UNIVERSITY OF CALIFORNIA

Los Angeles

Numerical and Deep Learning Study of Transpiration Cooling for Sharp Hypersonic Leading Edge

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Aerospace Engineering

by

Danny Donghyun Ko

2023

ABSTRACT OF THE DISSERTATION

Numerical and Deep Learning Study of Transpiration Cooling for Sharp Hypersonic Leading Edge

by

Danny Donghyun Ko
Doctor of Philosophy in Aerospace Engineering
University of California, Los Angeles, 2023
Professor Y. Sungtaek Ju, Chair

Recent escalating interest in the development of highly maneuverable hypersonic vehicles demands sharp leading edges. However, sharp leading edges induce severe aerothermal conditions where conventional passive or ablative thermal protection systems fail to protect the leading edge. This dissertation investigates transpiration cooling employing oxide coolants as an alternative system to thermally protect sharp leading edges. This dissertation studies two primary objectives that collectively assess the viability of transpiration cooling. The first objective is to characterize the performance of transpiration cooling for various coolant properties, flight conditions, and leading edge radii. We use semi-analytical boundary layer model to predict the surface temperature, evaporative mass flux, and boiling limit of the system. Our findings do not readily align with an optimal set of material properties for transpiration cooling. Instead, certain coolant properties are more appropriate for various flight conditions and leading edge sizes. The second objective is to characterize the liquid coolant flow through porous media for various hypersonic aerothermal conditions. We experimentally and numerically obtain the permeability of representative silicon carbide foams to estimate the necessary pressure gradient and assess the self-pumping capability of various coolants to meet the mass flow rate demands at the surface. We then utilize computationally efficiency deep learning models to characterize the porous media at the pore-scale to facilitate the design of the microstructures of porous leading edge. Our two numerical frameworks cohere both external and internal aspects of the system to evaluate the performance of transpiration cooling and optimize the coolant properties to effectively protect sharp leading edges, which are paramount for highly maneuverable hypersonic vehicles, for various hypersonic applications.

The dissertation of Danny Donghyun Ko is approved.

Tim Fisher

Xiaolin Zhong

Mitchell Spearrin

Philippe Sautet

Y. Sungtaek Ju, Committee Chair

University of California, Los Angeles

2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Hypersonic Flight

The pursuit of flight at hypersonic speeds has garnered significant attention since the early 1950s. The safe return of human astronauts from Earth's orbit in missions such as Mercury, Gemini, and Apollo served as the original impetus behind the extensive research in hypersonic flight [3]. Concurrently, investigations into the feasibility of sustained air-breathing vehicles capable of cruising at hypersonic velocities were prevalent during the formative years. The application potential of this class of vehicle is substantial, spanning diverse domains. For example, hypersonic vehicles and jets hold the promise of enabling rapid target engagement and enforcing the existing defense systems with unparalleled agility and speed [71, 87]. Furthermore, hypersonic sub-orbital spaceplanes, executing takeoffs and landings from conventional runways, have the capability to supplant the initial stage of traditional rockets for delivering payloads into orbit, thus enabling cost-effective and frequent operations, potentially on a daily basis [60]. Another transformative application lies in passenger hypersonic aircraft, poised to redefine the benchmarks set by previous passenger aviation. Early conceptual explorations of hypersonic passenger aircraft, exemplified by Boeing's Hypersonic Airliner, suggest the potential to drastically reduce intercontinental flight duration to under an hour, enabling global travel within a three-hour timeframe [62, 93].

However, until the present juncture, the domain of sustained air-breathing hypersonic flight has encountered limited prospects for comprehensive exploration and thorough study, primarily stemming from the intricate nature of the concomitant challenges and resource constraints. Development initiatives in the United States experienced recurring terminations during the 1960s and 1990s, primarily due to design prerequisites that surpassed the technological capabilities of the era [3]. Nonetheless, contemporary escalated demands and remarkable technological advancements have compelled the aerospace community to embark on the practical realization of systems capable of achieving enhanced velocities and extended ranges, accompanied by more substantial momentum and unprecedented engineering intricacies. Consequently, recent breakthroughs in this realm have unveiled a multitude of possibilities. Notably, the unmanned X-43's milestone flight in 2004 marked a transient hypersonic excursion, attaining Mach 10 for approximately 10 seconds, while the X-51's feat in 2013 reached Mach 5.1 for a duration slightly under 4 minutes [3, 81].

The pursuit of sustained air-breathing hypersonic flight presents formidable challenges, primarily arising from the intricate interplay between aerodynamic heating and the imperative of maintaining an optimal forebody configuration. A recent development gaining momentum involves the adoption of sharply defined leading edges, driven by the objective of bolstering maneuverability and maximizing the lift-to-drag ratio. However, this evolution introduces an escalating hurdle: as the curvature of the leading edge diminishes, the severity of aerothermal heating experiences an exponential escalation, as depicted in Figure 1.1 [95, 21]. The curvature reduction to millimeter scales precipitates an exponential surge in the incident aerothermal heat flux, approaching magnitudes of 10 MW/m^2. This unprecedentedly extreme heat flux mandates the strategic implementation of an efficient Thermal Protection System (TPS) to sustain the leading edge within operational temperature thresholds while simultaneously preserving the sharp geometry essential for unimpeded aerodynamic performance.

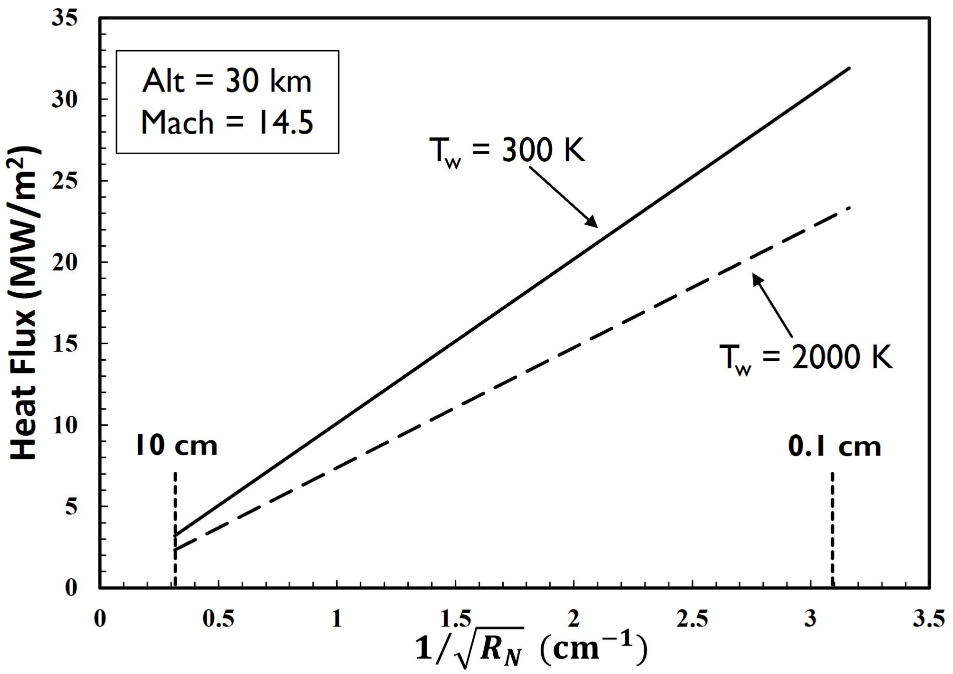


Figure 1.1: Magnitude of incident aerothermal heat flux at the stagnation point as a function of leading edge size. Decrease in leading edge size increases the incident heat flux quadratically. Adopted from Ref [21].

1.2 Reusable Insulation Tiles

Historically, passive TPS have established their reliability through missions like the Space Shuttle. These systems employ materials capable of enduring high temperatures and absorb-

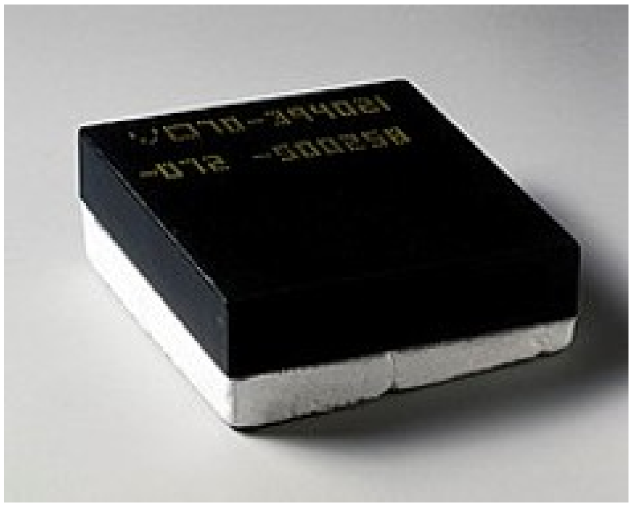


Figure 1.2: Image of thermal insulation tiles used for the Space Shuttle.

ing aerodynamic heat while maintaining surface temperatures below acceptable limits via radiation [32]. Reinforced carbon-carbon composites were integral to the Space Shuttle, effectively enduring extreme heating during re-entry and safeguarding the vehicle and crew. These composite tiles, experiencing temperatures up to 1300°C, displayed commendable oxidation resistance [19]. Notably, the insulation tiles demonstrated reusability, evident from the successful sequential Shuttle launches, which is a desirable trait for hypersonic vehicles.

However, traditional passive methods may insufficiently address the rigorous aerothermal conditions encountered by slender vehicles. Passive insulation tiles are susceptible to substantial oxidation and weight loss under continuous extreme aerothermal and reactive conditions, particularly near sharp leading edges. Stagnation temperatures in such circumstances can exceed radiation cooling limits, potentially leading to significant mass loss and oxidation. Consequently, passive insulation tiles are suited for blunt shapes with relatively large radii of curvature, and it is expected to fail for sharp leading edges.

1.3 Ablative Heat Shields

An alternative TPS approach employs material ablation through chemical reactions and surface vaporization. In the presence of severe aerothermal heating, this TPS undergoes endothermic chemical reactions, phase transitions, and thermal decomposition, effectively absorbing incident heat flux.

Historically, early re-entry vehicles and crewed spacecraft, including intercontinental ballistic missiles (ICBMs), Mercury, Gemini, and Apollo, relied on ablative TPS to ensure safe descent through Earth's atmosphere [32]. ICBMs experienced brief yet intense heating during re-entry, demanding a TPS capable of rapid heat dissipation. Similarly, crewed spacecraft returning from orbit employed ablative TPS for prolonged re-entry. In more recent times, interplanetary exploration landers such as Viking, Pathfinder, Curiosity, and InSight integrated aeroshells composed of fiber-reinforced resin to withstand high temperatures during transient re-entry and protect the main vehicle [27]. Technological advancements introduced diverse ablative materials in contemporary re-entry vehicles, while maintaining their fundamental function. Various Mars exploration spacecraft utilized aeroshells, often featuring an ablative heat shield constructed from honeycomb aluminum skeletons sandwiched between graphite-epoxy fiber-reinforced resin composite sheets at the front end. Some crewed vehicles adopted phenolic-impregnated carbon ablators (PICA), known for their lightweight nature and resilience, capable of withstanding over 3000 W/cm^2 [92].

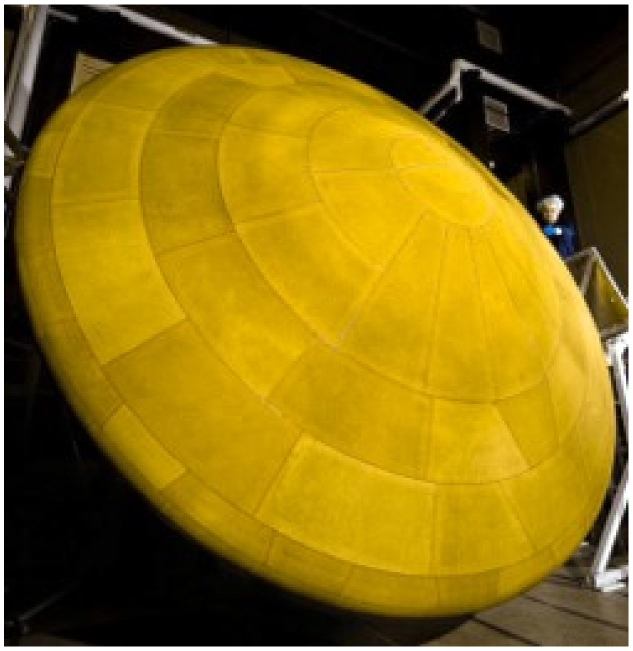


Figure 1.3: Image of PICA ablative heat shield typically used in Mars reentry vehicles.

Despite the established success of conventional passive and ablative TPS in numerous missions, they exhibit intrinsic limitations in terms of utility and reliability. The primary constraint of traditional TPS lies in maintaining shape and stability under extreme conditions. Vehicles designed for enhanced maneuverability require streamlined profiles throughout their flight. However, ablative TPS are unable to preserve shape over extended flight duration due to mass convection away from the surface [32]. Although ablative heat shields can endure the harsh hypersonic environment, they experience shape alteration. For example, typical PICA heat shields demonstrate approximately 5 mm total recession length at the stagnation point during brief periods [92], a characteristic unsuitable for serving as TPS for aerodynamically sharp hypersonic vehicles.

1.4 Transpiration Cooling

Transpiration cooling emerges as a promising technique for efficient thermal safeguarding of hypersonic vehicle leading edges, while concurrently maintaining shape integrity and achieving high lift-to-drag ratios [120, 116]. This methodology entails a permeable leading edge structure through which a working fluid is introduced, facilitating heat absorption and dissipation. The operational principle of transpiration cooling is illustrated in Figure 1.4. Internal supply of liquid originates either from capillary forces due to surface tension or external forces. Subsequently, the liquid traverses the porous leading edge to the surface where it experiences phase transition from liquid to vapor driven by the incident aerothermal heat flux. The resulting vapor is then transported away from the surface, entrained within the hypersonic flow.

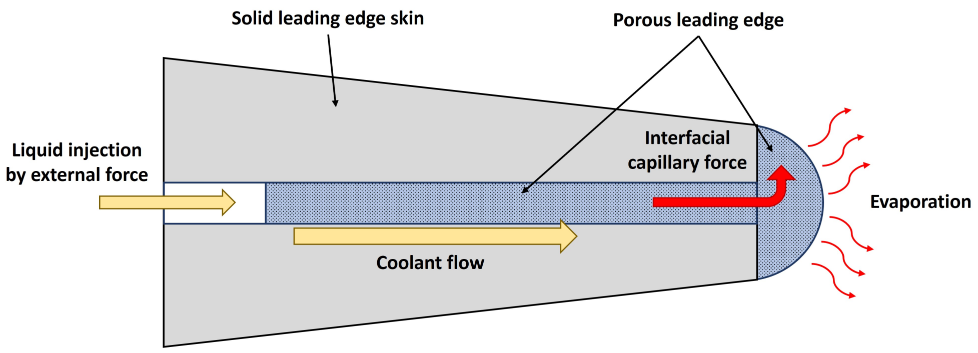


Figure 1.4: Schematic of transpiration cooling of a leading edge. Liquid supplied from the vehicle flows through porous leading edge surface and alleviates heat via evaporation.

A few recent experimental studies [96, 75, 99, 86, 38, 37] explored transpiration cooling with liquid coolants, which can take advantage of the high latent heat of phase change. Van Foreest et al. [96] experimentally demonstrated the effectiveness using a porous porcelain leading edge cooled with water as the working fluid for heat fluxes up to 2000 MW/m^2.

By supplying the coolant from a pressurized plenum, they maintained the leading edge temperature below 300K. However, the coolant solidified, changing the leading edge shape due to excessive coolant flow and the extremely low pressure conditions of the arc-jet facility. Huang et al. [38] also demonstrated transpiration cooling using water as the working fluid, driven by capillary action. Although they maintained surface temperature below 373K, their system reached a maximum heat flux limit of 1 MW/m^2 due to the lack of self-pumping capability of the porous media.

To achieve desired net liquid flow to the surface, sufficient pressure gradient is necessary to overcome the viscous and inertial losses caused by the fluid movement through porous media. Complex microscopic geometry of porous media and fluid interaction with it are often not trivial. Generally, the flow behavior can be characterized at the macroscopic level by Darcy-Forchheimer equation defined as

$$\frac{dP}{dx} = \frac{\mu u_D}{K} + \frac{\rho {u_D}^2}{C} \tag{1.1}$$

where volume averaged permeability, K, and inertial coefficient, C characterizes the morphology of the porous media. These two parameters characterize the viscous and inertial regimes, respectively, of the flow which are distinguished by the magnitude of pore Reynolds number defined as

$$Re_p = \frac{\rho u_p D_p}{\mu} \tag{1.2}$$

Here, D_p is the average pore diameter and u_p is the pore velocity defined as $u_p = u_D/\epsilon$ [49]. Figure 1.5 depicts the different flow regimes of porous media as a function of Re_p. The creep-

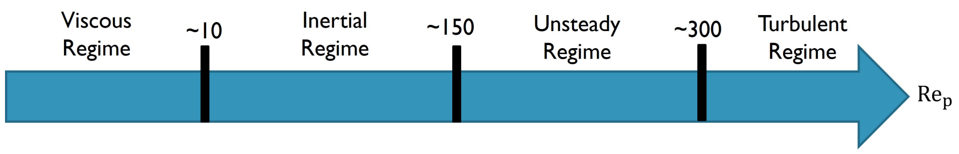


Figure 1.5: Flow regime in porous media as a function of pore Reynolds number. Pore Reynolds numbers below approximately 10 and 150 indicate viscous and inertial regimes, respectively. Adapted from [49].

ing or Darcy flow regime is when Re_p is less than approximately 10 and is mainly governed by viscous interaction. For Re_p below approximately 150, inertial effects are significant and increase flow resistance across the porous media. Higher Re_p introduces growth of unsteady flow and turbulence that induces further flow resistance.

For typical hypersonic flight conditions, the flow is well within the viscous regime. In the limit of small Re_p, Equation 1.1 is simplified to

$$\frac{dP}{dx} = \frac{\mu u_D}{K} \tag{1.3}$$

such that the pressure drop across the porous media is proportional to permeability and liquid viscosity. In a simple one-dimensional model, the driving pressure must be larger than the pressure loss:

$$P_D \geq \frac{\mu u_D}{K} L \tag{1.4}$$

Here, P_D is the driving force per unit area or pressure which can stem from capillary force or external force. A requisite positive pressure gradient is indispensable to ensure an ample supply of liquid to the vehicle's surface. Insufficient liquid provision can induce a recession of the liquid-vapor interface, creating localized hot-spots and surpassing the leading edge's thermal threshold, resulting in thermal runaway. Thus, precise and confident permeability comprehension proves pivotal to guarantee adequate fluid delivery and avert system failure.

Supplied liquid undergoes evaporation at the surface, harnessing its latent heat during liquid-vapor phase change to alleviate incident aerothermal heat flux. The amount of alleviated heat is proportional to the rate of evaporated mass flux:

$$q = \dot{m} h_{fg} \tag{1.5}$$

The rate of evaporation at the surface relies heavily on hypersonic flow characteristics within the vehicle's boundary layer. Mass transfer at the surface establishes functional connections with temperature and concentration which are inherently tied to mass transfer rates [85]. The coupled heat and mass transfer dynamics render predicting evaporation rates challenging without comprehensive insight into the complete boundary layer solution.

1.5 Thesis Objective

This dissertation explores transpiration cooling employing oxide coolants. In this TPS, solid particles of the coolant may be delivered towards the leading edge using a spring-piston assembly or similar mechanical systems. The particles melt and then the molten liquid flows through the high porosity porous leading edge to reach the surface. The liquid evaporates at the surface to absorb incident heat flux and discharges into the external hypersonic flow. Figure 1.6 depicts a schematic of transpiration cooling utilizing oxide coolant. Oxide coolants

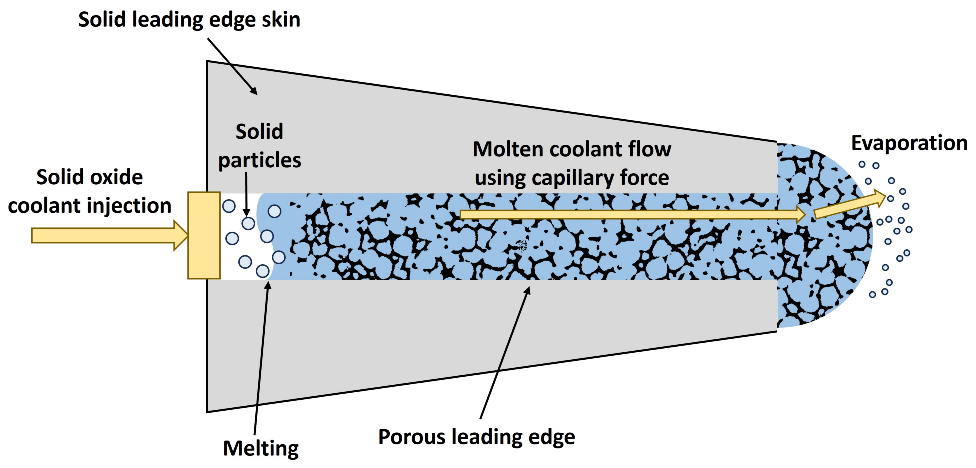


Figure 1.6: Schematic of transpiration cooling of a leading edge utilizing solid oxide coolants. Solid supplied from the vehicle melts and flows through the porous leading edge surface and alleviates heat via evaporation.

provide several advantages over water. High surface energies of molten oxides can increase the internal capillary force and thereby enhance the capillary driven flow through the porous leading edge. This reduces or potentially eliminates the need for an external liquid pumping system. Furthermore, oxides provide great chemical resistance against highly energetic atomic oxygen present in the hypersonic flow. Their wide range of available material properties also facilitates material selection that maximizes the for a given flight trajectory or application.

This dissertation studies two primary objectives that collectively investigate transpiration cooling. The first primary objective consists of the following sub-objectives:

1. Numerically assess the viability of transpiration cooling employing oxides as an alternative TPS to effectively protect sharp leading edges in hypersonic flight.

2. Characterize the surface temperature, evaporative mass flux, and boiling limit of transpiration cooling for different coolant properties, flight conditions, and leading edge radii to facilitate the optimization of the system.

The second primary objective comprise of the following sub-objectives:

1. Experimentally and numerically characterize the liquid coolant flow through high porosity porous media for various hypersonic aerothermal conditions to estimate the pressure gradient and assess the self-pumping capability of various coolants.

2. Develop computationally efficient deep learning models to characterize the flow through high porosity porous media at the pore-scale and to facilitate the design of the microstructures of porous leading edge to optimize the performance of transpiration cooling.

These two primary objectives investigate both external and internal aspects of the system and collectively contributes towards the development and optimization of transpiration cooling utilizing oxide coolant as an alternative system.

Chapter 2

Evaporation in Hypersonic Flow

The potential of transpiration cooling using liquid coolant has been demonstrated by various experimental studies. For example, Van Foreest et al. [96] experimentally showed the effectiveness of transpiration cooling by subjecting a porous alumina ceramic leading edge in a simulated hypersonic flow using an arc-jet test facility. They used liquid water as the working fluid and reduced the maximum stagnation temperature of the leading edge from 1900K to 500K. However, the TPS required an external system to deliver the coolant through the leading edge which can be expensive and cumbersome for practical hypersonic vehicles. To eliminate the use of an external pumping system, Huang et al. [38] incorporated transpiration cooling that solely relies on capillary action to deliver the flow. They subjected a circular copper coupon equipped with transpiration cooling using liquid water as the coolant and a hydrophilic porous medium to facilitate the capillary action. They successfully maintained the maximum temperature of the coupon below 373K. However, their system was limited to a maximum heat flux of 1 MW/m2 due to the self-pumping capability of the porous media.

To address the above challenges of transpiration cooling using a liquid coolant, we consider oxides as the working fluid in our study. In this TPS, solid particles of the coolant may be delivered towards the leading edge using a spring-piston assembly or similar mechanical systems. The particles melt and then the molten liquid flows through the porous medium to reach the surface. The liquid evaporates at the surface to absorb incident heat flux and discharges into the external hypersonic flow. Figure 1.6 depicts a schematic of transpiration cooling utilizing oxide coolant. Oxide coolants provide several advantages over water. High surface energies of molten oxides can increase the internal capillary force and thereby enhance the capillary driven flow through the porous leading edge. This reduces or potentially eliminates the need for an external liquid pumping system. Furthermore, oxides provide great chemical resistance against highly energetic atomic oxygen present in the hypersonic flow. Their wide range of available material properties also facilitates material selection that maximizes the for a given flight trajectory or application. Previous study on gas transpiration cooling [104] have shown that certain material properties, such as heat capacity and molar mass, can significantly influence the capability of TPS. For transpiration cooling using liquid coolant, other material properties including the latent heat of phase change and the saturation temperature are also critical.

In this chapter, we numerically investigate the evaporative transpiration cooling under representative external hypersonic flow conditions and explore its performance and limitations. Many previous works [7, 33, 79, 30, 15, 89, 88] have numerically characterized the evaporation process in hypersonic flows. However, these studies assumed the evaporating surface to be in a state of thermodynamic equilibrium. This assumption is only valid when the maximum mass transfer rate at the surface is limited by the diffusion rate through the boundary layer. At the stagnation point of sharp leading edges, the maximum mass diffusion rate through the boundary layer can be significant, potentially exceeding the maximum mass transfer rate predicted by the kinetic theory [85, 13]. The thermodynamic equilibrium assumption can thereby lead to considerable under-prediction of the surface temperature [54, 4]. An unexpectedly high surface temperature is of great concern since it can lead to the two main modes of failure of the TPS: softening or melting of the porous medium comprising the leading edge and nucleation of vapor bubbles within the porous flow paths which disrupt the coolant flow to the surface and causes surface dry-out [20, 30]. Recent studies of turbulence transition and recession of molten oxide layers during ablation have considered thermodynamic non-equilibrium conditions at the surface to characterize heat and mass transfer along hypersonic leading edges [65, 66, 18, 73]. But to our knowledge, there has been no systematic study of the evaporative transpiration cooling accounting for thermodynamic non-equilibrium conditions.

We study the evaporative transpiration cooling under thermodynamic non-equilibrium conditions and examine the effects of coolant material properties, flight conditions, and leading edge radii on the surface temperature, evaporative mass flux, and boiling. To this end, we utilize both the 2D axi-symmetric boundary layer theory and direct numerical simulation (DNS) utilizing a third-order 3D shock-fitting finite difference scheme and model the evaporation process at the stagnation point of a sharp, hemispherical leading edge. In the following, we parametrically characterize the TPS performance over a wide range of flight conditions, leading edge radii, and material properties. We then discuss results from DNS for eight representative flight conditions and a set of seven different materials to validate the boundary layer model. Our study provides the framework to examine the performance of evaporative transpiration cooling under various conditions and assess its viability for sharp leading edges of hypersonic vehicles with superior maneuverability.

2.1 Boundary Layer Model

Boundary layer theory have been utilized by many investigators [21, 85, 104] to solve viscous, hypersonic flow near a flat surface. This approach provides an accurate, low-computational cost model to investigate the hydraulic and thermodynamic behavior of the hypersonic flow at the stagnation point. Hence, we utilize the boundary layer theory at the stagnation point to investigate the evaporation process over a 2D axi-symmetric sharp, hemispherical leading edge. The general form of the governing equations of hypersonic flow are [3]

Continuity:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho V) = 0 \tag{2.1}$$

Conservation of Momentum:

$$\rho \frac{DV}{Dt} = -\nabla P + \nabla \cdot \tau \tag{2.2}$$

Conservation of Energy:

$$\rho \frac{Dh}{Dt} = \frac{DP}{Dt} - \nabla \cdot q + \Phi \tag{2.3}$$

where Φ is the dissipation function, and q is the heat flux vector defined as

$$\vec{q} = -k\nabla T + \sum_i \rho_i U_i h_i \tag{2.4}$$

Conservation of species i:

$$\frac{\partial n_i}{\partial t} + \nabla \cdot (n_i V_i) = \dot{N}_i \tag{2.5}$$

$$\frac{\partial \rho_i}{\partial t} + \nabla \cdot (\rho_i V_i) = \dot{w}_i \tag{2.6}$$

where equations are written in terms of number and mass density, n_i and ρ_i, respectively. The absolute velocity of species i is defined as

$$V_i = V + U_i \tag{2.7}$$

where U_i is the relative velocity of species i and V is the absolute velocity of the mixture. Hence, Equations (2.5) and (2.10) become

$$n \frac{DX_i}{Dt} + \nabla \cdot (n_i U_i) = \dot{N}_i + X_i \frac{n}{M} \frac{DM}{Dt} \tag{2.8}$$

$$\rho \frac{DC_i}{Dt} + \nabla \cdot (\rho_i U_i) = \dot{w}_i \tag{2.9}$$

Either Equation (2.8) or (2.9) can be derived from one another using the relationship between mass and mole fraction:

$$C_i = X_i \frac{M_i}{M} \tag{2.10}$$

Equation of state:

$$P = \rho \frac{R}{M} T \tag{2.11}$$

where R is the universal gas constant and the mixture molar mass is defined as

$$M = \sum_i X_i M_i \tag{2.12}$$

We then reduce the generic governing equations by utilizing the boundary layer theory assuming that the characteristic length of the body is greater than that of the viscous boundary layer near the surface. As a result, the simplified set of boundary layer equations are:

Continuity:

$$\frac{\partial}{\partial x}(\rho u r) + \frac{\partial}{\partial y}(\rho v r) = 0 \tag{2.13}$$

Conservation of x and y momentum:

$$\rho u \frac{\partial u}{\partial x} + \rho v \frac{\partial u}{\partial y} = \frac{\partial P}{\partial x} + \frac{\partial}{\partial y}\left(\mu \frac{\partial u}{\partial y}\right) \tag{2.14}$$

$$\frac{\partial P}{\partial y} = 0 \tag{2.15}$$

Conservation of energy:

$$\rho \bar{c_p}\left(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}\right) = u\frac{\partial P}{\partial x} + \mu\left(\frac{\partial u}{\partial y}\right)^2 + \frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right) + \sum_i c_{p_i}\rho_i U_i \frac{\partial T}{\partial y} \tag{2.16}$$

Conservation of species i:

$$\rho u \frac{\partial C_i}{\partial x} + \rho v \frac{\partial C_i}{\partial y} + \frac{\partial}{\partial y}\left(\rho_i j_i\right) = \dot{w}_i \tag{2.17}$$

which can, again, be written in terms of either the mass or mole fraction using Equation (2.9). A detailed derivation of boundary layer equations are presented by Anderson [3].

For a mixture of binary species, the diffusive flux follows the Fick's law of diffusion closely given that second order diffusion effects such as thermal diffusion, pressure gradient, and external forces, are negligible. When there are more than two different species in the flow, diffusive flux takes the following form [35]:

$$j_i = \sum_{j \neq i} \frac{M_i M_j}{M^2} \rho D_{ij} \frac{\partial X_j}{\partial y} \tag{2.18}$$

Boundary layer equations are indeed much simpler, but it is still a system of partial differential equations which, in its current form, cannot be solved without using advanced numerical method. However, by using similarity technique, a system of partial differential equations can be transformed into a system of ordinary differential equations which can be solved semi-analytically. Similarity technique has been the procedure to solve boundary layer equations in many previous boundary layer analysis [21, 57, 85]. Similarity variables and Mangler transformations defined previously by Lees [57] are used here. For axi-symmetric body, the similarity transformations become:

$$\eta = \frac{u_e}{\sqrt{2\varepsilon}} \int_0^y \rho r dy \tag{2.19}$$

$$\varepsilon = \int_0^x \rho_s \mu_s u_e r^2 dx \tag{2.20}$$

The stream function is, then, defined as:

$$\Psi = f\sqrt{2\varepsilon} \tag{2.21}$$

such that the stream function readily satisfies the continuity equation:

$$\rho u r = \frac{\partial \Psi}{\partial y} \quad \text{and} \quad \rho v r = -\frac{\partial \Psi}{\partial x} \tag{2.22}$$

We apply the above similarity transformation at the stagnation point in the following section to obtain a set of ordinary differential equations.

2.1.1 Frozen Flow at the Stagnation Point

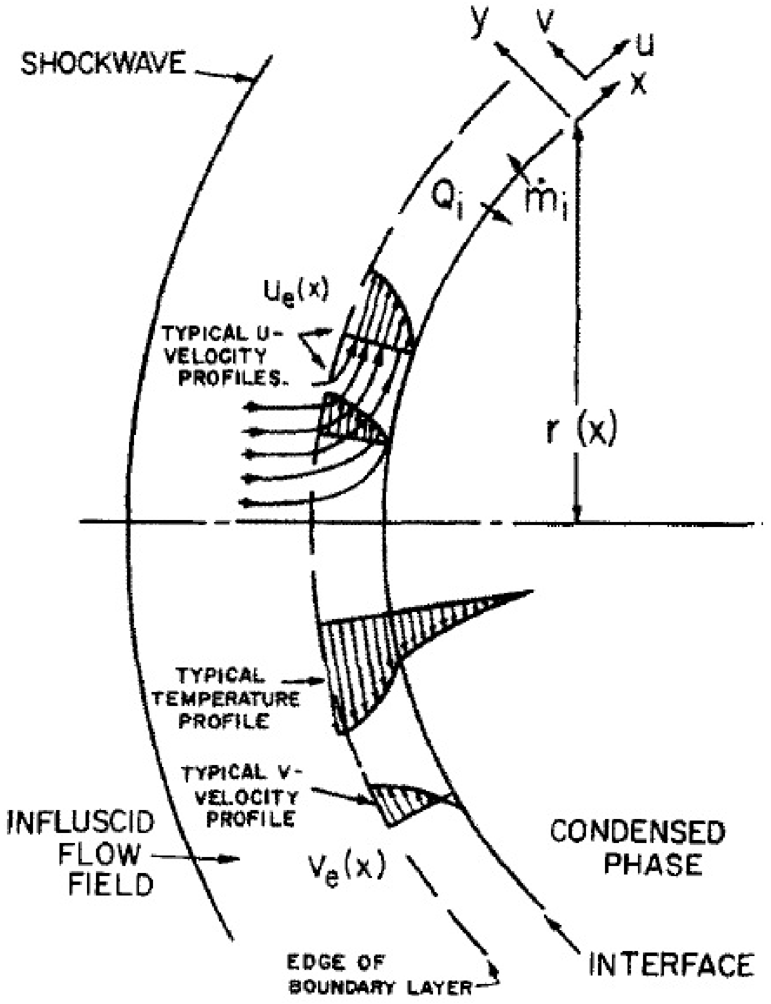


Figure 2.1: Depiction of the flow and coordinate system at the axi-symmetric, stagnation point. Adopted from [85]

Once the governing boundary layer equations have been transformed into a system of ordinary differential equations, similar solutions for the boundary layer can be obtained. In the case of axi-symmetric flow as shown in Figure 2.1, however, similarity transformation does not completely deduce the boundary layer equations to a set of ordinary differential equations. As mentioned, a similarity solution is obtainable if and only if the transformed equations are ordinary differential equations of one similarity variable η. It can be shown that only under the special case of the stagnation point, transformation results in a complete set of ordinary differential equations. This is because the change in pressure at the edge of boundary layer is a function of distance away from the stagnation point. In the problem of flat plate, for example, set of ordinary differential equations are obtainable for the whole plate since the free stream pressure is a constant. For a stagnation point analysis, it is assumed that

$$x \approx r \tag{2.23}$$

such that the angle between the axis of symmetry and radial vector to a position on the surface is very small. It can be further assumed that the flow velocity near the stagnation point is nearly zero and the viscous heating is negligible.

Furthermore, the inviscid flow velocity behind a shockwave, or the velocity at the edge of viscous boundary layer, can be determined by considering an infinitesimal increment of surface distance from the stagnation point such that

$$u_e = x \left(\frac{du_e}{dx} \right)_{x=0} \tag{2.24}$$

The validity of an assumption that the velocity at the boundary layer edge is linearly proportional along the axi-symmetric surface [3] has been demonstrated. Figure 2.2 illustrates the normalized velocity as a function of distance from the stagnation point for an axi-symmetric cone with a spherical nose calculated from more advanced CFD program. The proportionality constant in Equation (2.24) can be related to the pressure distribution using Euler

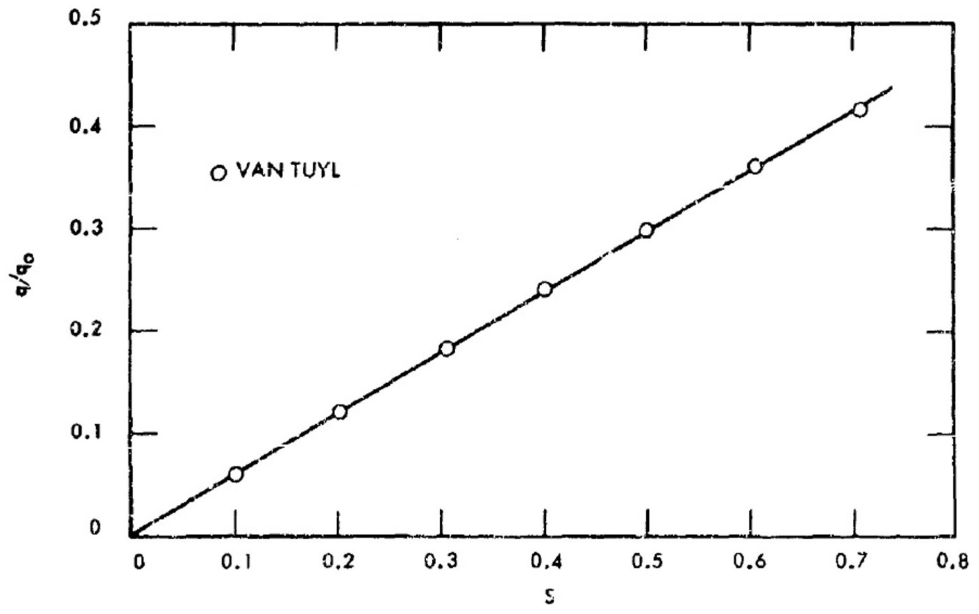


Figure 2.2: Velocity distribution for cone with sphere nose at Mach 8 compared with analytical value obtained by Van Tuyl [97]. q/q_o is the normalized velocity and S is the normalized distance from the stagnation point. Adopted from [3].

equation for inviscid flow

$$dP_e = \rho_e u_e du_e \tag{2.25}$$

The pressure distribution over the body is determined using the Newtonian model. This derivation is not accurate at all for low speed and incompressible flows because it does not capture the no-slip condition on the surface. However, at extreme hypersonic speeds, the shock waves are very thin and almost attached to the surface, and the flow behaves as described in the Newtonian model [3]. Using Newton's second law, it can be shown that

$$C_p = 2cos^2(\phi) \tag{2.26}$$

where angle ϕ is the angle between the axis of symmetry and radial vector to a position on the surface. Subsequent modification of the Newtonian model made by Lees [57] where the

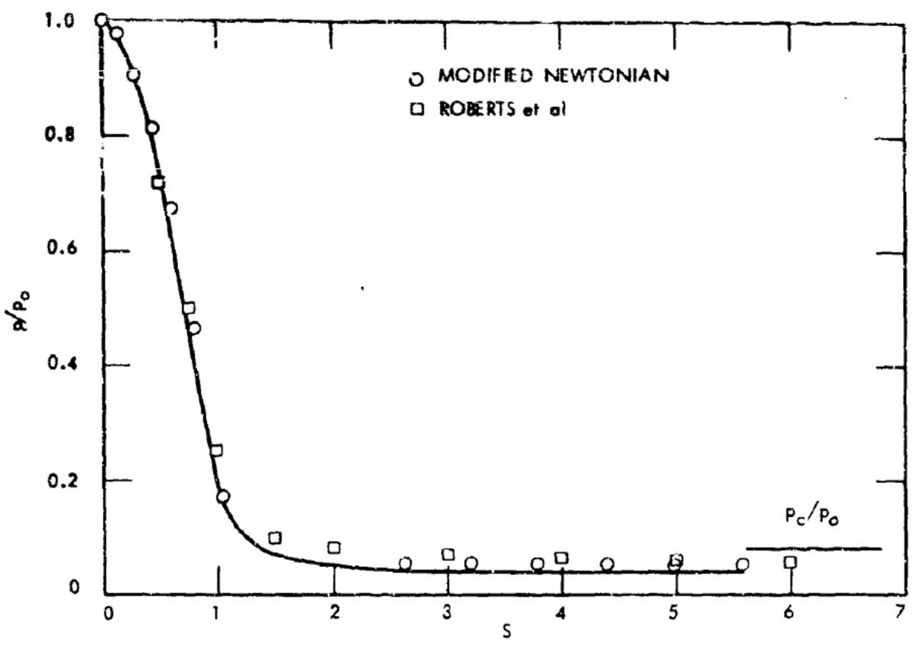


Figure 2.3: Pressure distribution for cone with sphere nose at Mach 8 compared with experimental data obtained by Roberts et al. [78]. p/p_o is the normalized pressure and S is the normalized distance from the stagnation point. Adopted from [3]

coefficient 2 is replaced by $C_{p,max}$ defined as

$$C_{p,max} = \frac{P_o - P_\infty}{\frac{1}{2}\rho_\infty V_\infty^2} \tag{2.27}$$

predicts the pressure distribution over the body with greater accuracy. Figure. 2.3 illustrates the accuracy of modified Newtonian model for pressure distribution when compared to more recent CFD calculation [3]. Using Equations (2.25) and (2.27), the proportionality constant can be written as

$$\left(\frac{du_e}{dx}\right)_{x=0} = \frac{1}{R_B}\sqrt{C_{p,max}\frac{P_e - P_\infty}{\rho_e}} \tag{2.28}$$

At hypersonic speeds, dissociation is an important aspect of the flow that must be considered. However, we assume the flow within the boundary layer is frozen such that the reaction rate is nearly zero or the characteristic time, of chemical reaction is much larger than that of advection of species. This is one extreme case of chemically reacting, non-equilibrium flow that simplifies the problem. The other extreme is in the case of infinite reaction rate

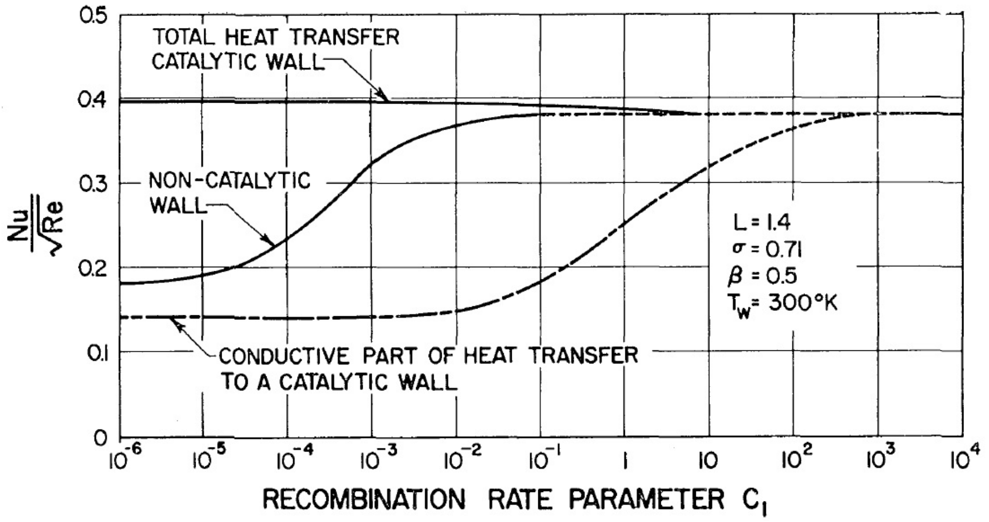


Figure 2.4: Total heat transfer rate as a function of chemical reaction rate C_1. Adopted from Ref. [21]. Total heat transfer rate at the surface is proportional to $Nu/\sqrt{Re}$.

