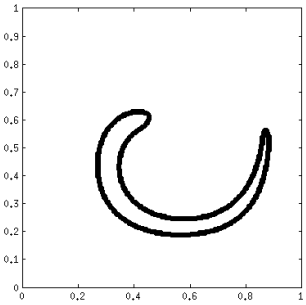
(e) $t = 4.0s$



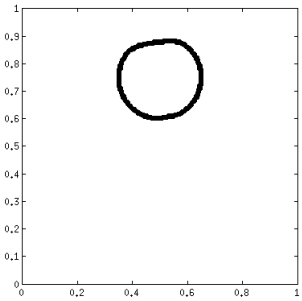
(f) $t = 5.0s$



(g) $t = 6.0s$

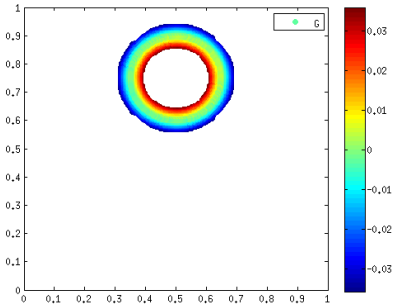


(h) $t = 7.0s$

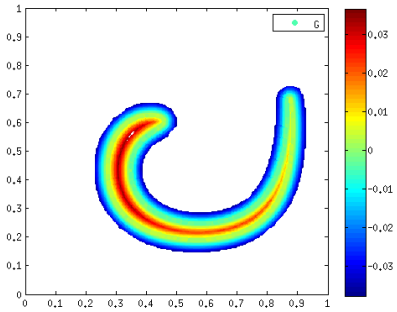


(i) $t = 8.0s$

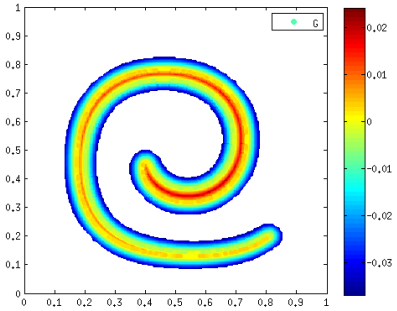
Figure 3.19: Zero Level Set for Circle in Deformation Field With $h_G = 1/256$



