Three Dimensional Computer Model of Dendrite Growth in Tertiary Al-Cu-Si Alloys

by

Marcias Martinez

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

(Master of Engineering)

Department of Mechanical and Aerospace Engineering

Carleton University

Ottawa, Ontario

September 2nd, 1999

Abstract

Form casting, precision casting, soldering, welding are all manufacturing processes that involve solidification. In casting, solid nuclei appear in the liquid, leading to the formation of columnar or dendritic (tree-like structures) crystal structures. The type of microstructures that forms, whether dendritic or columnar, will influence such important material properties as strength and toughness. In this paper a computer simulation of a dendrite in three dimensions (3-D) based on the cellular model is described. Our model assumes constant temperature and the dendrite develops due to concentration gradients. The simulation is carried out for Al-Si-Cu alloys. Concentration profiles in three dimensions are obtained from the transport equation of solute diffusion and changes in concentration due to phase field changes. Concentration at each cell with a liquid fraction is calculated for every time step. The interface velocity is calculated from the curvature at the interface, concentrations of each alloying component, undercooling and a kinetic coefficient constant. The time step and the velocity provide a new position of the solid-liquid interface for each iteration. In this model, curvature at each cell is calculated using an averaged phase field. The average phase field in 3-D is obtained by multiplying the solid fraction of each cell and its neighbors by weight factors. This model takes anisotropy effects into account in order to avoid splitting of the dendrite tips. The data obtained from the model allow us to analyze the dendrite morphology and parameters such as tip radius, spacing between secondary arms, growth velocity and their dependence on

undercooling. It is expected that the dendrite tip will grow with a stable parabola-like shape while the growth at the sides of the dendrite will be unstable. This instability at the solid liquid interface can result in the growth of secondary arms. The results are compared with analytical model data.

Acknowledgement

I would like to thank God for giving me the strength to finish this thesis. I would also like to thank my wife Lisbet and my daughter Vaythiare, for all the encouragement and strength they provided me during this work. Special thanks to my mother Emirva, because this thesis is as much hers as it is mine. I would like to dedicate this thesis to my father Marcias Martinez who is probably the only one in my family who understands this work. I would also like to dedicate it to my sister Marianela, my brother law Graeme, my nieces Kimberley and Cheryl Ann, and my friend Gratien who have no idea what a dendrite is, but are very proud or me anyway.

A very special thanks to my supervisor and friend Dr. Andrei Artemev and his wife Natasha Artemeva who have supported me in all my career decisions. Their dedication to their work and to the formation of engineers is an example of what all professors and mentors should be like. I think I would not exaggerate by saying that words can not describe the gratitude that I have toward you guys. Thanks Andrei and Natasha for your unconditional support.

Contents

List of Figures

List of Symbols

T_m	Melting Temperature of the host component
Δg	Gibbs energy per unit volume between the solid and liquid phase
ΔT	Undercooling (K)
Δh	Specific enthalpy of the solid and liquid
σ	Surface energy per unit area
"r"	Radii of the nuclei
c	Enthalpy of the solid and liquid
V	Velocity at the interface (m/sec)
μk	Kinetic coefficient of the alloy (m/sK)
C_{liq_1}	Liquid concentration of the first alloying element (%wt)
m_{l_1}	Liquidus slope on the phase diagram of the first alloying element (K/%wt)
C_{liq_2}	Liquid concentration of the second alloying element (%wt)
m_{l_2}	Liquidus slope on the phase diagram of the second alloying element (K/%wt)
Γ	Gibbs Thomson coefficient (Km)
k	Curvature of the solid-liquid interface (m^{-1})
D	Diffusivity constant
dCx/dx	Concentration gradient in the x direction
dCx/dt	Change of the concentration in time

N	Excess solute due to the solid-liquid interface movement
Δf^{l}_{ijk}	Change in the phase field
$C^{l}_{liq\_ijk}$	Liquid concentration in cell i,j,k at the current time step
$C^{l}_{sol\_ijk}$	Solid concentration in cell i,j,k at the current time step
Δx, Δy, Δz	Grid size in x, y and z coordinate direction.
∇g	Divergence of the solid-liquid interface
$\lvert\nabla g\rvert$	Absolute value of the divergence of the solid-liquid interface
φ	Phase field function
Ω	Region containing the phase field function
$d\Omega$	Boundaries of the phase field function.
ε	Phase field function constant
W	Phase field function constant (Joules/m^3)
$\beta(T)$	Monotonic increasing function of temperature
T	Temperature of the system
R	Universal gas constant and
v_m	Molar volume (assumed to be constant)
L	Differential operator
$\Im$	Helmholtz free energy
n	Normal to the boundary ($\partial\Omega$)
Δt	Time step
l	Time enumerator
i, j, k	Cell numbers along the x, y and coordinate system
S	Area of the interface boundary in a cell

f_{ijk}	Phase field value in cell i,j,k
Δf_{ijk}	Change in the phase field value per time step
C_{ijk}^{l+1}	New concentration in cell at the next time step
C_{ijk}^{l}	Concentration in that cell for the previous time step
Δf_{ijk}	Change in the phase field
θ	Step function
g	Averaged phase field
∇	Divergence operator
g_x	Partial derivative of the average phase field "g" with respect to x
g_y	Partial derivative of the average phase field "g" with respect to y
g_z	Partial derivative of the average phase field "g" with respect to z
g_{xx}	Partial derivative of the average phase field "g" with respect to x
g_{yy}	Partial derivative of the average phase field "g" with respect to y
g_{zz}	Partial derivative of the average phase field "g" with respect to x and z
g_{xy}	Partial derivative of the average phase field "g" with respect to x and y
g_{yz}	Partial derivative of the average phase field "g" with respect to y and z
α	Thermal or mass diffuisivity
τ	Time step
V_s	Velocity of sound in the liquid alloy
L	Latent heat of the alloy
g_x	Gradient of the average phase field in x
g_y	Gradient of the average phase field in y and
g_z	Gradient of the average phase field in z

phi (φ)	Angle between the normal to the interface and the z coordinate direction
theta (θ)	Angle between the normal to the interface and the x coordinate direction
arccos	Mathematical function that returns the angle in radians.
cos	Mathematical cosine function
k_{ijk}	Solid-liquid interface curvature for cell i,j,k.
C_{si_eff}	Effective concentration of Silicon
C_{si_liq}	Liquid concentration at the current temperature of the system
Ω	Effective Supersaturation
C_{bulk_Si}	Concentration of silicon at time zero far away from the solidifying solid-liquid interface
C_{bulk_cu}	Concentration of copper far away from the solid-liquid interface at time step zero
$mgap\_{cu}$	Copper miscibility gap at the current system temperature.
$mgap\_{si}$	Silicon miscibility gap at the current system temperature.
r_{tip}	Tip radius (meters)

Chapter 1: Introduction

1. Objective of the Thesis

The processing and manufacturing of most metal components are affected by solidification. During the initial stages of manufacturing of any metallic product, most metals are transformed from a liquid to a solid phase. The transformation of metal from liquid to solid phase is known as solidification. Solidification influences such processes as casting, welding, soldering, rapid solidification processing and directional solidification; thus solidification influences directly or indirectly every metallic product today. Solidification defects tend to remain during subsequent processes; therefore good control and understanding of the formation of the solid phase is of great importance. How the liquid metal is solidified determines the microstructure of the material and as a result the properties of the material. The property of any metal is dependent on parameters such as, for example, grain size, grain shape and solute composition. During solidification the solid phase forms complex patterns that determine the morphology of the grain. One of these complex patterns is a tree-like structure known as a dendrite.

The dendrite morphology develops as a tree-like structure with a primary trunk and secondary and tertiary branches. The external environment surrounding the dendrite, such as, for example, the temperature and concentration gradients around the dendrite influences this tree-like structure [1]. The dendrite morphology is also influenced by internal parameters such solid-liquid interface curvature, dendrite tip radius, and

secondary inter-dendritic arm spacing. It is the objective of this thesis to explain in detail a computer model that simulates the dendrite morphology, from which it was possible to obtain approximations on dendritic parameters such as dendrite tip radius and secondary inter-dendritic arm spacing.

In the next section, we present the layout of this thesis.

1.1. The Layout of The Thesis

This thesis is composed of the following chapters:

- Chapter 1, Introduction

  Chapter 1 outlines the importance of dendrite morphology and the objective and layout of the thesis.

- Chapter 2, Literature review

  Chapter 2 outlines the theory behind the solidification phenomenon. It discusses the parameters that affect the solidification process and analyses the importance of the process in manufacturing of materials.
  The literature review presented in this chapter is based on publications of the past 10 years (1989-1999). The first sections of this chapter are a summary of the process that affects dendrite growth and morphology. This chapter also presents the mathematics that describes dendrite growth.

- Chapter 3, Cellular model of dendrite growth

  Chapter 3 describes the cellular model and how it was used to simulate the growth of a dendrite in three dimension (3-D) for a tertiary alloy. This chapter presents all the parameters that are taken into account by the model and how the mathematical equations used by our model are numerically approximated using a finite difference scheme.

- Chapter 4, Results

  Chapter 4 discusses the results obtained from our model such as dendrite tip radius and growth velocity. Results on tests that validate the model such as the conservation of mass test are also presented in this chapter. The results of the model are then compared to the results presented by a parabolic approximation of the dendrite tip presented by Fisher and Kurz's model [1]

- Chapter 5, Conclusion and future work

  Chapter 5 provides conclusions on our model and discusses possible changes to the model that could improve its accuracy and performance.

Chapter 2: Literature Review

2.0 Introduction

This chapter outlines the theory of solidification phenomenon and dendrite growth. It discusses the parameters that affect the solidification process and analyses the importance of the process in manufacturing of materials. Later sections of this chapter present the mathematics that describes dendrite growth. The literature review presented in this chapter is based on publications of the past ten years (1989-1999).

2.1 Importance of solidification and dendrites

Due to the advances in computer technology, engineers and scientists in the field of computational metallurgy are now able to simulate dendrite growth in industrial alloys. This simulation allows us to approximate processes that occur in metals, e.g., the metals used in the aerospace and automotive industry. Solidification occurs in natural processes such as the freezing of lakes during the wintertime in some countries. Solidification is encountered in material processing as, for example, casting and directional solidification. When a liquid metal solidifies, it often forms a complex pattern of branch structures, similar to trees. Such tree-like structures are called dendrite [1]. The understanding of the dendrite morphology provides us with an insight into such parameters as tip radius,

distance between secondary arms and concentration distribution of solute. These parameters affect such important material properties as material strength and toughness. The knowledge of the dendrite morphology and its effects on material properties will allow material manufacturers to produce higher quality material products. The purpose of this thesis is to simulate the morphology of a dendrite in three dimensions (3-D), for three component alloys at constant temperature.

When a metal is solidified several processes occurs simultaneously. These processes are:

- Nuclei formation,
- Crystal growth,
- Heat transfer by heat diffusion, by convection,
- Mass transfer.

All these processes occur at the same time and interact making the analysis of the solidification phenomenon complicated. The first requirement for the formation of a crystal during solidification is the formation of a nuclei. The nuclei can then grow to form a crystal or it can re-melt and disappear depending on the nucleus radius. The formation of the crystal then leads to the formation of the grain in the metal, thus the importance in understanding crystal growth. An example of a dendrite is shown in figure 2-0.

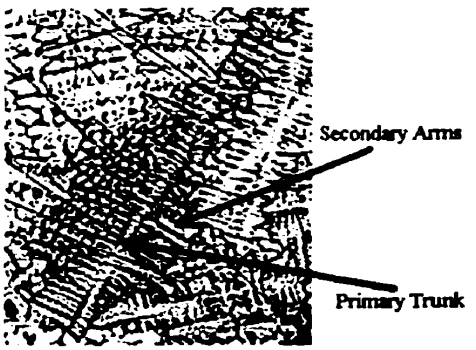


Figure 2-0: Dendrite Structure

Dendrite morphology determines the final microstructure of the material and thus it effects such material properties as Ultimate Tensile Strength (σ_{UTS}), ductility and crack resistance among others. These properties are critical and determine the usability that a material is given in industrial applications, if the material is too weak or toughness is too low then it is probable that the material will have very little if any use in real life applications.

In our model, mass transfer due to diffusion is the main driving force for the formation and growth of a dendrite. In the following chapters, it will become obvious that the computer time required to solve a three-dimensional model based on concentration gradients for a tertiary system is computationally intensive. In order to avoid the simulation of the heat transfer process our model assumes constant temperature. Analyzing a dendrite morphology based on heat diffusion would require a time step that is approximately one thousand times smaller than that for an analysis based on mass transport, for the same grid size. This would mean that a computer would require doing one thousand iteration in the mass transport domain for each iteration in the heat transfer domain. A smaller time step would require approximately 31 more cells in each coordinate direction (x, y and z) in order to solve the same problem, which would require more computing time.

For a dendrite to grow it is essential for a nucleus to exist in the liquid metal. Our simulation begins once the nucleus is developed. In the following pages, we will discuss very briefly the processes that take place for the formation of nuclei and the mathematics that describes these processes.

2.2 *Nucleation Theory*

Nucleation is the first stage of solidification. Nucleation is the formation of minute crystalline regions within the liquid metal. These minute crystalline regions are called nuclei, embryos or clusters. They are formed due to random fluctuations of patterns of atoms within the liquid metal. Atoms from the liquid metal can attach to these crystalline regions. Small crystalline particles can be formed even above the melting temperature of the liquid metal, although they are unstable. Thus, for the formation of a stable nucleus it is required that the temperature of the liquid metal is below its melting temperature [1]. The difference between the actual temperature of the metal and the equilibrium melting temperature of the system is known as supercooling or undercooling. Undercooling can be related to the driving force of transformation, Δg:

$$\Delta g = \frac{\Delta T \cdot \Delta h}{T_m} \quad (2\text{-}1)$$

where

- T_m is the melting temperature of the host component,
- Δg is the difference in Gibbs energy per unit volume between the solid and liquid phase
- ΔT is the undercooling and
- Δh is the difference in specific enthalpy of the solid and liquid [2].

The formation of a solid nuclei in the liquid phase leads to the variation of the Gibbs energy (ΔG). This variation of the Gibbs energy is described by equation 2-2 (assuming a spherical nucleus):

$$\Delta G = \sigma \cdot 4 \cdot \pi \cdot r^2 + \frac{\Delta g \cdot 4 \cdot \pi \cdot r^3}{3} \qquad (2\text{-}2)$$

where

- Δg is the difference in the Gibbs energy per unit volume between the solid and liquid phase
- σ is the surface energy per unit area
- "r" is the radii of the nuclei [1].

The first term of equation 2-2 represents the contribution of the surface energy while the second term represents the volume contribution of the Gibbs energy per unit volume between the solid and liquid phase. The second term of equation 2-2 is negative if the temperature of the system is below the melting temperature of the alloy and positive if vice versa. From equation 2-2, it can be observed that for a small nucleus the contribution of surface energy is greater than the volume contribution. If the temperature of the system is below the melting temperature, then the volume contribution to the Gibbs energy is negative. From this we can conclude that equation 2-2 will have a maximum Gibbs energy (ΔG) at a specific critical radius (r_c).

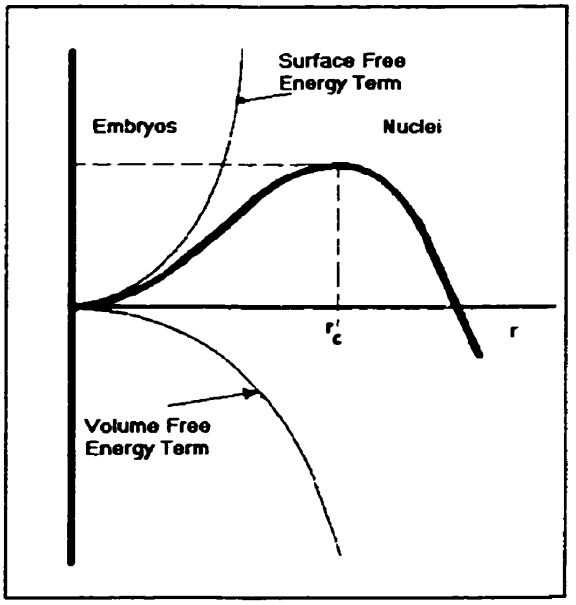


Figure 2-1: Gibbs Energy vs. Nuclei radii. [3]

From Figure 2-1 we can conclude that a nucleus with a radius greater than r_c, will grow. If the radius of the nucleus is below that critical radius then the nucleus will re-melt and disappear.

It is important to note that there are two types of nucleation:

- Homogeneous and
- Heterogeneous.

These two types of nucleation describe the presence or lack of an existent interface at the nucleation site. For example, if the nucleus initiates completely surrounded by the liquid phase, the formation of the nucleus is called homogeneous nucleation. If the nucleus forms on one of the walls of the cast, the formation of the nucleus is known as heterogeneous nucleation. In our model, the growth of the dendrite starts once the nucleus is greater than the critical nucleus. In the next section of this chapter, we will

describe the effects and processes that influence and contribute to the growth of a nucleus to a dendrite.

2.3 Crystal Growth

The second part of the process of the dendrite development and, eventually, the formation of the grain, is growth. The following effects and processes influence the growth of a dendrite:

- Kinetics of the interface (atom rearrangement)
- Heat and mass diffusion
- Capillarity effects.

In the following sub-sections, we will describe each of these processes and effects in more detail.

2.3.1 The kinetics of atom attachment at the solid-liquid Interface

The kinetic of the growth of the solid-liquid interface depends on the probability of the atoms being attached to the interface and the probability of those atoms remaining fixed on the interface. One way to describe the growth of the solid-liquid interface is to assume that atoms have a cubic structure. Any atom would have six different possible orientations (faces) for it to attach to the solid-liquid interface.

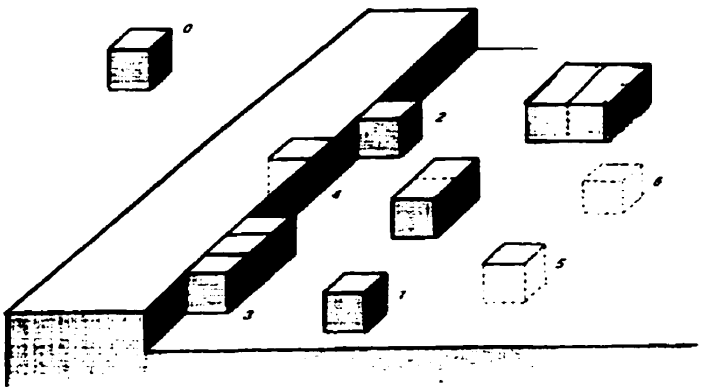


Figure 2-2: Atomic Rearrangement [1]

From figure 2-2, it is possible to see that an atom that is surrounded by liquid has zero atoms in solid phase as neighbors, while an atom that is completely immersed in the crystal has six atoms in solid phase as neighbors. The growth of the solid-liquid interface is determined by the probability of atoms being attached to the interface and the probability of atoms being fully immersed in the crystal, such as the case of atom no. 6 shown in figure 2-2. In the case of a faceted interface, an atom with two neighbors is more prominent to form new rows of atoms. These new rows would grow by addition of atoms of type 3. Once the complete layer is filled, a very high supercooling would be required for the addition of an atom of type 1. Type 1 atoms would constitute a new nucleation site for the formation of another layer. This formation of new layer of atoms constitutes the propagation of the interface. This growth can lead to the formation of two types of interface morphologies. These are:

- Faceted and

- Non Faceted

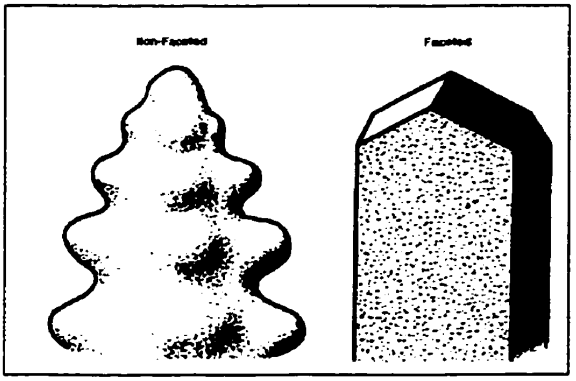


Figure 2-3: Faceted vs. Non Faceted morphology. [1]

A faceted interface can appear jagged at the microscopic level but is usually very smooth and flat at the atomic scale. This type of interface tends to maximize the atomic bonding between the crystal and the atoms on the interface, thus, leaving few sites where atoms arriving at the interface by diffusion from the liquid can attach themselves. The interface morphology is dependent on the growth process since substances that allow a non-faceted interface growth from a melt, also allow a faceted interface growth from a solution or vapor [1].

A non-faceted interface usually has many sites where atoms arriving from the liquid metal can attach. Non-faceted interfaces are most common in metals. This type of interface tends to be rough at the atomic scale.

The velocity at which the interface propagates is dependent on the following parameters:

- Kinetic coefficient,

- Curvature,
- Surface energy of the interface and
- Undercooling.

All these parameters can be calculated. The kinetic coefficient of most alloys is not known and thus it is usually approximated. This approximation is sufficient when the growth takes place at low supercooling, since the effects of curvature tend to have a greater influence on the growth under this condition [4].

The velocity of the solid-liquid interface can be mathematically expressed as:

$$V = \mu_k \cdot (\Delta T + C_{liq\_1} \cdot m_{l\_1} + C_{liq\_2} \cdot m_{l\_2} - \Gamma \cdot k) \qquad (2\text{-}3)$$

where

- V is the velocity at the interface (m/sec)
- μ_k is the kinetic coefficient of the alloy (m/sK)
- ΔT is the difference between the melting temperature of the pure host component and the actual temperature of the system (K)
- C_{liq_1} is the liquid concentration of the first alloying element (%wt)
- m_{l_1} is the liquidus slope on the phase diagram of the first alloying element (K/%wt)
- C_{liq_2} is the liquid concentration of the second alloying element (%wt)

- m_{l_2} is the liquidus slope on the phase diagram of the second alloying element (K/%wt)
- Γ is the Gibbs Thomson coefficient (Km)
- k is the curvature of the solid-liquid interface (m^{-1}) [5]

The sum of all parameters inside the brackets in equation 2-3 represents the kinetic undercooling of the solid-liquid interface. From equation 2-3 it can be observed that the velocity at which the interface moves in space increases as the kinetic undercooling increases. The liquidus slopes of both alloying components are calculated from the phase diagram. The slopes are approximated as straight lines. This approximation is used to simplify equation 2-3 because higher order degree polynomials would not contribute significantly to the accuracy of the model. The liquidus slopes of both alloying components are negative while the concentrations of the alloying components are always positive. Thus, the multiplication of the concentration and the liquidus slope produces a negative term in degree Kelvin. The curvature term of equation 2-3 is calculated at every point of the interface. Curvature is taken to be positive for a convex solid-liquid interface. The value of curvature multiplied by the Gibbs Thomson Coefficient is known as capillarity undercooling. Thus, the total kinetic undercooling is the addition of the capillarity, composition and thermal undercooling. The total kinetic undercooling when multiplied by the kinetic coefficient determines the velocity of the interface. The velocity at every point of the interface when multiplied by the time step determines the new position of the interface. Every point at the interface may have different values of concentration, curvature, undercooling and, therefore, may have different velocities. Therefore, by multiplying the velocities at each point of the interface by the same time

step per iteration, we obtain different positions of the solid-liquid interface at each point. In time, the position of each point on the interface leads to the development of an interface with a complex morphology. This leads to the formation of a dendritic structure [5].

The different values of concentration at every point of the interface are due to the movement of atoms within the solid-liquid interface, within the solidified region and within the liquid region. The movement of atoms is known as mass transport. The next section will describe in more detail how the mass transport occurs and the mathematics that describes this process.

2.3.2 Mass Transport

Diffusion is the movement of atoms in a solid, liquid or gaseous environment from areas of high concentration to areas of low concentration. Mathematically diffusion is described by Fick's First Law of Diffusion, which states that the movement of atoms from one point to another in space is proportional to the concentration difference between these points. Fick's First Law is a steady state equation and, therefore, it is time independent. The mass transport that occurs during solidification is strongly dependent on time. Thus, Fick's Second Law of diffusion describes this type of transport [4]. Mathematically Fick's Second law of diffusion in one dimension (1-D) is expressed as:

$$\frac{dC_x}{dt} = \frac{d}{dx}\left(D \cdot \frac{dC_x}{dx}\right) \quad (2\text{-}4)$$

where

- D is the diffusivity constant
- dCx/dx is the concentration gradient in the x direction
- dCx/dt is the rate of change of the concentration in time.

Equation 2-4 is not complete because it does not take into account the movement of the solid-liquid interface. During the solidification of alloys solute is rejected from the solid region into the liquid region, thus creating a region of excess solute also known as the diffusion boundary layer. This layer is created during a transient period. The solute is rejected because the solid-liquid interface moves in space. During this transient period, the solute can be rejected if the solubility of the solid in the liquid is greater than that in the solid. In other words this rejection occurs if the distribution coefficient ($k=C_{soild}/C_{liquid}$) is less than one. The solute rejected is then diffused through out the rest of the system, following the concentration gradients. In most cases, the rate of rejection of solute from the solid region and the growth rate of the solid-liquid interface are proportional to each other [1].

When the solid-liquid interface moves it produces a concentration change that mathematically can be expressed by equation 2-5:

$$N = \Delta f^{l}_{ijk} \cdot (C^{l}_{liq\_ijk} - C^{l}_{sol\_ijk}) \cdot \Delta x \cdot \Delta y \cdot \Delta z \quad (2\text{-}5)$$

where

- N is the excess solute due to the solid-liquid interface movement
- Δf^{l}_{ijk} is the change in the phase field.

- $C^{l}_{liq\_ijk}$ is the liquid concentration in cell i,j,k at the current time step
- $C^{l}_{sol\_ijk}$ is the solid concentration in cell i,j,k at the current time step
- Δx, Δy, Δz are the grid size in x, y and z coordinate direction [5].

The excess solute that is rejected is redistributed through out the liquid phase of the neighboring cells when the liquid fraction in that cell is less than 50% of the cell volume. When the liquid fraction in a cell is greater than 50% of the cell volume the excess solute is redistributed through out the cell. This will be explained in more detail in Chapter 3, since this is a principle part of the concentration redistribution function.

In the same way that solute is transferred from areas of high concentration to areas of low concentration, heat moves from hot to cold areas within space. Heat diffusion is not taken into account by our model since our model assumes isothermal conditions. Dendrite simulations are usually modeled either taking into account solute redistribution (mass transfer) or heat redistribution (heat transfer). To take both mass transfer and heat transfer into account is very computationally intensive.

Another effect that influences the growth of a dendrite is the capillarity effects, this effect is explained in more detail in the next section.

2.3.3 Capillary effects

Curvature at a point on the solid-liquid interface can be defined as the amount of degrees of bending or a tendency of that point to depart from a tangent drawn to the solid-liquid interface at that point [6]. The curvature of the interface multiplied by the Gibbs-Thomson coefficient is known as capillarity undercooling as shown in equation 2-6.

$$\Delta T_r = k \cdot \Gamma \quad (2\text{-}6)$$

where

- k is the curvature of the solid-liquid interface
- Γ is the Gibbs-Thomson coefficient

Capillary effects are only important in those interfaces which size is approximately less than 10 micrometers. The free enthalpy of a small particle in a liquid melt increases as the size of the particle decreases, while the free enthalpy of the liquid surrounding the particle remains constant. The free enthalpy of the liquid remains constant because the liquid metal surrounding the particle is greater than the size of the particle. The free enthalpy of a small curved particle can be defined as the multiplication of its internal pressure by its molar volume. This internal pressure can be mathematically expressed as the multiplication of the curvature of the interface and the interface energy. The interface energy is defined as: "the reversible work required for the creation of a new surface area. In the case of a solid-liquid interface, the specific interface energy can be set equal to the interfacial tension."[1, page 204]. In summary, as the curvature increases the internal pressure increases and thus the free enthalpy of the particle increases. Capillarity effect then plays an important role on nuclei formation, growth, dendritic and eutectic morphologies.

Mathematically curvature of an interface is calculated by equation 2-7:

$$K = \nabla \cdot \left(\frac{\nabla g}{|\nabla g|} \right) \quad (2\text{-}7)$$

where

- g is the averaged phase field of the solid-liquid interface
- $\nabla\cdot$ (including the dot) is the divergence of a vector
- ∇g is the gradient of the average phase field
- $\nabla g/|\nabla g|$ is the unit vector of the average phase field.[1]

Curvature multiplied by Gibbs-Thomson coefficient is known as capillary undercooling. Gibbs-Thomson coefficient is the ratio of the interface energy and the solidification entropy [5].

Chapter 3, presents in more detail how the curvature of the solid-liquid interface is calculated. In this chapter we just wanted to present the overall theory and concept of curvature and how this one affects the growth of the solid-liquid interface. In the next section, we briefly present the different numerical methods used to describe the formation and growth of a dendritic structure.

2.4 Dendrite formation and growth

Several models have been developed to describe crystal growth. Some of these are:

- Tracer point model,
- Phase field model and
- Cellular model.

The first two models will be briefly described in the next sections. The cellular model will be described in depth in Chapter 3.

2.4.1 Tracer points

The tracer point model consists in representing the solid-liquid interface as a set of points in space as shown in figure 2-4.

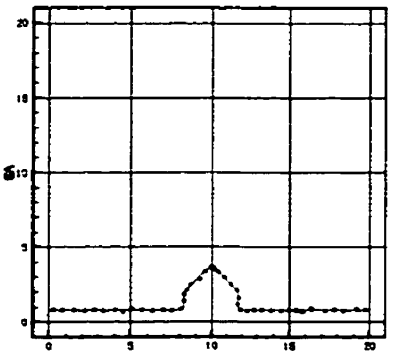


Figure 2-4: Tracer point representation

This method is useful when the shape of the solid-liquid interface is not very complex. As the dendrite grows the morphology of the solid-liquid interface becomes complex very fast, due to the formation of secondary and tertiary branches as shown in Figure 2-5.

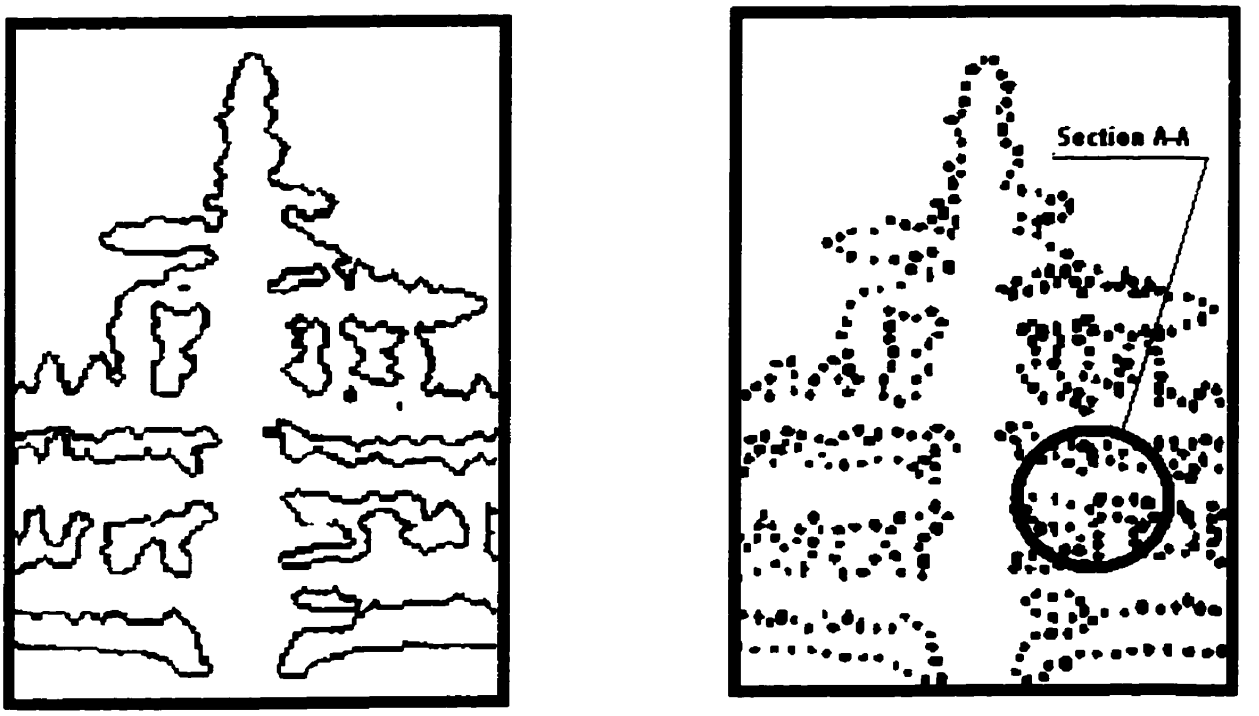


Figure 2-5: Complex Solid-Liquid Interface using tracer points [7]

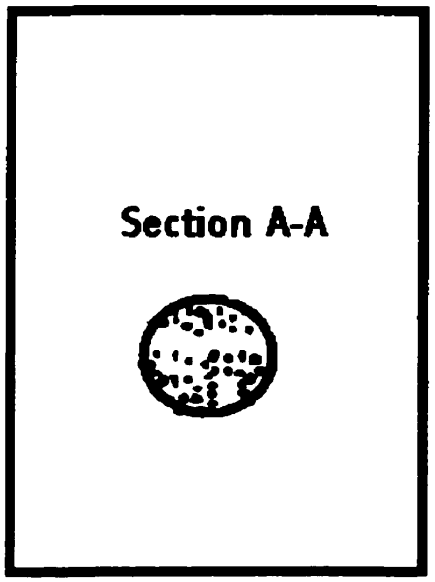


Figure 2-6: Section A-A

Figure 2-6, shows one of the disadvantages of using tracer points when the shape of the solid-liquid interface becomes complex. The use of tracer points to represent complex solid-liquid interface becomes very computationally intensive and it becomes very difficult to keep track of the position of each point with respect to its neighbors in time. A 3-D representation of the solid-liquid interface morphology would be even harder since the points might find themselves in situations that would allow them to move through the control volume 360° in all coordinate directions. The implementation of boundary conditions is not a trivial task.