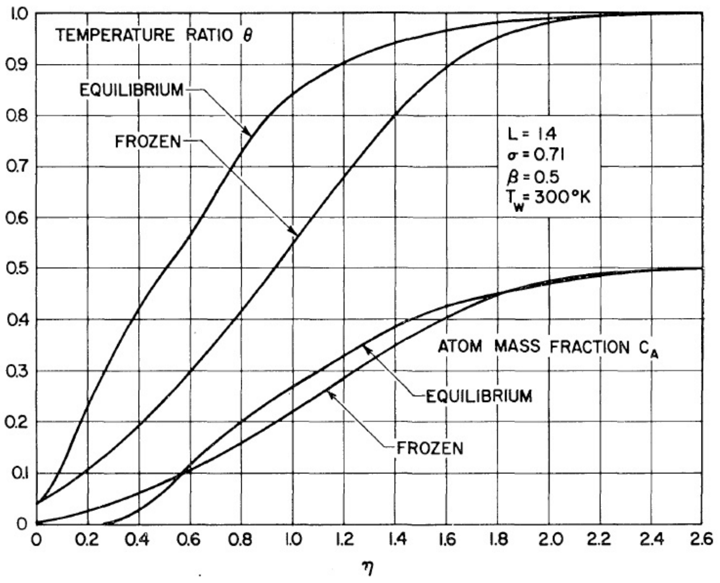


Figure 2.5: Boundary layer profiles at stagnation point. Adopted from Ref. [21]

which implies that local chemical equilibrium is reached at any point in the flow. However, as demonstrated from by Fay and Riddell in Figure 2.4, the total rate of heat transfer at the surface for frozen ($C_1 \to 0$) and local equilibrium flow ($C_1 \to \infty$) does not vary significantly given that the surface is considered to be fully catalytic. Similarly, the mass diffusion profile for both frozen and local equilibrium flow does not change noticeably as illustrated in Figure 2.5. Hence, frozen flow and fully catalytic surface assumptions provide a good approximation to examine the heat and mass transfer characteristics of in the boundary layer.

At the stagnation point for frozen flow, therefore, the transformed boundary layer equations become:

Conservation of momentum:

$$(lf'')' + ff'' + \frac{1}{2}\left(\frac{\rho_e}{\rho} - f'^2\right) = 0 \tag{2.29}$$

Conservation of energy:

$$\left(\frac{l}{Pr}c_p\theta'\right)' + fc_p\theta' - \frac{l}{Pr}\frac{\theta'}{M^2}\sum_i c_{p,i}M_i\left(\sum_{j\neq i} M_i Le_{ij} X'_j\right) = 0 \tag{2.30}$$

Conservation of species i:

$$\frac{M_i}{M}f\left(\frac{M'}{M}X_i - X'_i\right) + \left(\frac{l}{Pr}\frac{1}{M^2}\sum_{j\neq i} M_i M_j Le_{ij} X'_j\right)' = 0 \tag{2.31}$$

Equations (2.29-2.31) are a set of ordinary differential equations in terms of the similarity variable η that governs the hypersonic boundary layer. It can be solved using accessible ordinary differential equation solver given appropriate boundary conditions.

2.1.2 Boundary Conditions

Typical hypersonic flows over a sharp leading edge have high Reynolds number in which the shock distance from the surface is much larger than the boundary layer thickness. As discussed by Anderson et al. and Fay and Riddell [3, 21], the viscous boundary layer thickness is inversely proportional to Reynolds number whereas the shock distance is only a function of Mach number. Therefore, we assume the flow condition at the edge of the viscous boundary layer to be in chemical equilibrium. We use the normal shock properties summarized by Wittliff and Curtis [107] for air at chemical equilibrium to calculate the boundary conditions at various altitudes and flight speeds. To calculate the equilibrium concentration of air species, we use the tabulated results by Hilsenrath and Klein [34].

When evaporation occurs at the surface of the leading edge, additional species must be considered in the governing equations and the flow model. For simplicity, the gas mixture in this problem is considered to be a ternary mixture consisting of molecular air, dissociated atomic air, and vaporized species. For ternary mixture, two equations of conservation of species are required and one overall mass continuity equation. As a result, the problem becomes a boundary value problem with a set of nine ordinary differential equations requiring nine appropriate boundary conditions to fully close the formulation. The boundary conditions for frozen flow near the stagnation point are

$$\begin{array}{llll}
\eta = 0: & f = f_s & \eta \to \infty: & f' = 1 \\
 & f' = 0 & & \theta = 1 \\
 & \theta = \theta_s & & C_A = C_{A,e} \\
 & C_A = 0 & & C_K = 0 \\
 & C_K = C_{K,s} & &
\end{array}$$

In order to solve the boundary value problem using numerical ordinary differential equation solver, shooting method [6] is utilized to reduce the problem into an initial value problem. Boundary conditions that must be satisfied at $\eta \to \infty$ are converted to initial conditions such that the set of initial conditions are

$$\begin{array}{lll}
\eta = 0: & f = f_s & f'' = x_1 \\
 & f' = 0 & \theta' = x_2 \\
 & \theta = \theta_s & C_A' = x_3 \\
 & C_A = 0 & C_K' = x_4 \\
 & C_K = C_{K,s} &
\end{array}$$

Initial values, x_1, x_2, x_3, and x_4 are iterated until the boundary conditions at $\eta \rightarrow \infty$ is satisfied.

Lastly, the non dimensional stream function at the wall can be related to the mass transfer rate teat the wall by definition:

$$\dot{m}_s = -f_s \sqrt{2 \frac{\rho_s}{\mu_s} \left(\frac{du_e}{dx} \right)_{x=0}} \tag{2.32}$$

Mass transfer rate at the wall can also be expressed in terms of the sum of absolute fluxes of each component. For a ternary mixture considered in this problem

$$\dot{m}_s = (\rho_A V_A + \rho_M V_M + \rho_K V_K)_s \tag{2.33}$$

Under the assumption that the surface is fully catalytic and is impermeable to molecular and atomic air, mass balance must hold true such that

$$(\rho_A V_A)_s = -(\rho_M V_M)_s \tag{2.34}$$

Thus, the mass transfer rate at the wall simplifies to

$$\dot{m}_s = (\rho_k V_K)_s \tag{2.35}$$

and in terms of diffusive flux of vaporizing species K, it can be written as

$$\dot{m}_s = \frac{j_{K,s}}{1 - C_{K,s}} \tag{2.36}$$

because, by definition, the diffusive flux is related to the absolute flux by

$$\dot{m}_{K,s} = C_{K,s} \dot{m}_s + j_{K,s} \tag{2.37}$$

Consequently, the non-dimensional stream function at the wall requires either the mass transfer rate or the mass fraction of vaporizing species specified as a known quantity in order to obtain the solution. As a result, we impose two additional surface constraints based on conservation of energy and thermodynamics. First, we impose that the influx of aerodynamic heating is equal to evaporative heat flux at the stagnation point:

$$k_T \frac{\partial T}{\partial y} + \dot{m}_A \left(h_M - h_A\right) = \sigma \epsilon T^4 + \dot{m}_K \frac{L}{M_K} \tag{2.38}$$

where k_T is the thermal conductivity, h_M and h_A are heat of formation of molecular and atomic air, respectively, and h_{fg} is the latent heat of vaporization. Second, we use the Hertz-Knudsen equation [4, 66, 18] derived from Kinetic theory to model the evaporation process under thermodynamic non-equilibrium condition. For evaporating species i, the evaporation rate at the surface is:

$$\dot{m} = \alpha \sqrt{\frac{M}{2\pi R T}} P_s \left(X_{eq} - X\right) \tag{2.39}$$

where the superscript * refers to the saturation condition at the stagnation point, and X_s^* is calculated by the integral form of Clausius-Clapeyron equation:

$$X_{eq} = \frac{P^*}{P_s} \mathrm{exp} \left[\frac{L}{R}\left(\frac{1}{T^*} - \frac{1}{T_s}\right)\right] \tag{2.40}$$

where the superscript * denotes the saturation state at the standard pressure of 1 atm, R is the specific gas constant. The accommodation coefficient α represents the impact of molecular collisions at the interface on the rate of evaporation. We assume α to be equal to one which is typical for most liquids [85]. We also assume that the latent heat of evaporation L is a constant at the saturation condition at the standard pressure. By incorporating the Hertz-Knudsen equation, the evaporation rate at the surface depends on temperature and the mole fraction which are related to the adjacent hypersonic flow conditions.

2.1.3 Transport and Thermodynamic Properties

Material properties are estimated using theoretical derivations to the best ability. Theoretical models presented by Hirschfelder et al. [35] are primarily used to calculate the transport properties. Transport properties of molecular air are calculated using Lennard-Jones 6:12 potential model. Properties of atomic air, because of their monatomic structure, are estimated using simple rigid sphere Kinetic theory [35]. Molecular constants used for molecular and atomic air are presented in Table 2.1.

Table 2.1: Molecular constants for molecular and atomic air

Species	$\sigma(A)$	$M(g/mol)$	ϵ/k
Molecular air	3.557	29	113
Atomic air	2.92	14.5	-

Similarly, the properties of vaporizing species are also determined using simple kinetic theory due to lack of understanding of molecular constants and chemistry of vaporized refractory metal oxides available. Collision diameter is obtained through crude linear regression assuming that it is linearly proportional to the molar mass. Again, it is recognized that simple, rigid sphere model provides adequate estimation of material properties and should not affect the results significantly. We also assume that the diffusivity of each species is equal to the mixture diffusivity determined by a constant Schmidt number of 0.5 [64]. We then calculate the mixture viscosity and thermal conductivity using Wilke's semi-empirical relationships [106].

We consider a range of constant latent heats of evaporation from 100 kJ/mol to 800 kJ/mol, molar masses from 10 g/mol to 200 g/mol, and T^* from 1500 K to 3500 K. This range encompasses most oxides, as shown by the dashed area in Figure 2.7.

To validate the boundary layer theory results, we a set of seven different hypothetical materials, marked with symbols in Figure 2.7, using direct numerical simulation (DNS). The DNS model utilizes a 3$^{\text{rd}}$-order finite difference scheme with a shock-fitting algorithm.

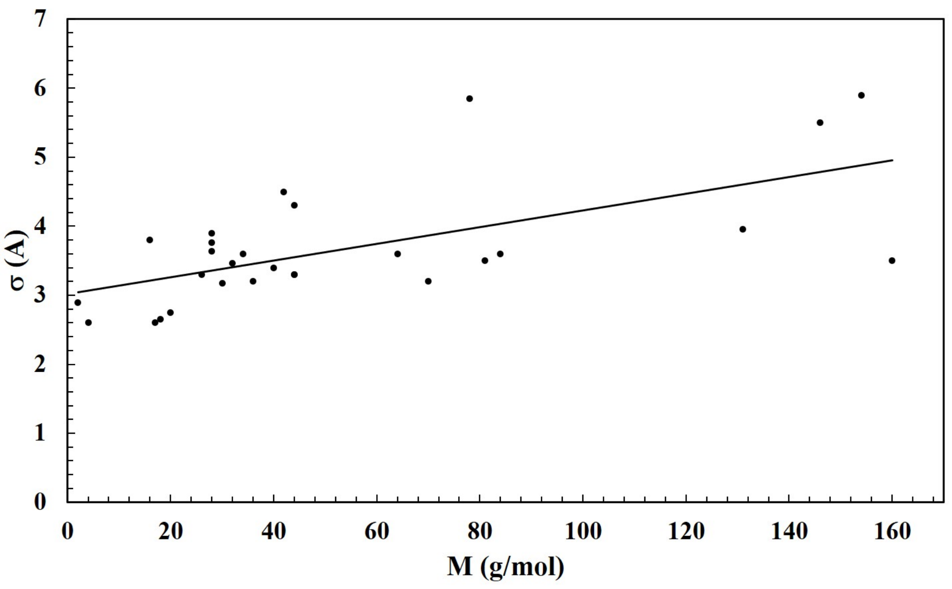


Figure 2.6: Kinetic diameter of typical gases obtained from Ref. [10]. Linear regression is used to relate the Kinetic diameter to molar mass.

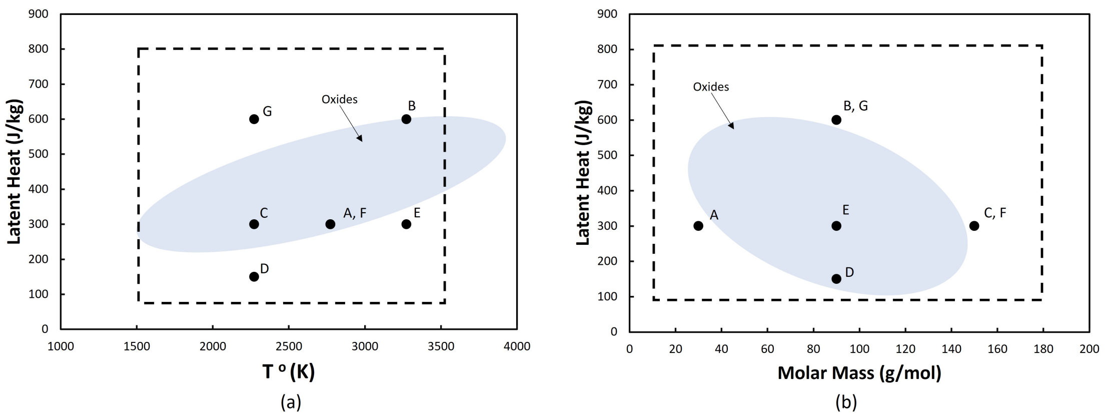


Figure 2.7: 3 Ranges of the material properties considered in our models. (a) latent heat vs. T^* (b) latent heat vs. molar mass. Values within the dashed rectangles are used to perform parametric study using the boundary layer model. Hypothetical materials marked as symbols are used in DNS. Material properties of typical oxides are obtained from [29]

Details of the model can be found in [65, 64, 66]. The properties of these seven materials are tabulated in Table 2.2.

Table 2.2: Material used in DNS to emulate the properties of typical metals and oxides.

Material	L [kJ/mol]	M [g/mol]	T_{bp} [K]
Material A	300	30	2773
Material B	600	90	3273
Material C	300	150	2273
Material D	150	90	2273
Material E	300	90	3273
Material F	300	150	2773
Material G	600	90	2273

2.2 Parametric Analyses for Frozen Flows

We now present the effect of material properties, flight conditions, and leading edge radii on evaporative transpiration cooling. We quantify the performance of the TPS using three metrics: surface temperature, evaporative mass flux, and boiling limit factor of safety. The operating surface temperature of a TPS sets the limit on the materials that can be used to construct the leading edge skin or the porous medium. A small evaporative mass flux may be desired to reduce the risk of dry-out at the surface and lower the cost and weight of TPS. The temperature within the porous leading edge must not exceed the saturation temperature of the coolant at the stagnation pressure to avoid nucleation that could block the flow and disrupt continuous coolant delivery to the surface. To assess this, we define the boiling limit factor of safety, F_s, as the ratio between the saturation temperature and the surface temperature at the stagnation point, T_s:

$$F_s = \frac{T_{\text{eq}}}{T_s} \tag{2.41}$$

T_{eq} is calculated using the Clausius-Clapeyron equation (equation 2.40) rearranged as follows:

$$T_{\text{eq}} = \left[-\frac{R}{L} \ln\left(\frac{P_s}{P^*}\right) + \frac{1}{T^*} \right]^{-1} \tag{2.42}$$

When the value of F_s is equal to or less than one, vapor nucleation within the porous leading edge can occur and disrupt the coolant flow to the surface.

2.2.1 Material Properties

We first discuss the effect of the coolant material properties: the latent heat of evaporation, molar mass, T^* of the coolant. These three material properties govern the heat and mass transfer behavior at the surface (equations 2.39 and 2.40). We vary the latent heat of evaporation from 100 kJ/mol to 800 kJ/mol, the molar mass from 10 g/mol to 200 g/mol, and T^* from 1500 K to 3500 K to encompass the properties of typical oxides. To isolate the effect of variation in the material properties, we fix the altitude, the speed, and the leading edge radius as constants at 30 km, Mach 15, and 3.1 mm, respectively, here. Figures 2.8 through 2.13 illustrate the surface temperature, evaporative mass flux, and F_s as a function of the three material properties. The DNS results are also shown as the symbols for comparison.

shows the relationship between the surface temperature of the leading edge and the three material properties. We find that the surface temperature is relatively independent of the latent heat and the molar mass. One might expect from equations 2.39 and 2.40 that a higher latent heat would lower the surface temperature since one can achieve a higher evaporative heat flux for a given evaporative mass flux. However, as shown in Figure 2.9, higher latent heat leads to lower mole fraction of evaporating species at the surface. A lower mole fraction at the surface reduces the vapor shielding effect, which thereby leads to an increase in the incident heat flux as shown in Figure 2.10. This increase in the incident heat flux requires higher evaporative heat flux, and thereby, higher surface temperature. These two opposing effects counteract each other, and the latent heat appears to have negligible impact on the surface temperature. Similar to the case for the latent heat, one might expect a lower molar mass would lower the surface temperature since the evaporative heat flux is proportional to $M^{-1/2}$. However, as shown in Figure 2.11, the incident heat flux increases as molar mass

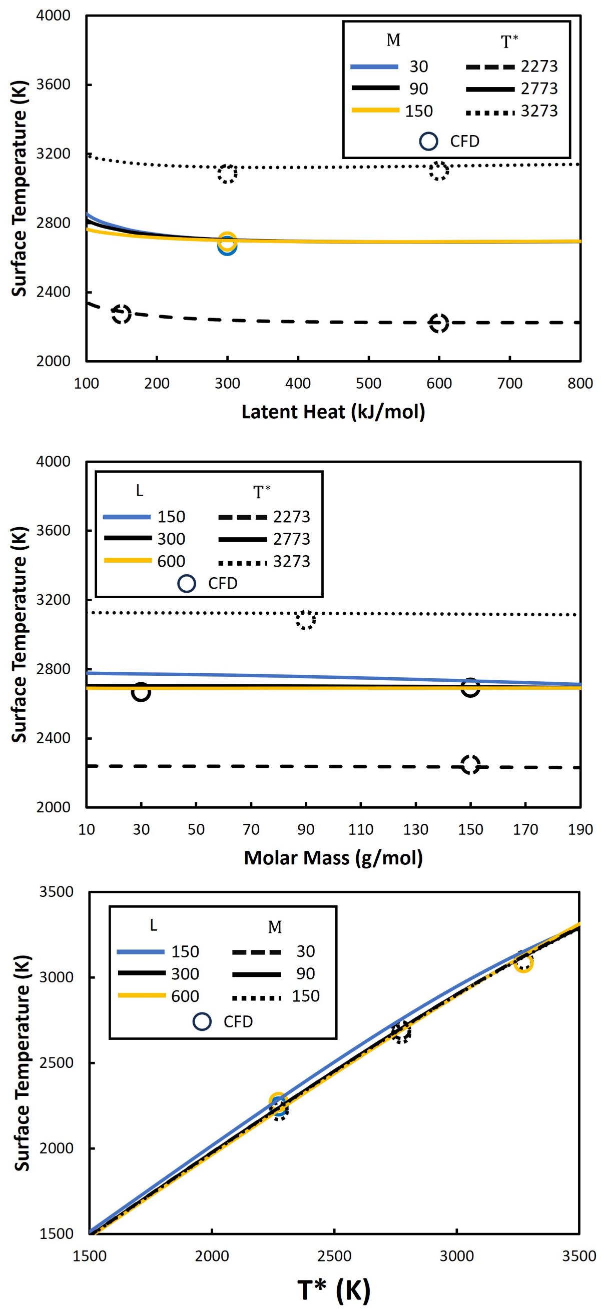


Figure 2.8: Predicted surface temperature as a function of the (a) latent heat of evaporation, (b) molar mass, and (c) T^*. Line colors and dash types represent the variation in material properties not shown in the axis. Symbols are obtained from DNS and have less than 2% difference with the boundary layer model results.

decrease because heavier species enhance the thermal shielding effect since the diffusivity of gas molecules is proportional to $M^{-1/2}$. This increase leads to higher evaporative mass flux, and thus, higher surface temperature. Again, these two opposing behaviors make the surface

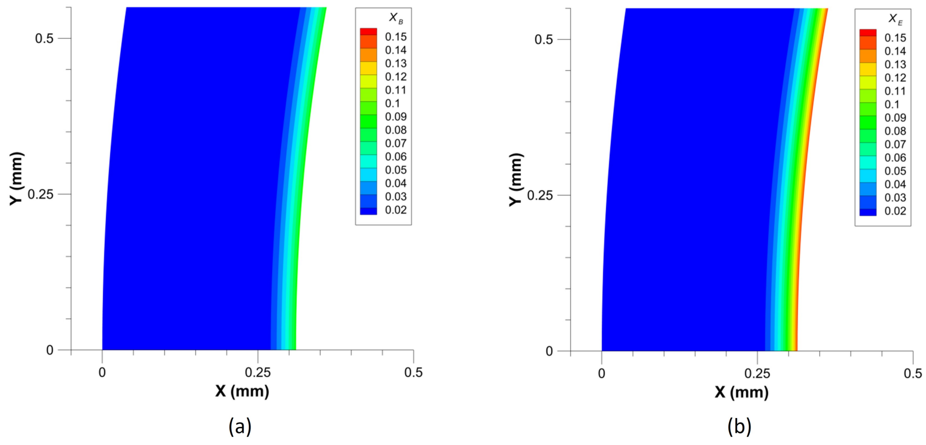


Figure 2.9: Predicted color contour of mole fraction for (a) material B and (b) material E. Material B has higher latent heat than material E. The left most boundary of the contour is immediately behind the shock, and the right most boundary of the contour is the leading edge surface. X is the distance from the shock, and Y is the distance from the stagnation line.

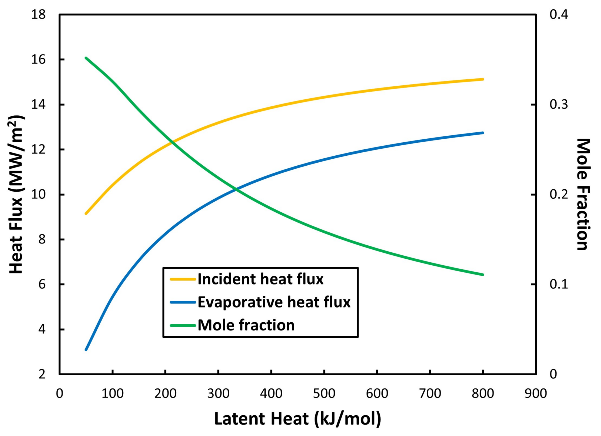


Figure 2.10: Predicted incident and evaporative heat flux at the surface as a function of the latent heat. Molar mass and T^* of the material is 90 g/mol and 2773 K, respectively.

temperature nearly independent of the molar mass. The predicted surface temperature in contrast is proportional to T^* of the coolant material. Because the mass transfer rate is proportional to $\exp^{(1/T^* - 1/T_s)}$, the surface temperature must be very close to T^* to have meaningful mass transfer rate.

These relationships suggest that T^* of the coolant material has a dominant effect on the operating surface temperature of the TPS. The porous media comprising the leading edges

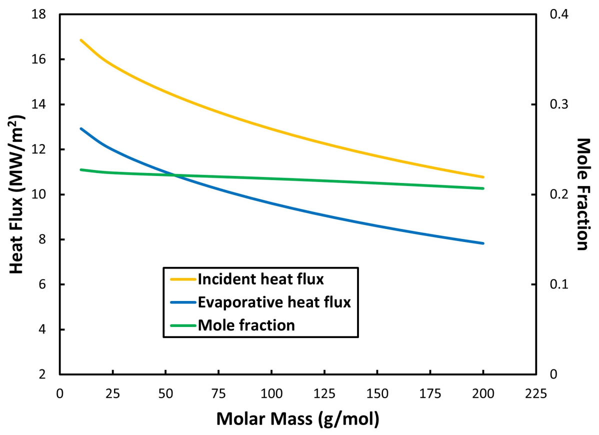


Figure 2.11: Predicted incident and evaporative heat flux at the surface as a function of the molar mass. Latent heat and T^* of the material is 300 kJ/mol and 2773 K, respectively.

must be able to tolerate temperature rises to at least T^* under a given flight condition.

Figure 2.12 shows the relationship between the evaporative mass flux and the three material properties. We find that evaporative mass flux is influenced by all three properties. A higher latent heat or a lower molar mass increases the specific latent heat of the material, reducing the mass transfer rate needed to absorb a given amount of incident energy. Although a higher latent heat and a lower molar mass increase the incident heat flux by approximately 50%, as shown in Figures 2.10 and 2.11, the change in the specific latent heat is greater (a factor of 8 to 10), resulting in lower evaporative mass fluxes. The evaporative mass flux also slightly decreases as T^* increases. Since the surface temperature is proportional to T^*, higher T^* leads to a lower incident heat flux by reducing the temperature gradient and promoting thermal radiation at the surface. Hence, the coolant material with a higher specific heat and a higher T^* is desired to minimize the required evaporative heat flux of the TPS. However, increasing the latent heat beyond approximately 500 kJ/mol gives a diminishing return in reducing the evaporative mass flux due to increase in the incident heat flux at the surface as shown in Figure 2.10.

Figure 2.13 shows the relationship between F_s and the three material properties. We find that F_s is nearly independent of the molar mass since molar mass does not significantly affect the surface temperature or the saturation condition as illustrated in Figure 2.8 and equation

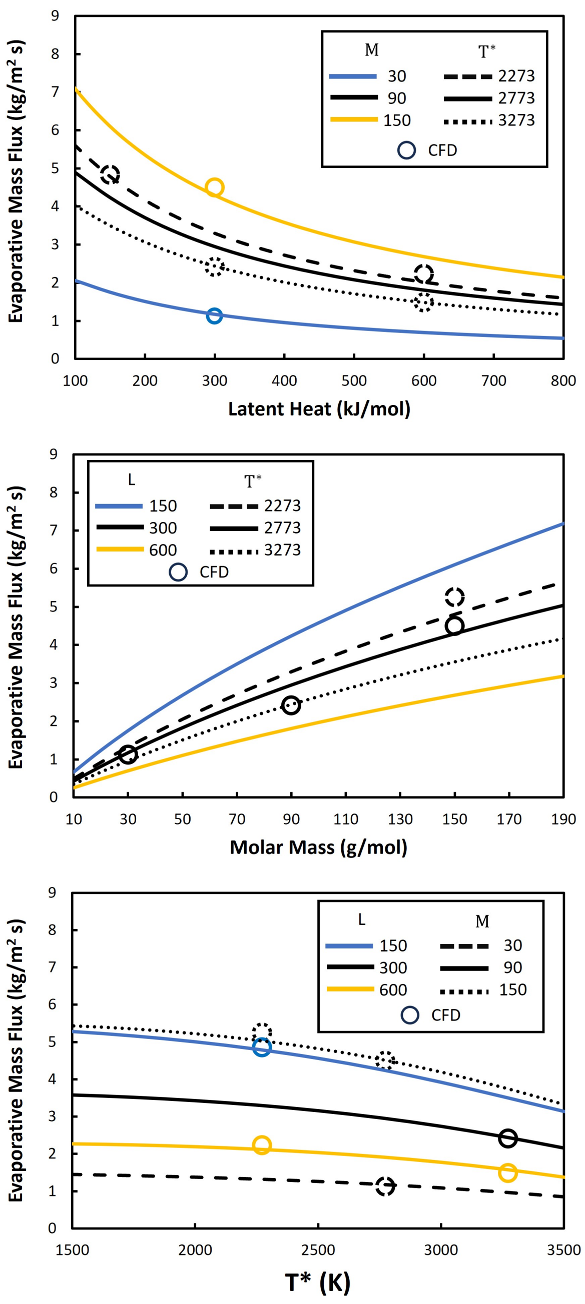


Figure 2.12: Predicted evaporative mass flux as a function of the (a) latent heat of evaporation, (b) molar mass, and (c) T^*. Line colors and dash types represent the variation in material properties not shown in the axis. Symbols are obtained from DNS and have less than 8% difference with the boundary layer model results.

2.42. The surface temperature is also nearly independent of the latent heat; however, a higher latent heat decreases T_{eq} as given by equation 2.42 leading to an increase in F_s. We also find that higher T^* leads to an increase in F_s. Surface temperature, which is the numerator of F_s,

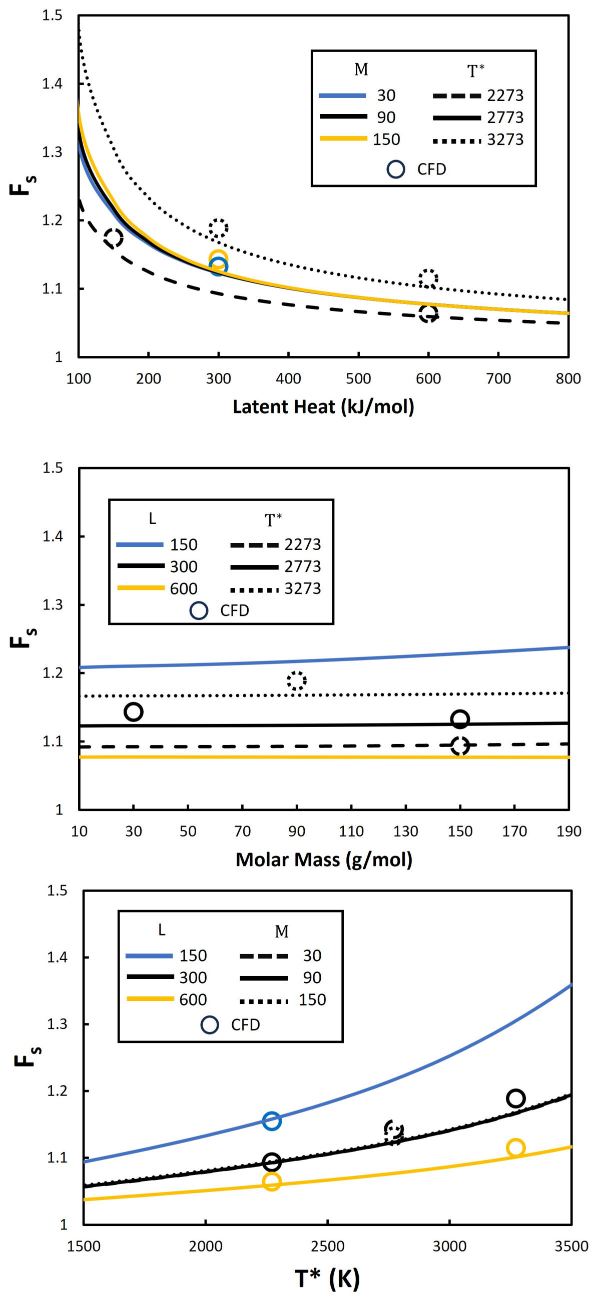


Figure 2.13: Predicted F_s as a function of the (a) latent heat of evaporation, (b) molar mass, (c) and T^*. Line colors and dash types represent the variation in material properties not shown in the axis. Symbols are obtained from DNS and have less than 2% difference with the boundary layer model results.

is approximately proportional to T^*, as shown in Figure 2.8. However, the denominator of F_s is proportional to $T^*/(1-T^*)$ as given by equation 2.42. Because the magnitude T_{eq} of grows faster than T_s, the factor of safety increases as T^* increases. From these relationships, a coolant material with a lower latent heat and a higher T^* may be preferable to reduce the risk of nucleation of vapor bubbles within the porous leading edge.

From the viewpoint of leading edge engineering, it may be desirable to reduce the peak temperature to avoid thermomechanical failures, to reduce the evaporative heat flux to decrease the cost and weight of the coolant, and to increase F_s to reduce the risk of surface dry out. However, the rather complex relationships presented above do not lend themselves readily to an optimal set of the material properties that satisfy all three requirements. For example, TPS utilizing a coolant with a lower T^* would achieve a low operating surface temperature of the TPS, but also result in a low F_s. TPS utilizing a coolant with a higher latent heat of evaporation would achieve a lower evaporative flux, but also a lower F_s.

2.2.2 Flight Conditions

One additional set of parameters one must account for in selecting a coolant material is the vehicle flight trajectory. We assess a range of flight conditions to examine their effects on the performance of the TPS and thereby guide the selection of the coolant properties. We consider flight altitudes from 20 km to 40 km and Mach numbers from 10 to 20. We choose these ranges to emulate the flight trajectories of typical hypersonic vehicles. Figure 2.14 shows altitudes and Mach numbers of typical hypersonic flight trajectories and the ranges that we considered.

In Figure 2.15, we show the surface temperature, evaporative mass flux, and F_s as a function of the altitude and Mach number to elucidate the relationship among them. The solid lines illustrate the boundary layer model results, and the symbols represent the DNS results. We consider a leading edge with a fixed radius of 3.1 mm. From our results presented in the previous section, we note that the predicted evaporative flux showed the largest differences between the two models. We therefore consider material D, which has the lowest specific latent heat among the seven materials, to help compare the two models for a wide range of evaporative mass fluxes at the surface. The maximum difference between the two models is

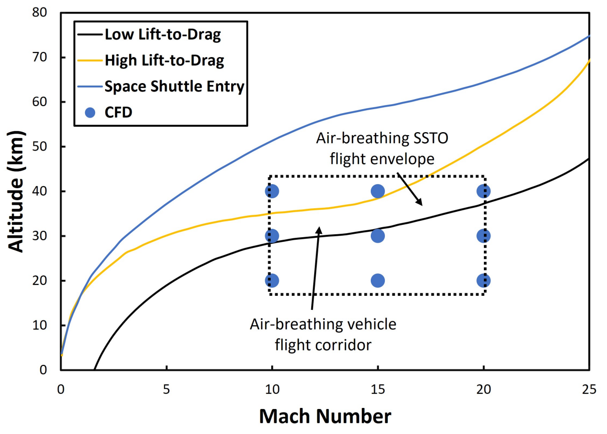


Figure 2.14: The velocity-altitude map showing the trajectories of typical hypersonic vehicles and a range of flight conditions used in this study. The range within the dashed rectangles are studied using the boundary layer model and the discrete conditions shown in symbols using DNS. The flight trajectories of typical hypersonic vehicles are adopted from [3]

below 5% for the surface temperature and F_s and below 25% for the evaporative mass flux. At high Mach numbers, the severity of the aerothermal conditions is exacerbated due to the high stagnation enthalpy of the hypersonic flows. Figure 2.15 indeed shows higher surface temperature, higher evaporative mass flux, and lower F_s as the Mach number increases. A higher Mach number significantly increases the incident heat flux which correspondingly requires an increase in the evaporative mass flux to absorb the excess energy. As a result, the surface temperature increases, approaching T_{eq} and thereby decreasing F_s. We note, however, that the changes in the evaporation rate are significantly higher than the changes in the other two parameters. For the range of Mach numbers illustrated in Figure 2.15, the evaporative mass flux changes by an order of magnitude whereas the surface temperature changes less than 50% due to the exponential relationship, as given in equations 2.39 and 2.40. F_s experiences an even smaller change less than 15% due to the increase in both surface temperature and T_{eq} at higher Mach numbers. Therefore, for vehicle trajectories with high Mach numbers, coolant materials with high specific latent heats can significantly reduce the evaporative mass flux. However, higher specific latent heats would decrease F_s. The reduction in F_s may be acceptable since the decrease in F_s at higher Mach numbers is relatively

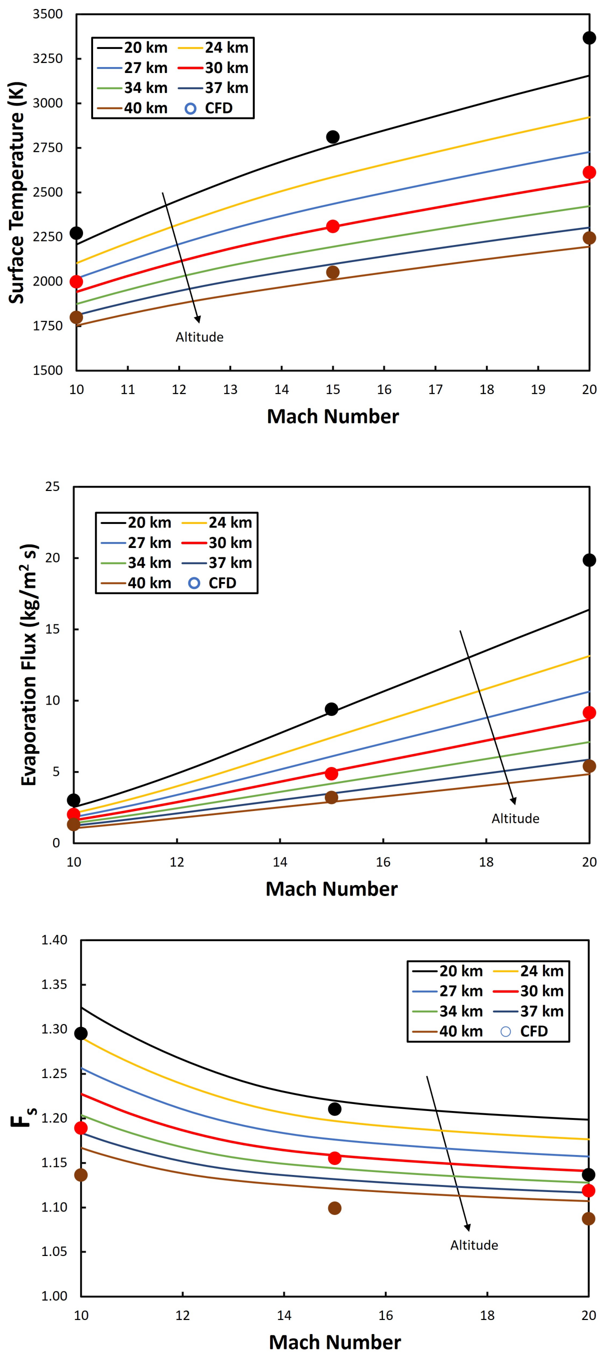


Figure 2.15: Predicted (a) surface temperature, (b) evaporative mass flux, and (c) F_s as a function of altitude and Mach number for material D and a 3.1 mm leading edge radius. Solid lines are obtained from boundary layer model and symbols from DNS.

small.

Conversely, an increase in the flight altitude partly alleviates the severity of the aerothermal condition due to lower densities. Figure 2.15 shows a decrease in the surface temperature and evaporative mass flux at higher altitudes. However, F_s decreases with increasing altitudes. This is because, at higher altitudes, the stagnation pressure lower, which reduces the T_{eq} of the coolant material. Similar to the results from varying Mach numbers, the evaporative mass flux changes significantly (by a factor of 3) compared to that in the temperature which varies less than 50%. F_s again experiences a smaller change below 15% due to the decrease in both the surface temperature and T_{eq} at higher altitudes. Because the relationship between F_s and the flight altitude is opposite of those for the surface temperature and the evaporative mass flux, different material properties would need to get more emphasis at different altitudes. For instance, a vehicle designed for high-altitude flight may utilize coolants with low specific latent heats and high T^* to increase F_s at the expense of higher surface temperatures and evaporative mass fluxes. Alternatively, a vehicle designed for low-altitude flights may utilize coolants with high latent heats and low T^* to decrease the surface temperature and evaporative mass flux at the expense of lower F_s, which is already relatively high at low altitudes.

2.2.3 Leading Edge Radius

Another parameter one must account for in selecting a coolant material is the leading edge radius. Sharp leading edges are essential in achieving superior aerodynamic performance and maneuverability. However, a sharper leading edge quadratically increases the rate of aerothermal heating [95] which can drastically change the performance of the TPS. Hence, we assess the performance of transpiration cooling TPS for sharp leading edges with radii from 5 mm to 0.1 mm and illustrate their impact on the surface temperature, evaporative mass flux, and F_s. We fix the flight condition at 30 km altitude and Mach 15 and use material D.

Figure 2.16 illustrates the relationship between the leading edge radius and the three performance parameters. Due to significant increase in the incident heat flux, sharper leading edges exacerbate the aerothermal heating. As a result, the surface temperature and the

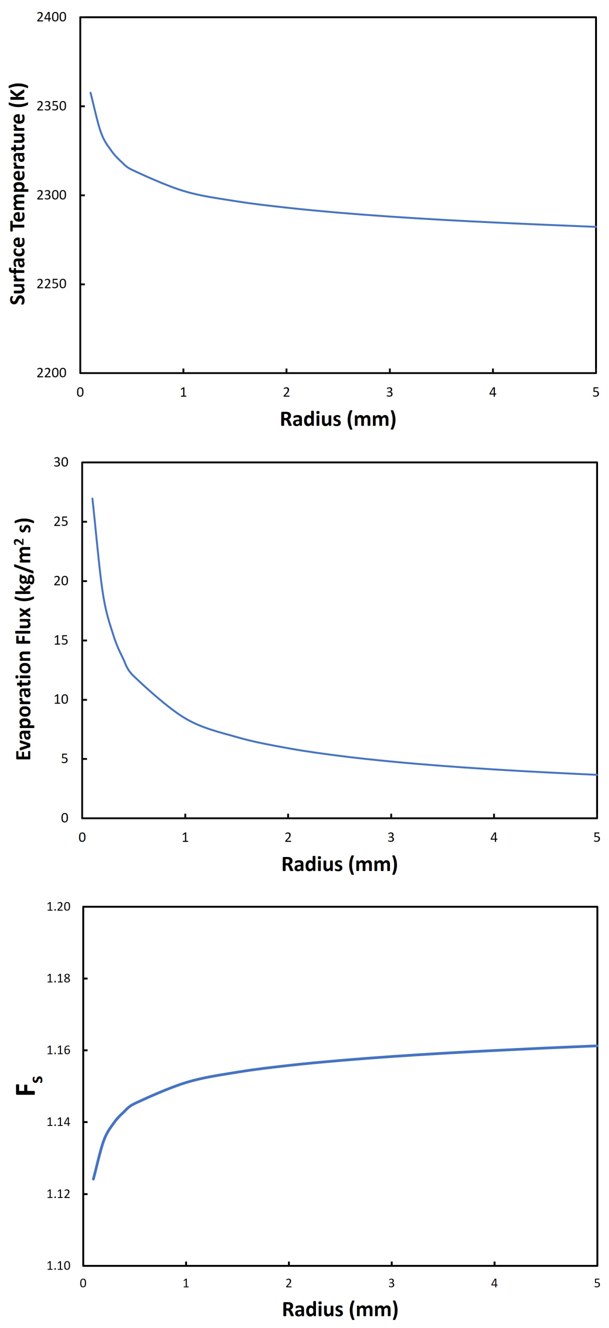


Figure 2.16: Predicted (a) surface temperature, (b) evaporative mass flux, and (c) F_s as a function of leading edge radius for 30 km altitude and Mach 15 using material D.

evaporative mass flux increase while F_s decreases. However, similar to the results in Section 2.2.2, the change in the evaporative mass flux is much greater compared with those in the surface temperature and F_s. Reducing the leading edge radius from 5 mm to 0.1 mm, the evaporation rate increases by a factor of 10, while the changes in the surface temperature and F_s are below 4%. This behavior is again due to the non-linear relationship between the surface temperature and the evaporative mass flux as given by equations 2.39 and 2.40. However, the changes in the surface temperature and F_s are much less compared with the results in Section 2.2.2 because the change in the leading edge radius does not affect the flow conditions at the stagnation point. Hence, the change in the surface temperature is only affected by the change in the evaporative mass flux, and the change in F_s is solely due to the change in the surface temperature. Because the change in surface temperature and F_s are almost negligible, the evaporative mass flux is the limiting factor of the TPS for very sharp leading edges. Hence, coolant materials with high specific latent heats would significantly reduce the evaporative mass flux at the expense of marginally higher surface temperature and F_s. We also note that evaporative transpiration cooling may be able to effectively protect very sharp leading edges with a radius of 0.1 mm as long as the necessary evaporative mass flux is achieved at the surface.

2.3 Summary and Conclusion

In this chapter, we investigated the viability of transpiration cooling employing oxide coolants as an alternative system to thermally protect sharp leading edges. We parametrically characterize the performance of transpiration cooling for various coolant properties, flight conditions, and leading edge radii using 2D axi-symmetric, semi-analytical boundary layer model incorporating thermodynamic non-equilibrium conditions which we validated with 3rd-order shock-fitting DNS. We quantified the performance of the TPS using three metrics: the surface temperature, the evaporative mass flux, and the boiling limit. We showed that the surface temperature depends solely on the saturation temperature of the coolant material, the boiling limit is independent of molar mass, and the evaporative mass flux is affected by all three material properties. We also illustrated that high Mach numbers and small leading edge sizes exacerbate the aerothermal condition resulting in higher surface temperature and evaporative mass flux and lower boiling limit factor of safety. Low altitude flights also increase the surface temperature and evaporative mass flux but result in higher boiling limit factor of safety.