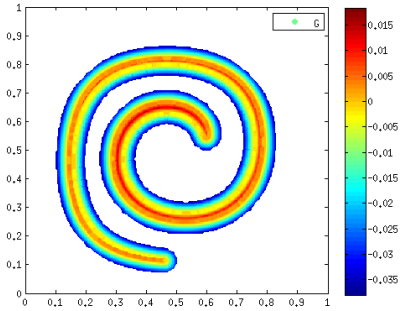
(a) $t = 0.0s$



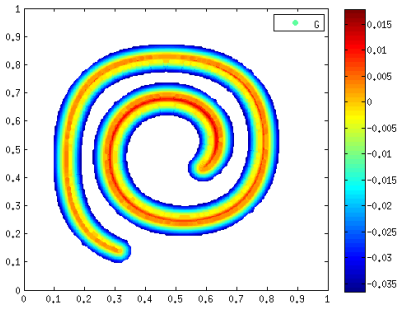
(b) $t = 1.0s$



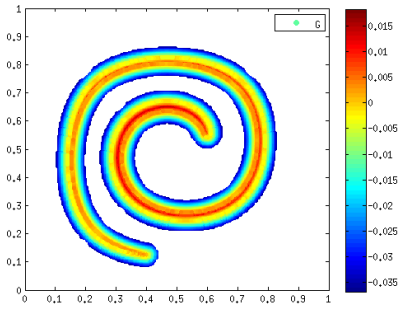
(c) $t = 2.0s$



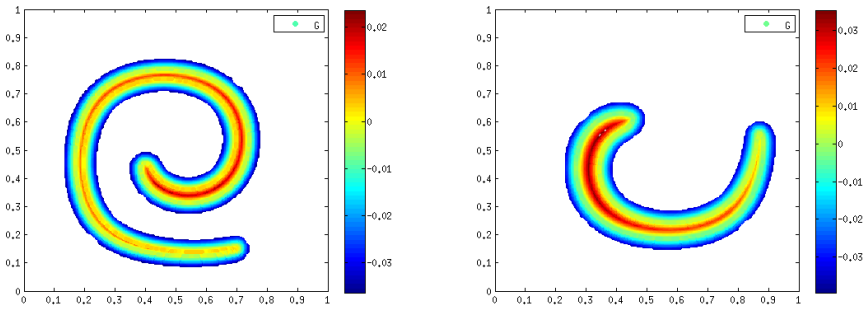
(d) $t = 3.0s$



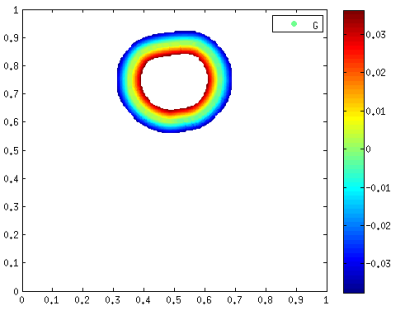
(e) $t = 4.0s$



(f) $t = 5.0s$

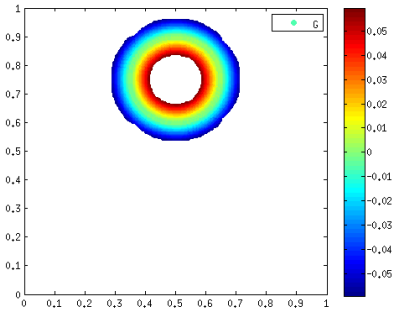


(g) $t = 6.0s$ (h) $t = 7.0s$

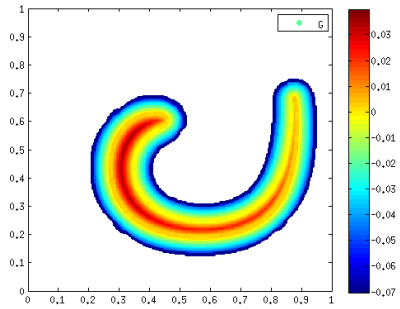


(i) $t = 8.0s$

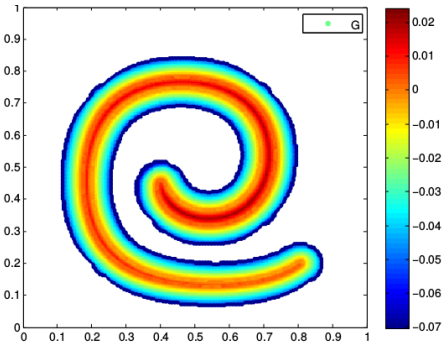
Figure 3.20: Level Set Values in "T" Band for Circle in Deformation Field With $h_G = 1/256$



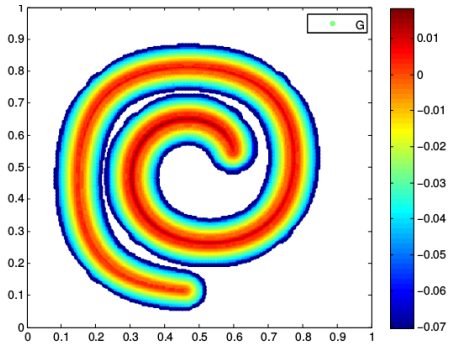
(a) $t = 0.0s$



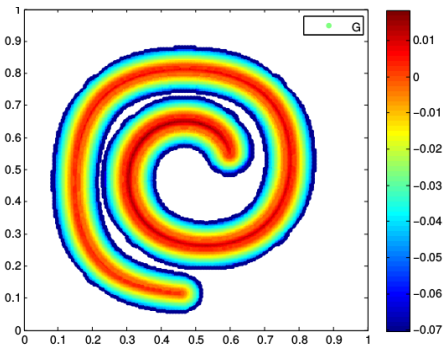
(b) $t = 1.0s$



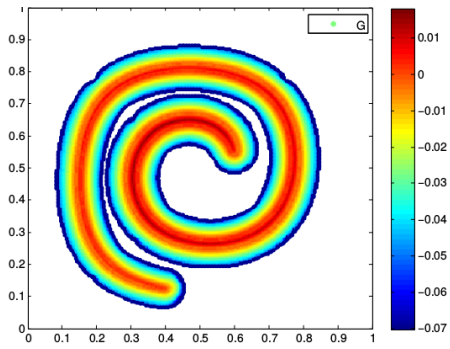
(c) $t = 2.0s$



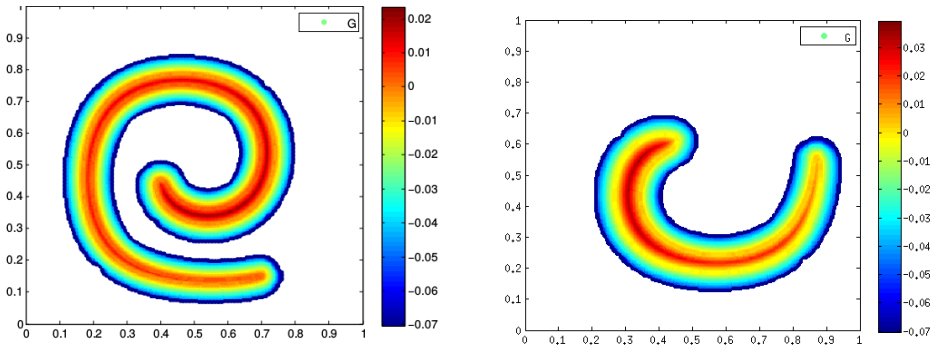
(d) $t = 3.0s$



(e) $t = 4.0s$

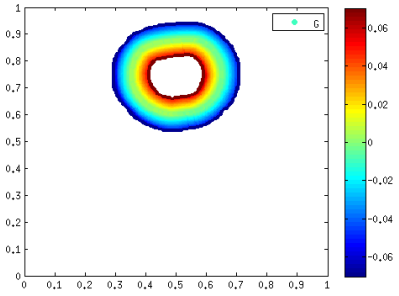


(f) $t = 5.0s$



(g) $t = 6.0s$

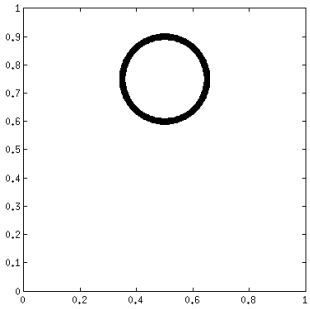
(h) $t = 7.0s$



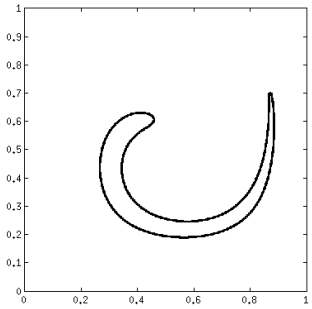
(i) $t = 8.0s$

Figure 3.21: Level Set Values in "X" Band for Circle in Deformation Field With $h_G = 1/256$