The use of the tracer point model is feasible when the solid-liquid interface is simple and two-dimensional. In section 3.2 we briefly describe another method to mathematically describe crystal growth.

2.4.2 Phase Field model

The last section of this chapter briefly describes another method used to simulate dendrite structures, cellular structures and processes such as Ostwald ripening among others. The phase field model provides a simple and elegant method for simulating the physical phenomenon already mentioned. The phase field model is also computationally simple to implement in comparison to methods such as the tracer point method.

The phase field model is based on a mathematical equation known as $\varphi(x,t)$ which characterizes the interface at each point in space and time. The phase field model assumes $\varphi(x,t)$ to be a constant. For example if $\varphi(x,t)= ½$ a phase field value greater than ½ would denote a solid region while a phase field value less than ½ would represent a liquid region. This phase field function exists within a fixed region symbolized by Ω. This region has boundary conditions described at $\partial\Omega$. Parameters such as temperature and concentration for the alloying components would be represented by $c(x,t)$, while the temperature would be represented as $T(x,t)$ [8].

Another important aspect of the phase field model is that it assumes that the Helmholtz free energy ($\Im$) is also a function of the $\varphi(x,t)$. Thus mathematically Helmholtz free energy would be described as:

$$\Im(\varphi,...) = \int_{\Omega}[f(\varphi,...) + \frac{1}{2}\varepsilon^2(\nabla\varphi)^2 + ...]\cdot d\Omega \quad (2\text{-}7)$$

where,

- φ is the phase field function
- Ω is the region containing the phase field function
- dΩ is the boundaries of the phase field function.
- ε is a constant

While the free energy density is described as:

$$f(\varphi,T) = W\int_0^{\varphi} p(p-1)[p-\frac{1}{2}-\beta(T)]\cdot dp \quad (2\text{-}8)$$

where

- W is a constant (Joules/m3)
- β(T) is a monotonic increasing function of temperature

The monotonic increasing function β(T) is a function of temperature such that at Tm (the freezing temperature) and at |β(T)| less than a half the monotonic increasing function β(T) is equal to zero.

By graphing the free-energy density function as function of the phase field value we obtain two minimums in which the phase field may exist in a stable condition. These minimums occur at a phase field of zero and a phase field value of 1 (completely liquid or completely solid regions). This would indicate that a change of phase from solid to liquid or liquid to solid within the region represented by Ω incurs an energy penalty. Any change in the phase field φ that departs the phase field value from zero or one would produce an increase in the total energy of the system. The restriction that |β(T)| less than one half ensures that these two minimums exist as seen in figure 2-7.

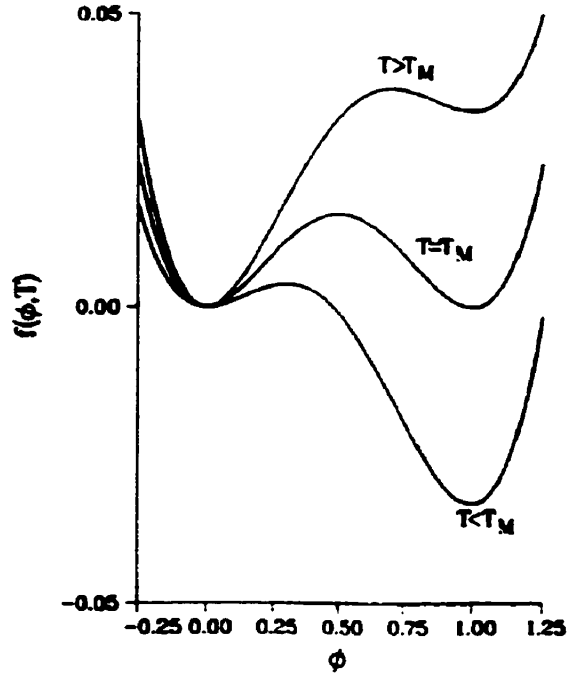


Figure 2-7: Free-energy density function [8].

Figure 2-7 also shows that when the system temperature is greater than the melting temperature of the host component, the global minimum on the free energy density curve occurs at a phase field value of zero (liquid state). It also shows that when the system temperature is less than the melting temperature of the host component then the global minimum of the free energy density curve occurs at a phase field value of one (solid state).

The phase field model can also be used to model the phase field evolution of a binary system in an isothermal condition, although the model can be extended to a tertiary and higher order systems. The free energy density for each component is of the form of equation 2-8 where the constants that describe this equation are unique to each alloying component. Assuming the following conditions:

$$T_M^B < T < T_M^A \quad then$$
$$-\frac{1}{2} < \beta_A(T) < 0 < \beta_B(T) < \frac{1}{2} \qquad (2\text{-}9)$$

where

- T_m is the melting temperature of component A or B
- T is the temperature of the system
- β is a monotonic increasing function of T.

Then the energy density function of a solution assuming it is an ideal solution is described by equation 2-10:

$$f(\varphi,c,T) = c \cdot f_B(\varphi,T) + (1-c) f_A(\varphi,T) + \frac{RT}{v_m}[c \ln c + (1-c)\ln(1-c)] \quad (2\text{-}10)$$

where

- R is the universal gas constant and
- v_m is the molar volume (assumed to be constant) [8]

The first two terms of equation 2-10 describes are the contribution of the free energy density of each component, while the last term of this equation relates to the mixing of the components in the solution (assuming ideal solution).

In the phase field model the phase field evolution is mathematically described as follows:

$$\frac{\partial \varphi}{\partial t} \propto L\left(\frac{\partial \Im}{\partial \varphi}\right) \quad (2\text{-}9)$$

where

- L is a differential operator

- $\Im$ is the Helmholtz free energy as described by equation 2-7 [8]

The last section to be described by the phase field model is the boundary conditions. These are described by equation 2-10:

$$\frac{\partial \varphi}{\partial n} = \frac{\partial c}{\partial n} = 0 \quad (2\text{-}10)$$

where

- n is the normal to the boundary ($\partial\Omega$)
- c is the composition of the solution and
- φ is the phase field function [8].

Equation 2-10 states that the change in composition of the system due to mass transport across the boundary is equal to zero.

One of the mayor advantages of the phase field model is that it describes very well the physical phenomenon that affects dendrite growth. This model has lead the way in understanding the parameters that affect dendritic morphology. The mayor disadvantage of this model is that its implementation for a three dimensional model of alloys is too computationally intensive. A significant improvement in finding a solution to this problem has been achieved for thermal dendrites. The computational efficiency of the phase field method has been enhanced with the introduction of the formulation allowing for the use of a larger boundary thicknesses [9,10]. The most important result of this improvement is a real possibility to produce a 3-D simulation. However, this opportunity

is limited to thermal dendrites of pure materials. The cellular model overcomes the limitations of the phase field model in implementing it for large dendrite simulations from a computational point of view. The cellular model does not describe with the same degree of exactness the solid-liquid interface morphology, and certain parameters such as curvature are estimated. The major advantage of the cellular model is its ease of implementation for large three-dimensional systems in binary or tertiary alloys.

The results that are described in this thesis of dendrite growth for tertiary alloys are the first to describe realistic dendrite growth in alloys.

In conclusion, the phase field model is one of the computational (mathematical) techniques used to describe physical phenomenon that occurs in nature such as the formation and growth of crystals during solidification. In the next chapter the cellular model, which was used to describe the growth of a dendrite, is explained and described in detail.

Chapter 3: Cellular Model of Dendrite Growth

3.0 Introduction

This chapter describes the cellular model and the ways it was used to simulate the growth of a dendrite in three dimensions (3-D) for a three component system. The chapter presents in detail the mathematical equations that describe the phase evolution of the system. It also contains the mass transport equations and explains how these are approximated using the finite difference method to obtain the solute concentration profiles. We then present the ways each cell uses the values of solute concentration, curvature of the solid-liquid interface and thermal undercooling to determine the interface velocity at each cell. This chapter also describes the functions that are part of two computer programs that define our model; these are Phase-maker and Phase-evolution. In the next pages, we will discuss in detail the following sections of our model:

- Phase field representation
- Mass transport equations
- Curvature analysis
- Stability criteria and time step
- Interface Velocity

- Phase field evolution
- Isothermal Conditions
- Anisotropy effects

3.1 Phase field representation

In the cellular model, the control volume is divided into small sections known as cells. The cells are assigned a phase field value that represents the physical state of the material in that cell. The phase field value is equal to the volume fraction of the solid phase in the cell.

The cells are assigned specific lengths in x, y and z direction. In our model, the actual length of each cell in every direction of the coordinate system is the same. The sum of all the cell in one direction determines the total length of the control volume in that direction. Figure 3-0, shows how the control volume is divided into a smaller cells, forming a numerical mesh.

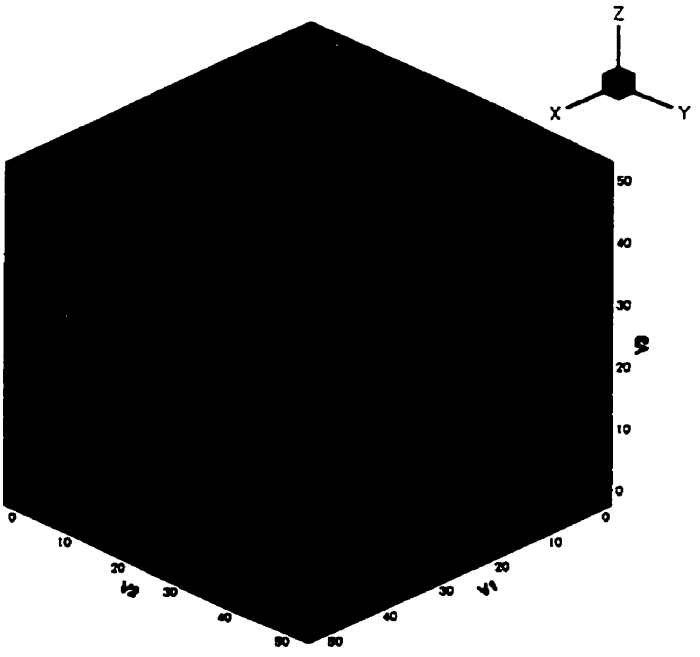


Figure 3-0: Control Volume

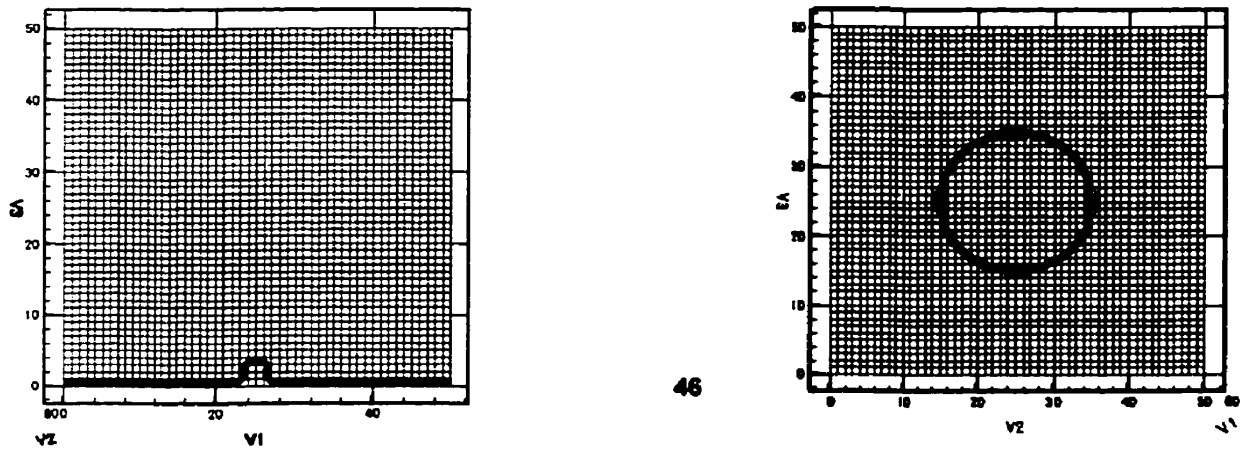
Each cell is assigned an identifier within the control volume that is equal to the cell number in the x, y, and z coordinate system. As an example, cell 30,20,40 is a cell located at 30 cells in the x direction, 20 in the y direction and 40 in the z direction. In order to create the initial phase field, a computer program known as Phase-maker was developed. Phase-maker is a simple computer program written in C language. The program is a for-loop that assigns a specified value equal to zero or one to the phase field array at every cell in the control volume. Phase-maker also assigns the initial concentration values of the two alloying elements. The third element is obtained from the fact that in each cell the sum of all elements adds to 100%. Phase-maker also asks the user for other physical and material parameters that are required by the phase evolution part of the model. These physical and material parameters are:

- Grid size of each cell (meters),
- Number of cell in the x and y coordinate system (i,j),
- Number of cell in the z coordinate system (k),
- Initial concentration of element 1 (in our study element 1 is Silicon),
- Initial concentration of element 2 (in our study element 2 is Copper),
- Temperature of the system (Kelvins),
- Is noise added or not to the system,
- Kinetic coefficient (m/(sK)),
- Diffusion constant of the liquid host (in our model the host is Aluminum),
- Liquid diffusion constant of Silicon in Aluminum (K/%wt),
- Liquid diffusion constant of Copper in Aluminum (K/%wt),
- Gibbs Thomson Coefficient of the alloy (mK),

- Liquidus slope of Silicon in Aluminum from the phase field diagram (K/%wt using the negative sign convention),
- Liquidus slope of Copper in Aluminum from the phase field diagram (K/%wt using the negative sign convention),
- Solidus slope of the Silicon in Aluminum from the phase field diagram (K/%wt using the negative sign convention),
- Solidus slope of the Copper in Aluminum from the phase field diagram (K/%wt using the negative sign convention),
- Melting Temperature of the host component (K),
- Boundary condition (Periodical in i,j but not in k or periodical in i,j, and k),
- Every how often will the code save the information to the hard drive.

All of these parameters and the initial phase field are stored in a text file to be read by the Phase-Evolution program. The actual code for Phase-maker is found in Appendix "C". An initial phase field is presented in figures 3-1 and 3-2. The visualization tools is known as Techplot (Amtec Engineering Inc), and allows us to visualize in three dimensions the contour plots of the solid-liquid interface.

Figure 3-1: Initial phase field cut in two dimensions (2-D)



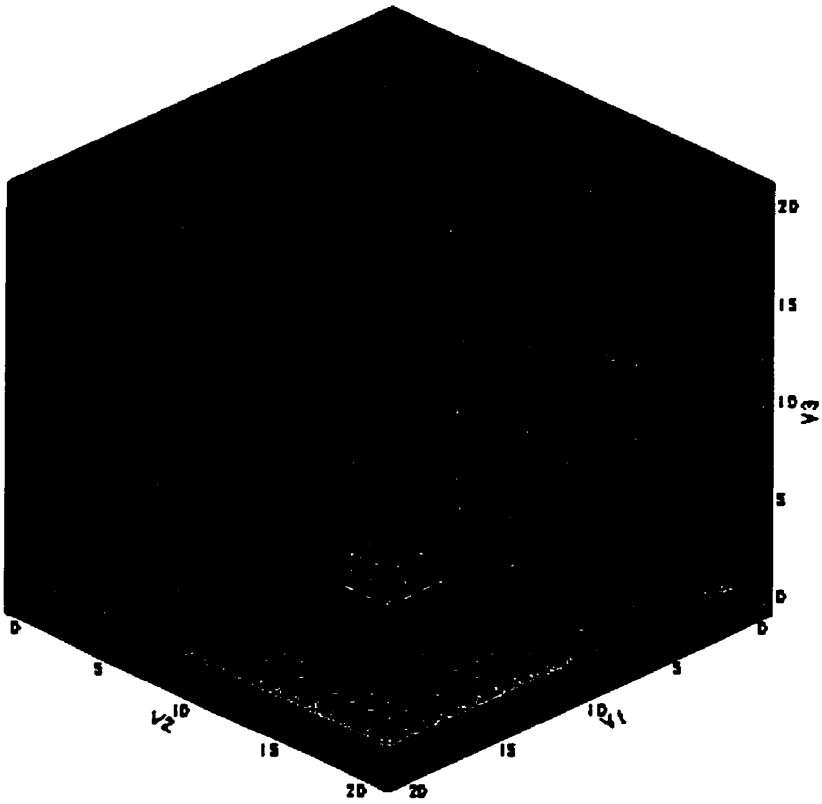


Figure 3-2: Initial Phase Field Contour Plot in 3-D

Figure 3-2, represents the initial nucleus that is required for the growth and formation of a dendrite. It is this initial information of the nuclei that Phase-maker produces as an output file known as "phasein.txt". The initial nucleus can be either on a flat plate or it can be perturbations on a sphere. Phase-maker creates this initial sphere by using a random generator. This random generator generates a random number of points within each cell. Phase-maker then compares the position of each random point in that cell to a radius that has been specified by the user. If all points are within the radius then the cell is assigned a phase field value of one, if the points are all outside the radius then the phase field in that cell is assigned a value of zero. If the cell has points that fall in and outside the

radius value it assigns the phase field value of that cell a percentage of the total number of points that fall within the radius.

Most runs in our analysis are done on the formation of a dendrite from a flat plate as shown in Figure 3-2. This is because computationally it is less time consuming. The Phase-evolution program models the evolution of the initial nuclei to the formation of the dendrite. The phase-evolution code can be found in Appendix "C". The formation of a dendrite requires a perturbation, otherwise an initial flat plat would solidify as a flat plate, while a perfect sphere will solidify as a perfect sphere. In nature, this is not likely to occur and special processes are required in order to achieve flat plate solidification as for example the formation of crystal wafers in the semiconductor industry. It is important to understand that no physical surface is completely flat, while mathematically one may represent a surface to be completely flat. The presence of the perturbation leads to the growth of the nucleus and eventually the formation of a dendrite. Phase-Evolution is capable of simulating the growth of the nucleus due to mass transport effects at the solid-liquid interface. This mass transport leads to a movement of the solid-liquid interface. This change in the position of the phase field in time can be mathematically expressed as:

$$\Delta f_{ijk}^{l} = f_{ijk}^{l+1} - f_{ijk}^{l} = \frac{\Delta t \cdot v \cdot S}{\Delta x \cdot \Delta y \cdot \Delta z} \quad (3\text{-}0)$$

where

- Δt is the time step
- l is the time enumerator
- i, j, k are the cell numbers along the x, y and coordinate system
- S is the area of the interface boundary in a cell
- f_{ijk} is the value of the phase field in cell i,j,k

- Δf_{ijk} is the change in the phase field value per time step
- Δx, Δy and Δz are the grid size of the cell in the x, y and z coordinate system [5].

From equation 3-0, it is possible to determine the new position of the phase field at every cell. The factors that affect the change in position of the solid-liquid interface are the velocity interface at each cell, the time step and that the cell has a fraction solid or one immediate neighbor that is completely solidified. The velocity of the interface is influenced by parameters such as, concentration, temperature and curvature of the solid-liquid interface. The solute transfer drives the growth of the nucleus from areas of higher concentration of solute to areas of lower concentration of solute. The mass transfer process is explained in more detail in section 3.2.

3.2 Mass Transport

Mass transport is the process by which mass is moved through the control volume in accordance to Fick's laws of diffusion. Concentration is diffused within the control volume from region of high concentration to regions of low concentration. Every cell is assigned a concentration value for the liquid and solid phase. In our model, mass transfer through the solid cells is not taken into account. This approximation is possible because the rate of diffusion in liquid is approximately one thousand times greater than in solid. The change in the liquid concentration of a cell for any alloying component can be derived from expanding Fick's Second Law of Diffusion in three dimensions (equation 2-

4). The finite difference approximation for the evolution of concentration in a cell can be mathematically expressed as (equation 3-2):

$$\frac{C_{ijk}^{l+1} - C_{ijk}^{l}}{\Delta t} = D \cdot \left[\left(\frac{C_{i+1,j,k}^{l} + C_{i-1,j,k}^{l} - 2 \cdot C_{ijk}^{l}}{\Delta x^2} \right) + \left(\frac{C_{i,j+1,k}^{l} + C_{i,j-1,k}^{l} - 2 \cdot C_{ijk}^{l}}{\Delta y^2} \right) + \left(\frac{C_{i,j,k+1}^{l} + C_{i,j,k-1}^{l} - 2 \cdot C_{ijk}^{l}}{\Delta z^2} \right) \right]$$

where

- Δt is the time step
- "l" is the time numerator.
- C_{ijk}^{l+1} is the new concentration in that cell.
- C_{ijk}^{l} is the concentration in that cell for the previous time step
- Δx, Δy, Δz are the grid size in each coordinate direction
- i,j,k are the coordinate indices of each cell along the x, y and z coordinate system and
- D is the diffusion constant [5].

The finite difference scheme was used since it is computationally easier to implement than the finite element approximation due to the local character of the evolution equations for fractions of solid.

Equation 3-0 allows us to calculate the new concentration values for each cell per time step. It is important to note that equation 3-0 does not include the effects of concentration build up at the interface due to the solid-liquid interface movement. In order to take into account the concentration change due to solid-liquid interface movement, it is necessary to add to the right hand side of equation 3.0 the excess solute released. The excess solute released due to solid-liquid interface movement is calculated as:

$$Excess\_Solute = \Delta f_{ijk} \cdot \left(C^{l}_{liq,ijk} - C^{l}_{s}(T_{ijk})\right) \quad (3\text{-}2)$$

where

- Δf_{ijk} is the change in the phase field
- $C^{l}_{liq\ ijk}$ is the liquid concentration of that cell
- $C^{l}_{s}(T_{ijk})$ is the solidus concentration of that cell at the current system temperature [5].

This solidus concentration is obtained from the phase diagram at the system temperature. Equation 3-2 can also be expressed as:

$$Excess\_Solute = \Delta f_{ijk} \cdot (C^{l}_{liq\ ijk} \cdot (1-k)) \quad (3\text{-}3)$$

where

- k is the distribution coefficient [5].

The distribution coefficient is the ratio of the solidus to liquidus concentration lines. This ratio is approximately 0.13. Equation 3-2 and 3-3 are equivalent to each other. In the cellular model the equations used to calculate the new concentration values for each cell is very dependent on the amount of liquid fraction in the cell. Our model assumes that when the fraction of solid in each cell is small the mass is transferred within the cell. As the cell is almost completely solidified (more than 50% of the cell) then the model assumes that the transfer of mass occurs proportionally to the amount of liquid in the neighboring cells. The neighboring cells are only the immediate neighbors that have at

least one face in common with the cell under study. In the following sections, we present how the new concentration values for each cell are calculated and how the diffusion effects are taken into account.

3.2.1 Mass Transfer in cells with liquid fractions greater than or equal to ½

In a cell that is completely liquid or has a small fraction of solid (a fraction of solid which is less than 50% of the cell) the concentration evolution of any alloying component is calculated as follow:

$$C_{ijk}^{l+1} = C_{ijk}^{l} + \frac{D_l \cdot \Delta t}{(1-f_{ijk}^{l})} \cdot \left[\begin{array}{l} \dfrac{(C_{i+1jk}^{l} - C_{ijk}^{l}) \cdot \theta(1-f_{i+1jk}^{l})}{\Delta x^2} - \dfrac{(C_{ijk}^{l} - C_{i-1jk}^{l}) \cdot \theta(1-f_{i-1jk}^{l})}{\Delta x^2} \\ \dfrac{(C_{ij+1k}^{l} - C_{ijk}^{l}) \cdot \theta(1-f_{ij+1k}^{l})}{\Delta y^2} - \dfrac{(C_{ijk}^{l} - C_{ij-1k}^{l}) \cdot \theta(1-f_{ij-1k}^{l})}{\Delta y^2} \\ \dfrac{(C_{ijk+1}^{l} - C_{ijk}^{l}) \cdot \theta(1-f_{ijk+1}^{l})}{\Delta z^2} - \dfrac{(C_{ijk}^{l} - C_{ijk-1}^{l}) \cdot \theta(1-f_{ijk-1}^{l})}{\Delta z^2} \end{array} \right] + \frac{\Delta f_{ijk}^{l} \cdot (C_{liq\,ijk}^{l} - C_{sol\,ijk}^{l}(T_{ijk}^{l}))}{(1-f_{ijk}^{l})} \quad (3\text{-}4)$$

where

- C^{l+1}_{ijk} is the new concentration value in a cell
- C^{l}_{ijk} is the concentration value at the previous time step
- i,j,k indicate the cell number and the location of the cell within the control volume
- f_{ijk} is the fraction of solid in a cell

- Δf^{l}_{ijk} is the change in the fraction of solid within the cell (phase field change).
- D_{liq} is the liquid diffusion constant
- Δt is the time step
- Δx, Δy, Δz is the grid size of the cell in the x, y and z coordinate system and
- θ is a step function [5].

Equation 3-4, takes into account the concentration diffusion from all neighboring cell as well as the effects in the change in position of the solid-liquid interface in that cell. The $(\theta(1-f^{l}_{ijk}))$ step function is determined as follow:

$$\begin{aligned} \theta(x) &= 0 \quad x \leq 0 \\ \theta(x) &= 1 \quad x > 0 \end{aligned} \tag{3-5}$$

The step function (Equation 3-5) is included in equation 3-4 to prohibit the diffusion of solute from the liquid phase into the solid phase. This is done to avoid solute diffusion from the liquid phase into the solid phase. In reality certain cells might receive solute from the liquid, but the physics behind this transfer is not taken into account in our model for simplification reason and also because it would not contribute significantly to a different morphology, due to the slow process of intake of solute by the solid phase. Equation 3-4 can be used also to simulate the concentration evolution in the solid phase. The only parameter that would change is the value of the diffusion constant.

The second scenario that is required to be simulated is when the liquid fraction in the cell is less than 1/2. This scenario is described in section 3.2.2.

3.2.2 Mass Transfer in cells with liquid fractions less than ½

When a cell solidifies the liquid percentage in the cell is reduced and the liquid fraction approaches zero, under these last stages of a solidification of the cell, the concentration approaches infinity. It approaches infinity because any concentration assigned to a cell is redistributed as a percentage of the liquid fraction remaining in that cell, and if this liquid fraction approaches zero then the concentration assigned to it would approach infinity, causing large concentration fluctuations.

If the cell suffers a positive change in the phase field then the excess solute is calculated based on equation 3-3. The excess solute is then redistributed to the neighboring cell. Each neighboring cell, receives a percentage of the total excess solute calculated for cell "i,j,k" based on the fraction of liquid remaining in each neighboring cell.

If the cell suffers a negative change in the phase field (re-melting of the cell), then the excess solute calculated is assigned to cell "i,j,k". It is important to note that the process of assigning the excess solute to one cell is not in accordance to the physical phenomenon, but it is done as a simplification to the problem. In most cases, most cells throughout the complete simulation do not suffer re-melting of the solid phase and therefore it is considered a valid approximation.

Mass transfer is not the only parameter that affects the velocity and evolution of the solid-liquid interface, an example of this is curvature. The curvature effects and how these are calculated for the solid-liquid interface are explained in more detail in the next section.

3.3 Curvature of the Solid-Liquid Interface

In order calculate the curvature of the solid-liquid interface, Phase-evolution computes an average phase field. Calculating curvature on the actual solid-liquid interface can provide abrupt changes on the curvature values from one cell to its neighbors. These abrupt changes can cause the curvature function to provide curvature values that are not representative of the actual curvature of the solid-liquid interface. Section 3.3.1 explains in more detail how an average phase field is obtained. This average phase field is then used to calculate the values of curvature at every cell.

3.3.1 Average Phase Field

The phase field represented by f_{ijk} at each cell is used to calculate an average phase field at each cell (g_{ijk}). In the Phase-Evolution program, the average phase field function assigns to each cell a weight factor. These weight factors multiply the solid fraction of each cell.

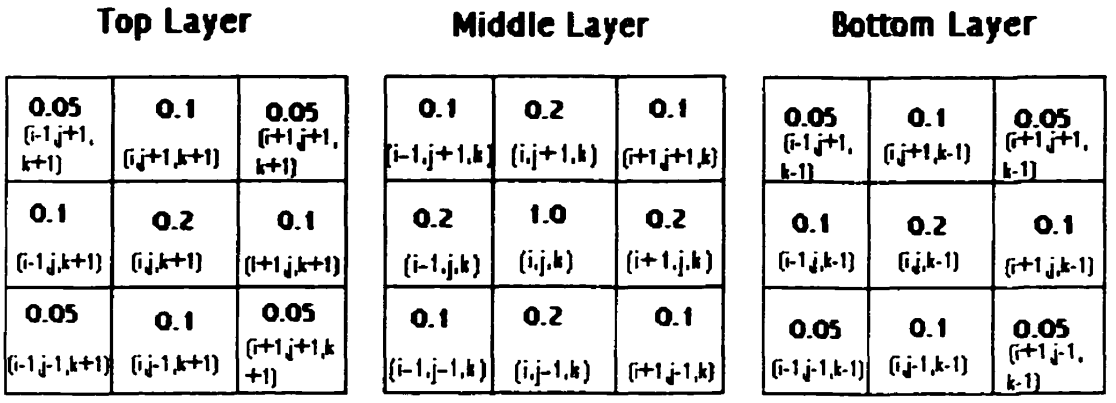


Figure 3-3, Weight Factors

Figure 3-3, shows the weight factors. The summation of all the weight factors adds up to a total value of 3.9. The average phase field for cell i,j,k is calculated as the sum of the solid fraction of each cell multiplied by its corresponding weight factor which is then divided by 3.9. The sum of all the weight factors are assigned to an average phase field array. This operation is the repeated for every cell in the control volume at every time step.

The weight factors where determined by creating a spherical solid-liquid interface. The curvature function is then used to compute the curvature at every cell on the sphere. The inverse of the curvature at each cell provides the radius of the sphere, which can then be compared to the theoretical radius of the sphere. The best weight factors obtained from this test are shown in Figure 3-4

Top Layer		
0.120	0.3	0.120
0.3	0.35	0.3
0.120	0.3	0.120

Middle Layer		
0.3	0.35	0.3
0.35	1.0	0.35
0.3	0.35	0.3

Bottom Layer		
0.120	0.3	0.120
0.3	0.35	0.3
0.120	0.3	0.120

Figure 3-4, Weight Factors for a spherical nucleus

Figure 3-5 and 3-6, show the test sphere and a cross section of it. The inner part of the sphere is assigned a phase field value of one (solid state), while the outer area is assigned a phase value of zero (liquid state). All cells in between the solid and liquid phase will have a phase field value between zero and one, corresponding to the fraction of solid in each cell.

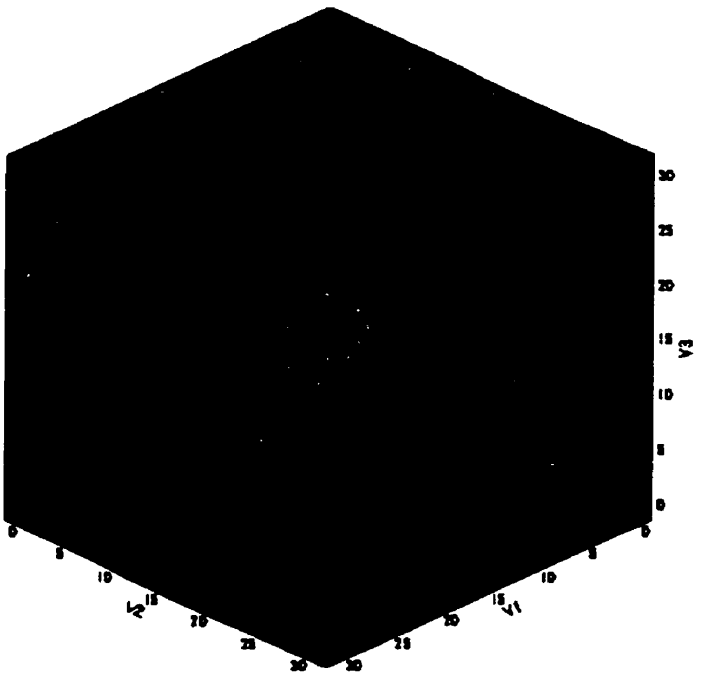


Figure 3-5: Test Sphere

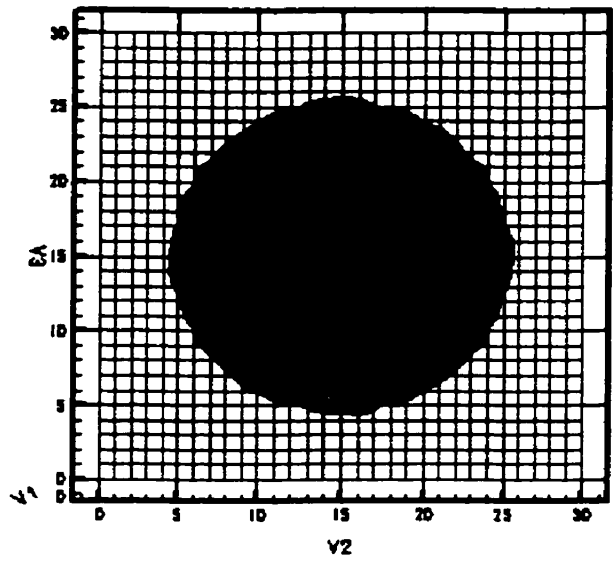


Figure 3-6: Cross Section of Test Sphere

The results of the curvature values obtained using the weight factors shown in figure 3-4 are presented in table 3-1.

