FLUID FLOW EFFECTS AND MICROSTRUCTURE FORMATION

DEVELOPING A COUPLED MICROSTRUCTURE FLUID FLOW MODEL FOR SOLIDIFICATION IN ADDITIVE MANUFACTURING

BY MAHDI PASHAEI, BC

A Thesis Submitted to the School of Graduate Studies in Partial
Fulfillment of the Requirements for
the Degree Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCE (2023)

Hamilton, Ontario, Canada (Material Science and Engineering)

TITLE: Microstructural Evolution in Additive Manufacturing: Coupling Fluid Flow with Solidification Modelling

AUTHOR: Mahdi Pashaei

BC (Material Science and Engineering),

Sharif University of Technology, Tehran, Iran

SUPERVISOR: Nana Ofori-Opoku

NUMBER OF PAGES: xvii, 73

Lay Abstract

This research aims to create a unified model that combines fluid flow and solidification processes to better simulate metal additive manufacturing, where molten metal flows and solidifies layer by layer to build parts. Our work provides a roadmap along with developed models for how to correctly combine these two physics, identifying the solidification and fluid flow models (whether turbulent or laminar) that best capture the dynamics of the process. To make the complex simulations feasible, we introduce simplifying assumptions and design examples to test and refine the model in stages. Through a structured approach, we develop and troubleshoot this combined model, ultimately creating a reliable tool to predict the quality and structure of manufactured metal parts.

Abstract

This research focuses on understanding additive manufacturing (AM) microstructure to enhance properties. Focusing on microstructure evolution during solidification, we aim to provide deeper insights into material formation by exploring the often-neglected or partially integrated aspect of fluid flow within the melt pool. This fluid flow is crucial as it influences concentration and temperature distribution, impacting microstructure development and therefore material behaviour. To bridge this gap, our project is developing a phase-field modelling microstructural model for solidification, coupled with the Navier-Stokes equation for fluid dynamics. Using the Finite Element Method (FEM) based library, FEniCS, we present our current development of the multiphysics approach needed. In this talk, we describe the model contributions of each physics contribution and our benchmark results against known literature. We discuss the need for the direct coupling of these into a singular, fully coupled solver, enhancing our understanding and control of AM processes.

Acknowledgments

I would like to sincerely thank my supervisor, Dr. Nana Ofori-Opoku, for his guidance, support, and encouragement throughout my research. His knowledge and helpful feedback have been essential in shaping this work and helping me grow both academically and professionally.

I am also deeply grateful to my family and friends for their constant support, patience, and encouragement. Their belief in me has given me strength and motivation throughout this journey.

Mahdi

February 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Additive Manufacturing

Metal additive manufacturing (AM), often referred to as metal 3D printing, is a groundbreaking innovation in manufacturing. Unlike conventional subtractive methods that remove material from a solid block, AM constructs parts layer by layer from digital models. This approach enables unprecedented design flexibility, making it possible to produce intricate geometries that would be challenging or unfeasible with traditional techniques [1, 2].

The origins of additive manufacturing date back to the 1980s, primarily focused on polymer-based systems [3, 4]. However, the development of metal AM technologies lagged behind due to the complexities associated with melting and fusing metals. It wasn't until the early 2000s that significant advancements were made, enabling the transition from prototyping to producing fully functional metal components. Today, metal additive manufacturing is an integral part of industries such as aerospace, automotive, healthcare and energy, driving innovation and efficiency [5] .

1.2 Features of Metal Additive Manufacturing

Additive manufacturing boasts several distinctive features that set it apart from traditional manufacturing methods. These are:

1. Layer-by-Layer Deposition – Metal AM builds parts by adding material layer-by-layer, using high-energy sources like lasers or electron beams to fuse the material. This approach allows for the precise construction of complex geometries with minimal waste [6].

2. Cyclic Phase Transformation – The process involves cyclic heating and cooling, causing phase transformations that affect the microstructure and mechanical properties of the material. Controlling these transformations is essential to achieve the desired part characteristics [7].

3. High-Temperature Gradients – Metal AM processes experience extreme temperature gradients, often as high as 10^5 – 10^6,°C/m [8, 9]. These gradients can lead to unique microstructures and residual stresses, which must be managed to ensure the quality of the part.

4. Rapid solidification – During metal AM, parts solidify rapidly, leading to the formation of fine microstructures that enhance mechanical properties, such as strength and toughness. However, the rapid cooling associated with this process can also introduce challenges, such as the formation of meta-stable microstructures and non-equilibrium phases, which can affect the uniformity of behaviour and materials across the entire part. Furthermore, rapid solidification can exacerbate issues such as porosity due to incomplete melting or gas entrapment during the construction process [10].

5. Reduction of Intermediate Steps and Tooling – Metal AM reduces or eliminates the need for intermediate steps and tooling, such as molds and dies. This

streamlines production, reduces costs, and allows greater flexibility and design customization [11].

The aforementioned features lead to interesting benefits and consequences for the build part, which are discussed next.

Benefits

Additive manufacturing offers several significant advantages over traditional manufacturing methods [12]:

- **Complex Geometries:** It allows for the creation of intricate and complex geometries that would be challenging or impossible to achieve using conventional manufacturing techniques.
- **Material Efficiency:** The layer-by-layer deposition process minimizes waste, making it a cost-effective solution.
- **Customization:** It enables the production of customized parts tailored to specific needs, such as patient-specific implants [13].
- **Reduced Lead Times:** Reduces the overall time from design to production.For example, in the production of quenching tools for the automotive industry, using AM reduced lead times from 220 hours (27.5 working days) to 116 hours (14.5 working days), nearly halving the time required [14].
- **Tooling Reduction:** Eliminates or significantly reduces the need for intermediate tooling and molds.

Current Limitations

Despite its many advantages, metal additive manufacturing also faces several challenges:

- **Porosity and Defects:** Parts can exhibit porosity and defects, which may compromise their mechanical properties [15].
- **Residual Stresses:** Rapid heating and cooling cycles can introduce residual stresses [16].
- **Surface Finish and Accuracy:** The surface finish of parts can be rough and may require post-processing.
- **Material Limitations:** The range of materials suitable for additive manufacturing remains limited, as each process is compatible with only specific material types, such as metals, polymers, ceramics, or resins. High costs, processing challenges (e.g., ceramics' high melting points), and application-specific requirements further restrict the options available for AM applications [17].
- **Cost and Scalability:** Although additive manufacturing minimizes material waste, the high costs of equipment and specialized materials, such as metal powders, present significant barriers [18].

The challenges associated with defects and residual stresses can be illuminated using high-resolution imaging studies and other detailed examinations of AM parts.

1.3 Physics of Additive Manufacturing

The benefits and challenges discussed previously arise as a direct consequence of the underlying physical phenomena governing the AM process. This section explores these phenomena in greater detail, referencing Fig. 1.1, which schematically illustrates the various physical processes involved in laser powder bed fusion (LPBF) manufacturing.

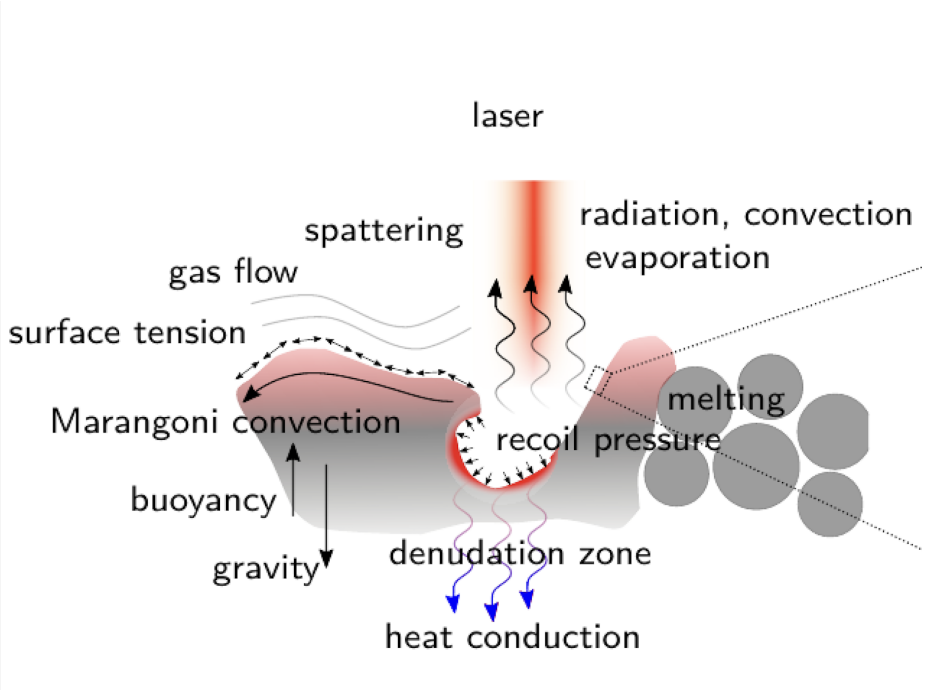


Figure 1.1: Schematic illustration of the physical phenomena occurring during LPBF AM. The image is reproduced from the MeltPoolDG repository [19].

Indeed, AM involves several complex physical phenomena that interact during the fabrication process. Understanding these interactions is crucial for optimizing the process and achieving high-quality output. These phenomena can be categorized into four main thematic areas: (i) laser-material interaction, (ii) thermal dynamics, (iii) fluid dynamics, and (iv) solidification dynamics.

1.3.1 Laser-Material Interaction

Laser-material interaction governs the transfer of energy from the laser to the material, driving processes such as melting, evaporation, and solidification. These interactions determine the quality of the melt pool, the microstructure of the resulting part, and potential defects.

Laser Irradiation: The process starts with a high-energy laser beam focused on the metal powder bed. The laser's energy is absorbed by the powder particles, leading

to rapid heating and melting. [20].

Radiation, Convection, and Evaporation: Upon absorbing laser energy, the powder heats up. Heat dissipates through radiation and convection, while some material evaporates, leading to material loss and impacting the quality of the part [20].

Recoil Pressure: Evaporation of material generates recoil pressure within the melt pool, affecting its shape and stability. This can cause spattering, where droplets of molten material are ejected from the melt pool [20].

1.3.2 Thermal Dynamics

Heat Conduction: Heat is transferred from the melt pool to the surrounding powder bed and the previously solidified layers. Proper heat conduction is essential for regulating the cooling rates and ensuring the solidification process is controlled.

Denudation Zone: This region experiences displacement of powder particles due to interaction with the laser beam and the vapor plume from evaporation, affecting the uniformity and consistency of the powder layer [10].

1.3.3 Fluid Dynamics

Marangoni Convection: Surface tension gradients within the melt pool, driven by temperature differences, induce Marangoni convection. This fluid flow within the molten metal helps distribute heat and material, influencing the microstructure and properties of the final part (Section 2.3.2, Surface Tension, Ref. [20]).

Buoyancy and Gravity: Buoyant forces due to density differences in the melt pool and gravity influence the flow, which influences the solidification kinetics of the molten metal.

1.3.4 Material Dynamics

Melting and Solidification: The laser energy melts the metal powder, which then solidifies as the laser moves away. Rapid cooling rates lead to fine microstructures but can also introduce residual stresses and porosity [21]. After the deposition of additional layers, the part undergoes a remelting and subsequent solidification process as the cycle continues.

Spattering: Small droplets of molten material can be ejected from the melt pool due to recoil pressure and other dynamic forces, potentially leading to defects in the final part [22].

Gas Flow: Gas flow, whether part of the process environment or arising from vaporized material, influences thermal dynamics through heat transfer and material dynamics by driving material redistribution, surface morphology changes, and chemical reactions within the process.

A fundamental understanding of the interplay of these physical phenomena (see Fig. 1.2) is essential for optimizing AM processes. Fine control over laser parameters, heat management, and material handling can significantly influence the quality, mechanical properties, and reliability of the manufactured parts [23]. Experimental studies and *in situ* part fabrication provide valuable insights into these phenomena; however, disentangling their complex behaviors and isolating the parameters that govern them remains a significant challenge [24]. To address this, theoretical and numerical modeling serve as indispensable tools for uncovering the fundamental mechanisms underlying these behaviors [25]. The following sections explore these approaches in detail.

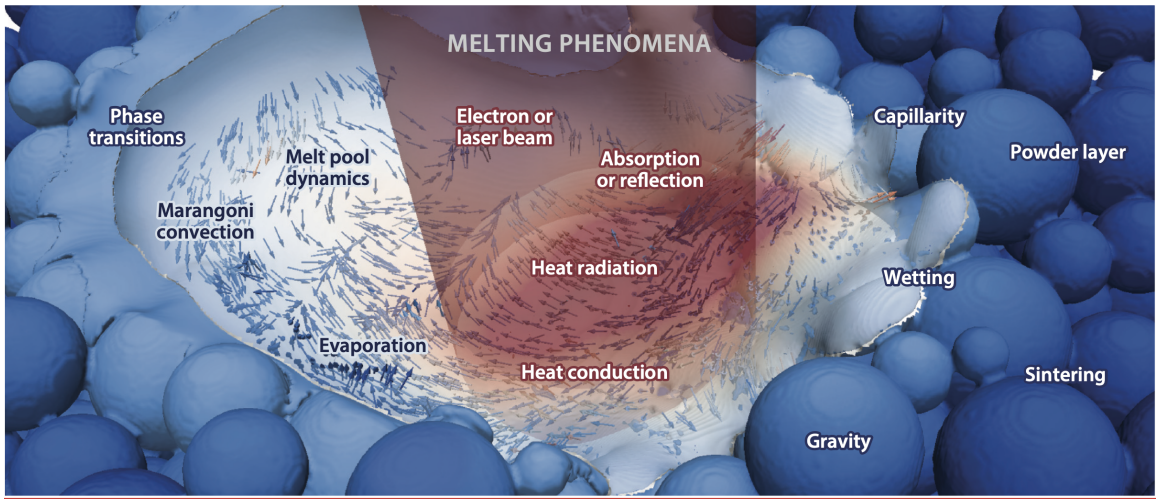


Figure 1.2: Melt pool dynamics and associated physical phenomena, as illustrated in the image, adapted from the article *Multiscale Modeling of Powder Bed–Based Additive Manufacturing* by Markl and Körner [26].

1.4 Modeling Trends in Additive Manufacturing

Additive manufacturing has seen significant advancements in modeling techniques, which are essential for understanding and optimizing the complex physical phenomena involved. These modeling trends help in predicting process outcomes, improving part quality, and reducing trial-and-error experimentation. The models can be classified as follows: (i) multiphysics modeling [20, 27], (ii) data-driven and machine learning modeling [28, 29], and (iii) computational fluid dynamics (CFD) modeling [30, 31].

1.4.1 Multiphysics Modeling

Multiphysics modeling combines thermal, fluid, and mechanical phenomena to simulate key aspects of the AM process, such as melt pool dynamics, solidification, and residual stresses. Advanced techniques like phase-field models are used to predict microstructural evolution and material properties [25].

Coupled Thermal-Fluid-Mechanical Models: These models integrate thermal,

fluid, and mechanical phenomena to simulate the entire AM process. By coupling heat transfer, fluid flow, and mechanical stress analysis, these models provide a comprehensive understanding of melt pool dynamics, solidification, and residual stress states [20].

Phase-Field Models: These models simulate the microstructure evolution during solidification. Phase-field models are especially valuable for predicting grain structure, orientation, and segregation patterns, which are crucial for assessing the mechanical properties of the final part [25].

1.4.2 Data-Driven and Machine Learning Models

Machine learning techniques analyze large datasets to optimize process parameters, predict defects, and improve part quality. Integrated with real-time monitoring systems, these models enable adaptive control of the AM process for consistent and efficient production [32, 28].

Machine Learning for Process Optimization: Machine learning algorithms are increasingly used to analyze large datasets generated during AM processes. These models can predict optimal process parameters, identify defects, and improve part quality by "learning" from historical data [32].

In-Situ Monitoring and Real-Time Control: Advanced sensors and data acquisition systems are integrated with machine learning models to enable real-time monitoring and control of the AM process. These systems can adjust process parameters on-the-fly to mitigate defects and ensure consistent quality [28].

1.4.3 Computational Fluid Dynamics (CFD) Models

Computational fluid dynamics models play a central role in simulating melt pool dynamics, gas flow, and powder behavior in additive manufacturing. These simulations provide insights into key phenomena such as convection, Marangoni flow, and keyhole-

induced porosity, helping to optimize process parameters and reduce defects [30, 31]. By leveraging advanced techniques, CFD enhances the understanding of complex interactions in laser powder bed fusion processes, ensuring better structural integrity and material performance.

High-Fidelity CFD: CFD models are used to simulate the flow of molten metal within the melt pool. These simulations help in understanding the effects of process parameters on melt pool behavior, such as convection currents, Marangoni flow, and solidification patterns [30].

Gas Flow and Powder Dynamics: The role of CFD extends to simulating the behavior of gas flow and powder dynamics within the powder bed fusion environment. This aids in our understand of how gas dynamics interact with molten metal, particularly in the formation and evolution of keyhole defects. As you can see in Fig. 1.3 simulation take into account the trapping of gases within keyhole cavities which is a pivotal factor in the development of keyhole-induced porosity. This aids in achieving uniform powder distribution and reducing defects such as porosity and lack of fusion, ensuring the structural integrity and performance of the final product. The incorporation of advanced techniques such as ray-tracing and Fresnel absorption within these models provides a deeper insight into the complex interplay of laser energy absorption, melt pool fluid flow, and the thermal dynamics that lead to porosity formation, particularly under high-energy input conditions typical of LPBF processes [31].

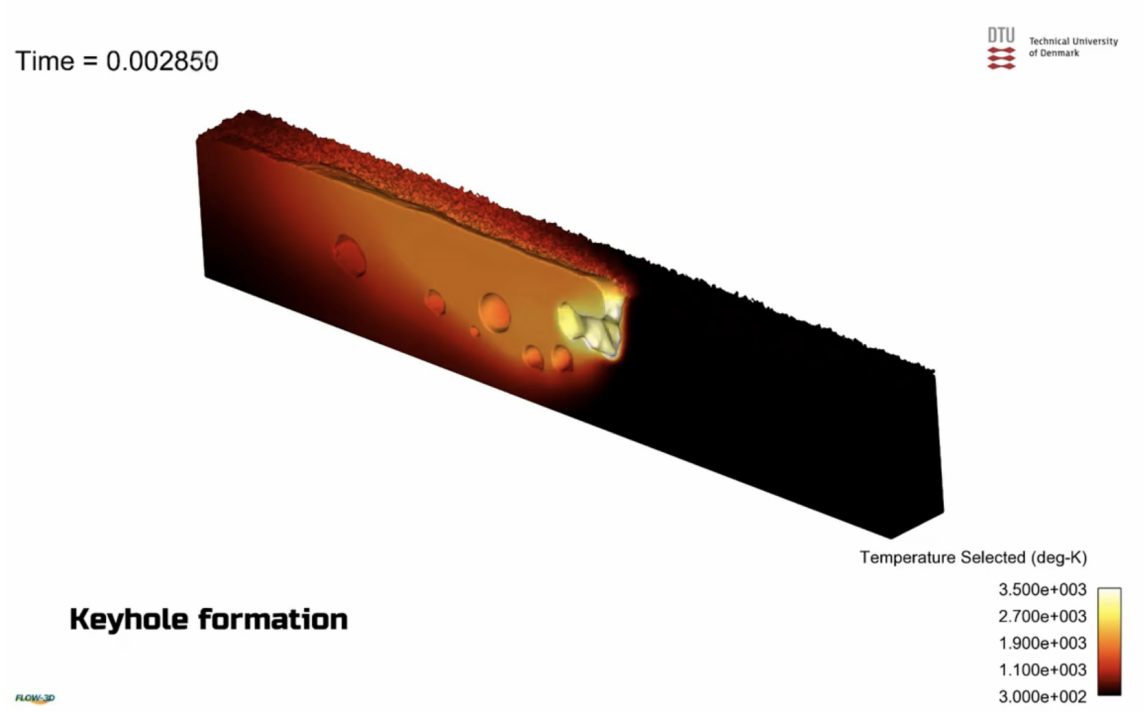


Figure 1.3: Simulation of melt pool dynamics and keyhole formation in LPBF processes modeled in FLOW-3D AM from Refs. [33, 31].

The current state-of-the-art of modelling in AM focuses on integrating multiphysics simulations [34], leveraging data-driven approaches, and employing advanced computational techniques. These models enhance our understanding of the AM process, improve part quality, and accelerates the development of new materials and designs. As these modelling techniques continue to evolve, they will play an increasingly essential role in advancing AM technologies.

1.5 Role of Fluid Flow in Additive Manufacturing

Among the physical phenomena previously discussed, two are fundamentally critical to part development during additive manufacturing (AM); fluid flow and solidification. Specifically, the coupling of fluid flow and solidification processes is essential for gaining a basic and fundamental understanding of AM [35, 36, 37, 38]. This section first examines the impacts of fluid flow on solidification, followed by an exploration of the

various approaches that have been employed to couple these mechanisms.

1.5.1 Impact of Fluid Flow in AM

Convection-driven fluid flow influences concentration, alters temperature and undercooling in the melt pool, and affects the resulting microstructure. Refer to Fig. 1.4, which compares dendrite growth patterns with and without the influence of a fluid flow field.

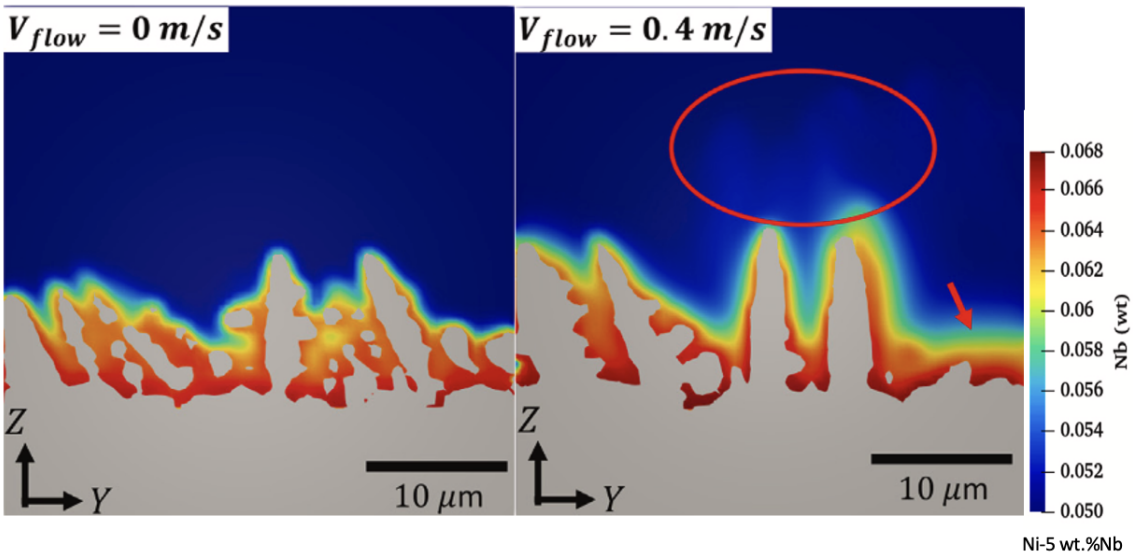


Figure 1.4: Comparison of Nb distribution in the middle of the simulation domain under a temperature gradient of 5×10^5 K/m and a solidification velocity of 0.02 m/s. Comparing the left-hand side (LHS) and right-hand side (RHS) images reveals the impact of fluid flow—present in the RHS but absent in the LHS—on the concentration ahead of the dendrite growth direction in the melt pool, which affects the growth pattern and undercooling . The regions in gray represent dendrites, highlighting the influence of fluid dynamics on dendrite formation and microsegregation in additive manufacturing from Ref. [36].

The work by Yu et al. [36] utilizes a two-way coupling approach (coded in C++) between the dendrite growth model and the computational fluid dynamics (CFD) model, allowing for dynamic interaction and mutual influence between the dendrite growth and fluid flow within the simulation environment. While this method effectively captures the complex interactions between fluid flow and dendritic structures, it

employs a cellular automaton algorithm, which lacks the precision and detailed microstructural evolution captured by phase field models [39]. Consequently, while the cellular automaton provides a computationally efficient approach, it may not resolve the finer details of phase transformations as accurately as phase field models.

Additionally, Yu and co-workers also report several key findings regarding the impact of fluid flow on dendrite growth and grain formation in additive manufacturing:

1. **Dendrite Morphology:** The study illustrates that fluid flow significantly influences the morphology of dendrites by altering solute distributions and concentrations in the melt pool, which in turn affects dendritic structures and growth kinetics.

2. **Microstructure Development:** Variations in fluid flow lead to changes in temperature gradients and solidification velocities, impacting the microstructural development and resulting in varied microsegregation patterns and dendritic arm spacings.

3. **New Grain Formation:** Fluid dynamics, especially varying flow velocities and temperature conditions, enhance the nucleation of new grains at the dendritic growth front, potentially leading to a more equiaxed grain structure.

4. **Modeling Approach:** The article utilizes a two-way coupled computational model between dendrite growth and fluid dynamics, providing a realistic simulation of the interactions that affect solidification processes in additive manufacturing.

5. **Model Comparison:** Although the cellular automaton algorithm used offers computational efficiency, it lacks the precision of phase field models, which are better suited for detailed phase transformations and microstructural evolution.

In summary, Yu et al. [36] found that fluid flow redistributes solutes like Nb,

enhancing undercooling near dendrites aligned with the temperature gradient (A-TGD), accelerating their growth, while reducing undercooling for misaligned dendrites (D-TGD), slowing their growth. This leads to uneven dendrite tip heights, increased primary dendrite arm spacing (PDAS), and reduced dendrite density. Moreover, fluid-driven solute accumulation promotes new grain nucleation, disrupting continuous dendrite growth.

Tang and Du [35] utilized OpenFOAM [40] to simulate the 3D melt pool dynamics and temperature evolution during laser AM of Ni-based alloy. They linearly interpolated the data from the flow simulation to a solidification model using a 2D phase-field method. After extracting the temperature from the fluid flow model, they set the phase-field variable $\phi = -1$ (indicating liquid) based on the liquidus point temperature and updated the dimensionless supersaturation variable U accordingly. **However, the phase-field model did not incorporate or update the velocity field, focusing solely on microstructural evolution driven by temperature and solute concentration.** They general results are;

1. **Model Validation**: The model accurately predicted the melt pool profile, dendrite size, morphology, and texture, showing good agreement with the experimental data.

2. **Solidification Process**: Initial solidification features planar growth transitioning to cellular and dendritic patterns. Competitive grain growth favours well-oriented grains over misoriented ones during both the planar-to-cellular transition and the later stages of solidification as evident in Fig. 1.5.

3. **Solute Trapping**: Significant solute trapping occurred at high solidification velocities, indicating deviations from local equilibrium.

4. **Multi-Physics Framework**: The integrated CFD and PF models effectively simulated dendritic solidification, providing insights into cellular microstructure

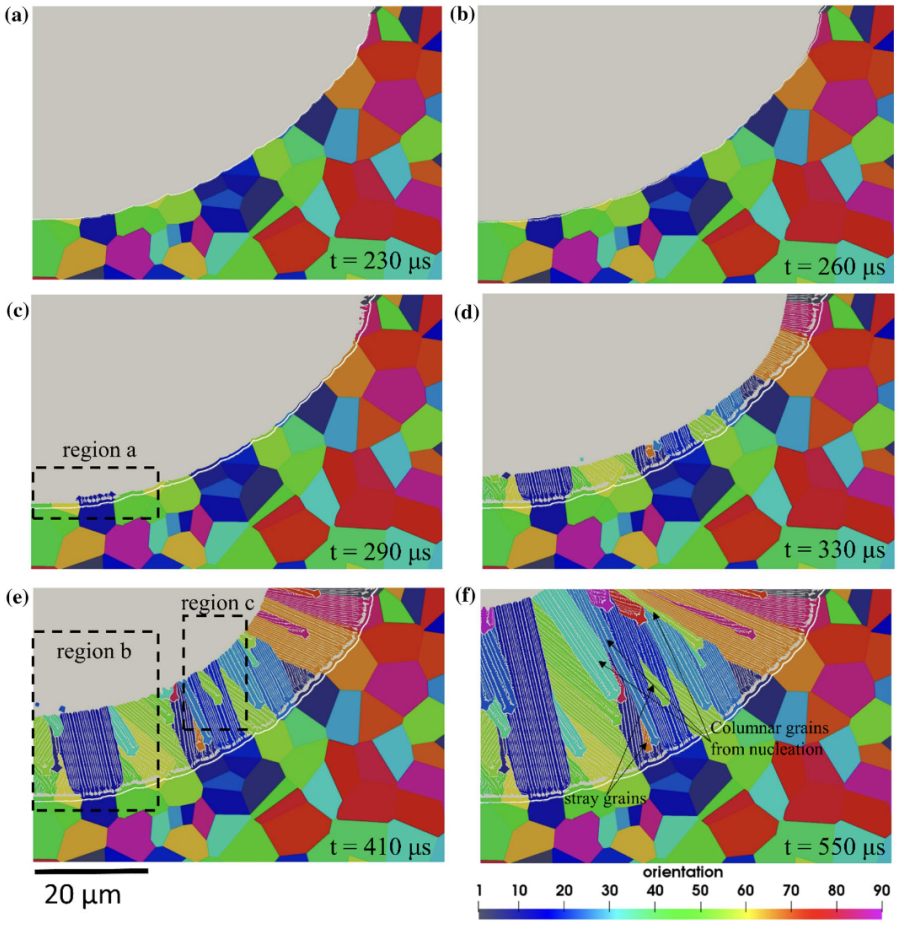


Figure 1.5: Temporal evolution of dendritic microstructure during solidification at different time steps: (a) 230 µs, (b) 260 µs, (c) 290 µs, (d) 330 µs, (e) 410 µs, and (f) 550 µs. The figure illustrates the progression from planar growth to cellular and dendritic structures, highlighting the planar-to-cellular transition and the formation of secondary side branches [35].

formation, competitive grain growth, and solute segregation.