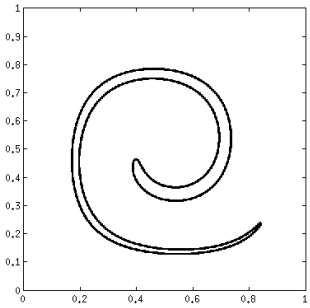
Figures 3.22, 3.23 and 3.24 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/512$.



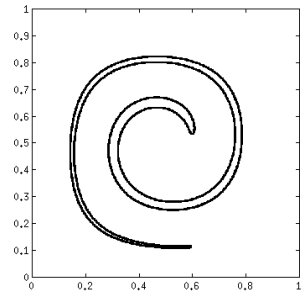
(a) $t = 0.0s$



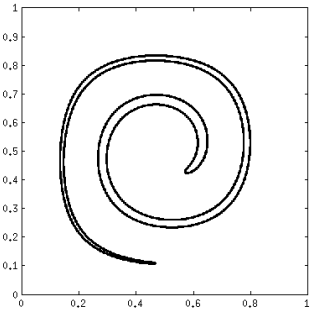
(b) $t = 1.0s$



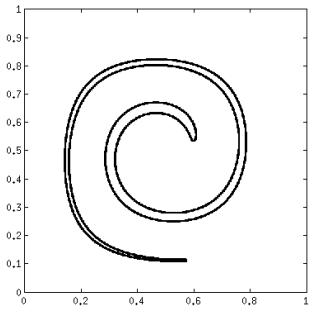
(c) $t = 2.0s$



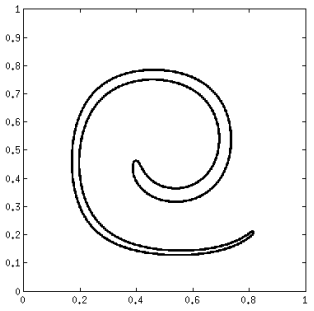
(d) $t = 3.0s$



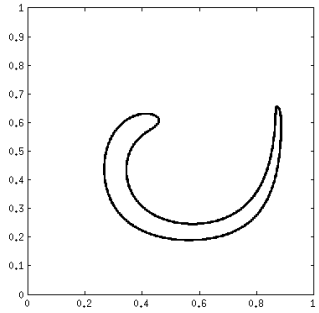
(e) $t = 4.0s$



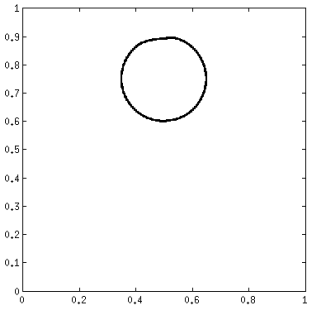
(f) $t = 5.0s$



(g) $t = 6.0s$

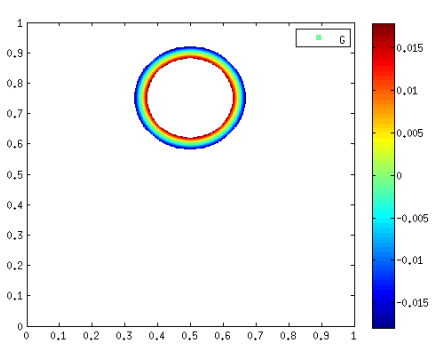


(h) $t = 7.0s$

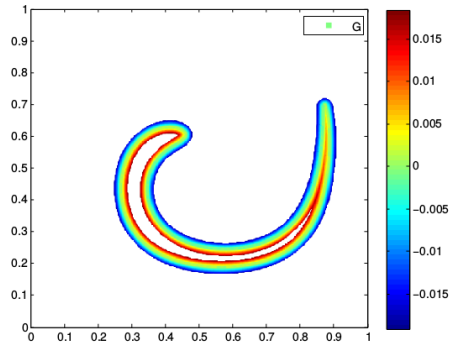


(i) $t = 8.0s$

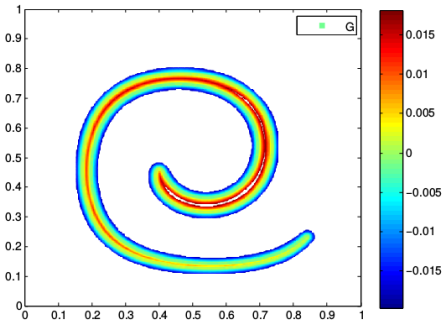
Figure 3.22: Zero Level Set for Circle in Deformation Field With $h_G = 1/512$