Our numerical results demonstrated that transpiration cooling employing oxide coolants may effectively cool the surface temperature to below the saturation temperature of the coolant material even for very sharp leading edges with radius of 0.1 mm and in extreme hypersonic conditions of Mach 20 and 20 km altitude, where incident heat fluxes are of the order of 85 MJ/m2, as long as the necessary amount of coolant is supplied to the surface. Our results do not lend themselves readily to a single optimal set of the material properties for transpiration cooling. Rather, different coolant properties are better suited for different flight conditions and leading edge sizes. Vehicles with high Mach number and low altitude trajectories or small radius of curvature of the leading edge may utilize coolant materials with high specific latent heats to significantly reduce the evaporative mass flux. Alternatively, vehicles with low altitude trajectories may benefit from coolants with low specific latent heats to avoid vapor nucleation within the porous leading to which they are more susceptible at higher altitudes.

Chapter 3

Permeability of High Porosity Reticulated Foams

In order to effectively protect the sharp leading edges of hypersonic vehicles, the amount of liquid evaporated must be replenished to avoid dry out at the surface. Lack of evaporation caused by surface dry out could lead to failure by formation of local hot spots above melting point of porous structure. To achieve a net liquid flow to the surface, the driving pressure must be larger than the pressure loss through the porous medium:

$$P_D \geq \frac{\mu u_D}{K} L \tag{3.1}$$

The pressure gradient is proportional to the inverse of the permeability of the porous media. Hence, it is important to accurately know the permeability of porous media to accurately predict the viscous pressure loss.

3.1 Experimental Measurement

Here, we characterize the permeability of porous media by measuring the flow rate for a given pressure gradient. Using Equation 1.1, we estimate the permeability and inertial coefficient of the reticulated foams by least squares regression analysis.

3.1.1 Experimental Setup

We constructed the test facility, illustrated in Figure 3.1, using 3/4" diameter Polyvinyl chloride (PVC) pipes. The set up can be divided into three sections: inlet, sample hous-

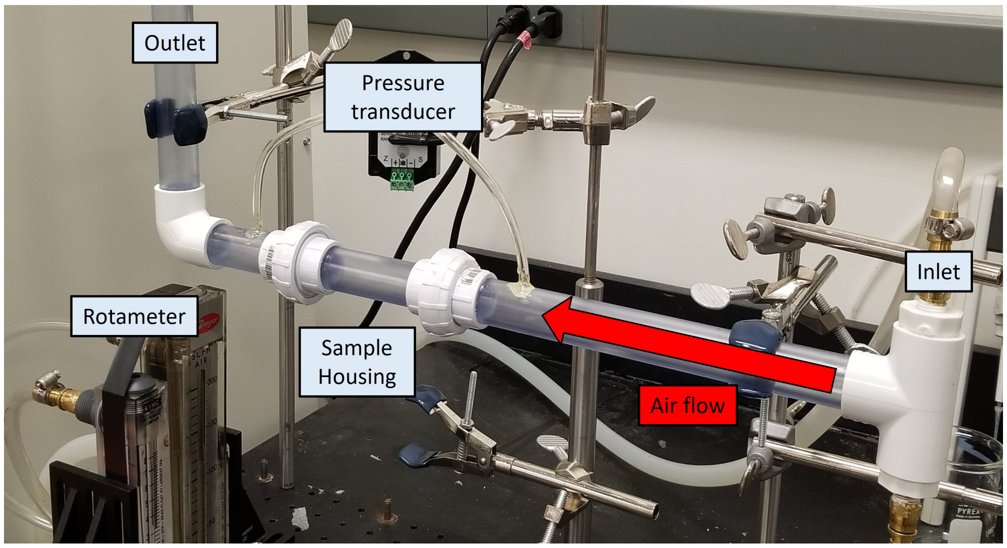


Figure 3.1: Experimental setup used to measure the pressure gradient across porous media samples at fixed flow rate.

ing, and outlet sections. The inlet section is connected to a pressurized air supply. The length of the inlet is sufficiently long to ensure fully developed flow. Sample housing section is connected to the inlet and outlet by threaded PVC pipe fittings with rubber o-rings. Threaded connection allows easy access to the samples foam and testing of different samples. Fluid is then discharged into ambient through the outlet. We control and measure the air flow rate using two rotameters (King Instrument Company Model 2C-02 and Dwyer Model RMC-103-SSV) with different full scales, 1.8 SCFH and 4 SCFH, respectively. Air flow rate is measured at the location upstream of the inlet. We use a differential pressure transducer (Dwyer 648C) to measure the pressure gradient at the beginning and end of the sample housing section. Two pressure taps are drilled into the PVC pipes, and they are connected with 1/8" diameter plastic tubing to the differential pressure transducer. Plastic tubing are secured using bonding adhesive to prevent leakage.

3.1.2 Sample Preparation

High porosity, reticulated silicon carbide (SiC) ceramic foams are provided by Ultramet. SiC Foams were manufactured using chemical vapor deposition (CVD) of SiC on to vitreous carbon preform constructed from pyrolyzing resin impregnated polyurethane foam. We use three samples with different pore densities: 80, 65, and 45 pores per inch (PPI). Foam samples are cut into cylinders with 3/4" diameter and 1" length as depicted in Figure 3.2. After cutting the samples, they are cleaned by vigorously shaking in acetone and water by

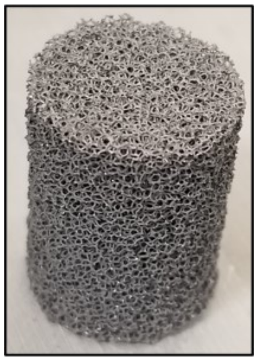


Figure 3.2: Image of prepared 3/4" x 1" cylindrical SiC reticulated foam sample. SiC foams were provided and manufactured by Ultramet using chemical vapor deposition onto preforms.

hand to clean off residual powders and dust. Samples are then dried with pressurized dry air. Finally, each sample is wrapped in Teflon tape around its perimeter to fill the small gaps between it and the pipe to prevent leakage around the sample.

3.1.3 Experimental Results

For each experimental run, flow rate is gradually increased until 40 standard cubic feet per hour (SCFH) to ensure laminar flow in the system. Figure 3.3 illustrates the measured differential pressure for a given flow velocity. The pressure gradient shows quadratic relationship with the flow velocity for all samples. For the range of pore Reynolds number in

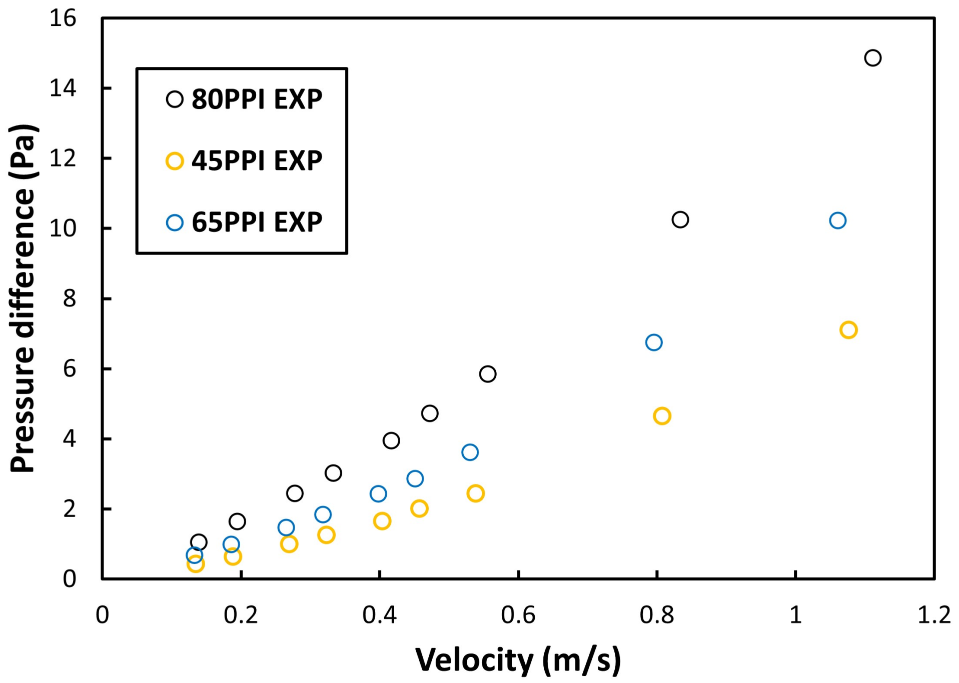


Figure 3.3: Measured differential pressure as a function of flow velocity for the three SiC samples. Quadratic relationship is observed in accordance with the Darcy-Forchheimer equation (equation 1.1)

our experiments, the flow through porous media is in inertial regime [49]. In this regime, the pressure gradient is related to the flow velocity by Darcy-Forchheimer equation (Equation 1.1, repeated here for convenience):

$$\frac{dP}{dx} = \frac{\mu u_D}{K} + \frac{\rho {u_D}^2}{C} \tag{3.2}$$

Here, the pressure gradient is proportional to the square of flow velocity which is in agreement with our experimental results. We perform least squares regression analysis to obtain the best quadratic fit to our experimental data and calculate permeability and inertial coefficient using the Darcy-Forchheimer equation. Obtained values of permeability and inertial coefficients are tabulated in Table 3.1. We also observe that higher pore density increases the flow resistance of the porous media which agrees with previous studies. This behavior is expected since both permeability and inertial coefficient are proportional to the pore diameter [77].

Table 3.1: Experimentally measured permeability and inertial coefficient

Pore density	Permeability, K (m^2)	Inertial Coefficient, C (m)
45 PPI	5.3 x 10^{-9}	2.4 x 10^{-4}
65 PPI	4.3 x 10^{-9}	7.3 x 10^{-4}
80 PPI	2.6 x 10^{-9}	1.9 x 10^{-4}

3.2 Numerical Simulations

With recent advances in imaging and digital reconstruction of porous media, one can obtain local, pore-scale details of flow through porous media using numerical simulations. Xu et al. [111] utilized CFD to solve the velocity field for a unit cell consisting of tetrakaidecahedron, a polygon with 14 faces, is often used to represent the structure of metal and ceramic foams. They compared their simulation with the experimental results from Lacroix et al. [55] and found relatively good agreement in the low Reynolds number regime. Their results showed larger deviation in the permeability at higher Reynolds number. Wu et al. also utilized tetrakaidecahedron geometry with blended surfaces at the ligament joints to create a smooth connection. Their model was able to adjust the radius of curvature of the blended surfaces at the ligament joints to manipulate the porosity and pore diameter of the foam. They also performed experiments on different ceramic foams but did not directly compare the two results. Although these numerical models predict the macroscopic characteristics reasonably, use of homogeneous porous media often experiences difficulty with a variety of reticulated foams who are highly heterogeneous with complex morphology.

For complex, heterogeneous reticulated foams, X-ray micro-computer tomography (mu-CT) images have been utilized to perform numerical simulations on the physical microstructures of the reticulated foams [69, 112, 113]. A digitally reconstructed 3D geometry from a series of 2D tomography images serves as the computational domain. Diani et al. [17] utilized X-ray μm-CT images to reconstruct reticulated copper foams with four different pore densities between 5 to 40 PPI. Their numerical simulation using digitally reconstructed foams demonstrated good agreement with experimental measurements obtained using air flow. Ranut et al. [74] incorporated similar methodology to numerically simulate aluminum foams with three different pore densities between 10 to 30 PPI to estimate the permeability. However, their results were not validated with experiments. Recently, Wu et al. [108] numerically simulated fluid and heat transport through a digitally reconstructed ceramic foam of 0.85 porosity and 30 PPI pore density. Numerically simulated pressure gradient and convective heat transfer rate agreed well with experimental results.

Numerical simulation provides advantages in estimating the macroscopic parameters, such as permeability and inertial coefficient, without conducting experiments which can be cumbersome. They can also provide details of the fluid flow which cannot be obtained experimentally to characterize the porous media flow at the microscopic scale. Here, we adopt a similar approach to numerically simulate the fluid flow through digitally reconstructed SiC reticulated foams from X-ray μ-CT images. We demonstrate the capability and accuracy of our numerical simulations by validating with experimental results.

3.2.1 Geometric Reconstruction

X-ray tomography is a non-invasive method that allows imaging and reconstruction of opaque materials. A sample is placed and rotated between an X-ray source and a detector. Projections at the detector are then taken at an incremental angle of rotation and is used to reconstruct into 2D planar images of the sample [74]. X-ray tomography images of SiC samples were obtained using SkyScan 1172, Bruker with a pixel resolution of 13.3 μm, and reconstruction of the X-ray images were performed using NRecon software (Bruker). Figure 3.4 shows the X-ray and reconstructed image for SiC sample. A set of 2D planar images were used to reconstruct a 3D binary array that consists of voids, "0", and solids, "1". 3D binary array is then converted to a 3D mesh using open source MATLAB script [2]. Converted 3D mesh consisted of approximately 50,000 triangle surfaces to represent the solid voxels in the 3D binary array. Figure 3.5 illustrates the reconstructed and decimated surface mesh for

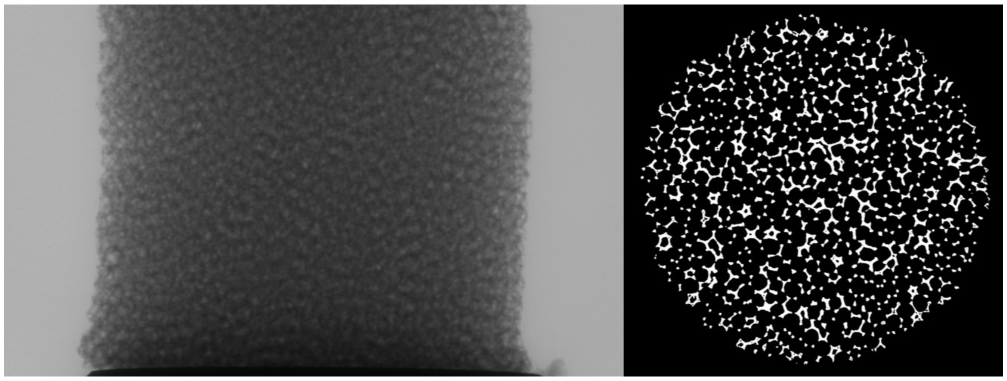


Figure 3.4: X-ray projection of the cylindrical SiC sample and 2D binary planer image

SiC foam.

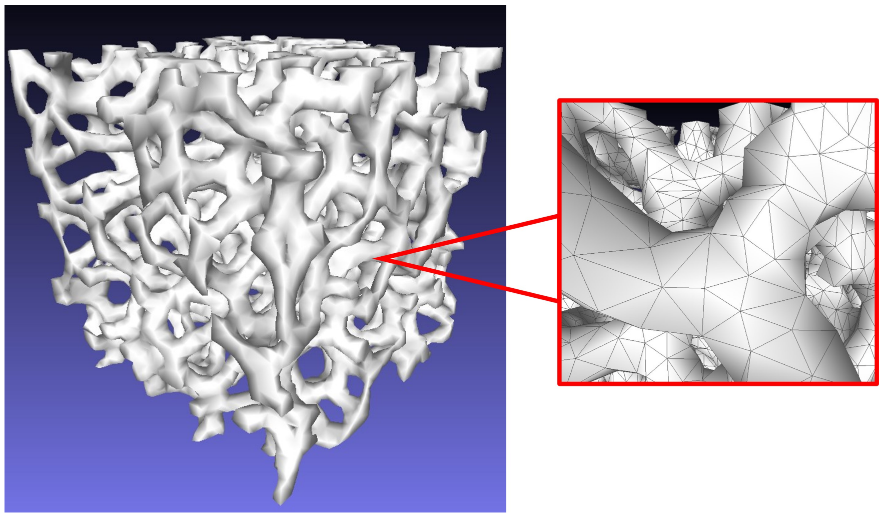


Figure 3.5: Reconstructed 3D surface mesh of SiC foam

3.2.2 Representative Elementary Volume

3D reconstruction can provide detailed description of microscopic structure of porous media. However, currently available computational time and resources significantly limit the ability to simulate the porous media at the macroscopic scale. To overcome this issue, geometric and transport parameters are calculated using a Representative Elementary Volume (REV). REV are sized at the microscopic scale, much smaller than the size of the entire physical domain, that is considered to represent the average macroscopic geometric and transport parameters for the entire domain [5]. Many studies have constructed REV of different porous media and of different sizes to investigate the relationship between REV size and geometric and transport parameters. Zhang et al. [114] studied porous media made of crushed glass beads and sandstone using X-ray tomography images for geometric and Lattice-Boltzmann method for transport parameters. They calculated the statistical average and standard deviation of porosity, specific surface area, and permeability for all divided domains with varying sizes. Consistent with traditional, qualitative description of REV [67, 5], they found that as size increased, the average and standard deviation values converged. They observed statistically insignificant change for sizes beyond approximately 2.6 mm and 0.27 mm for medium porosity crushed glass beads and low porosity sandstone, respectively. Recently, Panerai et al. [69] studied fibrous foams such as carbon and rayon felt using similar methodology to assess the thermal conductivity. They also found that as REV size is increased (above 1 mm), porosity and thermal conductivity values converged. Using similar method, Ozelim and Cavalcante [68] found consistent results by studying porous media made of glass beads and sands. In study by Rossetto et al. [17], they heuristically determined that the size of REV for high porosity copper foam is approximately 5 to 10 times larger than average pore size using numerical and experimental results.

We choose the size of REV to be approximately 3.3mm x 3.3mm x 3.3mm or 250^3 voxels such that size of REV is approximately 10 times larger than the largest average pore size possible, consistent with previous study by Rossetto et al. [17]. However, high heterogeneity are present within the SiC foams, as depicted in Figure 3.6, likely caused due to chemical vapor infiltration (CVI) process during the manufacturing process where gas reactants that develop ligament structure are depleted as it travels through the preforms. The gradient length scale of pore characteristics spans across the entire sample and is much larger than the typical length scale permitted by computational resources. Consequently, we analyze the average pore characteristics of SiC samples to determine the appropriate location of our REV. We investigated three geometric parameters directly from the X-ray tomography images:

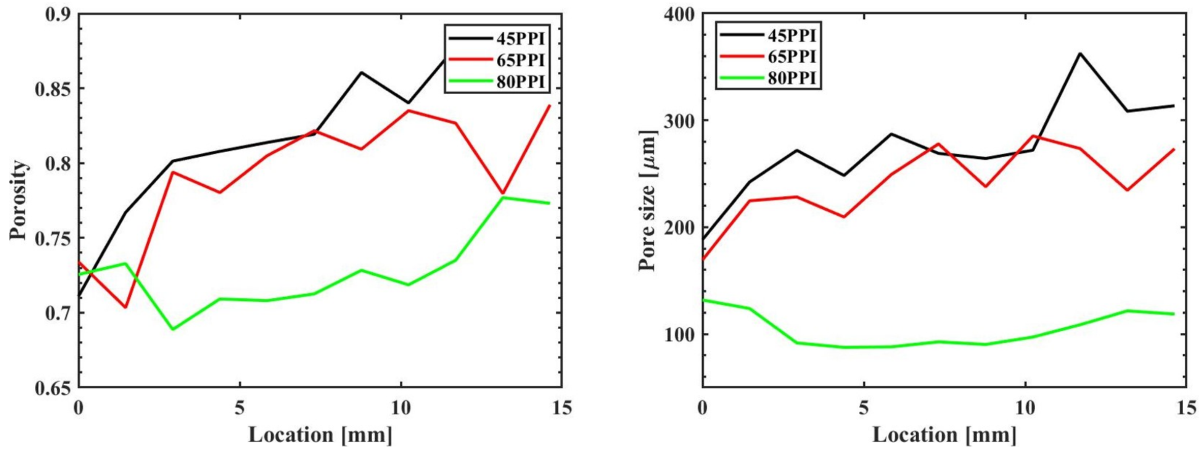


Figure 3.6: Positional variation in the porosity and average pore size along the stream-wise direction of the SiC foam samples due to their high heterogeneity.

porosity, average pore size, and tortuosity. Porosity and average pore sizes are calculated using open source code PoreSPY [25]. The porosity is calculated as the ratio of number of void voxels per the total number of voxels assuming that there were no closed pores. The average pore size is calculated using porosimetry method. The tortuosity is calculated using the fast marching algorithm [28] implemented in open source MATLAB library by Kroon [53]. Time of flight through the domain is calculated by setting planes perpendicular to the flow directions as inlet (source) and outlet (destination). Due to computational limitations, the entire sample is divided into several sub-domains of equal size for which the parameters were calculated. Overall average parameters are shown in Table 3.2. The location of REV is chosen such that the ratio of porosity, average pore size, and tortuosity given by equation

$$K \sim \frac{\epsilon d_p^2}{\tau - 1} \tag{3.3}$$

is within 5% of that for the overall sample. The ratio in Equation 3.3 follows the derivation by Du Plessis et al. in which the permeability is a function of porosity, average pore size, and tortuosity. For all three samples, the location of the REV is approximately at the center of the sample.

Table 3.2: Average porosity, pore size, and tortuosity of SiC samples used to locate appropriate REV.

Pore density	Porosity ϵ	Average pore size d_p (μm)	Tortuosity τ
45 PPI	0.829	323.5	1.0043
65 PPI	0.798	290.4	1.0048
80 PPI	0.759	146.1	1.0072

3.2.3 CFD Simulation

We next perform CFD simulations to numerically solve the steady-state Stokes equations using (ANSYS Fluent 18.2):

$$\nabla \cdot \mathbf{V} = 0 \tag{3.4}$$

$$\nabla P = \mu \nabla^2 \mathbf{V} \tag{3.5}$$

We neglect gravitational and body forces and assume constant fluid transport properties. Periodic boundary conditions are imposed at the inlet and the outlet. The computational domain is mirrored about the inlet plane perpendicular to the stream-wise direction to impose a velocity-periodic boundary condition. The pressure gradient is specified along the stream-wise direction. No-slip conditions are applied on the solid surfaces. Symmetry boundary conditions are applied to the side surfaces parallel to the stream-wise direction. Computational domain and boundary conditions are summarized in Figure 3.7. We perform a grid independence study such that increasing the number of elements resulted in less than 1% error in the permeability. A typical computational grid consisted of 5 million unstructured elements. We consider convergence is achieved when the relative change in the predicted permeability over 10 iterations is below 0.1%. We also perform a grid independence study such that decreasing the element size by half resulted in less than 0.1% error in the permeability.

Numerical simulation results are illustrated in Figure 3.8. The simulated results show a good agreement with the differential pressure values within 10% of those from our experi-

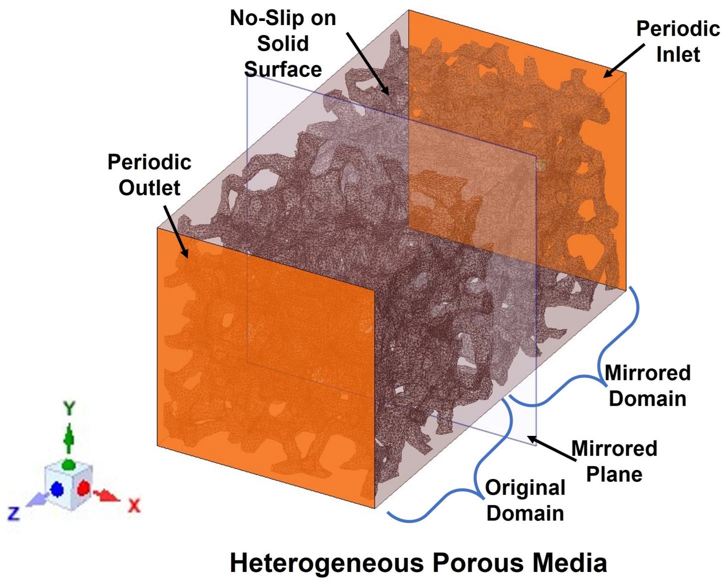


Figure 3.7: Computational domain for velocity field simulation. Periodic boundary condition was applied in the flow direction by mirroring the domain. No-slip condition was applied on the solid surfaces. Symmetric condition was applied on the boundaries of the computational domain.

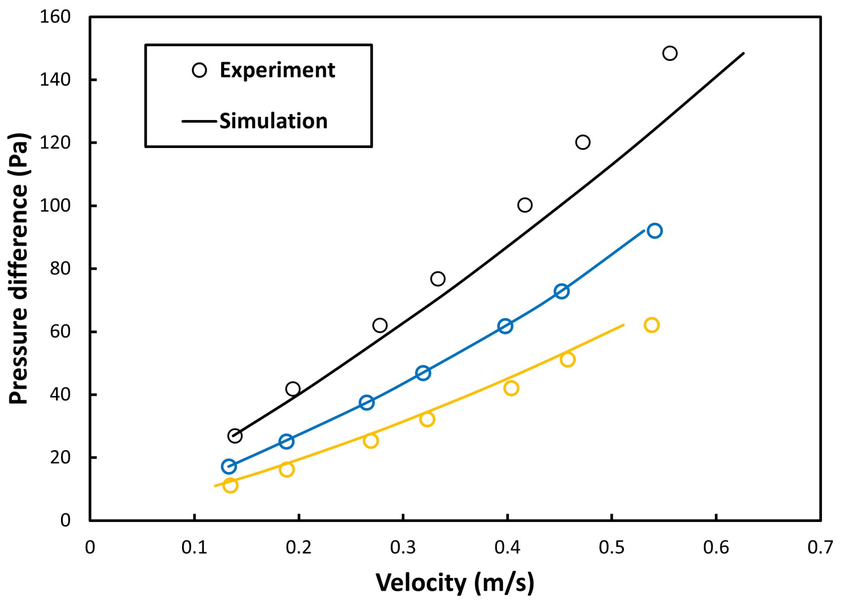


Figure 3.8: Pressure difference as a function of flow velocity obtained from numerical simulations. Computational results are within 10% of the experimental results.

ments for given flow velocities. We believe the difference between numerical simulation and experimental results is likely due to the limitations of REV that cannot represent the complete macroscopic flow behavior of the entire sample. Ideally, fluid flow through the entire sample must be solved to obtain accurate result, however, it would require unprecedented amount of computational resources and time.

3.3 Estimate of Viscous Pressure Loss

As described earlier, the driving pressure must be larger than the pressure loss through the porous medium to achieve net flow to the surface. Here, we estimate the viscous pressure loss using a 1-D model through a porous leading edge made of SiC reticulated foams characterized above. As given in equation 3.1, the viscous pressure loss is defined as

$$P_{\text{loss}} = \frac{\mu \dot{m}}{\rho K} L \tag{3.6}$$

We estimate the pressure losses as a function mass flux through the porous media. We also consider a coolant with a density of 4000 kg / m^3 and viscosity of 0.1 Pa s to simulate the extremes of typical oxide properties as shown in Table 3.3. Estimated viscous pressure loss

Table 3.3: Density and viscosity of typical oxides

Material	Density (kg/m^3)	Viscosity (Pa·s)
Al_2O_3	3950	0.066
FeO	5740	0.051
$MgSiO_3$	4103	0.0082
MgO	3580	0.0037

are illustrated in Figure 3.9 for a wide range of mass flow rate. Depending on the pore density of the reticulated foam, the necessary driving pressure to achieve a high flow rate of 20 kg/m^{2}s can range approximately from 200 Pa/mm to 100 Pa/mm. Based on this analysis, reticulated foams with low pore density is desired to minimize the viscous pressure loss through the porous leading edge volume.

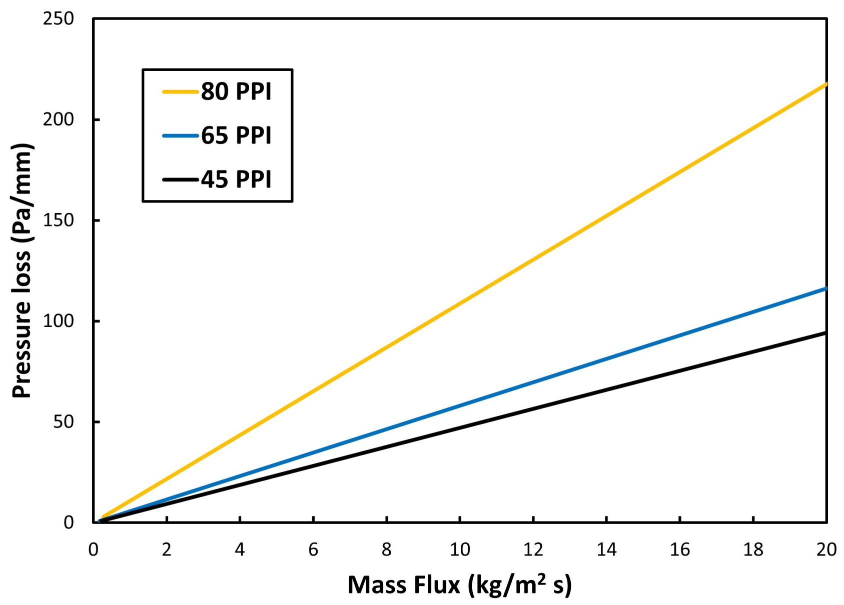


Figure 3.9: Estimated viscous pressure loss as a function of mass flux for different pore densities of reticulated foam.

3.4 Capillary Self-Pumping Capability

One promising advantage of evaporative transpiration cooling is its ability to sufficiently self-pump the liquid coolant to the surface. Oxide coolants have an advantage in this aspect since they typically possess high surface energy, and thereby, surface tension. We use a 1-D model using the permeability obtained in this chapter and Young-Laplace equation given as

$$P_{\text{cap}} = \frac{4\sigma\cos(\theta)}{D_{\text{p}}} \tag{3.7}$$

to calculate the ratio of capillary pressure and viscous pressure loss. The ratio is illustrated in Figure 3.10 for the three pore densities considered. The surface tension and the contact angle are assumed to be 0.1 N/m and 45 degrees, respectively, which are typical for most oxides. We find that porous leading edge with the lowest pore density achieves the highest ratio of the two pressures. This result suggests that low pore density is still favorable to maximize the capillary pumping capability of the porous leading edge. We hypothesize that

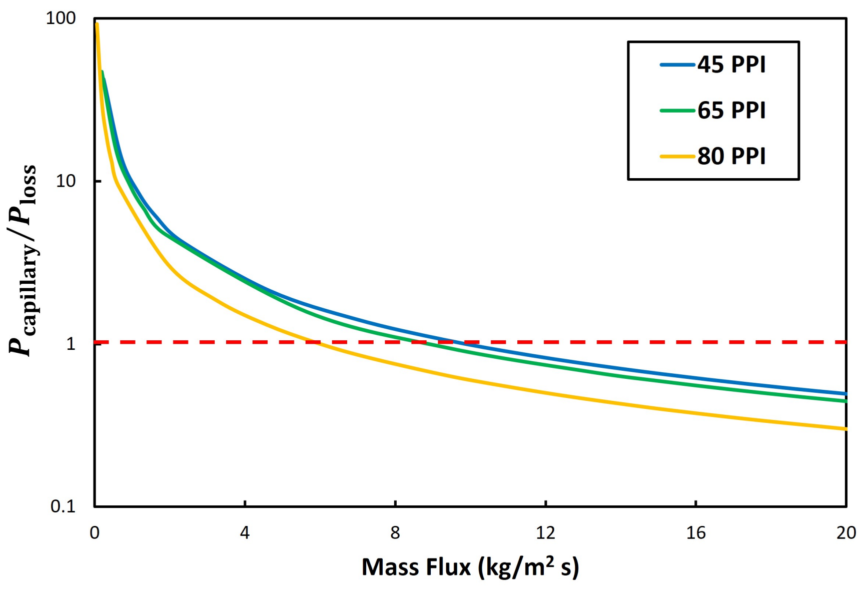


Figure 3.10: Ratio of capillary pressure and viscous pressure loss as a function of mass flux through a porous leading edge.

this behavior is because the capillary pressure linearly depends on the pore diameter whereas the permeability is proportional to the square of the pore diameter. Hence, the porous leading edge that maximizes the permeability also maximizes the capillary pumping capability. We also note that the reticulated foams that we considered cannot sustain necessary flow rate beyond approximately 10 kg/m^2 by relying solely on capillary pumping, requiring external means of pressure or force to deliver the fluid.

3.5 Summary and Conclusion

In this chapter, flow through porous leading edge is characterized at the macro-scale to predict the necessary driving pressure for estimated properties of typical oxides. We experimentally and numerically characterized the permeability of SiC foams with three different pore densities. Experimental method used differential pressure transducers and flow meters to obtain the permeability for various air flow rates. Numerical method utilized X-ray microtomography images to reconstruct the representative elemental volume of porous media and

CFD to solve the fluid flow. The permeability values of all SiC foams agreed well between the two methods. The permeability values are then used to estimate the necessary driving pressure. The porous media with the lowest pore density showed the minimum pressure gradient to achieve a given mass flow rate. We also assess the capillary pumping capability of the porous media and demonstrate that it is maximized using the porous media with the lowest pore density porous. Therefore, a porous leading edge consisting of low pore density is desired to minimize the necessary driving pressure and maximize the self-pumping capability.

Chapter 4

Deep Learning Prediction of Pore-scale Flow Using Synthetic Porous Media

To address the limitations of conventional simulations, data-driven deep learning (DL) models have been developed to predict flow through porous media [43, 22, 16, 45]. Different DL model architectures have been explored to study the flow characteristics of porous media. For instance, the physics-informed neural network (PINN) uses an autoencoder, consisting of multi-layer perceptrons (MLP), to predict velocity and pressure fields as a function of position and time [61, 31, 42, 12, 11]. The PINN incorporates the governing equations, boundary conditions, and initial conditions, which limits its applicability to specific computational domains and increases the computational cost of training. Similarly, the PointNet [72, 48] performs a direct mapping between the vertices of a computational grid and the corresponding velocity and pressure values using MLP. It has been used to predict the permeability of sandstones [45] as well as velocity fields using a similar approach as the PINN [46, 47].

The convolutional neural network (CNN) consists of convolutional layers that recognize patterns in an image and is suited for image-based flow field reconstruction. CNN has been widely used to predict flows through porous media [109, 119, 100, 91]. Sudakov et al. [90] used a combination of CNN and MLP to predict the permeability of certain sandstones with a small average error of approximately 4%. Similarly, Kamrava et al. [44] used a combination of CNN and MLP to predict the permeability of 3D sandstones with porosities of approximately 0.2. Zhang et al. [115] also used a mix of CNN and MLP to predict the permeability of 2D synthetic porous media of porosity between 0.4 and 0.8 using low-resolution images obtained via bi-cubic interpolation [50].

Unlike the above models, which only provided an integrated property, other recent CNN models were designed to predict pore-scale 3D velocity fields. Santos et al. [83] presented a CNN model to predict the velocity field, although only in the stream-wise direction, of various sandstones. Their model was trained using disordered packs of spherical grains with porosity between 10% and 30%. Their model showed errors as low as 1% in permeability, but it required four precomputed inputs: the Euclidean distance, the maximum inscribed sphere, and the time of flight in two directions, and was not trained to satisfy the law of mass conservation. The same group [84] reported a CNN model that extracts the relationship between the porous media microstructure and the velocity fields at different length scales. Wang et al. [102], presented a CNN model to predict 2D and 3D velocity fields in synthetic porous media created using the algorithm described by Liu and Mostaghimi [59] where they segment a field of uniformly distributed random numbers after applying a Gaussian blurring kernel of different sizes. Their model related the binary image and the Euclidean distance to the velocity field. The model predicted the permeability within 1% error for 2D synthetic porous media with permeability ranging from $1 \times 10^{-19}\text{m}^2$ to $1 \times 10^{-12}\text{m}^2$. However, the error increased by more than 10% for 3D synthetic porous media with permeability ranging from $1 \times 10^{-14}\text{m}^2$ to $1 \times 10^{-12}\text{m}^2$. They similarly observed larger voxel-wise errors in the 3D velocity fields.

In this chapter, we utilize CNN and implement a physics-informed loss function to predict the 3D velocity field of real reticulated foams from only its binary images. To our knowledge, a reasonable prediction of the full 3D velocity field at the pore-scale using only the binary image has not yet been reported. An ideal model should not require precomputation, which can be cumbersome. Moreover, the prediction of the 3D velocity field in both stream-wise and span-wise directions is essential for accurate transport analysis. The result of heat transfer analysis without the span-wise velocity is erroneous as shown in Figure 4.1.

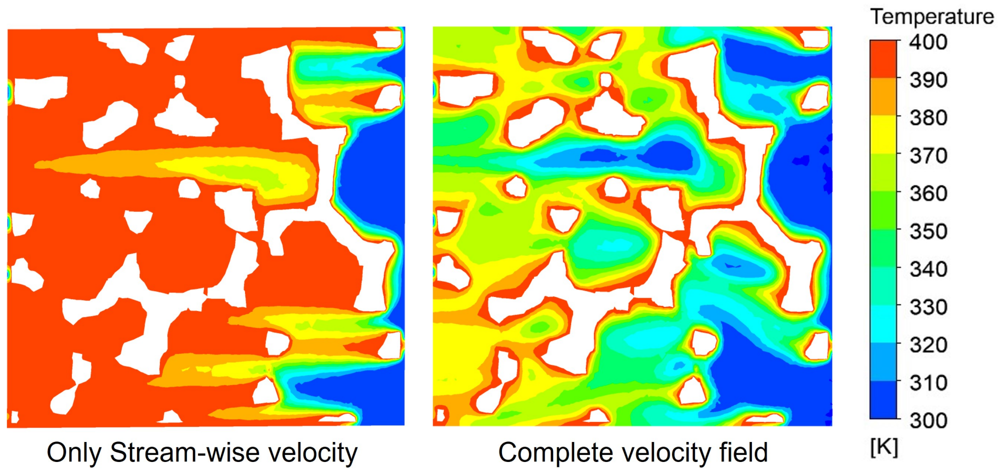


Figure 4.1: Example of the results from heat transfer analysis. Using only the stream-wise velocity field shows significant error in the temperature field as the upstream information cannot propagate through the tortuous paths of the porous media.

We also study complex, inhomogeneous reticulated foams which have microstructures significantly different from typical porous media considered in previous studies. Reticulated foams have high porosity, large specific surface area, mechanical robustness, and light weight, which are advantageous in many engineering applications including filtration [94, 70], catalytic reactions [14, 58, 23], latent energy storage [117, 39, 118], and heat-exchangers [9, 40]. Their unique pore-scale characteristics can increase the learning difficulty for deep learning models.

We design our model to have separately trained submodels that learn the 3D spatial relationship of each velocity component and a main model that enforces the law of mass conservation. We demonstrate that our model provides comparable accuracy to the traditional CFD methods both at the macro-scale and pore-scale, and our physics-informed loss function significantly reduces the divergence of the velocity fields. The prediction using our deep learning model only takes a few seconds on modern workstations while traditional CFD methods require a few hours. Our design is also memory efficient, so that the training process can be handled by a single Graphical Processing Unit (GPU). We further utilize our model in heat transfer analysis to demonstrate its accuracy and advantage in consecutive transport analysis for various engineering applications. The overall workflow is presented in Figure 4.2.

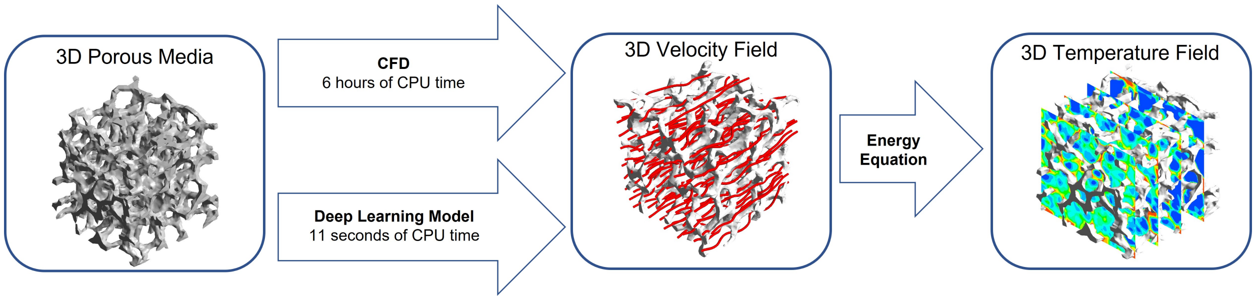


Figure 4.2: Workflow to predict the velocity field directly from the binary image of porous media using the deep learning model. Prediction of the velocity field takes only a few seconds using our deep learning model. Using our deep learning model, we perform heat transfer analysis to assess the accuracy and advantages of our deep learning model.

4.1 Neural Network Architecture

Our neural network (Figure 4.3) uses the convolutional U-Net structure [80]. It utilizes a series of stacked convolutional layers to extract both high- and low-level features and reconstruct the output. Our model decomposes the velocity field into component-wise submodels. It is similar to a previous work by [76] where the velocity components were trained separately [101]. However, we incorporate the submodels at the encoding branch of the U-Net to extract component-wise hierarchical features. Each submodel processes the input binary data S representing a porous medium and outputs a velocity field V_n, where subscript n

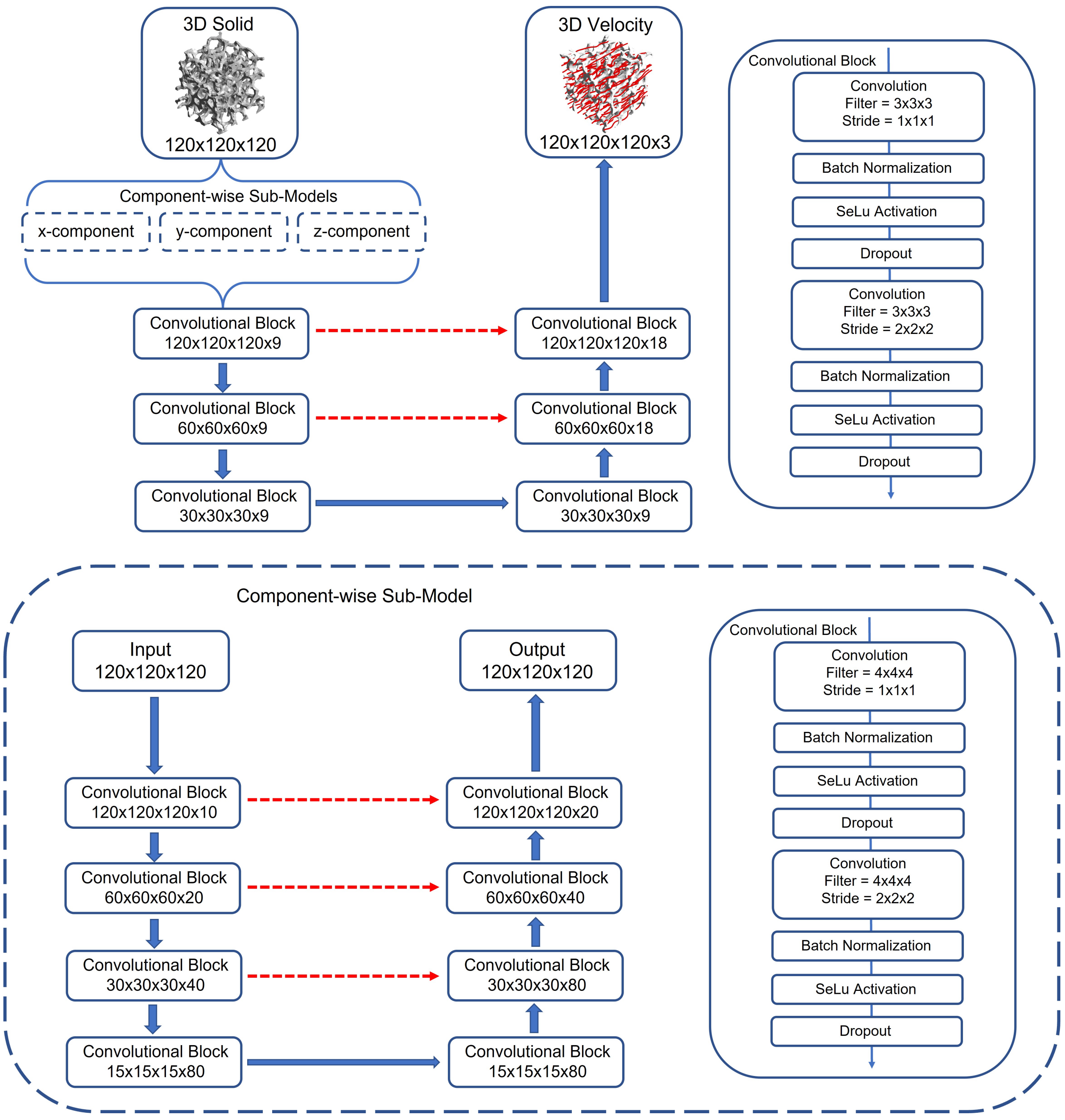

3D Solid
120x120x120
3D Velocity
120x120x120x3
Component-wise Sub-Models
x-component
y-component
z-component
Convolutional Block 120x120x120x9
Convolutional Block 60x60x60x9
Convolutional Block 30x30x30x9
Convolutional Block 120x120x120x18
Convolutional Block 60x60x60x18
Convolutional Block 30x30x30x9
Convolutional Block
Convolution Filter = 3x3x3 Stride = 1x1x1
Batch Normalization
SeLu Activation
Dropout
Convolution Filter = 3x3x3 Stride = 2x2x2
Batch Normalization
SeLu Activation
Dropout
Component-wise Sub-Model
Input 120x120x120
Output 120x120x120
Convolutional Block 120x120x120x10
Convolutional Block 60x60x60x20
Convolutional Block 30x30x30x40
Convolutional Block 15x15x15x80
Convolutional Block 120x120x120x20
Convolutional Block 60x60x60x40
Convolutional Block 30x30x30x80
Convolutional Block 15x15x15x80
Convolutional Block
Convolution Filter = 4x4x4 Stride = 1x1x1
Batch Normalization
SeLu Activation
Dropout
Convolution Filter = 4x4x4 Stride = 2x2x2
Batch Normalization
SeLu Activation
Dropout

Figure 4.3: Schematic of our deep learning model. The model incorporates three separately trained submodels. Outputs from the submodels are used to enforce the law of mass conservation in incompressible flow. Red dashed arrows indicate skip-connections.

denotes the velocity component in either x, y, or z direction such that:

$$V_n = \mathcal{F}_n\left(S; \mathbf{w}_n, \mathbf{b}_n\right) \tag{4.1}$$

Here, $\mathcal{F}_n$ is the trained submodel with optimized weights $\mathbf{w}_n$ and biases $\mathbf{b}_n$ for each velocity component. The outputs of the submodels are combined within the trained model $\mathcal{G}$ such that:

$$\mathbf{V} = \mathcal{G}\left(\left[V_x, V_y, V_z\right]; \mathbf{w}, \mathbf{b}\right) \tag{4.2}$$

where weights $\mathbf{w}$ and biases $\mathbf{b}$ are optimized to enforce the physics-informed loss function as described in Section 4.2. We use the batch normalization layer and the dropout layer to help generalize the model and prevent over-fitting. We use the scaled exponential linear unit (SeLu) [52] as the non-linear activation function instead of the rectified linear unit because the velocity can be negative in the span-wise direction.