Figure 1.5 depicts the temporal evolution of the dendritic microstructure during solidification at various time steps. The solidification process starts with planar growth at 230 µs, transitions to cellular growth by 290 µs, and develops into well-defined dendritic structures by 550 µs. The transition from planar to cellular growth and the formation of secondary side branches are clearly visible, illustrating the competitive growth dynamics between well-oriented and misoriented grains [35].

1.5.2 Types of Coupling for Fluid Flow

Coupling in fluid flow allows the interaction between physical phenomena. The variations and mthods of coupling include:

- **Thermal Coupling:** Interaction between temperature gradients and fluid flow, affecting heat transfer and solidification rates [37, 38].
- **Solutal Coupling:** Solute concentration gradients drive fluid flow, influencing microstructure and segregation [36].
- **Two-Way Coupling:** Feedback between fluid flow and solid growth, where flow alters dendrite morphology, and dendrites affect flow [35] as illustrated in the schematic shown in Figure 1.6.
- **Convective Coupling:** Heat and solute transport via convection, critical for uniformity and defect prevention [37].
- **Diffusive-Convective Coupling:** Combined diffusion and convection, shaping thermal and solutal fields in AM [38].

These couplings provide critical insights into microstructure.

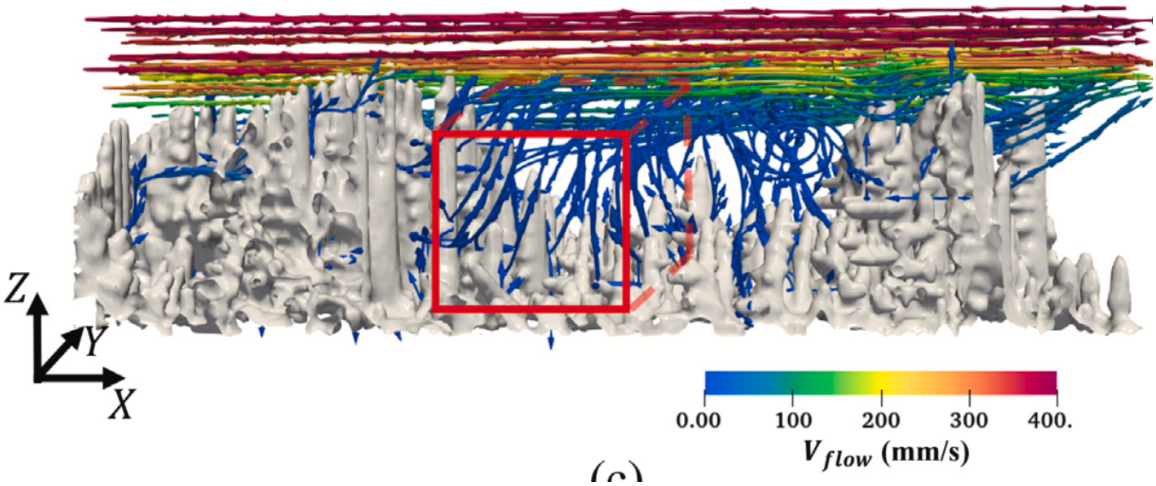


Figure 1.6: Streamlines of fluid flow demonstrating two-way coupling [36].

1.5.3 Coupled Fluid Flow and Solidification

The development of coupled fluid flow and solidification models in AM typically begins with a fundamental understanding of the individual physical phenomena. Initial models often use simplified assumptions and gradually incorporate more complexity as the understanding of the interactions between fluid flow, heat transfer, and solidification improves.

To begin the development, one should start by implementing a microstructural model. To date, the most comprehensive is the phase field dendritic models like those in [41] and [42]. Implementing these models ensures the consistency of the fluid flow and microstructure model. The computational resources and simulation domain required for these examples are significantly less demanding than those needed for the full AM domain, enabling quicker and more efficient testing.

After verifying these initial models, one can proceed to benchmark the result formulation against those already found in literature, e.g. Ref [27]. Benchmarking these models against established results is a indeed an essential step. Following successful benchmarking, more complex models may be attempted such as those presented in [42] and [36].

It is important to note that the fluid flow models in the vast majority of existing literature assume laminar flow. However, in fully coupled models, where the solid geometry may be complex, the interaction resulting fluid flow can lead to **turbulence**. Therefore, it is advisable to test thoroughly under various flow conditions to establish the range of validity of one's coupled model formulation.

1.6 Thesis Outline

After discussing the challenges in additive manufacturing (AM) and the importance of effectively coupling fluid flow and solidification, the subsequent chapters are briefly and

generally outlined. These chapters aim to address the development of a comprehensive model for simulating solidification and fluid flow in AM. The chapters are organized to provide a structured, step-by-step approach, with each phase of development focusing on a key aspect of the physical processes involved. The steps and their corresponding chapters in the thesis are highlighted below.

Solidification Microstructure Model Development: In the standard approach for this field, the development of a Phase-Field (PF) model will begin with simulations of solidification in binary alloys, as illustrated in Fig. 1.7. This model incorporates both thermal and solutal effects, allowing for detailed simulations of microstructural evolution during solidification. The accuracy of the model will be validated by benchmarking against established results and other theoretical formulations. This is discussed in Chapters 2.

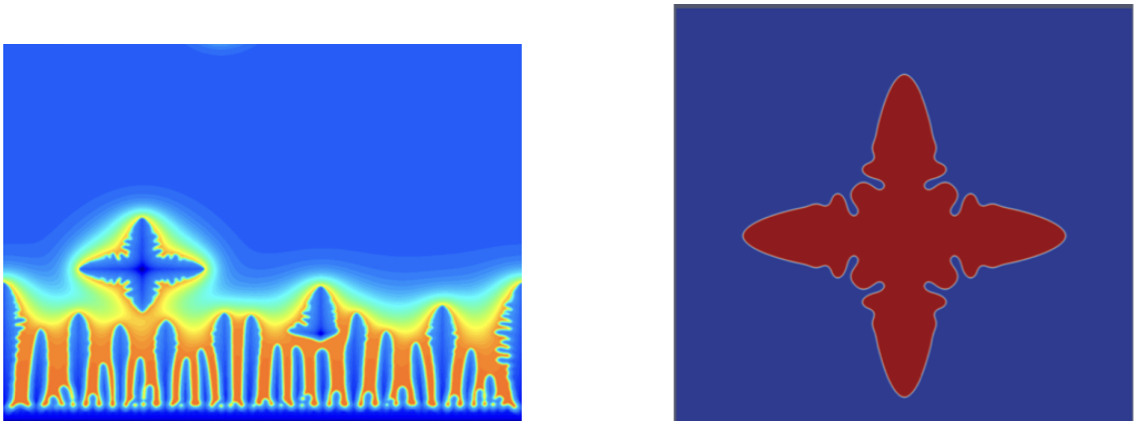


Figure 1.7: The image on the left, taken from [43], depicts the solidification process of an additively manufactured IN718 alloy within the melt pool. It specifically shows nucleation and dendrite growth during the early stages of solidification, where nuclei form in the liquid phase and grow into columnar and equiaxed dendrites. The field on the left represents the solute concentration of Nb, highlighting variations in solute distribution, while the field on the right represents the order parameter, capturing the transition from liquid to solid during the solidification process. The right image is an equiaxed from a simulation done in testing phase-field for this work.

Fluid Flow Model Development: After the successful development of the solidification model, attention will shift to the development of a fluid flow model based on the

Navier-Stokes equations. Key phenomena, such as Marangoni convection, buoyancy, and flow within the melt pool, will be integrated to accurately simulate the dynamics of fluid flow in additive manufacturing (AM) processes, as shown in Fig. 1.8. These phenomena are crucial for controlling temperature and concentration gradients during the build process. This topic is explored in detail in Chapter 3.

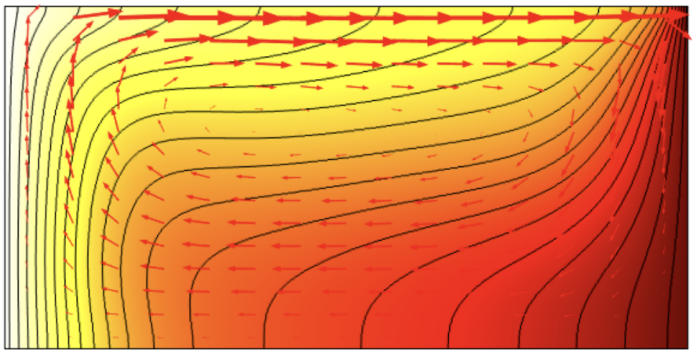


Figure 1.8: Fluid flow simulation taken from Ref. [44]. The image shows the effect of a temperature gradient between the left and right walls, which induces a Marangoni force on the top boundary due to the temperature-dependent surface traction coefficient. Both isotherm lines and velocity vectors, visualized using COMSOL, demonstrate the resulting fluid flow dynamics.

Optimizing Formulations: After developing the model, further optimization was required. A coordinate-based adaptive meshing scheme was implemented in FEniCS to reduce computational costs (see Figure 1.9).

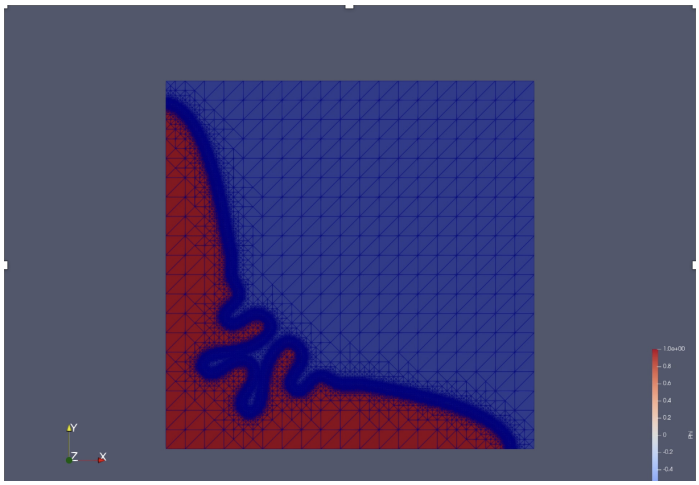


Figure 1.9: Adaptive meshing scheme. The adaptive meshing scheme depicted here refines the mesh specifically around the interface between the liquid and solid phases during solidification. The interface is identified by computing the gradient of the order parameter, and a defined threshold is used to selectively refine mesh elements in regions where the gradient surpasses this value. This approach enhances resolution in critical regions near the interface, allowing for accurate capture of interface dynamics, while maintaining a coarser mesh in less critical areas of the simulation domain to optimize computational efficiency and reduce unnecessary computational expense

Coupling PF and Fluid Flow Models: The final stage involved coupling the solidification and fluid flow models into a unified framework. This topic is discussed in Chapter 4. The fully coupled model enables the simultaneous simulation of phase changes and fluid dynamics during AM. Figure 1.10 illustrates an overview of this section. By capturing the interactions between fluid flow and dendritic growth, the model provides a more accurate prediction of microstructure evolution and material properties in the completed part.

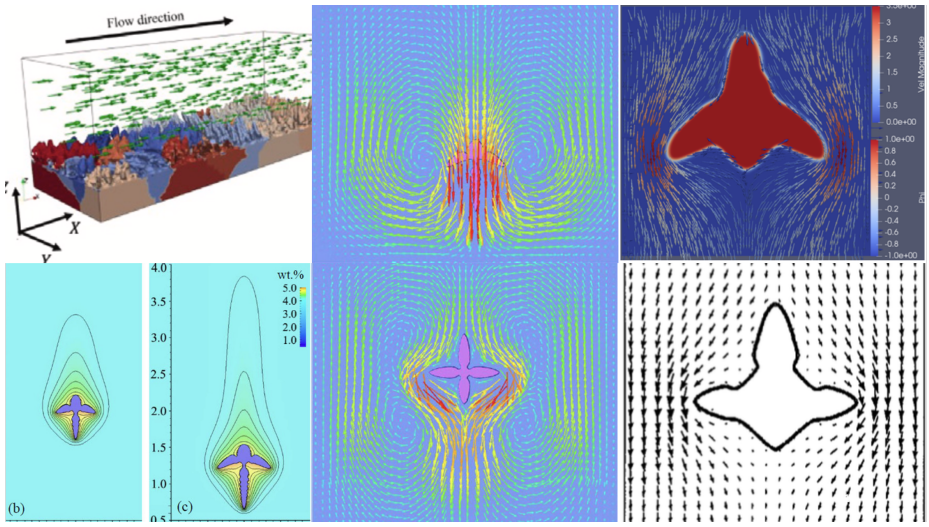


Figure 1.10: Coupled fluid flow and solidification models. The top-left image is taken from [36], while the top-middle, bottom-left, and bottom-middle images are taken from [27]. The bottom-right image is from [45]. The top-right image shows the simulation performed in Chapter 4 of this study, Implemented to benchmark the coupled model of solidification and fluid flow.

After establishing the coupled model formulations, the focus shifts to the application of the model. Before proceeding, the developed PF model is used to provide a reference microstructure without fluid, as detailed in Chapter 5, following established work in the literature that defines a set of temperature gradient and velocity parameters for AM. Subsequently, in Chapter 6, the model is applied to obtain preliminary results on the effects of fluid flow on length scale determination in AM.

Finally, Chapter 7 offers a summary, including concluding remarks, a discussion of the remaining challenges, and an outlook on the future directions of the work presented in this thesis.

Chapter 2

Phase Field Theory

The Phase Field Method (PFM) has become a powerful and versatile computational technique for modelling and simulating various phase transitions and microstructural evolutions in materials. Unlike traditional sharp interface models [46], the phase field method utilizes a diffuse interface approach. In Fig. 2.2, the order parameter ϕ shows a smooth and continuous transition from the solid to the liquid region, effectively capturing the dynamics of the interface. This chapter explores the principles and applications of the phase field method.

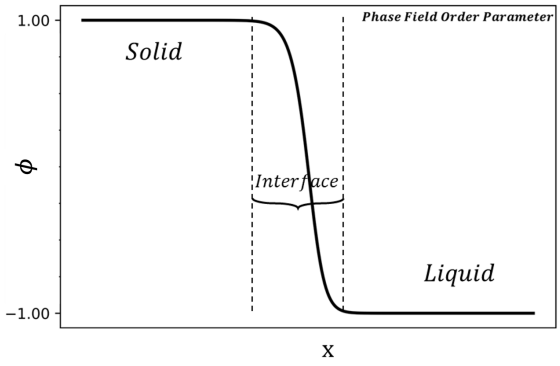


Figure 2.1: Order parameter provides a continuous representation of interfaces with greater computational simplicity and flexibility [47].

2.1 Principles of the Phase Field Method

The methodology typically begins with the phenomenological construction of a free energy functional, $\mathcal{F}$ [47, 48]. In the case of pure material solidification, this functional is expressed in terms of the temperature and polynomial terms of the order parameter ϕ employed to model the phase transition. The scalar order parameter (or phase-field) represents the two equilibrium phases, with$\phi = 1$ in the solid phase and $\phi = -1$ in the liquid phase. The field of the order parameter smoothly transitions between these two equilibrium values across a diffuse interface of thickness W, where $-1 < \phi < 1$. The evolution of this order parameter is governed by a set of partial differential equations derived from thermodynamic principles and kinetic considerations as consequence of the postulated function $\mathcal{F}$. The primary components of PFM can be reduced to; (i) order parameter field that represents the state of the system, with values indicating different phases or orientations, (ii) free energy functional ($\mathcal{F}$) which describes the thermodynamic potential of the system, incorporating bulk free energy and gradient energy contributions and (iii) evolution equations described by a set of PDEs that describe the temporal evolution of the order parameter field driven by the minimization of the free energy.

2.2 Overview of Phase-Field Modeling Principles

Building upon the Ginzburg-Landau [49] and Cahn-Hilliard [50] theories, the phase-field method employs an order parameter ϕ (Fig. 2.2) to smoothly represent phase transitions and microstructural evolution.The method begins with a free energy functional ($\mathcal{F}$), which governs system behavior. For a pure material, it is expressed as [51]:

$$\mathcal{F}[\phi, U] = \int d\mathbf{r} \left(\frac{1}{2} W(\mathbf{n}) |\nabla \phi|^2 + f(\phi) + \lambda U(T) g(\phi) \right), \tag{2.1}$$

For binary alloys, the functional includes a chemical free energy term, $f_c(\phi, T, c)$, that accounts for solute concentration, c:

$$\mathcal{F}[\phi, c] = \int d\mathbf{r} \left(\frac{1}{2} W(\mathbf{n}) |\nabla \phi|^2 + f(\phi) + f_c(\phi, T, c) \right). \tag{2.2}$$

The parameters are in the Tab. 2.1. Now, what are the evolution equations for the phase field and concentration? The phase field, ϕ, evolves according to:

$$\tau(\mathbf{n}) \frac{\partial \phi}{\partial t} = -\frac{\delta \mathcal{F}[\phi, c]}{\delta \phi}, \tag{2.3}$$

where $\tau(\mathbf{n})$ is a relaxation time that controls interface kinetics.

The concentration field, c, id the following dynamics:

$$\frac{\partial c}{\partial t} = \nabla \cdot \left(M(\phi, c) \nabla \frac{\delta \mathcal{F}[\phi, c]}{\delta c} \right), \tag{2.4}$$

where $M(\phi, c)$ sets the mobility of solute atoms, proportional to the diffusion coefficient. This equation can be rewritten in the form:

$$\frac{\partial c}{\partial t} + \nabla \cdot \mathbf{J} = 0, \tag{2.5}$$

with the flux, $\mathbf{J}$, expressed as:

$$\mathbf{J} = -M(\phi, c) \nabla \mu, \tag{2.6}$$

where $\mu \equiv \frac{\delta \mathcal{F}[\phi, c]}{\delta c}$ is the chemical potential.

Table 2.1: Summary of Parameters in the Phase-Field Model [51, 47].

Parameter	**Symbol**	**Description**
Interfacial energy	$W(\mathbf{n})$	Controls interfacial energy and introduces anisotropy via the interface normal $\mathbf{n}$. Anisotropy is modeled as: $$a(\mathbf{n}) = (1 - 3\epsilon_4)\left(1 + \frac{4\epsilon_4}{1 - 3\epsilon_4}(n_x + n_y + n_z)\right)$$
Bulk free energy	$f(\phi)$	Models the double-well potential, stabilizing $\phi = \pm 1$ for solid and liquid phases: $$f(\phi) = -\frac{\phi^2}{2} + \frac{\phi^4}{4}$$
Coupling constant	λ	Couples the capillary length d_0 and interface width W. It scales as: $$d_0 = a_1 \frac{W_0}{\lambda}$$ where $a_1 = 0.8839$ and $a_2 = 0.6267$ are constants based on asymptotic analysis.
Dimensionless temperature	$U(T)$	Normalized temperature, defined as: $$U(T) = \frac{T - T_m}{L/c_p}$$ where T is temperature, T_m is melting temperature, L is latent heat, and c_p is specific heat.
Interpolation function	$g(\phi)$	Smoothly interpolates phase transitions, satisfying $g(\pm 1) = 1$ and $g'(\pm 1) = 0$. A common form is: $$g(\phi) = \frac{15}{8}\left(\frac{\phi^5}{5} + 2\frac{\phi^3}{3} - \phi\right)$$
Chemical free energy	$f_c(\phi, T, c)$	Models solute concentration effects and corrects spurious kinetics. For dilute alloys: $$f_c(\phi, T, c) = f^A(T_m) - \Delta T s(\phi) + \frac{RT_m}{v_o}(c \ln c - c) + \epsilon(\phi)c$$ with $f^A(T_m)$ being the free energy of component A at melting, $s(\phi)$ is the interpolated entropy, R is the gas constant, v_o is molar volume (of the solvent), and $\epsilon(\phi)$ is interpolated internal energy. Corrections include anti-trapping fluxes and modified interpolation functions to eliminate spurious effects.

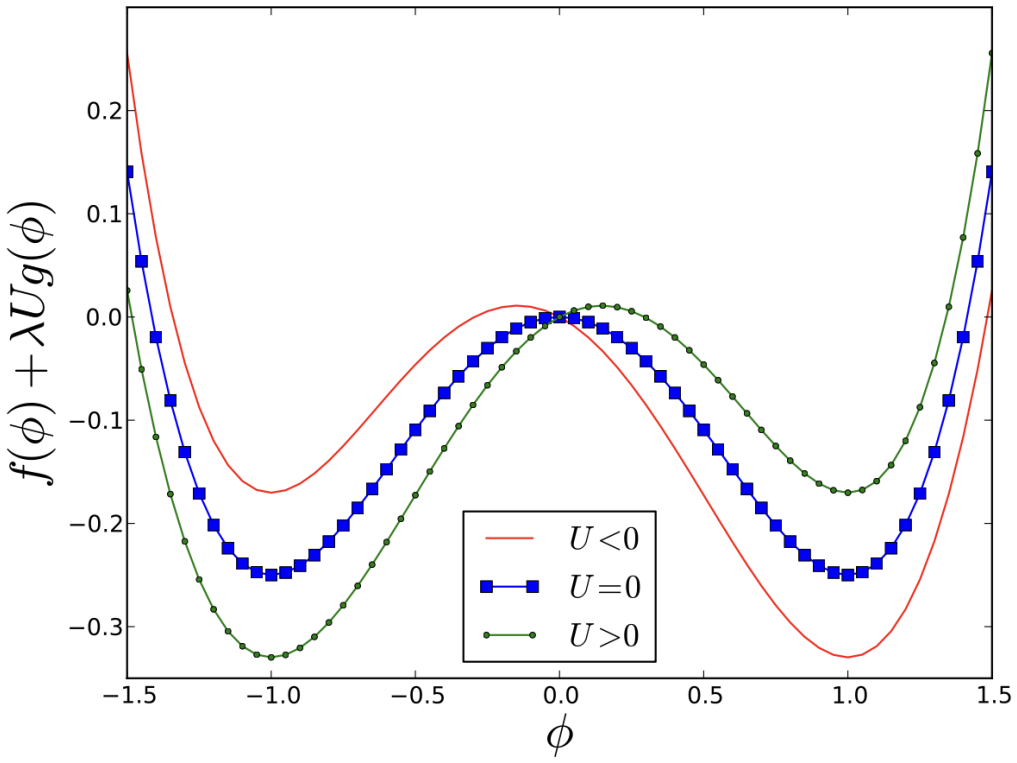


Figure 2.2: The double-well potential (Eq. 2.1) illustrating the energy density, $f(\phi) + \lambda U g(\phi)$, as a function of the order parameter (ϕ) for different temperatures (U). The curves show how the energy landscape evolves with temperature, with the liquid phase ($\phi = -1$) being favoured at high temperatures and the solid phase ($\phi = 1$) favoured at low temperatures [51].

The PFM provides a thermodynamically consistent framework for simulating microstructural evolution during phase transformations by defining phase transitions and interface dynamics through a free energy functional that incorporates interfacial energy, bulk energy, and coupling terms. Key parameters, such as the capillary length (d_0), are derived from microscopic properties using asymptotic analysis, ensuring quantitative agreement with sharp-interface models via thin-interface limits. Constants a_1 and a_2 are calibrated through asymptotic expansions to capture interface kinetics and curvature effects accurately. The resulting equations couple phase and concentration fields, enforcing mass conservation and energy minimization, while additional corrections—such as anti-trapping currents and anisotropic interface kinetics—enhance numerical stability and physical accuracy.

2.3 Solidification Modeling

Phase-field equations are particularly effective in describing the process of solidification, where a liquid transforms into a solid. In such scenarios, the phase field method captures the complex dynamics of interface movement, undercooling, and microstructure formation. The evolution of the solid-liquid interface during solidification is influenced by: (i) undercooling - the degree of undercooling (the temperature below the melting point) drives the nucleation and growth of solid phases. Higher undercooling rates can lead to finer microstructures. This is the driving force for the transformation and can also be described by the supersaturation. (ii) interface kinetics - The rate at which the solid-liquid interface moves is governed by kinetic coefficients and the local thermodynamic driving forces, and (iii) anisotropy - the direction dependence of surface energy and interface mobility can lead to the development of dendritic structures, which are commonly observed in solidified metals. In Fig. 2.3, a representative phase-field model simulation illustrates the solidification process of a pure liquid material, capturing the evolution of the solid-liquid interface and the formation of the microstructures

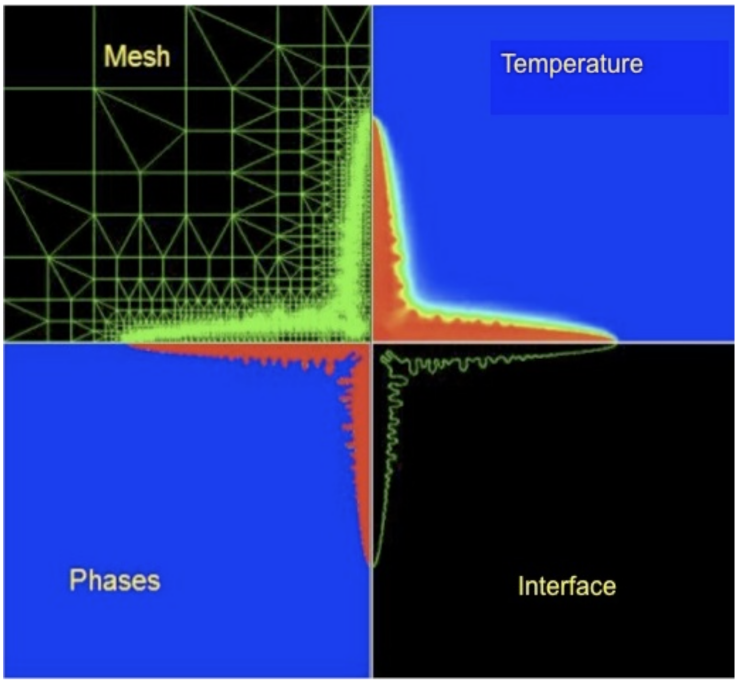


Figure 2.3: Snapshot capturing the evolution of a thermal dendrite for a pure material, showing a four-fold crystal growing into an undercooled liquid. The four frames, viewed clockwise from the top right, display: (1) temperature distribution (red indicates the highest temperature, blue the lowest), (2) interface position (where $\phi = 0$), (3) phase field (solid in red, liquid in blue), and (4) dynamically adapted mesh resolving the temperature and phase field [48].

There are numerous PF models and methodologies available; however, this work introduces and focuses on two well-known models. The first is for *isothermal alloy solidification*, while the second is for *thermosolutal alloy solidification*, which combines the physics of both mass and heat transfer.