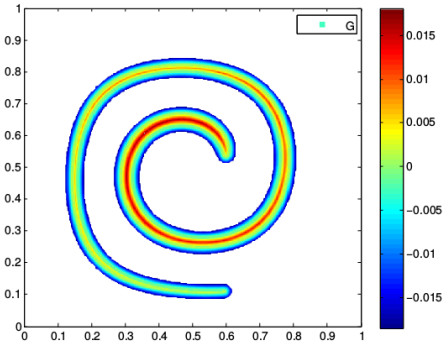
(a) $t = 0.0s$



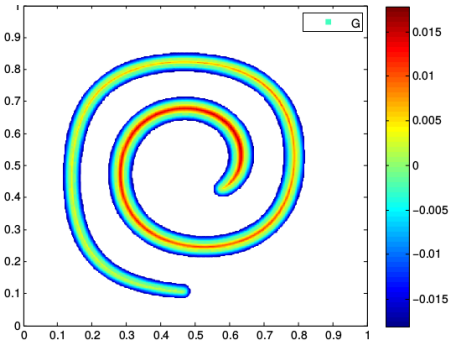
(b) $t = 1.0s$



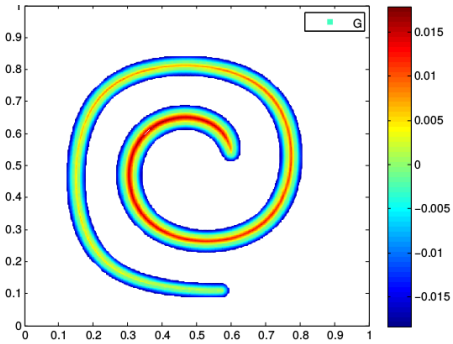
(c) $t = 2.0s$



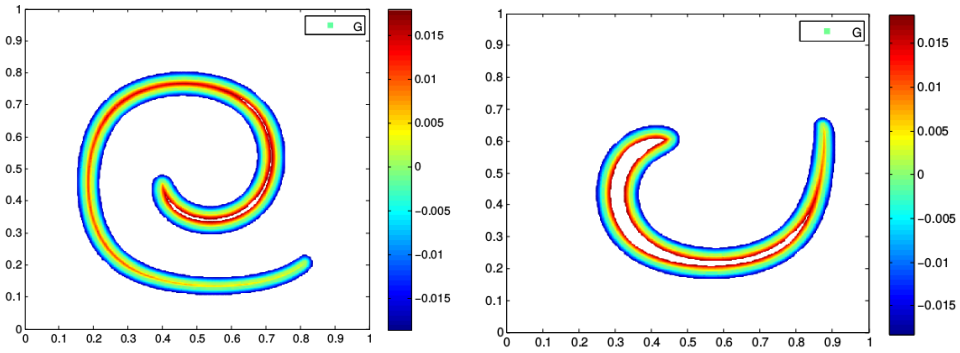
(d) $t = 3.0s$



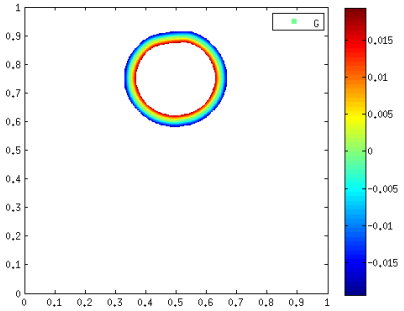
(e) $t = 4.0s$



(f) $t = 5.0s$

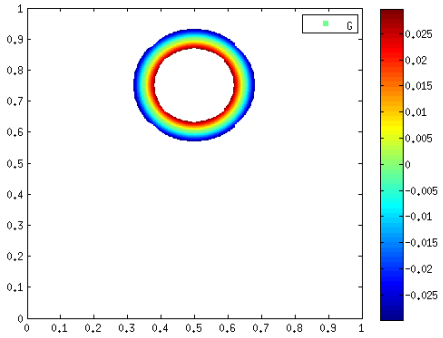


(g) $t = 6.0s$ (h) $t = 7.0s$

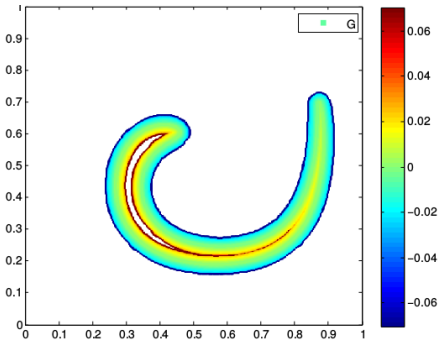


(i) $t = 8.0s$

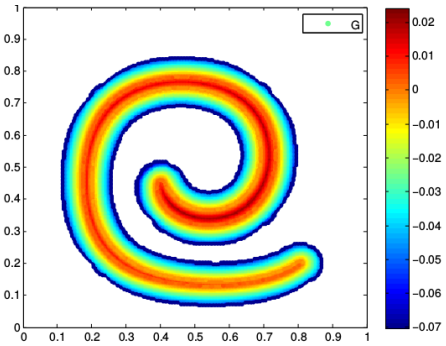
Figure 3.23: Level Set Values in "T" Band for Circle in Deformation Field With $h_G = 1/512$