We tuned the hyperparameters using a method of discrete grid search. We used the default parameters for all batch normalization layers. We used a dropout rate of 0.2 for x and y directions and 0.1 for z direction. A lower dropout rate of 0.001 was used for the main model. To train, we used the Adam optimizer for all models with different learning rates of 0.0008 for x and y directions and 0.0006 for z direction. We trained the main model with a default learning rate of 0.001. We used the TensorFlow library [1] to construct and train our model. We used a mini-batch size of four. We also applied early stopping criterion with 50 learning iterations to prevent preemptive stoppage and over-fitting. Each training run did not exceed 8 hours. We trained using a single NVIDIA V100 GPU with 16GB of memory.

A major challenge in deep learning is the amount of required memory for training [84]. All model parameters, including weights and biases, the inputs, and the outputs must be locally stored during training. Even a single batch consisting of $120 \times 120 \times 120$ tensor for the input and the output can consume considerable amount of the memory. Previous studies had to compensate with either reduction in the resolution of the velocity field or in the number of training data [83, 102]. We believe that our approach of decomposing the velocity into its components helps reduce the memory required to train our model. However, the batch size remains constrained by the limitations imposed by GPU memory. The total number of trainable parameters of our model is approximately 5 million which is an order of magnitude less than previously reported model [102].

4.2 Physics-Informed Loss Function

We implement a physics-informed loss function (PIMSE) that incorporates the differential form of the law of mass conservation for incompressible flows as part of the general mean squared error (MSE) loss function:

$$\text{PIMSE} = \frac{1}{N}\sum_{i}^{N}\left(\sum_{n}^{x,y,z}\left|\mathbf{V}_{n,i} - \mathbf{V}^{*}_{n,i}\right|^{2} + \alpha\left|\nabla\cdot\mathbf{V}^{*}_{i}\right|\right) \tag{4.3}$$

Here, N is the total number of voxels (L^3), super script * denotes the predicted value, and α is a penalty coefficient. The divergence of velocity field is calculated using the second-order central difference scheme.

Our PIMSE expands upon the previous work by [102], which only balanced the 2D planar mass flux. [63] have implemented the divergence free condition directly into the neural network. However, we implement the divergence free condition into the loss function to simplify the model. We choose to impose the divergence free condition for all voxels with an equal weight, and hence, define the divergence loss with L_1 penalization. The weighting factor α is chosen to be 3 such that the values of the MSE and the divergence error are comparable at the start of the training and that the learning process is not dominated by one metric.

We choose the MSE loss function over the mean absolute error or the mean absolute percent error which can be problematic when the velocity distribution spans several orders of magnitude with large number of voxels with near-zero velocity. It also puts more emphasis on the voxels with relatively higher velocity that govern the dominant flow paths and transport.

For training the sub-models, we use the general MSE loss function (equation (5.5) with $\alpha = 0$) since the divergence free condition cannot be imposed on a single velocity component. We also use the MSE loss function to train our model for comparison purposes.

4.3 Data Set Generation

For training and validation purposes, we generated synthetic porous media with randomly dispersed spherical pores to emulate the stochastic nature of the reticulated foams (Figure 4.4). We used the overlapping spheres function in open-source library PoreSpy [25]

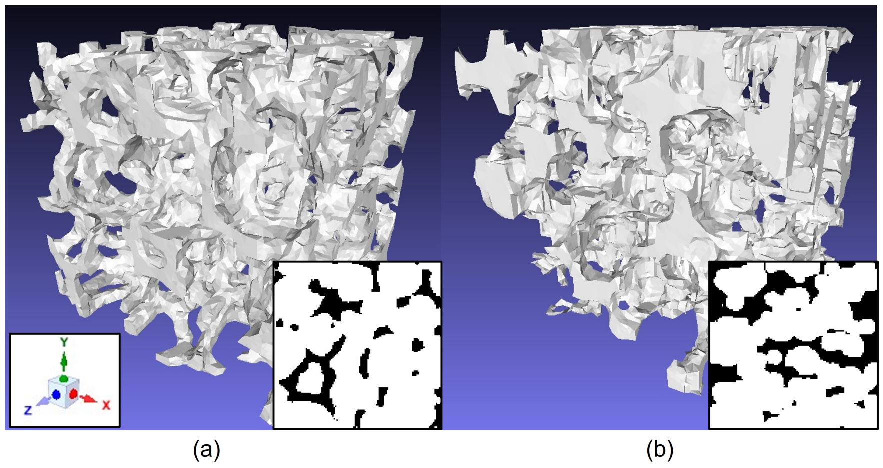


Figure 4.4: Example of (a) digitally reconstructed reticulated foam and (b) generated synthetic porous medium with randomly dispersed spheres. Insert images show the planar binary image of each porous medium where the black and the white represent the solid and the void, respectively.

to distribute spherical pores with radii of a specified mean value and standard deviation. The final porosity values varied from 70% to 80%. We chose the parameters of the synthetic porous media such that the range of normalized permeability (see equation (4.5)) is similar to that of our reticulated foams. To assess our model, we used SiC foams with experimentally and numerically characterized in Chapter 3. We used two REV, with at least five pores, for each target pore densities. Relevant pore characteristics of the synthetic porous media and the reticulated foams are summarized in Table 4.1.

We next performed CFD simulations to obtain the ground-truth velocity fields following the approach described in Chapter 3. We normalize the velocity fields before they are

Table 4.1: Summary of pore characteristics for the synthetic porous media and the reticulated foams used in this chapter. The porous media are named in the decreasing order in the normalized permeability.

Porous Media	**Porosity**	**Tortuosity**	**Normalized Permeability**
Synthetic 1	0.77	1.0186	11.5
Synthetic 2	0.79	1.0117	11.5
Synthetic 3	0.74	1.0155	10.2
Synthetic 4	0.75	1.0176	10.1
Synthetic 5	0.73	1.0191	8.6
Synthetic 6	0.76	1.0189	8.4
Synthetic 7	0.75	1.0143	7.7
Synthetic 8	0.69	1.0259	7.2
Synthetic 9	0.69	1.0284	7.1
Foam 1	0.85	1.178	11.5
Foam 2	0.78	1.207	10.4
Foam 3	0.75	1.202	9.1
Foam 4	0.81	1.159	8.2
Foam 5	0.83	1.170	7.8
Foam 6	0.79	1.168	6.6

provided as inputs to our deep learning model. Darcy's law [105] states that

$$\frac{dP}{dz} = \frac{\mu}{K} V_z \tag{4.4}$$

where K is the permeability of the porous media and assumed to be a constant. According to the Carman-Kozeny relation, which is a good approximation for a wide variety of porous media [121, 56, 49], the permeability scales as the square of the pore length scale. Based on these relations, we normalize the 3D velocity fields using the following equation:

$$\tilde{\mathbf{V}} = \frac{\mu}{dP/dz} \frac{1}{R^2} \mathbf{V} \tag{4.5}$$

where the normalized velocity vector is denoted as $\tilde{\mathbf{V}}$ and the image spatial resolution as R. Our normalization scheme allows us to have a consistent physical length scale across all data which is crucial in image-based learning. Since $\tilde{\mathbf{V}}$ is simply a multiple of $\mathbf{V}$ for a given image spatial resolution and flow conditions, we interchangeably refer to the normalized velocity as $\mathbf{V}$. The normalized velocity is then scaled to have a maximum value of 10 for training.

To create our training data, we performed 3D image augmentation on the original solid data and the full 3D velocity fields. We obtained 270 training data for the sub-models ($270 \times 120 \times 120 \times 120 \times 1$). We obtained a second set of training data consisting of 135 samples for the main model ($135 \times 120 \times 120 \times 120 \times 3$). The total number of training data is reduced to accommodate three velocity components. We applied random shifts along x and y directions. We also applied random flips about x and y axis to ensure that the network sees the different orientations of the data during training. We changed the velocity direction for the x and the y components accordingly to ensure the symmetric boundary condition. We randomly assigned 10% of the training data for validation and another 10% for testing.

4.4 Results and Discussion

4.4.1 Microstructure of Reticulated Foams

No previous studies to our knowledge have considered high porosity reticulated foams. Due to their unique pore characteristics, the reticulated foams are very different from typical porous media such as sandstones and bead packs, which can make the training process more difficult. As an example, we compare four pore characteristics of the reticulated foams and of Fontainbleau sandstone [83] to highlight the differences in Figure 4.5. The four pore characteristics are the Euclidean distance, the diameter of maximum inscribed sphere (MIS), and the detrended time of flight in directions along and against the flow. The Euclidean distance provides a compact representation of space available for fluid flow. The maximum inscribed sphere provides information about the local pores. The detrended time of flight provides information on the tortuous flow path within the porous media.

The histograms of the Euclidean distance and the maximum inscribed sphere are much wider than those of the sandstones. The wider distributions are due to the high porosity

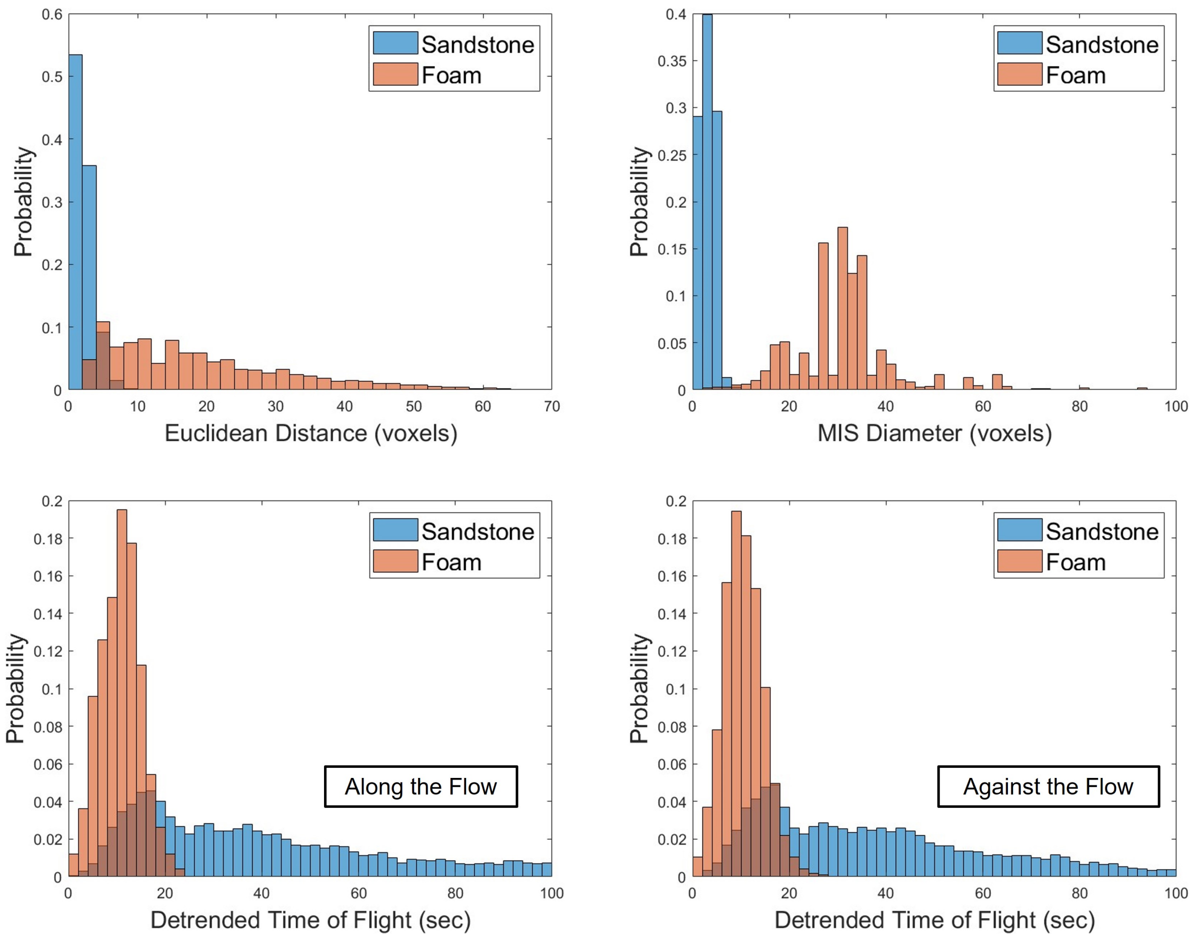


Figure 4.5: Comparison of the histogram of the four pore characteristics for the reticulated foams and Fontainbleau sandstone. The histograms of the Euclidean distance and the maximum inscribed sphere for the reticulated foams show much wider range with larger values. On the contrary, the histograms of the time of flight for the reticulated foams show a much narrower range with smaller values.

and large pore sizes. The reticulated foams have a porosity between 75% and 85%, as summarized in Table 4.1, while the Fontainbleau sandstones have a porosity between 5% and 40%. The pore sizes of the reticulated foams are on the order of $100\mu m$, but the pore sizes of Fontainbleau sandstones are on the order of $1\mu m$. High porosity and large pore sizes of reticulated foams cause considerable amount of the volume to serve as flow paths as shown in Figure 4.6. Figure 4.7 shows the distribution of the absolute velocity in each direction for the reticulated foams. The wide distribution of the magnitude of the velocity component can complicate the learning process. The oscillation in the probability at very low velocity

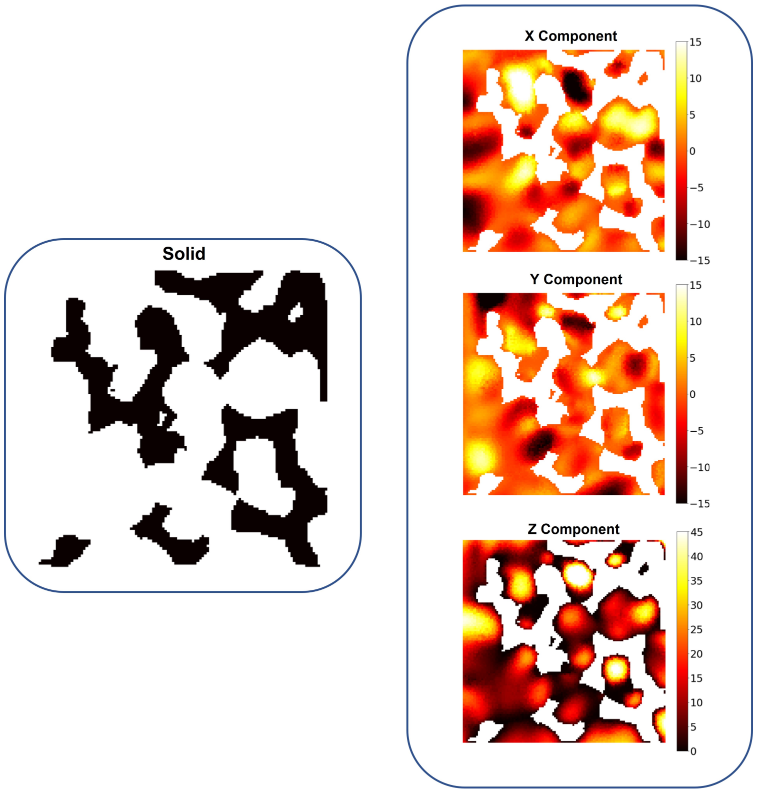


Figure 4.6: Velocity profiles of a reticulated foam at the cross-section orthogonal to the z direction. High porosity and large pore sizes cause a considerable amount of the volume to contribute to the fluid flow. The x and y velocity components also have orders of magnitude comparable to the z velocity component in both the positive and the negative directions.

is due to the limited accuracy of the numerical solver.

In contrast, the distributions of detrended time of flight are significantly skewed towards the minimum because of the low tortuosity of the reticulated foams. Fontainbleau sandstones have tortuosity above 1.5, and our reticulated foams have it around 1.2, as summarized in Table 4.1. However, the heterogeneity of the reticulated foams results in high velocities in the span-wise direction, as shown in Figure 4.6 and 4.7. The velocity distributions in both positive and negative directions are also identical, as shown in Figure 4.8. The two equal distributions exacerbate the nonunique spatial-velocity relationship that exacerbates the learning difficulty.

We believe that incorporating component-wise submodels helps the model to learn the spatial-velocity relationship considerably. We illustrate the averaged convolutional features

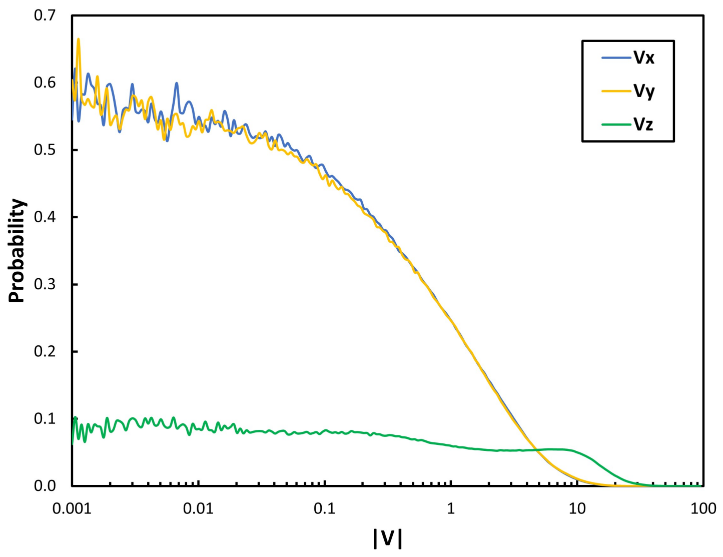


Figure 4.7: Distribution of the absolute velocity for each velocity component for the reticulated foams. Due to the high porosity, the distribution of the z velocity spreads across a wide range of velocity. The x and the y velocity components have comparable orders of magnitude as the z velocity component.

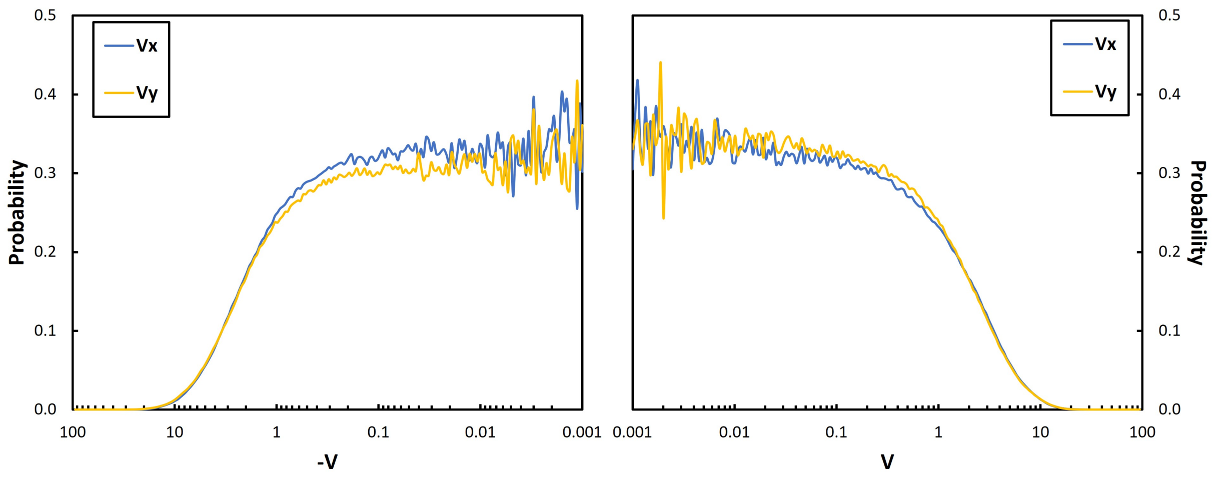


Figure 4.8: Distribution of the x and y velocity components in the negative and positive directions. The two distributions are identical due to the heterogeneity of the reticulated foams, which further complicates the non-unique spatial-velocity relationship.

at the cross-section orthogonal to the z direction after the first and last convolutional blocks of the encoding branch for each velocity component in Figure 4.9. The corresponding solid image and velocity field of the reticulated foam are also shown. The features extracted at the end of the first convolutional block mostly retain the binary image of the reticulated foam.

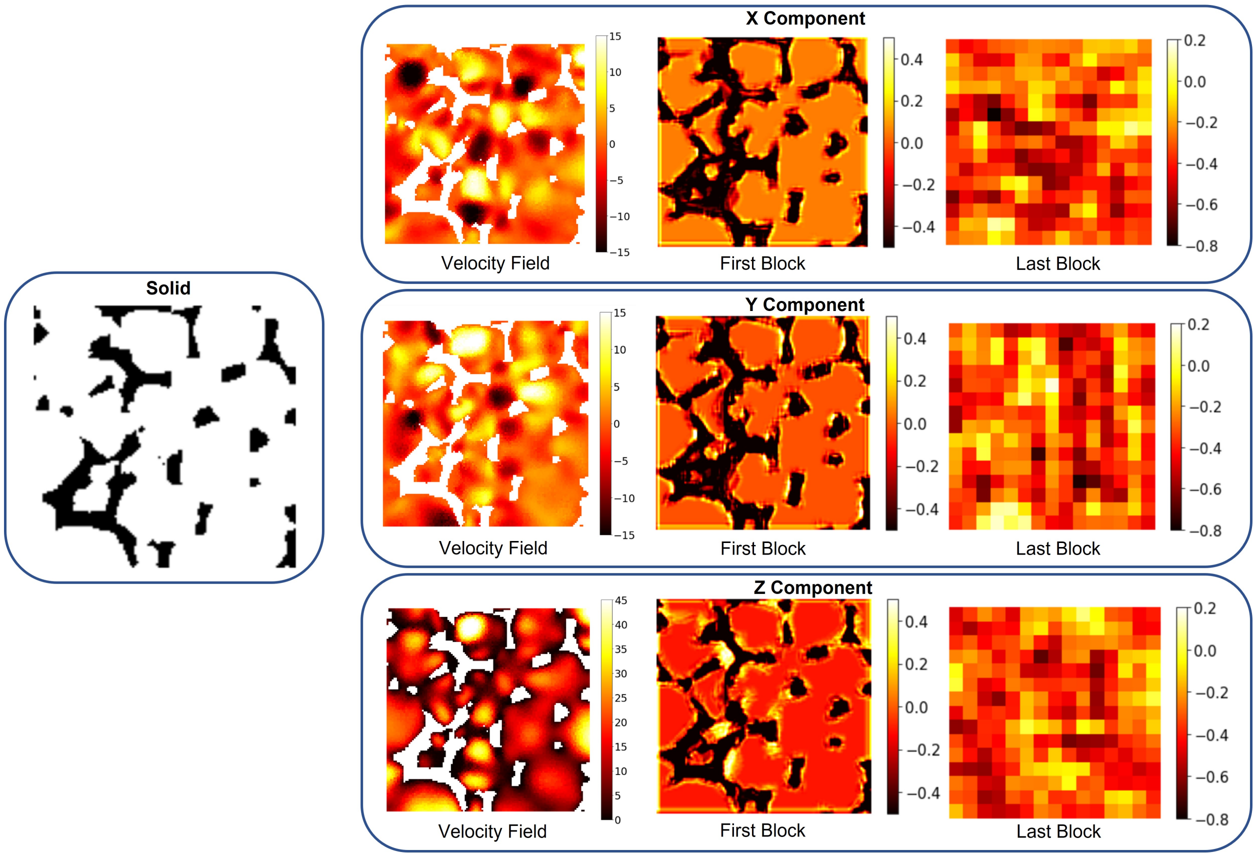


Figure 4.9: Extracted convolutional features for each velocity component at the end of the first and the last convolutional blocks of the submodels. The features are averaged and illustrated at the cross-section orthogonal to the z direction. The extracted features of each component show activations at different locations for different velocity components, suggesting that component-wise models help the learning process.

We expect this behavior since the first convolutional block needs to guide the reconstruction of the velocity field in the decoding branch through the skip-connection. We note that, however, they still show slightly different weights and biases.

Conversely, the low-level features illustrate a clear difference suggesting that the weights and biases of a submodel are optimized to fit the spatial-velocity relationship for the corresponding direction. Although the distributions of the x and y velocity components are nearly identical, the extracted 3D features show a difference. Because the two velocity components are rotated 90° from each other, a different spatial dependency is expected. It may be possible to merge the two submodels, but it would require further modification and is beyond the scope. We also mention that the low-level features do not completely resemble the velocity fields since the majority of the reconstruction is performed within the decoding branch. By incorporating component-wise submodels, we extract unique spatial-velocity relationships for each component and ensure efficient training of the model.

4.4.2 Permeability

We first quantify the accuracy of our model at the macro-scale. Figure 4.10 shows the comparison between the ground-truth and the predicted permeability for the test data of synthetic porous media and reticulated foams. The model shows excellent agreement in

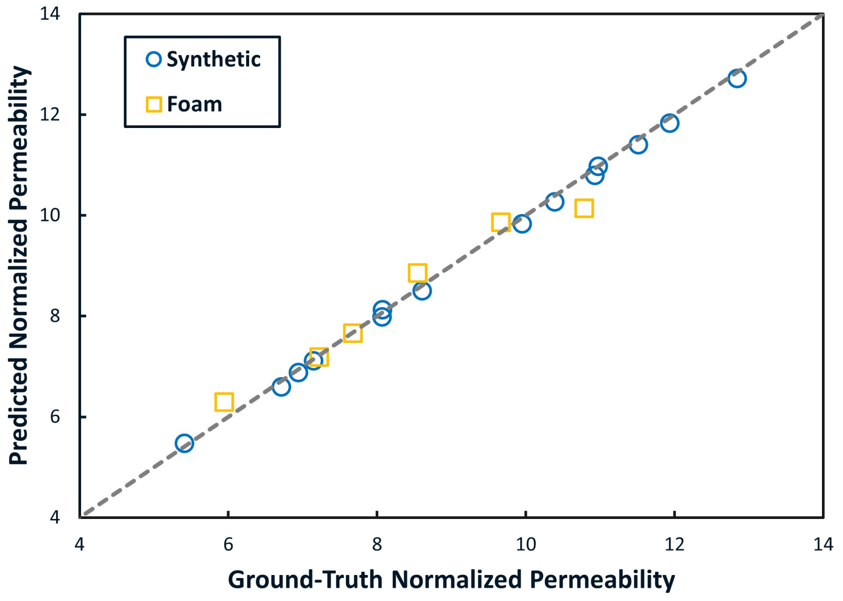


Figure 4.10: Comparison of the permeability of the synthetic porous media and reticulated foams. The predicted permeability is in excellent agreement with an average error of 1% and 3% for the synthetic porous media and reticulated foams, respectively.

predicting the permeability for both porous media. We obtain an average error of 1% and a maximum error of 2% for the synthetic porous media. For the reticulated foams, we obtain an average error of 3% and a maximum error of 6%.

We also compare the scaled total absolute flow error (STAFE) in Figure 5.8. The STAFE is defined as [102]:

$$\text{STAFE} = \sum_{n}^{x,y,z} \frac{|q_n - q_n^*|}{q_z} \tag{4.6}$$

where q_n is the planar flow rate in the n direction defined as

$$q_x = \sum_{j}^{N}\sum_{k}^{N} V_x|_{i,j,k}, \quad q_y = \sum_{i}^{N}\sum_{k}^{N} V_y|_{i,j,k}, \quad q_z = \sum_{i}^{N}\sum_{j}^{N} V_z|_{i,j,k}. \tag{4.7}$$

The STAFE accounts for error in the predicted mass flow rate in each direction and reflects additional details of the flow at the pore-scale than the permeability [102]. The average

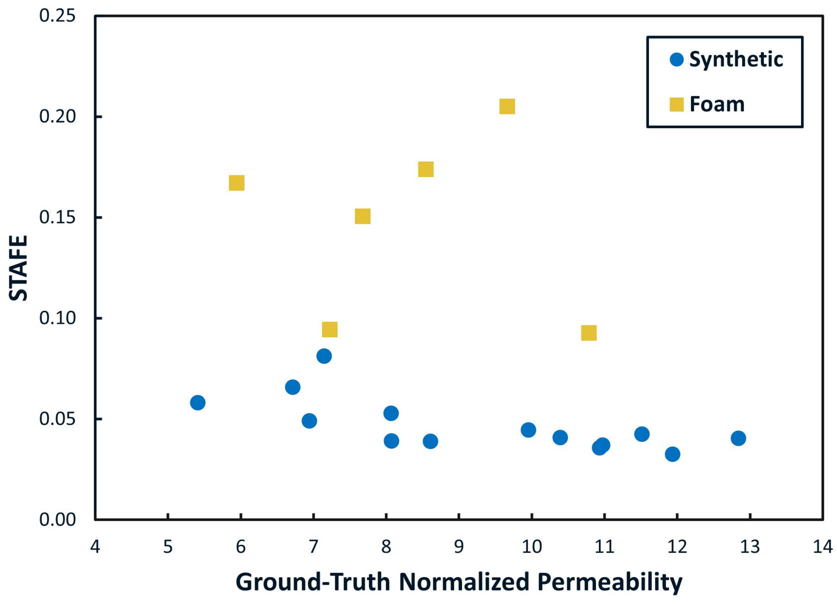


Figure 4.11: STAFE for the synthetic porous media and reticulated foams. The average STAFEs are 0.04 and 0.14, respectively. The STAFEs for the reticulated foams are slightly larger than those for the synthetic porous media due to the difference in the microstructures.

STAFE for the synthetic porous media is 0.04, and the average STAFE for the reticulated foams is 0.14. The average STAFE of our model are orders of magnitude smaller than that reported by [102] which signifies its accuracy. We note that the STAFE is slightly higher for the reticulated foams likely due to the intrinsic difference in the microstructures between the two porous media. The cellular structure of the reticulated foam is governed by energy minimization [24], and the pores typically take a tetrakaidecahedron (14-sided polygon) shape [77] which differs from randomly dispersed spherical pores. Porosity and tortuosity are slightly different even though normalized permeability is similar as summarized in Table 4.1. The velocity distributions also indicate a slight difference at moderate and high velocity, as shown in Figure 4.12.

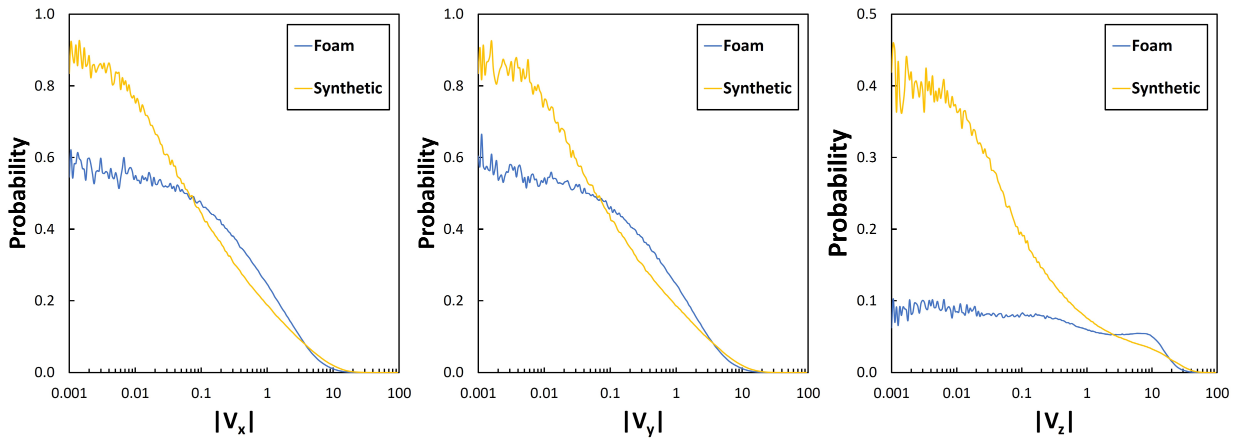


Figure 4.12: Histogram of the ground-truth velocity fields for the synthetic porous media and the reticulated foams. The two distributions show a slight difference at moderate and high velocity suggesting a subtle difference in the microstructures.

Our results are consistent with the previous studies [83, 84] where larger errors were reported for porous media that are different from the training data. Our training data can be expanded to cover various microstructures and permeability ranges; however, computational power is often the limitation.

4.4.3 Pore-Level Velocity Fields

We now consider the pore-scale accuracy of our model. Here, we focus on the reticulated foams. Figure 4.13 visualizes the pore-scale velocity fields. A good agreement is shown between the ground-truth and the predicted velocity fields. Figure 4.14 illustrates the 2D planar velocity fields to visualize the details. The planar velocity fields also show good qualitative agreement. We observe that no voxels exhibit a difference in the velocity direction, and only a few voxels show a difference in the velocity magnitude.

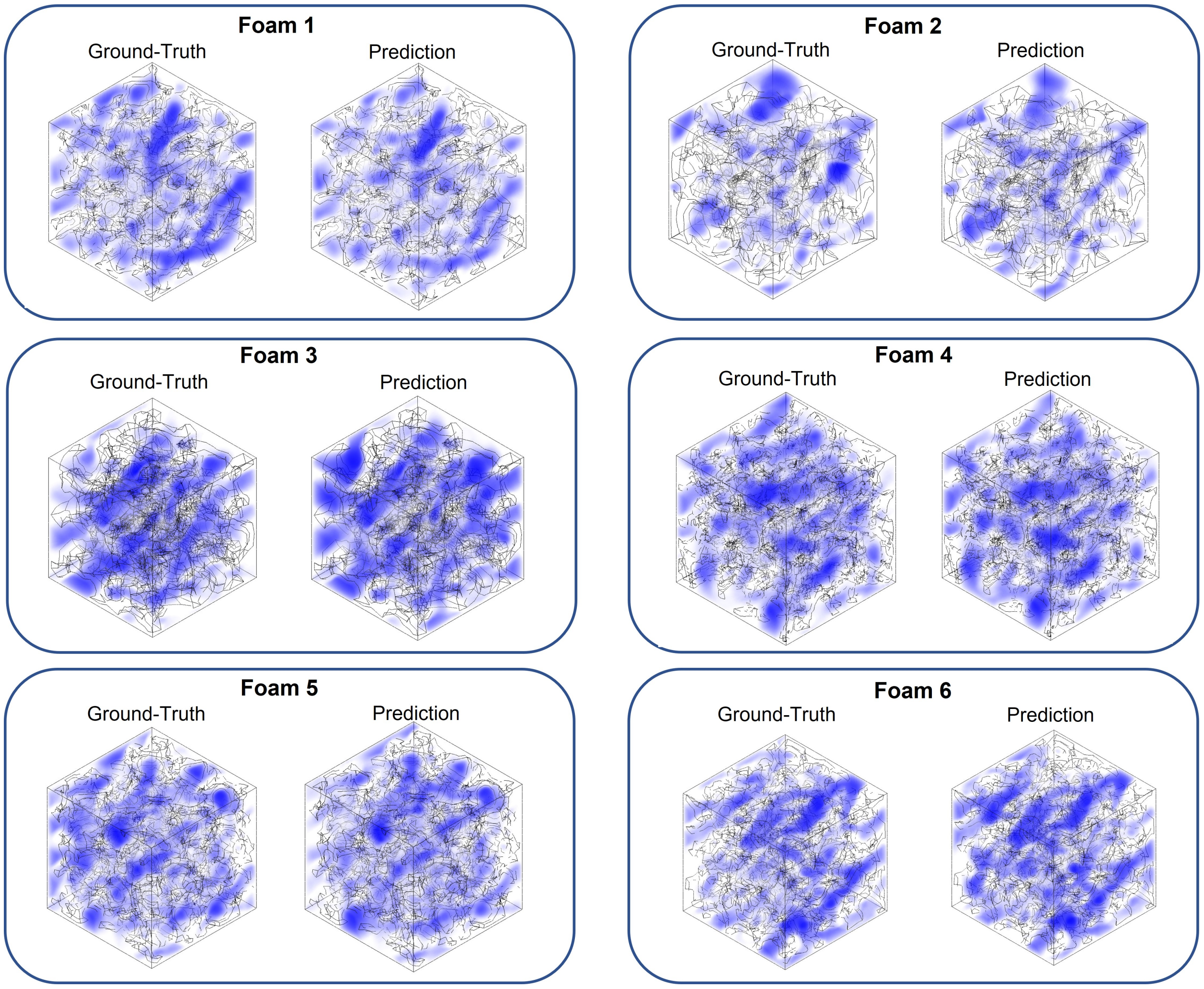


Figure 4.13: Visualization of the ground-truth and predicted velocity field for the reticulated foams. The velocity fields show good qualitative agreement.

A comparison of the velocity distribution for each component is illustrated in Figure 4.15. The distributions show excellent agreement at high velocity. However, the accuracy degrades at low velocity, below approximately two orders of magnitude of the maximum. We believe this trend is due to the use of MSE loss function that focuses on the region with larger velocity which allows very accurate predictions in the permeability and the STAFE. We note that most of the errors at very low velocity reside in the solid region where the velocity is zero. The model predicts small finite velocity in the solid region since the difference has negligible impact on the learning process. Hence, we apply a binary mask corresponding to assess the voxel-wise accuracy.

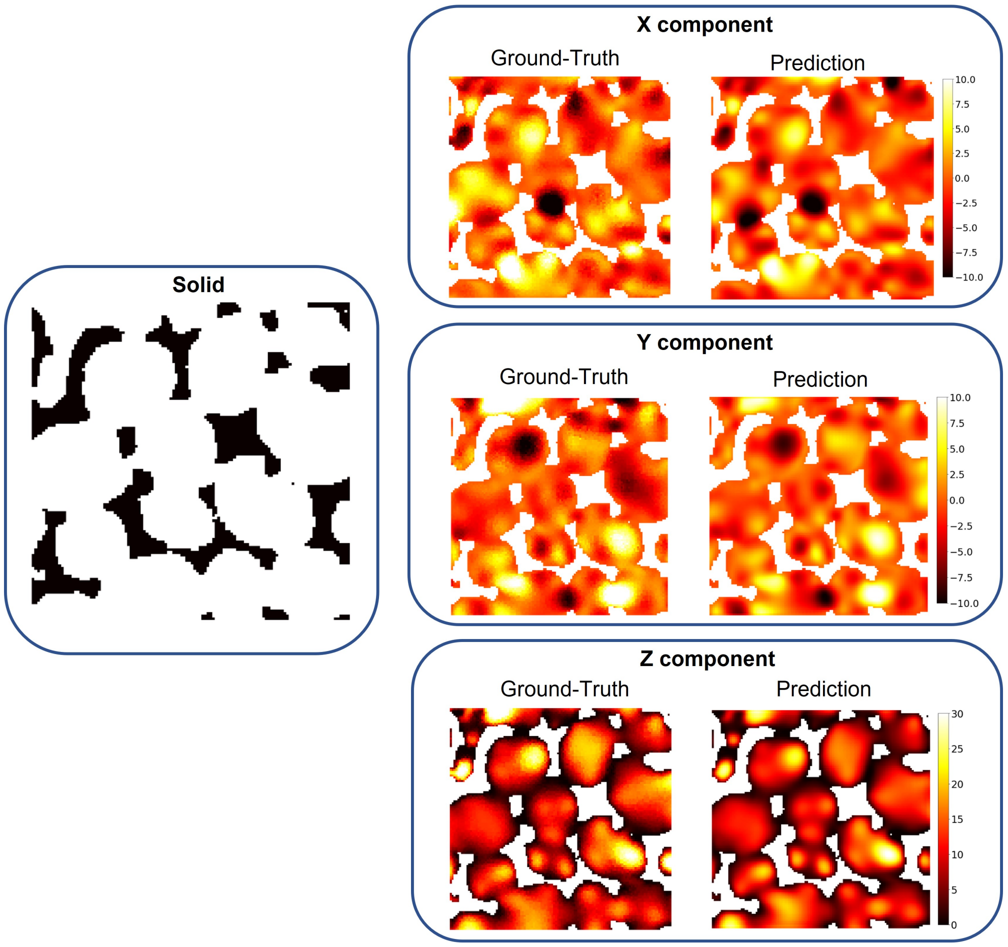


Figure 4.14: Velocity profiles of reticulated foam 1 at the cross-section orthogonal to the z direction. The two velocity profiles for each component show good qualitative agreement. No voxels exhibit a difference in direction. Only a few voxels show a difference in magnitude.

We also plot the velocity histogram in both the negative and positive direction for the x and y components in Figure 4.16 to assess the prediction in the direction of the velocity. The velocity fields in both positive and negative directions show excellent agreement at high velocity and demonstrate that the model can predict the velocity directions. But the model shows larger deviation at low velocity.

We emphasize that the required computational time and resources are significantly reduced by using our deep learning model. In this chapter, a complete CFD simulation required approximately 6 hours of CPU time (Intel Xeon Gold 6128) and maximum of 16 GB of memory. On the other hand, our deep learning model only required 11 seconds and a negligible

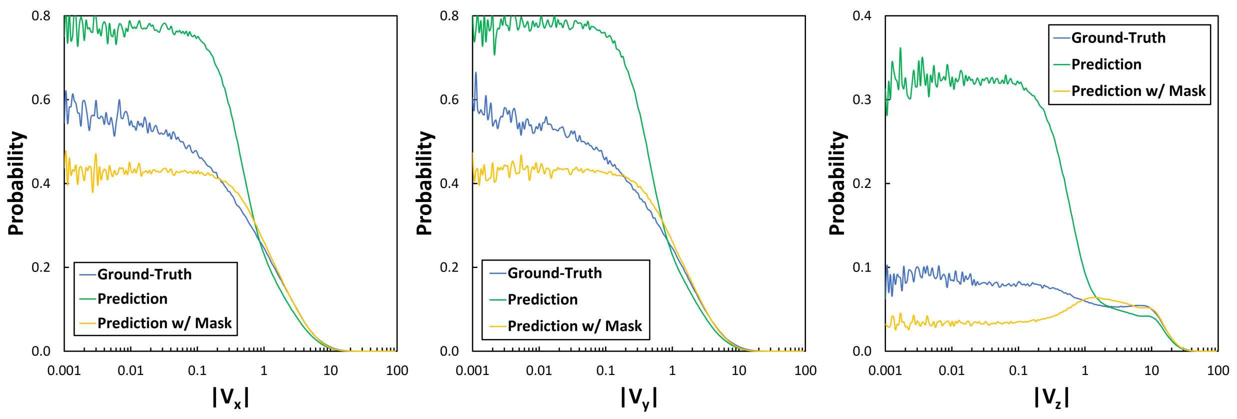


Figure 4.15: Distributions of the absolute ground-truth and predicted velocity. The comparison shows excellent agreement at high velocity and disagreement at low velocity. We apply a binary mask corresponding to the solid to eliminate errors at low velocity.

amount of memory on the same CPU. With a GPU equipped workstation, the inference time of our deep learning model is significantly reduced, taking less than a second, as compared to the order of ten minutes required by GPU-based LBM-RMT [103]. The deep learning model provides a significant advantage in conducting optimization and parametric study of porous media where computational speed is crucial.