Isothermal Alloy Solidification Model

The isothermal alloy solidification model is employed to describe the solidification process in alloys under fixed thermal conditions, where only mass transport is considered. In a special case of this formulation, adopted here, the model assumes a "frozen temperature" approximation, where the temperature gradient applied remains constant throughout the solidification process. The governing equations for this model,

derived for dilute alloys, are detailed in Ref. [52]. The equations read;

$$\tau_0 \left[1-(1-k)\frac{z-V_p t}{l_T}\right]\frac{\partial \phi}{\partial t} = W^2\nabla^2\phi + \phi - \phi^3 - \lambda g'(\phi)\left(U + \frac{z-V_p t}{l_T}\right) \tag{2.7}$$

$$\begin{aligned}\left(\frac{1+k}{2} - \frac{1-k}{2}h(\phi)\right)\frac{\partial U}{\partial t} \\ = \vec{\nabla}\cdot\Big(Dq(\phi)\vec{\nabla}U + a(\phi)W[1+(1-k)U] \\ \times\frac{\partial \phi}{\partial t}\frac{\vec{\nabla}\phi}{|\vec{\nabla}\phi|}\Big) + [1+(1-k)U]\frac{1}{2}\frac{\partial h(\phi)}{\partial t},\end{aligned} \tag{2.8}$$

where Eq. 2.7 is the kinetic equation for the order parameter, while Eq. 2.8 drive the mass transport process through the description of a dimensionless chemical potential, U.The variables and parameters used in these equations are detailed in Tab. 2.2, respectively. The variables used in this study are defined as follows. The dimensionless temperature is given by $\theta = \frac{T-T_m-mc_\infty}{L/c_p}$, while the dimensionless concentration is expressed as $U = \frac{\exp(u)-1}{1-k}$. The dimensionless chemical potential is represented by $u = \ln\left(\frac{2c}{c_\infty}\frac{1+k-(1-k)\phi}{1+k-(1-k)\phi}\right)$. The unit vector normal to the interface is defined as $n = \frac{\vec{\nabla}\phi}{|\vec{\nabla}\phi|}$. The scaled magnitude of the liquidus slope is described by $M = \frac{-m(1-k)}{L/c_p}$. The characteristic time scale is expressed as $\tau(n) = A(n)^2\left[\frac{1}{Le} + Mc_\infty[1+(1-k)U]\right]$, where the anisotropy function $A(n) = \epsilon(1+\ \cos 4\theta)$ depends on θ, the angle between the normal to the interface and a crystalline fixed axis (e.g., Z-axis). The Lewis number, $Le = \frac{\alpha}{D}$, represents the ratio of thermal diffusivity (α) to mass diffusivity (D). The interface width is defined as $W_0 = \frac{d_0\lambda}{a_1}$, involving the coupling constant λ, the capillary length d_0, and a constant a_1. Finally, the characteristic time scale is given by $\tau_0 = \left(\frac{a_1^2}{D}\right)a_2\frac{\lambda^3}{a_2^2}$.

Table 2.2: Summary of Parameters Used in the thermosolutatal model

Parameter	**Symbol**	**Description**
Characteristic time scale	τ_0	Determines the timescale for the evolution of the phase field
Partition coefficient	k	Ratio of solute concentration in the solid phase to that in the liquid phase at equilibrium
Spatial coordinate	z	Spatial position along the temperature gradient
Pulling velocity	V_p	Velocity at which the temperature gradient moves through the material
Thermal length	l_T	Characteristic length scale over which temperature varies
Phase field	ϕ	Order parameter distinguishing solid ($\phi = 1$) and liquid ($\phi = -1$) phases
Interface width	W	Thickness of the diffuse interface between solid and liquid phases
Coupling constant	λ	Controls the coupling strength between capillary length d_0 and interface width W
Dimensionless concentration	U	Non-dimensional concentration field
Solute diffusivity	D	Solute diffusion coefficient in the liquid phase
Gradient energy	$q(\phi)$	Modifies diffusivity based on phase field
Interpolation function	$a(\phi)$	Influences the mobility of the interface
Dimensionless temperature	θ	Temperature normalized relative to the melting point
Anisotropy function	$A(n)$	Direction dependence of surface energy
Lewis number	Le	Ratio of thermal diffusivity (α) to mass diffusivity (D)
Thermal diffusivity	α	Describes the rate of heat diffusion
Latent heat	L	Heat released during the phase transition
Specific heat	c_p	Heat capacity of the material
Initial solute concentration	c_∞	Far-field concentration of solute

Thermosolutal Alloy Solidification Model

The thermosolutal alloy model describes the solidification process in binary alloys, incorporating the effects of both heat and solute diffusion. This model is essential for understanding the microstructural evolution during the solidification of alloys, where the temperature and solute concentration fields are coupled. The governing equations

for the thermosolutal alloy model are presented in Ref [53]. These equations read,

$$\tau\frac{\partial\phi}{\partial t} = W^2\nabla^2\phi + \phi - \phi^3 - \lambda g'(\phi)\left(\theta + Mc_\infty U\right), \tag{2.9}$$

$$\begin{aligned}\frac{1+k}{2}\frac{\partial U}{\partial t} = \vec{\nabla}\cdot\left(D\frac{1-\phi}{2}\vec{\nabla}U + \frac{W}{2\sqrt{2}}[1+(1-k)U]\frac{\partial\phi}{\partial t}\frac{\vec{\nabla}\phi}{|\vec{\nabla}\phi|}\right)\\ +\frac{1}{2}\frac{\partial}{\partial t}\{\phi[1+(1-k)U]\},\end{aligned} \tag{2.10}$$

$$\frac{\partial\theta}{\partial t} = \alpha\nabla^2\theta + \frac{1}{2}\frac{\partial\phi}{\partial t}, \tag{2.11}$$

By design, the two sets of models are practically identical. It is important to highlight the introduction of the third kinetic equation, Eq. 2.11, which governs heat transport. The parameters in these equations are defined in Tab. 2.2.

2.4 Development of the Thermosolutal Model Algorithm

As part of this thesis and work, the author has developed algorithms that solve these model equations, specifically, Eqs. 2.9–2.11. The codes can be found in Appendix A. For Fig. 2.4, this study used 10 processors (CPUs) on one node with minimum 10 GB memory and 7 days running time using 2 x Intel Gold 6148 Skylake @ 2.4 GHz CPUs on the Béluga cluster [54]. This was for typical numerical parameters, and the timing is expected to change if numerical parameters change.

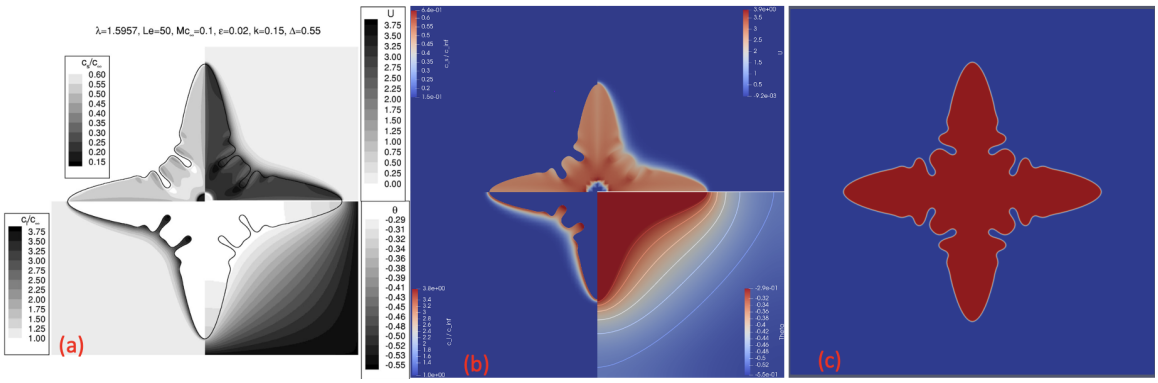


Figure 2.4: (a) Comparison between the simulation done by reference [53] (b) and this study's simulation. (c) Developed dendrite picture .

The conditions used to generate Fig. 2.4 are summarized in Tab. 2.3. The initial condition for the dimensionless concentration U is set to 0, while the initial condition for ϕ is defined using a hyperbolic tangent function. Since boundary conditions are not explicitly specified, the Finite Element Method (FEM) assumes natural Neumann boundary conditions by default.

Table 2.3: Summary of Parameters and Initial Conditions

Parameter	**Description** (Value)
dt	Time step size (0.0048)
dy	Grid spacing (0.4)
N_x	number of grid points in the x-direction (500)
N_y	number of grid points in the y-direction (500)
max_level	Maximum coarsening level (5)
w_0	Interface thickness, used as the characteristic length scale (set to 1)
τ_0	Characteristic time scale used as the reference time unit (set to 1)
a_t	anti-trapping constant, $a_t = \frac{1}{2\sqrt{2}}$
ϵ_4	Fourth-order anisotropy strength (0.02)
k_{eq}	Equilibrium partition coefficient (0.15)
λ	Coupling strength (12.7653)
Δ	Set initial condition of θ (0.55)
mc_∞	constant (0.5325)
a_1	Scaling constant (0.8839)
a_2	Scaling constant (0.6267)
d	Capillary length, $d = a_2 \cdot \lambda$
α	Thermal diffusivity, $\alpha = d \cdot le$
d_0	capillary length
Initial_Circle_radius	Initial circle radius ($65 \cdot d_0$)

2.5 Adaptive Mesh Refinement

Using an adaptive mesh provides the advantage of reducing the number of elements in the simulation domain while maintaining the required resolution in critical regions. As shown in Tab. 2.4, increasing the domain size from 80 to 160 (dimensionless by W_0) quadrupled the number of elements, from 80,000 to 320,000. Without adaptive meshing, computation time increased significantly, from 20 minutes to 1 hour and 30 minutes (4.5x). In contrast, with adaptive meshing, the increase was minimal, from 3 minutes to 3 minutes and 26 seconds, highlighting its efficiency in optimizing computational resources. This approach ensures that computational resources are focused on areas of interest, such as the interface or regions with steep gradients, while allowing coarser elements in less significant parts of the domain, thereby optimizing computational efficiency. As shown in Fig. 2.5, the mesh effectively "tracks" the interface, ensuring high resolution where it is needed, while avoiding unnecessary refinement in other regions of the domain.

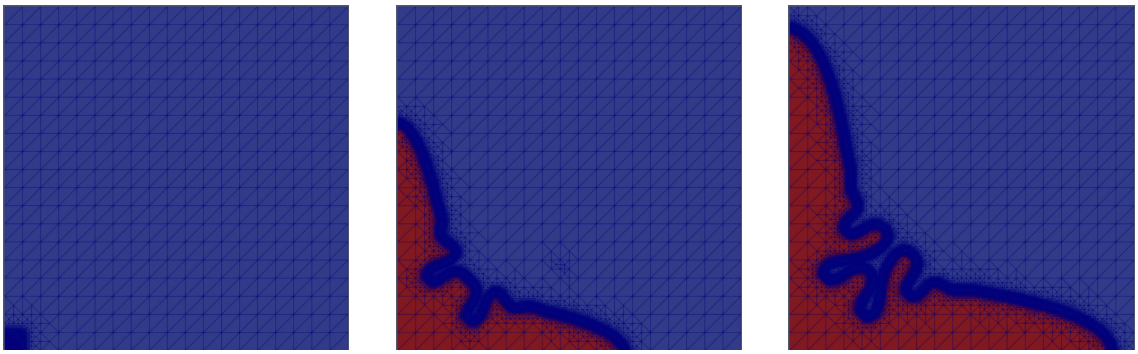


Figure 2.5: The adaptive mesh refinement process: (a) refinement around the seed, (b) tracking the interface between solid and liquid regions, and (c) expansion of the interface as the solid grows.

Using existing libraries and built-in functions in FEniCS, along with the implementation of additional functions to manage the determination of location of the diffuse interface region and parallel running communication, this study developed an adaptive meshing scheme. By incorporating ideas from [55], the adaptive meshing

scheme developed using the FEniCS library effectively tracks the interface and refines the mesh around it. See Fig. 2.5 which illustrates the mesh re-refinement at different stages.

Unlike the hierarchical mesh refinement schemes used by Freddi and Mingazzi [55], which rely on tracking parent-child relationships, this approach directly marks mesh elements based on spatial coordinates.

The method begins with a single coarse mesh and the simulation mesh. After a specified fixed number of iterations (e.g., 20), the interface coordinates (x, y) are identified based on the gradient of the order parameter ϕ. Using the coarse mesh, only the elements containing these interface region coordinates are refined. The refinement process is restricted to a specified number of refinement levels (e.g., four levels), where elements in the coarse mesh are iteratively marked and refined using FEniCS's built-in `refine()` function.

Although this approach works seamlessly in **serial execution**, challenges arise in a **parallel environment**, where the mesh is distributed across multiple processors. Since FEniCS redistributes mesh elements after each refinement, interface coordinates may correspond to elements that reside on different processors, causing potential synchronization errors.

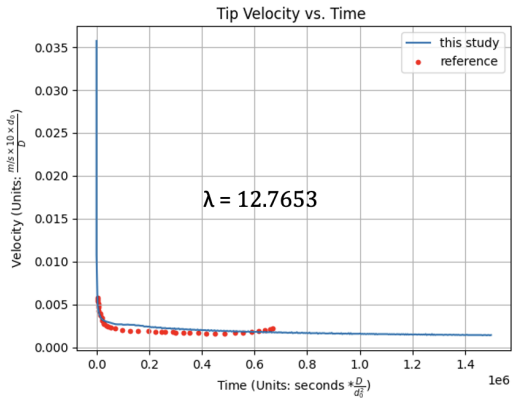
To address this, all interface coordinates are gathered on a designated root processor (e.g., `root=0`) and broadcasted to all processors. Each processor then loops through all interface coordinates, refining only the elements it owns and ignoring points outside its assigned mesh partition. This approach ensures that all interface points are considered, making the refinement process robust and compatible with parallel execution.

Table 2.4: Comparison of Adaptive and Non-Adaptive Meshing for Isothermal Phase Field Modeling. All simulations were conducted at 10^4 time steps and utilized 8 MPI processes on an M3 Apple Silicon chip with 32 GB of RAM. The relative L2 norm measures the normalized difference between the adaptive and non-adaptive solutions, using the non-adaptive solution as the reference.

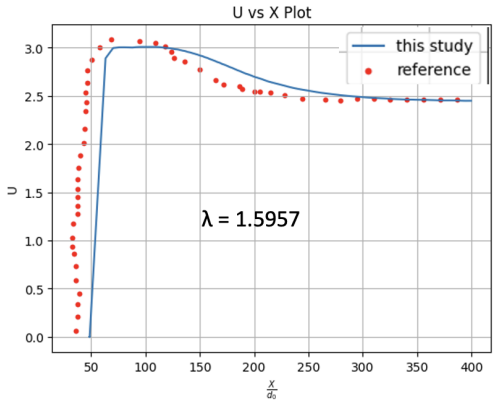
Domain Size	**Adaptive Meshing**	**Wall Clock Time (h:m:s)**	**Relative L2 Norm for ϕ Field**	**Number of Cells**
80×80	Yes	03:03	0.0001572	6,378-17,402
80×80	No	20:15	-	80,802
160×160	Yes	03:26	0.0009806	6,582-17,898
160×160	No	1:33:22	-	321,602

2.6 Thermosolutal Benchmark

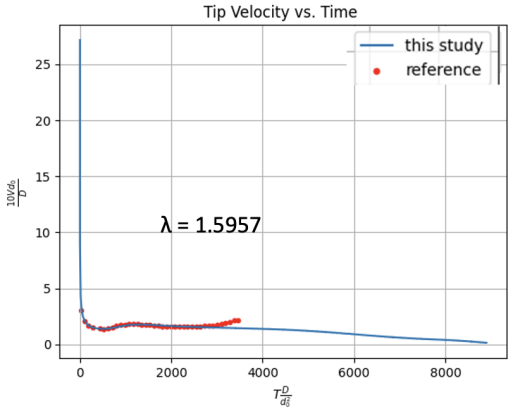
To validate and verify the algorithm, this study extracted data from the original reference article that introduced thermosolutal equations. Data extraction was performed using specialized software to obtain it from images/figures in the articles. The extracted data are represented by red dots in Fig. 2.6. The correspondence between this extracted data and the blue graph, which represents the data from this study, was then investigated. Some errors are related to the data extraction process from the article images.



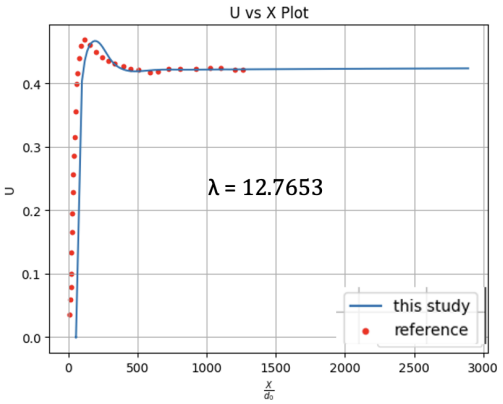
(a) Tip velocity in x-direction vs time.



(b) U or dimensionless concentration vs x-coordinate.



(c) Tip velocity in x-direction vs time.



(d) U or dimensionless concentration vs x-coordinate.

Figure 2.6: The blue line represents data from this study, while the red dots were extracted from the reference article [53] images using specialized software [56]. The different λ values used in the reference article demonstrate that the coupling parameter λ has no effect on the model's convergence.

The article [53] presents two sets of graphs: one showing the tip velocity along the x-axis versus time, and the other depicting the dimensionless concentration U with respect to the x-coordinate. The λ values used in the reference article demonstrate that the coupling parameter λ has no effect on the model's convergence.

The parameter λ couples the interface width W with the capillary length d_0, where the capillary length represents the physically significant factor, while the interface width W is an arbitrarily chosen numerical parameter. This distinction explains why the article uses two different λ values to validate the same convergence behavior,

confirming that the computed results remain consistent regardless of the choice of W.

A summary of the simulation parameters can be found in Tab. 2.3, which are used in the simulation shown in Fig. 2.4 and benchmarked in Fig. 2.6.

Chapter 3

Fluid Flow

3.1 Navier–Stokes Equation

The Navier–Stokes equations, developed in the 19th century by Claude-Louis Navier and George Gabriel Stokes, transformed the field of fluid mechanics. These equations capture the intricate interplay of inertia, viscosity, pressure, and external forces to describe fluid motion. Serving as the cornerstone of modern fluid dynamics, they enable the modeling of phenomena ranging from blood flow in veins to turbulent airflows around aircraft [57]. Despite their apparent simplicity, the Navier–Stokes equations are notoriously challenging to solve due to their nonlinear nature and the complex coupling between velocity and pressure fields. This inherent difficulty has driven extensive research, ranging from analytical solutions for simplified cases to advanced computational methods that tackle their complexity in real-world applications.

At the heart of fluid dynamics lies the Navier–Stokes equations, describing the conservation of momentum(Eq. 3.1) for a fluid. These equations are derived from Newton's second law and augmented with constitutive relations for fluid stresses. Mathematically, they are expressed as:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right)=-\nabla p+\nabla\cdot[\mu(\nabla\mathbf{u})]+\rho\mathbf{g}+\vec{f} \tag{3.1}$$

Accompanying the momentum equation is the continuity equation, which ensures the conservation of mass:

$$\frac{d\rho}{dt}+\rho\nabla\cdot\mathbf{u}=0 \tag{3.2}$$

Fluid motion rarely exists in isolation; it interacts with other physical phenomena in fascinating ways. One such interaction is the Marangoni effect, where surface tension variations due to temperature gradients induce fluid motion. This effect, coupled with buoyancy forces arising from density variations, plays a critical role in processes such as welding, crystal growth, and additive manufacturing. Mathematically, the contribution of surface tension to fluid motion is expressed as:

$$\vec{s}_{\text{surface tension}}=\frac{\partial\sigma}{\partial T}\gamma\nabla_t T,\quad \text{where}\quad \gamma=\left|\frac{d\sigma}{dT}\right|, \tag{3.3}$$

where σ represents the surface tension, T denotes the temperature, and $\nabla_t T$ corresponds to the tangential temperature gradient. The coefficient γ characterizes the sensitivity of surface tension to temperature variations, emphasizing the role of thermocapillary forces in influencing fluid dynamics.

To simulate fluid flow governed by these physical principles, an appropriate numerical method must be selected. Common approaches include the finite difference method, the finite volume method, and the FEM. In this work, FEM is employed due to its flexibility and accuracy in handling complex geometries and boundary conditions.

In FEM, equations such as (3.1) and (3.2) are initially presented in their **strong form**, which directly represents the governing partial differential equations (PDEs). However, FEM requires these equations to be reformulated into the **weak form** to enable their discretization and numerical solution.

The weak form is derived by multiplying the strong form of the equations by test functions and integrating over the domain, thereby redistributing the differentiability requirements across the solution space. This method is essential for solving PDEs using the FEM and forms the basis for the numerical simulations in this study.

For detailed methods on deriving and solving the weak form of the Navier–Stokes equations, readers can refer to Peterson et al. (2018) [58], which employs a monolithic approach, or Dokken et al. (2023) [59], which uses a splitting method. Both approaches offer valuable insights into addressing the challenges of fluid flow simulations using FEM.

3.2 Fluid flow coupled with Marangoni and Buoyancy

To develop the fluid flow model, the author began by implementing a FEM code using the FEniCS library. Before coupling the fluid flow model with other complex physics, such as solidification, it was crucial to verify the correctness of the implementation. This step ensures that the partial differential equations (PDEs) and their corresponding forms are accurately defined and solved. Verification is an essential process because, in coupled simulations, errors can arise from multiple sources, making it challenging to identify the root cause of convergence issues. By thoroughly testing and isolating the fluid flow model, the author ensures confidence in its accuracy. This foundation allows any subsequent errors to be attributed to the newly introduced physics, rather than the fluid flow model, streamlining debugging and validation efforts.

The benchmarked results are essential to validate the accuracy and reliability of the developed fluid flow model. These benchmarks, including the Marangoni effect and buoyancy-driven convection, are critical for understanding fluid behavior in additive

manufacturing processes. The examples were chosen because they represent key physical phenomena (see Fig 3.1) that dominate in such scenarios, providing a robust basis for comparing the FEniCS implementation with the established COMSOL Multiphysics results. By demonstrating agreement between these models, the benchmarks confirm the fidelity of the simulation framework and its capability to handle coupled fluid flow phenomena.

To develop this model, an initial step involves reproducing and benchmarking a simulation of fluid flow using COMSOL Multiphysics [60]. COMSOL was chosen for this purpose because it employs finite element discretization for solving physical phenomena, making it highly relevant to the problem at hand. Additionally, as a widely recognized and commercially available code, COMSOL is trusted in industry and academia, making it a reliable reference for validating and verifying the model.

The example in Ref. [60] showcases a simulation (see Fig. 3.2) of the Marangoni effect using COMSOL Multiphysics. This effect, occurring due to changes in surface tension caused by temperature differences across a fluid interface, is demonstrated in a 2D model of a vessel filled with silicone oil. The setup for the simulation involves applying a temperature gradient across the oil, which adjusts the surface tension and induces fluid flow. The primary aim is to analyze the resulting temperature field and flow pattern within the vessel. To achieve this, the model employs the Boussinesq approximation [61] for considering the density variations due to temperature changes, which influence the fluid dynamics. The summary of the simulation parameters is in Tab. 3.1.

Table 3.1: Physical constants and temperature parameters used in the simulation of the Marangoni effect [60].

Parameter	Value	Description
Physical Constants		
Gravitational acceleration (g)	$-10\,\mathrm{m/s^2}$	Acceleration due to gravity.
Density (ρ)	$760\,\mathrm{kg/m^3}$	Fluid density.
Dynamic viscosity (μ)	$4.94 \times 10^{-4}\,\mathrm{Pa \cdot s}$	Dynamic viscosity of the fluid.
Thermal conductivity (k)	$0.1\,\mathrm{W/(m \cdot K)}$	Thermal conductivity of the fluid.
Heat capacity (C_p)	$2090\,\mathrm{J/(kg \cdot K)}$	Specific heat capacity.
Thermal expansion coefficient (α)	$1.3 \times 10^{-3}\,\mathrm{1/K}$	Coefficient of thermal expansion.
Surface tension temperature derivative (γ_T)	$-8 \times 10^{-5}\,\mathrm{N/(m \cdot K)}$	Change in surface tension with temperature.
Temperature Constants		
Reference temperature (T_{ref})	$273.15\,\mathrm{K}$	Reference temperature.
Right boundary temperature (T_{right})	$273.15\,\mathrm{K}$	Temperature on the right boundary.
Temperature difference (ΔT)	$2\,\mathrm{K}$	Temperature difference across the domain.
Left boundary temperature (T_{left})	$T_{\mathrm{right}} + \Delta T = 275.15\ \ \mathrm{K}$	Temperature on the left boundary.

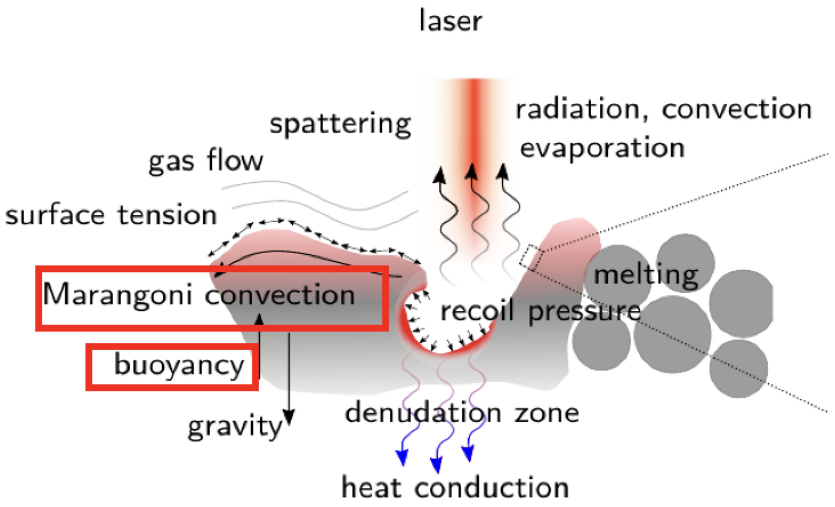


Figure 3.1: Key physical phenomena chosen for inclusion in fluid flow are highlighted here. The image is reproduced from the MeltPoolDG repository [19].

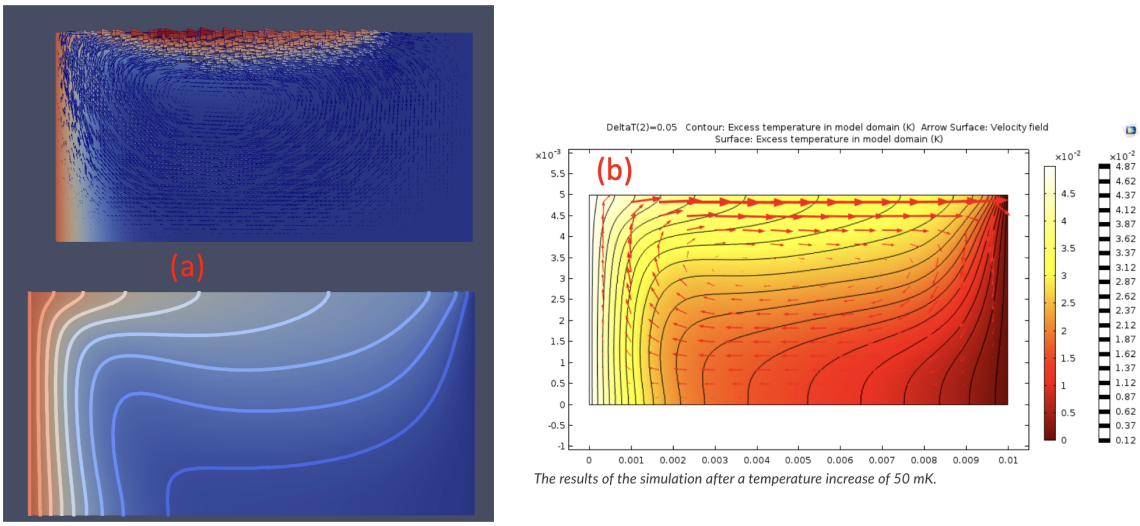


Figure 3.2: The temperature gradient between the left and right walls induces a force on the top boundary due to the temperature-dependent surface traction coefficient, representing the Marangoni force, which drives the fluid flow. In (a) (top), velocity vectors are illustrated using ParaView, while the corresponding isotherm lines are shown at the bottom. In (b), both isotherm lines and velocity vectors are plotted using COMSOL, enabling a comparative visualization of the models developed in FEniCS [62] and COMSOL [44].