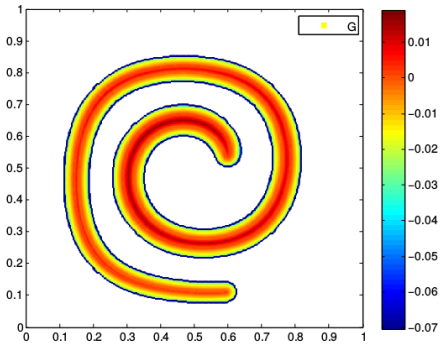
(a) $t = 0.0s$



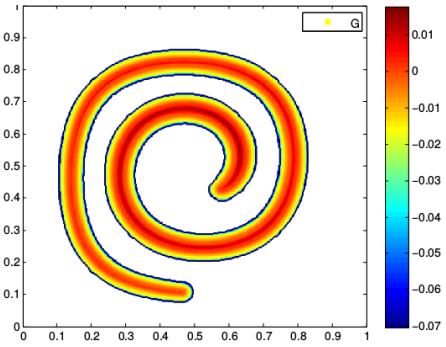
(b) $t = 1.0s$



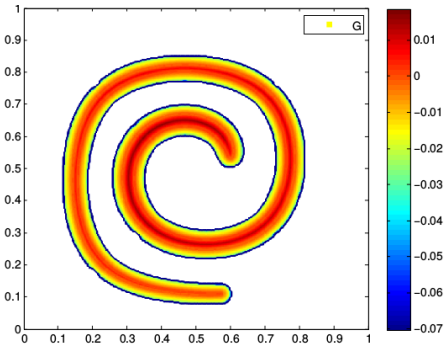
(c) $t = 2.0s$



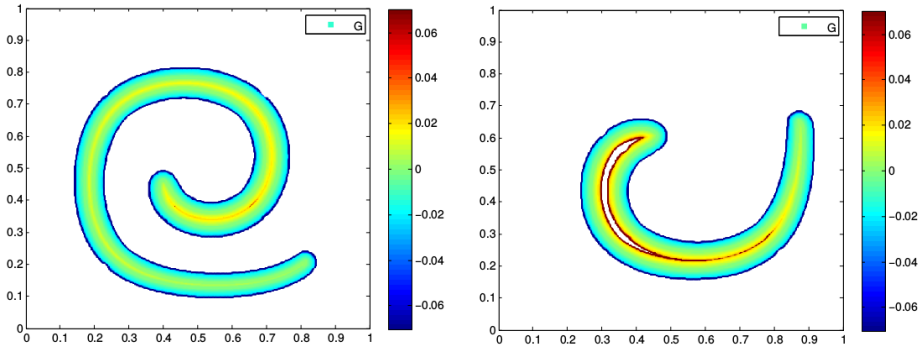
(d) $t = 3.0s$



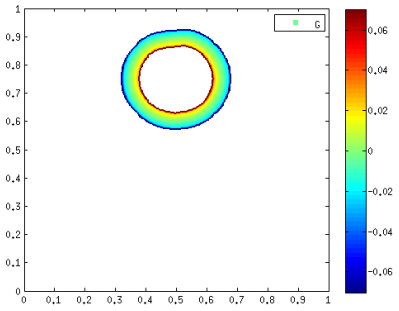
(e) $t = 4.0s$



(f) $t = 5.0s$



(g) $t = 6.0s$ (h) $t = 7.0s$



(i) $t = 8.0s$

Figure 3.24: Level Set Values in "X" Band for Circle in Deformation Field With $h_G = 1/512$

3.5 Plane in A Deformation Field

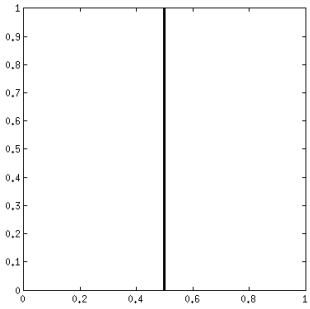
In this section the code is teseted for a plane in a deformation field. A vecrtical plane at $(x, y) = (0.5, 0.5)$ is placed inside a 1×1 box. The velocity field is :

$$\mathbf{u}(\mathbf{x}, t) = -2.0 sin^2(\pi x) sin(\pi y) cos(\pi y) cos(\pi t/T)$$

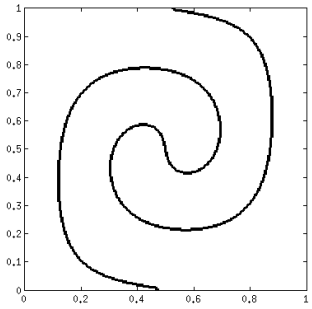
$$\mathbf{v}(\mathbf{x}, t) = -2.0 sin^2(\pi y) sin(\pi x) cos(\pi x) cos(\pi t/T)$$

The time step is $\Delta t = 1/256$ and $h_G = 1/256$.

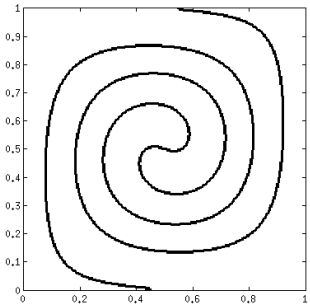
Figures 3.25, 3.26 and 3.27 show the results for interface(zero level set), "T" band and "X" band, respectively with $h_G = 1/256$.