4.4.4 Effect of the PIMSE

We train an equivalent model using the MSE loss function to compare and analyze the effect of our PIMSE loss function (Equation (5.5)). Figure 4.17 shows the divergence of the predicted velocity field for the reticulated foams. We find that it is significantly reduced by almost a factor of 10 when using the PIMSE loss function. Figure 4.18 shows the STAFE of models with different loss functions. We observe a slight improvement of 6% on average when using the PIMSE. Figure 4.19 illustrates the distribution of the velocity field. The velocity distributions, however, do not show a significant difference between the two models. We hypothesize that it may be due to the lack of depth and complexity of our model. The loss curve throughout the learning process (Figure 4.20(a)) indicates a slight under-fit. However, increasing the depth and complexity would exceed our available computational limit. Figure

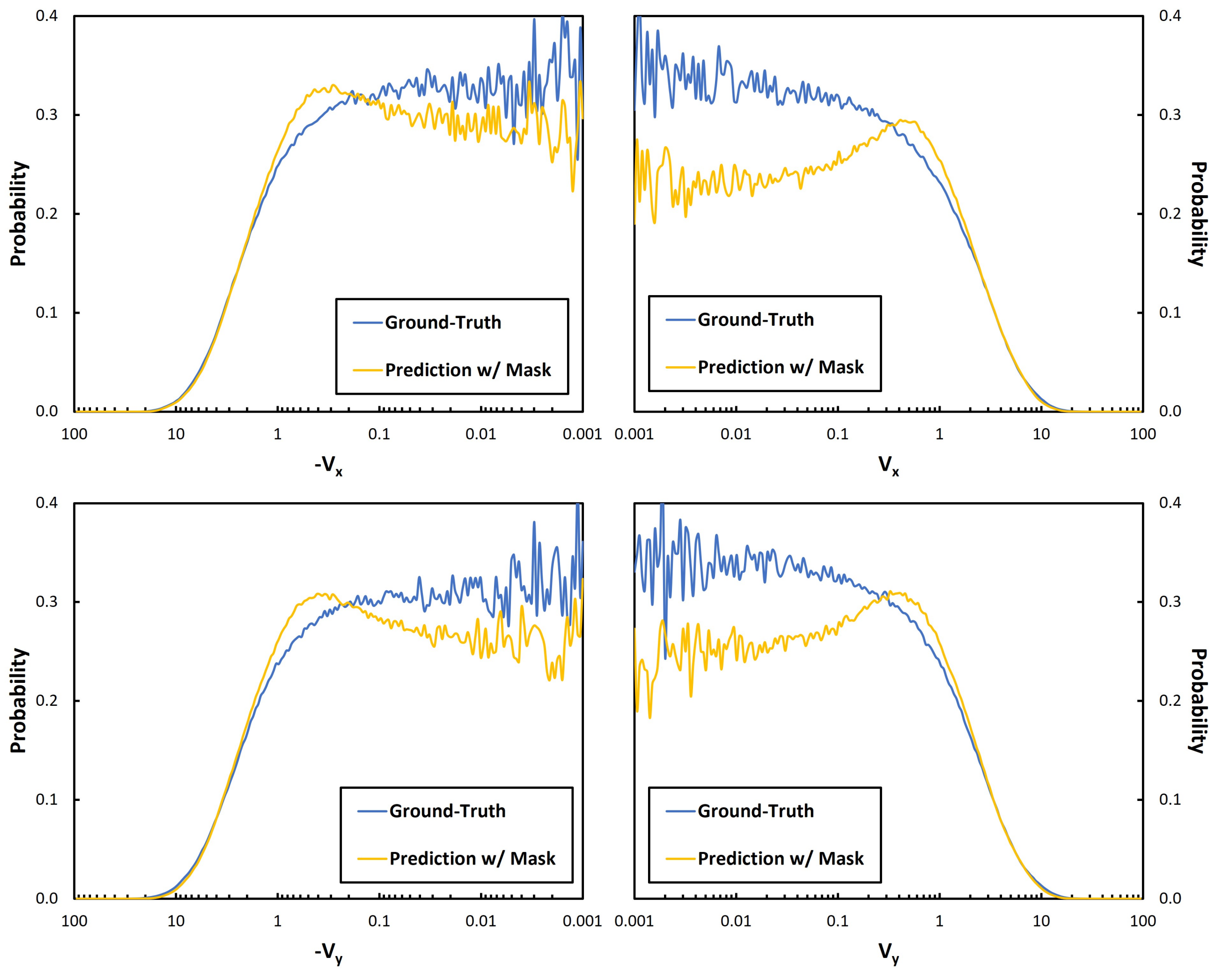


Figure 4.16: Distributions of the ground-truth and predicted velocity in span-wise direction. An excellent agreement is shown at high velocity. Model shows larger deviation at low velocity.

4.20(b) shows that the final loss values for the two models are comparable suggesting that the model with MSE loss function may already predict close to the expectation for the given complexity and depth of the model. The values for PIMSE are slightly higher due to the addition of the divergence free condition.

Another possible reason is due to the coarse uniform used in our deep learning model. In this work, a typical size of the computational domain for CFD simulations required approximately 5 million unstructured grids due to the complex geometry and tortuous paths. On the contrary, the computational domain used for our deep learning model consisted only of approximately 2 million uniformly structured grids. Figure 4.21 compares the number

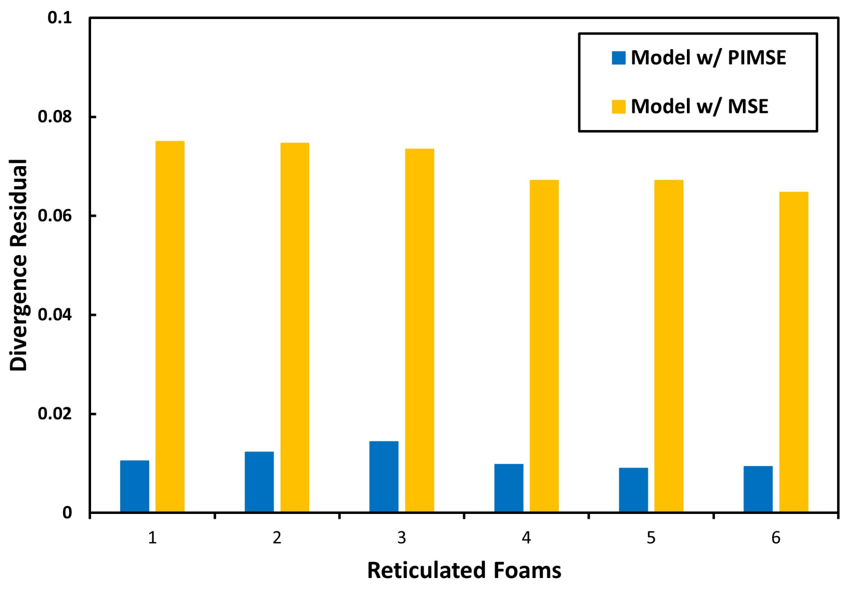


Figure 4.17: Divergence of the velocity field between models with different loss functions. We observe an order of magnitude decrease when the PIMSE is utilized as the loss function.

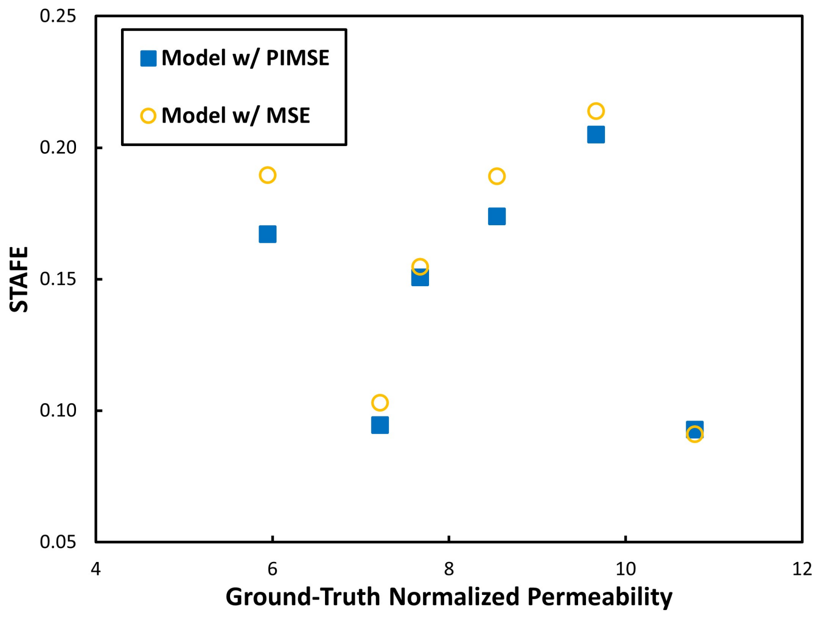


Figure 4.18: The STAFE of the predicted velocity field for models with different loss functions for the reticulated foams. STAFEs improved by an average of 6% when using the PIMSE.

of elements as a function of velocity. The 120 × 120 × 120 computational domain size of the deep learning model is inadequate to fully represent the computational domain of CFD simulations, especially at moderate velocity magnitudes. The pixelation [48] caused by the

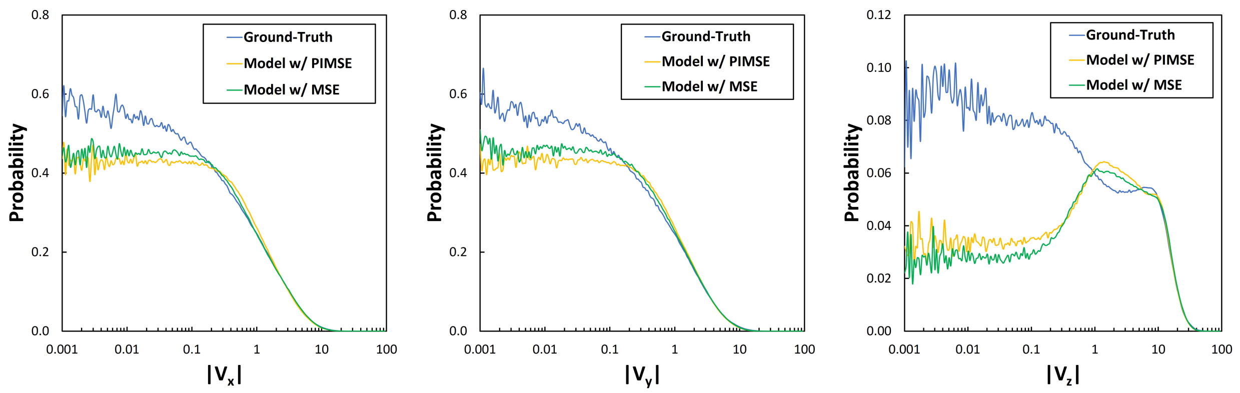


Figure 4.19: Distributions of the absolute ground-truth and the absolute predicted velocity fields using models with different loss functions. No significant difference is observed between the two models for all velocity components.

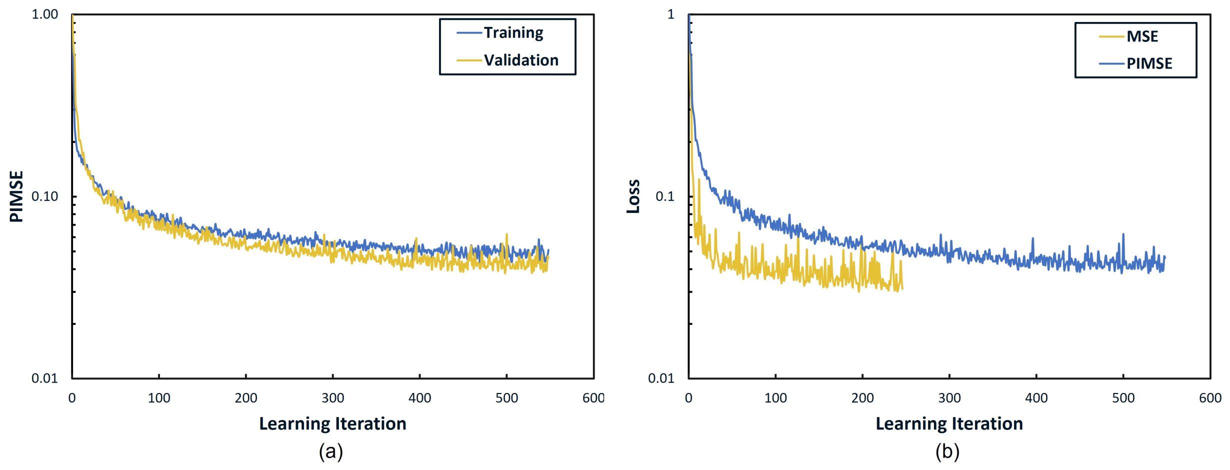


Figure 4.20: (a) Averaged loss curve for three identical models with the PIMSE loss function trained with different initialization states. It indicates a slight under-fit due to the lack of complexity and depth of our model. (b) Loss curve of the validation data for two models. The final loss values are comparable

transition to uniform structured grid and the reduction in the total number of elements may hinder accurate enforcement of the divergence free condition in complex geometries. While increasing the voxel resolution of the training data may enhance the benefits of PIMSE, it would exceed our computational capacity. We note that [84] proposed a multi-scale approach that performs inferences on a larger domain size than the training data and thereby could circumvent this limitation. They demonstrated accurate predictions of permeability for domain sizes of up to 512^3 from a training dataset of 256^3. However, it is worth noting

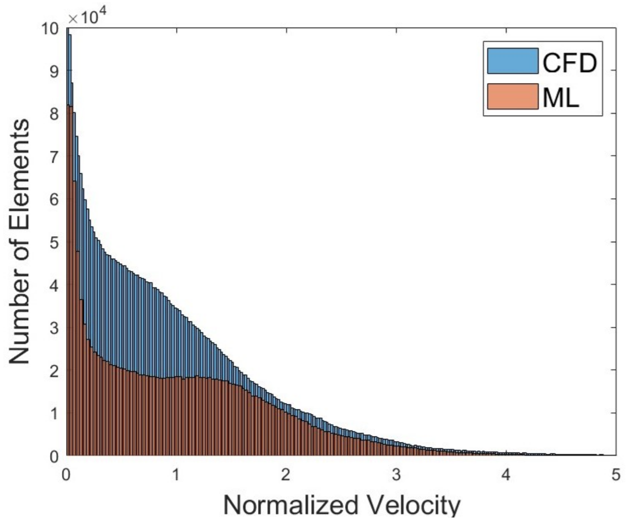


Figure 4.21: Histogram comparing the number of elements of computational domains for CFD and the deep learning model as a function of the velocity. The deep learning model lacks grid resolution, especially at moderate velocities, which could hinder accurate enforcement of divergence free condition.

that this approach may impact the effectiveness of PIMSE and requires further detailed investigation.

Nevertheless, we demonstrate that incorporating PIMSE significantly reduces the divergence of the velocity fields and improves the STAFE by an average of 6%. Our results demonstrate that physical laws can be enforced in the loss function to guide the learning process of image-based CNN models.

4.4.5 Heat Transfer Analysis

Here, we capitalize on the advantages of our deep learning model to perform heat transfer analysis of the reticulated foams as presented in Figure 4.2. We first performed similar CFD simulations (ANSYS Fluent 18.2) to obtain the ground-truth temperature fields of the reticulated foams. We numerically solved the Equation of conservation of energy under steady-state, negligible viscous dissipation, and no volumetric heat generation assumptions:

$$\rho C_p \left(\mathbf{V} \cdot \nabla T\right) = \nabla \cdot \left(k \nabla T\right) \tag{4.8}$$

where C_p is the fluid specific heat, T is the temperature, and k is the fluid thermal conductivity. We also assumed that the fluid properties are independent of temperature. We chose the fluid properties such that the Peclet number, defined as

$$\mathrm{Pe} = \frac{\rho C_p}{k} V_z D_p \tag{4.9}$$

where D_p is the average pore diameter, is approximately on the order of thousands to emphasize the effect of convection. Hot and cold constant temperature boundary conditions were applied at the solid surfaces and at the inlet, respectively. Symmetry boundary condition was applied to the side surface parallel to the stream-wise direction. Computational domain and boundary conditions for the numerical simulation are shown in Figure 4.22. We

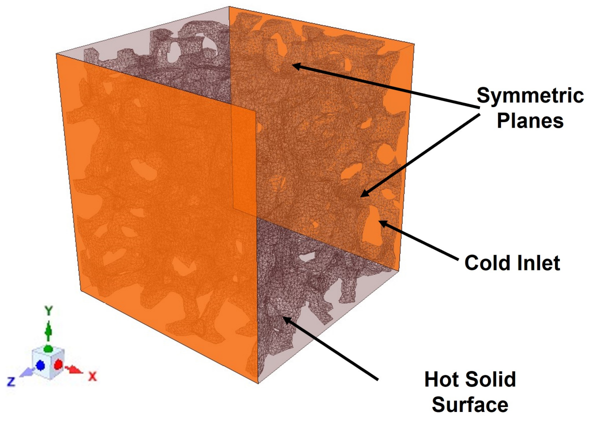


Figure 4.22: Computational domain for heat transfer analysis. Hot and cold constant temperature boundary conditions were applied at the solid surfaces and at the inlet, respectively. Symmetry boundary condition was applied to the side surface parallel to the stream direction.

considered a heat transfer problem where the velocity field is fully developed before energy with the porous media. This type of problem is relevant in many thermal applications such as in heat-exchanger systems.

We first compare the Nusselt number, Nu, for each of the reticulated foams defined as

$$Nu = \frac{hD_p}{k} \tag{4.10}$$

where h is the overall heat transfer coefficient defined as

$$h = \frac{\rho \overline{V} A_c}{A_s} \frac{\overline{T_o} - \overline{T_i}}{T_\infty - T_s} \tag{4.11}$$

where $\bar{V}$ is the average stream-wise velocity, A_c is the cross-sectional area, A_s is the solid surface area, $\overline{T_o}$ and $\overline{T_i}$ are the mean fluid temperature at the outlet and inlet, respectively, T_∞ is the cold inlet temperature, and T_s is the hot solid surface temperature. The Nusselt number describes the volume-averaged thermal performance of the porous media. Figure 4.23 presents the comparison of the Nusselt number obtained using the ground-truth and predicted velocity. We find a good agreement in the Nusselt number with an average error

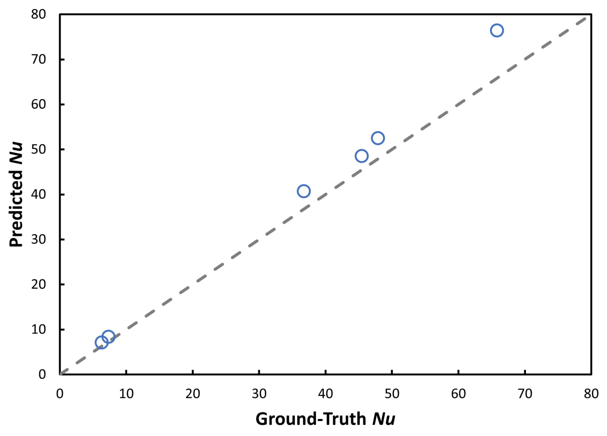


Figure 4.23: Comparison of the Nusselt number obtained using the ground-truth and predicted velocity. It shows good agreement with an average error of 11.6%.

of 11.6%. The Nusselt numbers show a slight over-prediction due to the over-prediction in the velocity, especially for the x and y components (Figures 4.15 and 4.16).

We now consider the accuracy of temperature fields at the pore-scale. Figure 4.24 illustrates the temperature fields at the cross-section parallel to the z direction for reticulated foam 2 which had the largest STAFE error. The corresponding planar ground-truth and predicted velocity fields are also presented. The temperature fields show good qualitative agreement. However, noticeable number of voxels show a difference due to the errors in the velocity field. To investigate in more detail, we compare the temperature profile at the center

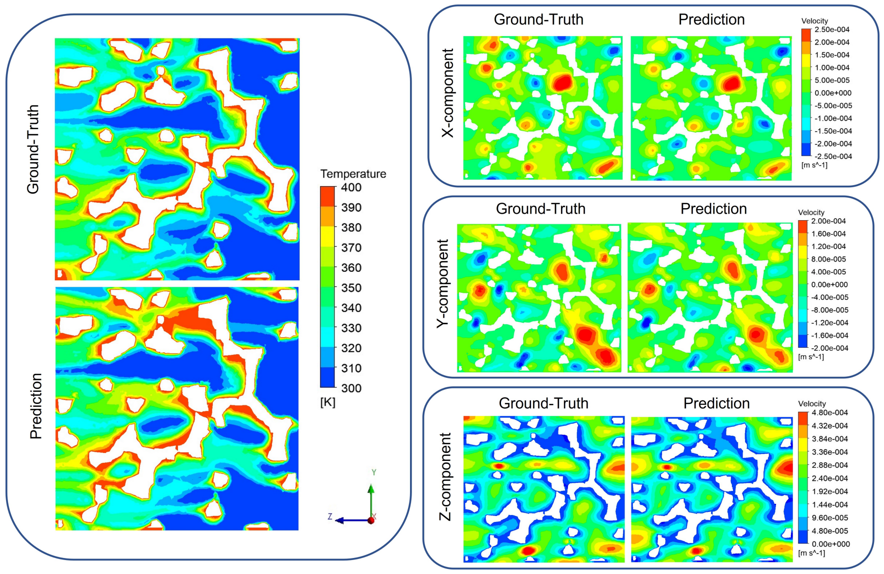


Figure 4.24: Temperature profile at the cross-section parallel to the z direction obtained using the ground-truth and predicted velocity fields. Corresponding planar ground-truth and predicted velocity fields are also shown. Temperature profiles show good qualitative agreement.

line through the computational domain in the stream-wise direction for all reticulated foams in Figure 4.25. The temperature fields using deep learning model adequately characterizes the overall transport of energy for all reticulated foams. However, we observe voxel-wise errors where the temperature values are both under and over predicted. This behavior is also reported by [102] where the voxel-wise errors in the velocity field cause under- and over-accumulation of species concentration in solving the mass transport problem.

Nonetheless, we drastically increase the computational speed to perform heat transfer analysis of the reticulated foams while demonstrating good agreement in the Nusselt number and temperature fields. Our approach can expand to other transport analysis where the knowledge of the velocity field is crucial such as filtration, solute transport, and mass transfer.

4.5 Summary and Conclusion

In this chapter, we designed a deep learning model to predict the velocity fields of reticulated foams from only their binary images and incorporated a physics-informed loss function

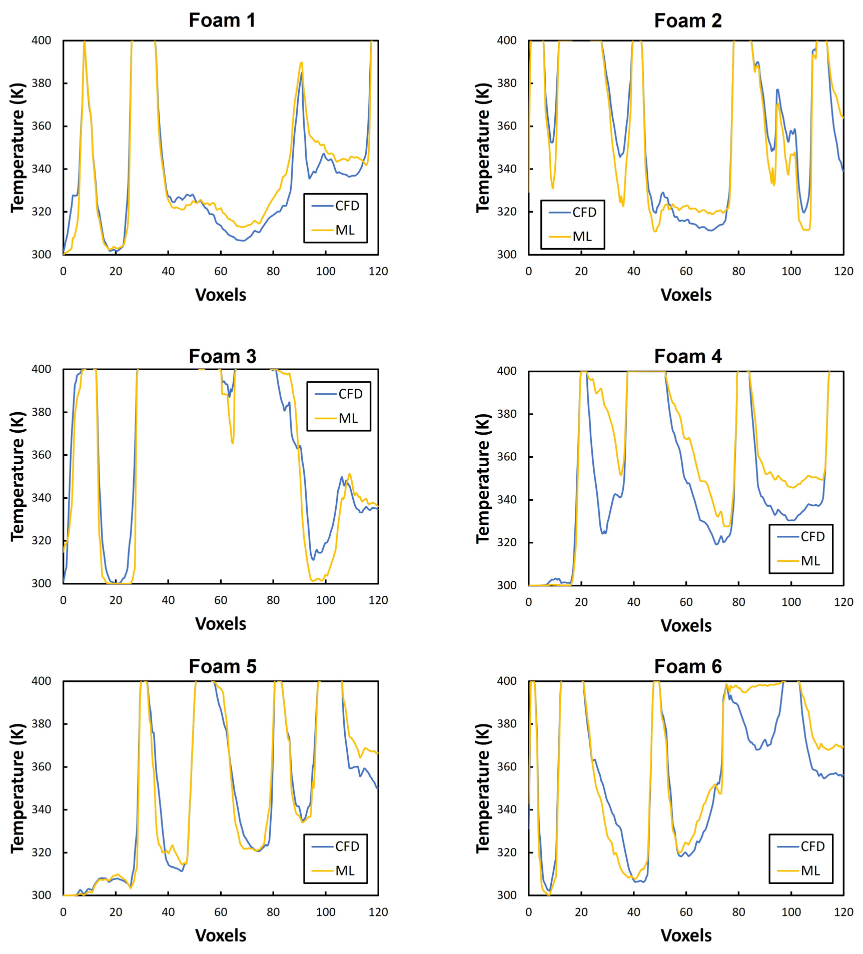


Figure 4.25: Temperature distribution at the center line of the computational domain (x = 60 voxels and y = 60 voxels) in the stream-wise direction. Temperature profiles qualitatively agree well. However, the voxel-wise errors in the predicted velocity fields directly translate to the voxel-wise errors in the temperature fields.

to enforce the law of mass conservation for incompressible flows. We demonstrated that our model, trained only with synthetic porous media, showed excellent accuracy in permeability with less than 6% and in STAFE with less than 0.2 although it decreased at the pore-scale. We also showed that our physics-informed loss function significantly reduced the divergence of the velocity fields by a factor of 10 and improved STAFE by 6% although the improvement in the was minor at the pore-scale. We further illustrated that our deep learning model provides accurate velocity fields as inputs to subsequent heat transfer analysis. We obtained an average error of 11.6% for the Nusselt number, but the voxel-wise error in the velocity field directly affected those in the temperature fields.

We also demonstrated a significant reduction in the amount of computational resources and time required to characterize the flow and transport through complex porous media. We shortened the computation time from more than 6 hours to 11 seconds. Our approach is advantageous in parametric and optimization for various engineering applications where hydrodynamic and transport behavior are essential such as filtration, solute transport, multiphase flow, and mass transfer.

Chapter 5

Deep Learning Prediction of Pore-scale Flow Using Periodic Structures

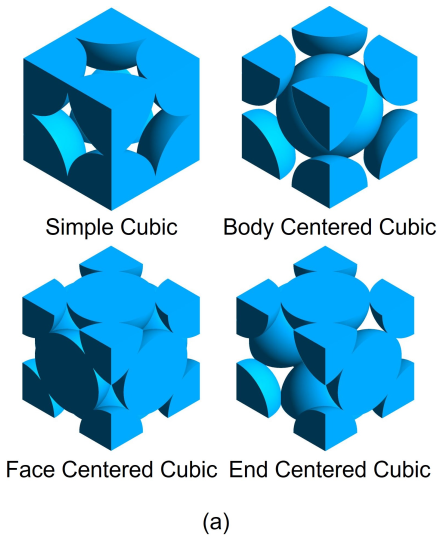
Past studies, and our results in Chapter 4 suggest that the CNN architecture is promising in predicting pore-scale flow fields in complex heterogeneous porous media. We have demonstrated a significant reduction in the amount of computational resources and time required to characterize the flow and transport through complex porous media. We shortened the computation time from more than 6 hours to 11 seconds while demonstrating good accuracy in the results. However, training a DL model requires a large set of accurate numerical simulation data, which are computationally very expensive to obtain, to serve as ground-truth. For example, Santos et al. [84] devoted five hours of computing cluster with 96 cores to create training data. A reasonable compromise may be to train DL models using simulation results from periodic unit cells which can be obtained analytically or numerically at a very small computational cost. Many engineering applications, including thermal insulation [110], heat exchangers [9, 40], and filtration [8, 98], utilize cellular structures, such as reticulated foams, whose microstructures exhibit a certain level of periodicity. Pores of the reticulated foams take the shape of Kelvin's tetrakaidecahedron in periodic manner to maximize space-filling. But other factors influence the orientation and connectivity of the pores, introducing heterogeneity [24].

In this chapter, we investigate the ability of CNN models trained in this manner to predict pore-scale velocity fields in complex heterogeneous porous media. We generate a set of training data using unit cells containing different sizes of mono-sized spherical pores in simple cubic, body-centered cubic, face-centered cubic, and end-centered cubic arrangements. We also consider more complex unit cells derived from crystal lattice structures. Accurate flow fields in periodic unit cells can be obtained analytically or numerically at a very small computational cost. We utilize synthetic porous media consisting of randomly dispersed spherical pores and real reticulated foams to demonstrate that our DL model accurately predicts the permeability and pore-scale velocity field of both porous media. Finally, we explore the potential of using the predicted velocity fields from our DL model as initial guesses to speed up the convergence of numerical simulation.

5.1 Periodic Unit Cell

We obtain training data for our DL models using periodic unit cells with mono-sized spherical pores. We generate four basic periodic unit cells using the simple cubic, body-centered cubic, face-centered cubic, and end-centered cubic arrangement, as illustrated in Figure 5.1(a). Starting from the minimum pore radius, we systematically increased the radius to generate data with uniformly distributed porosity. A total of ten unit-cells are generated for each arrangement with the porosity varying from 0.5 to 0.95. The corresponding permeability values, obtained from the numerical simulation described in Section 5.2, range from $5 \times 10^{-11}\text{m}^2$ to $2 \times 10^{-7}\text{m}^2$ as shown in Figure 5.1(b).

We also generate four additional complex unit cells that consist of poly-sized spherical pores arranged in different lattice structures, which are intended to further emulate het-



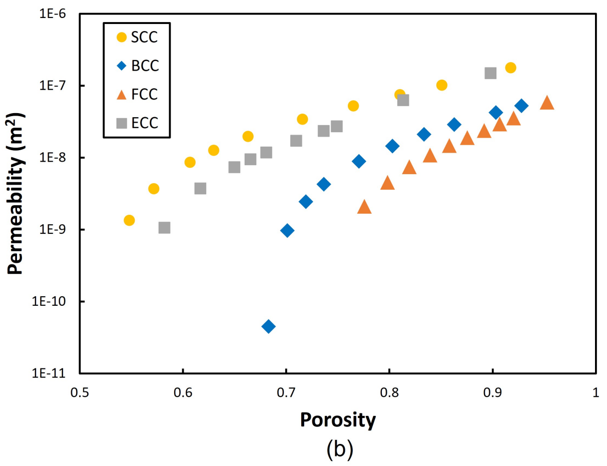
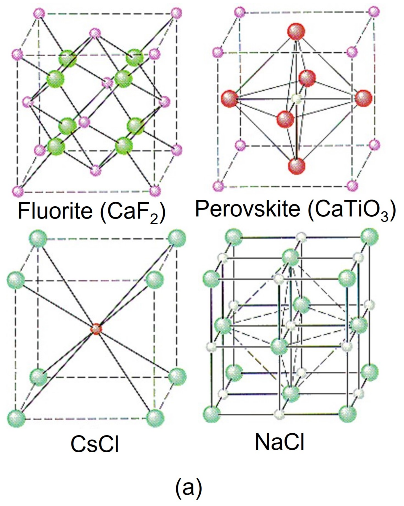


Figure 5.1: (a) Four basic unit cells used to generate the training data. The blue spheres represent the pores. A total of 10 unit cells were generated for each arrangement. (b) The porosity and the predicted permeability for the basic unit cells. The pore sizes are systematically varied to obtain uniform range of porosities.

erogeneity of porous media. The lattice structures we use are those of fluorite (CaF2), perovskite (CaTiO3), cesium chloride (CsCl), and sodium chloride (NaCl), as illustrated in Figure 5.2. The fluorite arrangement introduces spherical pores at the (±1/4 ,±1/4, ±1/4)



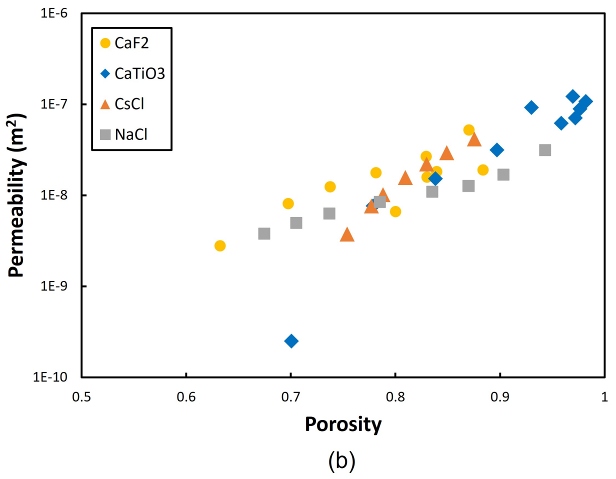


Figure 5.2: (a) Four complex unit cells that we use to generate training data. The spheres represent the pores. A total of 10 unit cells are generated for each lattice structure. Reprinted from [Unit Cells, 2023][26] (b) The porosity and the predicted permeability for the complex unit cells. The pore sizes are systematically varied to obtain uniform range of porosities.

location of the unit cell with two different pore radii. The perovskite arrangement combines the body-centered cubic and face-centered cubic arrangements and has three different pore radii. Similarly, the sodium chloride arrangement combines the body-centered cubic and the face-centered cubic arrangements with additional pores on each edge of the unit cell and has two different pore radii. Lastly, the cesium chloride arrangement is a body-centered cubic arrangement with two different pore radii. We again start from the minimum pore radius and systematically increase the radius to obtain data with uniformly distributed porosity. We change the size of one pore while fixing the other and repeat the procedure for all pores. A total of ten complex unit cells are generated for each type of arrangement with the porosity ranging from 0.6 to 0.98 which corresponds to permeability, obtained from the numerical simulation described in Section 5.2, of $2 \times 10^{-10}\text{m}^2$ to $1 \times 10^{-7}\text{m}^2$ as illustrated in Figure 5.2(b).

We vary the number of pores in each computational domain to produce additional training data. A single unit cell of a given arrangement is repeated two or four times in each direction to vary the number of pores and thereby the physical length scale of the pores, as illustrated in Figure 5.3. For example, a domain where unit cells are repeated twice in each direction

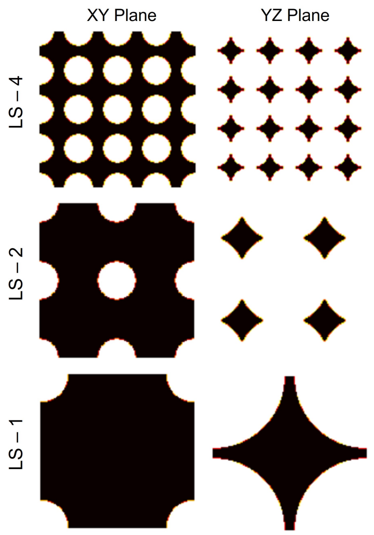
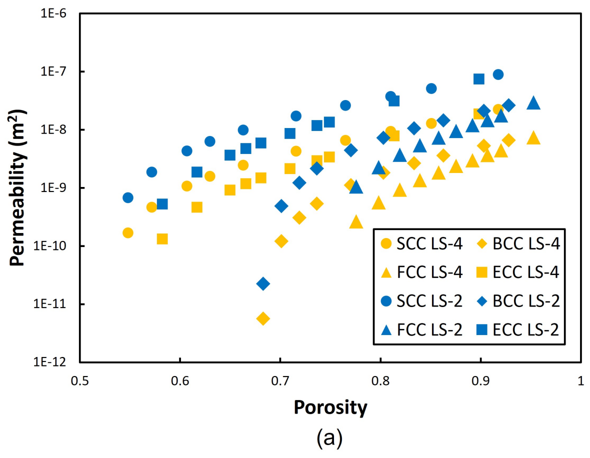


Figure 5.3: Cross-sectional binary images of simple cubic unit cells repeated two or four times to vary the number of pores. The black represents the solid, and the white represents the pore. They are denoted with the corresponding number of repeats.

is denoted as LS-2. Since our computation domain remains periodic, numerical simulation is performed using only a single unit cell to obtain flow fields. Figure 5.4 shows the the predicted permeability as a function of the porosity for the computational domains with different numbers of pores. A much wider range of permeability values, some as low as



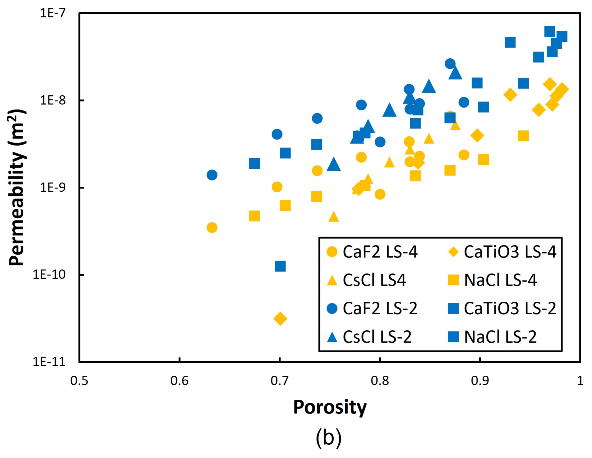


Figure 5.4: Plot of porosity and permeability for (a) the four basic arrangements and (b) the four complex arrangements after modifying the length scale by repeating the unit cell two and four times in each direction. Change in the length scale allows a wider variation in the permeability as low as 1×10^{-11}.

$1 \times 10^{-11}\mathrm{m}^2$, is obtained. We down-select unit cells with permeability values within one order of magnitude from those of the heterogeneous porous medium described in Section 4.3. The additional training data also help represent wider varieties of pore-scale flow characteristics, which in turn helps improve the accuracy of 3D velocity fields predicted using our DL models for complex heterogeneous porous media as discussed further in Section 5.5. To assess the accuracy of our DL model for complex, we utilize the synthetic porous media and reticulated foams used in Chapter 4.

5.2 CFD Simulation

We generate ground-truth velocity fields to train and test our DL models using numerical simulation outlined in Chapter 3. Since the unit cells are periodic, the computational domain did not required to be mirrored. Our numerical simulation domain and boundary conditions for periodic unit cells are summarized in Figure 5.5.

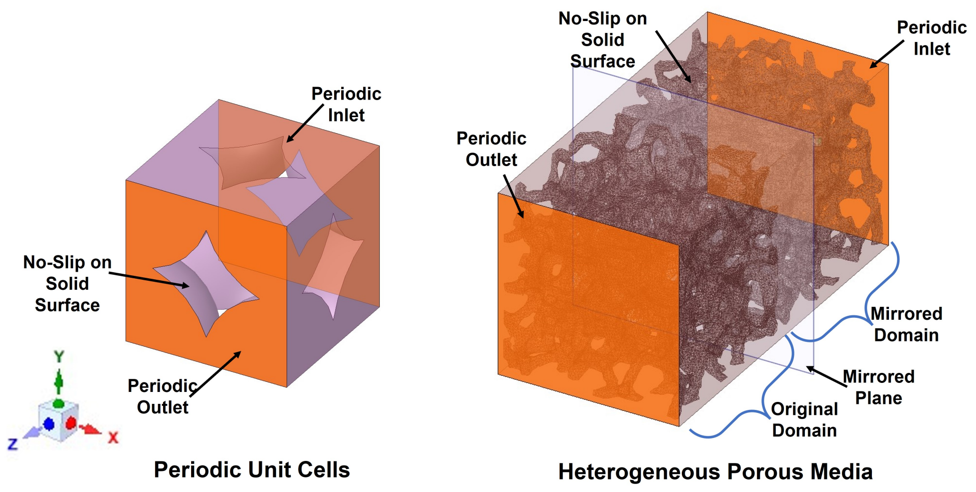


Figure 5.5: Numerical simulation domain for periodic unit cells. The primary flow direction is in the positive z direction.

The same convergence and grid independence criteria as those described in Chapter 3 were used. We note once again the computational efficiency of simulating flows in unit cells. A numerical simulation domain of a single unit cell requires less than 1% of the total number of grid points than that used for our reticulated foams. A numerical simulation run for the former only takes a few seconds, whereas a numerical simulation for the latter takes up to 7 hours on the same workstation.

5.3 CNN Architecture

Our neural network employs the convolutional U-Net architecture [80]. It utilizes a series of convolutional layers that is stacked with pooling layers to extract hierarchical features and to perform mapping from the input to the output. The present CNN model $\mathcal{F}$, shown in Figure 5.6, takes a 3D binary data X that represents the microstructure of a porous media as its input and outputs a 3D velocity field Y using the trained $\mathbf{p}_t$ and non-trainable $\mathbf{p}_n$

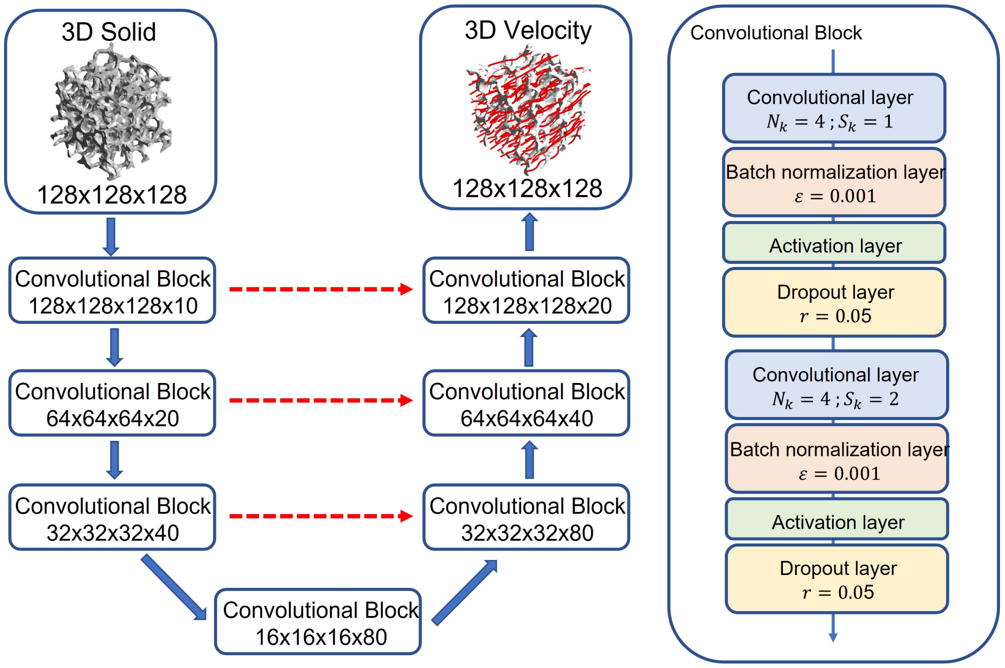


Figure 5.6: The architecture of the CNN model used in the present study to perform mapping between the 3D binary data that represents the microstructure of a porous media and the 3D velocity field

hyperparameters:

$$Y = \mathcal{F}\left(X; \mathbf{p}_t, \mathbf{p}_n\right) \tag{5.1}$$

We use the convolutional, batch normalization, nonlinear activation, and dropout layers in series to build each convolutional block. The convolutional layer performs the convolutional operation:

$$\mathbf{y} = \mathbf{k} * \mathbf{x} + \mathbf{b}_c \tag{5.2}$$

where $\mathbf{x}$ is the input, $\mathbf{y}$ is the output, $\mathbf{k}$ is the 3D convolutional kernel with a size of $N_k \times N_k \times N_k$ and a stride of S_k, and $\mathbf{b}_c$ is the bias. The batch normalization layer [41] applies a transformation that maintains the mean and the standard deviation of the output close to 0 and 1, respectively:

$$\mathbf{y} = \gamma \frac{\mathbf{x} - \bar{\mathbf{x}}}{\sqrt{\mathbb{V}\left(x\right) + \epsilon}} + \mathbf{b}_b \tag{5.3}$$

where γ is the scaling factor, $\bar{\mathbf{x}}$ is the mean of the input, $\mathbb{V}$ represents the variance, and ϵ

is a small constant used to avoid division by zero. The nonlinear activation layer applies a nonlinear activation function $\sigma(x)$. We use the rectified linear unit (ReLU) activation function for stream-wise velocities and the scaled exponential linear unit (SeLU) activation function for span-wise velocities, which can be negative. Finally, the dropout layer randomly sets a fraction, r, of the values to zero to prevent over-fitting of the model.