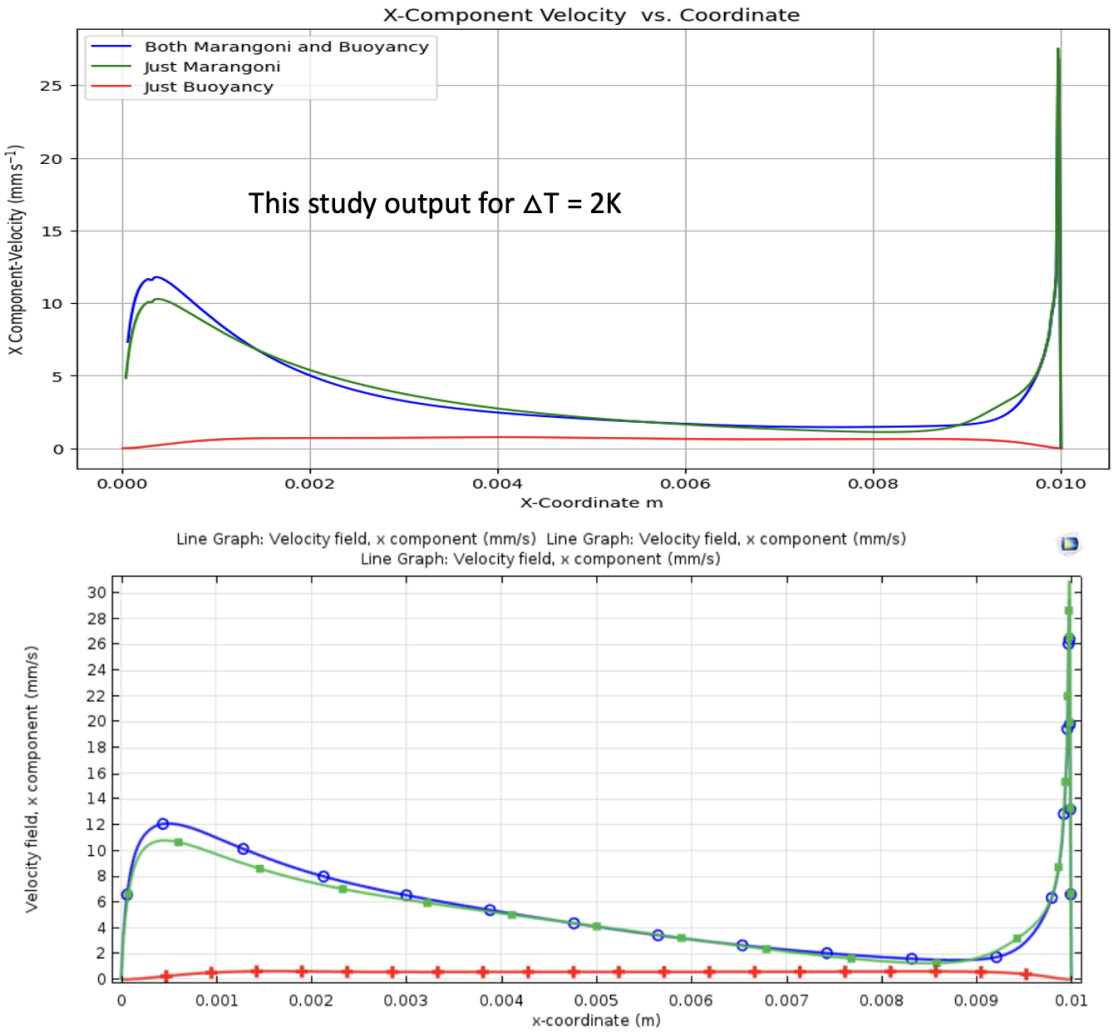


Figure 3.3: Quantitative comparison of the fluid flow models: FEniCS (top) and COMSOL (bottom) [44].

Figure 3.3 presents a quantitative comparison between the developed FEniCS model and the COMSOL example. The blue curve represents the combined effects of Marangoni convection and buoyancy, while the green and red curves depict the Marangoni and buoyancy effects individually. The results indicate that the Marangoni effect dominates over buoyancy at high temperature gradients, as shown by the greater surface velocities in the green curve compared to the red.

Chapter 4

Mutliphysics: Coupling Phase field with Fluid Flow

4.1 Alterations in Mathematical Formulations Due to Coupling

When a laser strikes the powder in additive manufacturing, it melts the material to create a molten pool. Within this melt pool, fluid flow occurs due to physical phenomena such as surface tension gradients and buoyancy forces. As the laser moves away and the temperature cools, solidification begins at the bottom of the melt pool. This growing solid alters the geometry of the melt pool and blocks fluid flow, disrupting existing flow patterns. The fluid flow within the melt pool also transports solute concentration and redistributes heat, affecting the temperature profile ahead of the solid-liquid interface. These changes, in turn, influence the growth dynamics of the solid, creating a tightly coupled interaction between fluid dynamics and solidification processes.

To simulate and model these governing physics, the PDEs describing fluid flow and solidification must be coupled and modified. The solidification equations, as introduced

in Eqs. (2.7) and (2.8), require the inclusion of convection terms to account for the influence of fluid flow (refer to Eqs. 4.1 and 4.3). Note in this model, the temperature field is not explicitly solved. Instead, temperature-driven effects such as buoyancy and Marangoni forces are incorporated through predefined temperature gradients or temperature-dependent coefficients. While directly solving the temperature field would provide greater precision, this approach simplifies the model, allowing for a step-by-step increase in complexity as the simulation framework evolves. Similarly, the Navier–Stokes equations, which govern the fluid flow, must account for the presence of the emerging solid. As the solid obstructs the flow, the equations need to adapt, ensuring that fluid flows around the solid rather than through it (refer to Eq.4.4) [27]. This interplay is essential for accurately capturing the multiphysics behavior in the melt pool.

$$\begin{aligned}\tau(\mathbf{n})\left[\frac{\partial\phi}{\partial t}+\mathbf{v}\cdot\nabla\phi\right] &= (\phi-\phi^3)-\lambda(\theta+Mc_\infty U)(1-\phi^2)^2 \\ &+\nabla\cdot[W(\mathbf{n})^2\nabla\phi]+\frac{\partial}{\partial y}\left(|\nabla\phi|^2W(\mathbf{n})\frac{\partial W(\mathbf{n})}{\partial(\partial_y\phi)}\right) \\ &+\frac{\partial}{\partial x}\left(|\nabla\phi|^2W(\mathbf{n})\frac{\partial W(\mathbf{n})}{\partial(\partial_x\phi)}\right)\end{aligned} \tag{4.1}$$

Adding convection $(\mathbf{v}\cdot\nabla c)$ to the equation of concentration evolution [27]:

$$\frac{\partial c}{\partial t}+\mathbf{v}\cdot\nabla c=\nabla\cdot\left[Dc\frac{(1-\phi)}{2}\nabla u-\mathbf{j}_{\text{at}}\right] \tag{4.2}$$

where c denotes the solute concentration, and $\mathbf{j}_{\text{at}}=-2W(1-k)c_0e^u\frac{\partial\phi}{\partial t}\frac{\nabla\phi}{|\nabla\phi|}$ represents the solute flux from solid to liquid along the interface normal, mitigating unphysical effects associated with the finite interface thickness. The dimensionless variable u for c is defined as

$u=\ln\left(\frac{2c}{c_0[1+k-(1-k)\phi]}\right)$,

capturing the deviation of the chemical potential from equilibrium at tempera-

ture T_0 with liquidus concentration c_0. To facilitate numerical computations, the transformation

$U = \frac{e^u - 1}{1-k}$

(see Tab. **??**) is introduced, enabling the reformulation of the solute conservation equation in Equation (2) using the dimensionless variable U [27]. It is important to note that this formulation assumes isothermal or frozen temperature conditions, neglecting the effects of explicit convective flow in heat transport.

$$\begin{aligned}\frac{[1+k-(1-k)\phi]}{2}\frac{\partial U}{\partial t} + \mathbf{v}\cdot\left\{\frac{[1+k-(1-k)\phi]}{2}\nabla U - \frac{[1+(1-k)U]}{2}\nabla\phi\right\} \\ = \nabla\cdot\left\{D\frac{(1-\phi)}{2}\nabla U + \frac{1}{2\sqrt{2}}[1+(1-k)U]\frac{\partial\phi}{\partial t}\frac{\nabla\phi}{|\nabla\phi|}\right\} \\ + \frac{[1+(1-k)U]}{2}\frac{\partial\phi}{\partial t}\end{aligned} \tag{4.3}$$

The effect on the Navier-Stokes equations is then related to ϕ, the phase indicator, where the density and viscosity are approximated using an interpolation function. For example, the viscosity is expressed as $\nu(\phi) = \frac{\nu_s(1+\phi)}{2} + \frac{\nu_l(1-\phi)}{2}$.

$$\begin{aligned}\frac{\partial\vec{v}}{\partial t} - \nabla\cdot[\nu(\phi)\nabla\vec{v}] + (\vec{v}\cdot\nabla)\vec{v} = -\nabla\frac{p}{\rho_l} \\ + \left(\frac{1-\phi}{2}\right)\alpha_c(c-c_0)\vec{g} \\ + \left(\frac{1+\phi}{2}\right)\left(\frac{\rho_s}{\rho_l} - 1\right)\vec{g}\end{aligned} \tag{4.4}$$

This equation is a modified Navier-Stokes equation adapted for a phase-field model, accounting for fluid dynamics in a two-phase solidification process. The equation details the evolution of the velocity field $\vec{v}$ and includes:

- Viscous force represented by $-\nabla\cdot[\nu(\phi)\nabla\vec{v}]$, where $\nu(\phi)$ adjusts viscosity between solid and liquid phases.
- Inertial force as $(\vec{v}\cdot\nabla)\vec{v}$.

- Pressure gradient effect $-\nabla \frac{p}{\rho_l}$ normalized by the liquid density ρ_l.
- Buoyancy force from solutal expansion $\left(\frac{1-\phi}{2}\right) \alpha_c (c - c_0)\vec{g}$, with α_c as the solutal expansion coefficient.
- Gravitational force difference between the phases $\left(\frac{1+\phi}{2}\right) \left(\frac{\rho_s}{\rho_l} - 1\right) \vec{g}$, depicting the density contrast between solid (ρ_s) and liquid (ρ_l).

The presented mathematical framework enables the simulation of fluid behavior during complex processes such as alloy solidification. In the solid region, viscosity is assumed to be extremely high to enforce a near-zero velocity, allowing a unified equation to model both solid and liquid phases simultaneously [27].

4.2 Algorithmic Considerations

To effectively couple fluid flow with the PFM, two distinct physics frameworks are employed: one for the Navier–Stokes equations and another for the phase-field equations. Each framework operates on its own computational mesh, enabling tailored discretizations suited to their respective physics.

The coupling process follows an iterative procedure. First, the Navier–Stokes equations are solved to compute the fluid velocity field. These velocities are then interpolated onto the phase-field mesh, where they are incorporated into the convection terms of the phase-field equations. Once the phase-field equations are solved, the resulting variables, such as the order parameter ϕ (indicating the solid or liquid fraction) and the dimensionless concentration U, are transferred back to the Navier–Stokes framework.

In the Navier–Stokes domain, ϕ is interpolated to the Navier–Stokes mesh to update the effective viscosity and density fields, ensuring that the fluid dynamics accurately reflect the evolving solid-liquid interface. This iterative loop continues until

convergence is achieved, ensuring a consistent and stable solution across both physical domains.

This is referred to as a partitioned (or *segregated*) approach, and can offer flexibility in choosing specialized solvers or techniques suited to each physics component. However, it can be challenging to achieve convergence, especially for strongly coupled problems. Conversely, in a monolithic approach, all equations governing the different physics involved are assembled into a single system of equations and solved simultaneously. This method generally provides better stability and convergence properties, especially in tightly coupled multiphysics problems, as it handles all interactions within a single solver framework. However, it can be computationally expensive leading to impractical times for simulation completion.

Different solvers can be employed like using an opensource software package, OpenFoam, to solve the fluid flow component, followed by the application of FEniCS for the PFM. This method leverages the robust finite volume method (FVM) and the well-established stabilization techniques available in OpenFoam, which are particularly advantageous for handling turbulent flow models. Subsequently, the velocity field data obtained from OpenFoam is transferred to FEniCS, where the FEM is employed to address the solidification processes. This iterative loop continues until the solution converges. The integration of OpenFoam and FEniCS has been successfully demonstrated in previous studies by authors such as Rodenberg et al. [63], showcasing the efficacy and potential of combining these tools for multiphysics simulations.

The author also developed phase-field model codes in OpenFOAM using the finite volume method (FVM) and implemented finite element method (FEM) codes for the Navier–Stokes and phase-field equations in MOOSE. The MOOSE framework's built-in MultiApps functionality was utilized to couple these physics in a segregated manner. Furthermore, the MultiApps approach was employed to test the coupling

of the FVM-based Navier–Stokes discretization with the heat equation, where fluid convection solved in the FVM domain was transferred to the FEM domain in MOOSE for further computations.

Furthermore, the models initially developed in FEniCS were also implemented in MOOSE, providing alternative frameworks for simulation. For more information and access to the codes, which may offer guidance or inspiration for implementation, refer to Section 7.2.

4.3 Benchmark Assessment: Dendritic Growth Under Convection

Beckermann et al. [45] present a simulation involving fluid inflow from the top while solidification occurs under flow conditions. This example provides an excellent basis for verifying the code, as it utilizes the same partial differential equations (PDEs) and governing physics applied in additive manufacturing. To adapt this example for the specific application introduced in Section 6, only the boundary conditions (BC) and initial conditions (IC) require modification. This approach guides the development of the model.

The governing equations for a pure material, based on the PF formulation presented in [47], are extended below to include convection terms for coupling fluid flow with phase evolution and temperature. These equations couple the evolution of the order parameter (ϕ), dimensionless temperature (u), velocity field ($\mathbf{v}$), viscosity (ν)—which depends on the order parameter—and density (ρ):

$$\begin{aligned}\tau(\mathbf{n})\left[\frac{\partial\phi}{\partial t}+\vec{v}\cdot\nabla\phi\right] &= (\phi-\phi^3)-\lambda u(1-\phi^2)^2 \\ +\nabla\cdot[W(\mathbf{n})^2\nabla\phi]+\frac{\partial}{\partial y}&\left(|\nabla\phi|^2W(\mathbf{n})\frac{\partial W(\mathbf{n})}{\partial(\partial_y\phi)}\right) \\ +\frac{\partial}{\partial x}&\left(|\nabla\phi|^2W(\mathbf{n})\frac{\partial W(\mathbf{n})}{\partial(\partial_x\phi)}\right)\end{aligned} \tag{4.5}$$

$$\frac{\partial u}{\partial t}+\vec{v}\cdot\nabla u = D\nabla^2 u+\frac{1}{2}\frac{\partial\phi}{\partial t} \tag{4.6}$$

The Navier–Stokes equations are solved using a segregated approach with respect to the phase-field equations. In this approach, the velocity and pressure fields are computed separately from the phase-field and temperature equations in each iteration. After solving the phase-field equations, the updated velocity and pressure fields are passed to the phase-field module to account for multiphysics coupling. The governing equations for the flow are:

$$\frac{\partial\vec{v}}{\partial t}-\nabla\cdot[\nu(\phi)\nabla\vec{v}]+(\vec{v}\cdot\nabla)\vec{v} = -\nabla\frac{p}{\rho_l} \tag{4.7}$$

$$\frac{d\rho}{dt}+\rho\nabla\cdot\vec{v}=0 \tag{4.8}$$

By solving Eqs. (4.5)–(4.8) under the conditions outlined in [45] and summarized in Tab. 4.1, the coupling implementation developed in this study was validated against the implementation of the example presented in the referenced work.

The boundary conditions enforce a no-slip velocity ($v=0$) on the left and right walls, while the top boundary applies an inlet-flow velocity with specified components (v_x, v_y). The bottom boundary sets the pressure to zero.

The initial condition for the Navier-Stokes equations assumes a zero velocity and zero pressure field. For the phase-field equation, a Neumann boundary condition (zero flux) is imposed for both the dimensionless temperature (u) and the order parameter

(ϕ).

The initial condition for ϕ is set to $\phi = 1$ in the solid and $\phi = -1$ in the liquid, with a smooth transition modeled using a tanh function. For the dimensionless temperature (u), the initial value is set to -0.55.

Table 4.1: Simulation Parameters for Phase-Field and Navier-Stokes Coupling (All Values in SI Units)

Parameter	**Value**	**Description**
Phase-Field Model Parameters		
dy	0.4	Mesh resolution (base level)
max_level	3	Maximum refinement level
Nx, Ny	250, 250	Domain resolution
dt	8×10^{-3}	Time step size
$seed\_center$	[125, 125]	Initial seed center
a_1, a_2	0.8839, 0.6637	Coefficients
w_0	2.589×10^{-7}	Interface thickness
τ_0	1.64×10^{-6}	Relaxation time
ep_4	0.05	Anisotropy parameter
k_{eq}	0.14	Equilibrium partition coefficient
$u_{initial}$	-0.55	Initial dimensionless concentration
$initial\_seed\_radius$	8.27	Initial seed radius
Navier-Stokes Parameters (Scaled to Phase-Field Dimensionalization)		
$gravity$	-10	Scaled gravitational acceleration
$\rho_{liquid}, \rho_{solid}$	2.45, 2.7	Densities of liquid and solid phases
μ_{fluid}	1.4×10^{-3}	Dynamic viscosity (scaled)
μ_{solid}	$\mu_{\text{fluid}} \times 10^6$	Viscosities of solid and liquid phases
α_c	9.2×10^{-3}	Solutal expansion coefficient
Inflow velocity	(0.0, -1.0)	Inflow velocity at top boundary condition

Figure 4.1 presents the simulation results obtained in this study, which are based on the same boundary conditions, initial conditions, and physical setup as those shown in Fig. 4.2 from Ref. [45]. While the two figures simulate the same physical scenario, they employ different mathematical models and PDE formulations to describe the physics. It is notable that the PDEs used by Beckermann et al. [45] do not explicitly include convection terms, such as $\vec{v} \cdot \nabla\phi$. Additionally, they highlight that their method does not require prescribing a variable viscosity across the diffuse interface region that increases significantly in the rigid solid. While this approach may be physically reasonable for certain materials, they argue that accurately defining the

viscosity variation for a rigid solid would be both challenging and impractical.

Furthermore, by incorporating the equations introduced in Eq. (4.3), this study captures the convection-driven growth of dendrite arms in a bent shape, aligned with the direction of fluid flow.

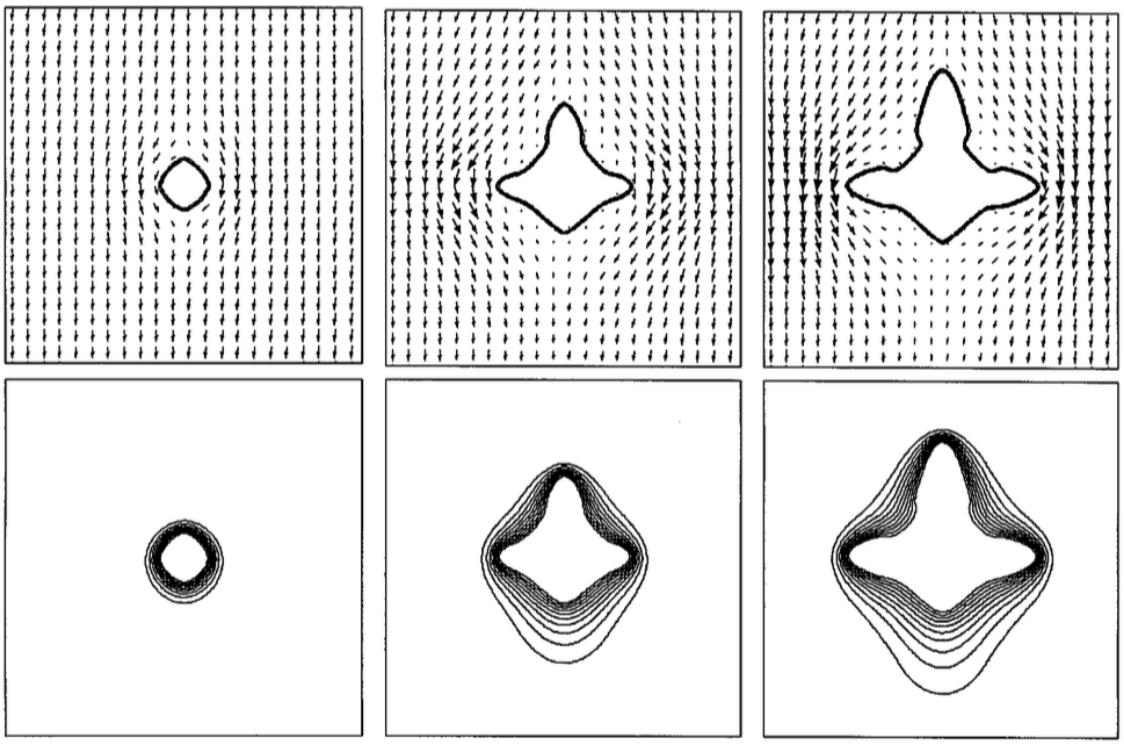
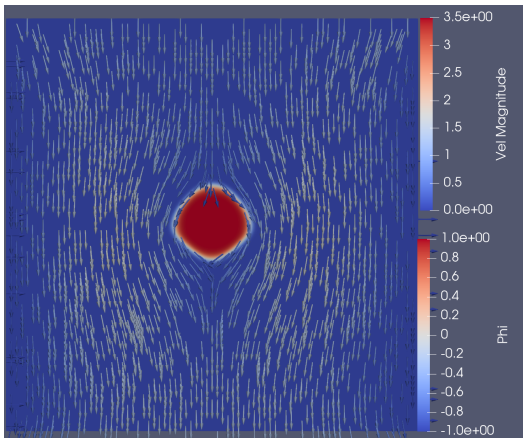
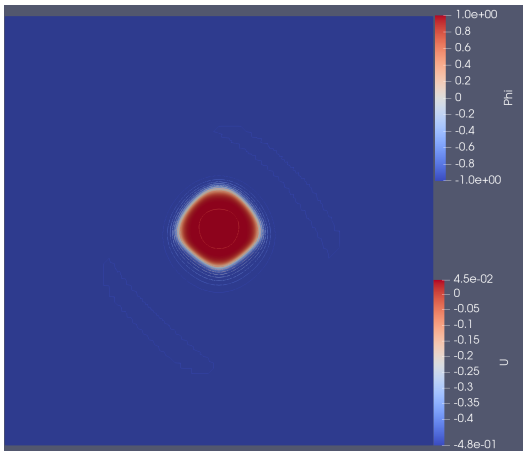


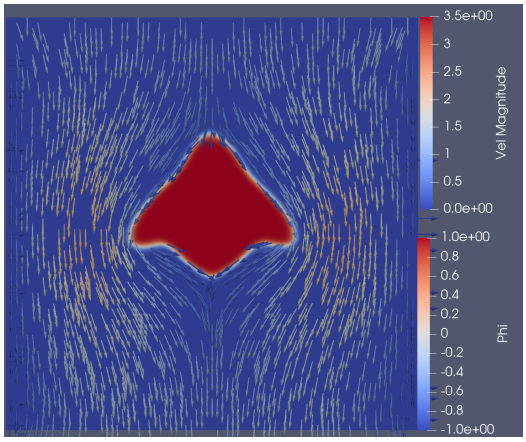
Figure 4.2: Top panels show computed phase-field contours and velocity vectors, while bottom panels display isotherms from the dendritic growth simulation with convection at $t = 15, 66$, and 96 (in units of τ, from left to right) [45].



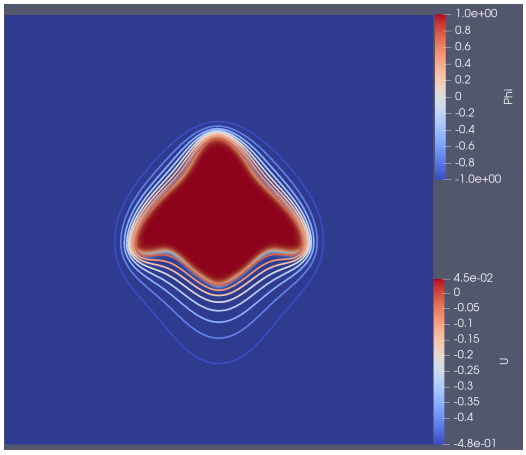
(a) Time = 15, Velocity Field



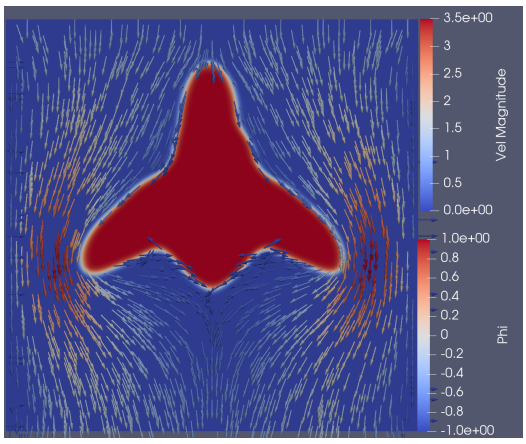
(b) Time = 15, Isotherms



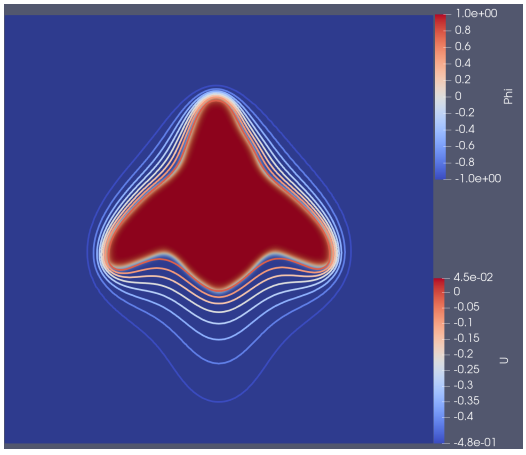
(c) Time = 66, Velocity Field



(d) Time = 66, Isotherms

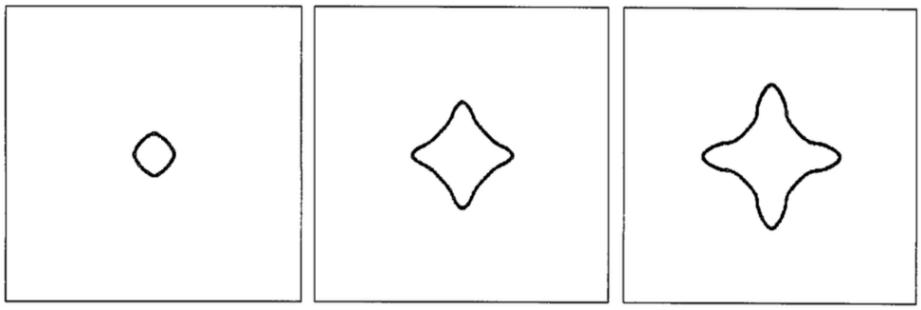


(e) Time = 96, Velocity Field

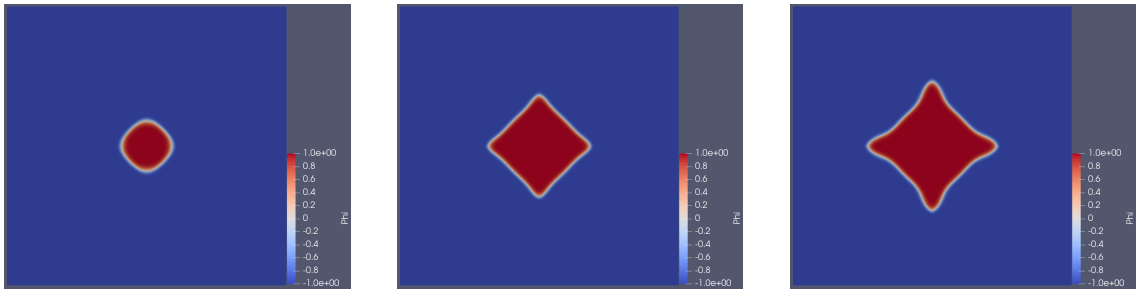


(f) Time = 96, Isotherms

Figure 4.1: Evolution of the velocity field and isotherms at different times (in units of τ.



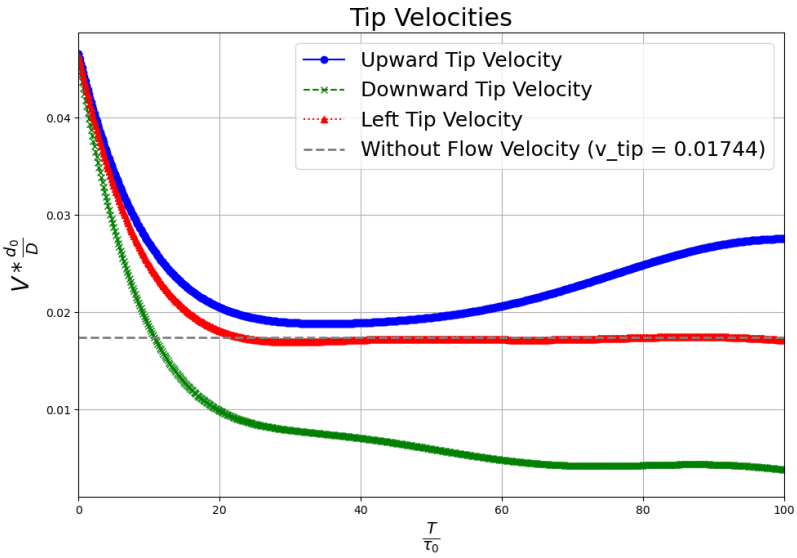
(a) Dendritic growth simulation without convection from reference [45].