Table No. 3-1: Curvature Results

Test Case No.	Delta	Radius	Theoretical Curvature.	Total No. of nodes	Pass within 30% (%)	Pass within 50% (%)	Pass within 60% (%)	Failures (%)	Average Curvature
1	0.25	3	0.33	2485	15	16	9.9	50	0.66
2	0.5	3	0.33	620	33	14	27	26	0.63
3	0.75	3	0.33	282	30	22	26	22	0.65
4	1.0	3	0.33	161	29	30	22	19	0.62
5	10.0	30	3.33E-2	161	29	30	22	19	0.062
6	1.0	4	0.25	308	31	25	20	23	0.48
7	1.0	5	0.20	408	22	23	24	31	0.39
8	1.0	6	0.17	659	25	22	27	27	0.32
9	1.0	7	0.14	840	22	27	16	35	0.29

As it is seen in Table 3-1, the average curvature produced using the weight factors shown in figure 3-4 overestimated the theoretical curvature. The incorrect weight factors can provide average phase fields that are not representative of the actual solid-liquid interface as it is shown in figure 3-7.

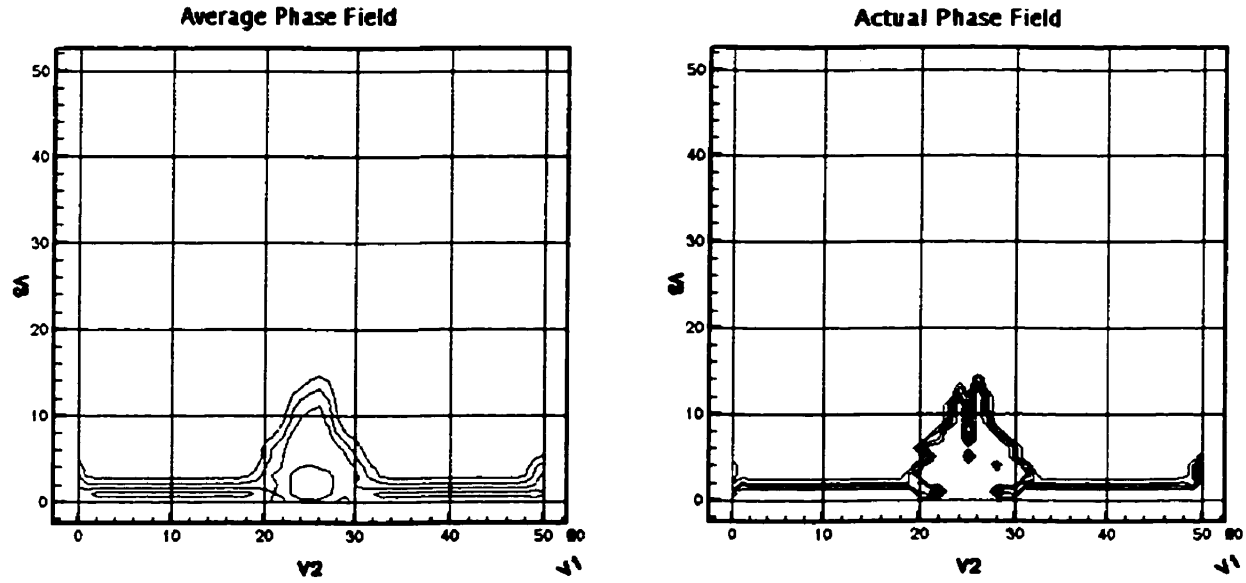


Figure 3-7: Average phase field using the weight factor for spherical nucleus.

The average phase field shown in figure 3-7 was obtained using the weight factors shown in figure 3-4. As it is seen in figure 3-7 the average phase field does not reflect the splitting that occurs at the tip of the dendrite. This over smoothening of the phase field is because the weight factors used overestimate the contribution of its neighbors on the ijk cell when computing the average phase field.

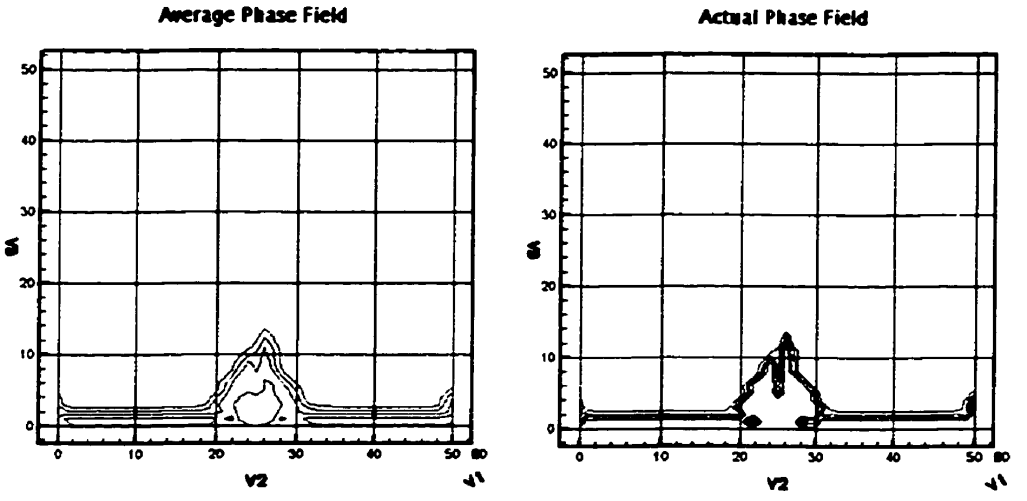


Figure 3-8: Average and Actual Phase Field using the weight factors of Figure 3-3

The phase field shown in figure 3-8 show the average and actual phase field of a dendrite with a split tip (anisotropy effects not taken into account). The average phase field reflects what is occurring on the actual phase field. A wrong estimation of the average phase field can be detrimental in the simulation of a dendrite, since curvature is an important parameter in the calculation of interface velocity. The calculation of curvature on the averaged phase field is explained in detail in the next section.

3.3.2 Computation of curvature on an average phase field.

Mathematically curvature can be defined as:

$$K = \nabla \cdot \left(\frac{\nabla g}{|\nabla g|} \right) \quad (3\text{-}6)$$

where,

- g is the averaged phase field
- $\nabla\cdot$ (including the dot) is the divergence of a vector
- ∇g is the gradient of the average phase field
- $\nabla g/|\nabla g|$ is the unit vector of the average phase field.

A more detailed form of equation 3-6 for a three dimensional system (3-D) is shown in equation 3-7:

$$K = \frac{g_x^2(g_{yy}+g_{zz})+g_y^2(g_{xx}+g_{zz})+g_z^2(g_{xx}+g_{yy})-2(g_x g_y g_{xy}+g_x g_z g_{xz}+g_y g_z g_{yz})}{(g_x^2+g_y^2+g_z^2)^{\frac{3}{2}}} \quad (3\text{-}7)$$

where

- K is curvature of the average phase field
- x, y and z are the coordinate directions within the control volume
- g_x is the partial derivative of the average phase field "g" with respect to x
- g_y is the partial derivative of the average phase field "g" with respect to y
- g_z is the partial derivative of the average phase field "g" with respect to z
- g_{xx} is the second partial derivative of the average phase field "g" with respect to x
- g_{yy} is the second partial derivative of the average phase field "g" with respect to y
- g_{zz} is the second partial derivative of the average phase field "g" with respect to x and z

- g_{xy} is the second partial derivative of the average phase field "g" with respect to x and y and
- g_{yz} is the second partial derivative of the average phase field "g" with respect to y and z.

Equation 3-7 allows us to calculate the curvature value on the average solid-liquid interface field. Curvature is only calculated on those cells that have a fraction of solid greater than but not equal to zero and less than but not equal to one. When the simulation is carried out with small cells (10 time smaller than the theoretical tip radius estimated by Fisher and Kurz model) the results obtained are quite accurate.

Although the curvature effects of the solid-liquid interface are an important parameter in the formation of the dendrite it is not the only one. Temperature is another important parameter that influences the morphology of the dendrite. In the next section, temperature and why the use of an isothermal conditions for the simulation of a dendrite is described in detail.

3.4 Isothermal Condition

Temperature is another parameter that affects the growth of a dendrite. In any casting system, a cooling process within the cast is present. The temperature variation from one side of the cast to the other side, might be large, but if analyzed at a single grain these temperature variations from one side of the grain to another are not likely to be that large. It is this reasoning that allows us to assume that for the simulation of a single dendrite we

can assume a constant temperature through the control volume of that the dendrite. Thus, an isothermal condition is not a probable condition when analyzing large distances within a casting system but can be assumed a good approximation for a single dendrite.
The temperature of the system for all our runs is assumed constant and below the melting temperature of the host component, thus providing a thermal undercooling.

3.5 Stability Criteria and Time Step

The Fourier Number defines the stability criterion for our model. The Fourier number is mathematically described as:

$$Fo = \frac{\alpha\tau}{s^2} \quad (3\text{-}8)$$

where

- s is the characteristic dimension of the body,
- α is the thermal or mass diffuisivity,
- τ is the time step.[11]

The Fourier number as described by equation 3-8 is useful for regular shapes with a characteristic dimension, such as a sphere, cylinders, blocks, etc. In the case of dendrite growth the final morphology is not defined or established beforehand. Dendrite growth is also a very time dependent problem. Transient numerical analysis is therefore required to simulate the growth of dendrites. The Fourier Number (Fo) describes a relationship between the time step and grid size within a numerical mesh. If the grid size is chosen,

the time step is automatically set. In numerical transient conduction problems, the relationship between time and grid spacing is described by the inverse of the Fo number.

$$M = \frac{\Delta x^2}{\alpha \cdot \Delta t} \quad (3\text{-}9)$$

where

- M is a number equal or greater than 2 for 1-D, to 4 for 2-D and 8 for 3-D,
- α is the diffusivity constant and
- Δt is the time step [11].

Our model uses equation 3-9 to determine the time step. The grid size is chosen to be equal to the tip radius estimated by Fisher and Kurz model or 2, 5, 10 or 20 times smaller than the estimated theoretical tip radius. From this theoretical tip radius we determine the grid size, and thus the time step is calculated from equation 3-9. If one did not have a theoretical estimation of the tip radius, one would reduce the size of the grid size until the tip radius value obtained converges to a number.

The use of small time steps is inherent in problems that are solved using the explicit method. An implicit solution would provide the use of larger time steps that one would initially consider an excellent advantage. The disadvantage of having an implicit solution in a non-linear problem like the one of simulating crystal morphologies is that the implicit solution for the whole control volume is much more difficult to implement computationally. The implementation of an implicit solution to a dendrite growth simulation is not impossible but one would fine that solving the solution implicitly would require more computational calculations in solving all the required equations even though the time step is larger, thus our choice to simulate dendrite growth explicitly.

3.6 Interface Velocity

All of the parameters described above influence the final shape and morphology of the dendrite. The formula that puts all these parameters together is the velocity equation. The interface velocity is mathematically described by equation 3-10:

$$V_{ijk} = \mu_k \cdot \left(T_m + C^l_{ijk\_1} \cdot m_{liq\_1} + C^l_{ijk\_2} \cdot m_{liq\_2} - \Gamma \cdot k - T_{ijk}\right) \quad (3\text{-}10)$$

where

- V_{ijk} is the velocity at cell i,j,k (meters per second)
- l is the time enumerator
- T_m is the melting temperature of the host component
- $C^l_{ijk\_1}$ is the liquid concentration of the first alloying component (%wt)
- m_{liq_1} is the liquidus slope from the phase diagram of the first alloying component (K/%wt)
- $C^l_{ijk\_2}$ is the liquid concentration of the second alloying component (%wt)
- Γ is the Gibbs-Thomson Coefficient (mK)
- k is the curvature of the solid-liquid interface at cell ijk
- T_{ijk} is the temperature of the cell ijk (K).[5]

The melting temperature of the host component is a constant. Due to our assumptions of isothermal conditions the temperature of each cell is also a constant. Thus, the thermal supercooling for each run is constant.

The concentration of the first and second alloying components varies for every time step following the laws of diffusion already explained above. These changes in concentrations at every iteration cause the velocity for the cell to vary at every iteration.

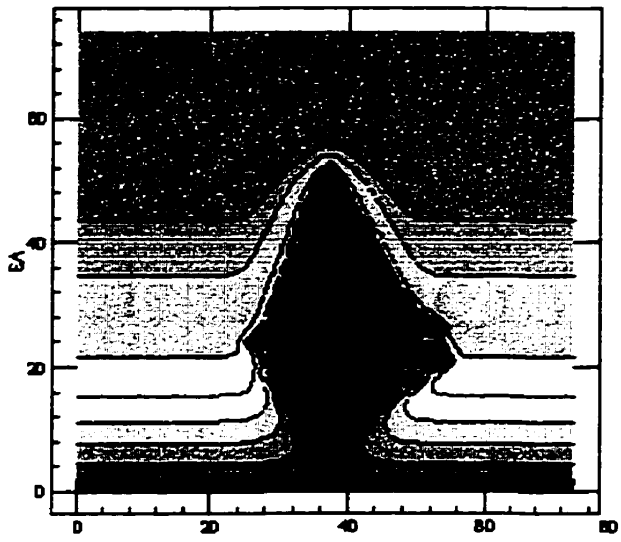


Figure 3-9: Concentration contours.

Figure 3-9, shows a cross section of a growing dendrite and the iso-concentration contours around the dendrite. The closer the contours are one to the other (near the tip) indicate a stronger gradient of concentration. Beyond the transient growth period, the concentration at the solid liquid interface near the tip becomes very close to the liquidus concentration at the interface temperature, taking into account the capillary undercooling[5]. At the steady state condition, the tip of the dendrite grows with a constant velocity, thus indicating an equilibrium condition between temperature, the concentration profiles in that region of the dendrite and a constant curvature.

The kinetic coefficient (μ_k) for most alloys is not known and is not easily determined, although it can be approximated using equation 3-11:

$$\mu_k = \frac{V_s \cdot L \cdot V_m}{R \cdot T_m^2} \quad (3\text{-}11)$$

where

- V_s is the velocity of sound in the liquid alloy
- L is the latent heat of the alloy
- V_m is the molar volume
- R is the gas constant and
- T_m is the melting temperature of the host component. [5]

Equation 3-11, is used as an approximation of the kinetic coefficient. The approximation is always good as long as the growth happens in the capillary regime. In other words, the effects of curvature tend to have a greater influence than the actual value of the kinetic coefficient. In our present model, we use a kinetic coefficient value of 0.11 m/(sK). The multiplication of the interface velocity with the time step allows us to determine how the interface moves in every cell for every time step. In the next section, we explain in more detail how the interface propagates within a cell and the conditions required to propagate into surrounding cells.

3.7 Phase Evolution

When the initial phase field is completely flat and has no perturbation, the phase field solidifies as a flat surface. In reality a completely flat surface with no perturbation is not an easy task and requires extreme care and very well controlled experiments. The flat phase field grows flat because there is no change in the concentrations ahead of the solid-

liquid interface, there is no change in the curvature of the solid-liquid interface and there is no change in the temperature of the system (in the case of our model). The initial perturbation, produces a change in the curvature and a change in the concentration ahead of the interface that leads to different velocities near the perturbation. These changes in velocities lead to the re-melting of the perturbation (negative velocities) or the formation of a dendrite (positive velocities).

The evolution of the solid-liquid interface is mathematically represented by equation 3-12:

$$f_{ijk}^{l+1} = f_{ijk}^{l} + \frac{(\Delta t \cdot V_{ijk})}{\Delta} \quad (3\text{-}12)$$

where

- f^{l}_{ijk} is the phase field value in cell i,j,k,
- l is the time enumerator,
- V_{ijk} is the velocity in the cell i,j,k (m/s),
- Δ is the grid size, in the present model the grid is cubic (m),
- Δt is the time step (s). [5]

Equation 3-12 shows that the new value of the phase field in cell i,j,k is equal to the old value of the phase field plus the time step multiplied by the velocity calculated in that cell and divided by the grid size.

When the velocity is negative in a cell, the fraction of solid is reduced (re-melting). In our model re-melting only occurs in those cells that have a fraction of solid greater than zero but less than one. In other words, once the cell is completely solidified our model does

not allow for re-melting of that cell. This approximation is possible because re-melting of the solid-liquid interface does not regularly occur during solidification and because it simplifies considerable the model. This still leaves the possibility that the change in phase field be greater than the solid fraction available in that cell (re-melting). In this case, the phase field of that cell is assigned a value of zero (completely liquid).

When the change in the phase field is positive (positive velocity), then the solid fraction in that cell is increased. If the change in the phase field is greater than the allowable liquid fraction to solidify the cell is assigned a value of one (completely solid). Mathematically the change in the phase field can be greater than the grid size. In order to avoid changes in the phase field that are greater than the grid size or greater than the available liquid to solidify in that cell the maximum allowable change in the phase field is calculated. This maximum allowable change is based on a thermodynamic equilibrium condition. The maximum allowable change in phase field is compared to the change in the phase field calculated by equation 3-12 and the smallest change in phase field is used for that time step. The thermodynamic equilibrium condition is based on the fact that solidification will not allow changes in concentration greater than that of the liquidus concentration for the given system temperature. If the concentration value of a cell reaches the liquidus concentration, it is considered to be at an equilibrium at which point there is no driving force for that cell to re-melt or to solidify. In the present model the liquidus and solidus concentration curves of the phase diagrams (Al-Si and Al-Cu) are approximated to be straight lines.

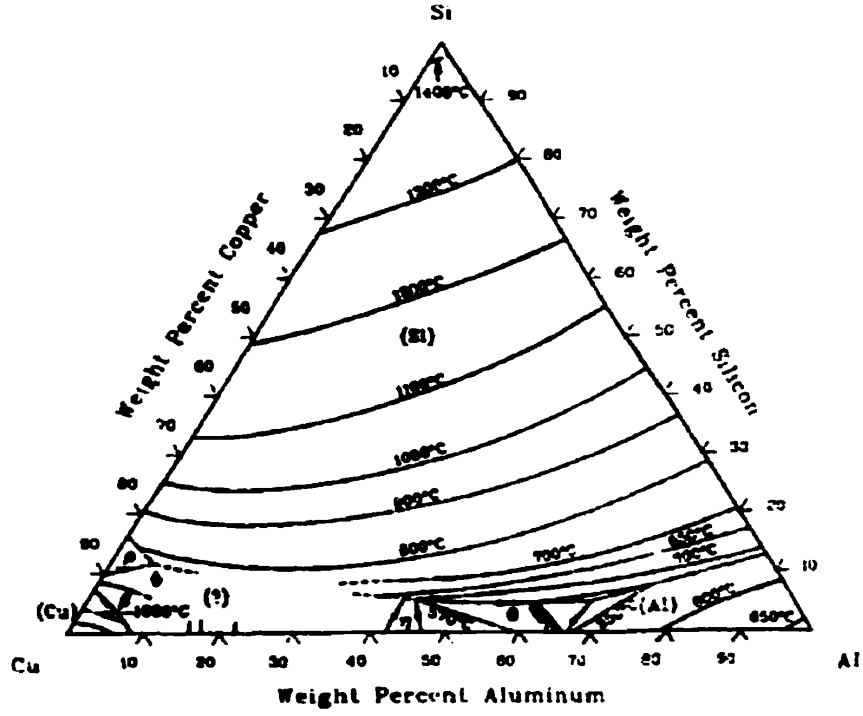


Figure 3-10: Phase Diagram of Al-Si-Cu, Isothermal Section at 955°C

In order for a cell that has no solid fraction start solidifying it has to have a neighbor that is completely solidified. This is taken into account in the present model by the neighbor function. The neighbor function checks all cells that have one face in common with the cell under study. If the cell under study has one of its six neighbors completely solidified then that cell under study is allowed to start solidification, otherwise, only changes in concentrations for both alloying components are computed.

In order to compute the concentration values and phase field values near the walls of the control volume a boundary condition function was required. The next section explains in detail how the model deals with the calculation of concentration and phase field near the walls of the control volume.

3.8 Boundary Condition

The present model assumes periodical boundary conditions near the walls of the control volume. In order to calculate the concentration values or curvatures at the walls of the control volume one needs to know the phase field and concentration gradients at the wall. These gradients require information on concentrations and phase field values on cells that are outside of the control volume. Thus, mathematically the model uses the concentration and phase field values at the opposite side of that wall. This creates a numerical effect of an infinite control volume. When analyzing the solidification of a dendrite from sphere this situation works quite well. It is also used for the sidewalls of the control volume when analyzing the solidification of a dendrite from a flat plate with a perturbation. Periodical boundary conditions are not as effective when dealing with the ceiling of the control volume. When computing the concentration and phase field gradients at the ceiling of the control volume, the boundary condition function uses phase field values and concentration values that are two cell layers below the ceiling. In essence, the ceiling of the control volume behaves like a mirror. If we had used a periodical condition at the ceiling for a dendrite solidifying from a flat surface, the effect would be the growth of the solid-liquid through the ceiling of the control volume.

In the last section of this chapter we will describe the effects of anisotropy and its influence when not taken into account in the simulation.

3.9 Anisotropy Effects

Anisotropy is an effect, which describes a material that has properties that differ in direction [12]. During solidification the dendrite can have directions of growth that are favored over others, this is what is known as growth anisotropy [1]. When simulating crystal growth the existence of a mesh (cell grids) also introduces an anisotropy effect.

Any simulation that uses a grid as part of the model of crystal growth introduces an anisotropy effect on the growth. This effect of the mesh is considered negative and its effect can not be eliminated. It is considered a negative effect since in a physical world dendrites do not have the influence of a mesh during their growth. To actually eliminate the influence that a mesh can have on the crystal growth is a substantial amount of work. One way to eliminate this influence on the growth is by rotating the mesh. This ensures that for every time step that the simulated crystal is allowed to grow the orientation of the mesh will be on a different direction, thus eliminating a preferred growth direction. A disadvantage of this method is that the implementation from a programming point of view is not easy. Another disadvantage, is that the computational time required increases substantially, thus reducing the size of the dendrite that can be modeled and increasing the computer time required to model them.

Al-Si-Cu alloys can form dendritic structures in which anisotropy effects influence the kinetics of the solid-liquid interface. These anisotropic properties influence the formation of the dendrite. It has been shown that dendrite form in preferred crystallographic directions as for example <001> for cubic crystal structure. The thermal and

compositional gradients as well as the anisotropic effects determine the growth direction of a dendrite. Our model is based on mass transfer (compositional gradients) and anisotropy effects. [13]

Growth anisotropy was initially not taken into account by our model. Figure 3-11 shows the effects of isotropic growth of a dendrite from a spherical nucleus.

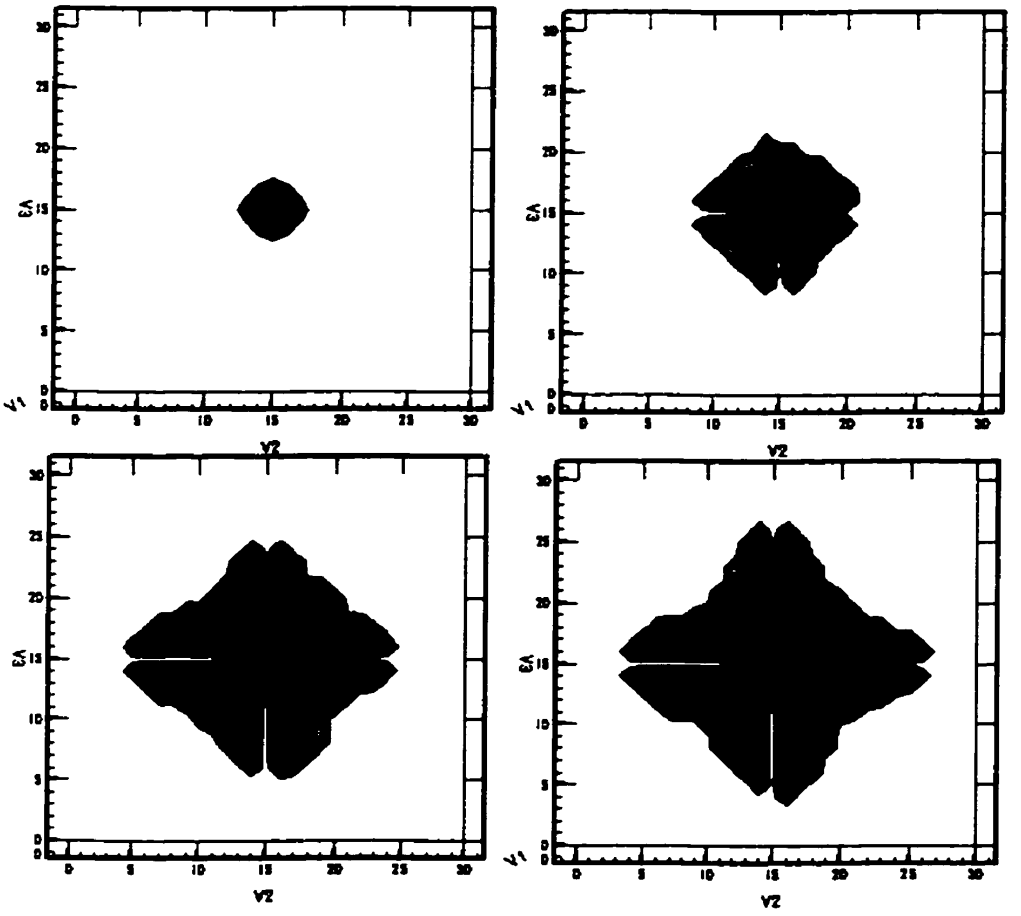


Figure 3-11: Anisotropy Effects

As it is seen in figure 3-11 the dendrite tip splits. In order eliminate this isotropic effect the model calculates the normal unit vectors components in the x, y and z directions.

$$n_x = \frac{g_x}{\sqrt{g_x^2 + g_y^2 + g_z^2}} \quad (3\text{-}13)$$

where

- g_x is the gradient of the average phase field in x
- g_y is the gradient of the average phase field in y and
- g_z is the gradient of the average phase field in z.

Similarly, the normal unit vector components in y and z are calculated where the numerator changes for the gradient of the average phase field in y and z respectively. The model then proceeds to calculate two angles. These two angles are the angles between the normal vector to the solid-liquid interface and the z and x direction coordinate system. These are mathematically calculated as follows.

$$\begin{aligned} \varphi &= \arccos(n_z) \\ \theta &= \arccos(n_x) \end{aligned} \quad (3\text{-}14)$$

where

- phi (φ) is the angle between the normal to the interface and the z coordinate direction
- theta (θ) is the angle between the normal to the interface and the x coordinate direction
- arccos is a mathematical function that returns the angle in radians.

Once these angles are known the capillary undercooling is obtained as follows:

$$Capillary\_undercooling_{ijk} = \Gamma \cdot \left(1 - 0.25 \cdot \cos(4 \cdot \varphi_{ijk}) - 0.25 \cdot \cos(4 \cdot \theta_{ijk})\right) \cdot k_{ijk} \quad (3\text{-}15)$$

where

- Γ is the Gibbs-Thomson coefficient
- cos is the mathematical cosine function
- phi is angle as calculated in equation 3-14 for cell i,j,k
- theta is the angle as calculated in equation 3-14 for cell i,j,k
- k_{ijk} is the solid-liquid interface curvature for cell i,j,k.

Equation 3-15, is the new value of the capillary undercooling that is used in calculating the velocity of the solid-liquid interface. Once the effects of anisotropy where taken into account by our model the splitting of the dendrite tip did not re-occur. Examples of these runs on dendrites growing from a perturbation of a flat surface will be presented in detail in Chapter 4. Simulations on a spherical nucleus where not continued because the amount of computer time required to produce a three-dimensional dendrite was considerably greater. A spherical nucleus will grow six very similar dendrite arms, since all conditions around the control volume are identical, Thus, the information provided by one of the main arms would be the same for all other arms. Thus, simulating the growth from a perturbation on a flat plate is equivalent to growing one of the main arms on a spherical nucleus and by doing so we save computer time. More details on these results are presented in chapter 4.

Chapter 4: Results

4.0 Introduction

This chapter presents the results of a few runs performed with our model. The results are analyzed and when possible compared to the results provided by Fisher and Kurz model [1]. A complete description of Fisher and Kurz model of dendrite growth is presented in Appendix "A". The tip radius for each run is presented and the relationship between the grid size and the tip radius obtained are also presented. Results on the conservation of mass test are shown in this chapter.

4.1 Phase Evolution and Grid Size

Several runs where performed under the following physical and material parameters:

Physical Parameters:

- Grid Size: 1.28e-008 m.
- No. of grids in i and j coordinate direction: 74
- No. of grids in k coordinate direction: 74
- Initial concentration of Silicon: 5%wt.
- Initial concentration of Copper: 5%wt.
- Constant temperature of the system: 873

- No noise is added to the system

Material Parameters:

- Kinetic Coefficient (m/(s*K)): 0.1
- Alloys Diffusion Constant Liquid host (m^2/sec): 3e-009
- Alloys Diffusion Constant Liquid element 1 (m^2/sec): 3e-009
- Alloys Diffusion Constant Liquid element 2 (m^2/sec): 3e-009
- Alloys Gibbs Thomson coefficient (m*k): 2e-007
- Liquid slope of element 1 (K/%wt use negative sign convention): -6
- Liquid slope of element 2 (K/%wt use negative sign convention): -3
- Solidus Slope of element 1 (K/%wt use negative sign convention): -50
- Solidus slope of element 2 (K/%wt use negative sign convention): -20
- Melting temperature of the host component (K): 933.3
- Periodical Boundary Conditions in i, j and not in k.

A theoretical value of the dendrite tip for the above physical and material parameters where predicted using Dr. Artemev 2-D computer model. His model is based on Fisher and Kurz model (Please refers to Appendix "A" of this thesis for further explanation on Fisher and Kurz model). Dr. Artemev's computer model provides results on tip radius in a non-dimensional form, while Phase-Evolution provides the results in a dimensional form. In order to compute the theoretical tip radius (dimensionally) a MathCad spreadsheet was created. An example of this spreadsheet can be found in Appendix "B". According the Dr. Artemev's computer model the theoretical tip radius was calculated to be approximately 5.1E-7 m. As seen above, the grid size found under the physical

parameters was taken to be forty times smaller than theoretical tip radius. Using Techplot (Visualization software by Amtec Engineering, Inc) a cut of the 3-D simulation is presented in Figure 4-1.

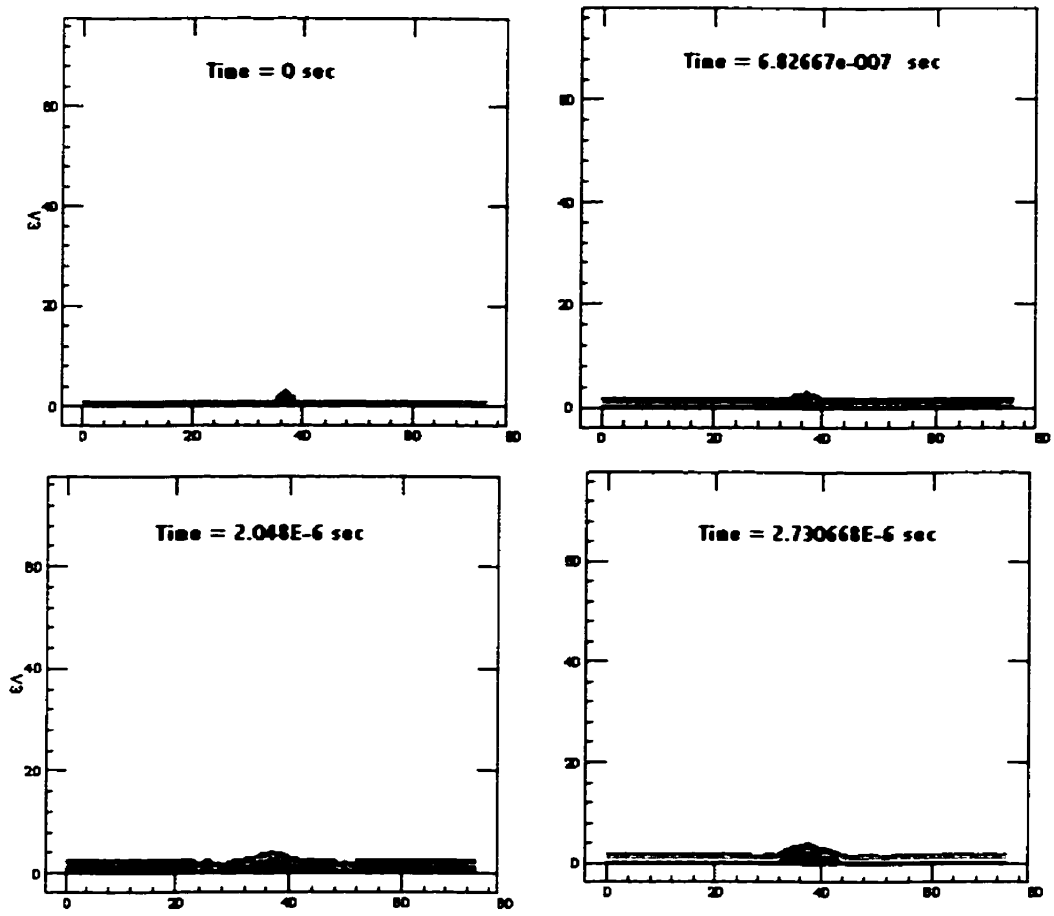


Figure 4-1: Grid Size at 40 times smaller than the theoretical tip radius.