For each convolutional block, we incorporate two iterations of the group of layers described above. We set $S_k = 2$ for the second convolutional layer in each block such that it also performs either the down-sampling or the up-sampling operation. The encoding and decoding branches of our U-Net consist of four down-sampling and up-sampling levels. Each level consists of a single convolutional block that is skip-connected between the branches at the same level.

5.4 Model Training

We train our CNN models using the training data discussed in Section 5.1 through Section 5.2. Two separate sub-models are trained to predict either the stream-wise (z) or span-wise (x and y) velocity fields. For brevity, we refer to the combination of these two submodels as one DL model. A summary of our models that are trained using different subsets of the training data is presented in Table 5.1.

We first train a baseline model (model A) using the simple unit cells with LS-4 to be consistent with the average number of pores in a representative elementary volume of the reticulated foams. We add complex unit cells for model B. We add domains with different numbers of pores to model C. Finally, model D includes both simple and complex unit cells and two different physical pore length scales in the training data.

We augment our training data further through 3D image rotation and shift operations to create 200 training data and 20 validation data for each model. The size of a single data is $128 \times 128 \times 128$. We apply random rotations between 0 and 90 degrees and shifts between

Table 5.1: A summary of training data that we use to train different deep learning models.

Name	**Direction**	**Arrangement**	**Length scale**
Model A-1	Stream-wise	Basic	LS-4
Model A-2	Span-wise	Basic	LS-4
Model B-1	Stream-wise	Basic + Complex	LS-4
Model B-2	Span-wise	Basic + Complex	LS-4
Model C-1	Stream-wise	Basic	LS-4 + LS-2
Model C-2	Span-wise	Basic	LS-2 + LS-1
Model D-1	Stream-wise	Basic + Complex	LS-4 + LS-2
Model D-2	Span-wise	Basic + Complex	LS-2 + LS-1

0 and $L/2$ for the stream-wise models. For the span-wise models, we only perform shift operations between 0 and $L/8$. The velocity fields are normalized using

$$V = \frac{V^* - c_{min}}{c_{max} - c_{min}} \tag{5.4}$$

where V^* is the velocity, V is the normalized velocity and c_{min} and c_{max} are constants. To keep the maximum magnitude of the velocity below 10, we use $c_{max} = 3 \times 10^{-9}$ for the stream-wise velocity, $c_{max} = 1.6 \times 10^{-9}$ for the span-wise velocity, and $c_{min} = 0$ for both velocity directions.

We use the mean squared error function as our loss function [109, 36, 82, 83, 102, 45]:

$$\text{Loss} = \frac{1}{N} \sum_{i}^{N} (V_i - V_{t,i})^2 \tag{5.5}$$

Here N is the total number of voxels in a batch of training data, V is the predicted velocity, and V_t is the ground-truth velocity obtained from the CFD simulations. It is often considered advantageous to optimize a model that predicts pore-scale velocity fields using the mean squared error function because it puts more emphasis on the higher magnitude velocities that govern the dominant flow paths and scalar transport in porous media.

We tune the hyperparameters using discrete grid search for the baseline model (model A). The same hyperparameters are used for other models. For training, we use Adam optimizer

[51] with a constant learning rate of 7×10^{-5}. We use NVIDIA A100 graphics processing units with 80 gigabytes of RAM. We use a mini-batch size of four and apply early stopping criterion with 50 learning iterations to prevent preemptive stoppage and over-fitting. Each training run did not exceed 5 hours.

5.5 Results and Discussion

5.5.1 Macro-scale Performance

We first quantify the accuracy of our DL models at the macro-scale by comparing the predicted and ground-truth permeability (Figure 5.7). All models predict the permeability

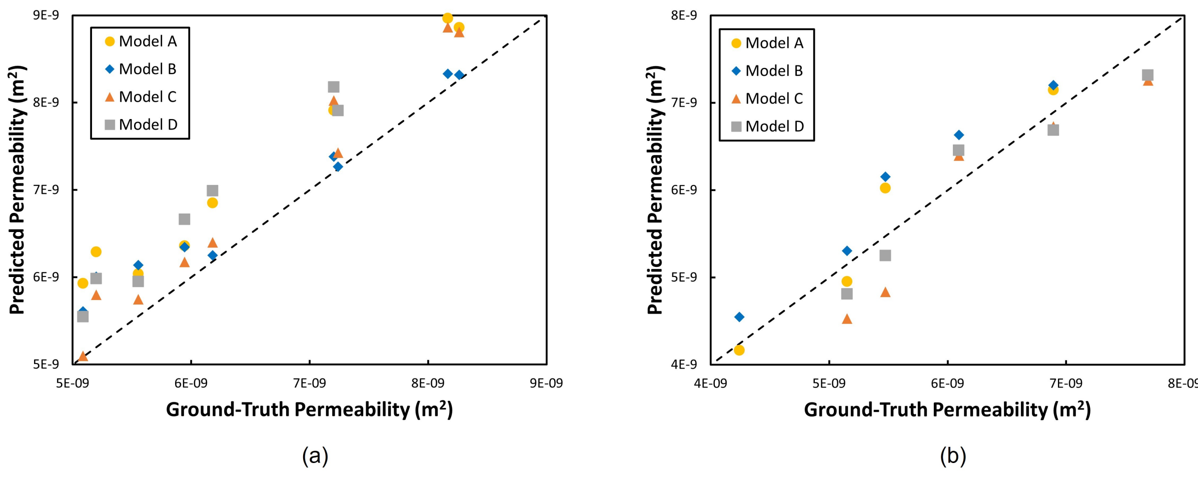


Figure 5.7: A comparison of predicted and ground-truth permeability for all (a) synthetic porous media and (b) reticulated foams. The errors in the permeability of the two porous media are similar.

with good accuracy. The average percent errors for synthetic porous media and reticulated foams are less than 10% and 9%, respectively, across all models. Despite the differences in the training data, all models show similar accuracy in predicting the permeability with the maximum difference in error below 5% and 4% for synthetic porous media and reticulated foams, respectively. However, we observe slightly lower accuracy for synthetic porous media. A possible explanation is that periodic unit cells better represent the structure of reticulated foams, which is quasi-periodic [24]. This is consistent with past studies [83, 84], although the difference in our work is small, that reported larger deviation when predicting porous media that are different from the training data.

We also compare the scaled total absolute flow error (STAFE) as defined by Wang et al. [102]:

$$\text{STAFE} = \sum_{m}^{x,y,z} \frac{\sum_n |q_{m,n} - q^*_{m,n}|}{\sum_k q_{z,k}} \tag{5.6}$$

where q_n is the planar flow rate vector in the direction n defined as:

$$q_{x,i} = \sum_j^N \sum_k^N V_x|_{i,j,k} \quad q_{y,j} = \sum_i^N \sum_k^N V_y|_{i,j,k} \quad q_{z,k} = \sum_i^N \sum_j^N V_z|_{i,j,k} \tag{5.7}$$

where i, j, and k represents the x, y, and z locations. The STAFE accounts for error in the predicted mass flow rate in each direction and reflects additional details of the flow on the pore scale than permeability [102]. The STAFE for all the reticulated foams are shown in Figure 5.8. The average STAFE for synthetic porous media and reticulated foams are

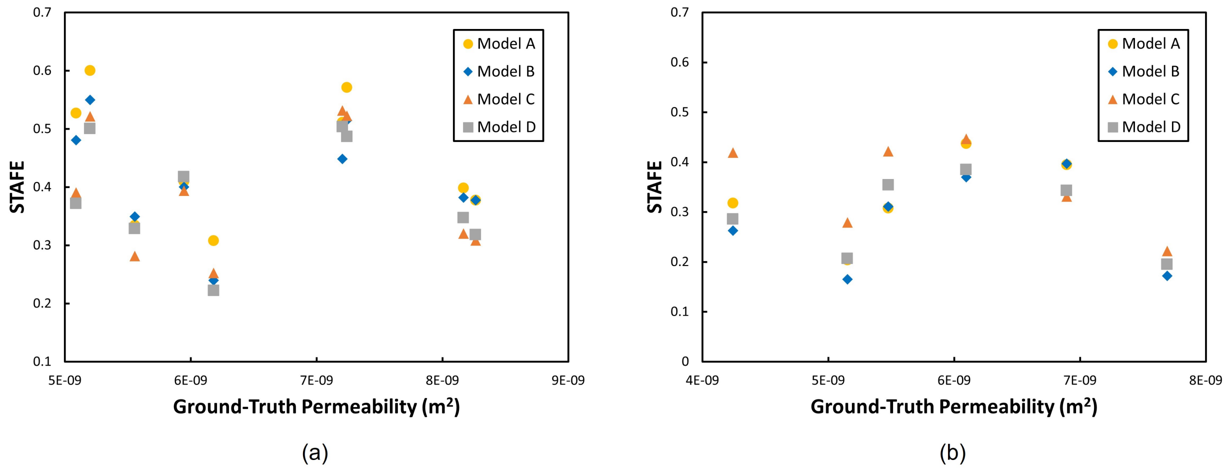


Figure 5.8: The STAFE of the predicted and ground-truth velocity fields for (a) synthetic porous media and (b) reticulated foams. The STAFEs of the two porous media are similar

0.41 and 0.31, respectively, across all models. Similar to the permeability, difference in the average STAFE among the models is relatively small, less than 0.07. We again observe slightly higher STAFE for synthetic porous media as seen in the prediction of permeability.

5.5.2 Pore-scale Performance

We next assess the accuracy of our DL models at the pore-scale. We focus our analysis on the reticulated foams as the results obtained from both heterogeneous porous media are similar. Additionally, reticulated foams are more representative of the complex heterogeneous porous media commonly utilized in various engineering applications. Figures 5.9 and 5.10 show the probability density functions of the predicted and ground-truth velocity fields in the stream-wise and span-wise directions, respectively. Only the x-direction velocity fields are presented in the span-wise direction since nearly identical results are obtained for the y-direction velocity fields.

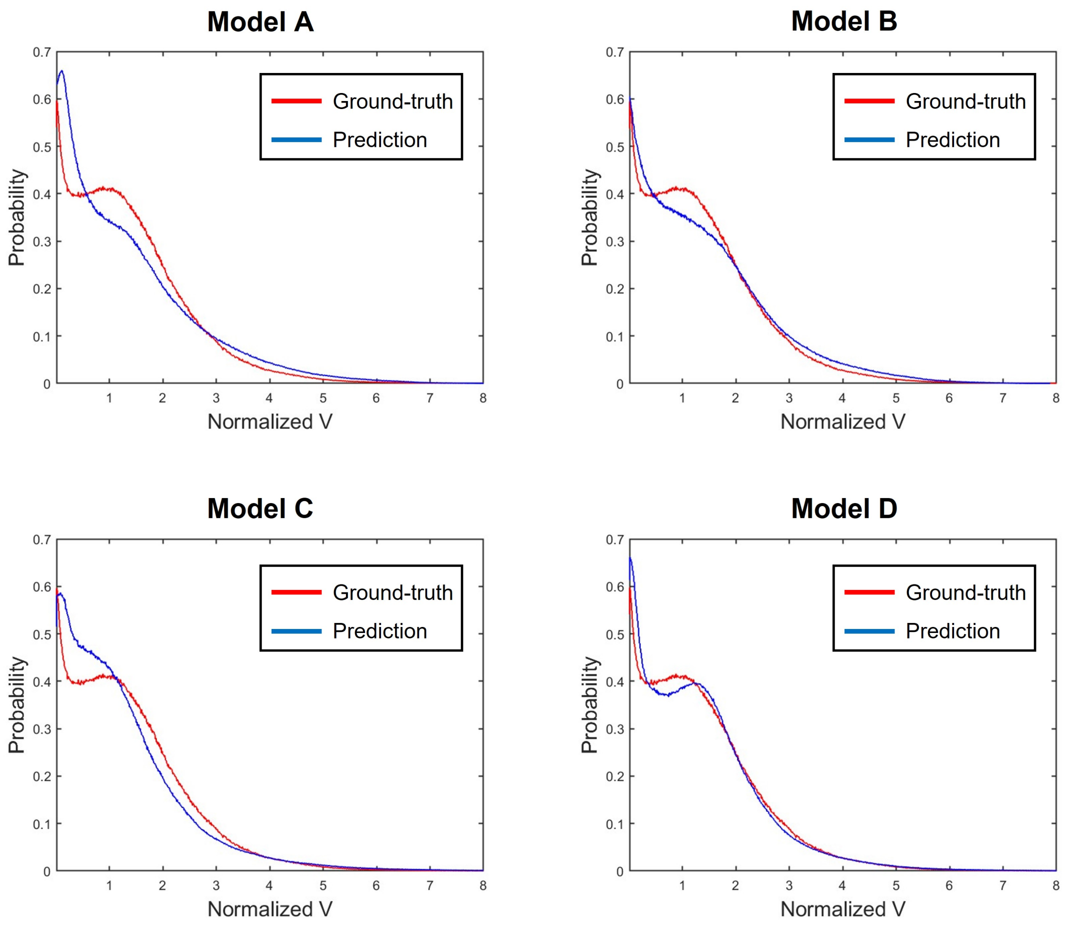


Figure 5.9: Probability density functions of the predicted and ground-truth velocity fields of all reticulated foams in the stream-wise direction

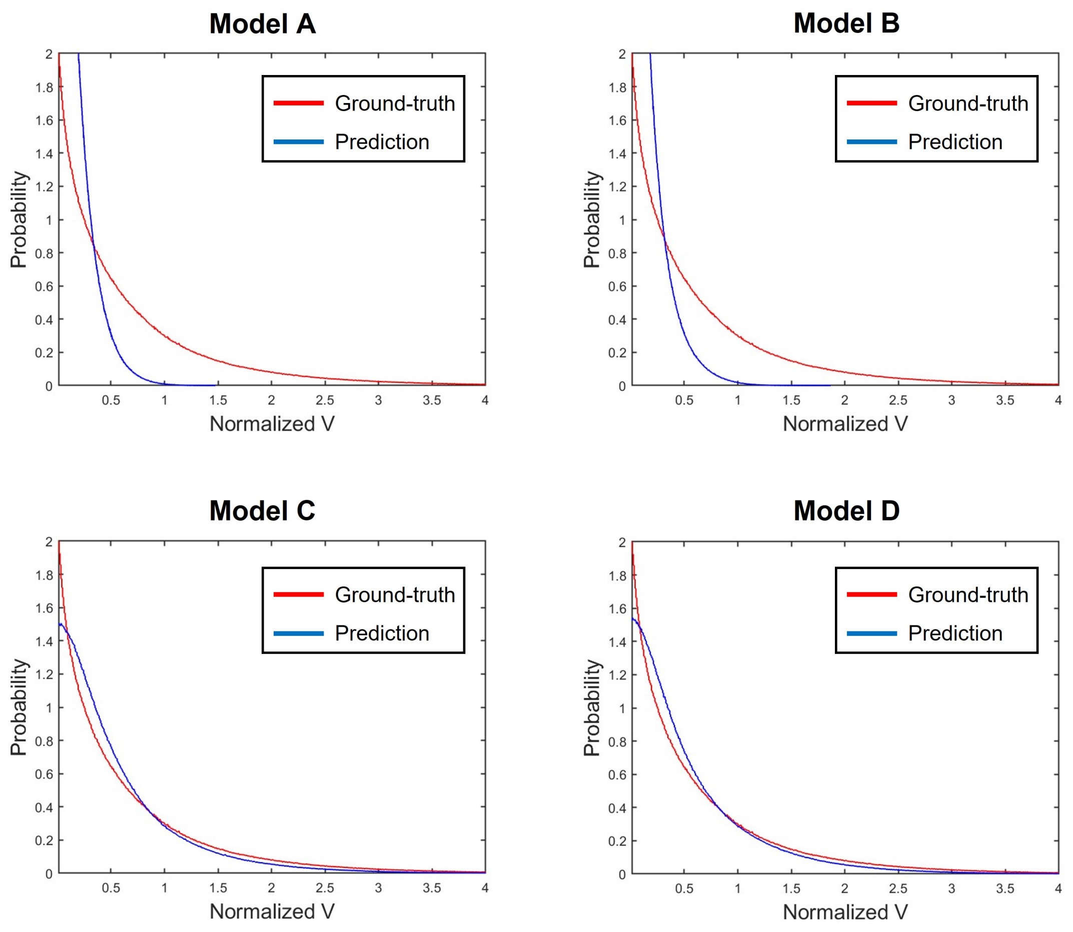


Figure 5.10: Probability density functions of the predicted and ground-truth velocity fields of all reticulated foams in the span-wise direction.

For the stream-wise direction (Figure 5.9), we find that adding the complex unit cells helps to better capture the velocity distribution, especially at small velocity magnitudes. This can be qualitatively explained by the fact that the complex unit cells provide a better representation of the pore distribution in the reticulated foams. To examine this further, we show the distribution of the maximum inscribed sphere diameter for the basic unit cells (model A) and the complex unit cells (model B) in Figure 5.11. The basic unit cells offer a relatively narrow distribution of the maximum inscribed sphere diameter. The complex unit cells offer a wider distribution, especially for smaller pores, which are better suited to capture the distribution of the reticulated foams.

Computational domains with smaller numbers of pores and hence larger physical pore

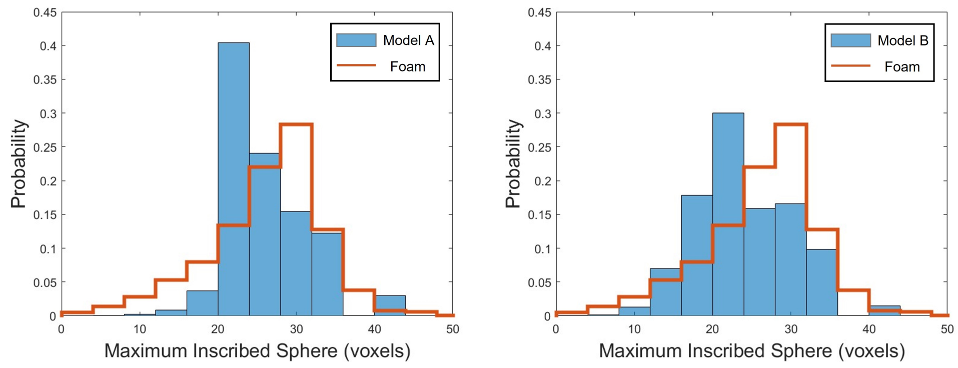


Figure 5.11: Distribution of the maximum inscribed sphere diameters for the training data with (model B) and without (model A) the complex unit cells. The distribution of one of the reticulated foams is also shown for comparison.

length scales (LS-2 or LS-1) have larger permeabilities and correspondingly wider velocity ranges. Larger pores serve as dominant high-velocity flow paths. Going from model A to model C leads to better prediction at higher velocity, but the difference is small. The maximum normalized velocity of approximately 10 for the computational domains with LS-2 is already comparable to that of approximately 8 for the computational domains with LS-4. Nonetheless, model D, which includes both training data with the complex unit cells and the larger physical pore length scales, performs best in predicting the velocity probability distribution of the reticulated foams.

In porous media with relatively high permeability, such as reticulated foams, heterogeneity in pore size, connectivity, and distribution results in large velocities even in the span-wise directions. Such relatively large span-wise velocities are not present in computational domains with LS-4. Indeed, we find significant errors in the predicted probability distributions for models A and B (Figure 5.10), which use computational domains with LS-4 for their training data. In contrast, the predicted probability density functions agree better with the ground-truths for models C and D, whose training data include those from computational domains with a smaller number of pores and hence larger physical pore length scales (LS-2 and LS-1).

We note that the larger span-wise velocity magnitudes of the reticulated foams stem mainly from the length scales of the flow path and not from constricted paths with high tortuosity. In Figure 5.12, we compare the numerically predicted streamlines for a reticulated foam and a simple cubic unit cell with different pore length scales. Qualitatively, the span-

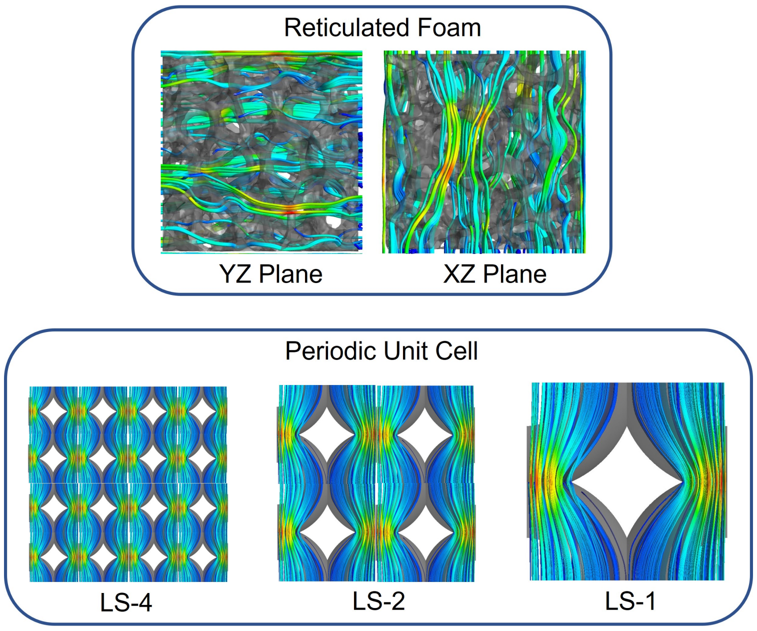


Figure 5.12: Visualization of the streamlines through the reticulated foam and the computational domain with unit cells at different physical length scales. The magnitude of the streamlines are shown in rainbow color scale with red corresponding to high velocity and blue corresponding to low velocity.

wise flow through the reticulated foams involves flow paths that have only one or two turns and whose curvatures are comparable to the size of the computational domain. Unit cells with fewer pores and larger physical pore length scales (LS-2 and LS-1) are a better representation of the span-wise flow behavior through the reticulated foams than those with smaller length scales (LS-4).

In contrast to the stream-wise direction, the addition of the complex unit cells in generating training data has little impact in the span-wise direction. Figure 5.13 shows that the change in the tortuosity is negligible when the complex unit cells are included. Tortuosity represents the ratio between the magnitude of the flow in the span-wise and stream-wise direction and is defined as:

$$\tau = \frac{L}{L_o} \tag{5.8}$$

Here L is the length of the actual flow path and L_o is the shortest possible length through the porous media. We utilize the fast-marching algorithm [28] to obtain the time of flight through porous media with a uniform velocity field of unity, which is then used to calculate the tortuosity. The predicted tortuosity values for Model A are comparable to those for Model B, both of which are appreciably smaller than those of the reticulated foams.

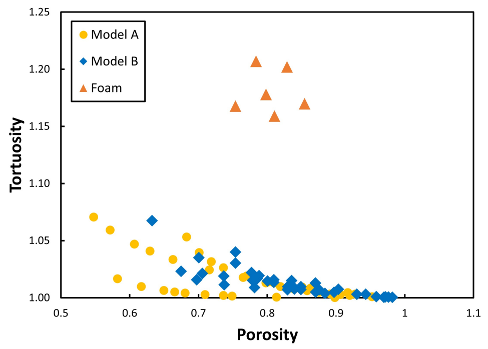


Figure 5.13: Computed tortuosity with (model B) and without (model A) the complex unit cells. The tortuosity of the reticulated foams is also shown for comparison.

5.5.3 Voxel-wise Accuracy of Predicted Velocity Fields

The accuracy of the predicted 3D velocity fields from our DL models can be affected by large voxel-wise errors, partly due to misalignment in the velocity fields [102].To evaluate the voxel-wise accuracy of our models, we compared the predicted and ground-truth 3D velocity fields for all reticulated foams (Figure 5.14). Qualitatively, the predicted 3D velocity fields exhibited good overall agreement for all foams, accurately capturing the dominant flow paths. However, noticeable voxel-wise errors are present in the predicted velocity fields. To illustrate the details, we present Figures 5.15 and 5.16, which display the 2D planar velocity fields predicted from model D for the span-wise and stream-wise directions, respectively, of reticulated foam 5. This foam had the largest difference in permeability and STAFE. In

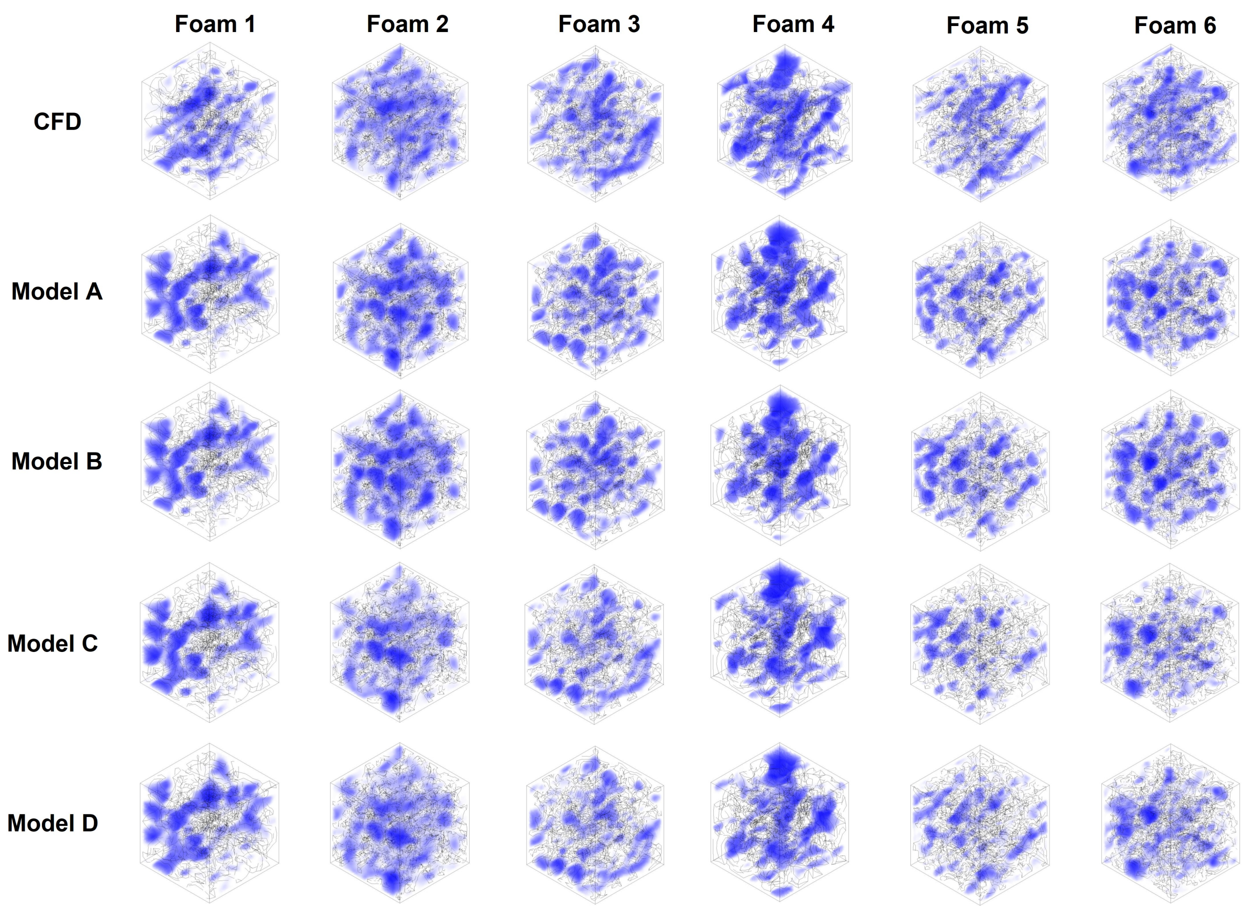


Figure 5.14: Visualization of the DL model-predicted and ground-truth 3D velocity fields of all the reticulated foams. The renderings are in blue monochromatic scale where lower opacity represents higher velocity magnitude.

Figure 5.15, we observe that almost no voxels exhibit a difference in the velocity direction. However, a few pores show a noticeable difference in the velocity magnitude. Similarly, a few pores in Figure 5.16 show difference in the velocity magnitude. We also note that the absolute errors for stream-wise direction are considerably smaller than those for span-wise direction, consistent with Figures 5.9 and 5.10.

We also quantify the voxel-wise errors by calculating the mean absolute percentage error for all voxels with velocity magnitudes above the average for each reticulated foam. We impose the threshold to prevent those voxels with very small magnitudes of velocity from dominating the calculated percent errors. Table 5.2 shows the voxel-wise mean absolute percentage error averaged across all reticulated foams. We observe a similar relationship

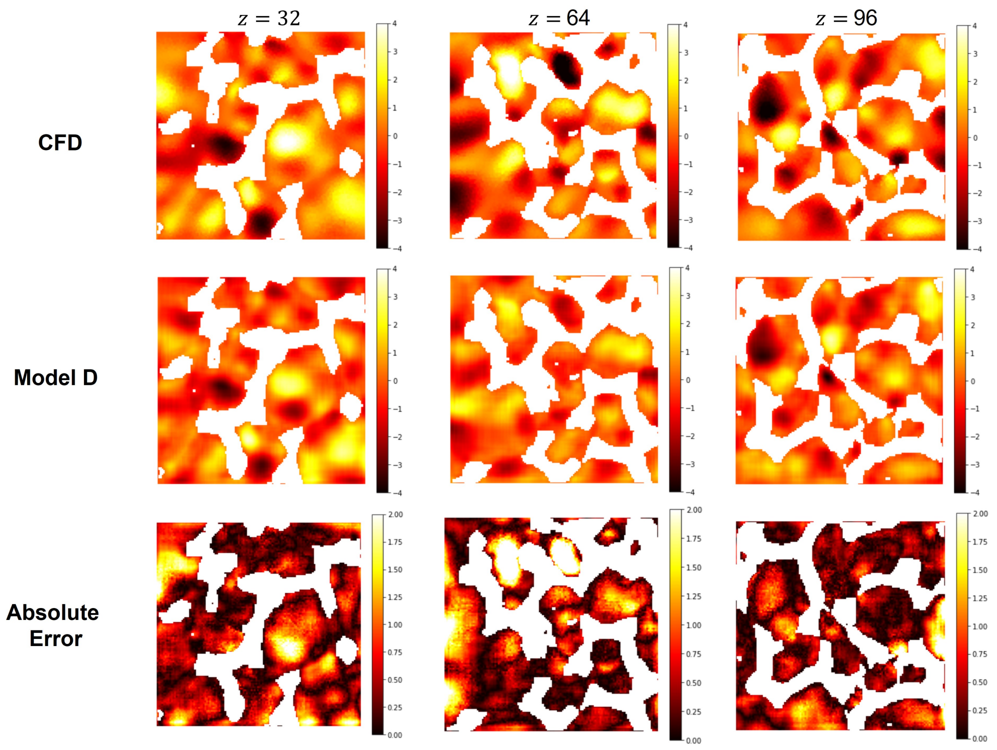


Figure 5.15: Span-wise velocity profiles of reticulated foam 5 at different cross-sections orthogonal to the stream-wise direction. Absolute errors are also shown for comparison.

Table 5.2: The mean absolute percentage error for all the deep learning models. The errors are averaged across all reticulated foams.

Model Name	**Stream-wise** [%]	**Span-wise** [%]
Model A	33	83
Model B	32	81
Model C	30	46
Model D	25	45

among the models to what we presented earlier in Figures 5.9 and 5.10. The additions of the complex unit cells and larger physical pore length scales reduce the voxel-wise errors in the stream-wise direction. In contrast, only the addition of the larger physical pore length scales results in significant improvement in the span-wise direction. Model D shows the best accuracy with the mean absolute percentage error of 25% in the stream-wise direction and 45% in the span-wise direction. Relatively large voxel-wise errors due to the misalignment in the velocity fields were also observed in other CNN-based deep learning studies [83, 102], which

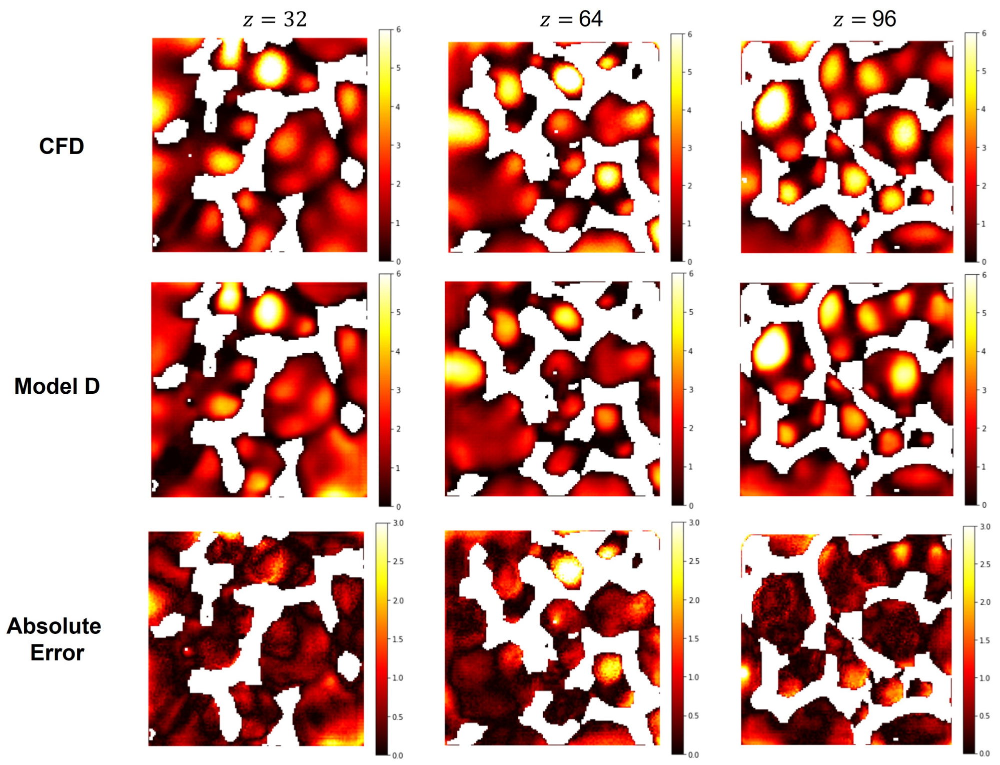


Figure 5.16: Stream-wise velocity profiles of reticulated foam 5 at different cross-sections orthogonal to the stream-wise direction. Absolute errors are also shown for comparison.

suggests a limitation of the convolutional neural network architecture for this application.

5.5.4 DL Model Assisted CFD Simulations

We have shown that our DL models, trained using unit cells, can capture overall flow behavior in real heterogeneous porous media. Although the voxel-wise accuracy of 3D velocity fields predicted from our DL models may not be sufficient in certain applications, they can still be useful serving as an initial guess for rigorous numerical simulation to accelerate its numerical convergence. Figure 5.17 shows the number of iterations required to reach a convergence for the reticulated foams with and without the assistance of our DL model. K_o represents the ground-truth permeability. We notice a considerable decrease in the required number of iterations, by a factor of 3 on average and by a factor of 7 in some cases, when utilizing our deep learning model. This improvement is comparable to a model trained using

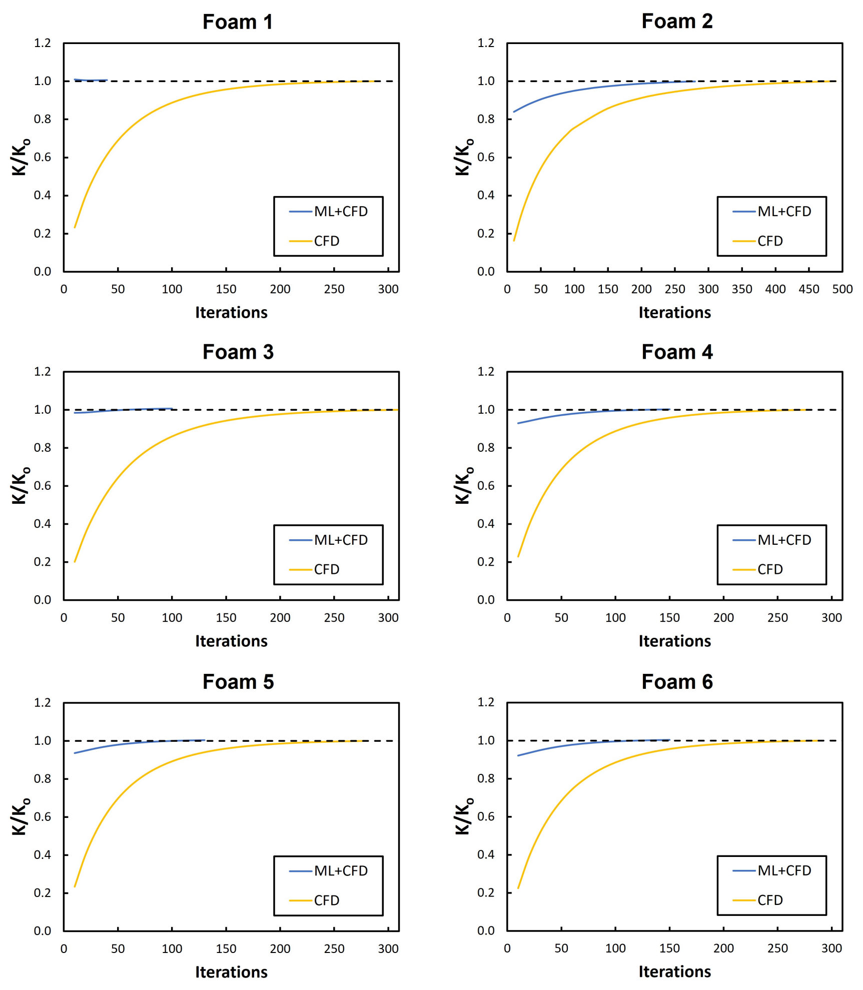


Figure 5.17: Convergence of the numerical simulation runs with and without the assistance from the deep learning model prediction. K_o represents the ground-truth permeability obtained from CFD.

synthetic porous media [102]. We emphasize that our deep learning model is trained using unit cells whose numerical simulations require very little computational time and resources. The past study also initialized their numerical simulation using the pressure field, which was predicted by a separately trained deep learning model, as well as the velocity fields. Here, the pressure fields are initialized as zero.

5.6 Summary and Conclusion

In this chapter, we assessed the accuracy of CNN models trained solely on periodic unit cells for predicting the velocity fields of complex heterogeneous porous media. We considered various unit cell arrangements, including simple cubic, body-centered cubic, face-centered cubic, and end-centered cubic structures, as well as more complex unit cells derived from crystal lattice structures and varying physical length scales. Our models accurately predicted the permeability and pore-scale flow behavior of synthetic porous media and real reticulated foams, with improved predictions observed in the stream-wise direction when using complex unit cells and in the span-wise direction when varying the physical length scale. We also showed that the predictions from our model can be used as initial guesses to significantly improve the convergence of numerical simulations.

By eliminating the need for computationally intensive simulations of heterogeneous porous media, our approach enhances computational efficiency and can facilitate the use of rigorous numerical simulations for exploring a large batch of complex porous media. Our results have potential applications in modeling the pore-scale hydrodynamic and transport behavior of porous media for various engineering applications.

Chapter 6

Conclusion

6.1 Thesis Conclusion

The primary objective of this thesis was to explore the viability of transpiration cooling employing oxide coolants as an alternative system to thermally protect sharp leading edges. This transpiration cooling injects solid oxide into the leading edge that melt and flow to the surface and absorb incident heat fluxes. Two primary objectives were sought in this thesis that collectively investigate the transpiration cooling. The first objective focused on the evaporation at the surface and the interaction with the external flow. The second objective sought to characterize the internal liquid flow through porous media.

We first parametrically characterized the performance of transpiration cooling for various coolant properties, flight conditions, and leading edge radii using 2D axi-symmetric, semi-analytical boundary layer model incorporating thermodynamic non-equilibrium conditions, which we validated with 3rd-order shock-fitting DNS. We quantified the performance of the TPS using three metrics: the surface temperature, the evaporative mass flux, and the boiling limit. We showed that the surface temperature depends solely on the saturation temperature of the coolant material, the boiling limit is independent of molar mass, and the evaporative mass flux is affected by all three coolant properties. We also illustrated that high Mach numbers and small leading edge sizes exacerbate the aerothermal condition resulting in higher surface temperature and evaporative mass flux and lower boiling limit factor of safety. Low altitude flights also increase the surface temperature and evaporative mass flux, however, result in higher boiling limit factor of safety. Our parametric results do not lend themselves readily to an optimal set of coolant properties. Rather, different coolant properties are better suited for different flight conditions and leading edge sizes. For instance, vehicles with high Mach number and low altitude trajectories or small radius of curvature of the leading edge may utilize coolant materials with high specific latent heats to significantly reduce the evaporative mass flux. Alternatively, vehicles with low altitude trajectories may benefit from coolants with low specific latent heats to avoid vapor nucleation within the porous leading to which they are more susceptible at higher altitudes. Our findings in this chapter demonstrated that transpiration cooling employing oxide coolants may effectively cool the surface temperature to below the saturation temperature of the coolant material even for very sharp leading edges with radius of 0.1 mm and in extreme hypersonic conditions, where incident heat fluxes are of the order of 85 MJ/m2, as long as the necessary amount of coolant is supplied to the surface.

We then characterize the liquid flow through high porosity reticulated foams, which are potential leading edge material for transpiration cooling, to assess their capability in achieving the necessary flow rate required by evaporation at the surface. We numerically and experimentally obtain the permeability of silicon carbide reticulated foams with three different pore densities: 45 PPI, 65 PPI, and 80 PPI. The numerical approach utilized reconstructed X-ray micro-tomography images and computational fluid dynamics to solve the governing equations for laminar Stokes flow through the porous media. The experimental approach measured the pressure gradient across the porous media subject to air flow. The two approaches agreed well and resulted in permeability of the order of $5 \times 10^{-9} m^2$. We found that the porous media with the lowest pore density showed the minimum pressure gradient to achieve a given mass flow rate. A maximum of 100 Pa/mm is necessary to achieve the highest flow rate predicted by the semi-analytical model in Chapter 2. We also assessed the capillary pumping capability of the porous media, which is enhanced by high surface energies of the molten oxides and low permeability of the reticulated foams. Our results demonstrated that the capillary pumping is most effective when using the porous media with the lowest pore density porous. For the lowest pore density that we characterized (45 PPI), we estimate that capillary pumping alone can drive sufficient flow to the surface for moderate altitudes and Mach numbers, but it would fail at low altitudes and high Mach numbers.

We further designed deep learning models to characterize the flow through porous media at the pore-scale from only their binary images, eliminating the need for computationally intensive CFD simulations of heterogeneous porous media. We developed two models: the first model consisted of more complex training data and architecture as described in Chapter 4, and the second model comprised of simpler training data and architecture as described in Chapter 5. The complex model utilized synthetic reticulated foams to train and a model that incorporated the law of mass conservation. This model showed excellent accuracy in predicting the permeability, with less than 6% difference with the CFD results, as well as in predicting the pore-scale velocity field, with STAFE less than 0.2. The simpler model employed periodic, homogeneous unit cells to train a model without incorporating any conservation laws. This model accurately predicted the permeability, with 10% difference with the CFD results, as well as the pore-scale velocity field, with STAFE less than 0.4. By eliminating the need for computationally intensive simulations of heterogeneous porous media, deep learning models facilitate rigorous characterization of the porous media which are crucial in designing and optimizing the porous leading edge that can support transpiration cooling.

6.2 Thesis Objectives

In summary, this thesis achieved the following objectives:

1. Demonstrated the viability of transpiration cooling employing oxides as an alternative TPS to effectively protect sharp leading edges in hypersonic flight.

2. Characterized the surface temperature, evaporative mass flux, and boiling limit of transpiration cooling for different coolant properties, flight conditions, and leading edge radii to facilitate the optimization of the system.

3. Experimentally and numerically characterized the liquid coolant flow through high porosity reticulated foams to predict the necessary pressure gradient that ensures continuous coolant flow to the surface and to assess self-pumping capability using capillary forces.