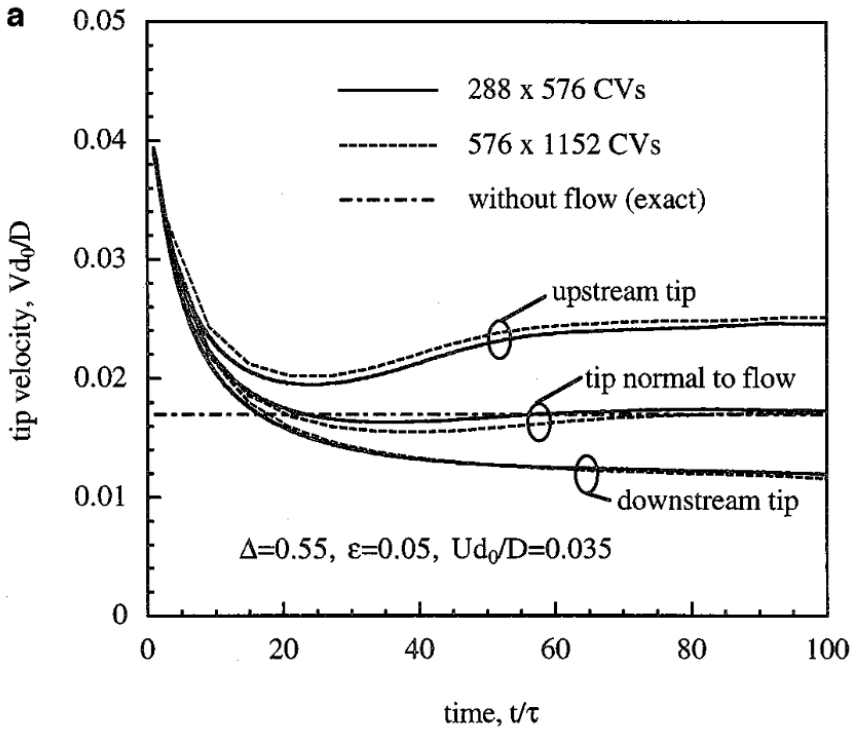
(b) Time = 16 (c) Time = 66 (d) Time = 96

Figure 4.3: Comparison of the reference image from [45] with the simulated states at different time steps.

This chapter has demonstrated the coupling of phase-field and fluid flow models to capture the complex interactions during solidification under convective flow conditions. By incorporating convection terms into the phase-field and solute conservation equations, and modifying the Navier-Stokes equations to account for solid obstruction, the model accurately simulates the dynamics of dendritic growth. The benchmark example of dendritic growth under convection, taken from [45], verified the coupling implementation and highlighted its ability to replicate realistic physics. Furthermore, since the same governing equations apply, the code developed can be adapted for additive manufacturing conditions with appropriate modifications to the initial conditions, boundary conditions and driving forces.



(a) Tip velocities from the simulation (Fig 4.1) : upward tip velocity shown as a blue line, downward tip velocity as a green line, left tip velocity as a red line, and flow velocity represented by a dashed grey line.



(b) Tips velocities of the reference [45].

Figure 4.4: Tip velocities of the dendritic interface were calculated by identifying the interface location ($\phi = 0$) at different time steps and fitting a polynomial to the position data. The derivative of the fitted polynomial provided the velocity, which was scaled using a conversion factor to match the physical units. The velocities are shown for downward, upward, and axis-parallel directions along the dendrite arms, illustrating the dynamics of interface motion in different orientations.

Chapter 5

Phase-Field Simulation of Laser Deposited Ni–Nb Alloy

In this chapter, we implement the developed phase-field microstructural model to study solidification under controlled thermal scenarios. These simulations serve as a baseline to assess the influence of fluid flow on AM processes in our subsequent work.

The thermal scenarios are inspired by the work of Ghosh et al. [64] and involve distinct cooling rates that mimic conditions observed in AM. A relatively slow cooling rate is explored, representative of typical equilibrium solidification. Additionally, a rapid cooling rate is simulated, capturing the non-equilibrium conditions characteristic of AM.

The focus of this chapter is to characterize microstructural evolution under these controlled thermal gradients without fluid flow. This includes evaluating how the cooling rate impacts growth dynamics, and overall microstructural patterns. The simulations provide a baseline for defining quantitative metrics, such as tip velocity and microsegregation profiles, which will be used to assess the effects of fluid flow in future work.

These results form a critical foundation for understanding how fluid flow modifies

microstructural evolution in AM, ensuring that the interplay between thermal and fluid effects can be isolated and quantified.

The governing equations for our simulations are presented in Eqs. (2.7) and (2.8), that is in the absence of flow. The two scenarios we explore are as follows:

(a) $\dot{T} = 3 \times 10^5\,\mathrm{K\,s^{-1}}$, calculated using the product $G \times V$, where $G = 10^7\,\mathrm{K/m}$.

(b) $\dot{T} = 1 \times 10^6\,\mathrm{K\,s^{-1}}$, similarly determined using $G = 10^7\,\mathrm{K/m}$.

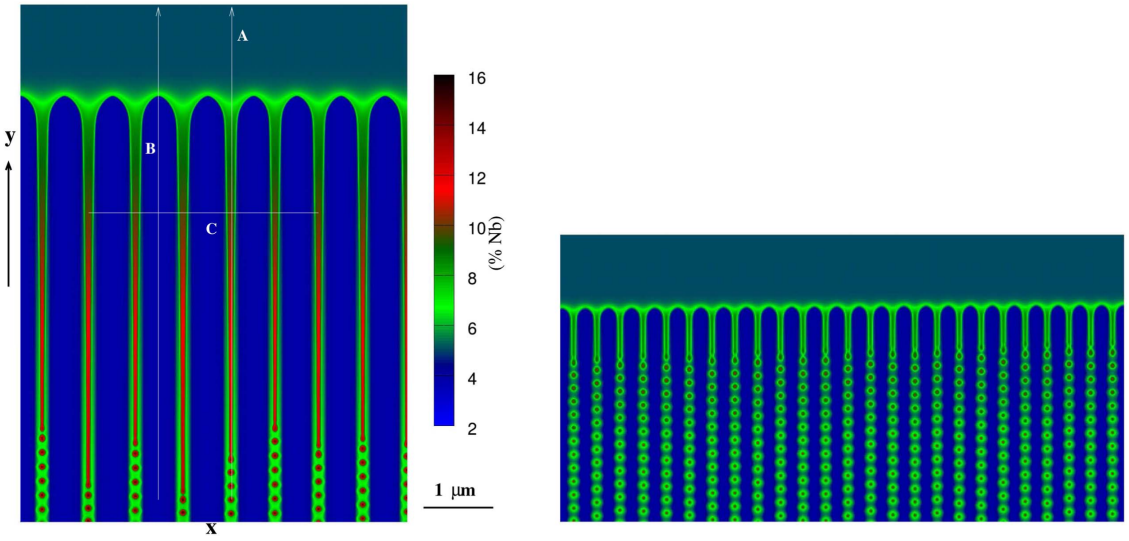
These conditions ensure that our simulations closely reflect the physical parameters commonly encountered in laser deposition processes, akin to those described by Ghosh et al. [64]. The primary objective is to examine the effects of varying cooling rates on solidification.

Table 5.1: Simulation Parameters for Phase-Field Simulation of Laser Deposited Ni–Nb Alloy

Parameter	**Value**	**Description**
Physical Parameters		
dt	0.13	Time step size
dy	0.8	Mesh resolution
Nx, Ny	500, 4000	Domain size
max_level	5	Maximum refinement level
y_solid	20	Initial height of Solid
w_0	1×10^{-8} m	Interface thickness scaling factor
τ_0	2.30808×10^{-8} s	Time scaling factor
G	1×10^{7} K/m	Thermal gradient
V	3×10^{-2} m/s	Pulling velocity
m_l	10.5 K/%	Slope of liquidus line
c_0	5 %	Initial solute concentration
ep_4	0.03	Anisotropy parameter
k_{eq}	0.48	Equilibrium partition coefficient
λ	1.377	Diffusion length parameter
a_1, a_2	0.8839, 0.6267	coefficients
d_l	0.863	Diffusivity in liquid phase
d_s	2.877×10^{-4}	Diffusivity in solid phase
d_0	8×10^{-9} m	Capillarity length
Solver Parameters		
abs_tol, rel_tol	$1 \times 10^{-6}, 1 \times 10^{-5}$	Absolute and relative solver tolerances
Nonlinear solver	Newton	Nonlinear solver for phase-field
Linear solver	MUMPS	Linear solver for phase-field
Preconditioner	ILU	Preconditioner for phase-field
Maximum iterations	50	Maximum iterations for phase-field solver

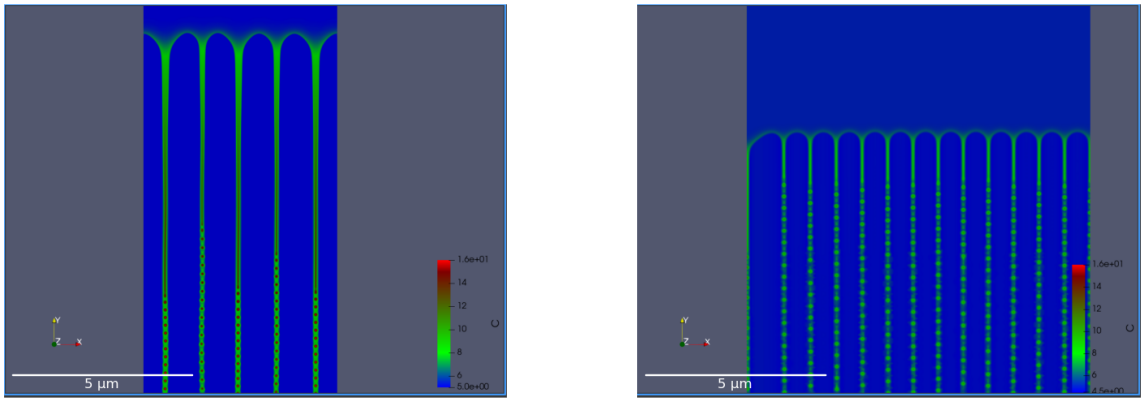
The job script for these simulations has been optimized to use 30 tasks per node,

with a total memory allocation of 245 GB per node, and a runtime limit of 7 days, ensuring efficient execution of the computational workload. The model was developed in FEniCS, and the setup parameters are provided in Tab. 5.1.



(a) (b)

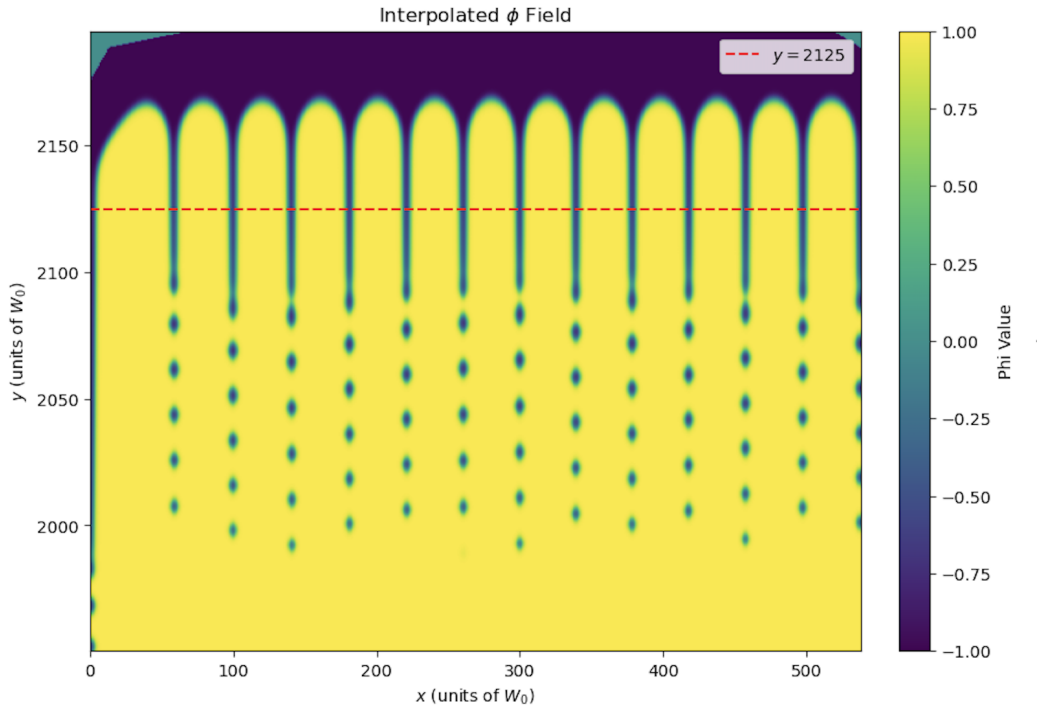
(a) Steady-state cellular growth fronts under two solidification conditions are shown as Nb composition maps. The colour bar indicates the Nb mass fraction, with the growth direction (y) oriented vertically. Key features include uniform cell spacing, composition variations across cells, and Nb-enriched droplets at cell grooves. In (a), for $\dot{T} = 3 \times 10^5$ K s^{-1}, the primary dendrite arm spacing (PDAS) is 0.71μm, while in (b), for $\dot{T} = 10^6$ K s^{-1}, the PDAS is 0.23μm. The coarser cells in (a) result from the lower cooling rate [64].



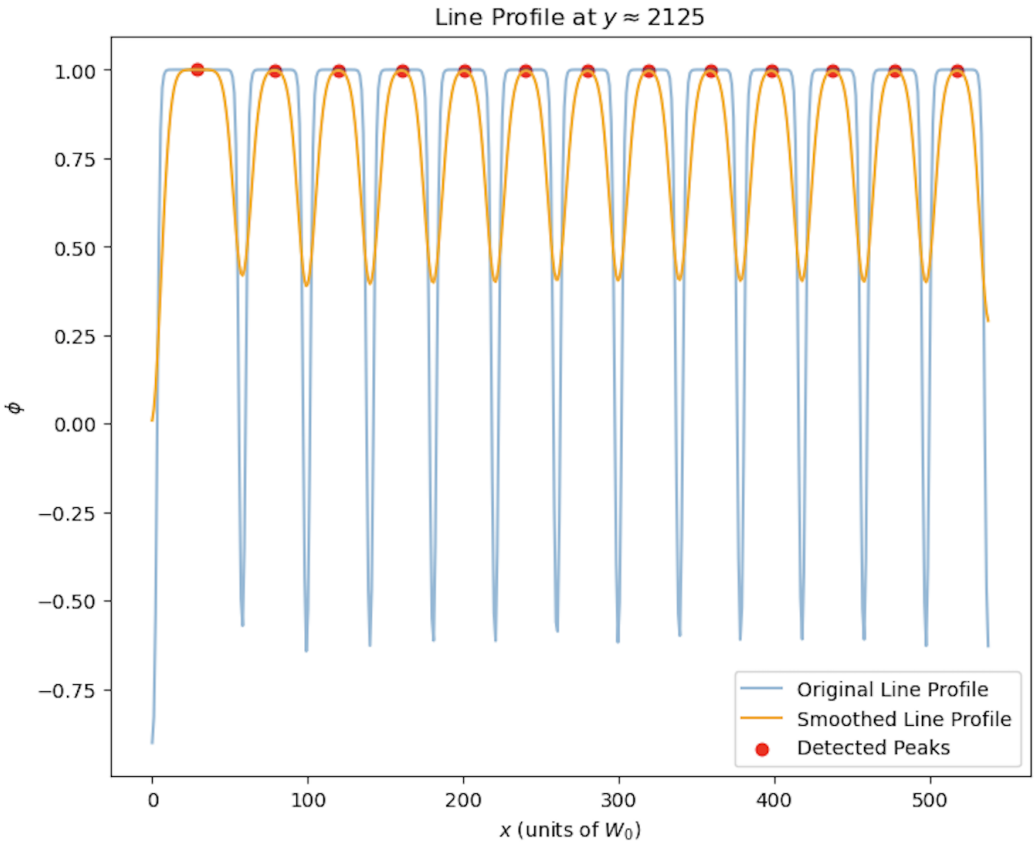
(b) Case a (c) Case b

Figure 5.1: Simulation results for Cases A and B are presented, with the top figure reproduced from the reference article [64] and the bottom figure generated from this study. It should be noted that the exact color bar from the reference article could not be replicated, as it was unavailable.

In this study, the primary dendrite arm spacing (PDAS) was measured. The approximate distance between the centers of the dendrite arms was determined by direct measurement. For Case A, the measured spacing was 92.8 units, while for Case B, it was 40 units. Applying a scaling factor of $W_0 = 10^{-8}$ m, these values correspond to 0.928 μm for Case A and 0.4 μm for Case B. The gap between primary dendrite arms was measured using Python by first exporting the data from ParaView as a CSV file. The CSV was then read into a Pandas DataFrame for processing. Using interpolation and line profile extraction, the gaps were identified by detecting peaks along a specific axis (see Fig. 5.2), allowing for precise calculation of dendrite arm spacing.



(a) phi field



(b) Detecting peaks along the red line in the top image.

Figure 5.2: In Case B, the dendrite intersects a line parallel to the y-axis, enabling the measurement of the gap between the arms. The calculated average dendrite arm spacing (PDAS) is approximately 40.67 units.

Chapter 6

Effect of Fluid Flow on Dendrite Growth in Additive Manufacturing

In the previous chapter, a baseline for the microstructural behavior of Ni–Nb alloys under varying cooling rates was established using the PFM. These simulations offered insights into dendrite growth and microstructure evolution in the absence of fluid dynamics. Building on these foundational results, this chapter introduces the influence of fluid flow on dendrite growth under AM conditions.

By incorporating fluid flow into the system, the study explores how convective forces impact the solidification process and alter the morphology of dendrite arms. The chapter focuses on the interaction between fluid flow and dendritic growth, providing a deeper understanding of microstructural evolution during AM processes.

The model presented in this chapter, along with the devised example and problem, is implemented and simulated, as illustrated in Figure 6.1. However, the model is not fully tuned and currently encounters convergence issues due to the complexity of the system, which involves solving numerous PDEs. These challenges result in slow and difficult-to-test simulations. The model was developed in FEniCS, and the corresponding repository is provided in Appendix A.

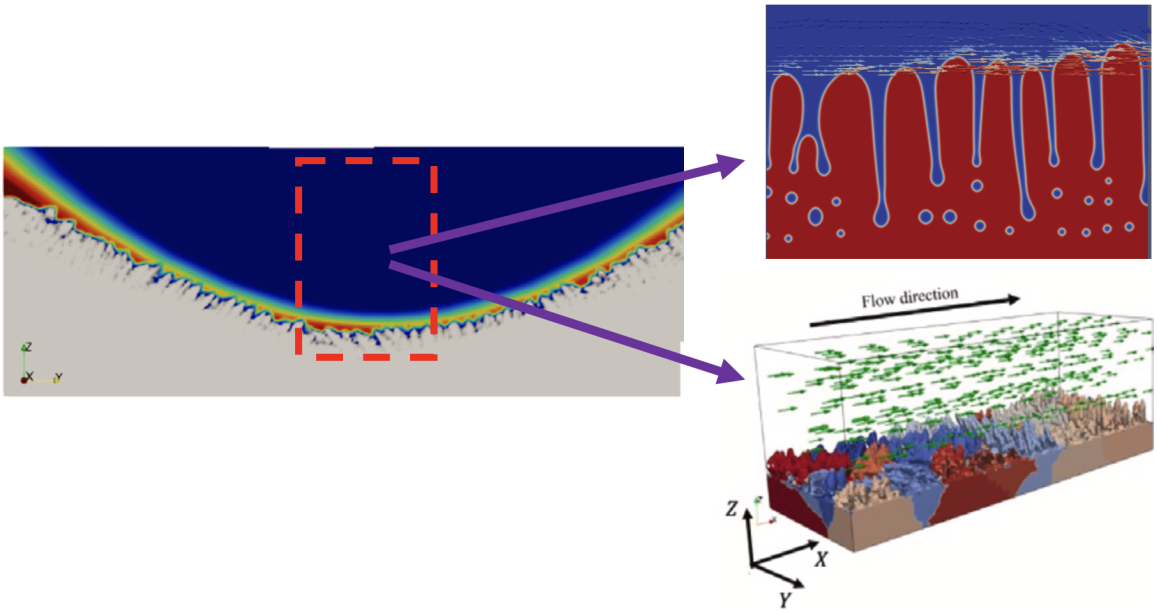


Figure 6.1: This figure illustrates the simulation domain and depicts the assumed inflow and outflow conditions in the dendritic region, designed to observe the behavior of fluid flow during solidification. Two of the figures, specifically those on the left-hand side and bottom right, are taken from [36].

6.1 Key Results and Challenges

The PDEs governing this simulation are identical to those benchmarked in Section 4.3, with the addition of a term to account for the thermal gradient. While reducing Δt and Δx sufficiently could, in theory, yield accurate results, this approach is computationally impractical and prohibitively expensive.

This simulation involves solving five coupled PDEs: two for the phase field and three for fluid flow (pressure and two velocity components). The coupling is inherently challenging, further complicated by the addition of a penalty term to enforce solid-phase behavior, as well as the nonlinear nature of the Navier-Stokes equations, which significantly impede convergence.

Direct Numerical Simulation (DNS) requires very small Δt and Δx demanding substantial computational resources. These constraints will limit the ability to thoroughly test, tune, and iterate on the model.

Table 6.1: Simulation Parameters for Phase-Field and Navier-Stokes Equations. Note: The scaled gravity is expressed as $g\frac{\tau_0^2}{W_0}$ to match the phase-field scaling with τ_0 and W_0. Similarly, the kinematic viscosity is expressed as $\frac{\mu}{\rho}\frac{\tau_0}{W_0^2}$.

Parameter	Symbol/Value	Description
Phase-Field Parameters		
Grid size	$\Delta y = 0.8$	Grid spacing.
Maximum refinement level	2	Maximum AMR level.
Domain size (x, y)	$N_x = 100, N_y = 100$	
Time step	$\Delta t = 13 \times 10^{-2}$	Time increment.
Solid region start	$y_{\text{solid}} = 2$	Initial solid height in bottom.
coefficients	$a_1 = 0.8839, a_2 = 0.6637$	Asymptotic parameters.
Interface width	$w_0 = 10^{-8}$	Interface width scaling.
Time scale	$\tau_0 = 2.30808 * 10^{-8}$	Time scale unit.
Capillary length	$d_0 = 8 \times 10^{-9}\, m$	Capillary length.
Thermal gradient	$G = 1 \times 10^{7}\,\text{K/m}$	Imposed thermal gradient.
Growth velocity	$V = 1 \times 10^{-1}\,\text{m/s}$	Solidification velocity.
Liquidus slope	$m_l = 10.5\,\text{K}\%^{-1}$	Slope of liquidus line.
Epsilon (anisotropy strength)	$\epsilon_4 = 0.03$	Anisotropy strength.
Equilibrium partition coefficient	$k_{eq} = 0.14$	Equilibrium solute partitioning.
Initial concentration	$c_0 = 5\%$	Initial solute concentration.
Navier-Stokes Parameters		
Gravitational acceleration	$g = -10\,\frac{m}{s^2}$	gravity.
Liquid density	$\rho_{\text{liquid}} = 7810\,\text{kg/m}^3$	Density of liquid phase.
Solid density	$\rho_{\text{solid}} = 8900\,\text{kg/m}^3$	Density of solid phase.
Dynamic viscosity	$\mu = 4.88 \times 10^{-3}\,\frac{\text{Pa}\cdot s}{\text{m}}$	Fluid viscosity.
Inlet velocity on Left-BC	$u = (0.01, 0.0)\,\frac{m}{s}$	Inlet velocity from left boundary.
Solver Parameters		
Phase-field solver tolerance	abs tol $= 10^{-4}$, rel tol $= 10^{-3}$	Solver tolerances for phase field.
Navier-Stokes solver tolerance	abs tol $= 10^{-4}$, rel tol $= 10^{-3}$	Solver tolerances for Navier-Stokes.
Linear solver (NS)	GMRES	Iterative linear solver.
Nonlinear solver (NS)	SNES	Nonlinear solver type.
Preconditioner (NS)	Hypre AMG	Preconditioner for NS.
Max iterations (NS)	100	Maximum iterations allowed.
Linear solver (PF)	GMRES	Linear solver for phase field.
Nonlinear solver (PF)	SNES	Nonlinear solver for phase field.
Preconditioner (PF)	Hypre AMG	Preconditioner for PF.
Max iterations (PF)	100	Maximum iterations allowed.

A model was developed to simulate the scenario depicted in Figure 6.1, with the implementation available in Appendix A. The parameters used in this simulation (Figure 6.3) are summarized in Tab. 6.1. Initial conditions for pressure and velocity were set to zero throughout the domain. For the phase field, a solid rectangular region with a height of 2 was initialized at the bottom. A perturbation was applied to the top of this solid region using random noise values ranging between -1 and 1. The perturbation height followed a sinusoidal function with a wavelength of $2\Delta y$ and an

amplitude of Δy. Furthermore, U, the dimensionless concentration, was initialized to -1 everywhere.

The boundary conditions for the Navier-Stokes equations included an inflow velocity at the left boundary, zero pressure at the right boundary, and a no-slip condition at the top boundary. For the phase field, Neumann boundary conditions were applied to all boundaries. To prevent concentration build-up at the left boundary, a Dirichlet boundary condition was applied for U.

The fully coupled model was employed for the simulation, with the assumption that the inlet flow began at the projection of the tallest dendrite in the domain (Figure 6.2). This assumption was introduced to address convergence issues that otherwise hindered the successful execution of the simulation.

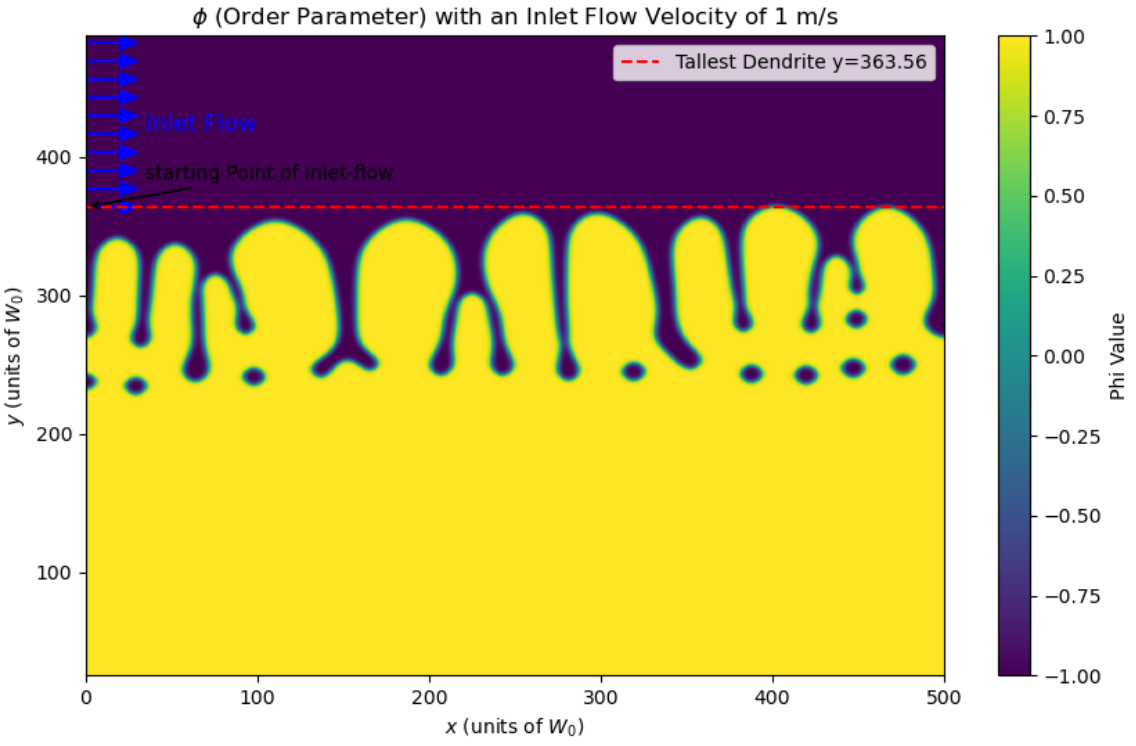


Figure 6.2: Visualization of the ϕ field and simulation domain for studying fluid flow impact on dendrite growth. The red dashed line marks the tallest dendrite projection, assumed as the inlet flow start to address convergence issues. Blue arrows show the inflow direction.