The results of this test indicate that the cell size was too small for the growth to occur. The cell size for this run was 1.28e-008m. The time step computed according the Fourier number was: 6.2866E-11 sec. Thus, the changes in phase field are so insignificant that the growth is controlled by numerical uncertainty rather than by the physics of the problem. As seen in Figure 4-1 in time the perturbation starts to disappear. This does not mean that a cell size of 1.28E-008 can not be used, it just means that the initial nucleus is

smaller than the initial critical radius required for the perturbation to be unstable as described in Chapter 2. Under the same cell size with a larger initial nucleus the dendrite would have grown.

Our second run was performed with a cell size that is approximately 20 times smaller than the theoretical tip radius under the same conditions. The results of the phase evolution are presented in figure 4-2.

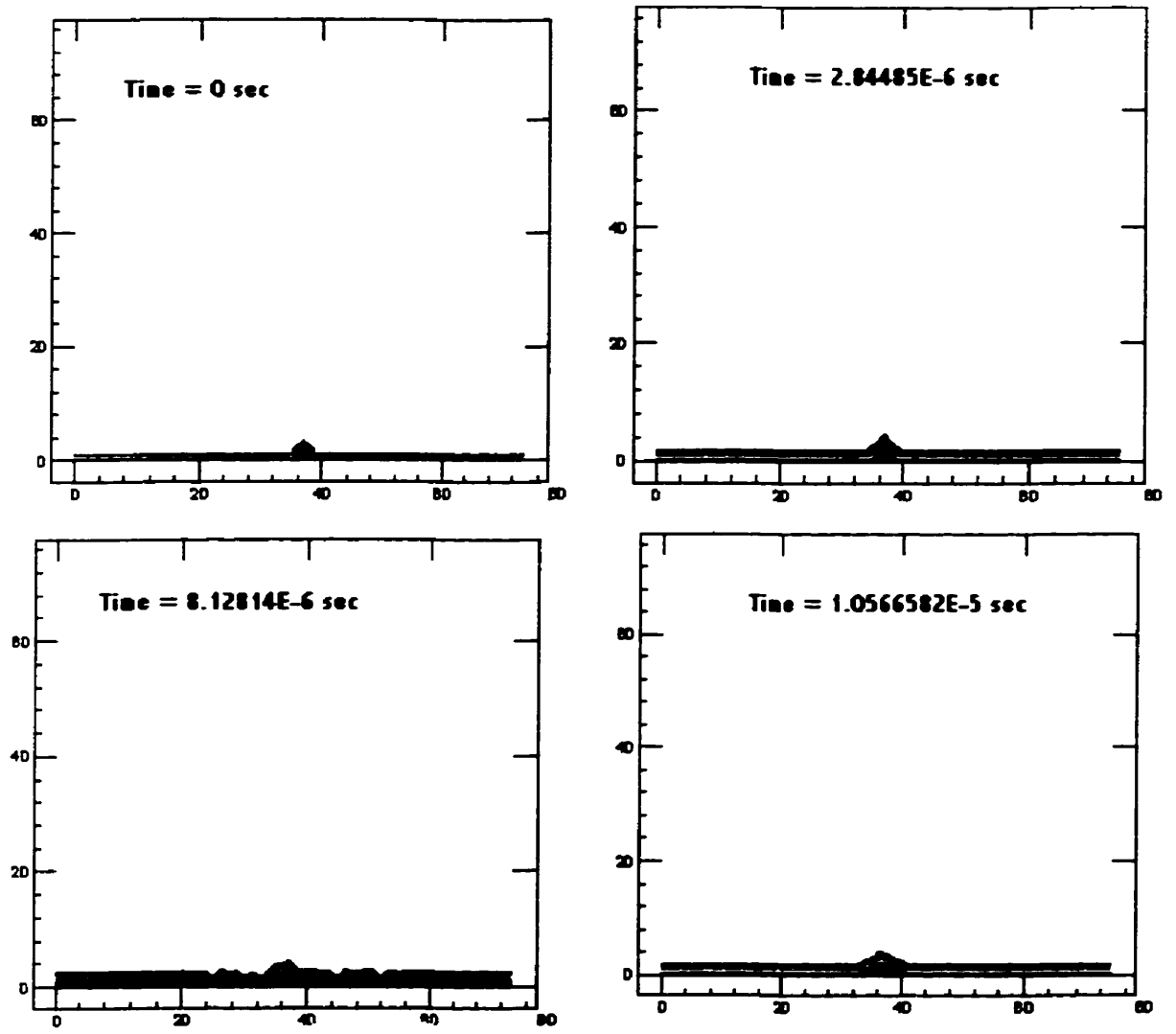


Figure 4-2: Cell size at 20 times smaller than the theoretical tip radius

Under a cell size of 20 times smaller than the tip radius predicted by Fisher and Kurz model, the cell is approximately 2.55E-8 m with a time step of 2.70938e-010 sec. This size of cell size was also found to be too small for the nucleus to grow.

When the cell size was set to be approximately 10 times smaller than the theoretical tip radius the results where found to be more interesting. At this cell size, the dendrite grew and it was possible to obtain pictures of the morphology and concentration profiles in 3-D. In order to analyze the tip radius a 2-D cut through the middle of the dendrite was performed and the curvatures at the tip were measured. Figure 4-3 presents the dendrite morphology as it evolved in time.

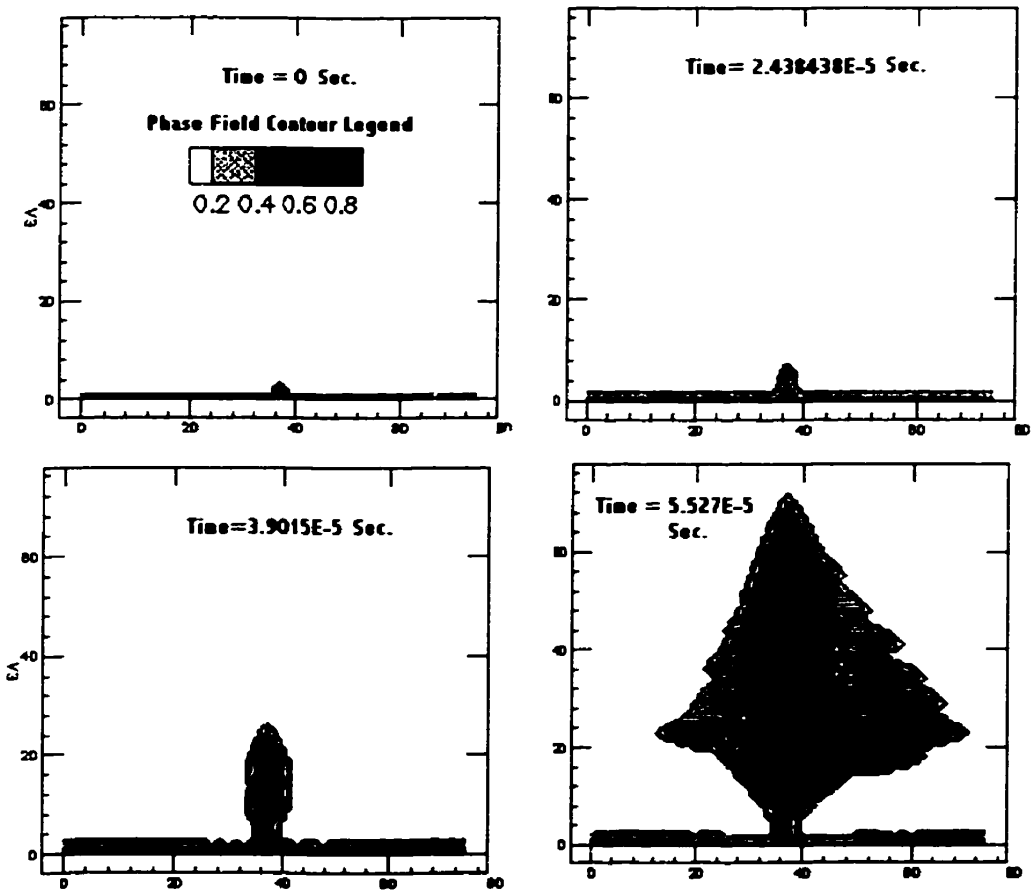


Figure 4.3: Dendrite growth with a cell size 10 times smaller than the theoretical radius

Figure 4-3 shows a 2-D cut of the dendrite morphology of Aluminum 5% Silicon 5% Copper alloys. The effective supersaturation taking into account both solutes is 0.11. The effective superstauration is obtained using equation 4-2:

$$\Omega_{Si\_eff} = \frac{C_{Si\_eff} - C_{Si\_liq}}{C_{Si\_liq}(1-k)} \quad (4\text{-}2)$$

where

- C_{si_eff} is the effective concentration of Silicon
- C_{si_liq} is the liquid concentration at the current temperature of the system
- k is the distribution coefficient [2].

The effective silicon concentration is computed as follows:

$$C_{Si\_eff} = C_{Bulk\_si} + C_{Bulk\_cu}\left(\frac{m_{l\_cu} \cdot Mgap\_cu}{m_{l\_si} \cdot Mgap\_si}\right) \quad (4\text{-}3)$$

where

- C_{bulk_Si} is the concentration of silicon at time zero far away from the solidifying solid-liquid interface
- C_{bulk_cu} is the initial concentration of copper far away from the solid-liquid interface at time step zero
- m_{l_Cu} the liquidus slope of copper (straight line approximation)
- m_{l_Si} is the liquidus slope of Si (straight line approximation)
- $mgap\_cu$ is the copper miscibility gap at the current system temperature.

- $mgap_{si}$ is the silicon miscibility gap at the current system temperature.

From Figure 4-3, we can observe that in the initial growth (approximately 0.2 microseconds) the dendrite tip takes a very needle shape like morphology. As the growth continues the tip essentially changes very little, but the sides of the primary trunk become bigger. The sides of the primary trunk provided sites for interface instability that lead to the formation of secondary arms. At approximately 0.6 microseconds, the dendrite has a very well defined morphology with primary secondary and possible sites for the formation of tertiary arms. The tertiary arms in our model are observed when the control volume is bigger. The model provides results that describe very well the coarsening effects of the secondary arms (unification of small secondary arms to form one big secondary arm). This is consistent with experimental results.

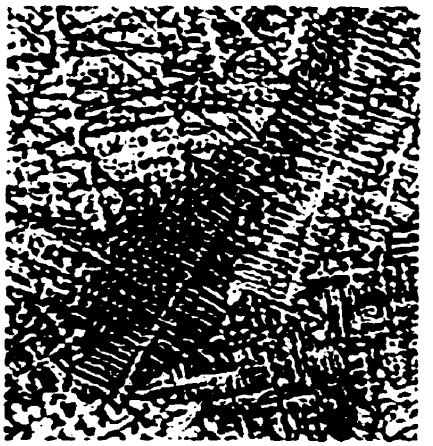


Figure 4-4: Actual dendrite morphology

As it is seen in Figure 4-4 the biggest secondary arms are found at the bottom part of the trunk (lower portion of the figure), while at the top very few secondary arms are encountered. Figure 4-5 shows a three-dimension representation of a single dendrite.

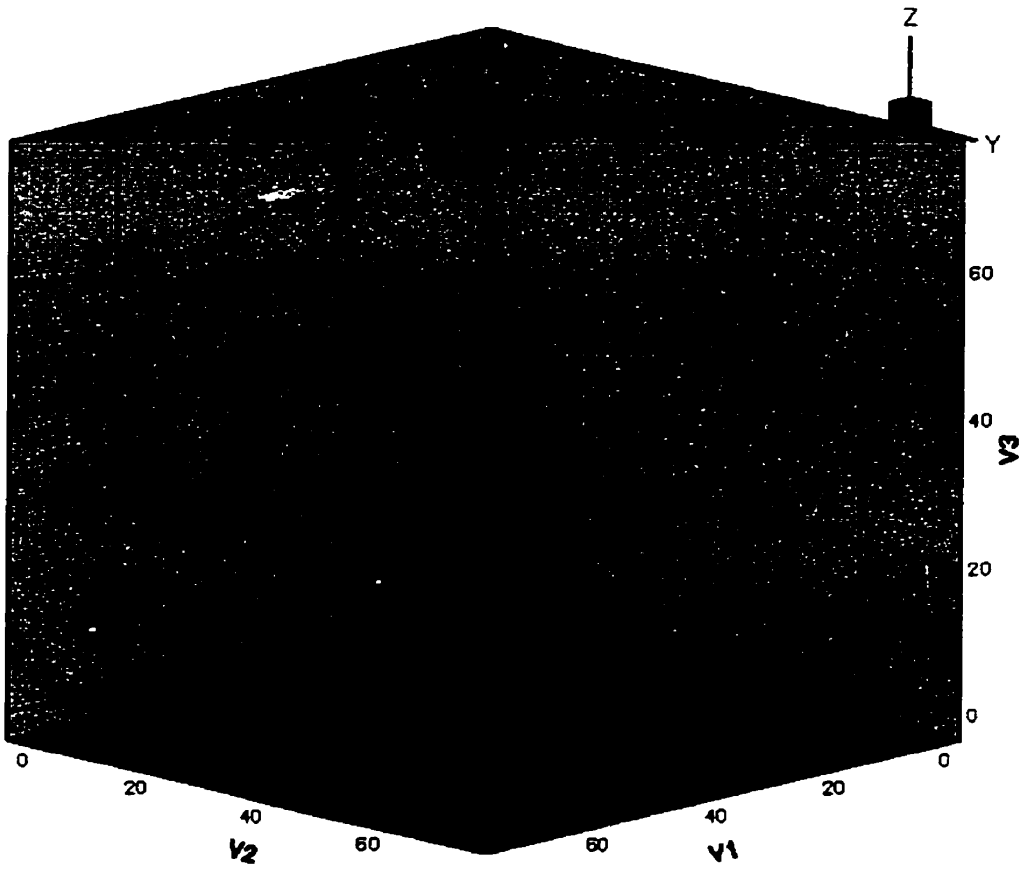


Figure 4-5: 3-D dendrite morphology.

Our computer model as part of its output file provides the concentration profiles that we can graph. These profiles are presented in figure 4-6.

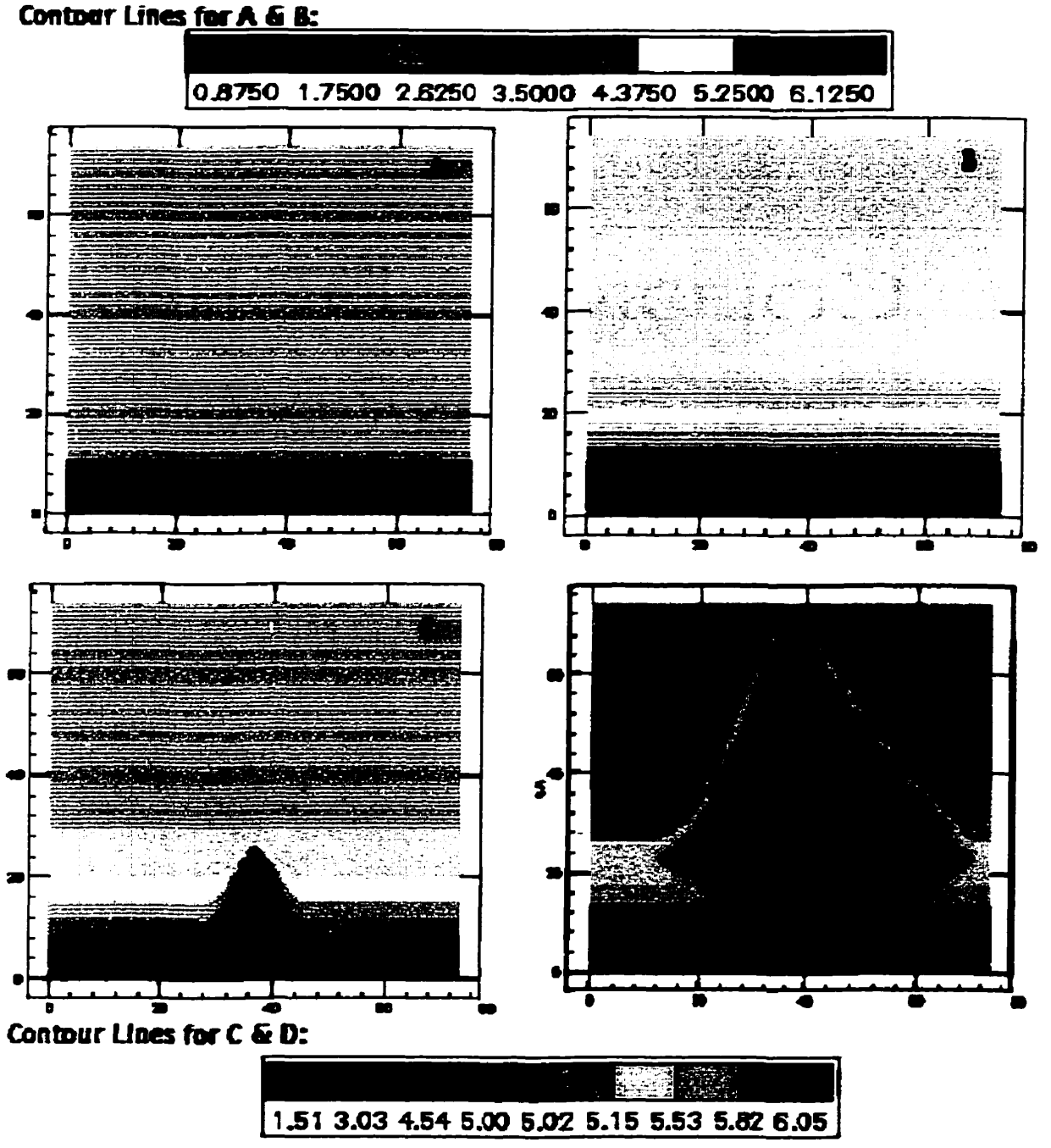


Figure 4-6 Silicon Concentration Profiles

At time zero a layer of 7 %wt. silicon was set on the first 10 cell layers, as seen in figure 4-6A. This was done to avoid abrupt changes in the phase field during the transient period. If this is not done, the flat surface around the dendrite could develop small

perturbations that will lead to other dendrite formation near the perturbation set by the model. These small perturbations can grow at a very fast rate and influence the dendrite morphology of the dendrite we are interested in. By assigning a concentration layer near but not exactly the liquidus concentration of the alloy we slow down the transient growth period. This is done because the initial concentration layer is required to diffuse before solute is removed from the solid-liquid interface, allowing then, the growth of the nucleus. The bulk concentration was set to 5 %wt. Silicon while the initial concentration layer was set to 7% wt. Silicon. Thus, during the first iterations the concentration layer is diffused from an area of high concentration (7% Silicon) to an area of low concentration (the bulk area set to 5% Silicon). As the cells are solidified, they are assigned a solidus concentration value at the system temperature. The solidus concentration value is calculated by approximating the solidus line of the of the aluminium silicon phase diagram by a straight line. Near the dendrite tips the concentration contour lines are a lot closer to each other. The proximity of these lines indicates the regions of higher concentration gradients. In these regions of high concentration changes is where the growth occurs faster due to faster removal of solutes. The concentration profiles of copper are very similar to those of silicon. The concentration profiles of copper are shown in figure 4-7.

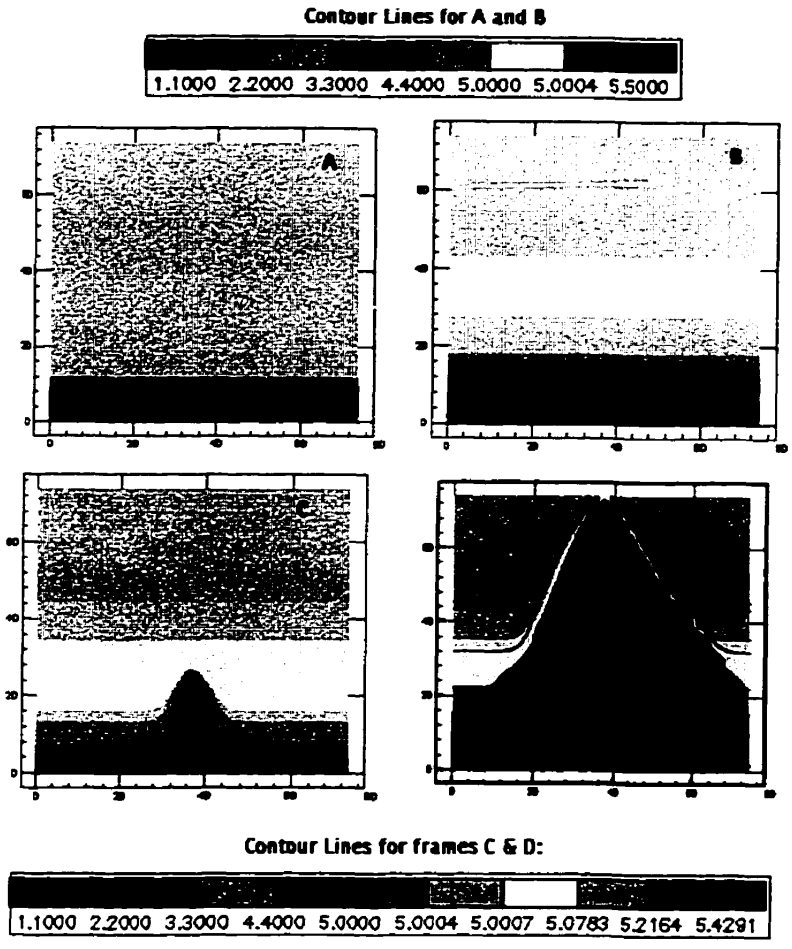
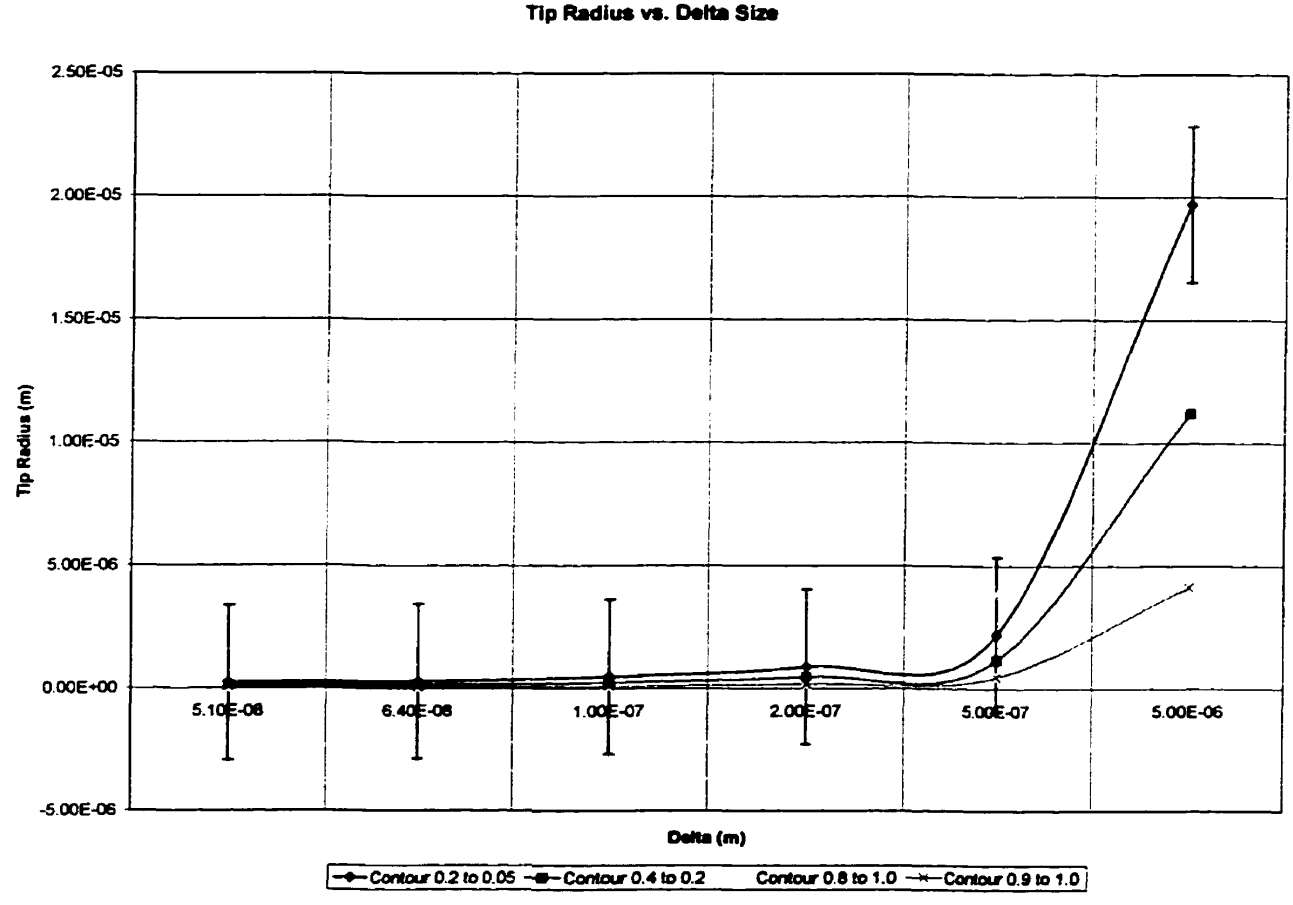


Figure 4-7: Copper Concentration Profile

The concentration profiles of copper have the same shape of curves as the concentration profiles of silicon. The initial concentration layer at time zero is of approximately 5.5%wt copper while the rest of the system was set to have a bulk concentration value of 5%wt copper. Figure 4-7D shows concentration contour lines. An interesting fact occurs at the top with the concentration contour lines which is that they go through the roof of the control volume, while at the side walls they are at the same height. This is due to the periodical boundary conditions at the sidewalls of the control volume. If the dendrite growth had been started from a homogeneous nucleus (at the center of the control

volume), then all the walls would have been setup with a periodical boundary condition. The roof and the floor of the control volume are setup with a constant concentration value of 5%wt copper.

Dendrite morphologies are not as difficult to obtain as one would initially think. However, to obtain dendritic morphologies that actually are produced by proper mathematical description of the physics behind the problem is a challenge all in itself. For this reason a series of tests, were performed in order to assure that the dendritic tip radius obtained from our model was independent of the grid size. The results are presented in the following graph.



Graph 4-1: Tip Radius vs. Grid Size

The results shown on graph 4-1, indicate that the tip radius at diverse contour levels all converge to a single value as the grid size is reduced. When the grid size is approximately equal to the theoretical value predicted by Fisher and Kurz Model the tip radius increases exponentially. The tip radius at several contour levels where measured using a graphical visualization tool (Techplot).

The numerical data presented in Figure 4-7 is presented in Table 4-1:

Table 4-1: Grid Size Vs Tip Radius data at different contour lines.

	Grid Size	0.2 to 0.05	0.4 to 0.2	0.8 to 0.4	1.0 to 0.8	1.0 to 0.9
10 Greater	5.00E-06	1.971E-05	1.120E-05	6.422E-06	4.572E-06	4.148E-06
Equal to	5.00E-07	2.206E-06	1.160E-06	6.735E-07	4.880E-07	4.607E-07
2.5 Smaller	2.00E-07	8.850E-07	4.740E-07	2.740E-07	1.860E-07	1.770E-07
5 Smaller	1.00E-07	4.677E-07	2.356E-07	1.334E-07	9.678E-08	9.206E-08
8 Smaller	6.40E-08	2.842E-07	1.483E-07	8.472E-08	6.045E-08	5.660E-08
10 Smaller	5.10E-08	2.190E-07	1.130E-07	6.600E-08	4.670E-08	4.460E-08

Note: All values in meters.

A comparison of the percentage difference between the cellular model at marginal stability and Fisher and Kurz Model is presented in Table 4-2.

Table 4-2: Parabolic Model vs. Cellular Model.

Grid Size	Marginal Stability	Cellular Model	% Difference
	Parabolic Tip Radius	**Tip Radius**	
10 Greater	5.10E-07	4.1478E-06	-713.30
Equal to	5.10E-07	4.6071E-07	9.66
2.5 Smaller	5.10E-07	1.7700E-07	65.29
5 Smaller	5.10E-07	9.2056E-08	81.95
8 Smaller	5.10E-07	5.6596E-08	88.90
10 Smaller	5.10E-07	4.4600E-08	91.25

Note: All tip radius in meters.

As it is seen in Table 4-2, all values are within 100% of the theoretical parabolic tip radius. The closest measurement of tip radius occurs when the grid size is equal to the tip radius. One of the differences with Fisher and Kurz model (Parabolic tip radius) is that they assume that the tip of the dendrite preserves its parabolic shape. This is done for simplification reasons, since the parabolic equation preserves its parabolic shape as the dendrite grows. A dendritic tip radius does not necessarily preserve a parabolic tip.

Another test that was performed to determine that our model was in agreement with the correct physical description was to check that the law of conservation of mass was satisfied. This will be described in more detail in the next section.

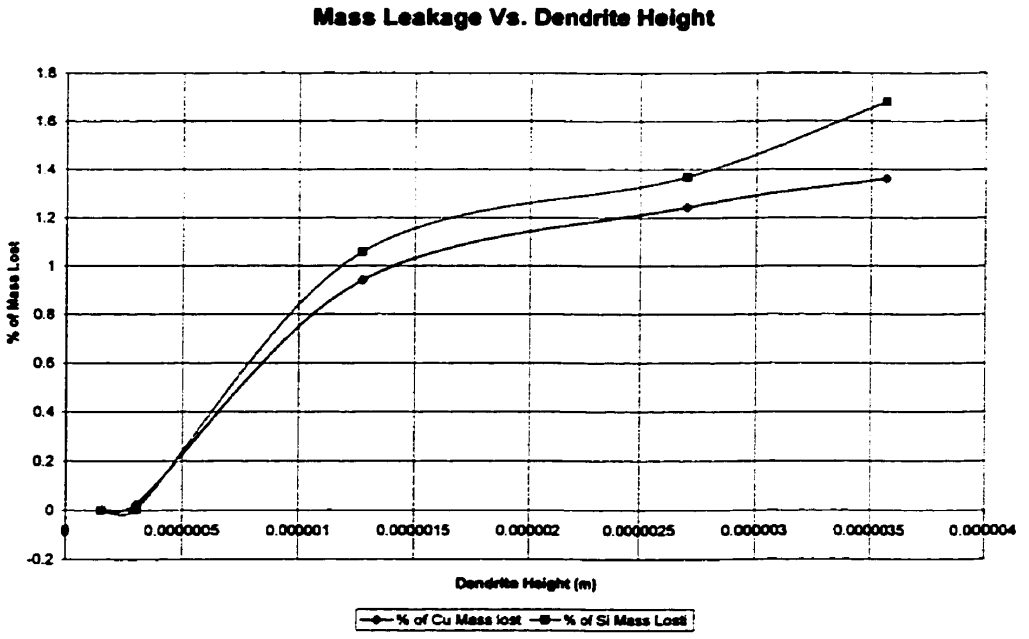
4.2 Conservation of Mass

The law of conservation of mass states that for a fixed control volume the rate of mass with respect to time is invariant. Thus, the total mass in the system at time zero is equal to the total mass at time infinite as long as the control volume under study remains closed and no leaks occurs.

In our model we created a conservation of mass function. At time step zero the conservation of mass function takes the liquid concentration of silicon and multiplies it by liquid fraction in each cell. At the same time step we take the solid concentration of each cell and multiply it by the solid fraction of each cell, the summation of these two terms provides us with the total concentration of silicon and copper in the system. Mathematically this can be expressed as:

$$Concentration\_total = \sum_{ijk=0}^{ijk=Max} g_{ijk}^{l} \cdot k \cdot C_{ijk\_liq}^{l} + (1 - g_{ijk}^{l}) \cdot C_{ijk\_liq}^{l} \quad (4\text{-}4)$$

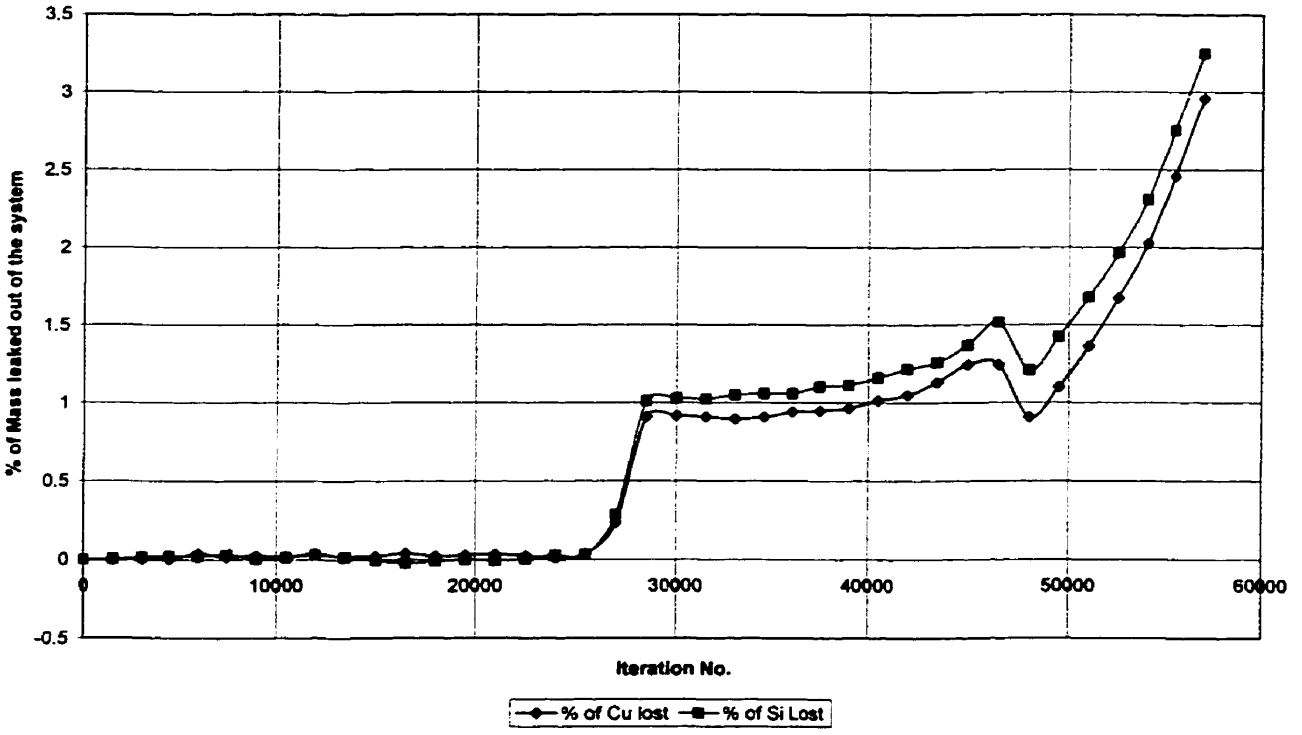
Every time the model saves the information to the hard disk, it performs the summation expressed by equation 4-4. We performed this test for a cell size 10 times smaller than the theoretical tip radius estimated by Fisher and Kurz model. The results of this test are presented graphed in graph 4-2.