4. Developed computationally efficient deep learning models to characterize the flow through high porosity reticulated foams at the pore-scale and to facilitate the design of the microstructures of porous leading edge to optimize the performance of transpiration cooling.

By achieving these objectives, this thesis demonstrated that transpiration cooling employing oxide coolants may effectively cool the surface temperature even for very sharp leading edges in extreme hypersonic conditions as long as the necessary amount of coolant is supplied to the surface. It also provides two numerical frameworks that coherently characterize both the external and internal aspect of transpiration cooling which facilitate the design and optimization of transpiration cooling utilizing oxide coolant for various hypersonic applications.

6.3 Future Works

One advantage of transpiration cooling employing oxide coolants is its ability to sufficiently self-pump the liquid coolant to the surface. However, even with high surface energy of oxide and high permeability of reticulated foams, capillary forces alone cannot drive the necessary flow under certain hypersonic conditions as shown in Section 3.4. We note, however, that capillary forces rising from interfacial forces are surface phenomena, whereas the opposing viscous pressure loss is a volumetric phenomenon that occurs throughout the porous leading edge. One may take advantage of these characteristics to construct a hybrid microstructure where the high pore density foams form the skin of the leading edge to increase the capillary forces and low pore density foams form the core of the leading edge to minimize the viscous pressure loss. Figure 6.1 illustrates a potential design of the hybrid foam for porous leading edges. Following a similar analysis presented in Section 3.4, the ratio of

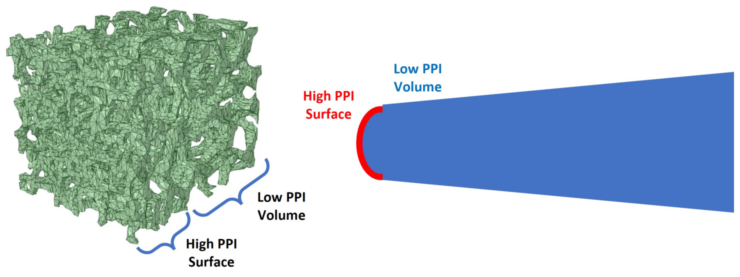


Figure 6.1: A potential design of porous leading edge using a hybrid reticulated foam consisting of both high and low pore density foams.

capillary pressure and viscous pressure loss is calculated and illustrated in Figure 6.2. The surface tension and the contact angle are assumed to be 0.1 N/m and 45 degrees, respectively, which are typical for most oxides. By increasing the interfacial capillary pressure using high pore density foam at the surface and reducing the permeability throughout the volume, hybrid foams amplify the ratio of the two pressures and can achieve self-pumping without any necessary external forces for, even for relatively large mass flow rate. However, the microstructure of the hybrid foam can significantly impact the flow behavior at

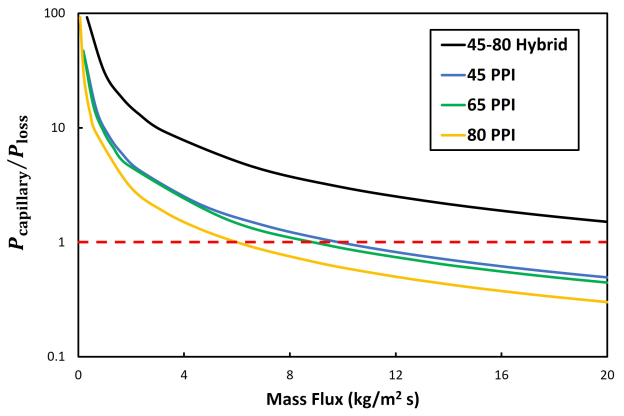


Figure 6.2: Ratio of capillary pressure and viscous pressure loss as a function of mass flux through a porous leading edge.

the pore-scale, especially near the surface since the length scale of the flow is comparable to that of the high pore density porous media. Hence, it is crucial to characterize the capillary interaction of the fluid with the microstructure at the pore-scale to accurately assess the self-pumping ability.

Appendix A

The data that support the findings of the study in Chapter 4 is openly available in Dryad:
https://datadryad.org/stash/dataset/doi:10.5068/D19Q37

The data that supports the findings of the study in Chapter 5 is openly available in Dryad:
https://datadryad.org/stash/dataset/doi:10.5068/D16108

The source code that supports the work in Chapter 4 and Chapter 5 are openly available in GitHub repository:
https://github.com/dko1217/DeepLearning-PorousMedia

Appendix B

The source code used to develop and train the deep learning model in Chapter 4 is provided as plain text here.

Initialization

System Information

```
    !cat /proc/cpuinfo


    gpu_info = !nvidia-smi
    gpu_info = '\n'.join(gpu_info)
    print(gpu_info)
```

Load Libraries

```
    !pip install hdf5storage
    from hdf5storage import loadmat


    import numpy as np
    from numpy.random import seed
    from numpy.random import default_rng


    from matplotlib import pyplot as plt
    import os
```

```
import scipy
from scipy import io

import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.compat.v1.keras.backend import set_session

from tensorflow.keras.models import Model
from tensorflow.keras.models import load_model
from tensorflow.keras.models import save_model

from sklearn.model_selection import train_test_split

tf.random.set_seed(280675)
np.random.seed(280675)

from google.colab import drive
drive.mount('/content/drive')

%cd /content/drive/My\ Drive/DarcyUnet Unit Cell/

print('tensorflow version : {}'.format(tf.__version__))
```

Functions

```
def minmax_transform(x, x_min=0, x_range=1):
  x = ( x - x_min ) / x_range
  return x



def shift_augmentation(my_solid, my_vel, shift_range, vel_dir):

  my_aug_solid = np.ones_like(my_solid)
  my_aug_vel = np.ones_like(my_vel)

  shift_x = shift_range[0]
  shift_y = shift_range[1]

  if(shift_x < 0):
    my_aug_solid[:shift_x,:,:] = my_solid[-1*shift_x:,:,:]
    my_aug_solid[shift_x:,:,:] = np.flip(my_solid, axis=0 )[:-1*shift_x,:,:]

    my_aug_vel[:shift_x,:,:] = my_vel[-1*shift_x:,:,:]
    if(vel_dir[0] == 'x'):
      my_aug_vel[shift_x:,:,:] = np.flip(-1*my_vel, axis=0 )[:-1*shift_x,:,:]
    elif(vel_dir[0] == 'y'):
      my_aug_vel[shift_x:,:,:] = np.flip(my_vel, axis=0 )[:-1*shift_x,:,:]
    else:
      my_aug_vel[shift_x:,:,:] = np.flip(my_vel, axis=0 )[:-1*shift_x,:,:]

  elif(shift_x > 0):
    my_aug_solid[:shift_x,:,:,:] = np.flip(my_solid, axis=0 )[-1*shift_x:,:,:,:]
    my_aug_solid[shift_x:,:,:,:] = my_solid[:-1*shift_x,:,:,:]

    if(vel_dir[0] == 'x'):
      my_aug_vel[:shift_x,:,:,:] = np.flip(-1*my_vel, axis=0 )[-1*shift_x:,:,:,:]
    elif(vel_dir[0] == 'y'):
      my_aug_vel[:shift_x,:,:,:] = np.flip(my_vel, axis=0 )[-1*shift_x:,:,:,:]
    else:
      my_aug_vel[:shift_x,:,:,:] = np.flip(my_vel, axis=0 )[-1*shift_x:,:,:,:]
    my_aug_vel[shift_x:,:,:,:] = my_vel[:-1*shift_x,:,:,:]
  else:
    my_aug_solid = my_solid
    my_aug_vel = my_vel

  my_solid = my_aug_solid
  my_vel = my_aug_vel

  my_aug_solid = np.ones_like(my_solid)
  my_aug_vel = np.ones_like(my_vel)

  if(shift_y < 0):
    my_aug_solid[:,:shift_y,:] = my_solid[:,-1*shift_y:,:]
    my_aug_solid[:,shift_y:,:] = np.flip(my_solid, axis=1 )[:,:-1*shift_y,:]

    my_aug_vel[:,:shift_y,:] = my_vel[:,-1*shift_y:,:]
    if(vel_dir[0] == 'x'):
      my_aug_vel[:,shift_y:,:] = np.flip(my_vel, axis=1 )[:,:-1*shift_y,:]
    elif(vel_dir[0] == 'y'):
      my_aug_vel[:,shift_y:,:] = np.flip(-1*my_vel, axis=1 )[:,:-1*shift_y,:]
    else:
      my_aug_vel[:,shift_y:,:] = np.flip(my_vel, axis=1 )[:,:-1*shift_y,:]

  elif(shift_y > 0):
    my_aug_solid[:,:shift_y,:] = np.flip(my_solid, axis=1 )[:,-1*shift_y:,:]
    my_aug_solid[:,shift_y:,:] = my_solid[:,:-1*shift_y,:]


    if(vel_dir[0] == 'x'):
      my_aug_vel[:,:shift_y,:] = np.flip(my_vel, axis=1 )[:,-1*shift_y:,:]
    elif(vel_dir[0] == 'y'):
      my_aug_vel[:,:shift_y,:] = np.flip(-1*my_vel, axis=1 )[:,-1*shift_y:,:]
    else:
      my_aug_vel[:,:shift_y,:] = np.flip(my_vel, axis=1 )[:,-1*shift_y:,:]
    my_aug_vel[:,shift_y:,:] = my_vel[:,:-1*shift_y,:]
  else:
    my_aug_solid = my_solid
    my_aug_vel = my_vel

  return my_aug_solid, my_aug_vel



def flip_augmentation(my_solid, my_vel, vel_dir, axis):
    my_aug_solid = np.flip(my_solid, axis=axis)

    if(vel_dir[0] == 'x'):
      if(axis == 0):
        my_aug_vel = np.flip(-1*my_vel, axis=axis)
      else:
        my_aug_vel = np.flip(my_vel, axis=axis)
    elif(vel_dir[0] == 'y'):
      if(axis == 1):
        my_aug_vel = np.flip(-1*my_vel, axis=axis)
      else:
        my_aug_vel = np.flip(my_vel, axis=axis)
    else:
      my_aug_vel = np.flip(my_vel, axis=axis)

    return my_aug_solid, my_aug_vel
```

Custom Loss

```
  def div_loss1(y_true, y_pred):

    scale = 1

    mse = tf.math.reduce_mean( tf.math.square(y_true - y_pred) )

    dVxdx_true = (y_true[:,2:,1:-1,1:-1,0] - y_true[:,:-2,1:-1,1:-1,0])/2
  dVydy_true = (y_true[:,1:-1,2:,1:-1,1] - y_true[:,1:-1,:-2,1:-1,1])/2
  dVzdz_true = (y_true[:,1:-1,1:-1,2:,2] - y_true[:,1:-1,1:-1,:-2,2])/2
  div_true = dVxdx_true + dVydy_true + dVzdz_true

  dVxdx_pred = (y_pred[:,2:,1:-1,1:-1,0] - y_pred[:,:-2,1:-1,1:-1,0])/2
  dVydy_pred = (y_pred[:,1:-1,2:,1:-1,1] - y_pred[:,1:-1,:-2,1:-1,1])/2
  dVzdz_pred = (y_pred[:,1:-1,1:-1,2:,2] - y_pred[:,1:-1,1:-1,:-2,2])/2
  div_pred = dVxdx_pred + dVydy_pred + dVzdz_pred

  div_loss = tf.math.reduce_mean( tf.math.abs(div_true - div_pred) )

  loss = mse + div_loss*scale

  return loss


def div_loss2(y_true, y_pred):

  scale = 3

  mse = tf.math.reduce_mean( tf.math.square(y_true - y_pred) )

  dVxdx_pred = (y_pred[:,2:,1:-1,1:-1,0] - y_pred[:,:-2,1:-1,1:-1,0])/2
  dVydy_pred = (y_pred[:,1:-1,2:,1:-1,1] - y_pred[:,1:-1,:-2,1:-1,1])/2
  dVzdz_pred = (y_pred[:,1:-1,1:-1,2:,2] - y_pred[:,1:-1,1:-1,:-2,2])/2
  div_pred = dVxdx_pred + dVydy_pred + dVzdz_pred
```

```
    div_loss = tf.math.reduce_mean( tf.math.abs(div_pred) )

    loss = mse + div_loss*scale

    return loss
```

PolySphere Data

```
  dir_data = 'Data/PolySphere'

  data_size = 120

  """
  X : ('x')
  Y : ('y')
  Z : ('z')
  """
  #velocity_dir = ('x')
  #velocity_dir = ('y')
  #velocity_dir = ('z')
  velocity_dir = ('x', 'y', 'z')

  domainRange = [1,2,3,4,5,6,7,8,9]

  channel_size = len(velocity_dir)
  uc_solid = np.zeros( (1,data_size, data_size, data_size) )
  uc_vel = np.zeros( (1,data_size, data_size, data_size, len(velocity_dir)) )

  for i in range(len(domainRange)):

    uc_solid_load = loadmat( '{}/PolySphere_domain{}_deci.mat'.format(dir_data,
domainRange[i]) )['solid'].astype('int')
    uc_solid_load = uc_solid_load < 1
    uc_solid = np.append( uc_solid, np.expand_dims(uc_solid_load, axis=0), axis=0 )
    del uc_solid_load

    uc_vel_load = np.zeros( (1, data_size, data_size, data_size, channel_size) )
    for j in range(channel_size):
     if(velocity_dir[j] == 'z'):
      uc_vel_load[0,:,:,:,j] = loadmat( '{}/PolySphere_domain{}_vfield{}.mat'.format(dir_data,
domainRange[i], '') )['vfield'].astype('float32')
     else:
      uc_vel_load[0,:,:,:,j] = loadmat( '{}/PolySphere_domain{}_vfield{}.mat'.format(dir_data,
domainRange[i], velocity_dir[j]) )['vfield'].astype('float32')
    uc_vel = np.append( uc_vel, uc_vel_load, axis=0 )
    del uc_vel_load


  uc_solid = uc_solid[1:,:,:,:]
  uc_vel = uc_vel[1:,:,:,:,:]
```

```
print('Image size : {}\nNumber of Data : {}  {}'.format(data_size, uc_solid.shape, uc_vel.shape))

from scipy.ndimage import distance_transform_edt
from scipy.ndimage import distance_transform
uc_solid_edt = distance_transform_edt(uc_solid)

""" Normalization
"""
res = 20e-6

x_min = 0
x_max = 8

uc_vel_norm = uc_vel/(res**2)*0.333/9270

print( '\nMean velocity : {}'.format( uc_vel_norm.mean() ) )
print( '\nMin velocity : {}'.format( uc_vel_norm.min() ) )
print( '\nMax velocity : {}'.format( uc_vel_norm.max() ) )

print('\nX_min : {} and X_max : {}'.format(x_min, x_max))

uc_vel_minmax = minmax_transform(uc_vel_norm, x_min=x_min, x_range=x_max-x_min)

print( '\nMean velocity : {}'.format( uc_vel_minmax.mean() ) )
print( '\nMin velocity : {}'.format( uc_vel_minmax.min() ) )
```

```
  print( '\nMax velocity : {}'.format( uc_vel_minmax.max() ) )

  augGen_seed = 10

  aug_iter = 15

  shift_range = 2
  flip_range = 5

  aug_uc_solid = np.zeros( (1, data_size, data_size, data_size) )
  aug_uc_vel = np.zeros( (1, data_size, data_size, data_size, channel_size) )

  """ Shift and Flip augmentation
  """

  shift = data_size//shift_range
  rnd_num_gen = default_rng(augGen_seed)
  rnd_num = rnd_num_gen.integers(low=-shift, high=shift, size=(uc_solid.shape[0],aug_iter,2),
endpoint=True)
  rnd_num2 = rnd_num_gen.integers(low=0, high=10, size=(uc_solid.shape[0],aug_iter,2))

  for i in range(uc_solid.shape[0]):

    my_solid = uc_solid[i,:,:,:]
    my_vel = uc_vel_minmax[i,:,:,:,:]
    for j in range(aug_iter):
     if(channel_size <= 1):

       my_aug_solid, my_aug_vel = shift_augmentation(my_solid, my_vel[:,:,:,0], rnd_num[i,j,:],
velocity_dir)
       if(rnd_num2[i,j,0] >= flip_range):
         my_aug_solid, my_aug_vel = flip_augmentation(my_aug_solid, my_aug_vel, velocity_dir,
0)
       if(rnd_num2[i,j,1] >= flip_range):
         my_aug_solid, my_aug_vel = flip_augmentation(my_aug_solid, my_aug_vel, velocity_dir,
1)

       aug_uc_solid = np.append(aug_uc_solid, np.expand_dims(my_aug_solid,axis=0), axis=0)
       aug_uc_vel                    =                    np.append(aug_uc_vel,
np.expand_dims( np.expand_dims(my_aug_vel,axis=0), axis=-1 ), axis=0)

     elif(channel_size == 3):

       my_aug_solid,   my_aug_vel_x   =   shift_augmentation(my_solid,   my_vel[:,:,:,0],
rnd_num[i,j,:], ('x'))
       my_aug_solid,   my_aug_vel_y   =   shift_augmentation(my_solid,   my_vel[:,:,:,1],
rnd_num[i,j,:], ('y'))
       my_aug_solid,   my_aug_vel_z   =   shift_augmentation(my_solid,   my_vel[:,:,:,2],
rnd_num[i,j,:], ('z'))

       if(rnd_num2[i,j,0] >= flip_range):
         my_aug_solid, my_aug_vel_x = flip_augmentation(my_aug_solid, my_aug_vel_x, ('x'), 0)
         my_aug_solid, my_aug_vel_y = flip_augmentation(my_aug_solid, my_aug_vel_y, ('y'), 0)
         my_aug_solid, my_aug_vel_z = flip_augmentation(my_aug_solid, my_aug_vel_z, ('z'), 0)
       if(rnd_num2[i,j,1] >= flip_range):
         my_aug_solid, my_aug_vel_x = flip_augmentation(my_aug_solid, my_aug_vel_x, ('x'), 1)
         my_aug_solid, my_aug_vel_y = flip_augmentation(my_aug_solid, my_aug_vel_y, ('y'), 1)
         my_aug_solid, my_aug_vel_z = flip_augmentation(my_aug_solid, my_aug_vel_z, ('z'), 1)

       my_aug_vel = np.append( np.append(np.expand_dims(my_aug_vel_x,axis=-1),
np.expand_dims(my_aug_vel_y,axis=-1), axis=-1), np.expand_dims(my_aug_vel_z,axis=-1),
axis=-1 )

       aug_uc_solid = np.append(aug_uc_solid, np.expand_dims(my_aug_solid,axis=0), axis=0)
       aug_uc_vel = np.append(aug_uc_vel, np.expand_dims(my_aug_vel,axis=0), axis=0)


   aug_uc_solid = aug_uc_solid[1:,:,:,:]
   aug_uc_vel = aug_uc_vel[1:,:,:,:,:]

   total_number = aug_uc_solid.shape[0]

   val_perc = 0.1
   test_perc = 0.1

   train_index, val_test_index = train_test_split(np.arange(total_number), test_size = val_perc +
test_perc, random_state = augGen_seed)
```

```
    val_index,        test_index        =        train_test_split(val_test_index,        test_size        =
test_perc/(val_perc+test_perc), random_state = augGen_seed)

    print( 'Number of training samples : {}\nNumber of validation samples : {}\nNumber of test
samples : {}\n'.format(len(train_index),len(val_index),len(test_index)) )

    train_data_solid = np.expand_dims(aug_uc_solid[train_index], axis=-1)
    val_data_solid = np.expand_dims(aug_uc_solid[val_index], axis=-1)
    test_data_solid = np.expand_dims(aug_uc_solid[test_index], axis=-1)

    train_data_vel = aug_uc_vel[train_index]
    val_data_vel = aug_uc_vel[val_index]
    test_data_vel = aug_uc_vel[test_index]

    print(train_data_solid.shape)
    print(val_data_solid.shape)
    print(test_data_solid.shape)

    print(train_data_vel.shape)
    print(val_data_vel.shape)
    print(test_data_vel.shape)
```

Data Check

Mass loss check

```
uc_solid_temp = uc_solid[0,:,:,:]

uc_solid_temp = np.expand_dims(uc_solid_temp, axis=(-1))


uc_vel_temp = uc_vel_minmax[0,:,:,:,:]


dVxdx = (uc_vel_temp[2:,1:-1,1:-1,0] - uc_vel_temp[:-2,1:-1,1:-1,0])/2

dVydy = (uc_vel_temp[1:-1,2:,1:-1,1] - uc_vel_temp[1:-1,:-2,1:-1,1])/2

dVzdz = (uc_vel_temp[1:-1,1:-1,2:,2] - uc_vel_temp[1:-1,1:-1,:-2,2])/2

div = dVxdx + dVydy + dVzdz


plt.imshow(uc_solid_temp[:,:,59,0])

plt.show()


plt.imshow(uc_vel_temp[:,:,59,2])

plt.colorbar()

plt.show()


plt.imshow(div[:,:,59], vmin=-0.3, vmax=0.3)

plt.colorbar()

plt.show()


div_flat = div.flatten()

plt.hist(div_flat, np.linspace(-0.3,0.3,100))

plt.show()


print(np.mean( np.abs(div_flat) ) )
```

```
xversion = 'X1-3'
x_model_name = 'UnetRS_Modelv{}'.format(xversion)
x_model = tf.keras.models.load_model( 'RandomSphere
Model/{}/{}.ckpt'.format(x_model_name, x_model_name ) )

yversion = 'Y1-1'
y_model_name = 'UnetRS_Modelv{}'.format(yversion)
y_model = tf.keras.models.load_model( 'RandomSphere
Model/{}/{}.ckpt'.format(y_model_name, y_model_name ) )

zversion = 'Z1-4'
z_model_name = 'UnetRS_Modelv{}'.format(zversion)
z_model = tf.keras.models.load_model( 'RandomSphere
Model/{}/{}.ckpt'.format(z_model_name, z_model_name ) )

vx_test_pred = np.float32( x_model.predict( x=[np.expand_dims(uc_solid_temp,axis=0)] ) )/2
vy_test_pred = np.float32( y_model.predict( x=[np.expand_dims(uc_solid_temp,axis=0)] ) )/2
vz_test_pred = np.float32( z_model.predict( x=[np.expand_dims(uc_solid_temp,axis=0)] ) )

vpred_test_temp = np.append(vx_test_pred, np.append(vy_test_pred, vz_test_pred,axis=-
1),axis=-1 )
print(vpred_test_temp.shape)

temp_vel = np.multiply(vpred_test_temp[0,:,:,:,:],uc_solid_temp[:,:,:,:])
#temp_vel = vpred_test_temp[0,:,:,:,:]
```

```
    dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
    dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
    dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
    div = dVxdx + dVydy + dVzdz

    print(uc_solid_temp.shape)
    plt.imshow(uc_solid_temp[:,:,59,0])
    plt.show()

    plt.imshow(temp_vel[:,:,59,2])
    plt.colorbar()
    plt.show()

    plt.imshow(div[:,:,59],vmin=-0.3,vmax=0.3)
    plt.colorbar()
    plt.show()

    div_flat = div.flatten()
    plt.hist(div_flat, np.linspace(-0.3,0.3,100))
    plt.show()

    print(np.mean( np.abs(div_flat) ) )

    temp_div_loss                    =                    div_loss1(np.expand_dims(uc_vel_temp,axis=0),
np.expand_dims(temp_vel,axis=0))
```

```
print(temp_div_loss)
```

Augmentation test

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator


temp = np.zeros( (1,20,20,1) )
temp[0,15:,:,0] = 10


plt.imshow(temp[0,:,:,0])
plt.show()


augGenTemp = ImageDataGenerator(#rotation_range=45,
                                #width_shift_range=(data_size//4,
                                height_shift_range=(5,5),
                                fill_mode='reflect')


iterator_temp = augGenTemp.flow(temp)




temp2 = iterator_temp.next()
print(temp2.shape)


plt.imshow(temp2[0,:,:,0])
plt.show()
```

```
temp = np.zeros( (20,20,20) )
temp[13:,13:,:] = 10

plt.imshow(temp[:,:,9])
plt.show()

rnd_num_gen = default_rng(10)

# 1: height (x)
# 2: width (y)
#rnd_num = rnd_num_gen.integers(low=-shift_range, high=shift_range, size=2, endpoint=True)
rnd_num = (-10,-10)

my_solid = temp
my_aug_solid = np.ones_like(temp)

if(rnd_num[0] < 0):
  my_aug_solid[:rnd_num[0],:,:] = my_solid[-1*rnd_num[0]:,:,:]
  my_aug_solid[rnd_num[0]:,:,:] = np.flip(my_solid, axis=0 )[:-1*rnd_num[0],:,:]
else:
  my_aug_solid[:rnd_num[0],:,:] = np.flip(my_solid, axis=0 )[-1*rnd_num[0]:,:,:]
  my_aug_solid[rnd_num[0]:,:,:] = my_solid[:-1*rnd_num[0],:,:]

my_solid = my_aug_solid
my_aug_solid = np.ones_like(temp)
```

```
if(rnd_num[1] < 0):
  my_aug_solid[:,:rnd_num[1],:] = my_solid[:,-1*rnd_num[1]:,:]
  my_aug_solid[:,rnd_num[1]:,:] = np.flip(my_solid, axis=1 )[:,:-1*rnd_num[1],:]
else:
  my_aug_solid[:,:rnd_num[1],:] = np.flip(my_solid, axis=1 )[:,-1*rnd_num[1]:,:]
  my_aug_solid[:,rnd_num[1]:,:] = my_solid[:,:-1*rnd_num[1],:]

plt.imshow(my_aug_solid[:,:,9])
plt.show()

plt.imshow(my_solid[:,:,9])
plt.show()

rnd_num = (10,10)

if(rnd_num[1] < 0):
  my_aug_solid[:,:rnd_num[1],:] = my_solid[:,-1*rnd_num[1]:,:]
  my_aug_solid[:,rnd_num[1]:,:] = np.flip(my_solid, axis=1 )[:,:-1*rnd_num[1],:]
else:
  my_aug_solid[:,:rnd_num[1],:] = np.flip(my_solid, axis=1 )[:,-1*rnd_num[1]:,:]
  my_aug_solid[:,rnd_num[1]:,:] = my_solid[:,:-1*rnd_num[1],:]

plt.imshow(my_aug_solid[:,:,9])
plt.show()
```

```
    temp = np.array([1,2,3,4,5,6,67,7,8,9])

    print(temp[:-3])

    #temp = np.zeros( (20,20,20) )
    #temp[13:,13:,:] = 10

    temp = uc_solid[0,:,:,:]
    temp_v = uc_vel[0,:,:,:,0]

    plt.imshow(temp[:,:,9])
    plt.show()

    plt.imshow(temp_v[:,:,9])
    plt.show()

    my_aug_solid = np.flip(temp, axis=0)
    my_aug_vel = np.flip(-1*temp_v, axis=0)

    plt.imshow(my_aug_solid[:,:,9])
    plt.show()

    plt.imshow(my_aug_vel[:,:,9])
    plt.show()
```

Save training data

```
    data_num = 1

    np.save('Hoffman Cluster/train data/train_solid_randomsphere_data{}'.format(data_num), train_data_solid)
    np.save('Hoffman Cluster/train data/train_vel{}_randomsphere_data{}'.format(velocity_dir, data_num), train_data_vel)

    np.save('Hoffman Cluster/train data/val_solid_randomsphere_data{}'.format(data_num), val_data_solid)
    np.save('Hoffman Cluster/train data/val_vel{}_randomsphere_data{}'.format(velocity_dir, data_num), val_data_vel)

    np.save('Hoffman Cluster/train data/test_solid_randomsphere_data{}'.format(data_num), test_data_solid)
    np.save('Hoffman Cluster/train data/test_vel{}_randomsphere_data{}'.format(velocity_dir, data_num), test_data_vel)

    velocity_dir = 'x'

    augGen_seed = 10
    aug_iter = 15

    train_data_solid = np.load('Hoffman Cluster/train data/train_solid_randomsphere_augGenSeed_{}_AugIter_{}.npy'.format(augGen_seed, aug_iter))
    train_data_vel = np.load('Hoffman Cluster/train data/train_vel{}_randomsphere_augGenSeed_{}_AugIter_{}.npy'.format(velocity_dir, augGen_seed, aug_iter))

    val_data_solid = np.load('Hoffman Cluster/train data/val_solid_randomsphere_augGenSeed_{}_AugIter_{}.npy'.format(augGen_seed, aug_iter))
    val_data_vel = np.load('Hoffman Cluster/train data/val_vel{}_randomsphere_augGenSeed_{}_AugIter_{}.npy'.format(velocity_dir, augGen_seed, aug_iter))

    test_data_solid = np.load('Hoffman Cluster/train data/test_solid_randomsphere_augGenSeed_{}_AugIter_{}.npy'.format(augGen_seed,  aug_iter))
    test_data_vel = np.load('Hoffman Cluster/train data/test_vel{}_randomsphere_augGenSeed_{}_AugIter_{}.npy'.format(velocity_dir, augGen_seed, aug_iter))

    print(train_data_solid.shape)
    print(val_data_solid.shape)
    print(test_data_solid.shape)

    print(train_data_vel.shape)
    print(val_data_vel.shape)
    print(test_data_vel.shape)
```

Data Check

```
vrange = np.linspace(0,10,100)
vel_norm_nvoxel = np.zeros(vrange.shape[0]-1)

test_data_vel_flat = test_data_vel.flatten()

for i in range(vel_norm_nvoxel.shape[0]):

  vel_norm_nvoxel[i] = np.sum( test_data_vel_flat[ (test_data_vel_flat >= vrange[i]) & (test_data_vel_flat < vrange[i+1]) ] )

plt.bar(vrange[:-1], vel_norm_nvoxel)
plt.xlabel('V')
plt.ylabel('Number of voxels')
plt.show()

# Voxel histogram of minmax velocity

percentile = np.linspace(-10,10,200)
vel_nvoxel = np.zeros(percentile.shape[0]-1)

for i in range(vel_nvoxel.shape[0]):

  vel_masked = np.ma.masked_outside(uc_vel_minmax[2,:,:,:], percentile[i], percentile[i+1])
  vel_filled = np.ma.filled(vel_masked, fill_value=0)
  vel_nvoxel[i] = vel_filled[vel_filled != 0].shape[0]
```

```
plt.bar(percentile[:-1], vel_nvoxel)

plt.xlabel('V')

plt.ylabel('Number of voxels')

plt.show()


raw_data_perm = np.zeros( (5, uc_solid.shape[0]) )

norm_data_perm = np.zeros( (5, uc_solid.shape[0]) )


x_max = 50

res = 20e-6


for i in range(raw_data_perm.shape[1]):

  norm_data_perm[0,i] = uc_vel_minmax[i,:,:,:].mean()

  raw_data_perm[0,i] = norm_data_perm[0,i]*x_max*(res**2)

  raw_data_perm[1,i] = uc_vel_minmax[i,:60,:60,:].mean()*x_max*(res**2)

  raw_data_perm[2,i] = uc_vel_minmax[i,:60,60:,:].mean()*x_max*(res**2)

  raw_data_perm[3,i] = uc_vel_minmax[i,60:,:60,:].mean()*x_max*(res**2)

  raw_data_perm[4,i] = uc_vel_minmax[i,60:,60:,:].mean()*x_max*(res**2)



plt.hist(raw_data_perm[0,:], bins=20)

plt.title('Permeability')

plt.show()

plt.hist(norm_data_perm[0,:], bins=20)

plt.title('Normalized Permeability')

plt.xlim([0.14, 0.24])
plt.show()


plt.subplot(221)
plt.hist(raw_data_perm[1,:], bins=10)


plt.subplot(222)
plt.hist(raw_data_perm[2,:], bins=10)


plt.subplot(223)
plt.hist(raw_data_perm[3,:], bins=10)


plt.subplot(224)
plt.hist(raw_data_perm[4,:], bins=10)
plt.show()


train_data_perm = np.zeros( (5, train_data_solid.shape[0]) )
train_data_norm = np.zeros( (5, train_data_solid.shape[0]) )


for i in range(train_data_perm.shape[1]):
  train_data_norm[0,i] = train_data_vel[i,:,:,:,0].mean()
  train_data_perm[0,i] = train_data_norm[0,i]*x_max*(res**2)
  train_data_perm[1,i] = train_data_vel[i,:60,:60,:,0].mean()*x_max*(res**2)
  train_data_perm[2,i] = train_data_vel[i,:60,:60,:,0].mean()*x_max*(res**2)
  train_data_perm[3,i] = train_data_vel[i,:60,:60,:,0].mean()*x_max*(res**2)
  train_data_perm[4,i] = train_data_vel[i,:60,:60,:,0].mean()*x_max*(res**2)
```

```
plt.hist(train_data_perm[0,:], bins=10)
plt.title('Raw Permeability')
plt.show()

plt.hist(train_data_norm[0,:], bins=10)
plt.title('Normalized Permeability')
plt.show()

plt.subplot(221)
plt.hist(train_data_perm[1,:], bins=10)
plt.title('I')

plt.subplot(222)
plt.hist(train_data_perm[2,:], bins=10)
plt.title('II')

plt.subplot(223)
plt.hist(train_data_perm[3,:], bins=10)
plt.title('III')

plt.subplot(224)
plt.hist(train_data_perm[4,:], bins=10)
plt.title('IV')

plt.show()
```

Illustrations

```
  # Raw data
  sample_number = 2

  slice = np.array( [0, 59, -1] )
  fig_title = ['Inlet', 'Midpoint', 'Outlet']

  fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )

  vel_magnitude = 1e-4

  for j in np.array( [0, 1, 2] ):
    im=axs[0,j].imshow(uc_solid[sample_number,:,:,slice[j]], clim=(0,1), cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[0,j],fraction=0.05)
    axs[0,j].axis('off')
    axs[0,j].set_title('%s Solid' % (fig_title[j]))

    im=axs[1,j].imshow(uc_vel[sample_number,:,:,slice[j],0],          clim=(0,vel_magnitude),
cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[1,j],fraction=0.05)
    axs[1,j].axis('off')
    axs[1,j].set_title('%s Velocity' % (fig_title[j]))

  # Norm data
  sample_number = 0
```

```
slice = np.array( [0, 59, -1] )
fig_title = ['Inlet', 'Midpoint', 'Outlet']

fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )

for j in np.array( [0, 1, 2] ):
  im=axs[0,j].imshow(uc_solid[sample_number,:,:,slice[j]], clim=(0,1), cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[0,j],fraction=0.05)
  axs[0,j].axis('off')
  axs[0,j].set_title('%s Solid' % (fig_title[j]))

  im=axs[1,j].imshow(uc_vel_norm[sample_number,:,:,slice[j]], clim=(0,40), cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[1,j],fraction=0.05)
  axs[1,j].axis('off')
  axs[1,j].set_title('%s Velocity' % (fig_title[j]))

# Minmax data
sample_number = 5

slice = np.array( [0, 59, -1] )
fig_title = ['Inlet', 'Midpoint', 'Outlet']

fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )

vel_magnitude = 10
```

```
    for j in np.array( [0, 1, 2] ):
      im=axs[0,j].imshow(uc_solid[sample_number,:,:,slice[j]], clim=(0,1), cmap=plt.cm.hot)
      fig.colorbar(im,ax=axs[0,j],fraction=0.05)
      axs[0,j].axis('off')
      axs[0,j].set_title('%s Solid' % (fig_title[j]))

      im=axs[1,j].imshow(uc_vel_minmax[sample_number,:,:,slice[j],0],  clim=(0,vel_magnitude),
cmap=plt.cm.hot)
      fig.colorbar(im,ax=axs[1,j],fraction=0.05)
      axs[1,j].axis('off')
      axs[1,j].set_title('%s Velocity' % (fig_title[j]))

    # Data check - train data
    sample_number = 2

    slice = np.array( [0, 59, -1] )
    fig_title = ['Inlet', 'Midpoint', 'Outlet']

    fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )

    for j in np.array( [0, 1, 2] ):
      im=axs[0,j].imshow(train_data_solid[sample_number,:,:,slice[j],0],                    clim=(0,1),
cmap=plt.cm.hot)
      fig.colorbar(im,ax=axs[0,j],fraction=0.05)
      axs[0,j].axis('off')
      axs[0,j].set_title('%s Solid' % (fig_title[j]))

      im=axs[1,j].imshow(train_data_vel[sample_number,:,:,slice[j],0],                    clim=(-2,2),
cmap=plt.cm.hot)
      fig.colorbar(im,ax=axs[1,j],fraction=0.05)
      axs[1,j].axis('off')
      axs[1,j].set_title('%s Velocity' % (fig_title[j]))
```

Darcy Unet Training

Unet Model

```
  from tensorflow.keras.models import *
  from tensorflow.keras.layers import Input, Conv3D, Conv3DTranspose, BatchNormalization,
Activation, Concatenate, Dropout, Multiply

  from numpy import floor, ceil

  def encoder_block(inputs, strides, filter_num, filter_size, activation, momentum, rate):

    path = Conv3D(filter_num, filter_size, padding='same', strides=strides)(inputs)
    path = BatchNormalization(momentum=momentum)(path)
    path = Activation(activation=activation)(path)
    path = Dropout(rate)(path)

    return path
  def decoder_block(inputs, strides, filter_num, filter_size, activation, momentum, rate):

    path = Conv3DTranspose(filter_num, filter_size, padding='same', strides=strides)(inputs)
    path = BatchNormalization(momentum=momentum)(path)
    path = Activation(activation=activation)(path)
    path = Dropout(rate)(path)

    return path


  def UnetV1(input_shape, filter_num = 5, filter_size = 3, activation = 'selu', momentum = 0.99,
rate = 0.2):

    inputs = Input(shape = input_shape)

    skip_connection = []

    for i in range(8):

      if(i <= 0):
        path_encoder = encoder_block(inputs, 1, filter_num*(2**floor(i/2)), filter_size, activation,
momentum, rate)
      else:
        if(i % 2 == 1):
          path_encoder = encoder_block(path_encoder, 1, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
          skip_connection.append(path_encoder)
        else:
          path_encoder = encoder_block(path_encoder, 2, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)

    for i in reversed(range(6)):

      if(i >= 5):
        path_decoder = decoder_block(path_encoder, 2, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
        path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])
      else:

        if(i % 2 == 0):
          path_decoder = decoder_block(path_decoder, 1, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
        else:
          path_decoder = decoder_block(path_decoder, 2, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
          path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])


    path = Conv3D(filter_num, filter_size, padding='same')(path_decoder)
    path = Conv3D(1, 1, padding='same')(path)

    #for version 2
    #path = Multiply()([inputs, path])

    return Model(inputs=inputs, outputs=path)


  def UnetV2(input_shape, x_model, y_model, z_model, filter_num = 5, filter_size = 3, activation
= 'selu', momentum = 0.99, rate = 0.2):

    v_scale = 1

    inputs = Input(shape = input_shape)

    x_input = tf.math.multiply( x_model(inputs, training=False), 0.5*v_scale )
    y_input = tf.math.multiply( y_model(inputs, training=False), 0.5*v_scale )
    z_input = tf.math.multiply( z_model(inputs, training=False), 1*v_scale )

    path = Concatenate()( [x_input, y_input, z_input] )

    skip_connection = []

    for i in range(8):

      if(i <= 0):
        path_encoder = encoder_block(path, 1, filter_num, filter_size, activation, momentum, rate)
      else:

        if(i % 2 == 1):
          path_encoder = encoder_block(path_encoder, 1, filter_num, filter_size, activation,
momentum, rate)
          skip_connection.append(path_encoder)
        else:
          path_encoder = encoder_block(path_encoder, 2, filter_num, filter_size, activation,
momentum, rate)

    for i in reversed(range(6)):

      if(i >= 5):
        path_decoder = decoder_block(path_encoder, 2, filter_num, filter_size, activation,
momentum, rate)
        path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])
      else:

        if(i % 2 == 0):
          path_decoder = decoder_block(path_decoder, 1, filter_num, filter_size, activation,
momentum, rate)
        else:
          path_decoder = decoder_block(path_decoder, 2, filter_num, filter_size, activation,
momentum, rate)
          path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])
    path = Conv3D(filter_num, filter_size, padding='same')(path_decoder)
    path = Conv3D(3, 1, padding='same')(path)

    return Model(inputs=inputs, outputs=path)
```

Model Training

```
  version = 'Z1-4-Seed3'

  model_name = 'UnetRS_Modelv{}'.format(version)

  dir_save   = 'RandomSphere Model/{}'.format(model_name)
  try:
    os.mkdir(dir_save)
  except OSError as error:
    print(error)

  filter_size = 4
  num_filters = 10

  learning_rate = 0.0006
  batch_size    = 4

  momentum = 0.9
```

```
    rate = 0.1


    epochs       = 1000
    patience_training = 50


    metrics=['MAE', 'MSE']
    optimizer = tf.keras.optimizers.Adam(learning_rate = learning_rate)


    model = UnetV1( input_shape = (data_size, data_size, data_size, 1), filter_num = num_filters,
filter_size = filter_size, activation = 'selu', momentum = momentum, rate = rate)


    #model.summary(line_length = 250)


    model.compile(     loss     =     tf.keras.losses.mean_squared_error,     optimizer=optimizer,
metrics=metrics[:] )


    nan_terminate = tf.keras.callbacks.TerminateOnNaN()
    early_stop    = tf.keras.callbacks.EarlyStopping(monitor ='val_loss', min_delta = 0,
                                      patience = patience_training,
                                      verbose = True, mode = 'auto', baseline = None)


    csv_logger                                                                              =
tf.keras.callbacks.CSVLogger("{}/training_log_{}.csv".format(dir_save,model_name))


    checkpoint = ModelCheckpoint('{}/{}.ckpt'.format(dir_save,model_name),
                          monitor = 'val_loss',
                          verbose = 1,
                          save_best_only = True,
                          mode = 'min', save_weights_only = False)

    callbacks_list = [nan_terminate,
                early_stop,
                checkpoint,
                csv_logger]

    from keras.utils.vis_utils import plot_model

    imgtype = ('LR','TB')
    for i in imgtype:
      plot_model(model,                   to_file='RandomSphere                   Model/Model
Image/{}_{}.jpg'.format(model_name, i), rankdir = i, show_layer_names = False)

    from timeit import default_timer as timer
    from datetime import timedelta

    start = timer()

    model.fit( x = train_data_solid, y = train_data_vel,
            epochs = epochs, batch_size = batch_size,
            validation_data = (val_data_solid, val_data_vel),
            validation_freq = 1,
            verbose = 1,
            callbacks = callbacks_list)

    end = timer()
    print('Elapsed time for training : {}'.format(timedelta(seconds=end-start)))
```

XYZ Model Training

```
    version = 'XYZ1-8-Seed3'

    model_name = 'UnetRSXYZ_Modelv{}'.format(version)

    dir_save   = 'RandomSphere XYZ Model/{}'.format(model_name)
    try:
      os.mkdir(dir_save)
    except OSError as error:
      print(error)

    filter_size = 3
    num_filters = 9

    learning_rate = 0.001
    batch_size    = 4

    momentum = 0.99
    rate = 0.0001
    epochs        = 1000
    patience_training = 50

    metrics=['MAE', 'MSE']
    optimizer = tf.keras.optimizers.Adam(learning_rate = learning_rate)


    version_x = 'X1-3'
    x_model_name = 'UnetRS_Modelv{}'.format(version_x)
    x_model            =            tf.keras.models.load_model(            'RandomSphere
Model/{}/{}.ckpt'.format(x_model_name, x_model_name ) )
    x_model._name = 'xmodel'
    x_model.trainable = False

    version_y = 'Y1-1'
    y_model_name = 'UnetRS_Modelv{}'.format(version_y)
    y_model            =            tf.keras.models.load_model(            'RandomSphere
Model/{}/{}.ckpt'.format(y_model_name, y_model_name ) )
    y_model._name = 'ymodel'
    y_model.trainable = False

    version_z = 'Z1-4'
    z_model_name = 'UnetRS_Modelv{}'.format(version_z)
    z_model            =            tf.keras.models.load_model(            'RandomSphere
Model/{}/{}.ckpt'.format(z_model_name, z_model_name ) )
    z_model._name = 'zmodel'
    z_model.trainable = False

    model = UnetV2( input_shape = (data_size, data_size, data_size, 1), x_model = x_model,
y_model = y_model, z_model = z_model,
                filter_num = num_filters, filter_size = filter_size, activation = 'selu', momentum =
momentum, rate = rate)

    #model.summary(line_length = 250)