As shown in Figure 6.3, the simulation of this example was conducted under the conditions $G = 1 \times 10^7\,\mathrm{K/m}$ and $V = 3 \times 10^{-2}$. We compare the results with and without an inlet flow velocity of 1 m/s at a time step of 40,000 (equivalent to 0.11076 ms). The inlet flow velocity in this simulation is approximately 1 m/s, which is consistent with the typical maximum flow velocity observed in the melt pool during additive manufacturing processes, as reported in [65, 66, 67].The fluid flow inside the melt pool is primarily driven by temperature gradients, which induce thermocapillary (Marangoni) forces and buoyancy effects. While a more precise approach would involve solving the temperature field incorporating the thermal PDE—accounting for thermal convection and diffusion—the current focus is on establishing a base model. Incorporating a fully coupled temperature field with fluid flow should be addressed in future work to capture thermal-fluid interactions more accurately. In Figure 6.3, the directional growth of dendrites is illustrated, comparing morphologies with and without an inlet flow velocity. The figure captures the differences in dendrite structures at the same stage of growth, with the left panels showing results without flow and the right panels showing results with flow. The presence of fluid flow noticeably alters the dendrite morphology, reducing the spacing between primary dendrite arms and promoting more uniform growth. This highlights the influence of convective transport in refining the dendritic structure by redistributing solute more effectively near the solid-liquid interface.These results underscore the significant impact of fluid flow on the solidification process, emphasizing how convective forces interact with thermal and solutal gradients to shape the microstructure. The observed changes in spacing and morphology demonstrate the importance of considering fluid dynamics in simulations of solidification.

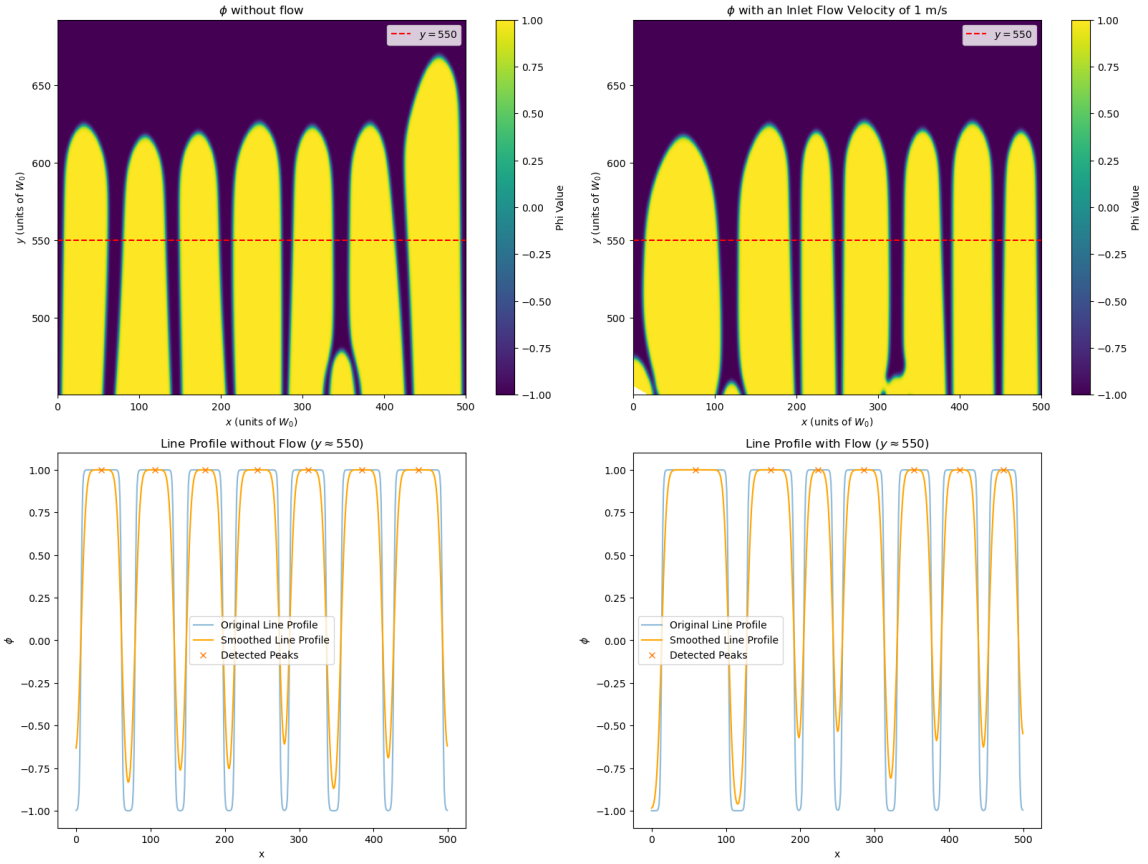


Figure 6.3: A comparison of dendrite arm morphologies with and without an inlet flow velocity of 1 m/s is shown. Both images at the top were captured at the same time step, 35,000*steps*. The left panels present the simulation results without flow, while the right panels include the inlet flow. The primary dendrite arm spacing (PDA) was measured along the red dashed line at $y = 550$. For the simulation with flow, the measured PDAS was 69.14 (physical units), whereas for the simulation without flow, it was 71.14 (W_0 units). The simulation parameters are $W_0 = 1 \times 10^{-8}$ m, $\Delta t = 1.3 \times 10^{-2}$, and $\tau_0 = 2.30808 \times 10^{-8}$ s.

Figure 6.4 illustrates the evolution of primary dendrite arm spacing (PDAS) over time for simulations conducted with and without fluid flow. The x-axis represents the time steps (scaled for clarity), while the y-axis shows the PDAS in units of W_0. The blue curve corresponds to the case without fluid flow, while the orange curve represents the case with fluid flow.

In the early stages of growth, the PDAS is significantly smaller in the presence of fluid flow compared to the case without flow. This highlights the impact of convective

solute transport in refining the dendritic structure by promoting more uniform solute redistribution. Over time, the PDAS increases in both cases; however, the rate of increase is notably slower in the presence of flow. This indicates that fluid flow helps maintain tighter dendritic spacing by actively reducing solute concentration gradients near the solid-liquid interface.

The convective forces generated by fluid flow enhance the transport of solute away from the solid-liquid interface, preventing solute pile-up and promoting smaller dendritic spacing. These findings are consistent with previous studies, such as [68], where in that work, their Fig. 7 demonstrates the reduction in primary dendrite arm spacing with increasing fluid flow intensity, and [69], where Fig. 11 in that publication illustrates this.

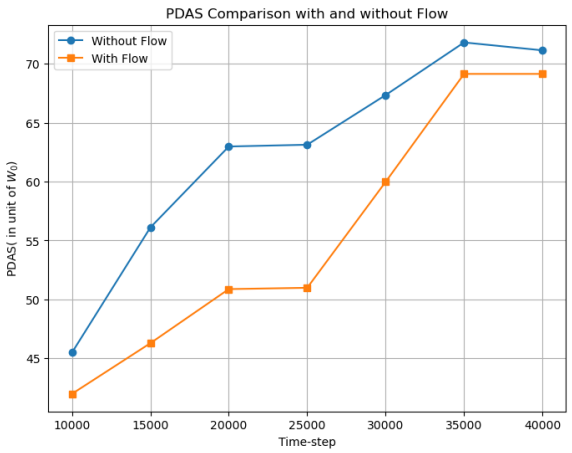


Figure 6.4: Comparison of Primary Dendrite Arm Spacing (PDAS) evolution over time with and without fluid flow. The blue curve represents the PDAS in the absence of fluid flow, while the orange curve shows the PDAS with fluid flow.

This highlights the impact of the fluid flow field on dendrite growth. The presence of fluid flow perturbs the initial conditions by convecting both the concentration and the order parameter, resulting in noticeable changes in the dendrite growth pattern, as shown in the Figure 6.4.

Chapter 7

Summary and Future Outlook

In this thesis, the Navier-Stokes module was implemented separately from the phase-field solidification module. Both modules were benchmarked individually and verified using a single dendrite convection example (refer to Section 4.3). Subsequently, the coupled code for solidification and fluid flow was benchmarked. The solidification model was further extended to simulate the laser deposition of a Ni-Nb alloy (Chapter 5). The final objective was to simulate the laser deposition process while incorporating fluid flow around the dendrite arms. However, significant challenges were encountered, particularly in achieving convergence and stability.

Fully coupling the fluid flow and solidification models presents inherent difficulties. As noted by [70], numerical instabilities—especially in convection-dominated problems—often arise due to the challenges associated with stabilizing advection terms. These issues make fully coupled approaches highly susceptible to instability and necessitate the use of advanced stabilization techniques to achieve convergence.

7.1 Future Work

Future work should explore implementing the phase-field model within established computational fluid dynamics (CFD) software like OpenFOAM, which uses the Finite

Volume Method (FVM), or MOOSE, which supports both the Finite Element Method (FEM) and the development of its own FVM-based turbulence models. Additionally, leveraging the capability of MOOSE[71] to integrate with NekRS through the Cardinal[72] framework offers potential advantages in handling complex fluid dynamics. Given the challenges associated with the CFD component of this coupling, utilizing the strengths of different software platforms and numerical methods beyond FEM could significantly enhance model stability and accuracy.

7.1.1 Flow Simulation Around Dendrite Geometry in OpenFOAM

To analyze the fluid flow behavior in the melt pool and evaluate the corresponding Reynolds number, an example simulation was devised based on the dendrite geometry generated in Chapter 6. The steady-state geometry, corresponding to a specific stage of dendritic growth, was extracted. Using Python, the contour data of this geometry was processed to generate an STL file. This STL file was then used to create the mesh for the simulation using OpenFOAM's 'snappyHexMeshDict', which handled the refinement and smoothing to capture the detailed dendrite structure.

The resulting 3D finite volume mesh was employed within OpenFOAM's turbulence model, specifically the k-ω SST model. Key parameters for the simulation included the flow velocity and kinematic viscosity, which was set to $1.5 \times 10^{-3} \left[\frac{m^2}{s}\right]$ in OpenFOAM . This setup allowed for an analysis of the flow behavior around the dendrite geometry, enabling an assessment of whether the flow was laminar or turbulent.

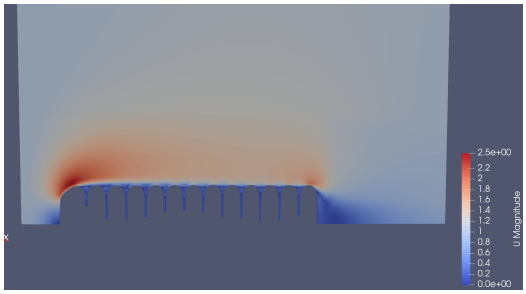


Figure 7.1: Velocity field magnitude in the simulated flow around the dendrite geometry.

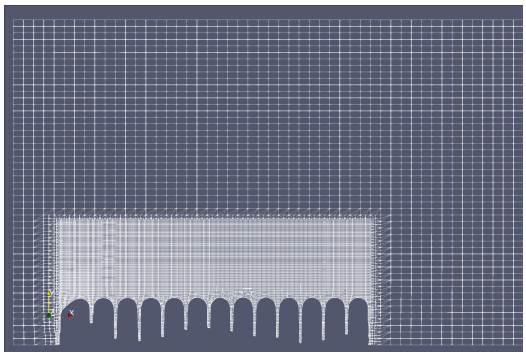


Figure 7.2: Mesh generation for the dendrite geometry using snappyHexMesh in OpenFOAM.

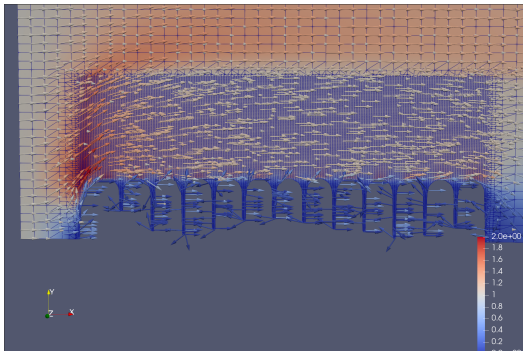


Figure 7.3: Combined mesh and velocity magnitude field.

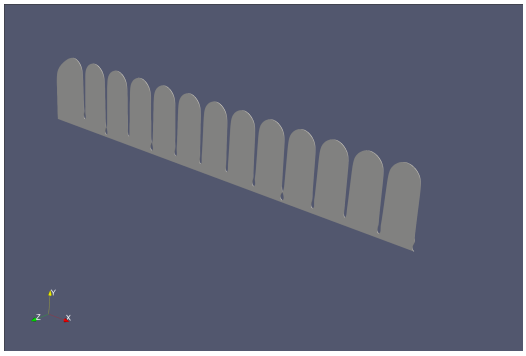


Figure 7.4: 3D representation of the dendrite geometry and mesh.

Figure 7.5: Flow and mesh visualization around the dendrite geometry were performed in OpenFOAM. The dendrite geometry data was exported from ParaView as a CSV file and subsequently processed in Python to generate the mesh.

7.2 Segregated Approach and MultiApps Framework in MOOSE

This thesis focuses on the development and coupling of multiphysics models using FEniCS. While FEniCS provides flexibility for custom implementations, MOOSE offers a modular framework for structured, coupled simulations through its MultiApps structure.

MOOSE facilitates the execution of distinct physics models in a parent-sub-application configuration, allowing each application to operate on separate meshes

with specific physics applied to different regions. Features such as adaptive mesh refinement and checkpointing significantly enhance computational efficiency within MOOSE.

The research aimed to couple a finite element method (FEM)-based phase-field model with a finite volume method (FVM)-based fluid flow model in MOOSE. Initial efforts involved testing FEM-based heat transfer coupled with FVM-based fluid flow using MOOSE's Transfer system and auxiliary kernels. This foundational example was instrumental in understanding the implementation process.

Subsequent work included recreating the "Assessment of Dendritic Growth Under Convection" example, with a focus on excluding solid regions from fluid flow simulations using penalty methods or variable viscosity approaches. However, challenges with MOOSE's FVM module required debugging and modifications, which constrained further testing.

Additionally, FEM-based Navier-Stokes equations were implemented, and phase-field and fluid flow models were coupled using MOOSE's MultiApps framework. The codes developed in MOOSE for this research are available in Appendix A.

Appendix A

Codes and Repositories

Table A.1: FEniCS Repositories

Index	Repository Name	Repository Link	Description
1	Isothermal Simulation	GitHub	Models isothermal phase-field simulations Figure 2.5.
2	Thermosolutal Simulation	GitHub	Simulates thermosolutal alloy solidification process Figure 2.4.
3	Benchmarked Marangoni Simulation	GitHub	Marangoni convection simulations Figure 3.2.
4	Coupled-Pure	GitHub	Coupled phase-field with fluid flow Figure 4.1.
5	AM condition solidification	GitHub	Simulation code based on [64] . see Figure 5.1
6	dendrite Inlet flow	GitHub	Inlet simulations for flow around dendrites. see Figures 6.2 and 6.1.

Table A.2: MOOSE Repositories

Index	Repository Name	Repository Link	Description
1	Coupled Heat-Fluid Simulation	GitHub	Simulates coupled heat transfer using FEM and fluid flow models using FVM. Segregated approach.
2	Coupled Simulation	GitHub	Demonstrates fully coupled physics simulations in MOOSE.
3	AM condition solidification	GitHub	Implementsbased [64] model for dendrite growth.
4	Fluid Component Development	GitHub	Fluid modeling for Navier-Stokes equations.
5	Coupled-Pure	GitHub	Coupled phase-field with fluid flow (Figure 4.1) in MOOSE.
6	dendrite Inlet flow	GitHub	Inlet simulations for flow around dendrites. see Figures 6.2 and 6.1.
7	Isothermal Model	GitHub	Models isothermal processes in MOOSE.

Table A.3: OpenFOAM Repositories

Index	Repository Name	Repository Link	Description
1	Isothermal Simulation	GitHub	Simulates isothermal processes using OpenFOAM (FVM) see C.2.
2	Dendrite Geometry Flow Visualization	GitHub	Visualizes flow and mesh around dendrite geometry using OpenFOAM and Turbulent model. 7.4

Appendix B

Fully Coupled Code Implementation

Introduction

In this section, we briefly describe the code implementation of this project.For more information see this repository [**pashaei2024master**].

B.1 Phase-Field Problem Class

This class defines the phase-field problem using FEniCS, encapsulating methods for model setup, initialization, and solving. The structure is as follows:

Inputs:

- Parameters dictionary for constants in the PDE.
- Mesh to define the domain of the problem.

Outputs:

- Solved phase-field variables, including order parameters and concentration.

Class Methods:

1. `__init__`: Initializes the class, setting up the mesh, function space, and solution fields.

2. `define_function_space`: Creates function spaces for solution vectors, using mixed function spaces for order parameter and concentration fields.

3. `define_dependent_variables`: Defines the dependent variables and performs intermediate calculations, such as computing gradients for anisotropic effects.

4. `formulate_weak_form`: Constructs the weak form of the PDEs, covering time evolution, diffusion, and source terms.

5. `define_solver`: Configures the nonlinear problem and solver, including solver parameters like tolerances and preconditioners.

6. `set_initial_conditions`: Establishes the initial conditions for the problem, using a custom expression for initializing order parameter and concentration fields.

7. `solve_problem`: Executes the solver to obtain the solution for the current time step and updates the solution vectors.

```
1 import fenics as fe
2 import numpy as np
3
4 class ClassPF:
5
```

```
    def __init__(self, mesh, params, nsproblem= None, old_sv_=None,
old_sv=None):

        self.mesh = mesh
        self.params = params
        self.dt = params["dt"]
        self.w0 = params['w0']
        self.ep4 = params['ep4']
        self.X = fe.SpatialCoordinate(self.mesh)
        self.Y = self.X[1]
        self.G = params['G']
        self.V = params['V']
        self.ml = params['ml']
        self.c0 = params['c0']
        self.keq = params['keq']
        self.lamda = params['lamda']
        self.ds = params['ds']
        self.dl = params['dl']
        self.Wscale = params['Wscale']
        self.Tscale = params['Tauscale']
        self.T = params['Time']
        reltol = params['reltol']
        abstol = params['abstol']
        linearsolverpf = params['linearsolverpf']
        nonlinearsolverpf = params['nonlinearsolverpf']
        preconditionerpf = params['preconditionerpf']
        maximumiterationspf = params['maximumiterationspf']
        self.solver_parameters = {'nonlinear_solver':
nonlinearsolverpf,
            'snes_solver': {'linear_solver': linearsolverpf,
            'report': False,"preconditioner": preconditionerpf,
            'error_on_nonconvergence': False,'absolute_tolerance':
abstol,
```

```
36          'relative_tolerance': reltol,'maximum_iterations':
    maximumiterationspf,}}
37
38      self.dy = params['dy']
39      self.y_solid = params['y_solid']
40      self.old_sv_ = old_sv_
41      self.old_sv = old_sv
42      self.nsproblem = nsproblem
43      if self.nsproblem is not None:
44          self.u_npf = nsproblem.sv_.split(deepcopy=True)[0]
45      self.func_space()
46      self.inco()
47      self.depvar()
48      self.form()
49      self.defsol()
50
51  def func_space(self, degree=2):
52
53      P1 = fe.FiniteElement("Lagrange", self.mesh.ufl_cell(), 1)
54      P2 = fe.FiniteElement("Lagrange", self.mesh.ufl_cell(), 1)
55      self.Vs = fe.VectorFunctionSpace(self.mesh, 'P', degree)
56      self.u_n = fe.Function(self.Vs)
57      element = fe.MixedElement([P1, P2])
58      self.fs = fe.FunctionSpace(self.mesh, element)
59      self.v_phi, self.v_c = fe.TestFunctions(self.fs)
60      self.sv = fe.Function(self.fs)
61      self.sv_ = fe.Function(self.fs)
62      self.phi, self.c = fe.split(self.sv)
63      self.phi_, self.c_ = fe.split(self.sv_)
64      self.spacepf, _ = self.fs.sub(0).collapse(collapsed_dofs=
    True)
65      self.spacec, _ = self.fs.sub(1).collapse(collapsed_dofs=True
    )
```

```
66          if self.nsproblem is not None:
67              fe.LagrangeInterpolator.interpolate(self.u_n, self.u_npf
    )
68
69      def depvar(self):
70
71          self.tol= fe.sqrt(fe.DOLFIN_EPS)
72          grad_phi = fe.grad(self.phi_)
73          self.mgphi = fe.inner(grad_phi, grad_phi)
74          dpx = fe.Dx(self.phi_, 0)
75          dpy = fe.Dx(self.phi_, 1)
76          dpx = fe.variable(dpx)
77          dpy = fe.variable(dpy)
78          # Normalized derivatives
79          nmx = -dpx / fe.sqrt(self.mgphi)
80          nmy = -dpy / fe.sqrt(self.mgphi)
81          norm_phi_4 = nmx**4 + nmy**4
82          an = fe.conditional(
83              fe.lt(fe.sqrt(self.mgphi), self.tol),
84              fe.Constant(1-3*self.ep4),
85              1-3*self.ep4+ 4*self.ep4*norm_phi_4)
86          self.wn = self.w0 * an
87          self.dwnx = fe.conditional(fe.lt(fe.sqrt(self.mgphi), self.
    tol), 0, fe.diff(self.wn, dpx))
88          self.dwny = fe.conditional(fe.lt(fe.sqrt(self.mgphi), self.
    tol), 0, fe.diff(self.wn, dpy))
89
90      def form(self):
91
92          self.taun = (self.wn/self.w0)**2
93          opk, omk= 1+self.keq, 1-self.keq
94          if self.nsproblem is not None:
95              term1ad = - fe.inner((self.taun) * fe.dot(self.u_n, fe.
```

```
grad(self.phi)), self.v_phi)
96            grad_phi = fe.grad(self.phi)
97            # Advection Term LHS goes to RHS(Negative): V  {[(1+k-(1-
k)  )/2]  U  -[(1+(1-k)U)/2]     }:
98            term10_1 = fe.dot(self.u_n,(opk-omk*self.phi)/2*fe.grad(
self.c)) # V  (1+k-(1-k)  )/2]  U
99            term10_2 =  fe.dot(self.u_n,-(1+omk*self.c)/2*grad_phi)
# - V  [(1+(1-k)U)/2]
100           term2ad = - fe.inner(term10_1+term10_2, self.v_c)
101       else:
102           term1ad = fe.Constant(0)*self.v_phi
103           term2ad = fe.Constant(0)*self.v_c
104       # first equation
105       term0 = (self.G* self.Wscale)*(self.Y-self.V*(self.T*self.
Tscale/self.Wscale))/(self.ml* self.c0/self.keq*(1-self.keq))
106       term4in = self.mgphi*self.wn*self.dwnx
107       term5in = self.mgphi*self.wn*self.dwny
108       term4 = -fe.inner(term4in, self.v_phi.dx(0))
109       term5 = -fe.inner(term5in, self.v_phi.dx(1))
110       term3 = -(self.wn**2*fe.inner(fe.grad(self.phi),fe.grad(self
.v_phi)))
111       term2 = fe.inner((self.phi - self.phi**3)-self.lamda*(self.c
 + term0)*(1-self.phi**2)**2, self.v_phi)
112       term1 = -fe.inner((self.taun) * (self.phi-self.phi_) / self.
dt, self.v_phi)
113       self.eq1 = term1+term2+term3+term4+term5+term1ad
114       self.eq1 = self.eq1*fe.dx
115       # second equation
116       d = self.ds*(1+self.phi)/2+self.dl*(1-self.phi)/2
117       dphidt = (self.phi-self.phi_)/self.dt
118       term6 = -fe.inner(((opk) / 2 - (omk) * self.phi / 2) * (self
.c - self.c_) / self.dt, self.v_c)
119       term7 = -fe.inner(d * (1 - self.phi) / 2 * fe.grad(self.c),
```

```
fe.grad(self.v_c))
        term9 = (1 + (omk) * self.c) * dphidt / 2 * self.v_c
        self.eq2 = term6+term7+term9+term2ad
        self.eq2 = self.eq2*fe.dx

    def defsol(self):

        L = self.eq1 + self.eq2
        J = fe.derivative(L, self.sv)
        problem = fe.NonlinearVariationalProblem(L, self.sv, J=J)
        self.solverpf = fe.NonlinearVariationalSolver(problem)
        self.solverpf.parameters.update(self.solver_parameters)

    def solve(self):

        self.solverpf.solve()
        self.sv_.vector()[:]= self.sv.vector()

    def inco(self):

        class InitialConditions(fe.UserExpression):
            def __init__(self, dy, y_solid, **kwargs):
                super().__init__(**kwargs)
                self.dy = dy
                self.y_solid = y_solid
            def eval(self, values, x):
                xp = x[0]
                yp = x[1]
                perturbation_amplitude = 1*self.dy
                perturbation_wavelength = 4*self.dy
                perturbation = perturbation_amplitude*np.sin(2 *np.
pi*xp/perturbation_wavelength)
                if yp < self.y_solid - perturbation_amplitude :
```

```
151                     values[0] = 1
152                     values[1] = -1
153                 elif self.y_solid - perturbation_amplitude  <= yp <=
     self.y_solid + perturbation_amplitude:
154                     values[0] = perturbation
155                     values[1] = -1
156                 else:  # liquid
157                     values[0] = -1
158                     values[1] = -1
159             def value_shape(self):
160                 return (2,)
161
162         if self.old_sv_ is not None:
163
164             fe.LagrangeInterpolator.interpolate(self.sv_, self.
    old_sv_)
165             fe.LagrangeInterpolator.interpolate(self.sv, self.old_sv
    )
166
167         else:
168             self.sv_.interpolate(InitialConditions(self.dy, self.
    y_solid, degree=2))
169             self.sv.interpolate(InitialConditions(self.dy, self.
    y_solid, degree=2))
```

Listing B.1: Phase Field Class

B.2 Navier-Stokes Class

The Navier-Stokes (NS) class defines and solves the incompressible Navier-Stokes equations for solid-liquid systems using FEniCS. It supports fluid flow coupled with solidification by interacting with a phase-field (PF) problem.

Key Methods

- **Initialization:** Sets up physical parameters, time step, mesh, and solver parameters.
- **Function Spaces:** Defines velocity-pressure function spaces with Lagrange elements.
- **Formulation:** Constructs the weak form of the Navier-Stokes equations, adding penalization to ensure zero velocity in solid regions.
- **Boundary Conditions:** Implements inflow, outflow, and wall boundary conditions using velocity profiles.
- **Solver:** Nonlinear solver is set up for solving the system iteratively.
- **Viscosity Handling:** Computes viscosity as a weighted combination based on solid or liquid phase.