Graph 4-2: Conservation of mass vs. dendrite height

Graph 4-2, indicates that the total mass is preserved within 1.5 % of the total initial mass within the control volume. As the dendrite grows we start having mass leakage and some mass is lost due to the boundary conditions established at the roof and bottom of the control volume.

Graph 4-3 shows the total mass leaked out of the system vs. the number of iterations performed by the model to develop a complete dendrite.



Graph 4-3: Mass Leakage vs. No. of Iterations

In graph 4-3 it is possible to see how as the number of iterations increase the percentage of mass leaked out of the system increases exponentially. At less than thirty thousand iterations, the percentage of mass leaked is due to rounding off errors. Between thirty thousand and fifty thousand iterations the concentration profiles start going through the roof of the control volume. At approximately fifty thousand iterations, the dendrite tip starts going through the roof of the control volume, which is when the system starts loosing a lot of mass and thus breaking the law of conservation of mass. This is not of concern because the tip radius is measured before the concentrations profiles start trespassing the roof of the control volume.

Another parameter that was required to analyze was the growth velocity of the dendrite. The results are described in more detail in the next section.

4.3 Growth Velocity

An important parameter that was verified to determine that the growth is consistent with Fisher and Kurz model was the growth velocity of the dendrite. The growth velocity is described by a non-dimensional parameter known as the Peclet number:

$$Pc = \frac{v \cdot r_{tip}}{2 \cdot D} \quad (4\text{-}5)$$

where

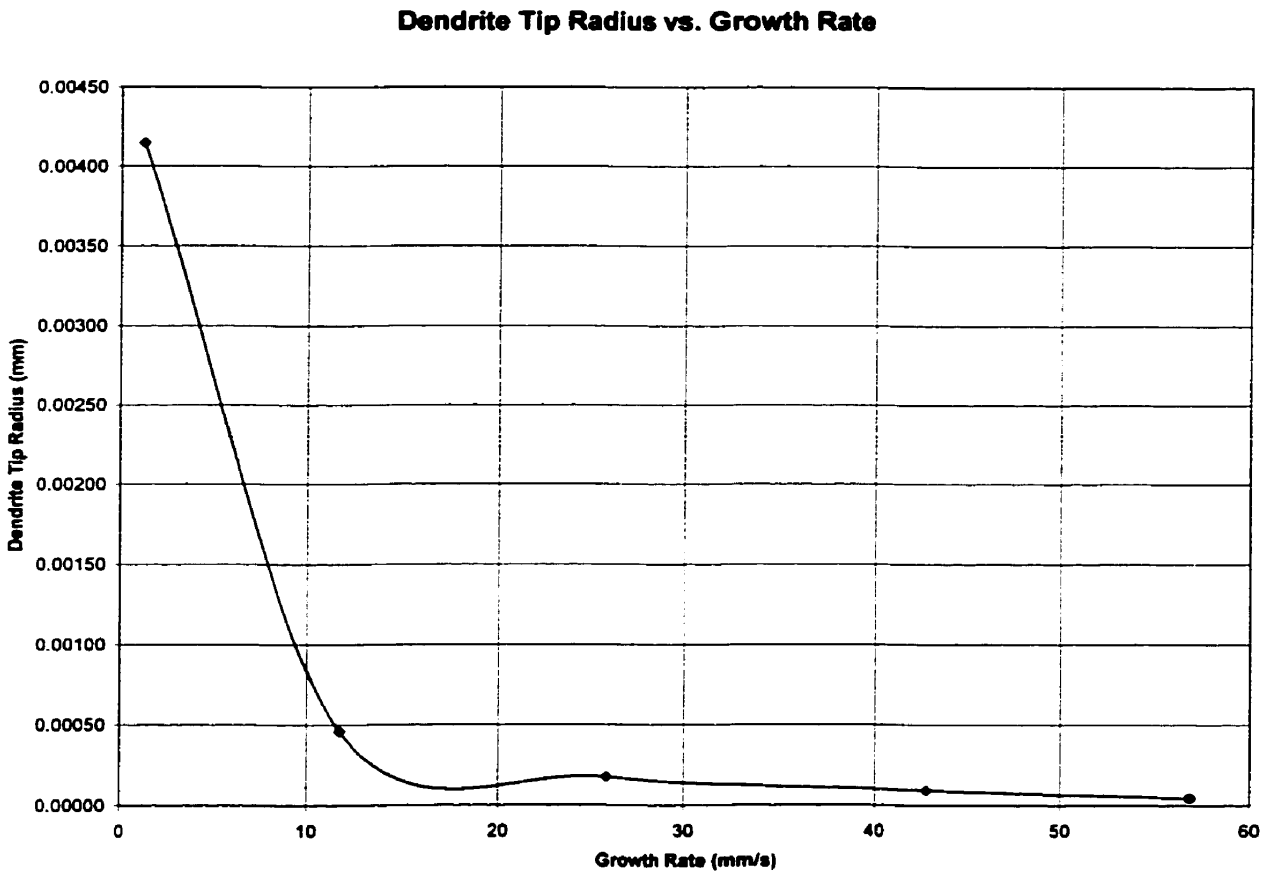
- r_{tip} is the tip radius in (meters)
- D is the diffusion constant (m2/s)
- v is the velocity (m/s) [2].

According to Fisher and Kurz's model the peclet number for an effective supersaturation of 0.11 (5%wt. Silicon, 5% wt. Copper and 90% Al) is 0.11. Table 4-3 describes the relationship of the Peclet number for a grid size that is approximately ten times smaller than the theoretical tip radius.

Table No. 4-3

Approximately 10 times smaller grid size than the theoretical radius							
Time	Iterations	Grid Size	Contour Level	Tip Radius	Distance	Velocity	Peclet Number
(sec)		(meter)		(m)	(m)	(m/s)	
5.20E-05	48000	5.10E-08	0.2 to 0.05	2.19E-07	2.96E-06	0.057	2.08
			0.4 to 0.2	1.13E-07	2.96E-06	0.057	1.07
			0.8 to 0.4	6.60E-08	2.96E-06	0.057	0.63
			1.0 to 0.8	4.67E-08	2.96E-06	0.057	0.44
			1.0 to 0.9	4.46E-08	2.96E-06	0.057	0.42

From Table 4-3, it is possible to see that the peclet number varies significantly between contour levels for the same dendrite. This is because the tip radius varies significantly with the fraction of solid in each cell. The velocity presented in table 4-3 is an average velocity. It is not the local velocity at the tip. The average velocity is obtained by determining the height of the nuclei and the height of the dendrite at the time step that the dendrite was well developed but not touching the top wall of the control volume. This provides us the total distance grown by the dendrite over a period of time. Dividing the total grown distance by the total time provides us with an average velocity. The tip radius is also the average radius at the tip of the dendrite. Techplot outputs at a specified contour level the approximate curvatures at specified number of points (set by the user) on the solid-liquid interface from which we can calculate the average tip radius. These two parameters and the diffusion constant provide the peclet number. The peclet number provided by our model is considerable larger than that provided by Fisher and Kurz model. One possible cause of this difference between the peclet number determined by our model and that provided by Fisher and Kurz is that our tip radius is not assumed to be of a parabolic shape.



Graph 4-4: Dendrite tip radius vs. growth rate

Graph 4-4 shows the relationship of the growth rate of the dendrite with respect to its tip radius. As seen in this figure as the growth rate decreases the dendrite tip increases.

The last chapter of this thesis presents the conclusion of our research and results. It also presents some possible future work required to obtain a better understanding of the results and possible improvements to the model.

Chapter 5: Conclusion & Future Work

5.0 Introduction

Chapter 5, is the final chapter of this thesis. This chapter presents the conclusions of our work on dendrite growth using the cellular model. The chapter outlines possible areas that could be examined to increase the accuracy and performance of our dendrite growth model.

5.1 Conclusions

One of the important aspects of our research was the use of the cellular model to simulate the growth of a dendrite. The results presented in this thesis allow us to conclude that the cellular model is a feasible way to simulate dendrite growth in three dimensions. The size of the control volume that we simulate is limited by the computer system running the simulation and not by the complexity of the solid-liquid interface of a dendrite. The results show important results such as the formation of secondary, and in some cases, tertiary arms. The correct choice of cell size is also of great importance and has to be carefully picked by means of simpler analytical techniques before valuable computer time is wasted. It was shown that simulations with cell sizes that are larger than the theoretical tip radius predicted by Fisher and Kurz model can also reproduce dendrite morphologies with a tip radius that is in complete disagreement with theoretical and experimental results.

Another important aspect of the results presented by this thesis is how the model can predict the concentration profiles around the dendrite for a tertiary system. This is important because it allows us to see the influence of a third alloying component on the growth of the dendrite (solute undercooling).

One of the most challenging aspects of the model was in determining the curvature of the solid-liquid interface. The use of the "correct" weight factors for determining the average phase field from which curvature of the solid-liquid interface is calculated showed the influence that capillary undercooling has on dendrite growth. The results show that the use of the "sphere test" in the determining the correct weight factors is not necessarily the best test to use, since it can produce weight factors that produce average solid-liquid interfaces that are not representative of the actual interface morphology.

This thesis showed how anisotropy plays an extremely important role on dendrite tip morphology. Not taking anisotropy into account would have resulted on the splitting of the dendrite tips, thus prohibiting the comparison of our model to Fisher and Kurz, or other analytical solutions.

5.2 Future Work

The mathematical and computational model described in this thesis can be improved in many ways. In this last section of the thesis it is out intention only to mention a few of the ways that the model can be improved.

The curvature function can be substantially improved by determining if the weight factors used are the most optimum in the analysis of dendrites. The development of curvature test that describe the curvature not only of spheres but of dendrite-like structure for which the exact curvature is known could be developed and set as a standard problem set to serve as a guide line in determining the accuracy of curvature functions. Other methods (mathematical approximation) for calculating curvature could be easily implemented into our model by replacing the curvature function in Phase Evolution.

The model can be modified to take into account temperature variations through the development of the dendrite. Runs where the temperature is not maintained constant but that is constantly dropping at a fix rate can be easily implemented. The use of a constant temperature drop in the model would provide results on dendrite tip that are more realistic with what is seen in the real physical world.

Further analysis of secondary arm spacing can be also examined. Analytical results provided by Fisher and Kurz can serve as a basis of comparison to the results provided by our model.

At some point in time, during the development of future work, experimental data that provides physical results that validate our computer model will be required. Experimental analysis will provide insights into other physical parameters that are required to be modeled and that could improve the accuracy of our computational models.

Bibliography

1. Kurz and Fisher, Fundamental of Solidification 3rd Edition, Trans Tech Publications, 1989.

2. A. Artemev, Computational Metallurgy Course Notes, Lecture 2. Unpublished.

3. Goujian Ma, Modeling Equiaxed Dendritic Solidification, 1990

4. Lawrence H. Van Vlack, Elements of Materials Science and Engineering 6th Edition, 1989

5. A. Artemev and J. A. Goldak, Cellular Simulation of the Dendrite Growth in Al-Si Alloys, 1995.

6. English Dictionary, www.dictionary.com, Webster's Revised Unabridged Dictionary

7. A. Artemev, Dendrite Simulation Conference.

8. A. A. Wheeler, W. J. Boettinger and G. B. McFadden, Phase-field model for isothermal phase transition in binary alloys, Physical Review A, Volume 45 No. 10, 1992

9. Karma, A. & W.J. Rappel 1996. Phase-field method for computationally efficient modeling of solidification with arbitrary interface kinetics. Phys. Rev. E53: R3017-R3020

10. Karma, A. & W.J. Rappel 1996. Numerical Simulation of three-dimensional dendritic growth. Phys. Rev. Lett. 77: 4050-4053.

11. J.P. Holman, Heat Transfer 7th Edition, Mc Graw Hill, 1990

12. Jacobs and Kilduff, Engineering Material Technology, 3rd Edition, Prentice Hall, 1994

13. R. Trevedi, V. Seetharaman, and M.A. Eshelman, The Effects of Interface Kinetics Anisotropy on the Growth Direction of Cellular Microstructures, Metallurgical Transactions A, Volume 22A, February 1991

Appendix A

Appendix A

Fisher and Kurz Model of Dendrite Growth in Undercooled Alloy Melts.[1]

1. Introduction

The objective of this appendix is to describe and explain the mathematical relationship that describes the evolution of a dendrite tip in an alloy. Fisher and Kurz's model of dendrite growth is used as a comparison and guideline to the results obtained by the cellular model (Phase Evolution Program) described in this thesis.

2. Extremum vs. Marginal Growth

Figure B-1 shows the relation of the growth rate vs. tip radius. This figure shows the sum of the capillary and diffusion effects. As it is seen in figure B-1, the figure shows a maximum point indicating the existence of a maximum growth rate with a specific tip radius (R_c). This value of tip radius (R_c) represented a unique solution in determining the value of the tip radius of a dendrite, since the derivative of the growth equation set to zero would provide a unique solution.

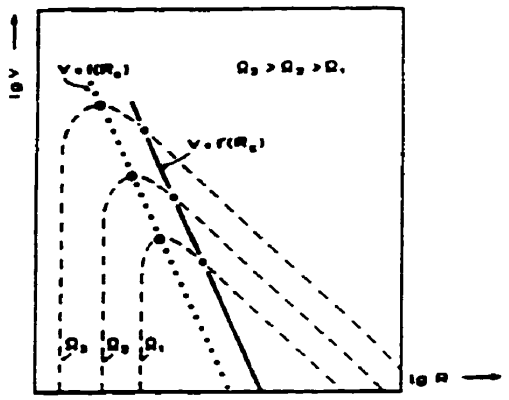


Figure B-1: Growth Rate vs. Tip Radius

It was not until 1977 when Muller and Krumbhaar argued that the dendrite grows at the limit of stability. This is represented in Figure B-1 as the R_s point. This stability criterion is known as the marginal stability criterion. Mathematically it is represented by equation (B-1):

$$R_s = \lambda_i \quad \text{(B-1)}$$

where

- R_s is the tip radius at the marginal stability criterion,
- λ_i is the stability limit.

Fisher and Kurz's model compute the total undercooling as a function of the thermal, solute and capillary undercooling. Thus, in order to find the tip radius according to the marginal stability criterion one would need to determine the total undercooling. The total undercooling is described by equation B-2:

$$\Delta T = \Delta T_t + \Delta T_c + \Delta T_r \quad \text{(B-2)}$$

where

- ΔT_t is the thermal undercooling,
- ΔT_c is solute undercooling,
- ΔT_r is the curvature undercooling and
- ΔT is the total undercooling

The thermal undercooling in Fisher and Kurz model is represented as a function of the Ivanstov solution for a paraboloid dendrite tip:

$$\Delta T_t = \theta_t I(P_t) \quad \text{(B-3)}$$

where

- $I(P_t)$ is the Ivanstov function.
- P_t is the thermal Peclet number
- θ_t is the thermal undercooling ($\Delta h_f/c$, where c is the volumetric specific heat and Δh_f is the latent heat of fusion per unit volume.)

The Ivanstov solution is mathematically expressed as:

$$I(P) = \cfrac{P}{P + \cfrac{1}{1 + \cfrac{1}{P + \cfrac{2}{1 + \cfrac{2}{P + \ldots}}}}} \quad \text{(B-4)}$$

where

- P is the Peclet number

Similarly the solute undercooling is described mathematically as:

$$\Delta T_c = mC_o \cdot (1 - A(P_c)) \quad \text{(B-5)}$$

where

- Co is the initial alloy concentration
- m is the liquidus slope
- $A(P_c) = (1\text{-}pI(P_c)]^{-1}$

The last term in equation B-3 is the capillary undercooling. The capillary undercooling is determined using equation B-6:

$$\Delta T_r = \frac{2 \cdot \Gamma}{R} \quad \text{(B-6)}$$

where

- Γ is Gibbs Thomson coefficient
- R is the tip radius.

Equation B-2 to B-6 defines the total undercooling. These equations are not enough to find a unique solution to the problem of determining a specific radius to a specific total

undercooling value. This is where the stability criterion plays an important role. The stability criterion developed by Langer and Muller-Krumbhaar provide a solution which results are found to be very close to experimental results.

Equation B-1, proposed that the tip radius is equal to the wavelength of a perturbation in a planar solid-liquid interface. The stability limit of the perturbation is described by equation B-7.

$$\lambda_i = 2 \cdot \pi \cdot \left(\frac{\Gamma}{\phi} \right)^{\frac{1}{2}} \quad \text{(B-7)}$$

where

- Γ is the Gibbs Thomson Coefficient
- ϕ is the degree of constitutional undercooling.

Using equation B-1 and B-7 one determines that the tip radius of the dendrite is computed as:

$$R = 2 \cdot \pi \cdot \left(\frac{\Gamma}{mG_c - \overline{G}} \right)^{\frac{1}{2}} \quad \text{(B-8)}$$

where,

- mG_c is the liquidus temperature gradient
- G is the imposed temperature gradient.

As seen in equation B-8 the degree of constitutional undercooling is the difference between the liquidus temperature gradient (mGc) and the temperature gradient imposed at the interface (G). Thus, as long as the liquidus temperature gradient is greater than that of the temperature gradient imposed at the interface the perturbation will grow.

A flux balance determines the concentration gradient (Gc) in front of a perturbation. Mathematically this flux balance is shown in equation B-9:

$$V \cdot C_l \cdot (1-k) = -D \cdot G_c \quad \text{(B-9)}$$

where

- k is distribution coefficient,
- Cl is the liquid concentration at the solid-liquid interface (perturbation),
- V is the growth velocity of the perturbation,
- D is the diffuisivity constant,
- Gc is the concentration gradient.

Using the Peclet number and isolating velocity in terms of the Peclet number one obtains:

$$V = \frac{2 \cdot D \cdot P_c}{R} \quad \text{(B-10)}$$

where

- D is the diffusion constant,
- Pc is the Peclet number

- R is the tip radius.

Inserting equation B-10 into equation B-9, one then obtains the concentration gradient ahead of the tip radius as:

$$G_c = \frac{-2 \cdot P_c \cdot (1-k) \cdot C_l}{R} \quad \text{(B-11)}$$

where,

- C_l is the liquid concentration which can be expressed as a function of the solute supersaturation (C_l = Co/(1-Supersaturation(1-k)))
- R is the tip Radius,
- P_c is the Peclet number
- K is the distribution coefficient.

Now that the concentration gradient ahead of the tip radius, the temperature gradient ahead of the dendrite tip has to be also defined. The temperature gradient has the same structure as that presented by equation B-11. Thus, the temperature can be mathematically expressed as:

$$G_t = \frac{-2 \cdot P_t \cdot \Delta h_f}{c \cdot R} \quad \text{(B-12)}$$

where

- P_t is the thermal Peclet number
- R is the tip radius,

- C is the volumetric specific heat
- Δh_f latent heat of fusion per unit volume.

The temperature gradient is determined not only on the liquid phase but also at the solid phase. Assuming a similarity in the thermal conductivity between the solid and the liquid phase ($K_s = K_l$), and assuming that Ivantsov dendrite is isothermal it is determined that G (bar) is half of the temperature gradient in the liquid phase ahead of the interface.

Thus the tip radius can be expressed as:

$$R = \frac{\frac{\Gamma}{\frac{1}{4}\cdot\pi^2}}{\phi_t P_t - 2\cdot P_c \cdot m \cdot C_o \cdot (1-k)\cdot A(P_c)} \quad \text{(B-13)}$$

where

- $A(P_c) = 1/[1-(1-k)I(P_c)]$
- K is the equilibrium distribution coefficient,
- M is the liquidus slope
- P_c is the concentration Peclet number
- P_t is the thermal Peclet number
- θ_t is the thermal Peclet number
- C_o is the is the initial alloy concentration
- Γ is the Gibbs Thomson coefficient

From analyzing equation B-13, the only unknowns are the Pt and Pc. The solute Peclet number and the thermal Peclet number are related by the ratio of the thermal to mass transport diffusion constant as shown in equation B-14.

$$Pc = Pt \cdot \frac{a}{D} \quad \text{(B-14)}$$

where,

- a is the thermal diffuisivity,
- D is the mass diffusion constant.

The only unknown is then the product of VR (Peclet number). The velocity and the tip radius are related as shown in Figure B-1, thus for every velocity there is a specific tip radius for a given supersaturation. Experimental results show that the tip radius and the velocity occur at the marginal stability criterion or close to that point.

Appendix B

Input Parameters and Results of Andreis Mathematica code:

Input Parameters:

Temperature := 873.– 273.15 In Celsius

$m_{l\_cu} := -2.6$ Copper Liquidus Slope

k := 0.13 Distribuition coefficient

$m_{l\_si} := -6.59$ Silicon Liquidus Slope

$C_{o.cu} := 5$ Bulk Concentration of Copper

$C_{o.si} := 5.$ Bulk Concentration of Silicon

$m_{s\_si} := -50.30$ Silicon Solidus Slope

$m_{s\_cu} := -19.85$ Copper Solidus Slope

$D := 3 \cdot 10^{-9}$ Diffusivity Coefficient

$Gibbs\_thomson := 2.0 \cdot 10^{-7}$

Calculation of %wt of Si:

In Liquid state:

$$X_{Liq\_si} := \frac{(Temperature - 660)}{m_{l\_si}} \qquad X_{Liq\_si} = 9.127$$

$$X_{Liq\_cu} := \frac{(Temperature - 660)}{m_{l\_cu}} \qquad X_{Liq\_cu} = 23.135$$

In Solid State:

$$X_{Sol\_si} := \frac{(Temperature - 660)}{m_{s\_si}} \qquad X_{Sol\_si} = 1.196$$

$$X_{Sol\_cu} := \frac{(Temperature - 660)}{m_{s\_cu}} \qquad X_{Sol\_cu} = 3.03$$

Calculation of Miscibility gap:

$Miscb\_gap\_si := X_{Liq\_si} - X_{Sol\_si}$ $\qquad Miscb\_gap\_si = 7.932$

$Miscb\_gap\_cu := X_{Liq\_cu} - X_{Sol\_cu}$ $\qquad Miscb\_gap\_cu = 20.104$

Calculation of Effective Bulk Concentration:

Input Parameters:

$$C_{Si\_effective} := C_{o.si} + C_{o.cu} \cdot \frac{(m_{l\_cu} \cdot Miscb\_gap\_cu)}{(m_{l\_si} \cdot Miscb\_gap\_si)}$$

$C_{Si\_effective} = 10$

Calculation of Supersaturation for Silicon & Cu:

$$Supersaturation\_si := \frac{(X_{Liq\_si} - C_{o.si})}{X_{Liq\_si} \cdot (1 - k)}$$

$$Supersaturation\_cu := \frac{(X_{Liq\_si} - C_{o.cu})}{X_{Liq\_cu}\cdot(1-k)}$$

Supersaturation_si = 0.52

Supersaturation_cu = 0.205

Effective Supersaturation:

$$Supersaturation\_effective := \frac{(C_{Si\_effective} - X_{Liq\_si})}{X_{Liq\_si}\cdot(1-k)}$$

Supersaturation_effective = 0.11

Changing from Non Dimensional values to dimensional values:

Input parameters from Andreis Code:

$R := 133.247$

Non Dimensional Tip Radius

$$Curv := \frac{1}{R}$$

$Curv = 7.505 \cdot 10^{-3}$

Non Dimensional Curvature value

$V := 0.00165105$

Non Dimensional Velocity

Length Scale $lc := \frac{Gibbs\_thomson}{(-Miscb\_gap\_si \cdot m_{l\_si})}$ $lc = 3.826 \cdot 10^{-9}$

Thus:

$tip\_radius := lc \cdot R$ $tip\_radius = 5.098 \cdot 10^{-7}$

Thus: $Curvature := \frac{1}{tip\_radius}$ $Curvature = 1.961 \cdot 10^{6}$

Velocity Scale

$$Vc := \frac{-D\cdot Miscb\_gap\_si \cdot m_{l\_si}}{Gibbs\_thomson}$$ $Vc = 0.784$

$Velocity\_tip := Vc \cdot V$ $Velocity\_tip = 1.294 \cdot 10^{-3}$

Theoretical Peclet Number: $Pc := \frac{(Velocity\_tip \cdot tip\_radius)}{2 \cdot D}$ $Pc = 0.11$

Experimental Peclet Number $vel\_exp := 0.025714$ $tip\_exp := 1.77 \cdot 10^{-7}$

$$Pc\_exp := \frac{(vel\_exp \cdot tip\_exp)}{2 \cdot D} \qquad Pc\_exp = 0.759$$

Appendix C

Phase Maker – Appendix C

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <math.h>
#include <iomanip.h>

main()
{
/*
Programmer: Marcias J. Martinez
Date: July 24th 1998
Supervisor: A. Artemev
Name of the Function: Phasemaker

Purpose:
The purpose of this function is to create a phase field. The location of the center of this phase field can be set by the user to be at the center or at the corner of the
control volume The phase field is created in order to test the curvature function. The values of the phases are between 0 and 1. A phase value of 0 represents a
completed liquid cell while a phase value of 1 represents a completely solidified cell.


The Phase field is created by analyzing a set of random number at every cell. The random number locations are compared to the specific radius of the sphere (set by
the user). If the a point is within the radius is considered a solid (1), while outside of the radius is considered a liquid (0). The fraction of liquid to solid (points within
to outside the radius) is considered the phase fraction.

Modification of October 31 1998,
        We included the posbility of creating a flat plate phase field with a disturbance of the middle of the plate.

            Modification of Nov 23 1998

        Since the phase evolution model seems to be providing propper results I decided to have the  phase maker be able to create a phase that had different number
        of cells in I, J, K  So that K could be larger (so that the dendrite have space to grow) while the I and J directions  are not necessarily as large.

\
Modification of Nov 27 1998

        We added all the material parameters that phase evolution uses to phase maker, since that way we can have them all read as part of an input file.
```

Modification of March 24th 1999
I needed to add a variable to this program so that I can change the width of the cell from 1 cell to a 3 cells width. */

// INITIALIZATION OF FILES

```
ofstream fout("phase.dat"); // Compatible file used to be read by TechPlot
ofstream curvout("phase.in"); // The actual phase field file to be read by the curvature function

if(!fout){
        cout<< " Cannot open output file\n";
return 1;
}
if(!curvout){
        cout<< " Cannot open output file\n";
return 1;
}
```

// VARIABLE DECLARATION

```
int i;
int j;  // 3-D integer for each cell
int k;
int p; // number of iterations;
int max_ij, max_k;


float x, xrand; // The absolute x- coordinator of a point wrt to the origin of the control volume
float y, yrand; // The xrand represent the random generator position of a point with respect to its cell.
float z, zrand;

int liquidcounter; // Number of points that are considered liquid within a cell
int solidcounter;  // Number of points that are considered solid within a cell
int points;
```

```
float delta;                // Actual dimension of the cell;
float radius;               // Radius of the  sphere.
float position;             // position of each point in the cell [i][j][k];
float phasearray[300][300][300]; // This array contains the actual phase field
float liquidfraction;
float solidfraction;
float ct_liqele1, ct_liqele2;
int LocationBoolean, LiqphaseBoolean;
float  conctin;
float  Temperature;
int  noise_boolean ;
float kinetic_coefficient;
float heat_capacity;
float latent_heat;
float Diffiusion_liqhost;
float Diffiusion_liqele1;
float Diffiusion_liqele2;
float Gibbs_thomson;
float mlele1;
float mlele2;
float msele1;
float msele2;
float Temp_melt_host;
int ancho;
int fileint;
int Boundary_boolean;
int Matrix_boolean;
int velocityflag;
float kdistr;
int flayerflag;
int layerbool;
float ct_layer1;
float ct_layer2;
```

```
// VARIABLE INITIALIZATION

        x = 0.0;
        y = 0.0;
        z = 0.0;
        liquidcounter =0;
        solidcounter=0;
        liquidfraction= 0;
        solidfraction =0;
        position = 0;
        mlele1 = (float) -6.0;                  // Liquidus slope of Aluminium Si
        mlele2 = (float) -2.6;                  // Liquidus slope of Aluminium Cu
        Temp_melt_host =  (float) 933.3; // In this case the melting temperature of Aluminium (Kelvin)
        //middlepoint = max/2;


// READING OF INPUT PARAMETERS

        cout << "MARCIAS MARTINEZ PHASE MAKER: " << "\n";
        cout << "Original phase field? (1=Spere Center|2= Sphere Corner|3=Flat Plate): ";
        cin >> LocationBoolean;
        cout << "Enter cell dimension : ";
        cin >> delta;
        cout  << "Would you like a concentration liquid phase (1= Yes| 2= No)? ";
        cin >> LiqphaseBoolean;
        cout << "Maximum size of the control volume (I, J)---> Not bigger than 200: ";
        cin >> max_ij;
        cout << "Maximum size of the control volume (K)---> Not bigger than 200: ";
        cin >> max_k;
        cout << "Would you like the initial concentration based on temperature <1=Y|2=N>: ";
        cin >> conctin;

        if(conctin == 2){
                cout << "Initial concentration of element 1 (silicon) " <<"\n";
                cin >> ct_liqele1;
                cout << "Initial concentration of element 2 (Copper) " << "\n";
```

```
        cin >> ct_liqele2;
} // end if

cout << "Temperature of the system (less than 933K and greater than 860K): ";
cin>>  Temperature;
cout << "Would you like noise added to the system <1=Y|2=N>: ";
cin >> noise_boolean;
cout << "\n";
cout << "Material Parameters: "<< "\n";
cout << "\n";
cout << "Kinetic Coefficient (m/(s*K)): ";
cin >> kinetic_coefficient;
cout << "Alloys heat Capacity (J/m^3*K): ";
cin >> heat_capacity;
cout << "Alloys Latent heat (J/m^3):  ";
cin >> latent_heat;
cout << "Alloys Diffiusion Constant Liquid host (m^2/sec): ";
cin >> Diffiusion_liqhost;
cout << "Alloys Diffiusion Constant Liquid element 1 (m^2/sec): ";
cin >> Diffiusion_liqele1;
cout << "Alloys Diffiusion Constant Liquid element 2 (m^2/sec): ";
cin >> Diffiusion_liqele2;
cout << "Alloys Gibbs thomson coefficient (m*k): ";
cin >> Gibbs_thomson;
cout << "Liquid slope of element 1 (K/%wt use negative sign convention): ";
cin >> mlele1;
cout << "Liquid slope of element 2 (K/%wt use negative sign convention): ";
cin >> mlele2;
cout << "Solidus Slope of element 1 (K/%wt use negative sign convention): ";
cin >> msele1;
cout << "Solidus slope of element 2 (K/%wt use negative sign convention): ";
cin >> msele2;
cout << "Melting temperature of the host component (K): ";
cin>> Temp_melt_host;
cout << "Would you like the nucleus to be 3 cells wide: <1=Y,0=N>";
cin >> ancho;
cout << "Every how many iterations would you like to safe the information: ";
```

```
cin >> fileint;
        cout << "Boundary Conditions <0= PERIODICAL IN I,J BUT NOT in K |  1=ALL PERIODICAL>: ";
        cin >> Boundary_boolean;
        cout << "Would you like to save the phase field in matrix form: <1=No|2=Yes>";
        cin >> Matrix_boolean;
        cout << "Would you like to save the velocity information in a file: <1=No|2=Yes>";
        cin >> velocityflag;
        cout << "Distribuition Coefficient (k): ";
        cin >> kdistr;
        cout <<"First 10 cell layer with a conct equal to the liquidus conct? <1=No|2=Yes>";
        cin >> flayerflag;

        if(flayerflag ==2){
                cout << "Layer based on Liquidus concentration <1=Y|2=N>";
                cin >> layerbool;
        }
        if(flayerflag !=2){
                cout << "Concentration of 1st 10 rows for element 1: ";
                cin >> ct_layer1;
                cout << "Concentration of 1st 10 rows for element 2: ";
                cin >> ct_layer2;
        } // End if



        // This nested for loop allows for the creation of the phase field.
        if(LocationBoolean !=3){

// INPUT PARAMETERS REQUIRED FOR SPHERE

                cout << "Enter Radius ";
                cin >> radius;
                cout << "Enter number of points per cell : ";
                cin >> points;
```