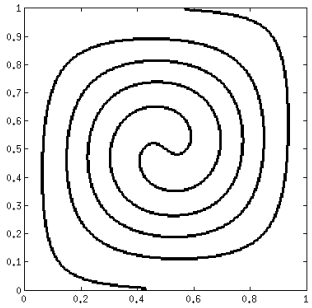
(a) $t = 0.0s$



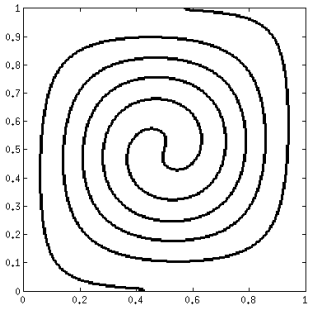
(b) $t = 1.0s$



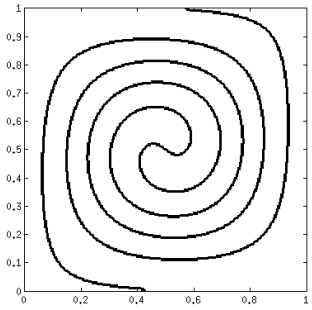
(c) $t = 2.0s$



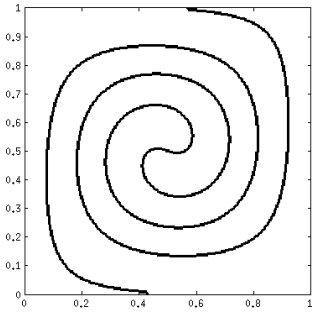
(d) $t = 3.0s$



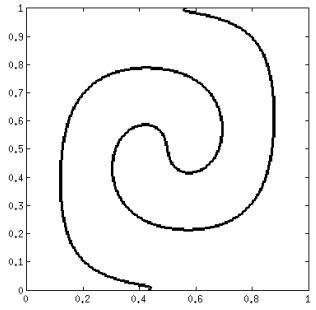
(e) $t = 4.0s$



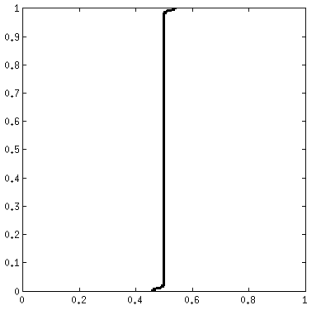
(f) $t = 5.0s$



(g) $t = 6.0s$

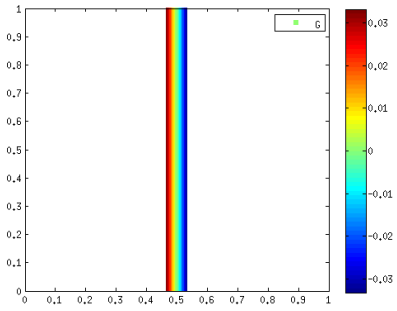


(h) $t = 7.0s$

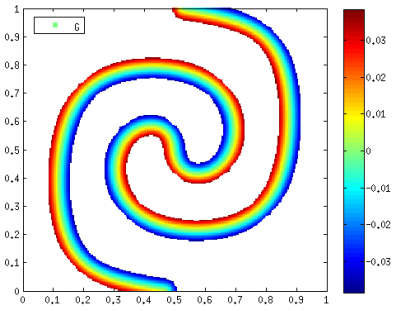


(i) $t = 8.0s$

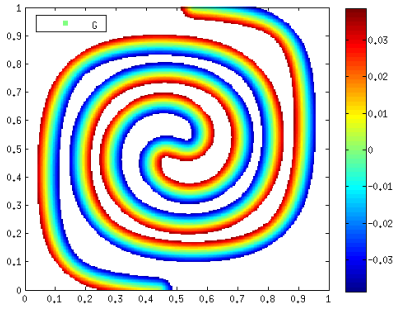
Figure 3.25: Zero Level Set for Plane in Deformation Field With $h_G = 1/256$