This class enables efficient simulation of fluid dynamics within a system undergoing phase transitions between solid and liquid states.

```
1  import fenics as fe
2  import numpy as np
3
4  class NS:
5
6      def __init__(self, mesh, parameters_dict, nsproblem=None ,
       pfproblem=None):
7
8          self.mesh = mesh
9          self.nsproblem = nsproblem
10         self.pfproblem = pfproblem
11         self.parameters_dict = parameters_dict
```

```
12        self.dt = fe.Constant(parameters_dict["dt"]) # Time step
13        self.rho = fe.Constant(parameters_dict["rho_solid"])
14        self.rho_solid = fe.Constant(parameters_dict["rho_solid"])
15        self.rho_liquid = fe.Constant(parameters_dict["rho_liquid"])
16        self.viscosity_solid = fe.Constant(parameters_dict["
    viscosity_solid"])
17        self.viscosity_liquid = fe.Constant(parameters_dict["
    viscosity_liquid"])
18        self.wscale = parameters_dict["Wscale"]
19        self.wscale = parameters_dict["Wscale"]
20        self.tscale = parameters_dict["Tauscale"]
21        self.kinsolid =(self.viscosity_solid/self.rho_solid)*(self.
    tscale/self.wscale**2)
22        self.kinliq =(self.viscosity_liquid/self.rho_liquid)*(self.
    tscale/self.wscale**2)
23        self.mu = self.kinliq
24        self.pfproblem = pfproblem
25        self.solver_parameters = {
26            'nonlinear_solver': self.parameters_dict["
    nonlinearsolverns"],
27            'snes_solver': {
28                'linear_solver': self.parameters_dict["
    linearsolverns"],
29                'report': False,
30                "preconditioner": self.parameters_dict["
    preconditionerns"],
31                'error_on_nonconvergence': False,
32                'absolute_tolerance': self.parameters_dict["
    abs_tol_ns"],
33                'relative_tolerance': self.parameters_dict["
    rel_tol_ns"],
34                'maximum_iterations': self.parameters_dict["
    maximumiterationsns"],
```

```
35              }
36          }
37          if self.nsproblem is not None:
38              self.old_sv_ = self.nsproblem.sv_
39              self.old_sv = self.nsproblem.sv
40          if self.pfproblem is not None:
41              self.phi_, self.c_ = self.pfproblem.sv_.split(deepcopy=
    True) # use self.phi_interp
42          # Initialize
43          self.func_space()
44          self.form()
45          self.BC()
46          self.InitialC()
47          self.solver()
48
49      def func_space(self, degree=2):
50
51          P1 = fe.VectorElement("Lagrange", self.mesh.ufl_cell(), 2)
52          P2 = fe.FiniteElement("Lagrange", self.mesh.ufl_cell(), 1)
53          EL = fe.MixedElement([P1, P2])
54          self.fs = fe.FunctionSpace(self.mesh, EL)
55          self.v, self.q = fe.TestFunctions(self.fs)
56          self.sv = fe.Function(self.fs)
57          self.sv_ = fe.Function(self.fs)
58          self.u, self.p = fe.split(self.sv)
59          self.u_, self.p_ = fe.split(self.sv_)
60          self.space_u, _ = self.fs.sub(0).collapse(collapsed_dofs=
    True)
61          self.space_p, _ = self.fs.sub(1).collapse(collapsed_dofs=
    True)
62          self.phi_interp = fe.Function(self.space_p)
63          if self.pfproblem is not None:
64              fe.LagrangeInterpolator.interpolate(self.phi_interp,
```

```
    self.phi_)
65
66      def form(self):
67
68
69          beta = 1e6  # Adjust as needed
70          epsilon = 0.05
71          H = self.smooth_heaviside(self.phi_interp, epsilon)
72          penalization = beta * H * fe.inner(self.u, self.v) * fe.dx
73          Fp = fe.inner(fe.div(self.u)/self.dt, self.q)
74          Fu = (
75              fe.inner((self.u - self.u_)/self.dt, self.v)
76              + fe.inner(fe.dot(self.u,fe.grad(self.u)),self.v)
77              + self.mu*fe.inner(self.sigma(self.u, self.p), self.
    epsilon(self.v))
78              + fe.inner(penalization_term, self.v)
79          )
80          self.F = (Fp + Fu)*fe.dx
81
82      def epsilon(self,u):
83          return 0.5*(fe.grad(u)+fe.grad(u).T)
84
85      def sigma(self,u, p):
86          return 2*self.epsilon(u) - p*fe.Identity(len(u))
87
88      def BC(self):
89
90          Nx = self.parameters_dict.get("Nx")
91          Ny = self.parameters_dict.get("Ny")
92          max_y = self.parameters_dict.get("y_solid")
93          velx = self.parameters_dict.get("velx")# m/s
94          velx = velx*self.tscale/self.wscale
95          # # Parabolic inflow profile: velocity is zero below max_y,
```

```
    parabolic above max_y
96          # inflow_profile = fe.Expression(('(x[1] >= max_y) ? velx *
    (1 - pow((x[1] - max_y) / (Ny - max_y), 2)) : 0.0', '0.0'),
97          #                               velx=velx, Ny=Ny, max_y=max_y,
    degree=2)
98
99          inflow_profile = (fe.Expression((
100             'velx * (tanh((x[1] - max_y) / eps) + 1.0) / 2.0 * (1.0
    - pow((x[1] - max_y) / (Ny - max_y), 2))',
101             '0.0'),
102             velx=velx, Ny=Ny, max_y=max_y, eps=0.05, degree=2))
103
104         inflow = 'near(x[0],0)'
105         outflow = f'near(x[0],{Nx})'
106         walls = f'near(x[1],0) || near(x[1],{Ny})'
107         bcu_inflow = fe.DirichletBC(self.fs.sub(0), inflow_profile,
    inflow)
108         bcp_outflow = fe.DirichletBC(self.fs.sub(1), fe.Constant(0),
     outflow)
109         bc_walls = fe.DirichletBC(self.fs.sub(0), fe.Constant((0.0,
    0.0)), walls)
110         self.Bc = [bcu_inflow, bcp_outflow, bc_walls]
111
112     def solver(self):
113
114         J= fe.derivative(self.F, self.sv)
115         self.problem= fe.NonlinearVariationalProblem(self.F, self.sv
    , self.Bc, J)
116         self.solver = fe.NonlinearVariationalSolver(self.problem)
117         self.solver.parameters.update(self.solver_parameters)
118
119     def InitialC(self):
120
```

```
121         class InitialConditions_ns(fe.UserExpression):
122 
123             def __init__(self, params, **kwargs):
124                 super().__init__(**kwargs)
125 
126             def eval(self, values, x):
127                 values[0] = 0.0
128                 values[1] = 0.0
129                 values[2] = 0.0
130 
131             def value_shape(self):
132                 return (3,)
133 
134         if self.nsproblem is not None:
135             fe.LagrangeInterpolator.interpolate(self.sv_, self.
    old_sv_)
136             fe.LagrangeInterpolator.interpolate(self.sv, self.old_sv
    )
137         else:
138             self.sv_.interpolate(InitialConditions_ns(self.
    parameters_dict, degree=2))
139             self.sv.interpolate(InitialConditions_ns(self.
    parameters_dict, degree=2))
140 
141     def solve(self):
142 
143         self.solver.solve()
144         self.sv_.vector()[:]= self.sv.vector()
145 
146     def smooth_step(self, phi):
147 
148         return 3 * phi**2 - 2 * phi**3
149 
```

```
150     def viscosity(self):
151
152         return (
153             (self.kinsolid) * self.smooth_step(self.phi_interp)
154             + (self.kinliq) * (1 - self.smooth_step(self.phi_interp)
    ))
```

Listing B.2: Navier-Stockes Class

B.3 Adaptive Mesh Refinement

The `meshrefiner` class performs adaptive mesh refinement (AMR) based on the gradient of the phase field, ϕ. The refinement process identifies areas where the gradient exceeds a defined threshold, indicating significant changes in ϕ, and refines the mesh accordingly. This implementation is designed to handle parallel computations effectively. The refinement steps are as follows:

1. **value_coor_dof**: This function gathers the gradient values of ϕ at the degrees of freedom (DOFs) across all processors. It collects this information from each process and then shares it with all processors to ensure consistent refinement decisions in parallel execution.

2. **coordinates_of_int**: Identifies the mesh coordinates corresponding to regions with high ϕ gradients or interface, which are marked as candidates for refinement.

3. **mark_mesh**: This function identifies the cells in the mesh that require refinement based on the coordinates where the gradient of ϕ exceeds the threshold. It uses a mesh function to flag these cells for refinement.

4. **refine_to_min**: Applies the refinement process to the marked cells and generates a new refined mesh.

5. **refine_mesh**: This function repeatedly refines the mesh, up to a specified maximum refinement level. The mesh is adapted in areas of high gradient until the required resolution is reached.The refinement process begins from a predefined coarse mesh and iteratively refines the flagged regions until the specified resolution or maximum refinement level is reached. This ensures that the mesh adapts dynamically to capture fine details in areas of high ϕ gradient while maintaining efficiency in regions with less variation.

6. **initialize**: This function coordinates the entire process by first gathering the gradient values, identifying the areas to refine, and then executing the mesh refinement process.

The class ensures efficient parallel computation by gathering and sharing the necessary refinement data across all processors, allowing the mesh to be refined consistently across different computational domains. This class, while not guaranteed to offer maximum efficiency due to being developed by a less experienced author, is designed to function effectively in adaptively refining the mesh. Although there may be room for optimization, it fulfills the intended purpose of refining the mesh based on the gradient of the field and will work as expected to capture important features in the simulation.

```
import fenics as fe
import numpy as np


class meshrefiner:
    def __init__(self, params, pfproblem, comm):
        self.params= params
        self.dy= params['dy']
        self.max_level= params['max_level']
        self.interface_threshold_gradient= params["
```

```
interface_threshold_gradient"]
        self.comm = comm
        self.pfproblem= pfproblem
        self.spacepf= self.pfproblem.spacepf
        self.sv_= self.pfproblem.sv_
        self.mesh_coarse= params['mesh_coarse']

    def value_coor_dof(self):

        (phi_, u_) = fe.split(self.sv_)
        coordinates_of_all = self.spacepf.tabulate_dof_coordinates()
        grad_phi = fe.project(fe.sqrt(fe.dot(fe.grad(phi_), fe.grad(
phi_))), self.spacepf)
        phi_value_on_dof = grad_phi.vector().get_local()
        all_Val_dof = self.comm.gather(phi_value_on_dof, root=0)
        all_point = self.comm.gather(coordinates_of_all, root=0)
        # Broadcast the data to all processors
        all_point = self.comm.bcast(all_point, root=0)
        all_Val_dof = self.comm.bcast(all_Val_dof, root=0)
        # Combine the data from all processors
        all_Val_dof_1 = [val for sublist in all_Val_dof for val in
sublist]
        all_point_1 = [point for sublist in all_point for point in
sublist]
        self.dofcoor = np.array(all_point_1)
        self.valdof = np.array(all_Val_dof_1)

    def coordinates_of_int(self):

        high_gradient_indices = np.where(self.valdof > self.
interface_threshold_gradient)[0]
        self.listcoorint = self.dofcoor[high_gradient_indices]

```

```
39     def mark_mesh(self, coarse_mesh_it):
40
41         mf = fe.MeshFunction("bool", coarse_mesh_it, coarse_mesh_it.
    topology().dim(), False)
42         len_mf = len(mf)
43         Cell_Id_List = []
44         tree = coarse_mesh_it.bounding_box_tree()
45         for Cr in self.listcoorint:
46             cell_id = tree.compute_first_entity_collision(fe.Point(
    Cr))
47             if cell_id != 4294967295 and 0 <= cell_id < len_mf:
48                 Cell_Id_List.append(cell_id)
49
50         Cell_Id_List = np.unique(np.array(Cell_Id_List, dtype=int))
51         mf.array()[Cell_Id_List] = True
52         return mf
53
54     def refine_to_min(self, coarse_mesh_it):
55
56         mf = self.mark_mesh(coarse_mesh_it)
57         rfmesh = fe.refine(coarse_mesh_it, mf, redistribute=True)
58         return rfmesh
59
60     def refine_mesh(self):
61
62         coarse_mesh_it = self.mesh_coarse
63         for res in range(self.max_level):
64             self.mesh_new = self.refine_to_min(coarse_mesh_it)
65             coarse_mesh_it = self.mesh_new
66
67         self.mesh_info = {
68             'n_cells': fe.MPI.sum(self.comm, self.mesh_new.num_cells
    ()),
```

```
69              'hmin': fe.MPI.min(self.comm, self.mesh_new.hmin()),
70              'hmax': fe.MPI.max(self.comm, self.mesh_new.hmax()),
71              'dx_min': fe.MPI.min(self.comm, self.mesh_new.hmin()) /
    fe.sqrt(2),
72              'dx_max': fe.MPI.max(self.comm, self.mesh_new.hmax()) /
    fe.sqrt(2),}
73
74     def initialize(self):
75
76         self.value_coor_dof()
77         self.coordinates_of_int()
78         self.refine_mesh()
```

Listing B.3: AMR Class

Appendix C

Phase-Field Karma Model Code Implementation

Introduction

This appendix contains the source code for the implementation of the Phase-Field Karma model using OpenFOAM and MOOSE. The code provided here highlights critical components of the solver setup, numerical schemes, and custom modifications necessary for solving the equations detailed in the main body of this thesis.

The objective of implementing the Phase-Field Karma model in both OpenFOAM and MOOSE was to leverage the built-in CFD capabilities of these powerful software frameworks. By integrating the phase-field model with their existing computational fluid dynamics tools, future work can directly utilize the extensive CFD functionality for more advanced simulations involving fluid-structure interactions and solidification processes.

C.1 MOOSE Implementation

To solve equations 2.9 and 2.10, the following MOOSE code was developed by refining and simplifying the existing Kobayashi kernels[**moose˙kobayashi˙kernel**] in MOOSE, which were overly complex. This approach streamlines the implementation, making it more efficient and easier to use. you can find the codes in this repository [**moose˙isothermal**].

Listing C.1: MOOSE Input File for Isothermal Simulation

```
[Mesh]
    type = GeneratedMesh
    dim = 2
    nx = 25
    ny = 25
    xmin = 0.0
    xmax = 400.0
    ymin = 0.0
    ymax = 400.0
[]


[Variables]
    # Variables for the two equations
    [./phi]
        family = LAGRANGE
        order = FIRST
    [../]

    [./u]
        family = LAGRANGE
        order = FIRST
    [../]
```

```
[]

[Kernels]
    #phi
    [./phi_time_derivative]
        type = ADTimeop
        variable = phi
    [../]
    [./phi_terms1]
        type = ADkarma1
        variable = phi
    [../]
    [./phi_src]
        type = ADsrc
        variable = phi
        UH = u
    [../]
    # U
    [./U_time_derivative]
        type = E1
        variable = u
        op = phi
    [../]
    [./Div_term]
        type = E2
        variable = u
        op = phi
    [../]
    [./u_term_3]
        type = E3
        variable = u
        op = phi

```

```
    [../]
[]
[ICs]
    [./phiIC]
        type = SmoothCircleIC
        variable = phi
        x1 = 0.0
        y1 = 0.0
        radius = 8
        outvalue = -1 # liquid
        invalue = 1 # solid
    [../]
    [./uIC]
        type = SmoothCircleIC
        variable = u
        x1 = 0.0
        y1 = 0.0
        radius = 8
        outvalue = -0.55 # liquid
        invalue = -0.55 # solid
    [../]
[]
[Materials]
    [./Theta]
        type = ADtheta
```

```
90          op = phi
91      [../]
92  []
93
94  [Adaptivity]
95      initial_steps = 5
96      initial_marker = refine_region
97      max_h_level = 5
98      marker = err_phi
99      interval = 10
100
101     [./Markers]
102         [./err_phi]
103             type = ErrorFractionMarker
104             coarsen = 0.3
105             refine = 0.95
106             indicator = ind_phi
107         [../]
108         [./refine_region]
109                 type = BoxMarker
110                 bottom_left = '0 0 0'
111                 top_right = '21 21 0'
112                 inside = refine
113                 outside = do_nothing
114         [../]
115     [../]
116     [./Indicators]
117         [./ind_phi]
118             type = GradientJumpIndicator
119             variable = phi
120         [../]
121     [../]
122 []
```

```
123
124 [Preconditioning]
125     [./advanced_precond]
126       type = SMP
127       full = false
128     [../]
129   []
130
131 [Executioner]
132     type = Transient
133     solve_type = PJFNK
134     nl_abs_tol = 1e-6
135     nl_rel_tol = 1e-5
136     l_max_its = 100
137     nl_max_its = 50
138     petsc_options_iname = '-ksp_type -pc_type -pc_hypre_type -
    ksp_gmres_restart -pc_hypre_boomeramg_strong_threshold'
139     petsc_options_value = 'gmres hypre boomeramg 30 0.7'
140     dt = 0.018
141     end_time = 75000
142
143 []
144
145
146 [Outputs]
147     [./exodus]
148       type = Exodus
149       interval = 100
150       file_base = simulation
151       execute_on = 'initial timestep_end'
152     [../]
153     [./checkpoint]
154             type = Checkpoint
```

```
            time_step_interval = 100
            num_files = 2
    [../]
  []
```

For the anisotropy component, we utilize the formulation $A(\theta) = 1 + \epsilon\cos(4\theta)$ as part of equation 2.9. The derivation of this expression can be found in the referenced literature [64], and its implementation is straightforward.

The first term we need is $A(\theta) = 1+\epsilon\cos(4\theta)$, where θ is defined as $\theta = \arctan\left(\frac{\partial_y \phi}{\partial_x \phi}\right)$. To compute θ as a material property in MOOSE, the following file was created, using the order parameter (‘op‘) and its gradient.

Listing C.2: MOOSE kernel for defining theta

```
#include "ADtheta.h"
#include "MooseMesh.h"
#include "MathUtils.h"

registerMooseObject("diffApp", ADtheta);

InputParameters
ADtheta::validParams()
{
    InputParameters params = Material::validParams();
    params.addClassDescription("2D interfacial anisotropy");
    params.addParam<Real>(
        "epsilon", 0.02, "Strength of the anisotropy");
    params.addParam<Real>(
        "w0", 1.0, "Interface width");
    params.addRequiredCoupledVar("op", "Order parameter defining the
    solid phase");

  return params;
```

```
}

ADtheta::ADtheta(const InputParameters & parameters)
  : Material(parameters),
    _epsilon(getParam<Real>("epsilon")),
    _w0(getParam<Real>("w0")),
    _eps(declareADProperty<Real>("eps")),
    _deps(declareADProperty<Real>("deps")),
    _op(adCoupledValue("op")),
    _grad_op(adCoupledGradient("op"))
{
  // this currently only works in 2D simulations
  if (_mesh.dimension() != 2)
    mooseError("ADInterfaceOrientationMaterial requires a two-
    dimensional mesh.");
}


void
ADtheta::computeQpProperties()
{
    const ADReal grad_x_phi = _grad_op[_qp](0);  //   x
    const ADReal grad_y_phi = _grad_op[_qp](1);  //   y

    // Compute the magnitude of the gradient
    const ADReal grad_norm = std::sqrt(grad_x_phi * grad_x_phi +
    grad_y_phi * grad_y_phi);

    // Initialize theta
    ADReal theta = 0.0;

    // Avoid undefined atan2 when gradient is zero
    if (grad_norm > 1e-14) // Use a small tolerance to avoid
```

```
    division by zero
50   {
51       theta = std::atan2(grad_y_phi, grad_x_phi);
52   }
53
54   // Compute anisotropy functions
55   _eps[_qp] = _w0 * (1.0 + _epsilon * std::cos(4.0 * theta));
56   _deps[_qp] = _w0 * (- _epsilon * 4.0 * std::sin(4.0 * theta));
57 }
```

It is now possible to define the terms in equation 2.9. The 'ADTimeOp.C' kernel is responsible for implementing the term $\tau_0 A(\theta)^2 \frac{\partial \phi}{\partial t}$.

Listing C.3: Kernel for Time Derivative of Order Parameter

```
1  #include "ADTimeop.h"
2
3  registerMooseObject("diffApp", ADTimeop);
4
5  InputParameters
6  ADTimeop::validParams()
7  {
8      InputParameters params = ADTimeKernelValue::validParams();
9      params.addClassDescription("Implements the time derivative term
       with anisotropy a(theta)^2 and coefficient tau_0.");
10     params.addParam<Real>("tau0", 1.0, "Coefficient tau_0 for the
       time derivative term.");
11     params.addParam<MaterialPropertyName>("eps_name", "eps", "The
       anisotropic interface parameter");
12     return params;
13 }
14
15 ADTimeop::ADTimeop(const InputParameters & parameters)
16   : ADTimeKernelValue(parameters),
```

```
    _tau0(getParam<Real>("tau0")),
    _eps(getADMaterialProperty<Real>("eps_name"))
{
}

ADReal
ADTimeop::precomputeQpResidual()
{
    // Compute the time derivative term: tau_0 * a(theta)^2 * d(phi)
   /dt
    return _tau0 * _eps[_qp]* _eps[_qp]* _u_dot[_qp];
}
```

The next terms to define are the weak forms of the following expressions:

$$W_0^2 \nabla \cdot \left[A(\theta)^2 \nabla \phi\right] - \frac{\partial}{\partial x}\left[A(\theta)A'(\theta)\frac{\partial \phi}{\partial y}\right] + \frac{\partial}{\partial y}\left[A(\theta)A'(\theta)\frac{\partial \phi}{\partial x}\right]$$

Listing C.4: Kernel Anisotropic terms

```
#include "ADkarma1.h"

registerMooseObject("diffApp", ADkarma1);

InputParameters
ADkarma1::validParams()
{
  InputParameters params = ADKernelGrad::validParams();
  params.addClassDescription("Anisotropic gradient Krama");
  params.addParam<MaterialPropertyName>("eps_name", "eps", "The
   anisotropic interface parameter");
  params.addParam<MaterialPropertyName>(
      "deps_name",
      "deps",
```

```
14       "The derivative of the anisotropic interface parameter with
     respect to angle");
15   return params;
16 }
17
18 ADkarma1::ADkarma1(const InputParameters & parameters)
19   : ADKernelGrad(parameters),
20     _eps(getADMaterialProperty<Real>("eps_name")),
21     _deps(getADMaterialProperty<Real>("deps_name"))
22 {
23 }
24
25 ADRealGradient
26 ADkarma1::precomputeQpResidual()
27 {
28   // Set modified gradient vector
29   const ADRealGradient v(-_grad_u[_qp](1), _grad_u[_qp](0), 0);
30
31   // Define anisotropic interface residual
32   return _eps[_qp] * _deps[_qp]  * v + _eps[_qp] * _eps[_qp]  *
     _grad_u[_qp];
33 }
```

In the file ‘ADSrc.C‘, the following terms are implemented:

$$\left(\phi - \phi^3\right) - \lambda U \left(1 - \phi^2\right)^2$$

Listing C.5: Kernel for nonlinear source contributions to the model.

```
1 #include "ADsrc.h"
2
3 registerMooseObject("diffApp", ADsrc);
4
```

```
InputParameters
ADsrc::validParams()
{
  InputParameters params = ADKernelValue::validParams();
  params.addParam<Real>("lambda", 3.1913, "Coefficient of the source
    term");
  params.addCoupledVar("UH", "Variable U from the second equation");
  return params;
}

ADsrc::ADsrc(const InputParameters & parameters)
  : ADKernelValue(parameters),
    _lambda(getParam<Real>("lambda")),
    _UH(adCoupledValue("UH"))
{
}

ADReal
ADsrc::precomputeQpResidual()
{
  const ADReal u_val = _u[_qp];
  const ADReal U_val = _UH[_qp];
  return -(u_val - _lambda * U_val * (1.0 - std::pow(u_val, 2))) *
   (1.0 - std::pow(u_val, 2)); // negative sign becuse LHS
}
```

Now, let's proceed to the implementation of the second PDE, equation 2.10.

The file 'E1.C' implements the following term:

$$\left(\frac{1+k}{2} - \frac{1-k}{2}h(\phi)\right)\frac{\partial U}{\partial t}$$

Listing C.6: Kernel for time derivative of U.

```
#include "E1.h"

registerMooseObject("diffApp", E1);

InputParameters
E1::validParams()
{
    InputParameters params = ADTimeKernelValue::validParams();
    params.addClassDescription("Time derivative.");
    params.addParam<Real>("k", 0.15, "Coefficient tau_0 for the time
    derivative term.");
    params.addRequiredCoupledVar("op", "Order parameter defining the
    solid phase");

    return params;
}

E1::E1(const InputParameters & parameters)
  : ADTimeKernelValue(parameters),
    _k(getParam<Real>("k")),
    _op(adCoupledValue("op"))
{
}

ADReal
E1::precomputeQpResidual()
{
    const ADReal _scalar = ((1+_k)-(1-_k)*_op[_qp])/2;

    return  _scalar* _u_dot[_qp];
}
```

On the right-hand side, the file ‘E2.C‘ implements the following term:

$$\vec{\nabla} \cdot \left(D\frac{1-\phi}{2}\vec{\nabla}U + \frac{W}{2\sqrt{2}}[1+(1-k)U]\frac{\partial\phi}{\partial t}\frac{\vec{\nabla}\phi}{|\vec{\nabla}\phi|} \right)$$

Listing C.7: Kernel for diffusion and antitraping term of U.

```
#include "E2.h"

registerMooseObject("diffApp", E2);

InputParameters
E2::validParams()
{
  InputParameters params = ADKernelGrad::validParams();
  params.addClassDescription("divergence term of EQ2.");
  params.addRequiredCoupledVar("op", "Order parameter defining the
   solid phase");
  params.addParam<Real>("k", 0.15, "Coefficient tau_0 for the time
   derivative term.");
  params.addParam<Real>("D", 2, "Coefficient tau_0 for the time
   derivative term.");

  return params;
}

E2::E2(const InputParameters & parameters)
    : ADKernelGrad(parameters),
    _op(adCoupledValue("op")),
    _grad_op(adCoupledGradient("op")),
    _k(getParam<Real>("k")),
    _D(getParam<Real>("D")),
    _v_dot(adCoupledDot("op"))
```

```
24
25 {
26 }
27
28 ADRealGradient
29 E2::precomputeQpResidual()
30 {
31
32   const Real tol = 1e-10;
33   ADReal q_phi = (1 - _op[_qp]) / 2;
34   ADRealGradient term1 = _D * q_phi * _grad_u[_qp];
35   ADReal norm_grad_phi = _grad_op[_qp].norm();
36   ADReal term2 = 0.0;
37
38   if (norm_grad_phi > tol)
39   {
40     term2 = 1/norm_grad_phi ;
41   }
42
43   ADRealGradient term_anti = (term1*_grad_op[_qp])*_v_dot[_qp
      ]*(1+(1-_k)*_u[_qp])/std::pow(8,0.5);
44
45   return term1;
46 }
```

The final term for U is the latent heat-like term, which is implemented in the 'E3.C' file:

$$+\frac{1}{2}\frac{\partial}{\partial t}\left\{\phi[1+(1-k)U]\right\}$$

This term represents the coupling effect due to the evolving phase field and concentration, similar to a latent heat contribution.

Listing C.8: Kernel term of U.

```
#include "E3.h"

registerMooseObject("diffApp", E3);

InputParameters
E3::validParams()
{
    auto params = ADKernelValue::validParams();
    params.addClassDescription("Eq2 last term LHS sign negative . ")
   ;
    params.addRequiredCoupledVar("op", "Coupled variable");
    params.addParam<Real>("k", 0.15, "Coefficient tau_0 for the time
    derivative term.");
    return params;
}

E3::E3(const InputParameters & parameters)
  : ADKernelValue(parameters),
  _v_dot(adCoupledDot("op")),
  _k(getParam<Real>("k"))
{
}

ADReal
E3::precomputeQpResidual()
{
    ADReal _scale = (1+(1-_k)*_u[_qp])/2;

    return -(_scale * _v_dot[_qp]);
}
```

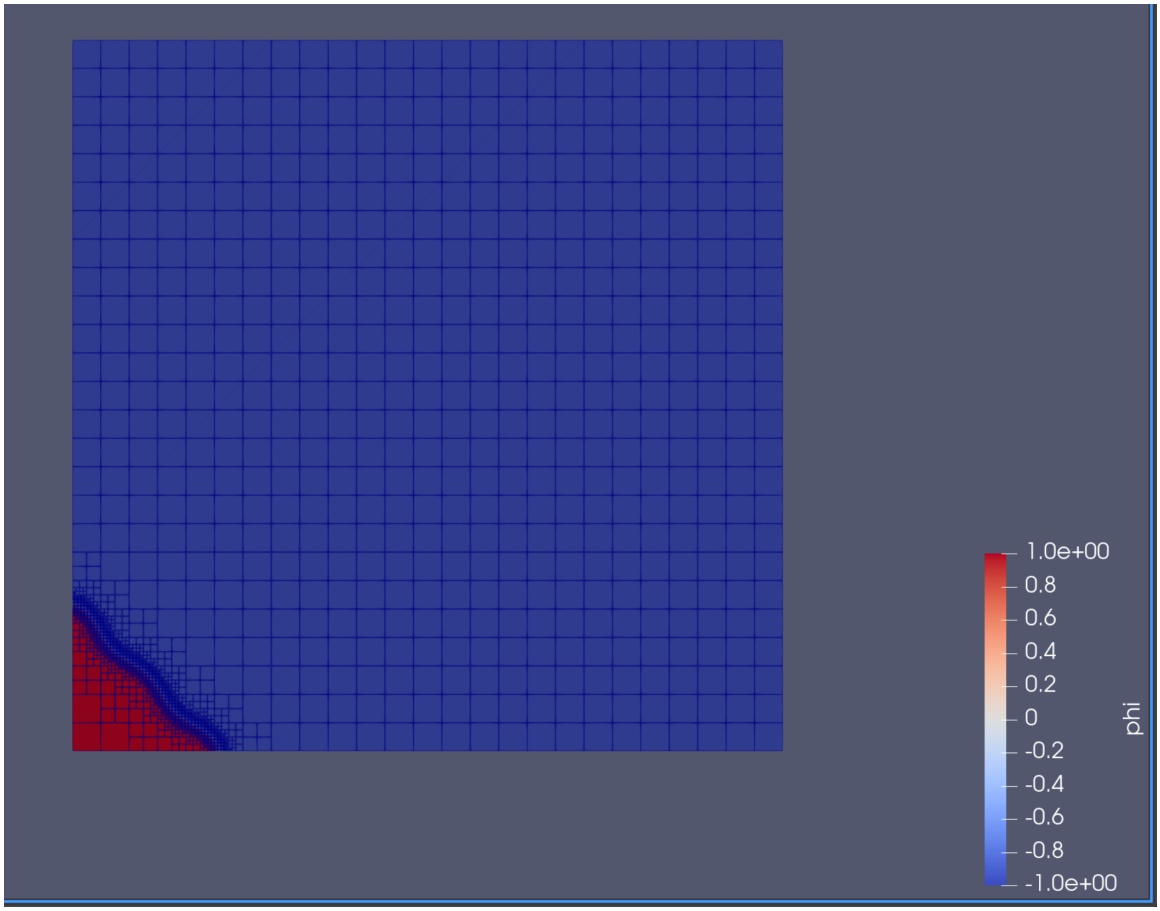


Figure C.1: MOOSE Implementation

C.2 OpenFOAM Implementation

The implementation in OpenFOAM was performed using version 9, as the author found the adaptive meshing features in this version more straightforward to work with, facilitating integration into the project. Please note that this implementation is an initial attempt by the author, who has limited experience with OpenFOAM, so use it at your own discretion. For more details, refer to the GitHub repository [**openfoam˙isothermal**].