```
if(LiqphaseBoolean == 1){
        cout << "Temperature: (between 933K and 821K) ";
        cin >> Temperature;
        ct_liqele1 = ((Temperature - Temp_melt_host)/(mlele1));
   ct_liqele2 = ((Temperature - Temp_melt_host)/(mlele2));
        cout << " Concentration of Silicon: " << ct_liqele1 << "\n";
        cout << " Concentration of Copper : " << ct_liqele2 << "\n";
} // end if



for(i =0; i <= max_ij; i++)
  for (j=0; j <= max_ij; j++)
        for (k= 0; k <= max_k; k++){

        // " P " NUMBER OF RANDOM POINTS IN A CELL LOOP

        for (p = 0; p < points; p++){
                xrand = (float) rand();
                yrand = (float)rand();
                zrand = (float)rand();
                if (LocationBoolean == 1){
                        x =(float) delta*(max_ij+1)/2 -i*delta - (xrand /32767)*delta;
                        y =(float) delta*(max_ij+1)/2 -j*delta - (yrand /32767)*delta;
                        z =(float) delta*(max_k+1)/2 -k*delta - (zrand /32767)*delta;
                        position =(float) pow(x,2) +(float)pow (y,2) +(float) pow(z,2);
                        position = (float)pow(position,0.5);
                } // End if

                if (LocationBoolean == 2){

                        x =i*delta + (xrand /32767)*delta;
                        y =j*delta + (yrand /32767)*delta;
                        z =k*delta + (zrand /32767)*delta;
                        position =(float) pow(x,2) +(float)pow (y,2) +(float) pow(z,2);
                        position = (float)pow(position,0.5);
```

```
                    }// End if (N)

                if (position > radius) {
                 liquidcounter= liquidcounter +1;

                }
                else {
                 solidcounter = solidcounter +1;
                }

        } // for of for (p) loop

  liquidfraction = (float)liquidcounter;
  solidfraction  =(float) solidcounter;
  phasearray[i][j][k] = solidfraction/points;

  // REINITIALIZATION OF THE LIQUID AND SOLID COUNTER

        liquidcounter = 0; // reinitializing the counter for a new cell
        solidcounter = 0; // reinitializing the conter for a new cell

// REINITIALIZATION OF VARIABLES FOR A NEW CELL
                x = 0;
                y = 0;
                z = 0;
        } // End of For loop i,j,k loop
} // End if (Location Boolean !=3)

// CODE FOR FLAT PLATE
        if(LocationBoolean ==3){



                for(i =0; i <= max_ij; i++)
                 for (j=0; j <= max_ij; j++)
                        for (k= 0; k <= max_k; k++){
```

```
        if(k==0){
          phasearray[i][j][k] = 1;
        } // End if i== 0;
        else{
          phasearray[i][j][k] = 0;
        }


        if ((i== (int) (max_ij+1)/2)&&(j==(int) (max_ij+1)/2)){ // original
                if((k== 1)||(k==2)||(k==3)){
                phasearray[i][j][k]  = 1.;
                } // End of if k ==1
        } // End of i and j = middle point


if(ancho == 1){
        if ((i== (int) ((max_ij+1)/2)+1)&&(j==(int) (max_ij+1)/2)){
                if((k== 1)||(k==2)){
                phasearray[i][j][k]  = 1.;
                } // End of if k ==1
        } // End of i and j = middle point

        if ((i== (int) ((max_ij+1)/2)-1)&&(j==(int) (max_ij+1)/2)){
                if((k== 1)||(k==2)){
                phasearray[i][j][k]  = 1.;
                } // End of if k ==1
        } // End of i and j = middle point

        if ((i== (int) (max_ij+1)/2)&&(j==(int) ((max_ij+1)/2)+1)){
                if((k== 1)||(k==2)){
                phasearray[i][j][k]  = 1.;
                } // End of if k ==1
        } // End of i and j = middle point

        if ((i== (int) (max_ij+1)/2)&&(j==(int) ((max_ij+1)/2)-1)){
```

```
        if((k== 1)||(k==2)){
        phasearray[i][j][k]  = 1.;
        } // End of if k ==1
} // End of i and j = middle point

if ((i== (int) ((max_ij+1)/2)-1)&&(j==(int) ((max_ij+1)/2)+1)){ //esquinas i-1, j+1
        if((k== 1)||(k==2)){
        phasearray[i][j][k]  = 1.;
        } // End of if k ==1
} // End of i and j = middle point

if ((i== (int) ((max_ij+1)/2)+1)&&(j==(int) ((max_ij+1)/2)+1)){ //esquinas i+1, j+1
        if((k== 1)||(k==2)){
        phasearray[i][j][k]  = 1.;
        } // End of if k ==1
} // End of i and j = middle point


if ((i== (int) ((max_ij+1)/2)-1)&&(j==(int) ((max_ij+1)/2)-1)){ //esquinas i-1, j-1
        if((k== 1)||(k==2)){
        phasearray[i][j][k]  = 1.;
        } // End of if k ==1
} // End of i and j = middle point

if ((i== (int) ((max_ij+1)/2)+1)&&(j==(int) ((max_ij+1)/2)-1)){ //esquinas  i+1, j-1
        if((k== 1)||(k==2)){
        phasearray[i][j][k]  = 1.;
        } // End of if k ==1
} // End of i and j = middle point

if ((i== (int) (max_ij)/2)&&(j==(int)(max_ij)/2)){
        if(k==3){
        phasearray[i][j][k]  = 1.;
        } // End of if k ==1
} // End of i and j = middle point
```

```
                } //End Ancho =1

                }// End for
}// End if for locationboolean = 3

// OUTPUT OF PHASE FIELD TO CURVOUT FILE
// OUTPUT INPUT PARAMETER TO FILE

                curvout << radius << "\n";
                curvout << points<< "\n";  // Output to a file
                curvout << delta<< "\n";
                curvout << max_ij<< "\n"; // With this we assure that the ij plane can have a different dim
                curvout << max_k << "\n";           // than the k plane.
                curvout << LiqphaseBoolean <<"\n";
                curvout << conctin << "\n";

                if(conctin == 2){
                        curvout << ct_liqele1 << "\n";
                        curvout << ct_liqele2 << "\n";
                } // end if

                curvout <<  Temperature << "\n";
                curvout <<  noise_boolean << "\n";
                curvout <<  kinetic_coefficient << "\n";
                curvout << heat_capacity << "\n";
                curvout << latent_heat << "\n";
                curvout << Diffiusion_liqhost << "\n";
                curvout << Diffiusion_liqele1 << "\n";
                curvout << Diffiusion_liqele2 << "\n";
                curvout << Gibbs_thomson << "\n";
                curvout << mlele1 << "\n";
                curvout << mlele2 << "\n";
                curvout << msele1 << "\n";
                curvout << msele2 << "\n";
                curvout << Temp_melt_host << "\n";
                curvout << fileint << "\n";
                curvout << Boundary_boolean << "\n";
```

```
    curvout << Matrix_boolean << "\n";
    curvout << velocityflag << "\n";
                curvout << kdistr << "\n";
                curvout << flayerflag << "\n";


                if(flayerflag ==2){
                        curvout << layerbool<< "\n";
                }
                if(flayerflag !=2){
                        curvout << ct_layer1<< "\n";
                        curvout << ct_layer2<< "\n";
                } // End if


                // This line makes the file techplot compatible
                fout << " ZONE I= " << max_k+1 <<", J= "<< max_ij+1<<", k=" << max_ij +1 <<", F=POINT" << "\n"; // Techplot line


// ACTUAL PHASE FIELD
        for(i =0; i <= max_ij; i++)
                for (j=0; j <= max_ij; j++)
                        for (k= 0; k <= max_k; k++){


  if (LiqphaseBoolean == 1){
            curvout << phasearray[i][j][k]*ct_liqele1;
            curvout.width(18);
            curvout << phasearray[i][j][k]*ct_liqele2 <<"\n";
     fout  << i*delta << "        " << j*delta <<"        " << k*delta << "        " << phasearray[i][j][k] << "\n";

  } // End if
  if ((LiqphaseBoolean == 2)&&(LocationBoolean != 3)){
    curvout << phasearray[i][j][k] << "\n"; // <<setw(10)<<i<< setw(3) <<  j<< setw(3) <<k<< "\n";
    fout  << i*delta << "        " << j*delta <<"        " << k*delta << "        " << phasearray[i][j][k] << "\n";

  } // End if if LiqphaseBoolean == 0
```

```
  if (LocationBoolean == 3){
          curvout << phasearray[i][j][k] << "\n";
     fout  << i << "        " << j <<"        " << k << "        " << phasearray[i][j][k] << "\n";


  } // End if

}// End For

// CLOSING OF ALL OUTPUT FILES
          fout.close();
          curvout.close();
          return 0;
} // End of main
```

Phase Evolution

Phase Evolution – Appendix C

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <math.h>
#include <iomanip.h>
#include <stdio.h>
#include <time.h>
#define MAX_SIZE 150



struct alloy {
 float kinetic_coefficient;
 float Gibbs_thomson;
 float mlele1;
 float msele1;
 float msele2;
 float mlele2;
 float heat_capacity;
 float latent_heat;
 float Diffiusion_liqhost;
 float Diffiusion_liqele1;
 float Diffiusion_liqele2;
 float Temp_melt_host;
 float kdistr;

// float Temp_melt_ele1;
// float Temp_melt_ele2;
// float Diffiusion_solele2;
// float Diffiusion_solele1;
// float Diffiusion_solhost;
// float k_host;
// float k_ele1;
// float mlhost;
// float mshost;

};
```

Phase Evolution – Appendix C

```
void conserv_mass_function(double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],double
ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                        double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], alloy alloys,
                                        int max_ij, int max_k, int counter);

void Averagephase(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE],int max_ij, int max_k,
int index[6], int Boundary_boolean);


void velocity_function(alloy alloys, int max_ij, int max_k, float Temperature,
                       double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                                    int velocityflag, int counter, int headerflag,
                                                    double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE]);


void curvature_function (double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE], int index[6], int max_ij, int max_k, int points,
                         float delta, float radius, alloy alloys, int Boundary_boolean,float
curvaturearray_old[MAX_SIZE][MAX_SIZE][MAX_SIZE] );

void Boundary_Condition(int i, int j, int k, int index[6], int max_ij, int max_k, int Boundary_boolean);


int neigbour_function(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], int i, int j, int k, int index[6], int max_ij, int max_k,int
Boundary_boolean);

float time_step_function(float const delta, alloy alloys);


void filesave_function(int counter, int i, int j, int k, int max_ij, int max_k, double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],
          double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
          float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE], double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE],
          float Total_time, float time_step, int tecplotflag, alloy alloys,
```

```
                double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], float Temperature,
                int points, float delta, int LiqphaseBoolean, int Tempboolean, int noise_boolean,
                int Boundary_boolean, int Matrix_boolean, int fileint, float radius, float curvaturearray_old[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                int velocityflag);

void Matrix_savefunction(int counter, int i, int j, int k, int max_ij,  int max_k,  double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE], double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        float Total_time, float time_step, int tecplotflag, alloy alloys,
                        double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], float Temperature);


float greatest_function(float Item1, float Item2);

void phase_evolution(int max_ij,int max_k, double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE], double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE], float
Total_time,
                        float time_step, int tecplotflag, int counter, float delta, int fileint, int index[6], int Boundary_boolean,
                        double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                        alloy alloys, float Temperature);

void Concentration_function(int max_ij, int max_k, float time_step, float delta, double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], alloy alloys, float Temperature, int index[6],
                                double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double
ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double
ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                int Boundary_boolean);
```

Phase Evolution – Appendix C

```
float theta_function(double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], int x, int y, int z);

int Temperature_range(alloy alloys, float Temperature, double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                      double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE]);

void Shift_down_function(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double
ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double
ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         int counter, int max_ij, int max_k);

void noise_function(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], int max_ij, int max_k);
```

```
main(){
    /* Programmer: Marcias J. Martinez
       Start up Date: August 17 1998
       Last modified: Nov 5th 1998
       Version: 1.00.001
       Supervisor: A. Artemev
       Name of the function: Phase Envolution

       Purpose:
       To determine the morphology of a dendrite, having an initial concentrations and phase field.
       The evolution of the dendrites morphology is based on flicks diffusion laws.

       Modification of Nov 4rth.
           We found that the cells due to the neighbouring funciton where being allowed to solidify
           without these having a fully crystallized neighbour. So we modified this function.

       Modificaiton of Nov 5th
```

Phase Evolution – Appendix C

I eliminated from the time step function the calculation of time step based on heat transfer parameter. Reason for this is the fact that heat transfer is not taken into account in this model.. This should speed up the time of evolution.

I also modified this day the temperature range function, that ensure that we are 10 degrees below the liquidus temperature and at least 5 degrees above the solidus temperature..

Modification of Nov 12 98

It was necessary to modify the concentration function such that it only shows 2 scenarios
1. When 1-phase < 0.5 excess soute is redistributed and then it is diffused by equation (7) in Prof. Artemevs article
2. All other diffusion is done by equation (7) in Prof. Artemevs article.

I have also added a subroutine (function) that will calculate for every iteration the new time step. This is necessary so that the time step is constantly changing and then a too large time step that can produce negative concentrations due to fluctuation did not ocure.

Modification of Nov 17 1998

It was necessary to add the shift down function so that the bottom layers don't grow faster and inhibit us to see the growth of the column due to the fast growth of the surrounding cells.

Modification of Nov 19 1998

The time step calculated due to remelting could cause the time to go to zero
(in the recalculation of time function) and thus,
no evolution would ocurre. Thus It was necessary to modify this criteria so that it would calculate it based on the fact that it can only remelt 1% of the size of the cell.
I was also recalculating the time in the pase evolution fiedl and this was not necesary.
so I eliminated this part of the function, since the time step is being passed to the function.
I also modified the concentration function so that it had an if and else statement instead of two if statements.

Modification of Nov 23 1998

This version of phase evolution differs from version 7 that the space volume in ij can be

different than in k.

Modification of Nov 27 1998

During this day we added the noise function. This function adds at the end of each cycle to each cell that is between zero and 1 but not zero nor 1 a random fraction of solid.
This fraction is normailized to 1 and then it is divided by the percentage of the solid fraction at that cell.

Modification of Dec 6th 1998

I decided to change the noise function to go from 0.5 to -0.5 and dividing it by 1 million without multiplying it by the liquid fraction of each cell.
I also decided to creat a property text file that will output all the parameters used to run each test.

Modification of Dec 10th 1998

It was necessary to modify the curvature function to take into account anisotropy effects. Instead of creating a second array we decided to include into the curvature values the modified gibbs thomson coefficient. Thus, in the velocity function it is not necessary to multiply it time the Gibss Thomson value any more.
Since the curvature array already includes this value.

Modification of Dec 15th 1998

The value for the averaging constants in the average phase field function where changed to 0.2, 0.1, 0.05..

Modification of January 5th 1999

The Boundary Condition function was changed since it was not working due to the difference between an = and an == comparison sign.
Version 10-2 and 10-0 should only differ in the averaging scheme constants. This is the most upto date version.

The weight factors that should be used or that seem to provide better results are:

// Definition of Weight Factors

```
M1=(float) 0.3;          // Middle Layer
M2=(float) 0.35;         // Middle Layer
M3=(float) 0.3;          // Middle Layer
M4=(float) 0.35;         // Middle Layer
M5=(float) 1.0;          // Middle Layer
M6=(float) 0.35;         // Middle Layer
M7=(float) 0.3;          // Middle Layer
M8=(float) 0.35;         // Middle Layer
M9=(float) 0.3;          // Middle Layer

U1=(float) 0.120;        //Upper Layer
U2=(float) 0.3;          // Upper Layer
U3= (float) 0.120;       // Upper Layer
U4=(float) 0.3;          // Upper Layer
U5=(float) 0.35;         // Upper Layer
U6=(float) 0.3;          // Upper Layer
U7=(float) 0.120;        // Upper layer
U8=(float) 0.3;          // Upper Layer
U9=(float) 0.120;        // Upper Layer

L1=(float) 0.120;
L2=(float) 0.3;          // Lower Layer
L3=(float) 0.120;        // Lower Layer
L4=(float) 0.3;          // lower Layer
L5=(float) 0.35;         // Lower Layer
L6=(float) 0.3;          // Lower Layer
L7=(float) 0.120;        // Lower Layer
L8=(float) 0.3;          // Lower Layer
L9=(float) 0.120;        // Lower Layer
```

Modification January 6th 1999

Creation of the recovery function. This function will allow us to re-start a run from the last iteration that was saved.

Phase Evolution – Appendix C

Modification of February 5th 1999
Phase Evolution 10-2 differs from Phase Evolution 10-25 by

The recalculation of the time function was eliminated.
Time is then only calculated once.. And this one is based on the stability criteria.
For the frist 1000 iteration the time based on the stability criteria is multiplied by 0.01
After iteration 1001 then the time is multiplied by ten so that is only 0.1 of the value provided by the stability criteria.

The phase evolution function was modified significantly in order to calculate the maximum allowed change in the phase field. If the change in phase field is greater than the maximum allowed then new position in the phase field is recalculated based on the maximum allowed phase field. Sveral new local parameters that will remain local to the phase field function had to be created and a few of the global variables were required to be passed to the phase evolution function.

Modification of February 8th 1999

The results provided by February 5th (test 5, section 1 and 2) gave us an insight on several actions to take.

1. The reduction on the time step from 0.01 of the stability criteria to 0.001 of the stabilit criteria.
2. During the if statement found in the phase field function we realized that the if statement would be required to be an absolute value of chnage in the phasefield. This is because the actual change could be negative in which case the if statement would not satisfied. Other changes in the formula had to be made since I did not type them in properly.. For example the k (Distribution coefficient was inversed) and the phase field used was the new phase field when the old one was the one we had to use.

3. After the dendrite is well formed then we can speed up the process by increasing the size of the time step.
4. An if statment in the concentration function for checking if the new phase field is greater than or equal to 1.0 was changed to 0.9999999 since it could happen that the phase field would take a value very close to 1.0 but still be matematically less than one. This would

cause the concentration to fluctuate and could provide negative concentrations which is physically impossible. Thus we changed it to 0.99999

All these changes where taken into account during this revision of Phase Evolution.

Modifications of February 18th

1. From the modification of Febraury 8th it was found that the concentrations function was not completely correct. Prof. Artemev found that I was neglecting the component transport. From the discusion we had February 17th we needed to redo the concentrations function.
Phase Evolution 26 is basically different from Phase Evolution 2. in that this error is solved.
This will allow us to run the simulation at a much faster rate. In other words at a bigger time step.

The Transport equation used was Equation (3) of his article.

Modification of March 3rd 99

1. When Analyzing the tip raidus I realized that the curvature value that I was outputing to the file had the anysotropy factor already taken into account. Thus I had to creat a curvature array old that does not include the anisotropy values. This will allow me to compare tip radis with the analytical data obtained from Andreis Code.

Modification of March 24th 99

1. I needed to add an extra output file of information at the velocity equation in order to determine what factors affect this equation, to see what are the causes that the dendrite tip is growing in a needle shape form.

Modification of March 31 99

1. Prof. Artemev asked me to do a run with a single Al-Si only and the problem is that the program gave negative concentration in the Cu array.
So we add a flat where EL2-flag is equal to one then we calculate the concentration profile of this element, otherwise we don't.
Also we make sure that the liquius slope of the second compoent are zero

to avoide it affecting the velocity..

Modification of April 2nd 99

1. Prof. Artemev did not agree with the fact of adding the e2-flag.. I explained that if we did not do that we would obtain a negative concentration in Cu due to the change in the phase field and he ageed. So instead of using the Cs(Temp) I will be modifying the concentration function and replacing Cs by (1-k). Where kdistr (to distinguish it from counter k) is distribuition coefficient and has a value of 0.13.

Modification of April 5th 99

1. With PE-V2 we eliminate the mass transport to cells that only have a point or a line in common with the center cell.
2. With PE-V2-2.exe we add another feature, this feature is that the first 10 cells of the CV. acutally have the liquidus concentration assigned to them.

Modification of April 9th 99

1. After analyzing some of the cells, we realized that it seems that the code is breaking the law of conservation of mass. This is because at some cells the ijk value is not being reduces, which is extremely strange. Thus, it is possible that this is occuring because of floating to double values of the variables. The changes will be made to the Concentration function.

Modification of April 21st 99

It was seen with previous runs that in terciary system one could peak the values of concentration that would produce remelt. When the program is not capable of taking into account the remelting of completely solidified cells then the dendrite does not grow nor does it disaappear. Remelting of cell that are partially solidified is possible but not of' those that are completely solidified. Thus I added the option that the 1st 10 layers one could peak concentraitons based on temperature (liquidus concentration or based on concentration values submitted by the user.

Modification of May 3rd 1999,

Addition of an extra function that basically computes the total mass every n number of iterations. That way we can calculate if the conservation of mass is being violated This is done by multiplying each cell with the solid fraction by its corresponding solid concentration and the liquid fraction by the liquid concentration and then

adding all this up. For this I decided to create the conserv_mass_function;

```
*/

// DEFINITION OF GLOBAL VARIABLES

    double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE];        // Phase field 3-D array
    double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE];            // Phase field in 3-D at previous time step
    float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE];              // Averaged Phase Field
    double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE];                  // Curvature array
        float curvaturearray_old[MAX_SIZE][MAX_SIZE][MAX_SIZE]; // Curvature array withouth anysotropy effects.
    double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE];         // Velocity Array
    double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE];   // Liquidus concentration of element 1. at time = t+1
    double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE];       // Liquidus concentration of element 1. at time = t
    double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE];   // Liquidus concentration of element 2. at time = t+1
    double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE];       // Liquidus concentration of element 2. at time = t


// DEFINITION OF GLOBAL INTEGERS AND CONSTANTS

    int index[6];  // Index array on Boundary conditions (Periodical Boundary Conditions)
        int max_ij;      // This integer indicates the max size of the node
    int max_k;
    int points;    // When creating the original phase field base on the number of points over and under
                   // a specific radius.
    float delta;   // Spacing between the center of a cell and any of its neigbours in X, Y or Z
    float radius;  // Radius of the original spherical phase field.
    int i, j, k;   // Indices indicating the actual node.
    int stop;          // Dummy variable for debugging purposes.

    int counter;   // Number of iterations
    int tecplotflag; // This variable is used in order to insert in output text file the command lines required
```

```
                    // by TECHPLOT to graph the data.
int fileint;
int LiqphaseBoolean, conctin;
int noise_boolean;
int Boundary_boolean, Matrix_boolean;
char recover_bool;
float Temperature;                          // This is the temperature of the melt at the liquid-solid interface.
int Tempboolean;                            // If =1 yes it falls within the range othewise tempboolean =0;
float time_step;
double Total_time;
int dummy_i, dummy_j, dummy_k;
int timeflag, velocityflag, headerflag, flayerflag;
        int layerbool;
        float ct_layer1, ct_layer2;




// DECLARATION OF ALLOY STRUCTURE TYPE

    alloy alloys;

// OPENING FOR OUTPUT OF PROPERTY FILES
        // Opening of file name
    ofstream prop;
    prop.open("Property.txt", ios::app);

    if(!"Property.txt"){
        cout<< "Can not open file \n";
        // Handle error
        cin >> stop;
        exit (0);
    }
```

Phase Evolution – Appendix C

```
// REQUIREMENT CODE FOR INPUT AND OUTPUT FILES
    ifstream in;    // input


// RECOVERY SUBSECTION
    // First we check if this is a recovery file by checking the phasein.txt for an R and
    // a counter to go to and read the information.
    // In this section we read from the last iteration that was saved.
    // If the fist Letter in the phasein.txt is R then it is required to recover from
    // counter number file.

        ifstream recover;
        recover.open("recover.txt");

        if (!recover){
            cout<< " Connot open file\n";
            // handle error
            cin >> stop;
            exit(0);
        } // End if

        // Here I assign recover_bool a letter different than R to make sure that it is reading
        // the information correctly.
        recover_bool = 'C'; // C as in CONTINUE
        recover >> recover_bool;
        recover >> counter;
        recover.close();

        // IF RECOVERY IS REQUIRED
                // This is only to open the file fo the last iteration that was saved


        if (recover_bool == 'R'){
```

```
        // Variable Declaration for Filesave_function
        char fileName[40];
        char *baseFileName="newphase"; // Base File Name
        char stop;

        // Editing of file name command.
        sprintf (fileName, "%s%d.txt", baseFileName, counter);

        // Opening of file name
        ifstream retrieve;
        retrieve.open(fileName, ios::app);

        if(!fileName){
            cout<< "Can not open file \n";
            // Handle error
            cin >> stop;
            exit (0);
        }// End if of filename

// Output to the screen that we are in recovery mode.
    cout << "PHASE EVOLUTION IS RECOVERING FROM AN OLD RUN" << "\n";


// At this point we start reading the data in and introducing it in the respective
// variables
        retrieve >> recover_bool >> "\n";
        retrieve >> recover_bool >> counter >>"\n";
        retrieve >> recover_bool >> Temperature >> "\n";
        retrieve >> recover_bool >> Total_time >> "\n";
        retrieve >> recover_bool >> time_step >> "\n";
        retrieve >> recover_bool >> alloys.kinetic_coefficient >> "\n";
        retrieve >> recover_bool >> alloys.Gibbs_thomson >> "\n";
        retrieve >> recover_bool >> alloys.mlele1 >> "\n";
        retrieve >> recover_bool >> alloys.msele1 >> "\n";
        retrieve >> recover_bool >> alloys.msele2 >> "\n";
        retrieve >> recover_bool >> alloys.mlele2 >> "\n";
```

```
retrieve >> recover_bool >> alloys.heat_capacity >> "\n";
retrieve >> recover_bool >> alloys.latent_heat >> "\n";
retrieve >> recover_bool >> alloys.Diffiusion_liqhost >> "\n";
retrieve >> recover_bool >> alloys.Diffiusion_liqele1 >> "\n";
retrieve >> recover_bool >> alloys.Diffiusion_liqele2 >> "\n";
retrieve >> recover_bool >> alloys.Temp_melt_host >> "\n";
retrieve >> recover_bool >> points >> "\n";
retrieve >> recover_bool >> delta >> "\n";
retrieve >> recover_bool >> max_ij >> "\n";
retrieve >> recover_bool >> max_k >> "\n";
retrieve >> recover_bool >> LiqphaseBoolean >> "\n";
retrieve >> recover_bool >> noise_boolean >> "\n";
retrieve >> recover_bool >> Tempboolean >> "\n";
retrieve >> recover_bool >> Boundary_boolean >> "\n";
retrieve >> recover_bool >> Matrix_boolean >> "\n";
retrieve >> recover_bool >> fileint >> "\n";
retrieve >> recover_bool >> radius >> "\n";
                                        retrieve >> recover_bool >> velocityflag >> "\n";
                                        retrieve >> recover_bool >> alloys.kdistr >>"\n";

for(i=0; i<=111; i++){
    // This is to skip a header and tecplot header
    retrieve >> recover_bool;}


// At this point we read into the arrays the variables by means of a for loop

for(i=0; i <= max_ij; i++)
    for(j=0; j <= max_ij; j++)
        for(k = 0; k <= max_k; k++) {
        retrieve >> dummy_i;
        retrieve >> dummy_j;
        retrieve >> dummy_k;
        retrieve >> gzone[i][j][k];
        retrieve >> gavg[i][j][k];
        retrieve >> gzone_new[i][j][k];
        retrieve >> curvaturearray[i][j][k];
```

```
                                                                retrieve >> curvaturearray_old[i][j][k];
                    retrieve >> velocity[i][j][k];
                    retrieve >> ct_liqele1[i][j][k];
                    retrieve >> ct_liqele1_new[i][j][k];
                    retrieve >> ct_liqele2[i][j][k];
                    retrieve >> ct_liqele2_new[i][j][k];
                } // end for for recovery loop

            recover_bool = 'R'; // This is to make sure that it does not go in and ask for the information
                                // again
            // Output to the screen that we are in recovery mode.
            cout << "PHASE EVOLUTION IS RECOVERING FROM AN OLD RUN" << "\n";
            cout << "Press the number one to continue: ";
            cin >> stop;

        }// End if for recover_bool = R



// INPUT PHASE FIELD
        // The Initial phase field is taken from another program known as PHASEMAKER.EXE
        // This initial field is then read into the program from phasein.txt file
if (recover_bool != 'R'){

        ifstream indata;
        indata.open("phasein.txt"); //open file for input

        if (!indata){
                cout<< " Connot open file\n";
                // handle error
                cin >> stop;
                exit(0);
        } // End if


// Input of the initial four variables in the Phase.in file into its corresponding variables
```