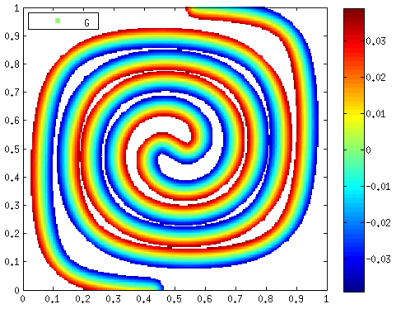
(a) $t = 0.0s$



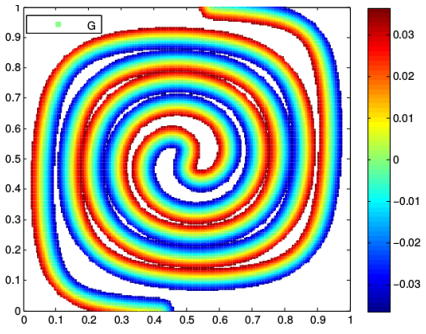
(b) $t = 1.0s$



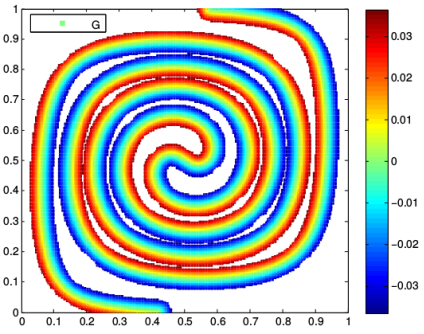
(c) $t = 2.0s$



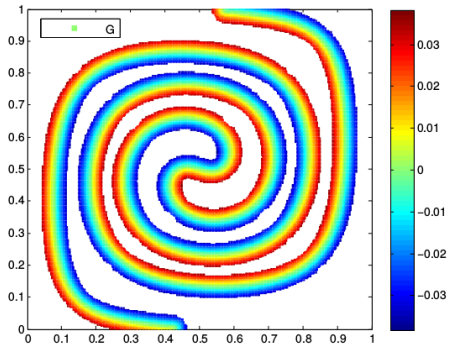
(d) $t = 3.0s$



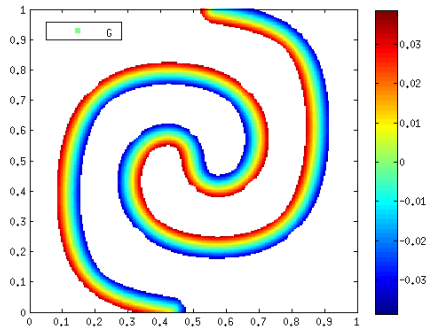
(e) $t = 4.0s$



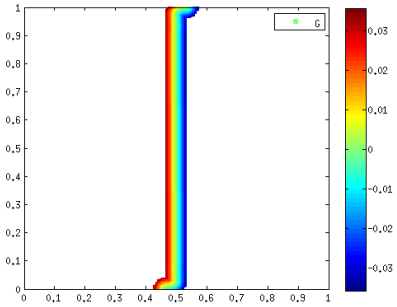
(f) $t = 5.0s$



(g) $t = 6.0s$

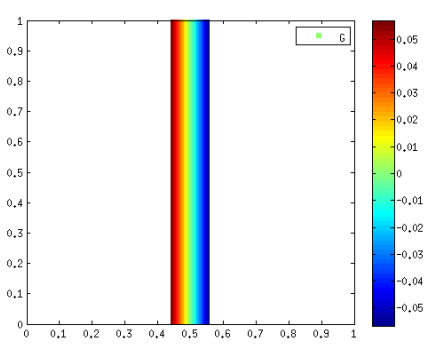


(h) $t = 7.0s$

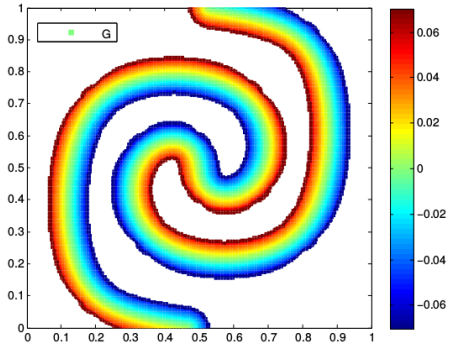


(i) $t = 8.0s$

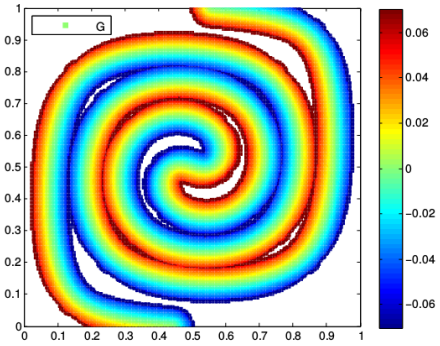
Figure 3.26: Level Set Values in "T" Band for Plane in Deformation Field With $h_G = 1/256$