The file ‘theta.H‘ implements the function for calculating θ:

$$A(\theta) = 1 + \epsilon \cos(4\theta)$$

This function captures the anisotropic effects by defining the dependence of $A(\theta)$ on the angle θ.

Listing C.9: function for theta.

```

void calculateTheta(
    const volScalarField& psi,  // Input field for phi
    volScalarField& theta       // Reference to the result field
)
{
    // Calculate the gradient of phi
    volVectorField gradPhi = fvc::grad(psi);

    // Calculate theta = atan2(gradPhi_y, gradPhi_x)
    theta = Foam::atan2(gradPhi.component(1), gradPhi.component(0));

}
```

The next file is the function for implementation of 2.9 in openfoam:

$$\tau \frac{\partial \phi}{\partial t} = W^2 \nabla^2 \phi + \phi - \phi^3 - \lambda g'(\phi) U$$

Listing C.10: Phase-field PDE.

```
#ifndef PHI_EQUATION_H
#define PHI_EQUATION_H
#include "fvCFD.H"
#include "volFields.H"
#include "dimensionedScalar.H"
// Function to solve the phi equation
void solvePhiEquation(
    volScalarField& psi,                    // Field for phi
    volScalarField& epsilon,          // Field for epsilon(theta)
```

```
10      volScalarField& epsilonDerivative, // Field for epsilon'(theta)
11      volScalarField& u,                  // Field for u
12      dimensionedScalar& lambda,
13      dimensionedScalar& tau0,           // Diffusion coefficient for u
14      dimensionedScalar& W0,
15      dimensionedScalar& Gr,
16      dimensionedScalar& Vspeed,
17      dimensionedScalar& Tscale,
18      dimensionedScalar& Wscale,
19      scalar ml,
20      scalar Conc0,
21      scalar k,
22      fvMesh& mesh,
23      Time& runTime                     // Time object for simulation
    time
24  )
25  {
26      scalar currentTime = runTime.value();
27      // Compute the gradient of phi
28      volVectorField gradPhi = fvc::grad(psi);
29      // Calculate the diffusion term: div(epsilon^2 * grad(phi))
30      // volScalarField diffusionTerm = fvc::div(epsilon * epsilon *
    gradPhi);
31      fvScalarMatrix diffusionTerm = fvm::laplacian(epsilon * epsilon,
     psi);
32      // Calculate the cross terms:
33      volVectorField TermX = fvc::grad(epsilon * epsilonDerivative *
    gradPhi.component(vector::Y));
34      volVectorField TermY= fvc::grad(epsilon * epsilonDerivative *
    gradPhi.component(vector::X));
35      volScalarField TermX_X = TermX.component(vector::X);
36      volScalarField TermY_Y = TermY.component(vector::Y);
37      volScalarField yCoord = mesh.C().component(vector::Y);
```

```
38
39      volScalarField epsilonSquared = epsilon/W0 * epsilon/W0 ;
40       // Define the PDE using fvScalarMatrix
41      fvScalarMatrix phiEqn(
42          tau0 * epsilonSquared * fvm::ddt(psi)
                    // Time derivative
43          - diffusionTerm                                          //
    Diffusion term
44          + TermX_X                                                //
    Cross term in x-direction
45          - TermY_Y                                                //
    Cross term in y-direction
46          - ( psi - pow(psi,3) )
47          + lambda*(u+ (Gr* Wscale)*(yCoord/W0 - Vspeed *(currentTime*
     Tscale/Wscale) )/(ml*Conc0/k*(1-k)))* pow((1 - psi* psi),2)//
    LHS of the equation
48      );
49      // Relax and solve the equation
50      phiEqn.relax();
51      phiEqn.solve();
52  }
53
54  #endif
```

Second PDE implementation 2.10:

$$\frac{1+k}{2}\frac{\partial U}{\partial t} = \vec{\nabla}\cdot\left(D\frac{1-\phi}{2}\vec{\nabla}U + \frac{W}{2\sqrt{2}}[1+(1-k)U]\frac{\partial\phi}{\partial t}\frac{\vec{\nabla}\phi}{|\vec{\nabla}\phi|}\right) + \frac{1}{2}\frac{\partial}{\partial t}\{\phi[1+(1-k)U]\}$$

Listing C.11: U PDE

```
1
2   #include "fvCFD.H"
```

```
#include "volFields.H"
#include "dimensionedScalar.H"
// #include "laplacianScheme.H"
#include "fvSchemes.H"
void solveUEqn(volScalarField& u, volScalarField& psi,
    dimensionedScalar& D_u, scalar k,dimensionedScalar&  W0  ,Time&
    runTime)
{
    Info << "Solving U equation at time = " << runTime.timeName() <<
    endl;
    // Calculate coefficients
    volScalarField coeff = (1 + k - (1 - k) * psi) / 2;
    volScalarField gradPhiMag = mag(fvc::grad(psi));
    dimensionedScalar SMALL("SMALL", gradPhiMag.dimensions(), 1e-12)
    ;
    volVectorField normalizedGradPhi = fvc::grad(psi) / (gradPhiMag
    + SMALL);

    // Define the diffusion term
    volScalarField diffTerm = D_u * (1 - psi) / 2;

    volVectorField Term2 = W0 / (2.0 * pow(2, 0.5)) * (1 + (1 - k) *
    u) *fvc::ddt(psi) * normalizedGradPhi;

    // Define the equation
    fvScalarMatrix UEqn
    (
        -fvm::ddt(coeff, u)      // // LHS of the equation
    );
    // Add other terms
    // surfaceVectorField diffTermInterpolated = fvc::interpolate(
    diffTerm);
    UEqn += fvm::laplacian(diffTerm, u);
```

```
    UEqn += fvc::div(Term2);
    UEqn += (1 + (1 - k) * u) / 2 * fvc::ddt(psi);
    // Apply under-relaxation to the equation (if needed)
    UEqn.relax();
    // Solve the equation
    UEqn.solve();
    Info << "Finished solving U equation" << endl;
}
```

next is the time loop in the openfoam which use the previous functions:

Listing C.12: Time loop

```
#include "fvCFD.H"
#include "solveUEqn.H"
#include "solvePsiEquation.H"
#include "epsilon.H"
#include "theta.H"
#include "InitialCond.H"
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    initializeFields(psi, u, mesh, radius, delta, runTime);
    // Manually write the fields to the disk
    runTime.write();  // Write the initial conditions to the 0 time
   step
    while (runTime.loop())
    {
        Info << "Time = " << runTime.timeName() << nl << endl;
        ////////////////////////// Solve the Karma model
   ////////////////////////////////
```

```
20          calculateTheta(psi, theta);
21          calculateEpsilon(theta, W0, ep4, anisotropy, epsilon);
22          calculateEpsilonDerivative(theta, W0, ep4, anisotropy,
    epsilonDerivative);
23          // Solve the equation for psi
24          solvePhiEquation(psi, epsilon, epsilonDerivative, u, lambda,
     tau0, W0, Gr, Vspeed, Tscale, Wscale, ml, Conc0, k, mesh,
    runTime);
25          // Solve the equation for u
26          solveUEqn(u, psi, D_u, k, W0, runTime);
27          ////////////////////////////////////  END
    ////////////////////////////////////
28          runTime.write();  // Write the results
29          Info << "ExecutionTime = " << runTime.elapsedCpuTime() << "
    s" << nl
30               << "ClockTime = " << runTime.elapsedClockTime() << " s"
     << nl << endl;
31      }
32
33      Info << "End\n" << endl;
34      return 0;
35  }
```

and finally the parameters and fields is defined in a seperate file:

Listing C.13: Create fields

```
1  // Create scalar field for psi
2  volScalarField psi
3  (
4      IOobject
5      (
6          "psi",
7          runTime.timeName(),
```

```
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
volScalarField u
(
    IOobject
    (
        "u",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
// Create scalar field for theta
volScalarField theta
(
    IOobject
    (
        "theta",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", dimensionSet(0, 0, 0, 0, 0, 0, 0),
   0.0) // Initialize to zero
);
```

```
// Create scalar field for epsilonDerivative
volScalarField epsilonDerivative
(
    IOobject
    (
        "epsilonDerivative",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", dimensionSet(0, 1, 0, 0, 0), 0.0) // Initialize to zero
);
// Create scalar field for epsilon
volScalarField epsilon
(
    IOobject
    (
        "epsilon",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,  // Set to NO_READ if calculated internally, change to MUST_READ if initialized from a file
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", dimensionSet(0, 1, 0, 0, 0), 0.0) // Initialize to zero, will be calculated based on theta
);
dimensionedScalar D_u("D_u", dimensionSet(0, 2, -1, 0, 0), 2.0); // Corrected dimensions for diffusivity [L^2 T^-1]
```

```
69 dimensionedScalar tau0("tau0", dimensionSet(0, 0, 1, 0, 0), 1.0);
      // Corrected dimensions for tau0 [T]
70 dimensionedScalar W0("W0", dimensionSet(0, 1, 0, 0, 0), 1.0);
71 dimensionedScalar ep4("ep4", dimensionSet(0, 0, 0, 0, 0), 0.02);
72 dimensionedScalar lambda("lambda", dimensionSet(0, 0, 0, 0, 0),
      3.1913); // Lambda is dimensionless
73 dimensionedScalar anisotropy("anisotropy", dimensionSet(0, 0, 0, 0,
      0), 4.0); // Mode number of anisotropy [dimensionless]
74 dimensionedScalar Wscale("Wscale", dimensionSet(0, 1, 0, 0, 0, 0, 0)
      , 1e-8); // meters (m)
75 dimensionedScalar Tscale("Tscale", dimensionSet(0, 0, 1, 0, 0, 0, 0)
      , 2.308e-8); // seconds (s)
76 dimensionedScalar Gr("Gr", dimensionSet(0, -1, 0, 0, 0, 0, 0), 1e7);
       // inverse meters (m^-1)
77 dimensionedScalar Vspeed("Vspeed", dimensionSet(0, 1, -1, 0, 0, 0,
      0), 1e-2); // meters per second (m/s)
78 scalar k = 0.15;
79 scalar radius = 20.0;  // this is the radius of the solid circle
      squared
80 scalar delta = 0.55;  // this is the initial value of u in the
      liquid region or omega
81 // scalar Wscale = 1e-8;
82 // scalar Tscale = 2.308e-8;
83 // scalar Gr = 1e7; // k/m
84 // scalar Vspeed = 1e-2; // m/s
85 scalar ml = 10.5; // liquidus slope
86 scalar Conc0 = 5; // % of initial concentration
```

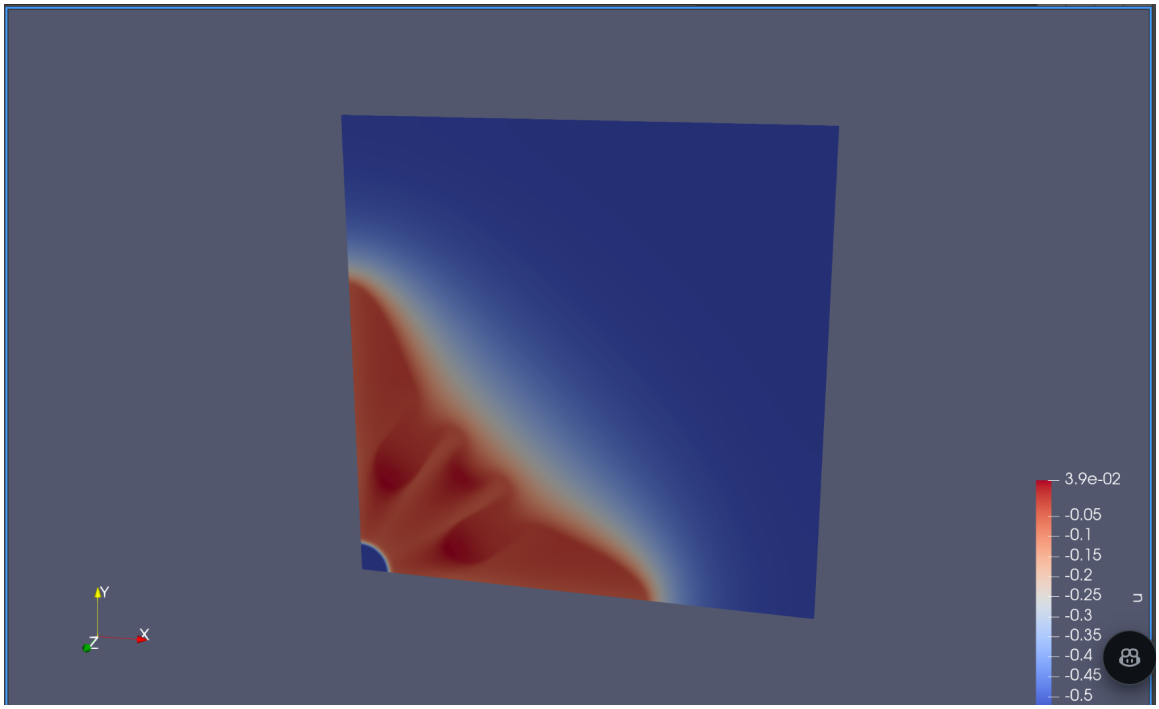


Figure C.2: Openfoam Implementation

Bibliography

[1] Kh. Moeinfar, F. Khodabakhshi, S.F. Kashani-bozorg, M. Mohammadi, and A.P. Gerlich. "A review on metallurgical aspects of laser additive manufacturing (LAM): Stainless steels, nickel superalloys, and titanium alloys". In: 16 (2022), pp. 1029–1068.

[2] Michel Rivero, Sayra Orozco, and Alberto Beltrán. "Numerical investigation of the melting process of gallium under inclination and partial heating". In: 59 (2023), p. 106510.

[3] Charles W. Hull. "Apparatus for production of three-dimensional objects by stereolithography". 4,575,330. 1986.

[4] Tuan D. Ngo, Alireza Kashani, Gabriele Imbalzano, Kate T.Q. Nguyen, and David Hui. "Additive manufacturing (3D printing): A review of materials, methods, applications and challenges". In: 143 (2018), pp. 172–196. URL: https://doi.org/10.1016/j.compositesb.2018.02.012.

[5] M. Adam Khan and J. T. Winowlin Jappes. *Innovations in Additive Manufacturing*. Switzerland, 2022.

[6] Jyotsna Dutta Majumdar, Dileep Madapana, and Indranil Manna. "3-D Printing by Laser-Assisted Direct Energy Deposition (LDED): The Present Status". In: 6 (2021), pp. 933–953.

[7] Tarasankar DebRoy, Hai-Lin Wei, Joseph S. Zuback, T. Mukherjee, John W. Elmer, John O. Milewski, et al. “Additive manufacturing of metallic components – Process, structure and properties”. In: 92 (2018), pp. 112–224.

[8] Jonathan Yoshioka and Mohsen Eshraghi. “Temporal evolution of temperature gradient and solidification rate in laser powder bed fusion additive manufacturing”. In: 59 (2023), pp. 1155–1166.

[9] Paul A Hooper. “Melt pool temperature and cooling rates in laser powder bed fusion”. In: 22 (2018), pp. 548–559.

[10] Dirk Herzog, Vanessa Seyda, Eric Wycisk, and Claus Emmelmann. “Additive manufacturing of metals”. In: 117 (2016), pp. 371–392.

[11] Milan Brandt. *Laser Additive Manufacturing: Materials, Design, Technologies, and Applications*. 2017.

[12] Longfei Zhou, Jenna Miller, Jeremiah Vezza, Maksim Mayster, Muhammad Raffay, Quentin Justice, Zainab Al Tamimi, Gavyn Hansotte, Lavanya Devi Sunkara, and Jessica Bernat. “Additive Manufacturing: A Comprehensive Review”. In: 24 (2024), p. 2668.

[13] Uma Radhakrishna, Shubham Bhaumik, and Kaushik Biswas. “Additive manufacturing techniques for biomedical applications: A review”. In: 6 (2020), p. 72.

[14] Martin Kurdve, Karl-Eric Persson, Magnus Widfeldt, Johan Berglund, and Alexander Drott. “Lead-Time Effect Comparison of Additive Manufacturing with Conventional Alternatives”. In: *Advances in Transdisciplinary Engineering*. Vol. 13. 2020, pp. 672–679.

[15] K. Kempen, L. Thijs, B. Vrancken, S. Buls, J. Van Humbeeck, and J.-P. Kruth. “Producing Crack-Free, High Density M2 HSS Parts by Selective Laser Melting: Pre-heating the Baseplate”. In: 29.4 (2013), pp. 123–134.

[16] C. Li, Z. Y. Liu, X. Y. Fang, and Y. B. Guo. "Residual Stress in Metal Additive Manufacturing". In: *Procedia CIRP*. Vol. 71. 2018, pp. 348–353.

[17] Mohammed Alghamdy, Rafiq Ahmad, and Basel Alsayyed. "Material Selection Methodology for Additive Manufacturing Applications". In: *Procedia CIRP*. Vol. 84. 2019, pp. 486–490.

[18] Yong Huang, Ming C. Leu, Jyoti Mazumder, and Alkan Donmez. "Additive Manufacturing: Current State, Future Potential, Gaps and Needs, and Recommendations". In: 137.1 (2015), p. 014001.

[19] Nils Much Magdalena Schreter-Fleischhacker Peter Munch. *MeltPoolDG: Discontinuous Galerkin Methods for Simulating Melt Pool Dynamics*. 2024. URL: https://github.com/MeltPoolDG/MeltPoolDG.

[20] P. S. Cook and A. B. Murphy. "Simulation of melt pool behaviour during additive manufacturing: Underlying physics and progress". In: 31 (2020), p. 100909.

[21] K. Kempen, L. Thijs, B. Vrancken, S. Buls, J. Van Humbeeck, and J.-P. Kruth. "Producing Crack-Free, High Density M2 HSS Parts by Selective Laser Melting: Pre-Heating the Baseplate". In: (2013).

[22] Zachary A. Young, Qilin Guo, Niranjan D. Parab, Cang Zhao, Minglei Qu, Luis I. Escano, Kamel Fezzaa, Wes Everhart, Tao Sun, and Lianyi Chen. "Types of spatter and their features and formation mechanisms in laser powder bed fusion additive manufacturing process". In: 36 (2020), p. 101438.

[23] Zhao Zhang, Yifei Wang, Peng Ge, and Tao Wu. "A Review on Modelling and Simulation of Laser Additive Manufacturing: Heat Transfer, Microstructure Evolutions and Mechanical Properties". In: 12.9 (2022), p. 1277.

[24] Lequn Chen, Guijun Bi, Xiling Yao, Jinlong Su, Chaolin Tan, Wenhe Feng, Michalis Benakis, Youxiang Chew, and Seung Ki Moon. "In-situ process mon-

itoring and adaptive quality enhancement in laser additive manufacturing: a critical review". In: (2024).

[25] D. L. McDowell, K. T. Story, and L. G. St-Pierre. "Phase-field modeling of microstructure evolution in additive manufacturing". In: 72 (2020), pp. 1564–1574.

[26] Matthias Markl and Carolin Körner. "Multiscale Modeling of Powder Bed-Based Additive Manufacturing". In: 46 (2016), pp. 93–123.

[27] Xin Bo Qi, Yun Chen, Xiu Hong Kang, Dian Zhong Li, and Tong Zhao Gong. "Modeling of coupled motion and growth interaction of equiaxed dendritic crystals in a binary alloy during solidification". In: 7.45770 (2017), pp. 1–16.

[28] Luke Scime and Jack Beuth. "Anomaly detection and classification in a laser powder bed additive manufacturing process using a trained computer vision algorithm". In: 19 (2018), pp. 114–126.

[29] J. Wang, H. Liu, H. J. Qi, and D. Fang. "Data-driven modeling and predictive control for additive manufacturing processes". In: 185 (2020), p. 109946.

[30] S. A. Khairallah, A. T. Anderson, A. Rubenchik, and W. E. King. "Laser powder-bed fusion additive manufacturing: Physics of complex melt flow and formation mechanisms of pores, spatter, and denudation zones". In: 108 (2016), pp. 36–45.

[31] Mohamad Bayat, Aditi Thanki, Sankhya Mohanty, Ann Witvrouw, Shoufeng Yang, Jesper Thorborg, Niels Skat Tiedje, and Jesper Henri Hattel. "Keyhole-induced porosities in Laser-based Powder Bed Fusion (L-PBF) of Ti6Al4V: High-fidelity modelling and experimental validation". In: 30 (2019), p. 100835.

[32] Zhuo Wang, Wenhua Yang, Qingyang Liu, Yingjie Zhao, Pengwei Liu, Dazhong Wu, Mihaela Banu, and Lei Chen. "Data-driven modeling of process, structure

and property in additive manufacturing: A review and future directions". In: 77 (2022), pp. 13–31.

[33] Inc. Flow Science. *FLOW-3D, Version 2023R1, Computer software*. Santa Fe, NM, 2023. URL: https://www.flow3d.com/.

[34] Mohamad Bayat, Wen Dong, Jesper Thorborg, Albert C. To, and Jesper H. Hattel. "A review of multi-scale and multi-physics simulations of metal additive manufacturing processes with focus on modeling strategies". In: 47 (2021), p. 102278.

[35] Chao Tang and Hejun Du. "Phase Field Modelling of Dendritic Solidification Under Additive Manufacturing Conditions". In: 74.8 (2022), pp. 2996–3009.

[36] Yefeng Yu, Lu Wang, Jun Zhou, Hongxin Li, Yang Li, Wentao Yan, and Feng Lin. "Impact of Fluid Flow on the Dendrite Growth and the Formation of New Grains in Additive Manufacturing". In: 55 (2022), p. 102832.

[37] T. Mukherjee, H. L. Wei, A. DebRoy, and T. DebRoy. "Heat and fluid flow in additive manufacturing—Part II: Powder bed fusion of stainless steel, and titanium, nickel and aluminum base alloys". In: 150 (2018), pp. 369–380.

[38] A. Raghavan, H. L. Wei, T. A. Palmer, and T. DebRoy. "Heat Transfer and Fluid Flow in Additive Manufacturing". In: 25.5 (2013), p. 052006.

[39] Abhik Choudhury, Klemens Reuther, Eugenia Wesner, Anastasia August, Britta Nestler, and Markus Rettenmayr. "Comparison of phase-field and cellular automaton models for dendritic solidification in Al–Cu alloy". In: 55 (2012), pp. 263–268.

[40] Hrvoje Jasak, Aleksandar Jemcov, and Zeljko Tukovic. *OpenFOAM: A C++ Library for Complex Physics Simulations*. 2007. URL: https://www.openfoam.com.

[41] Jun-Ho Jeong, Nigel Goldenfeld, and Jonathan A. Dantzig. "Phase Field Model for Three-Dimensional Dendritic Growth with Fluid Flow". In: (2001).

[42] R. Spatschek, E. A. Brener, D. E. Temkin, and S. G. Lipnizki. "Phase-Field Modeling of Equiaxed Dendritic Solidification in Multicomponent Alloys". In: 58.11 (2010), pp. 4015–4027.

[43] X. Wang, P.W. Liu, Y. Ji, Y. Liu, M.H. Horstemeyer, and L. Chen. "Investigation on Microsegregation of IN718 Alloy During Additive Manufacturing via Integrated Phase-Field and Finite-Element Modeling". In: 28 (2019), pp. 657–665.

[44] Eric Favre. *Modeling Marangoni Convection with COMSOL Multiphysics®*. 2015. URL: https://www.comsol.com/blogs/modeling-marangoni-convection-with-comsol-multiphysics.

[45] C. Beckermann, H.-J. Diepers, I. Steinbach, A. Karma, and X. Tong. "Modeling Melt Convection in Phase-Field Simulations of Solidification". In: 154 (1999), pp. 468–496.

[46] Maxime Theillard, Frédéric Gibou, and Tresa Pollock. "A Sharp Computational Method for the Simulation of the Solidification of Binary Alloys". In: 63 (2015), pp. 330–354.

[47] Alain Karma and Wouter-Jan Rappel. "Quantitative phase-field modeling of dendritic growth in two and three dimensions". In: 57.4 (1998), pp. 4323–4349.

[48] Nikolas Provatas and Ken Elder. *Phase-Field Methods in Material Science and Engineering*. 2008.

[49] V. L. Ginzburg. "On the Theory of Superconductivity". In: *Il Nuovo Cimento* 2.6 (1955), pp. 1064–1082.

[50] John W. Cahn and John E. Hilliard. “Free energy of a nonuniform system. I. Interfacial free energy”. In: *The Journal of Chemical Physics* 28 (1958), pp. 258–267.

[51] Nana Ofori-Opoku. “Phase-Field-Crystal Models for Microstructural Evolution and Phase Selection in Materials Science”. PhD thesis. McMaster University, 2013.

[52] Blas Echebarria, Roger Folch, Alain Karma, and Mathis Plapp. “Quantitative phase-field model of alloy solidification”. In: 70.6 (2004), p. 061604.

[53] J.C. Ramirez, C. Beckermann, A. Karma, and H.-J. Diepers. “Phase-field modeling of binary alloy solidification with coupled heat and solute diffusion”. In: 69.5 (2004), p. 051607.

[54] Alliance Canada. *Béluga Cluster Documentation*. 2024. URL: https://docs.alliancecan.ca/wiki/B%C3%A9luga/en.

[55] F. Freddi and L. Mingazzi. “Adaptive mesh refinement for the phase field method: A FEniCS implementation”. In: 14 (2023), p. 100127.

[56] Ankit Rohatgi. *WebPlotDigitizer*. Version 5.2. URL: https://automeris.io.

[57] S. M. Hosseini, R. Vinuesa, P. Schlatter, A. Hanifi, and D. S. Henningson. “Direct numerical simulation of the flow around a wing section at moderate Reynolds number”. In: 61 (2016), pp. 117–128.

[58] John W. Peterson, Alexander D. Lindsay, and Fande Kong. “Overview of the incompressible Navier–Stokes simulation capabilities in the MOOSE framework”. In: 119 (2018), pp. 68–92.

[59] Hans Petter Langtangen Jørgen S. Dokken Anders Logg. *The Navier-Stokes Equations*. 2025. URL: https://jsdokken.com/dolfinx-tutorial/chapter2/navierstokes.html.

[60] COMSOL Inc. COMSOL Modeling Software. *Marangoni Effect*. URL: https://www.comsol.com/model/marangoni-effect-20329.

[61] E. A. Spiegel and G. Veronis. "On the Boussinesq Approximation for a Compressible Fluid". In: 131 (1960), pp. 442–447.

[62] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. "The FEniCS Project Version 1.5". In: 3.100 (2015).

[63] Benjamin Rodenberg, Ishaan Desai, Richard Hertrich, Alexander Jaust, and Benjamin Uekermann. "FEniCS–preCICE: Coupling FEniCS to other simulation software". In: 16 (2021), p. 100807.

[64] Supriyo Ghosh, Li Ma, Nana Ofori-Opoku, and Jonathan E. Guyer. "On the Primary Spacing and Microsegregation of Cellular Dendrites in Laser Deposited Ni–Nb Alloys". In: 25.6 (2017), p. 065002.

[65] Chao Zeng, Yun Jia, Jiutian Xue, Xiangyao Liu, and Qingqing Dong. "Influence of the thermal-fluid behavior on the microstructure evolution during the process of selective laser melting of Ti6Al4V". In: 8 (2022), e11725.

[66] Jincheng Wang, Rui Zhu, Yujing Liu, and Laichang Zhang. "Understanding melt pool characteristics in laser powder bed fusion: An overview of single- and multi-track melt pools for process optimization". In: 2 (2023), p. 100137.

[67] Ashish Kumar Mishra and Arvind Kumar. "Effect of Process Parameters on Melt Pool Characteristics and Solidification Process During Laser Powder Bed Fusion of AlSi10Mg Alloy". In: 11 (2024), pp. 260–283.

[68] Hongda Wang, Mohamed S. Hamed, and Sumanth Shankar. "Interaction between primary dendrite arm spacing and velocity of fluid flow during solidification of Al–Si binary alloys". In: *Journal of Materials Science* 53 (2018), pp. 9771–9789.

[69] P. K. Galenko. “Convection effect on solidification microstructure: dendritic arm spacing and microsegregation”. In: *The European Physical Journal Special Topics* 232 (2023), pp. 1261–1271.

[70] A. Snow and B. Mulder. “Coupled Snow and Fluid Flow: Challenges of Stabilizing Full Coupling”. In: (2018), pp. 123–135.

[71] Guillaume Giudicelli et al. “3.0 - MOOSE: Enabling massively parallel multiphysics simulations”. In: 26 (2024), p. 101690.

[72] Elia Merzari et al. “Cardinal: A Lower-Length-Scale Multiphysics Simulator for Pebble-Bed Reactors”. In: *Nuclear Technology* (2021).