Phase Evolution – Appendix C

```
cout << "MARCIAS MARTINEZ DENDRITE EVOLUTION SIMULATION" << "\n";
        cout << "Press the number one to continue: ";
cin >> stop;
        prop << "Properties running on Dendrite Evolution Simulation" << "\n";
indata >> radius;
prop << "Radius: " << radius << "\n";
//cout << "Radius: " << radius <<"\n";
indata >> points;
//cout << "Points  " << points << "\n";
prop << "Points used to creat initial phase field: " << points << "\n";
indata >> delta;
//cout << "Delta: " << delta << "\n";
prop << "Delta used in the grid: " <<delta<<"\n";
indata >> max_ij;
//cout << "Maximum value of the array in ij: " << max_ij << "\n";
prop << "Maximum value of the array in i and j: " << max_ij << "\n";
indata >> max_k;
//cout << "Maximum value of the array in k: " << max_k << "\n";
prop << "Maximum value of the array in k: " << max_k << "\n";
indata >> LiqphaseBoolean;
//cout << "Liquid phase only <1=Y|2=N>: " << LiqphaseBoolean;
prop << "Liquid phase only <1=Y|2=N>: " << LiqphaseBoolean << "\n";
//cout << "\n";
indata >> conctin;
// cout << "Would you like the initial concentration based on temperature <1=Y|2=N>: "<<conctin<<"\n";

if(conctin ==2){
     indata >> ct_liqele1[0][0][0] ;
//     cout << "Initial concentration of Silicon: "<< ct_liqele1[0][0][0] <<"\n";
     prop << "Initial concentration of Silicon: "<< ct_liqele1[0][0][0] <<"\n";
     indata >> ct_liqele2[0][0][0];
//     cout << "Initial concentration of Copper: "<< ct_liqele2[0][0][0] <<"\n";
     prop << "Initial concentration of Copper: "<< ct_liqele2[0][0][0] <<"\n";
}// End if;

indata >> Temperature;
//   cout << "Initial temperature (less than 933K and greater than 860K): "<< Temperature;
```

```
    prop << "Temperature of the system: "<< Temperature <<"\n";
//  cout << "\n";
    indata >> noise_boolean;
//  cout << "Would you like noise added to the system <1=Y|2=N>: "<< noise_boolean << "\n";
    prop << "Is noise added to the system <1=Y|2=N>: " << noise_boolean << "\n";
//  cout << "Material Parameters: "<< "\n";
    prop << "\n";
    prop << "Material Parameters: "<< "\n";
    prop << "\n";
    indata >> alloys.kinetic_coefficient;
//  cout << "Kinetic Coefficient (m/(s*K)): " << alloys.kinetic_coefficient << "\n";
    prop << "Kinetic Coefficient (m/(s*K)): " << alloys.kinetic_coefficient << "\n";
    indata >> alloys.heat_capacity;
//  cout << "Alloys heat Capacity (J/m^3*K): " <<alloys.heat_capacity << "\n";
    prop << "Alloys heat Capacity (J/m^3*K): " <<alloys.heat_capacity << "\n";
    indata >> alloys.latent_heat;
//  cout << "Alloys Latent heat (J/m^3):  " << alloys.latent_heat << "\n";
    prop << "Alloys Latent heat (J/m^3):  " << alloys.latent_heat << "\n";
    indata >> alloys.Diffiusion_liqhost;
//  cout << "Alloys Diffiusion Constant Liquid host (m^2/sec): " << alloys.Diffiusion_liqhost <<"\n";
    prop << "Alloys Diffiusion Constant Liquid host (m^2/sec): " << alloys.Diffiusion_liqhost <<"\n";
    indata >> alloys.Diffiusion_liqele1;
//  cout << "Alloys Diffiusion Constant Liquid element 1 (m^2/sec): " << alloys.Diffiusion_liqele1 <<"\n";
    prop << "Alloys Diffiusion Constant Liquid element 1 (m^2/sec): " << alloys.Diffiusion_liqele1 <<"\n";
    indata >> alloys.Diffiusion_liqele2;
//  cout << "Alloys Diffiusion Constant Liquid element 2 (m^2/sec): " << alloys.Diffiusion_liqele2 <<"\n";
    prop << "Alloys Diffiusion Constant Liquid element 2 (m^2/sec): " << alloys.Diffiusion_liqele2 <<"\n";
    indata >> alloys.Gibbs_thomson;
//  cout << "Alloys Gibbs thomson coefficient (m*k): " << alloys.Gibbs_thomson <<"\n";
    prop << "Alloys Gibbs thomson coefficient (m*k): " << alloys.Gibbs_thomson <<"\n";
    indata >> alloys.mlele1;
//  cout << "Liquid slope of element 1 (K/%wt use negative sign convention): " << alloys.mlele1 <<"\n";
    prop << "Liquid slope of element 1 (K/%wt use negative sign convention): " << alloys.mlele1 <<"\n";
    indata >> alloys.mlele2;
//  cout << "Liquid slope of element 2 (K/%wt use negative sign convention): " << alloys.mlele2 <<"\n";
    prop << "Liquid slope of element 2 (K/%wt use negative sign convention): " << alloys.mlele2 <<"\n";
    indata >> alloys.mselel;
```

```
//    cout << "Solidus Slope of element 1 (K/%wt use negative sign convention): " << alloys.msele1 <<"\n";
      prop << "Solidus Slope of element 1 (K/%wt use negative sign convention): " << alloys.msele1 <<"\n";
      indata >> alloys.msele2;
//    cout << "Solidus slope of element 2 (K/%wt use negative sign convention): " << alloys.msele2 <<"\n";
      prop << "Solidus slope of element 2 (K/%wt use negative sign convention): " << alloys.msele2 <<"\n";
      indata >> alloys.Temp_melt_host;
//    cout << "Melting temperature of the host component (K): " << alloys.Temp_melt_host <<"\n";
      prop << "Melting temperature of the host component (K): " << alloys.Temp_melt_host <<"\n";

// THIS SECTION WE WOULD NEED TO INCLUDE AS VARIABLES IN THE INPUT FILE
            indata >> fileint;
      //cout << "Every how many iterations would you like to safe the information: "<< fileint << "\n";
      prop << "Every how many iterations would you like to safe the information: "<< fileint<<"\n";

            indata >> Boundary_boolean;
            //cout << "Boundary Conditions <0= PERIODICAL IN I,J BUT NOT in K | 1=ALL PERIODICAL>: "<< Boundary_boolean <<"\n";
      prop << "Boundary Conditions <0= PERIODICAL IN I,J BUT NOT in K | 1=ALL PERIODICAL>: "<< Boundary_boolean <<"\n";

            indata >> Matrix_boolean;
            //cout << "Would you like to save the phase field in matrix form: <1=No|2=Yes>"  << Matrix_boolean << "\n";
      prop << " Would you like to save the phase field in matrix form: <1=No|2=Yes>" << Matrix_boolean << "\n";

            indata >> velocityflag;
            //cout << "Would you like to save the velocity information in a file: <1=No|2=Yes>" << velocityflag << "\n";
            prop << "Would you like to save the velocity information in a file: <1=No|2=Yes>" << velocityflag << "\n";

            indata >> alloys.kdistr;
            //cout << "Distribuition Coefficient (k): "<< alloys.kdistr <<"\n";
            prop << "Distribuition Coefficient (k): "<< alloys.kdistr <<"\n";

            indata>> flayerflag;
            //cout <<"First 10 cell layer with a conct equal to the liquidus conct? <1=No|2=Yes>"<< flayerflag<< "\n";
            prop <<"First 10 cell layer with a conct equal to the liquidus conct? <1=No|2=Yes>"<< flayerflag<< "\n";

            if (flayerflag ==1){
                  indata >> ct_layer1;
                  indata >> ct_layer2;
```

```
        } // end if

//      cout << "Concentration of 1st 10 rows for element 1: " <<ct_layer1 <<"\n";
        prop << "Concentration of 1st 10 rows for element 1: " <<ct_layer1 <<"\n";
//      cout << "Concentration of 1st 10 rows for element 2: " <<ct_layer2 <<"\n";
        prop << "Concentration of 1st 10 rows for element 2: " <<ct_layer2<<"\n";

        if(flayerflag == 2){
                indata >> layerbool;
        } // end if

        prop.close();


// CHECK FOR TEMPERATURE RANGE that Tm-ms*Csi - msol*CtCt < Temperatre < Tm -msi*Csi - mcu*CCu


      Tempboolean = Temperature_range(alloys, Temperature, ct_liqele1, ct_liqele2);

      do{
           if( Tempboolean == 0){
           cout << "ERROR TEMPERATURE CHOOSEN IS OUT OF THE ALLOWED TEMPERATURE RANGE" << "\n";
           cout << " Initial temperature (less than 933K and greater than 860K): ";
           cin >>  Temperature;
           Tempboolean = Temperature_range(alloys, Temperature, ct_liqele1, ct_liqele2);

           } // end if of tempboolean ==0;
      } while(Tempboolean !=1);



// INPUT DATA FROM THE ORIGINAL PHASE FIELD FILE

   for(i=0; i <= max_ij; i++)
        for(j=0; j <= max_ij; j++)
             for(k = 0; k <= max_k; k++) {
```

```
if (LiqphaseBoolean == 2){
 // If LiqphaseBoolean ==2 this means that only the phase field is in the phasein.txt file
                                        indata >> gzone[i][j][k];

 // At this point the phase field at time = t and time = t+1 is the same for the first iteration
    gzone_new[i][j][k] = gzone[i][j][k];

 // Calculation of the liquidus concentrations based on the current temperature
    if (conctin == 1){
        ct_liqele1[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele1));
        ct_liqele1_new[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele1));
        ct_liqele2[i][j][k] = ((Temperature- alloys.Temp_melt_host)/(alloys.mlele2));
        ct_liqele2_new[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele2));

    }// End if of conctin 1

    if (conctin == 2){


                                                    ct_liqele1[i][j][k] = ct_liqele1[0][0][0];
        ct_liqele2[i][j][k] = ct_liqele2[0][0][0];
        ct_liqele1_new[i][j][k] = ct_liqele1[0][0][0];
        ct_liqele2_new[i][j][k] = ct_liqele2[0][0][0];


    } // End if of conctin 2;

    } // End if the liquid phase boolean
if (LiqphaseBoolean == 1){
    // If liqphaseBoolean ==1 then the phase filed is a liquid concnetration of element 1 and 2
        gzone[i][j][k] = 0;
        indata >> ct_liqele1[i][j][k] >> ct_liqele2[i][j][k];
        ct_liqele1_new[i][j][k] = ct_liqele1[i][j][k];
        ct_liqele2_new[i][j][k] = ct_liqele2[i][j][k];

} // End if
```

```
    }     // iteration identical the the initial position, since evolution does not ocure
          // at every cell but only those with a phase field diffrente than 1.0 or 0.0


   // If firstflag == 2 = yes then I reassing the concentration values of the first
   // 10 layer

   if(flayerflag ==2){
            //cout << "Layer based on Liquidus concentration <1=Y|2=N>";

            if(layerbool == 1){
            for(i=0; i <= max_ij; i++)
for(j=0; j <= max_ij; j++)
     for(k =0; k <= 10; k++) {

                                        // Addition of April 5th 99
                                        if(ct_liqele1[0][0][0] != 0){
                                                ct_liqele1[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele1));
                                                ct_liqele1_new[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele1));
                                        }// End if of ct1
                                        if(ct_liqele2[0][0][0] !=0){
                                                ct_liqele2[i][j][k] = ((Temperature- alloys.Temp_melt_host)/(alloys.mlele2));
                                                ct_liqele2_new[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele2));
                                        }// End if of ct2


   }// End for
            }// end if layer bool
            else{

            for(i=0; i <= max_ij; i++)
for(j=0; j <= max_ij; j++)
     for(k =0; k <= 10; k++) {

                                        // Addition of April 5th 99
```

```
                                        if(ct_liqele1[0][0][0] != 0){
                                                ct_liqele1[i][j][k] = ct_layer1;
                                                ct_liqele1_new[i][j][k] = ct_layer1;
                                        }// End if of ct1
                                        if(ct_liqele2[0][0][0] !=0){
                                                ct_liqele2[i][j][k] = ct_layer2;
                                                ct_liqele2_new[i][j][k] = ct_layer2;
                                        }// End if of ct2


        }// End for
                }// end else
        }// End if

        if(flayerflag ==1){
                for(i=0; i <= max_ij; i++)
    for(j=0; j <= max_ij; j++)
        for(k =0; k <= 10; k++) {

                                        // Addition of April 5th 99
                                        if(ct_liqele1[0][0][0] != 0){
                                                ct_liqele1[i][j][k] = ct_layer1;
                                                ct_liqele1_new[i][j][k] = ct_layer1;
                                        }// End if of ct1
                                        if(ct_liqele2[0][0][0] !=0){
                                                ct_liqele2[i][j][k] = ct_layer2;
                                                ct_liqele2_new[i][j][k] = ct_layer2;
                                        }// End if of ct2
                                        } // end for
        }// ENd if of flayer == 1


for(i=0; i <= max_ij; i++)
    for(j=0; j <= max_ij; j++)
        for(k = 0; k <= max_k; k++) {
                //This for loops re-assigns a value only to those cell that have a phase field of 1.
            if (gzone[i][j][k]==1){
```

```
                    ct_liqele2[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.msele2));
                    ct_liqele2_new[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.msele2));
                    ct_liqele1[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.msele1));
                    ct_liqele1_new[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.msele1));;

                    //ct_liqele2_new[i][j][k] = ((Temperature - alloys.Temp_melt_host)/(alloys.msele2));
                } // End if

            }// END FOR
// Close of INPUT FILE
            indata.close();

// INITIALIZATION OF TOTAL TIME VARIABLE
    Total_time =0;
    counter =0;
        headerflag = 2; // For the velocity function


// CALCULATION OF TIME STEP
    // It is important to note that this initial time step is based on the stability criteria

    time_step= time_step_function(delta, alloys); // This calls the time step function
    time_step = time_step *0.01; // The multiplication by 0.001 is to assure that is below the
                                 // stability criteria. After it reaches 1/3 of the size of the
                                 // control volume it will change to a larger time step.

    tecplotflag = 1;                          // Sets the TECHPLOT Flag to 1
    timeflag = 0;                             // Sets the timeflag to 0;

} // END IF of Input phase when Recover boolean is NOT equal to R


do{

// CALL OF THE PHASE EVOLUTION FUNCTION
```

Phase Evolution – Appendix C

```
// Oct: 12th 1998I included into the phase evolution a check such that time step* velocity is not greater than
// than delta for that cell.
// If I do this I am thinking that the sphere might not grow evenly.


phase_evolution(max_ij,max_k, gzone,gzone_new,curvaturearray, gavg, velocity, Total_time,
                time_step, tecplotflag, counter, delta, fileint, index, Boundary_boolean,
                ct_liqele1_new, ct_liqele1, ct_liqele2_new, ct_liqele2, alloys, Temperature);

// CALCULATION OF CURVATURE

        curvature_function(gzone_new, gavg, curvaturearray,index, max_ij, max_k, points, delta, radius,alloys, Boundary_boolean,
curvaturearray_old);

// CALCULATION OF CONCENTRATION

        Concentration_function(max_ij, max_k, time_step, delta, gzone_new, gzone, alloys, Temperature, index,
                               ct_liqele1_new, ct_liqele1, ct_liqele2_new, ct_liqele2,Boundary_boolean);

// CALCULATION OF VELOCITY

        velocity_function( alloys, max_ij, max_k, Temperature, curvaturearray, velocity, ct_liqele1, ct_liqele2, velocityflag, counter, headerflag,
gzone);

// CALLS FOR THE SAVE FUNCTION TO SAVE ALL THE VALUES

    if(counter % fileint == 0.0){

        for(i=0; i <= max_ij; i++)
            for(j=0; j <= max_ij; j++)
                for(k = 0; k <= max_k; k++) {
                filesave_function( counter, i, j, k, max_ij, max_k, gzone,      gzone_new, curvaturearray,
                        gavg, velocity,    Total_time, time_step, tecplotflag, alloys,
                        ct_liqele1,ct_liqele1_new, ct_liqele2, ct_liqele2_new, Temperature,
                        points, delta,LiqphaseBoolean, Tempboolean, noise_boolean,
                        Boundary_boolean, Matrix_boolean, fileint, radius, curvaturearray_old,
```

```
                                                                    velocityflag);

                tecplotflag =0;

        } // End of For

         // We only save the Matrix of the phase field if the user wishes to do so
                if(Matrix_boolean == 2){
                    Matrix_savefunction( counter, i, j, k, max_ij, max_k, gzone,     gzone_new, curvaturearray, gavg, velocity,   Total_time,
time_step, tecplotflag, alloys,
                        ct_liqele1, ct_liqele1_new, ct_liqele2, ct_liqele2_new, Temperature);
                }// End if of matrix boolean

                        // Conservation of mass calculation
                    conserv_mass_function(gzone_new, ct_liqele1_new,ct_liqele2_new,alloys,
                                                max_ij, max_k, counter);


    } // ENd if

// COPY OF GZONE NEW TO GZONE OLD AND CONCENTRATION NEW TO BECOME CONCENTRATION OLD

    for(i=0; i<=max_ij; i++)
        for(j=0; j<=max_ij; j++)
            for(k=0; k<=max_k; k++){
                gzone[i][j][k] = gzone_new[i][j][k];

                ct_liqele1[i][j][k] = ct_liqele1_new[i][j][k];
                ct_liqele2[i][j][k] = ct_liqele2_new[i][j][k];
            } // End of for loop



// INCREASEES COUNTER AND TOTAL TIME AND REDUCES TEMPERATURE (COOLING EFFECT)
```

```
        counter = counter +1;
        Total_time =Total_time + time_step; // Increment of Total Time
        Temperature =Temperature -(float) 0.0;   // New temperature value
        tecplotflag =1;
                                headerflag = 1;

// RECALCULATION OF TIME STEP;
        // For the first 1/3 of the C.V. it will use a small time step once it has
        // reached this level it will start growing with a larger time step.

        if((gzone_new[(max_ij-1)/2][(max_ij-1)/2][max_k-40] != 0.0)&&(timeflag == 0)){
            time_step = time_step*1; // Here we deciede to which level we wish
            timeflag = 1;                                   // to increase the time step.
        } // End if


// CALL OF SHIFT DOWN FUNCTION

        Shift_down_function(gzone, gzone_new,ct_liqele1_new, ct_liqele1,ct_liqele2_new,
                        ct_liqele2,counter, max_ij, max_k);


// NOISE FUNCTION CALL
    // This function is only called if the user has specified noise to be introduced into the system
    if(noise_boolean == 1){
        noise_function(gzone, max_ij, max_k); // the noise is added to gzone and not gzone new because
                                                    // immediately after the phase evolution is calculated
    }// ENd if of noise function call

} while((gzone[(max_ij-1)/2][(max_ij-1)/2][max_k-5]==0)||(gzone[2][(max_ij-1)/2][(max_k-1)/2]==0)||(gzone[(max_ij-1)/2][2][(max_k-1)/2]==0));//
End of while loop;


    // So that the last file is tecplot compatible we set the tecplot flag =1;
    tecplotflag =1;

    for(i=0; i <= max_ij; i++)
```

```
        for(j=0; j <= max_ij; j++)
            for(k = 0; k <= max_k; k++) {
            filesave_function( counter, i, j, k, max_ij, max_k, gzone, gzone_new, curvaturearray, gavg, velocity,    Total_time, time_step,
tecplotflag, alloys,
                              ct_liqele1, ct_liqele1_new, ct_liqele2, ct_liqele2_new, Temperature,
                              points, delta, LiqphaseBoolean, Tempboolean, noise_boolean,
                              Boundary_boolean, Matrix_boolean, fileint, radius, curvaturearray_old,
                                                                                    velocityflag);

                tecplotflag =0;
            } // End of For
        if(Matrix_boolean == 2){
            Matrix_savefunction( counter, i, j, k, max_ij, max_k, gzone,      gzone_new, curvaturearray, gavg, velocity,    Total_time, time_step,
tecplotflag, alloys,
                              ct_liqele1, ct_liqele1_new, ct_liqele2, ct_liqele2_new, Temperature);
        } // End if of matrix boolean

return 0;

} // END OF MAIN



//***************************** VELOCITY FUNCTION ************************************************//

void velocity_function(alloy alloys, int max_ij,int max_k, float const Temperature,
                              double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                              double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                              double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                              double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                                                    int velocityflag, int counter, int headerflag,
                                                                    double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE]){



        // This function calculates the velocity at every cell that is located at the phase boundary
    // The velocity if the phase field = 1.0 or 0 is zero.
```

Phase Evolution – Appendix C

```
// Declaration of variables

int i, j,k, stop;

// Velocity output file

  // Opening of file name
ofstream vel;
vel.open("Velocity.txt", ios::app);

if(!"Velocity.txt"){
      cout<< "Can not open file Velocity.txt \n";
      // Handle error
      cin >> stop;
      exit (0);
}

          if(velocityflag == 2){
                    // Print the headear of hte columns in the output file.
                    if(headerflag == 2){
                              vel << "Iteration_No.";
                              vel.width(20);
                              vel << "Cell_No.";
                              vel.width(20);
                              vel << "Gzone";
                              vel.width(20);
                              vel << "CONCT_LIQ_E1";
                              vel.width(20);
                              vel << "Liq_Slope_E1";
                              vel.width(20);
                              vel << "Supercooling_E1";
                              vel.width(20);
                              vel << "CONCT_LIQ_E2";
                              vel.width(20);
                              vel << "Liq_Slope_E2";
                              vel.width(20);
```

```
                vel << "Supercooling_E2";
                vel.width(20);
                vel << "Curvat,AE_Incl";
                vel.width(20);
                vel << "T.M_Host";
                vel.width(20);
                vel << "Temperaturc";
                vel.width(20);
                vel << "Thermal_Superc";
                vel.width(20);
                vel << "VELOCITY";

                vel << "\n";
                headerflag = 1; // This is done so it only prints in the file once.
            }// End if of Header Flag

        }// End if Velocity Flag == 2



for(i=0; i<=max_ij; i++)
    for(j=0; j<=max_ij; j++)
        for(k=0; k<=max_k; k++){
        velocity[i][j][k] =(float) alloys.kinetic_coefficient*(alloys.Temp_melt_host
                                +ct_liqele1[i][j][k]*alloys.mlele1  // removed the division by 100  because it is all in %wt
                                +ct_liqele2[i][j][k]*alloys.mlele2  // same here100
                                -curvaturearray[i][j][k] - Temperature);
                                // The curvature array already includes the Gibbs Thomson Coefficient

            // In order to determine the effects of each supercooling effect in the kinetic equation
            // For every time step I wll be outputing a files with all the information of the velocity
            // equation. The file name is Velocity.txt
                                    if(velocityflag == 2){
                                        if((i == 38)&&(j== 38)){
```

```
                    if(k == 4){
                            vel << counter;
                            vel.width(27);
                            vel << i << "," << j << "," << k;
                            vel.width(20);
                            vel << gzone[i][j][k];
                            vel.width(20);
                            vel << ct_liqele1[i][j][k];
                            vel.width(20);
                            vel <<alloys.mlele1;
                            vel.width(20);
                            vel << ct_liqele1[i][j][k]*alloys.mlele1      ;
                            vel.width(20);
                            vel <<ct_liqele2[i][j][k] ;
                            vel.width(20);
                            vel << alloys.mlele2;
                            vel.width(20);
                            vel << ct_liqele2[i][j][k]*alloys.mlele2;
                            vel.width(20);
                            vel << curvaturearray[i][j][k];
                            vel.width(20);
                            vel << alloys.Temp_melt_host;
                            vel.width(20);
                            vel << Temperature ;
                            vel.width(20);
                            vel << alloys.Temp_melt_host - Temperature;
                            vel.width(20);
                            vel << velocity[i][j][k];
                            vel << "\n";

                        } // End if k
                    }// End if I, J
                } // End if Velocity flag == 2

    } // End for
} // End of Velocity_function
```

```
//****************************** END OF VELOCITY FUNCTION ********************************************//



//****************************** TIME STEP FUNCTION ***************************************************//

float time_step_function(float const delta, alloy alloys)
{
     float time_step_var2, timestep, Diffiusion;
     //float  time_step_var1;

     Diffiusion = greatest_function(alloys.Diffiusion_liqhost, alloys.Diffiusion_liqele1);

// Calculation of time step according to the stability criteria

     time_step_var2 = (float) (pow(delta,2.0))/(8*Diffiusion);


     timestep = time_step_var2;


     return timestep;

} // End of time_step_function;

//****************************** END OF TIME STEP FUNCTION ********************************************//


//****************************** NEIGHBOUR FUNCTION  **************************************************//

int neigbour_function(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], int i, int j, int k, int index[6], int max_ij, int max_k, int
Boundary_boolean){
// This function checks that a specific gzone value of value zero is allowed to crystallize it does this,
// by checking if any of its neighbours is or not sorrounded by a totally crystallized neigbour with a
```

```
// gzone value of 1.0

    // Declaration of variables used in this function
        int iu, ju, ku, il, jl, kl;
        int neighbour_flag; // If = 1 then it can crystalize if 0 it can not (False)

    // Setting of the BOundary Conditions. This is done just in the case that the dendrite is close to the
    // boundary  of the control volume


        Boundary_Condition(i, j,k, index, max_ij, max_k, Boundary_boolean);
        il = index[0];
        jl = index[1];
        kl = index[2];
        iu = index[3];
        ju = index[4];
        ku = index[5];



    // The function of this function is to add up  the phase field of all the neighbouring cells to the
    // the cell that has a phase field value of zero

            neighbour_flag = 0; // We start assuming that no neighbours are solidified, if that is note
                                // the case then neighbour_flag = 1;
// Middle Surface

        if(gzone[i][ju][k]== 1) {
            neighbour_flag = 1;
        }
        if(gzone[il][j][k]== 1) {
            neighbour_flag = 1;
        }
        if(gzone[iu][j][k]== 1) {
            neighbour_flag = 1;
        }
        if(gzone[i][jl][k]== 1) {
```

```
            neighbour_flag = 1;
        }


// Upper surface;

        if(gzone[i][j][ku]== 1) {
            neighbour_flag = 1;
        }

// Lower surface;
        if(gzone[i][j][kl] == 1) {
            neighbour_flag = 1;
        }

return neighbour_flag;
} // End of Neigbour Function

//***************************** END OF NEIGHBOUR FUNCTION ************************************************//

//***************************** BOUNDARY CONDITION FUNCTION ************************************************//

void Boundary_Condition(int i, int j, int k, int index[6], int max_ij, int max_k,int  Boundary_boolean){

    // Mirror Boundary Condition
    // This function should reflect the same condition as the one found in Curvature 3
    // July 20 th 1998

    index[0] = i-1;
    index[1] = j-1;
    index[2] = k-1;
    index[3] = i+1;
    index[4] = j+1;
    index[5] = k+1;

// Definition of Periodical Boundary Conditions
    if(Boundary_boolean == 1){
```

```
        if(i-1<0)  { index[0]= max_ij;}
        if(j-1<0)  { index[1]= max_ij;}
        if(k-1<0)  { index[2]= max_k;}
        if(i+1>max_ij){index[3] = 0;}
        if(j+1>max_ij){index[4] = 0;}
        if(k+1>max_k){index[5] = 0;}
    } // End if for Boundary_boolean 1 = Periodical

// Definition for Periodical in i and j but NOT k
    if(Boundary_boolean== 0){
        if(i-1<0)  { index[0]= max_ij;}
        if(j-1<0)  { index[1]= max_ij;}
        if(k-1<0)  { index[2]= max_k-2;}
        if(i+1>max_ij){index[3] = 0;}
        if(j+1>max_ij){index[4] = 0;}
        if(k+1>max_k){index[5] = max_k-2;}

    }// End if    0= PERIODICAL IN I,J BUT NOT in K



} // End of function Boundary_condition

//*****************************END OF BOUNDARY CONDITION FUNCTION ***************************************//

//****************************** AVERAGE PHASE FUNCTION ***************************************************//


void Averagephase(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],  float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE],int max_ij, int max_k,
int index[6], int Boundary_boolean){

    float M1, M2,M3,M4, M5, M6, M7, M8, M9,U1, U2,U3, U4, U5, U6,U7, U8,U9,L1, L2,L3, L4, L5, L6,L7, L8,L9; // Weight Factors
    int i, j, k;
    int il, iu;
    int jl, ju;
    int kl, ku;
    double Total;
```

```
double cell;
// Definition of Weight Factors

M1=(float) 0.1;          // Middle Layer
M2=(float) 0.20;         // Middle Layer
M3=(float) 0.1;          // Middle Layer
M4=(float) 0.20;         // Middle Layer
M5=(float) 1.0;          // Middle Layer
M6=(float) 0.20;         // Middle Layer
M7=(float) 0.1;          // Middle Layer
M8=(float) 0.20;         // Middle Layer
M9=(float) 0.1;          // Middle Layer

U1=(float) 0.05;         //Upper Layer
U2=(float) 0.1;          // Upper Layer
U3= (float) 0.05;        // Upper Layer
U4=(float) 0.1;          // Upper Layer
U5=(float) 0.20;         // Upper Layer
U6=(float) 0.1;          // Upper Layer
U7=(float) 0.05;         // Upper layer
U8=(float) 0.1;          // Upper Layer
U9=(float) 0.05;         // Upper Layer

L1=(float) 0.05;
L2=(float) 0.1;          // Lower Layer
L3=(float) 0.05;         // Lower Layer
L4=(float) 0.1;          // lower Layer
L5=(float) 0.20;         // Lower Layer
L6=(float) 0.1;          // Lower Layer
L7=(float) 0.05;         // Lower Layer
L8=(float) 0.1;          // Lower Layer
L9=(float) 0.05;         // Lower Layer

Total = M1+M2+M3+M4+M5+M6+M7+M8+M9+  U1+U2+ U3+U4+ U5+ U6+ U7+U8+U9+L1+L2+L3+L4+L5+L6+L7+L8+L9;

for(i=0; i <= max_ij; i++)
```

```
for(j=0; j <= max_ij; j++)
    for(k = 0; k <= max_k; k++){

// This part of the program calculates the average phase field for all cell.
    /*   il = i-1;
         jl = j-1;
         kl = k-1; */


         Boundary_Condition(i,j,k,index, max_ij, max_k, Boundary_boolean);

         il = index[0];
         jl = index[1];
         kl = index[2];
         iu = index[3];
         ju = index[4];
         ku = index[5];



// This is the averaging function

cell = M1*gzone[il][ju][k]      + M2*gzone[i][ju][k]  + M3*gzone[iu][ju][k]
             + M4*gzone[il][j][k]   + M5*gzone[i][j][k]   + M6*gzone[iu][j][k]
             + M7*gzone[il][jl][k]  + M8*gzone[i][jl][k]  + M9*gzone[iu][jl][k];

             // Upper surface;
cell = cell +  U1*gzone[il][ju][ku]+   U2*gzone[i][ju][ku] + U3*gzone[iu][ju][ku]
             +  U4*gzone[il][j][ku] +    U5*gzone[i][j][ku]  + U6*gzone[iu][j][ku]+
             +  U7*gzone[il][jl][ku]+   U8*gzone[i][jl][ku] + U9*gzone[iu][jl][ku];

             // Lower surface;

             cell = cell +L1*gzone[il][ju][kl]+     L2*gzone[i][ju][kl] + L3*gzone[iu][ju][kl]
             +  L4*gzone[il][j][kl] +    L5*gzone[i][j][kl]  + L6*gzone[iu][j][kl]+
             +  L7*gzone[il][jl][kl]+   L8*gzone[i][jl][kl] + L9*gzone[iu][jl][kl];
```

```
                    gavg[i][j][k] = cell/Total;
    } // Enf of For loop


} // End of function Averagephase

//****************************** END OF AVERAGE PHASE FUNCTION ***************************************************//

//************************************** CURVATURE FUNCTION ***************************************************//

void curvature_function (double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE], int index[6], int max_ij, int max_k, int points,
                         float delta, float radius, alloy alloys,int Boundary_boolean, float
curvaturearray_old[MAX_SIZE][MAX_SIZE][MAX_SIZE])
{

// Definition of Local Variables in the Curvature Function
int i;
int j;          // Indices
int k;
int const min=0;
int flagtotal;
int il, iu;     // il = i -1, jl = j-1, and kl = k-1 while iu = i+1, ju = j+1 and ku= k+1
int jl, ju;
int kl, ku, passed, failed; // Passed fail represent the number of nodes that have a phase field and
                            // the curvature is between 30%
int passed50, passed60;
int space;
int Total_points;

// Definition of Float variables in the Curvature Function

double gx;      // partial derivative of the gzone wrt to x
double gy;      // partial derivative of the gzone wrt to y
double gz;      // partial derivative of the gzone wrt to z
```

```
double gxx;
double gyy;          // second partial derivatives of the gzone wrt to x, y and z
double gzz;
double gxy;
double gyz;          // Second partial derivatives of the gzone
double gxz;
double curvature;// curvature at the point x, y,  and z.
double dofcurvature;          // Denominator of curvature is it a dummy variable.
double numerator;
double denominatortopow;
float Cofcell;

float upperlimit, lowerlimit;
float upperlimit50, lowerlimit50;
float upperlimit60, lowerlimit60;
double nx, ny, nz;
double theta, phi;
double Ani_cst;
double gibbst;
double denominator_normal;


// Requirement code for  and output to files

     ofstream out;   // output


// Initialization of variables
          space = 12;
          flagtotal = 2;
          passed = 0;
          failed = 0;
          passed60 =0;
          passed50 =0;

          gx=0.0;
          gy=0.0;
```

```
gz=0.0;
gxx=0.0;
gyy=0.0;
gzz=0.0;
gxy=0.0;
gyz=0.0;
gxz=0.0;
curvature=99;
dofcurvature=0.0;
numerator=0.0;
denominatortopow=0.0;
Cofcell=0.0;

upperlimit=0.0;
lowerlimit=0.0;
upperlimit50=0.0;
lowerlimit50=0.0;
upperlimit60=0.0;
lowerlimit60=0.0;
Total_points =0;

Ani_cst = 0.25;

// Printing the headres of each column into its respective files


// Averaging function of the Phase field is called


        Averagephase(gzone, gavg,max_ij, max_k, index, Boundary_boolean);

// This section calculates the components in order to calculate the curvature value
// Calculation of the partial derivative of the gavg[x][y][z] with respect to x, y, z
// for only those points in the phase field which are NOT COMPLETLY Solid or liquid (1 or 0)

for(i=min; i <= max_ij; i++)
    for(j=min; j <= max_ij; j++)
```

```
for(k =min; k <= max_k; k++){


// Boundary Conditions

Boundary_Condition(i,j,k,index, max_ij, max_k, Boundary_boolean);

    il = index[0];
    jl = index[1];
    kl = index[2];
    iu = index[3];
    ju = index[4];
    ku = index[5];


// The curvature function is only calculated for those cells that are not equal to 0 nor equal to 1.0

    // Initialization of Flags
    flagtotal = 2;

if((gzone[i][j][k]!=0)&&(gzone[i][j][k]!=1)){


    //Gradient calculation
    gx = (gavg[iu][j][k]- gavg[il][j][k])/((float)2.*delta);
    gy = (gavg[i][ju][k]- gavg[i][jl][k])/((float)2.*delta);
    gz = (gavg[i][j][ku]- gavg[i][j][kl])/((float)2.*delta);



    if ((gx ==1e-06)&&(gy ==1e-06)&&(gz==1e-06)){
        flagtotal = 1;
        // This line is necessary to have a new line in the outdata file

    }
```

```
if (gavg[i][j][k] ==0) {
      flagtotal =1;
      // This insures that the curvature will not be calculated if the cell that we are analysing is completed surrounded
      // by solid cell or completely surrounded by liquid cells.

}
if (gavg[i][j][k] == 1) {
      flagtotal =1;
      // This insures that the curvature will not be calculated if the cell that we are analysing is completed surrounded
      // by solid cell or completely surrounded by liquid cells.

}

if(flagtotal != 1){

// Calculation of the second partial derivatives of the gzone[x][y][z] with respect to x, y, z
      gxx = (gavg[iu][j][k]-(float) 2.*gavg[i][j][k]+ gavg[il][j][k])/(delta*delta);
      gyy = (gavg[i][ju][k]-(float) 2.*gavg[i][j][k]+ gavg[i][jl][k])/(delta*delta);
      gzz = (gavg[i][j][ku]-(float) 2.*gavg[i][j][k]+ gavg[i][j][kl])/(delta*delta);


// Calculation of the second partial derivatives of the gzone[x][y][z] wrt x and y

      gxy = (gavg[iu][ju][k] -gavg[il][ju][k]- gavg[iu][jl][k]+gavg[il][jl][k])/((float)4.*delta*delta);


// Calculation of the second partial derivatives of the gzone[x][y][z] wrt y and z

      gyz = (gavg[i][ju][ku] - gavg[i][jl][ku]- gavg[i][ju][kl]+ gavg[i][jl][kl])/((float)4.*delta*delta);



// Calculation of the second partial derivatives of the gzone[x][y][z] wrt x and z

      gxz = (gavg[iu][j][ku] - gavg[il][j][ku]- gavg[iu][j][kl]+gavg[il][j][kl])/((float)4.*delta*delta);
```

```
// Calculation of curvature at that point:

numerator = pow(gx,2.)*(gyy+gzz)+pow(gy,2.)*(gxx+gzz)+pow(gz,2.)*(gxx+gyy)-2.*(gx*gy*gxy+gx*gz*gxz+gy*gz*gyz);
dofcurvature =(double)pow(gx,2.) + (double)pow(gy,2.) + (double)pow(gz,2.);
denominatortopow =(float) pow(dofcurvature,1.5);
curvature =(float)(-1*numerator)/(2*denominatortopow);
        curvaturearray_old[i][j][k] = curvature;

// ANISOTROPY EFFCTS

    // This part of the function takes anisotropy effects into account
    // Calculation of the unit vector in x, y and z directions
    denominator_normal =  pow(dofcurvature, 0.5);
    nx = (gx)/(denominator_normal);
    ny = (gy)/(denominator_normal);
    nz = (gz)/(denominator_normal);

    //Calculation of theta and phi
    // Theta is the angle between the normal and the surface (component in the z direccion)
    // while phy is the component in x

    phi = acos(nz);
    theta = acos(nx);

    // Calculation of modified Gibbs-thomson coefficinet

    gibbst= alloys.Gibbs_thomson*(1- Ani_cst*cos(4*phi)-Ani_cst*cos(4*theta));

    curvaturearray[i][j][k] = curvature*gibbst;



// Calculation of upper ald lower limits of the error for 30% margin
upperlimit = radius + radius*(float)0.3;
lowerlimit = radius - radius*(float)0.3;
upperlimit50 = radius + radius*(float)0.5;
lowerlimit50 = radius - radius*(float)0.5;
```

```
upperlimit60 = radius + radius*(float)0.6;
lowerlimit60 = radius - radius*(float)0.6;



} // If statement of curvature calculation if flagtotal != 1;


// Reinitialization of variables for next cell

    il= 99;
    jl= 99;
    kl= 99;
    iu = 99;
    ju = 99;
    ku = 99;
    curvature = 99.0;
    flagtotal = 3;



} // End of if statement of gzone[i][j][k] != 0 nor !=1.0

// Output to a cuvature output to be read by another function and calculate velocity
if((flagtotal != 3)&&(flagtotal !=1)){

    curvaturearray[i][j][k] = 0.0;
    flagtotal = 2;
}// End if Flagtotal = 1;que

} // End of curvature for loop


// Output of statistics to file
Total_points = passed+passed50+passed60+failed;

}// End of curvature function
```

```
//****************************** END OF CURVATURE  FUNCTION ***********************************************//

//****************************** FILE SAVE FUNCTION ***********************************************//


void filesave_function(int counter, int i, int j, int k, int max_ij, int max_k,  double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE], double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       float Total_time, float time_step, int tecplotflag, alloy alloys,
                       double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                       double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],float  Temperature,
                       int points, float delta, int LiqphaseBoolean, int Tempboolean, int noise_boolean,
                       int Boundary_boolean, int Matrix_boolean, int fileint, float radius,
                                                                              float
curvaturearray_old[MAX_SIZE][MAX_SIZE][MAX_SIZE], int velocityflag){


    // Variable Declaration for Filesave_function
        char fileName[40];
        char *baseFileName="newphase"; // Base File Name
        char stop;

    // Editing of file name command.

    sprintf (fileName, "%s%d.txt", baseFileName, counter);

    // Opening of file name
    fstream out;
    out.open(fileName, ios::app);

    if(!fileName){
        cout<< "Can not open file \n";
        // Handle error
        cin >> stop;
```

```
    exit (0);
}



// Information that we need to save at this time step;

if(tecplotflag ==1)

{
    out << "# ";
    out.width(20);
    out <<"PHYSICAL_&_MATERIAL_PROPERTIES"<< "\n";

    out << "# ";
    out.width(20);
    out << counter;
    out.width(16);
    out <<"Counter"<< "\n";

    out << "# ";
    out.width(20);
    out <<  Temperature;
    out.width(20);
    out << "Temperature" << "\n";

    out << "# ";
    out.width(20);
    out <<  Total_time;
    out.width(19);
    out << "Total_Time" << "\n";

    out << "# ";
    out.width(20);
    out <<  time_step;
    out.width(19);
```

```
out << "Time_Step " <<"\n";

out << "# ";
out.width(20);
out <<  alloys.kinetic_coefficient;
out.width(28);
out << "Kinetic_Coefficient" << "\n";

out << "# ";
out.width(20);
out <<  alloys.Gibbs_thomson;
out.width(34);
out << "Gibbs_Thomson_Coefficient" << "\n";

out << "# ";
out.width(20);
out << alloys.mlele1;
out.width(34);
out << "Solpe_of_Liquid_element_1" << "\n";

out << "# ";
out.width(20);
out << alloys.msele1;
out.width(33);
out << "Solpe_of_Solid_element_1" << "\n";

out << "# ";
out.width(20);
out << alloys.msele2;
out.width(33);
out << "Solpe_of_Solid_element_2" << "\n";

out << "# ";
out.width(20);
out << alloys.mlele2;
out.width(34);
out << "Solpe_of_Liquid_element_2" << "\n";
```

```
out << "# ";
out.width(20);
out << alloys.heat_capacity;
out.width(22);
out << "Heat_Capacity" << "\n";

out << "# ";
out.width(20);
out << alloys.latent_heat;
out.width(20);
out << "Latent_Heat" << "\n";

out << "# ";
out.width(20);
out << alloys.Diffiusion_liqhost;
out.width(31);
out << "Diffiusion_Liquid_Host" << "\n";

out << "# ";
out.width(20);
out << alloys.Diffiusion_liqele1;
out.width(36);
out << "Diffiusion_Liquid_Element_1" << "\n";

out << "# ";
out.width(20);
out << alloys.Diffiusion_liqele2;
out.width(36);
out << "Diffiusion_Liquid_Element_2" << "\n";

out << "# ";
out.width(20);
out << alloys.Temp_melt_host;
out.width(30);
out << "Host_Melt_temperature" << "\n";
```

```
out << "# ";
out.width(20);
out << points;
out.width(44);
out << "Points_used_for_initial_phase_field" << "\n";

out << "# ";
out.width(20);
out << delta;
out.width(19);
out << "Grid_delta" << "\n";

out << "# ";
out.width(20);
out << max_ij;
out.width(29);
out << "Max_value_in_I_and_J" << "\n";

out << "# ";
out.width(20);
out << max_k;
out.width(23);
out << "Max_value_in_K" << "\n";

out << "# ";
out.width(20);
out << LiqphaseBoolean;
out.width(36);
out << "Liquid_phase_only_<1=Y|2=N>" << "\n";

out << "# ";
out.width(20);
out << noise_boolean;
out.width(40);
out << "Noise_added_to_system_<1=Y|2=N>" << "\n";
```

```
out << "# ";
out.width(20);
out << Tempboolean;
out.width(28);
out << "Temperature_boolean" << "\n";

out << "# ";
out.width(20);
out << Boundary_boolean;
out.width(61);
out << "B.C_<0=Periodical_in_I,J_not_in_K_|_1=All_periodical" << "\n";

out << "# ";
out.width(20);
out << Matrix_boolean;
out.width(45);
out << "Phase_field_in_Matrix_form_<1=N|2=Y>" << "\n";

out << "# ";
out.width(20);
out << fileint;
out.width(52);
out << "Every_how_many_number_of_Iterations_to_save" << "\n";

out << "# ";
out.width(20);
out << radius;
out.width(33);
out << "Radius_of_initial_sphere" << "\n";

                        out << "# ";
out.width(20);
out << velocityflag;
out.width(43);
out << "Velocity_info_in_file:<1=No|2=Yes>" << "\n";
```

```
                    out << "# ";
out.width(20);
out << alloys.kdistr;
out.width(33);
out << "Distribuition Coeficient" << "\n";


out <<"\n";




// FROM THIS SECTION WE START PRINTING THE DATA THAT WILL BE READ BY TECPLOT
// ALL THE INFORMATION PREVIOUS TO THIS WHERE JUST COMMENTS AND MATERIAL PROPERTIES
// THAT ARE ONLY READ IF WE ARE RECOVERING FROM A FAILED OR A STOPPED RUN.

                    out << "i";
                    out.width(10);
                    out << "j";
                    out.width(10);
                    out << "k";
                    out.width(18);
                    out << "GZONE";
                    out.width(18);
                    out << "GAVG";
                    out.width(18);
                    out << "NEW GZONE";
                    out.width(18);
                    out << "CURV-GIBB"; // Thus the old phase field remains and it does not evolve
                    out.width(18);
                                                                    out << "CURVATURE";
                                                                    out.width(18);
                    out << "VELOCITY";
                    out.width(18);
                    out << "Conct Si";
```

```
                    out.width(18);
                    out << "Conct Si-New";
                    out.width(18);
                    out <<  "Ct Cu";
                    out.width(18);
                    out << "Ct Cu-NEW";
                    out << "\n";

                    out<< "ZONE I= " << max_k+1 << ", J= " << max_ij+1 << ", k= "<< max_ij+1 <<", F=POINT" << "\n";
                    tecplotflag =0;
} // End If

out << i;
out.width(10);
out << j;
out.width(10);
out << k;
out.width(18);
        out << gzone[i][j][k];
out.width(18);
out << gavg[i][j][k];
out.width(18);
out << gzone_new[i][j][k];
out.width(18);
out << curvaturearray[i][j][k]; // Thus the old phase field remains and it does not evolve
        out.width(18);
        out << curvaturearray_old[i][j][k];
out.width(18);
out << velocity[i][j][k];
out.width(18);
out << ct_liqele1[i][j][k];
out.width(18);
out << ct_liqele1_new[i][j][k];
out.width(18);
out << ct_liqele2[i][j][k];
out.width(18);
out << ct_liqele2_new[i][j][k];
```

```
    out << "\n";

    // Closing of file name;


out.close();

}// End of filesave_function

//****************************** END OF FILE SAVE FUNCTION ***************************************************//

//********************************** GREATEST FUNCTION ***************************************************//


float  greatest_function(float Item1, float Item2){

    float biggest;

    // This function returns the biggest of 3 parameters

    if(Item1 >= Item2)
        { biggest = Item1;
        }
    else{ biggest = Item2;};

//    if(Item3 > biggest){
//        biggest = Item3;}

    return biggest;

} // End of Function Greatest_function;

//******************************** END OF GREATEST FUNCTION ***************************************************//

//******************************** PHASE EVOLUTION  FUNCTION ***************************************************//

void phase_evolution(int max_ij,int max_k, double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],
```

```
                    double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                    float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE], double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE], float
Total_time,
                    float time_step, int tecplotflag, int counter, float delta, int fileint, int index[6], int Boundary_boolean,
                    double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                    double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                    alloy alloys, float Temperature){


// Intialization of variables

    int i, j, k;
    int crystallize_flag;
    float time;
    double ct_liqele1_mgap, ct_liqele2_mgap, ct_solele1, Miscibility_gap, phi, nd_slope, nd_conct_1, nd_conct_2;
    double nd_k, ct_solele2, max_diff_gzone_num, max_diff_gzone_dinom, max_diff_gzone;


    if(fileint == 0){
        cin >> fileint;
    }



// Evolution of the phase field
    for(i=0; i<=max_ij; i++)
        for(j=0; j<=max_ij; j++)
            for(k=0; k<=max_k; k++){

            /* "Crystallization can take place in cells with 0< gzone <1 or in cells with gzone = 0
            having at least one completely crystallized neighbour cell. " From Prof. A. Artemev article.*/

            crystallize_flag = neigbour_function(gzone, i, j, k, index, max_ij, max_k, Boundary_boolean);

            time = time_step; // Here we assign a local variable the time step calculated by the stability
                              // criteria.
```