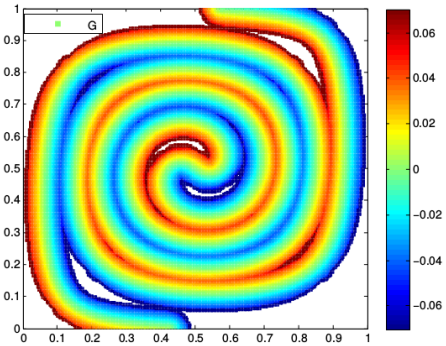
(a) $t = 0.0s$



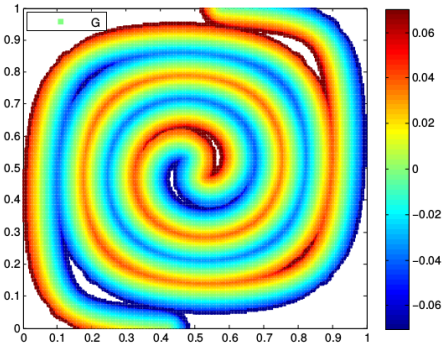
(b) $t = 1.0s$



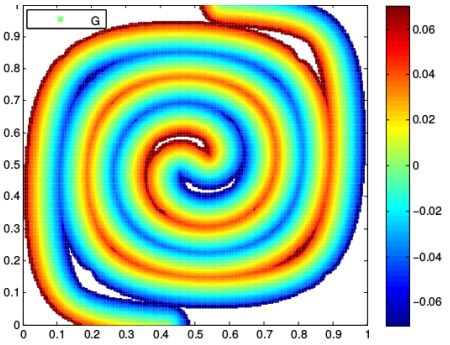
(c) $t = 2.0s$



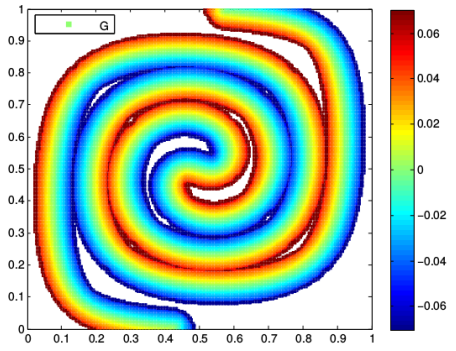
(d) $t = 3.0s$



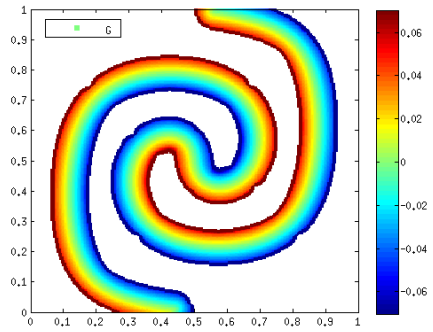
(e) $t = 4.0s$



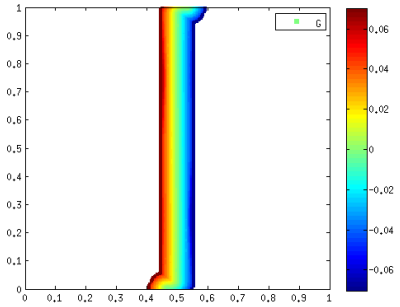
(f) $t = 5.0s$



(g) $t = 6.0s$



(h) $t = 7.0s$



(i) $t = 8.0s$

Figure 3.27: Level Set Values in "X" Band for Plane in Deformation Field With $h_G = 1/256$

Chapter 4

CONCLUSION

In this thesis 33,000 lines of FORTRN code were rewritten in C++. The C++ code was tested for different test cases such as veritical and hoizontal plane in a pure horizontal and vertical velocity field, respectively, Zalesak's disk in a rotational velocity field and circle in a deformation field. The code is fast and the results are exactly matched with FORTRAN code.

There were several challenges in this work. The C++ code was written in object-oriented form. To do this, first the variables and functions in FORTRAN difined for a particular purpose, are identified. Then these variables and fnctions are gathered in particular classes. Thus, in C++ code we work with user defined objects rather than variables. This helps to hide the unncessary data and modify the code without changing a larg part of that. Moreover, the C++ code can be easily combined with the other C++ codes such as flow solver code.

Working with multi-dimensional arrays in FORTRAN is more convenient. The C++ standard tempelate library such as containers and smart pointers which make the memory handling easier, are used. In FORTRAN the lower limit of the arrays can be specified as any integer value. For instance, the first index of an array can be -11 or 9. In C++, arrays always starts from zero. In FORTRAN code these types of arrays are used frequently. Recognizing and shifting the arrays' lower band was a challeng and caused many bugs in the code. By overloading "[]" operator the C++ arrays can have a similar behaviour as FORTRAN. However, it is not efficient and the arrays indexes should be shifted by subtracting the lower band. The MPI library functions are different for FORTRAN and C++. The MPI functions in C++ usually

get the pointers as their arguments. While in FORTRAN the arguments are direcly passed to the MPI functions by value. In C++ working with the strings are better and parsing the input file is easier.

For the future work, speed-up and scalability test will be implemented for the C++ code to study the efficiency and ablitiy of the code in parallel. The results will be compared to the FORTRAN code. The code will be also implemented on the GPU.

Chevalier, C. and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering", Parallel Computing **34**, 6, 318–331 (2008).

Chopp, "Numerical methods for moving interfaces", Lecture Notes (2012).

Chopp, D. L., "Computing minimal surfaces via level set curvature flow", Journal of Computational Physics **106**, 1, 77–91 (1993).

Cormen, T. H., C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to algorithms* (2009).

Harten, A., B. Engquist, S. Osher and S. R. Chakravarthy, "Uniformly high order accurate essentially non-oscillatory schemes, iii", Journal of computational physics **71**, 2, 231–303 (1987).

Herrmann, M., "A balanced force refined level set grid method for two-phase flows on unstructured flow solver grids", Journal of Computational Physics **227**, 4, 2674–2706 (2008).

Jiang, G.-S. and D. Peng, "Weighted eno schemes for hamilton–jacobi equations", SIAM Journal on Scientific computing **21**, 6, 2126–2143 (2000).

Peng, D., B. Merriman, S. Osher, H. Zhao and M. Kang, "A pde-based fast local level set method", Journal of Computational Physics **155**, 2, 410–438 (1999).

Sethian, J. A., "Fast marching methods", SIAM review **41**, 2, 199–235 (1999a).

Sethian, J. A., *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*, vol. 3 (Cambridge university press, 1999b).

Shu, C.-W., "Total-variation-diminishing time discretizations", SIAM Journal on Scientific and Statistical Computing **9**, 6, 1073–1084 (1988).

Stroustrup, *THE C++ PROGRAMMING LANGUAGE*, vol. 4 (Cambridge university press, 2013).

Sussman, M., P. Smereka and S. Osher, "A level set approach for computing solutions to incompressible two-phase flow", Journal of Computational physics **114**, 1, 146–159 (1994).