Phase Evolution – Appendix C

```
if (((gzone[i][j][k] ==0.0)&&(crystallize_flag==1))||((gzone[i][j][k]> 0.0)&&(gzone[i][j][k]<1.0))){

// Calculation of Maximum phase field change allowed:

    //Calculation of Miscibility gap;


        ct_liqele1_mgap = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele1));
        ct_liqele2_mgap = ((Temperature - alloys.Temp_melt_host)/(alloys.mlele2));
        ct_solele1 = ((Temperature - alloys.Temp_melt_host)/(alloys.msele1));

        Miscibility_gap = ct_liqele1_mgap - ct_solele1;



    //Calculation of phi;
    phi = (Temperature - alloys.Temp_melt_host)/(Miscibility_gap*alloys.mlele1);


    // Calculation of non dimensional slope
    nd_slope = alloys.mlele2/ alloys.mlele1;

    // Calculation of non dimensional concentration one

    nd_conct_1 = ct_liqele1[i][j][k]/Miscibility_gap;

    // Calculation of non dimensional concentration two.

    nd_conct_2 = ct_liqele2[i][j][k]/Miscibility_gap;


    // Calculation of non dimensional curvature

    nd_k = curvaturearray[i][j][k]/(alloys.mlele1*Miscibility_gap);
```

Phase Evolution – Appendix C

```
// Calculation of the solid concentration of element 2 based on the linear
// system of the phase diagram

ct_solele2 = ((Temperature - alloys.Temp_melt_host)/(alloys.msele2));

// Calculation of Maximum Change in phase field allowed

max_diff_gzone_num = (phi - nd_conct_1 - nd_slope*nd_conct_2 - nd_k)*(0.5);

max_diff_gzone_dinom = (phi - nd_conct_1 - nd_slope*nd_conct_2 - nd_k) +
                       (nd_conct_1*(1-(ct_solele1/ct_liqele1_mgap))) +
                       (nd_slope*nd_conct_2*(1-(ct_solele2/ct_liqele2_mgap)));

max_diff_gzone = max_diff_gzone_num / max_diff_gzone_dinom;

// Calculation of the change in phase field

    gzone_new[i][j][k] = gzone[i][j][k] + (time*velocity[i][j][k]/delta);

// Checking that the change in phase field is not greater than the thermodynamically
// allowed.

if(fabs((gzone_new[i][j][k] - gzone[i][j][k])) > fabs(max_diff_gzone)){


    gzone_new[i][j][k] = gzone[i][j][k] + max_diff_gzone;

}// End if of Change in phase field being greater than max_diff_gzone



// Here we need to check that we have not gone over the aceptable phase field value

if(gzone_new[i][j][k] >=1.0){          // Then the new value of phase field for that cell has become to great
    gzone_new[i][j][k] = 1.0;
}// end if
```

```
                if(gzone_new[i][j][k] < 0.0){
                    // Then remelting has ocurred and the cell is completely liquid
                    gzone_new[i][j][k] = 0.0;
                } // end if

            } // End if This ensures that the evolution of the phase field only happens in cells that are not
              // all solidified or are not all liquid.
            else{

                }
            //    tecplotflag =0;
        } // End for

} // End of Phase field Evolution

//****************************** END OF PHASE FIELD EVOLUTION FUNCTION ************************************************//
//****************************** CONCENTRATION FUNCTION ***************************************************//


void Concentration_function(int max_ij, int max_k, float time_step, float delta, double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], alloy alloys, float Temperature, int index[6],
                double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                int Boundary_boolean){



// Definition of variables variables

    int i, j, k; // Counters
    int iu, ju, ku;
    int il, jl, kl;
    double liq_perct;
    double Excess_ele1, Excess_ele2;
    float theta_iu;
```

```
float theta_il;
float theta_ju;
float theta_jl;
float theta_ku;
float theta_kl;
float Diffiusion;
double change_in_phasefield;



// This function only (08-23-98) takes into a ccount diffusion in liquid

Diffiusion = alloys.Diffiusion_liqhost;

// Concentration evolution

for(i=0; i<=max_ij; i++)
    for(j=0; j<=max_ij; j++)
        for(k=0; k<=max_k; k++){

if(gzone_new[i][j][k] == 1.0){
    // IF for any reason the new position of the phase field for that cell has completely
    // solidified then the concentration at that cell is zero.
    ct_liqele1_new[i][j][k] =0;
    ct_liqele2_new[i][j][k] =0;
} // End if



// This section defines concentration in liquid or fraction of liquid, solid phase.
// If the concentration of gzone is = 1 then this complete section would be skipped.

if(gzone_new[i][j][k] != 1.0){
    // The concentration functions is only for cells with gzone value of 0 to 1 but not 1. .
```

```
Boundary_Condition(i,j,k, index, max_ij, max_k, Boundary_boolean);

    il = index[0];
    jl = index[1];
    kl = index[2];
    iu = index[3];
    ju = index[4];
    ku = index[5];


change_in_phasefield = gzone_new[i][j][k]-gzone[i][j][k];

if(1-gzone[i][j][k]<= 0.50000){

      // Diffusion in Liquid of Element 1 Silicon

    ct_liqele1_new[i][j][k] =  ct_liqele1[i][j][k]+ (time_step*Diffiusion/(float)pow(delta,2.0))*(
            (ct_liqele1[iu][j][k] + ct_liqele1[il][j][k] -2*ct_liqele1[i][j][k])+
            (ct_liqele1[i][ju][k] + ct_liqele1[i][jl][k] -2*ct_liqele1[i][j][k])+
            (ct_liqele1[i][j][ku] + ct_liqele1[i][j][kl] -2*ct_liqele1[i][j][k]));



      // Diffusion in Liquid of Element 2 Cupper


    ct_liqele2_new[i][j][k] = ct_liqele2[i][j][k]+  (time_step*Diffiusion/(float)pow(delta,2.0))*(
            (ct_liqele2[iu][j][k] + ct_liqele2[il][j][k] -2*ct_liqele2[i][j][k])+
            (ct_liqele2[i][ju][k] + ct_liqele2[i][jl][k] -2*ct_liqele2[i][j][k])+
            (ct_liqele2[i][j][ku] + ct_liqele2[i][j][kl] -2*ct_liqele2[i][j][k]));


  if (change_in_phasefield >=0.0){
```

```
// If the fraction of solid is large, then the solute transfer to all neighboring cells
// should be taken into account

Excess_ele1 = (gzone_new[i][j][k] - gzone[i][j][k])*(ct_liqele1[i][j][k]*(1-alloys.kdistr));

Excess_ele2 = (gzone_new[i][j][k] - gzone[i][j][k])*(ct_liqele2[i][j][k]*(1-alloys.kdistr));



// This excess solute is required to be re-distributed through the liquid part of the cell volume
// In order to do that we need to determine the total amount of liquid around the cell in which we
// can distribute the excess solute.

liq_perct = (1-gzone[il][j][k])+ (1-gzone[iu][j][k])+(1-gzone[i][ju][k])+(1-gzone[i][jl][k])+
            (1-gzone[i][j][k]);



// Upper surface;

liq_perct = liq_perct +  (1-gzone[i][j][ku]);


// Lower surface;

liq_perct = liq_perct  +(1-gzone[i][j][kl]);
```

```
                    // Excess solute redistribution of Element 1 (Silicon);

// Middle Neighbours Elelement 1

ct_liqele1_new[i][ju][k] = ct_liqele1_new[i][ju][k] +(1-gzone[i][ju][k])*Excess_ele1/(liq_perct);
ct_liqele1_new[il][j][k] = ct_liqele1_new[il][j][k] +(1-gzone[il][j][k])*Excess_ele1/(liq_perct);
ct_liqele1_new[i][j][k]  = ct_liqele1_new[i][j][k]  +(1-gzone[i][j][k])*Excess_ele1/(liq_perct);
ct_liqele1_new[iu][j][k] = ct_liqele1_new[iu][j][k] +(1-gzone[iu][j][k])*Excess_ele1/(liq_perct);
ct_liqele1_new[i][jl][k] = ct_liqele1_new[i][jl][k] +(1-gzone[i][jl][k])*Excess_ele1/(liq_perct);



// Upper Neighbours Element 1
ct_liqele1_new[i][j][ku] = ct_liqele1_new[i][j][ku]  +(1-gzone[i][j][ku])*Excess_ele1/(liq_perct);



// Lower Neighbours     Element 1
ct_liqele1_new[i][j][kl] =ct_liqele1_new[i][j][kl]  +(1-gzone[i][j][kl])*Excess_ele1/(liq_perct);


// Excess solute redistribution of Element 2 (Copper);

// Middle Neighbours Elelement 2
ct_liqele2_new[i][ju][k] = ct_liqele2_new[i][ju][k] +(1-gzone[i][ju][k])*Excess_ele2/(liq_perct);
ct_liqele2_new[il][j][k] = ct_liqele2_new[il][j][k] +(1-gzone[il][j][k])*Excess_ele2/(liq_perct);
ct_liqele2_new[i][j][k]  = ct_liqele2_new[i][j][k]  +(1-gzone[i][j][k])*Excess_ele2/(liq_perct);
ct_liqele2_new[iu][j][k] = ct_liqele2_new[iu][j][k] +(1-gzone[iu][j][k])*Excess_ele2/(liq_perct);
ct_liqele2_new[i][jl][k] = ct_liqele2_new[i][jl][k] +(1-gzone[i][jl][k])*Excess_ele2/(liq_perct);



// Upper Neighbours Element 1
ct_liqele2_new[i][j][ku] = ct_liqele2_new[i][j][ku]  +(1-gzone[i][j][ku])*Excess_ele2/(liq_perct);
```

```
// Lower Neighbours     Element 1
ct_liqele2_new[i][j][kl]  =ct_liqele2_new[i][j][kl]  +(1-gzone[i][j][kl])*Excess_ele2/(liq_perct);


                    if(ct_liqele1_new[i][j][k]< 0){
                        cout << " Problem of Si is less than zero" << "\n";
                    }

                    if(ct_liqele2_new[i][j][k]< 0){
                        cout << " Problem of Cu is less than zero" << "\n";
                    }



                } // if change field is > 0


if(change_in_phasefield < 0.0){


    // If the fraction of solid is large, then the solute transfer to all neighboring cells
                    // should be taken into account


    Excess_ele1 = (gzone_new[i][j][k] - gzone[i][j][k])*(ct_liqele1[i][j][k]*(1-alloys.kdistr));


    ct_liqele1_new[i][j][k]  = ct_liqele1_new[i][j][k]  +(1-gzone[i][j][k])*Excess_ele1;


            Excess_ele2 = (gzone_new[i][j][k] - gzone[i][j][k])*(ct_liqele2[i][j][k]*(1-alloys.kdistr));
```

```
ct_liqele2_new[i][j][k]  = ct_liqele2_new[i][j][k]  +(1-gzone[i][j][k])*Excess_ele2;



            // Excess solute redistribution of Element 1 (Silicon);


            if(ct_liqele1_new[i][j][k]< 0){
                cout << " Problem of Si is less than zero" << "\n";
            }

            if(ct_liqele2_new[i][j][k]< 0){
                cout << " Problem of Cu is less than zero" << "\n";
            }




        } // End if of change field
    }
else{// end if of (1-gzone) <=0.5


        theta_il = theta_function(gzone, il, j, k);
        theta_jl = theta_function(gzone, i, jl, k);
        theta_kl = theta_function(gzone, i, j, kl);
        theta_iu = theta_function(gzone, iu, j, k);
        theta_ju = theta_function(gzone, i, ju, k);
        theta_ku = theta_function(gzone, i, j, ku);

        ct_liqele1_new[i][j][k] = ct_liqele1[i][j][k]+ (Diffiusion*time_step/(1-gzone_new[i][j][k]))*(
                (ct_liqele1[iu][j][k]-ct_liqele1[i][j][k])*(theta_iu)/(float)pow(delta, 2.0) -
                (ct_liqele1[i][j][k] - ct_liqele1[il][j][k])*(theta_il)/(float)pow(delta, 2.0) +
```

```
                (ct_liqele1[i][ju][k] - ct_liqele1[i][j][k])*(theta_ju)/(float)pow(delta, 2.0) -
                (ct_liqele1[i][j][k] - ct_liqele1[i][jl][k])*(theta_jl)/(float)pow(delta,2.0)  +
                (ct_liqele1[i][j][ku] - ct_liqele1[i][j][k])*(theta_ku)/(float)pow(delta, 2.0) -
                (ct_liqele1[i][j][k] - ct_liqele1[i][j][kl])*(theta_kl)/(float)pow(delta, 2.0)) +
                ((gzone_new[i][j][k] - gzone[i][j][k])*ct_liqele1[i][j][k]*(1-alloys.kdistr))/(1-gzone_new[i][j][k]);


        ct_liqele2_new[i][j][k] = ct_liqele2[i][j][k]+ (Diffiusion*time_step/(1-gzone_new[i][j][k]))*(
                (ct_liqele2[iu][j][k]-ct_liqele2[i][j][k])*(theta_iu)/(float)pow(delta, 2.0) -
                (ct_liqele2[i][j][k] - ct_liqele2[il][j][k])*(theta_il)/(float)pow(delta, 2.0) +
                (ct_liqele2[i][ju][k] - ct_liqele2[i][j][k])*(theta_ju)/(float)pow(delta, 2.0) -
                (ct_liqele2[i][j][k] - ct_liqele2[i][jl][k])*(theta_jl)/(float)pow(delta,2.0)  +
                (ct_liqele2[i][j][ku] - ct_liqele2[i][j][k])*(theta_ku)/(float)pow(delta, 2.0) -
                (ct_liqele2[i][j][k] - ct_liqele2[i][j][kl])*(theta_kl)/(float)pow(delta, 2.0)) +
                ((gzone_new[i][j][k] - gzone[i][j][k])*ct_liqele2[i][j][k]*(1-alloys.kdistr))/(1-gzone_new[i][j][k]);




            if(ct_liqele1_new[i][j][k]< 0){
                cout << " Problem of Si is less than zero" << "\n";
            }

            if(ct_liqele2_new[i][j][k]< 0){
                cout << " Problem of Cu is less than zero" << "\n";
            }


        }// End of else
} // End gzone not equal to 1.



}// End of for loop
```

```
} // End of Concentration Function

//****************************** END OF CONCENTRATION FUNCTION ************************************************//

//****************************************** THETA FUNCTION ***************************************************//


float theta_function(double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], int x, int y, int z){

    float theta;
    // This function returns the values of theta (the basis function);
    //Initialization of theta
    theta = 99;

    if((1-gzone_new[x][y][z])<= 0.0){theta = 0;};
    if((1-gzone_new[x][y][z])> 0.0) {theta = 1.0;};

    return theta;

return 0;
} // End of theta_function;

//****************************** END OF THETA FUNCTION FUNCTION ************************************************//

//****************************** TEMPERATURE RANGE FUNCTION ***************************************************//

int Temperature_range(alloy alloys, float Temperature, double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                      double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE]){

double Temp_allowed_liquid, Temp_allowed_solid, TempD_liq, TempD_sol;
float stop;
int Temp;

    Temp_allowed_liquid = alloys.Temp_melt_host + alloys.mlele1*ct_liqele1[0][0][0] + alloys.mlele2*
                          ct_liqele2[0][0][0];

    TempD_liq = Temp_allowed_liquid - Temperature;
```

```
if ( TempD_liq < .1){
     cout << " The temp. Diff calc (K) of the allowed liquidus temp: " << TempD_liq << "\n";
     cout << " Liquidus Temperature calculated: " << Temp_allowed_liquid << "\n";
     cout << " Temperature provided by you: " << Temperature << "\n";
     cout << " You will be required to input a lower temperature value"<< "\n";
     TempD_liq = 0;
     cout << " Press any number to continue: " << "\n";
     cin >> stop;

}// End if


Temp_allowed_solid  = alloys.Temp_melt_host + alloys.msele1*ct_liqele1[0][0][0] + alloys.msele2*ct_liqele2[0][0][0];

TempD_sol =Temperature-Temp_allowed_solid;

if (TempD_sol < 0.1){
     cout << " The temp. Diff (K) of the allowed solid temp: " << TempD_sol<< "\n";
     cout << " Solidus Temperature calculated: " << Temp_allowed_solid << "\n";
     cout << " Liquidus Temperature calculated: " << Temp_allowed_liquid << "\n";
     cout << " Temperature provided by you: " << Temperature << "\n";
     cout << " You will be required to input a lowe temperature value";
     TempD_sol = 0;
     cout << " Press any number to continue: " << "\n";
     cin >> stop;

}// End if


if((TempD_liq >=0.1)&&(TempD_sol >= 0.1)){

     Temp = 1;

} // End if
else{
     Temp = 0;
```

```
    } // End of else

return Temp;

} // End of Temperature range function


//****************************** END OF TEMPERATURE RANGE FUNCTION ***********************************************//

//****************************** MATRIX SAVE FUNCTION ************************************************//


void Matrix_savefunction(int counter, int i, int j, int k, int max_ij, int max_k,  double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double curvaturearray[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         float gavg[MAX_SIZE][MAX_SIZE][MAX_SIZE], double velocity[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         float Total_time, float time_step, int tecplotflag, alloy alloys,
                         double ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],float  Temperature){

    // Variable Declaration for Filesave_function
        char fileName[40];
        char *baseFileName="Matrix"; // Base File Name
        char stop;
        int n; // Counter

    // Editing of file name command.

    sprintf (fileName, "%s%d.txt", baseFileName, counter);

    // Opening of file name
    ofstream out;
    out.open(fileName, ios::app);

    if(!fileName){
```

```
        cout<< "Can not open file \n";
        // Handle error
        cin >> stop;
        exit (0);
}



// Information that we need to save at this time step;

if(tecplotflag ==1)

{
        out << "#" << "Temperature: " << Temperature << " Total Time: " <<Total_time << "\n";


                                out << "Plane i";
                                out << "\n";
                                tecplotflag =0;
} // End If
for(i=0; i <= max_ij; i++){
        out <<"Plane " << i << "\n";
         //This is to creat the tile region at the top
        for(n=0; n<= max_ij; n++){
        out.width(5);
        out << "L"<< n;
        if(n==max_ij){ out << "\n";}
        } // end of for n loop
        for (k=0; k<=max_k; k++)
                for(j=0; j<=max_ij; j++){
                        if(j < 10){out.width(10);}
                        if(j>= 10){out.width(11);}
                        out << gzone[i][j][k];
                        if(j== max_ij){out << "\n";}
                        if(k== max_k){out << "\n";}
                }// double for j and k
```

```
    } // End for i


}// End of Matrixsave_function

//****************************** END OF MATRIX SAVE FUNCTION ************************************************//

//****************************** SHIFT DOWN FUNCTION ********************************************//
void Shift_down_function(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE],double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double
ct_liqele1[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], double
ct_liqele2[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                         int counter, int max_ij, int max_k){
// This function checks that if layer 0 and 1 are completely solidified then the phase field old and new,
// concentration old and new are shifted down  so that the columnar cells have a chance to grow withouth
// disapearing due to the surroundings.

    // Declaration of Variables
    int shift_boolean_gzone; // 1 = yes shift down 0 = no
    int shift_boolean_gzone_new;
    int i, j, k;

    // Initialization of Variables

    shift_boolean_gzone = 1;
    shift_boolean_gzone_new = 1;

if(counter > 2){
    for(i=0; i<=max_ij-1; i++)
        for(j=0; j<=max_ij-1; j++)
            for(k=0; k<=1; k++){

            shift_boolean_gzone = shift_boolean_gzone*gzone[i][j][k];
            // If all the cells in the layer are solidified
            // then the multiplication of the functions should
            // equal to 1.0 otherwise they equal to some other
```

```
                // value.
                shift_boolean_gzone_new = shift_boolean_gzone_new*gzone[i][j][k];
        }// end for

                if((shift_boolean_gzone == 1.0)&&(shift_boolean_gzone_new == 1.0)){
                    // Then we shift down everything
                    for(i=0; i<=max_ij; i++)
                        for(j=0; j<=max_ij; j++)
                            for(k=1; k<=max_k-1; k++){
                            gzone[i][j][k-1] = gzone[i][j][k];
                            gzone_new[i][j][k-1] = gzone_new[i][j][k];
                            ct_liqele1_new[i][j][k-1] = ct_liqele1_new[i][j][k];
                            ct_liqele1[i][j][k-1] = ct_liqele1[i][j][k];
                            ct_liqele2[i][j][k-1] = ct_liqele2[i][j][k];
                            ct_liqele2_new[i][j][k-1] = ct_liqele2[i][j][k];
                    }// End for
                }
                else{
                    // We don't shift down anaything
                }
} // End if counter = 2


} // End shift down function
//****************************** END OF SHIFT DOWN FUNCTION ************************************************//

//************************************** NOISE FUNCTION ************************************************//
void noise_function(double gzone[MAX_SIZE][MAX_SIZE][MAX_SIZE], int max_ij, int max_k){

    // Declaration of variables
    int i, j, k;
    float gzone_rand;



  /* Seed the random-number generator with current time so that
   * the numbers will be different every time we run.
```

```
*/
        srand( (unsigned)time( NULL ) );

              for(i=0; i<=max_ij; i++)
               for(j=0; j<=max_ij; j++)
                for(k=1; k<=max_k; k++){
                    if((gzone[i][j][k] > 0)&&(gzone[i][j][k] < 1)){

                                        gzone_rand = (float) rand(); // Random part of the gzone (NOISE)
                                        gzone[i][j][k]= gzone[i][j][k] +(0.5-(gzone_rand/32767))*(1-gzone[i][j][k])/1E+4;
                                        // Dividing it by two times 32767 normalizes it to +or-  0.5 and then
                                        // multiply by the phase fraction makes sure that is a percentage of the gzone.
                                        // for then divide it by 1E6

                                        if(gzone[i][j][k] >1.0){
                                             cout << "stop";
                                        }// end if
                    } // End if
               } // End of for loop
}// End of noise function

//************************************************ END OF NOISE FUNCTION ************************************************//

//************************************** CONSERVATION OF MASS FUNCTION ************************************************//
void conserv_mass_function(double gzone_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],double
ct_liqele1_new[MAX_SIZE][MAX_SIZE][MAX_SIZE],
                                                double ct_liqele2_new[MAX_SIZE][MAX_SIZE][MAX_SIZE], alloy alloys,
                                                int max_ij, int max_k, int counter){


    // Declaration of variables
    int i, j, k, stop;
            float totalmass1, totalmass2;

    // Initialization of variable;
                        totalmass1 = 0; // Silicon
                        totalmass2 = 0; // Copper
```

```
        for(i=0; i<=max_ij; i++)
  for(j=0; j<=max_ij; j++)
     for(k=1; k<=max_k; k++){
                                totalmass1 = totalmass1+  gzone_new[i][j][k]*alloys.kdistr*ct_liqele1_new[i][j][k] +
                                                          (1-gzone_new[i][j][k])*ct_liqele1_new[i][j][k];

                                totalmass2 = totalmass2 + gzone_new[i][j][k]*alloys.kdistr*ct_liqele2_new[i][j][k] +
                                                          (1-gzone_new[i][j][k])*ct_liqele2_new[i][j][k];

                          }// End for

// OPENING FOR OUTPUT OF PROPERTY FILES
// Opening of file name
ofstream mass;
mass.open("mass.txt", ios::app);

if(!"mass.txt"){
     cout<< "Can not open file \n";
     // Handle error
     cin >> stop;
     exit (0);
}

        mass << "Phase Evolution Conservation of Mass test Iteration No. "<< counter << "\n";
        mass << " Total mass of element 1 (Silicon): " << totalmass1 << "\n";
        mass << " Total mass of element 2 (Copper) : " << totalmass2 << "\n";
        mass << "\n";


}// End of conservation of mass function


//****************************** CONSERVATION OF MASS FUNCTION ************************************************//
```