    #model.compile(    loss    =    tf.keras.losses.mean_squared_error,    optimizer=optimizer,
metrics=metrics[:] )
    model.compile( loss = div_loss2, optimizer=optimizer, metrics=metrics[:])

    nan_terminate = tf.keras.callbacks.TerminateOnNaN()
    early_stop    = tf.keras.callbacks.EarlyStopping(monitor ='val_loss', min_delta = 0,
                                    patience = patience_training,
                                    verbose = True, mode = 'auto', baseline = None)

    csv_logger                                                                    =
tf.keras.callbacks.CSVLogger("{}/training_log_{}.csv".format(dir_save,model_name))

    checkpoint = ModelCheckpoint('{}/{}.ckpt'.format(dir_save,model_name),
                        monitor = 'val_loss',
                        verbose = 1,
                        save_best_only = True,
                        mode = 'min', save_weights_only = False)
```

```
callbacks_list = [nan_terminate,
                  early_stop,
                  checkpoint,
                  csv_logger]

from keras.utils.vis_utils import plot_model

imgtype = ('LR','TB')
for i in imgtype:
  plot_model(model,                    to_file='RandomSphere                    Model/Model
Image/{}_{}.jpg'.format(model_name, i), rankdir = i, show_layer_names = False)

from timeit import default_timer as timer
from datetime import timedelta

start = timer()

model.fit( x = train_data_solid, y = train_data_vel,
           epochs = epochs, batch_size = batch_size,
           validation_data = (val_data_solid, val_data_vel),
           validation_freq = 1,
           verbose = 1,
           callbacks = callbacks_list)

end = timer()
```

```
  print('Elapsed time for training : {}'.format(timedelta(seconds=end-start)))
```

Model Evaluation

```
  #version = 'Z1-4'
  #eval_model_name = 'UnetRS_Modelv{}'.format(version)
  #eval_model = tf.keras.models.load_model( 'RandomSphere Model/{}/{}.ckpt'.format(eval_model_name, eval_model_name ) )

  version = 'XYZ1-0'
  eval_model_name = 'UnetRSXYZ_Modelv{}'.format(version)
  eval_model = tf.keras.models.load_model( 'RandomSphere XYZ Model/{}/{}.ckpt'.format(eval_model_name, eval_model_name ) )
  #eval_model = tf.keras.models.load_model( 'RandomSphere XYZ Model/{}/{}.ckpt'.format(eval_model_name, eval_model_name ), custom_objects = {'div_loss2': div_loss2} )

  for i in range(test_data_solid.shape[0]):
    print('\nSample number : {}'.format(i+1))
    eval_model.evaluate(test_data_solid[i:i+1,:,:,:,:], test_data_vel[i:i+1,:,:,:,:])

  vz_test_pred = np.float32( eval_model.predict( x=[test_data_solid] ) )

  print(vz_test_pred.shape)

  # Overall Permeability
```

```
perm_true = test_data_vel[:,:,:,:,2].mean(axis=(1,2,3))
perm_pred = vz_test_pred[:,:,:,:,2].mean(axis=(1,2,3))

perm_error = abs( perm_true - perm_pred )/abs(perm_true)*100

print( 'Overall permeability error : {:.3f}\n'.format(perm_error.mean()) )

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(np.arange(perm_error.shape[0]), perm_error)
plt.title('Permeability Error')
plt.show()

"""
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(np.arange(perm_error.shape[0]),test_data_vel[:,:,:,:,0].mean(axis=(1,2,3)))
plt.title('Average velocity')
plt.show()
"""

print(perm_error)

# Voxel-wise MAPE Absolute value velocity
v_max = 5
```

```
  channel = 2

  vel_true_flat = test_data_vel[:,:,:,:,channel].flatten()
  vel_pred_flat = vz_test_pred[:,:,:,:,channel].flatten()

  v_range = np.linspace(0.0001,v_max,100)

  vel_true_flat_nonzero = vel_true_flat[ (vel_true_flat < 0) | (vel_true_flat > 0)]
  vel_pred_flat_nonzero = vel_pred_flat[ (vel_true_flat < 0) | (vel_true_flat > 0)]

  vel_mape_flat = (vel_true_flat_nonzero - vel_pred_flat_nonzero)/vel_true_flat_nonzero*100

  plt.plot(vel_true_flat_nonzero,
vel_mape_flat,'bo',markersize=0.1,figure=plt.figure(figsize=[8,6]))
  plt.ylim(-100,100)
  plt.show()



  vel_mape_flat_range = np.zeros_like(v_range)

  for i in range(v_range.shape[0]-1):
    if(v_range[i] != 0):
      vel_mape_flat_vrange = np.abs( vel_mape_flat[ (np.abs(vel_true_flat_nonzero) > v_range[i])
& (np.abs(vel_true_flat_nonzero) < v_range[i+1]) ] )

    if(vel_mape_flat_vrange.shape[0] > 0):
      vel_mape_flat_range[i] = np.mean(vel_mape_flat_vrange)

  plt.plot(v_range[:-1], vel_mape_flat_range[:-1], 'bo',markersize=4,figure=plt.figure(figsize=[8,6]))
  plt.ylim(0,100)
  plt.xlabel('velocity')
  plt.ylabel('% error')
  plt.show()



  threshold = 9.17/x_max

  vel_mape_flat_threshold = vel_mape_flat[ (vel_true_flat_nonzero <= (-1*threshold)) | (vel_true_flat_nonzero >= (1*threshold)) ]
  print("% Error above threshold : {}".format(np.mean(vel_mape_flat_threshold[vel_mape_flat_threshold > 0])))

  #print(v_range)
  #print(vel_mape_flat_range)

  test_sample = 1
  print( 'Maximum Sample Number : {}'.format(test_data_solid.shape[0] - 1) )

  mySolid = test_data_solid[test_sample,:,:,:,0]
  mySolid_mask = np.ma.masked_less(mySolid,1)
```

```
myVel_true = test_data_vel[test_sample,:,:,:,2]
myVel_true = np.ma.array( myVel_true, mask=np.ma.getmask(mySolid_mask) )

myVel_pred = vz_test_pred[test_sample,:,:,:,2]
myVel_pred = np.ma.array( myVel_pred, mask=np.ma.getmask(mySolid_mask) )

myVel_true = myVel_true*x_max
myVel_pred = myVel_pred*x_max

test_slice = np.array( [0, 59, -1] )
fig_title = ['Inlet', 'Midpoint', 'Outlet']

fig, axs = plt.subplots( nrows=3, ncols=3,figsize=(20,20) )

vel_range = (0,50)

for j in range(3):

  im=axs[j,0].imshow(mySolid[:,:,test_slice[j]], clim=(0,1), cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,0],fraction=0.05)
  axs[j,0].axis('off')
  #axs[j,0].set_title('{} Solid'.format(fig_title[j]))

  im=axs[j,1].imshow(myVel_true[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,1],fraction=0.05)
  axs[j,1].axis('off')
  #axs[j,1].set_title('{} Simulation'.format(fig_title[j]))


  im=axs[j,2].imshow(myVel_pred[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,2],fraction=0.05)
  axs[j,2].axis('off')
  #axs[j,2].set_title('{} Prediction'.format(fig_title[j]))

div = np.zeros( (vz_test_pred.shape[0],118,118,118) )
for i in range(vz_test_pred.shape[0]):

  temp_vel = vz_test_pred[i,:,:,:,:]


  dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
  dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
  dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
  div[i,:,:,:] = dVxdx + dVydy + dVzdz

print( np.mean( np.abs(div), axis=(1,2,3)) )


div_flat = div.flatten()
plt.hist(div_flat, np.linspace(-0.1,0.1,100))
plt.show()


print(np.mean( np.abs(div_flat) ))


#temp_vel = vz_test_pred[0,:,:,:,:]*test_data_solid[0,:,:,:,:]
```

```
temp_vel = vz_test_pred[0,:,:,:,:]

dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
div = dVxdx + dVydy + dVzdz

plt.imshow(test_data_solid[0,:,:,59,0])
plt.show()

plt.imshow(temp_vel[:,:,59,2])
plt.colorbar()
plt.show()

plt.imshow(div[:,:,59])
plt.colorbar()
plt.show()

div_flat = div.flatten()
plt.hist(div_flat, np.linspace(-0.1,0.1,100))
plt.show()

print(np.mean( np.abs(div_flat) ))

temp_vel = test_data_vel[0,:,:,:,:]
```

```
    dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
    dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
    dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
    div = dVxdx + dVydy + dVzdz

    plt.imshow(test_data_solid[0,:,:,5,0])
    plt.show()

    plt.imshow(temp_vel[:,:,5,2])
    plt.colorbar()
    plt.show()

    plt.imshow(div[:,:,5])
    plt.colorbar()
    plt.show()

    div_flat = div.flatten()
    plt.hist(div_flat, np.linspace(-0.1,0.1,100))
    plt.show()
```

Evaluate SiC Data

Load SiC Data

```
    sic_dir_data = 'Data/SiC Data'
    data_size = 120
    denRange = [45, 65, 80]
    domainRange = [1, 2]

    # X : 'x'
    # Y : 'y'
    # Z : ''
    #velocity_dir = ('x')
    #velocity_dir = ('y')
    #velocity_dir = ('z')
    velocity_dir = ('x', 'y', 'z')

    channel_size = len(velocity_dir)

    sic_solid = np.zeros( (1,data_size,data_size,data_size,1) )
    sic_vel = np.zeros( (1,data_size,data_size,data_size,channel_size) )

    for i in range(len(denRange)):
      for j in range(len(domainRange)):

        sic_solid_load = loadmat( '{}/{}PPI_domain{}_deci.mat'.format(sic_dir_data, denRange[i],
domainRange[j]) )['solid'].astype('int')
        sic_solid_load = sic_solid_load <= 0
        sic_solid = np.append( sic_solid, np.expand_dims(sic_solid_load, axis=(0,-1)), axis=0 )

        sic_vel_load = np.zeros( (1,data_size,data_size,data_size,channel_size))
        for k in range(channel_size):
          if(velocity_dir[k] == 'z'):
            sic_vel_load[0,:,:,:,k] = loadmat( '{}/{}PPI_domain{}_vfield{}.mat'.format(sic_dir_data,
denRange[i], domainRange[j], '') )['vfield'].astype('float32')
          else:
            sic_vel_load[0,:,:,:,k] = loadmat( '{}/{}PPI_domain{}_vfield{}.mat'.format(sic_dir_data,
denRange[i], domainRange[j], velocity_dir[k]) )['vfield'].astype('float32')

        sic_vel = np.append( sic_vel, sic_vel_load, axis=0 )

    sic_solid = sic_solid[1:,:,:,:]
    sic_vel = sic_vel[1:,:,:,:]

    print('Image size : {}\nNumber of Data : {}    {}'.format(data_size, sic_solid.shape,
sic_vel.shape))

    """ Normalization
    """

    res = 26.708e-6
    res2 = res/2

    x_min = 0
    x_max = 8

    sic_vel_norm = sic_vel[:4,:,:,:,:]/(res**2)*0.333/9270
    sic_vel_norm = np.append(sic_vel_norm,sic_vel[4:,:,:,:,:]/(res2**2)*0.333/9270,axis=0)

    print( '\nMean velocity : {}'.format( sic_vel_norm.mean() ) )
    print( '\nMin velocity : {}'.format( sic_vel_norm.min() ) )
    print( '\nMax velocity : {}'.format( sic_vel_norm.max() ) )

    sic_vel_minmax = minmax_transform(sic_vel_norm, x_min=x_min, x_range=x_max-x_min)

    print( '\nMean velocity : {}'.format( sic_vel_minmax.mean() ) )
    print( '\nMin velocity : {}'.format( sic_vel_minmax.min() ) )
    print( '\nMax velocity : {}'.format( sic_vel_minmax.max() ) )
```

SiC Data check

```
data_num = 1

np.save('Data/SiC Data/sic_solid_{}'.format(data_num), sic_solid)

vrange = np.linspace(-30,30,200)
vel_norm_nvoxel = np.zeros(vrange.shape[0]-1)

sic_vel_norm_flat = sic_vel_norm.flatten()

for i in range(vel_norm_nvoxel.shape[0]):
      vel_norm_nvoxel[i] = np.abs( np.sum( sic_vel_norm_flat[ (sic_vel_norm_flat >= vrange[i])
& (sic_vel_norm_flat < vrange[i+1]) ] ) )

    plt.bar(vrange[:-1], vel_norm_nvoxel)
    plt.xlabel('V')
    plt.ylabel('Number of voxels')
    plt.show()

    print(np.sum(sic_vel_norm_flat[sic_vel_norm_flat >= 25]))

    percentile = np.linspace(-5,5,100)
    vel_norm_nvoxel = np.zeros(percentile.shape[0]-1)

    for i in range(vel_norm_nvoxel.shape[0]):

      vel_norm_masked = np.ma.masked_outside(sic_vel_minmax, percentile[i], percentile[i+1])
      vel_norm_filled = np.ma.filled(vel_norm_masked, fill_value=0)
      vel_norm_nvoxel[i] = vel_norm_filled[vel_norm_filled != 0].shape[0]

    plt.bar(percentile[:-1], vel_norm_nvoxel)
    plt.xlabel('V')
    plt.ylabel('Number of voxels')
    plt.show()

    raw_sic_perm = np.zeros( (5, sic_solid.shape[0]) )
    norm_sic_perm = np.zeros( (5, sic_solid.shape[0]) )
```

```
x_max = 50
res = 26.708e-6

for i in range(raw_sic_perm.shape[1]):
  norm_sic_perm[0,i] = sic_vel_minmax[i,:,:,:].mean()
  raw_sic_perm[0,i] = norm_sic_perm[0,i]*x_max*(res**2)

plt.hist(raw_sic_perm[0,:], bins=20)
plt.title('Raw Permeability')
plt.show()
plt.hist(norm_sic_perm[0,:], bins=20)
plt.title('Normalized Permeability')
plt.xlim([0.1, 0.24])
plt.show()

# Data check
sample_number = 5

slice = np.array( [0, 59, -1] )
fig_title = ['Inlet', 'Midpoint', 'Outlet']

v_max = 9

fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )
```

```
  for j in np.array( [0, 1, 2] ):
    im=axs[0,j].imshow(sic_solid[sample_number,:,:,slice[j],0], clim=(0,1), cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[0,j],fraction=0.05)
    axs[0,j].axis('off')
    axs[0,j].set_title('%s Solid' % (fig_title[j]))

    im=axs[1,j].imshow(sic_vel_minmax[sample_number,:,:,slice[j],0],          clim=(0,v_max),
cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[1,j],fraction=0.05)
    axs[1,j].axis('off')
    axs[1,j].set_title('%s Velocity' % (fig_title[j]))
```

Evaluation

```
  case_num = 3

  if(case_num == 1):
    version = 'Z1-4'
    eval_model_name = 'UnetRS_Modelv{}'.format(version)
    eval_model          =          tf.keras.models.load_model(          'RandomSphere
Model/{}/{}.ckpt'.format(eval_model_name, eval_model_name ) )

  elif(case_num == 2):
    version = 'XYZ1-0'
    eval_model_name = 'UnetRSXYZ_Modelv{}'.format(version)
    eval_model = tf.keras.models.load_model( 'RandomSphere XYZ Model/{}/{}.ckpt'.format(eval_model_name, eval_model_name ) )

  elif(case_num == 3):
    version = 'XYZ1-8'
    eval_model_name = 'UnetRSXYZ_Modelv{}'.format(version)
    eval_model = tf.keras.models.load_model( 'RandomSphere XYZ Model/{}/{}.ckpt'.format(eval_model_name, eval_model_name ), custom_objects = {'div_loss2': div_loss2} )

  """
  for i in range(sic_solid.shape[0]):
    print('Sample number : {}'.format(i+1))
    eval_model.evaluate(sic_solid[i:i+1,:,:,:,:], sic_vel_minmax[i:i+1,:,:,:,:])
  """

  from timeit import default_timer as timer
  from datetime import timedelta

  start = timer()
  vz_test_pred = np.float32( eval_model.predict( x=[sic_solid] ) )
  end = timer()
  print('Elapsed time : {}'.format(timedelta(seconds=end-start)))

  print(vz_test_pred.shape)
```

```
eval_model.save('Saved Model/{}'.format(eval_model_name))

# Overall Permeability

perm_true = sic_vel_minmax[:,:,:,:,2].mean(axis=(1,2,3))
perm_pred = vz_test_pred[:,:,:,:,2].mean(axis=(1,2,3))

perm_error = abs( (perm_true - perm_pred)/perm_true )*100

print( 'Overall permeability error : {:.3f}\n'.format(perm_error.mean()) )

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(np.arange(perm_error.shape[0]),perm_error)
plt.title('Permeability Error')
plt.show()
"""
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(np.arange(perm_error.shape[0]), vz_test_pred[:,:,:,:].mean(axis=(1,2,3,4)))
plt.title('Average velocity')
plt.show()
"""
print(perm_error)

# Voxel-wise MAPE Absolute value velocity
```

```
  v_max = 2
  channel = 2

  vel_true_flat = sic_vel_minmax[:,:,:,:,channel].flatten()
  vel_pred_flat = vz_test_pred[:,:,:,:,channel].flatten()

  v_range = np.linspace(0.0001,v_max,100)

  vel_true_flat_nonzero = vel_true_flat[ (vel_true_flat < 0) | (vel_true_flat > 0)]
  vel_pred_flat_nonzero = vel_pred_flat[ (vel_true_flat < 0) | (vel_true_flat > 0)]

  vel_mape_flat = (vel_true_flat_nonzero - vel_pred_flat_nonzero)/vel_true_flat_nonzero*100

  plt.plot(vel_true_flat_nonzero,
vel_mape_flat,'bo',markersize=0.1,figure=plt.figure(figsize=[8,6]))
  plt.ylim(-100,100)
  plt.show()



  vel_mape_flat_range = np.zeros_like(v_range)

  for i in range(v_range.shape[0]-1):
    if(v_range[i] != 0):
      vel_mape_flat_vrange = np.abs( vel_mape_flat[ (np.abs(vel_true_flat_nonzero) > v_range[i])
& (np.abs(vel_true_flat_nonzero) < v_range[i+1]) ] )
```

```
      if(vel_mape_flat_vrange.shape[0] > 0):
        vel_mape_flat_range[i] = np.mean(vel_mape_flat_vrange)

    plt.plot(v_range[:-1],                                                     vel_mape_flat_range[:-1],
'bo',markersize=4,figure=plt.figure(figsize=[8,6]))
    plt.ylim(0,100)
    plt.xlabel('velocity')
    plt.ylabel('% error')
    plt.show()



    threshold = 8.93/x_max

    vel_mape_flat_threshold = vel_mape_flat[ (vel_true_flat_nonzero <= (-1*threshold)) |
(vel_true_flat_nonzero >= (1*threshold)) ]
    print("%                 Error                 above                 threshold                 :
{}".format(np.mean(vel_mape_flat_threshold[vel_mape_flat_threshold > 0])))

    #print(v_range)
    #print(vel_mape_flat_range)

    test_sample = 5
    print( 'Maximum Sample Number : {}'.format(sic_solid.shape[0] - 1) )

    mySolid = sic_solid[test_sample,:,:,:,0]
    mySolid_mask = np.ma.masked_less(mySolid, 1)
```

```
myVel_true = sic_vel_minmax[test_sample,:,:,:,2]
myVel_true = np.ma.array( myVel_true, mask=np.ma.getmask(mySolid_mask) )

myVel_pred = vz_test_pred[test_sample,:,:,:,2]
myVel_pred = np.ma.array( myVel_pred, mask=np.ma.getmask(mySolid_mask) )

myVel_true = myVel_true*x_max
myVel_pred = myVel_pred*x_max

test_slice = np.array( [0, 59, -1] )
fig_title = ['Inlet', 'Midpoint', 'Outlet']

fig, axs = plt.subplots( nrows=3, ncols=3,figsize=(20,20) )

vel_range = (0,40)

for j in range(3):

  im=axs[j,0].imshow(mySolid[:,:,test_slice[j]], clim=(0,1), cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,0],fraction=0.05)
  axs[j,0].axis('off')
  #axs[j,0].set_title('{} Solid'.format(fig_title[j]))

  im=axs[j,1].imshow(myVel_true[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,1],fraction=0.05)
  axs[j,1].axis('off')
  #axs[j,1].set_title('{} Simulation'.format(fig_title[j]))

  im=axs[j,2].imshow(myVel_pred[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,2],fraction=0.05)
  axs[j,2].axis('off')
  #axs[j,2].set_title('{} Prediction'.format(fig_title[j]))

div = np.zeros( (vz_test_pred.shape[0],118,118,118) )
for i in range(vz_test_pred.shape[0]):

  temp_vel = vz_test_pred[i,:,:,:,:]

  dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
  dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
  dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
  div[i,:,:,:] = dVxdx + dVydy + dVzdz

print( np.mean( np.abs(div), axis=(1,2,3)) )

div_flat = div.flatten()
plt.hist(div_flat, np.linspace(-0.1,0.1,100))
plt.show()

print(np.mean( np.abs(div_flat) ))
```

```
    version_x = 'X1-3'
    x_model_name = 'UnetRS_Modelv{}'.format(version_x)
    x_model = tf.keras.models.load_model( 'RandomSphere Model/{}/{}.ckpt'.format(x_model_name, x_model_name ) )
    x_model._name = 'xmodel'
    x_model.trainable = False

    version_y = 'Y1-1'
    y_model_name = 'UnetRS_Modelv{}'.format(version_y)
    y_model = tf.keras.models.load_model( 'RandomSphere Model/{}/{}.ckpt'.format(y_model_name, y_model_name ) )
    y_model._name = 'ymodel'
    y_model.trainable = False

    version_z = 'Z1-4'
    z_model_name = 'UnetRS_Modelv{}'.format(version_z)
    z_model = tf.keras.models.load_model( 'RandomSphere Model/{}/{}.ckpt'.format(z_model_name, z_model_name ) )
    z_model._name = 'zmodel'
    z_model.trainable = False

    vx_test_pred = np.float32( x_model.predict( x=[sic_solid] ) )/2
    vy_test_pred = np.float32( y_model.predict( x=[sic_solid] ) )/2
    vz_test_pred = np.float32( z_model.predict( x=[sic_solid] ) )
```

```
  vz_test_pred = np.append(vx_test_pred, np.append(vy_test_pred, vz_test_pred,axis=-1),axis=-
1 )
  print(vz_test_pred.shape)

  div = np.zeros( (vz_test_pred.shape[0],118,118,118) )
  for i in range(vz_test_pred.shape[0]):

    temp_vel = vz_test_pred[i,:,:,:,:]

    dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
    dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
    dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
    div[i,:,:,:] = dVxdx + dVydy + dVzdz

  print( np.mean( np.abs(div), axis=(1,2,3)) )

  div_flat = div.flatten()
  plt.hist(div_flat, np.linspace(-0.1,0.1,100))
  plt.show()
```

Save prediction

```
  save_index = 0
  case_name = '45PPI_domain1_vfield{}'.format(velocity_dir)
  scipy.io.savemat(                    'Prediction/{}.mat'.format(case_name),                    {'vfield_ML':
vz_test_pred[save_index,:,:,:]} )

  # Individual Model
  case_name = (45, 65, 80)

  for i in range(6):
    save_name = '{}PPI_domain{}_vfield{}'.format(case_name[i//2], i%2+1, velocity_dir)
    scipy.io.savemat(      'Prediction/UnetRS_Model/{}.mat'.format(save_name),      {'vfield_ML':
np.squeeze( vz_test_pred[i,:,:,:,:] )} )

  # Comprehensive Model
  case_name = (45, 65, 80)
  vel_dir_save = ('x', 'y', '')

  for i in range(6):
    for j in range(3):
      save_name = '{}PPI_domain{}_vfield{}'.format(case_name[i//2], i%2+1, vel_dir_save[j])
      scipy.io.savemat(     'Prediction/UnetRS_Model/{}.mat'.format(save_name),      {'vfield_ML':
np.squeeze(vz_test_pred[i,:,:,:,j]) } )
```

Divergence check

```
  sic_solid_temp = sic_solid[1:2,:,:,:,:]
  temp_vel = sic_vel_minmax[1,:,:,:,:]
```

```
dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
div = dVxdx + dVydy + dVzdz



plt.imshow(sic_solid_temp[0,:,:,59,0])
plt.show()


plt.imshow(temp_vel[:,:,59,2], vmin=0, vmax=6)
plt.colorbar()
plt.show()


plt.imshow(div[:,:,59],vmin=-0.3,vmax=0.3)
plt.colorbar()
plt.show()


div_flat = div.flatten()
plt.hist(div_flat, np.linspace(-0.3,0.3,100))
plt.show()


print(np.mean( np.abs(div_flat) ) )


sic_solid_temp = sic_solid[1:2,:,:,:,:]


xversion = 'X1-3'
```

```
    x_model_name = 'UnetRS_Modelv{}'.format(xversion)
    x_model = tf.keras.models.load_model( 'RandomSphere
Model/{}/{}.ckpt'.format(x_model_name, x_model_name ) )

    yversion = 'Y1-1'
    y_model_name = 'UnetRS_Modelv{}'.format(yversion)
    y_model = tf.keras.models.load_model( 'RandomSphere
Model/{}/{}.ckpt'.format(y_model_name, y_model_name ) )

    zversion = 'Z1-4'
    z_model_name = 'UnetRS_Modelv{}'.format(zversion)
    z_model = tf.keras.models.load_model( 'RandomSphere
Model/{}/{}.ckpt'.format(z_model_name, z_model_name ) )

    vx_test_pred = np.float32( x_model.predict( x=[sic_solid_temp] ) )/2
    vy_test_pred = np.float32( y_model.predict( x=[sic_solid_temp] ) )/2
    vz_test_pred = np.float32( z_model.predict( x=[sic_solid_temp] ) )

    print(vz_test_pred.shape)

    temp_vel = np.append(vx_test_pred, np.append(vy_test_pred, vz_test_pred,axis=-1),axis=-1 )
    print(temp_vel.shape)

    temp_vel = temp_vel[0,:,:,:,:]
    dVxdx = (temp_vel[2:,1:-1,1:-1,0] - temp_vel[:-2,1:-1,1:-1,0])/2
    dVydy = (temp_vel[1:-1,2:,1:-1,1] - temp_vel[1:-1,:-2,1:-1,1])/2
```

```
dVzdz = (temp_vel[1:-1,1:-1,2:,2] - temp_vel[1:-1,1:-1,:-2,2])/2
div = dVxdx + dVydy + dVzdz



plt.imshow(sic_solid_temp[0,:,:,59,0])
plt.show()


plt.imshow(temp_vel[:,:,59,2],vmin=0,vmax=6)
plt.colorbar()
plt.show()


plt.imshow(div[:,:,59],vmin=-0.3,vmax=0.3)
plt.colorbar()
plt.show()


div_flat = div.flatten()
plt.hist(div_flat, np.linspace(-0.3,0.3,100))
plt.show()


print(np.mean( np.abs(div_flat) ) )


temp_div_loss = div_loss1(sic_vel_minmax[1:2,:,:,:,:], np.expand_dims(temp_vel,axis=0))


print(temp_div_loss)
```

Appendix C

The source code used to develop and train the deep learning model in Chapter 4 is provided as plain text here

System Information

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
print(gpu_info)
```

Initialize Package

```
!pip install hdf5storage

Installing collected packages: hdf5storage
Successfully installed hdf5storage-0.1.19

from hdf5storage import loadmat

import numpy as np
from numpy.random import seed
from numpy.random import default_rng

from matplotlib import pyplot as plt
import os
import scipy
from scipy import io
from scipy.ndimage import zoom
from scipy.stats import ks_2samp

import tensorflow as tf
print('tensorflow version : {}'.format(tf.__version__))
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.models import Model
from tensorflow.keras.models import load_model
from tensorflow.keras.models import save_model

from sklearn.model_selection import train_test_split

seed = 10

tf.random.set_seed(seed)
np.random.seed(seed)
```

```
tensorflow version : 2.11.0
```

```
%cd /content/drive/My\ Drive/DarcyUnet Unit Cell/
```

```
/content/drive/My Drive/DarcyUnet Unit Cell
```

Functions

```
def minmax_transform(x, x_min=0, x_range=1):
  x = ( x - x_min ) / x_range
  return x



def shift_augmentation(my_solid, my_vel, shift_range, vel_dir):
  # my_solid : NxNxN
  # my_vel : NxNxN
  # shift_range : 2x1 in voxels
  # vel_dir : string

  # Augmentation placeholder
  my_aug_solid = np.ones_like(my_solid)
  my_aug_vel = np.ones_like(my_vel)

  # Ratio of shift value
  shift_x = shift_range[0]
  shift_y = shift_range[1]
  shift_z = shift_range[2]

  # X Shift
  if(shift_x < 0):
```

```
  my_aug_solid[:shift_x,:,:] = my_solid[-1*shift_x:,:,:]
  my_aug_solid[shift_x:,:,:] = np.flip(my_solid, axis=0 )[:-1*shift_x,:,:]

  my_aug_vel[:shift_x,:,:] = my_vel[-1*shift_x:,:,:]
  if(vel_dir[0] == 'x'):
    my_aug_vel[shift_x:,:,:] = np.flip(-1*my_vel, axis=0 )[:-1*shift_x,:,:]
  else:
    my_aug_vel[shift_x:,:,:] = np.flip(my_vel, axis=0 )[:-1*shift_x,:,:]

elif(shift_x > 0):
  my_aug_solid[:shift_x,:,:] = np.flip(my_solid, axis=0 )[-1*shift_x:,:,:]
  my_aug_solid[shift_x:,:,:] = my_solid[:-1*shift_x,:,:]

  my_aug_vel[shift_x:,:,:] = my_vel[:-1*shift_x,:,:]
  if(vel_dir[0] == 'x'):
    my_aug_vel[:shift_x,:,:] = np.flip(-1*my_vel, axis=0 )[-1*shift_x:,:,:]
  else:
    my_aug_vel[:shift_x,:,:] = np.flip(my_vel, axis=0 )[-1*shift_x:,:,:]

else:
  my_aug_solid = my_solid
  my_aug_vel = my_vel

my_solid = my_aug_solid
my_vel = my_aug_vel
```

```
# Y Shift
my_aug_solid = np.ones_like(my_solid)
my_aug_vel = np.ones_like(my_vel)

if(shift_y < 0):
  my_aug_solid[:,:shift_y,:] = my_solid[:,-1*shift_y:,:]
  my_aug_solid[:,shift_y:,:] = np.flip(my_solid, axis=1 )[:,:-1*shift_y,:]

  my_aug_vel[:,:shift_y,:] = my_vel[:,-1*shift_y:,:]
  if(vel_dir[0] == 'y'):
    my_aug_vel[:,shift_y:,:] = np.flip(-1*my_vel, axis=1 )[:,:-1*shift_y,:]
  else:
    my_aug_vel[:,shift_y:,:] = np.flip(my_vel, axis=1 )[:,:-1*shift_y,:]

elif(shift_y > 0):
  my_aug_solid[:,:shift_y,:] = np.flip(my_solid, axis=1 )[:,-1*shift_y:,:]
  my_aug_solid[:,shift_y:,:] = my_solid[:,:-1*shift_y,:]

  my_aug_vel[:,shift_y:,:] = my_vel[:,:-1*shift_y,:]
  if(vel_dir[0] == 'y'):
    my_aug_vel[:,:shift_y,:] = np.flip(-1*my_vel, axis=1 )[:,-1*shift_y:,:]
  else:
    my_aug_vel[:,:shift_y,:] = np.flip(my_vel, axis=1 )[:,-1*shift_y:,:]

else:
  my_aug_solid = my_solid
  my_aug_vel = my_vel

my_solid = my_aug_solid
my_vel = my_aug_vel

# Z Shift
my_aug_solid = np.ones_like(my_solid)
my_aug_vel = np.ones_like(my_vel)

if(shift_z < 0):
  my_aug_solid[:,:,:shift_z] = my_solid[:,:,-1*shift_z:]
  my_aug_solid[:,:,shift_z:] = np.flip( my_solid, axis=2 )[:,:,:-1*shift_z]

  my_aug_vel[:,:,:shift_z] = my_vel[:,:,-1*shift_z:]
  my_aug_vel[:,:,shift_z:] = np.flip( my_vel, axis=2 )[:,:,:-1*shift_z]

elif(shift_z > 0):
  my_aug_solid[:,:,:shift_z] = np.flip( my_solid, axis=2 )[:,:,-1*shift_z:]
  my_aug_solid[:,:,shift_z:] = my_solid[:,:,:-1*shift_z]

  my_aug_vel[:,:,:shift_z] = np.flip( my_vel, axis=2 )[:,:,-1*shift_z:]
  my_aug_vel[:,:,shift_z:] = my_vel[:,:,:-1*shift_z]

else:
  my_aug_solid = my_solid
  my_aug_vel = my_vel
```

```
    return my_aug_solid, my_aug_vel
```

Custom Loss

```
  def div_loss1(y_true, y_pred):

    scale = 1

    mse = tf.math.reduce_mean( tf.math.square(y_true - y_pred) )

    dVxdx_true = (y_true[:,2:,1:-1,1:-1,0] - y_true[:,:-2,1:-1,1:-1,0])/2
    dVydy_true = (y_true[:,1:-1,2:,1:-1,1] - y_true[:,1:-1,:-2,1:-1,1])/2
    dVzdz_true = (y_true[:,1:-1,1:-1,2:,2] - y_true[:,1:-1,1:-1,:-2,2])/2
    div_true = dVxdx_true + dVydy_true + dVzdz_true

    dVxdx_pred = (y_pred[:,2:,1:-1,1:-1,0] - y_pred[:,:-2,1:-1,1:-1,0])/2
    dVydy_pred = (y_pred[:,1:-1,2:,1:-1,1] - y_pred[:,1:-1,:-2,1:-1,1])/2
    dVzdz_pred = (y_pred[:,1:-1,1:-1,2:,2] - y_pred[:,1:-1,1:-1,:-2,2])/2
    div_pred = dVxdx_pred + dVydy_pred + dVzdz_pred

    div_loss = tf.math.reduce_mean( tf.math.abs(div_true - div_pred) )

    loss = mse + div_loss*scale
    return loss


  def div_loss2(y_true, y_pred):

    scale = 5

    mse = tf.math.reduce_mean( tf.math.square(y_true - y_pred) )

    dVxdx_pred = (y_pred[:,2:,1:-1,1:-1,0] - y_pred[:,:-2,1:-1,1:-1,0])/2
    dVydy_pred = (y_pred[:,1:-1,2:,1:-1,1] - y_pred[:,1:-1,:-2,1:-1,1])/2
    dVzdz_pred = (y_pred[:,1:-1,1:-1,2:,2] - y_pred[:,1:-1,1:-1,:-2,2])/2
    div_pred = dVxdx_pred + dVydy_pred + dVzdz_pred

    div_loss = tf.math.reduce_mean( tf.math.abs(div_pred) )

    loss = mse + div_loss*scale

    return loss
```

Input Parameter

```
# Model A-Z1 / A-X1
#domainType_ls4 = ['SCC', 'BCC', 'FCC', 'TCC']
#domainRange_ls4 = [ range(1,11), range(1,11), range(1,11), range(1,11)]
```

```
    # Model B-Z1 / B-X1
    domainType_ls4 = ['SCC', 'BCC', 'FCC', 'TCC', 'CaF2', 'CaTiO3', 'CsCl', 'NaCl']
    domainRange_ls4 = [ range(1,11), range(1,11), range(1,11), range(1,11), range(1,11),
range(1,11), range(1,11), range(1,11), range(1,11)]

    # Model C-Z1
    #domainType_ls4 = ['SCC', 'BCC', 'FCC', 'TCC']
    #domainRange_ls4 = [ range(2,11),  [2,3,4,7,8,10], [2,3,5,6,7,8,9,10], [1,2,3,6,7,8,9,10] ]

    #domainType_ls2 = ['SCC', 'BCC', 'FCC', 'TCC']
    #domainRange_ls2 = [ [1,2,3,4,5,6,7,9,10], [1,2,3,4,6,7,8,10], range(1,11), [1,2,5,6,7,8,9,10] ]

    # Model D-Z1
    #domainType_ls4 = ['SCC', 'BCC', 'FCC', 'TCC', 'CaF2', 'CaTiO3', 'CsCl', 'NaCl']
    #domainRange_ls4 = [ range(2,11),  [2,3,4,7,8,10], [2,3,5,6,7,8,9,10], [1,2,3,6,7,8,9,10],
[1,2,3,4,5,6,7,8,10], [1,2,3,4,5,6,7,9,10], [1,3,4,5,6,7,8,10], range(1,10)]

    #domainType_ls2 = ['SCC', 'BCC', 'FCC', 'TCC', 'CaF2', 'CaTiO3', 'CsCl', 'NaCl']
    #domainRange_ls2 = [ [1,2,3,4,5,6,7,9,10], [1,2,3,4,6,7,8,10], range(1,11), [1,2,5,6,7,8,9,10],
range(1,11), [1,2,3,4,5,6,7,8,10], range(1,11), range(1,11)]

    # Model E-X1
    #domainType_ls2 = ['SCC', 'BCC', 'FCC', 'TCC']
    #domainRange_ls2 = [ range(2,11),  [2,3,4,7,8,10], [2,3,5,6,7,8,9,10], [1,2,3,6,7,8,9,10] ]

    #domainType_ls1 = ['SCC', 'BCC', 'FCC', 'TCC']
```

```
  #domainRange_ls1 = [ [1,2,3,4,5,6,7,9,10], [1,2,3,4,6,7,8,10], range(1,11), [1,2,5,6,7,8,9,10] ]


  # Model F-X1
  #domainType_ls2 = ['SCC', 'BCC', 'FCC', 'TCC', 'CaF2', 'CaTiO3', 'CsCl', 'NaCl']
  #domainRange_ls2  =  [  range(2,11),    [2,3,4,7,8,10],  [2,3,5,6,7,8,9,10],  [1,2,3,6,7,8,9,10],
[1,2,3,4,5,6,7,8,10], [1,2,3,4,5,6,7,9,10], [1,3,4,5,6,7,8,10], range(1,10)]


  #domainType_ls1 = ['SCC', 'BCC', 'FCC', 'TCC', 'CaF2', 'CaTiO3', 'CsCl', 'NaCl']
  #domainRange_ls1 = [ [1,2,3,4,5,6,7,9,10], [1,2,3,4,6,7,8,10], range(1,11), [1,2,5,6,7,8,9,10],
range(1,11), [1,2,3,4,5,6,7,8,10], range(1,11), range(1,11)]



  # Velocity direction
  channel = 2
  data_size = 128



  # Normalization
  x_min = 0


  if channel < 2:
    x_max = 1.6e-9
  else:
    x_max = 3e-9


  res_rs = 18.75e-6
```

```
    res_uc = 160e-6



    # Augmentation
    rotation = 90
    shift = 1/2

    aug_sample_total = 240

    val_perc = 0.1
    test_perc = 0.1
```

Unit Cell Data

Import data

```
    # length scale X4
    dir_data = 'Data/UnitCell'

    uc_solid_ls4 = np.zeros( (1,data_size, data_size, data_size) )
    uc_vel_ls4 = np.zeros( (1,data_size, data_size, data_size) )

    for i in range(len(domainType_ls4)):
        for j in domainRange_ls4[i]:
```

```
        uc_solid_load                                                                    =
loadmat( f'{dir_data}/{domainType_ls4[i]}_domain{j}_solid.mat' )['solid'].astype('int')
        uc_solid_ls4 = np.append( uc_solid_ls4, np.expand_dims(uc_solid_load, axis=0), axis=0 )
        del uc_solid_load

        uc_vel_load                                                                      =
loadmat( f'{dir_data}/{domainType_ls4[i]}_domain{j}_vfield.mat' )['vfield'].astype('float32')
        uc_vel_ls4 = np.append( uc_vel_ls4, np.expand_dims(uc_vel_load[:,:,:,channel], axis=0),
axis=0 )
        del uc_vel_load

    uc_solid_ls4 = uc_solid_ls4[1:,:,:,:]
    uc_vel_ls4 = uc_vel_ls4[1:,:,:,:]

    print(f'Solid data : {uc_solid_ls4.shape}\nVelocity data : {uc_vel_ls4.shape}')

    # Length scale X2
    dir_data = 'Data/UnitCell'

    uc_solid_ls2 = np.zeros( (1,data_size, data_size, data_size) )
    uc_vel_ls2 = np.zeros( (1,data_size, data_size, data_size) )

    for i in range(len(domainType_ls2)):
      for j in domainRange_ls2[i]:
        uc_solid_load                                                                                =
loadmat( f'{dir_data}/{domainType_ls2[i]}_domain{j}_solid.mat' )['solid'].astype('int')
        uc_solid_ls2 = np.append( uc_solid_ls2, np.expand_dims(uc_solid_load, axis=0), axis=0 )
        del uc_solid_load

        uc_vel_load                                                                                  =
loadmat( f'{dir_data}/{domainType_ls2[i]}_domain{j}_vfield.mat' )['vfield'].astype('float32')
        uc_vel_ls2 = np.append( uc_vel_ls2, np.expand_dims(uc_vel_load[:,:,:,channel], axis=0),
axis=0 )
        del uc_vel_load

  uc_solid_ls2 = uc_solid_ls2[1:,:,:,:]
  uc_vel_ls2 = uc_vel_ls2[1:,:,:,:]

  print(f'Solid data : {uc_solid_ls2.shape}\nVelocity data : {uc_vel_ls2.shape}')

  # Length scale X1
  dir_data = 'Data/UnitCell'

  uc_solid_ls1 = np.zeros( (1,data_size, data_size, data_size) )
  uc_vel_ls1 = np.zeros( (1,data_size, data_size, data_size) )

  for i in range(len(domainType_ls1)):
    for j in domainRange_ls1[i]:
        uc_solid_load                                                              =
loadmat( f'{dir_data}/{domainType_ls1[i]}_domain{j}_solid.mat' )['solid'].astype('int')
        uc_solid_ls1 = np.append( uc_solid_ls1, np.expand_dims(uc_solid_load, axis=0), axis=0 )
        del uc_solid_load

        uc_vel_load                                                                =
loadmat( f'{dir_data}/{domainType_ls1[i]}_domain{j}_vfield.mat' )['vfield'].astype('float32')
        uc_vel_ls1 = np.append( uc_vel_ls1, np.expand_dims(uc_vel_load[:,:,:,channel], axis=0),
axis=0 )
        del uc_vel_load

    uc_solid_ls1 = uc_solid_ls1[1:,:,:,:]
    uc_vel_ls1 = uc_vel_ls1[1:,:,:,:]

    print(f'Solid data : {uc_solid_ls1.shape}\nVelocity data : {uc_vel_ls1.shape}')
```

Normalization

```
    # Raw Velocity normalization to Random Sphere Length Scale
    # Length scale X4

    uc_vel_norm_ls4 = minmax_transform( uc_vel_ls4*(res_rs**2)/( (res_uc/1)**2 )*0.1/1000,
x_min=x_min, x_range=x_max-x_min )
    print(f'Mean velocity Length scale X4 : {uc_vel_norm_ls4.mean()}')

    # Raw Velocity normalization to Random Sphere Length Scale
```

```
  # Length scale X2

  uc_vel_norm_ls2 = minmax_transform( uc_vel_ls2*(res_rs**2)/( (res_uc/2)**2 )*0.1/1000,
x_min=x_min, x_range=x_max-x_min )
  print(f'Mean velocity Length scale X2 : {uc_vel_norm_ls2.mean()}')

  # Raw Velocity normalization to Random Sphere Length Scale
  # Length scale X1

  uc_vel_norm_ls1 = minmax_transform( uc_vel_ls1*(res_rs**2)/( (res_uc/4)**2 )*0.1/1000,
x_min=x_min, x_range=x_max-x_min )
  print(f'Mean velocity Length scale X1 : {uc_vel_norm_ls1.mean()}')
```

Length Scale

```
  # Zoom augmentation to create different length scale
  # Length scale X4

  for i in range(uc_solid_ls4.shape[0]):

    solid_zoom_temp = zoom( uc_solid_ls4[i,:,:,:], (1/4,1/4,1/4) )
    vel_zoom_temp = zoom( uc_vel_norm_ls4[i,:,:,:], (1/4,1/4,1/4) )

    uc_solid_ls4[i,:,:,:] = np.tile( solid_zoom_temp, (4,4,4) )
    uc_vel_norm_ls4[i,:,:,:] = np.tile( vel_zoom_temp, (4,4,4) )
  uc_solid_ls4 = np.around(uc_solid_ls4)

  print(f'Solid data : {uc_solid_ls4.shape}\nVelocity data : {uc_vel_norm_ls4.shape}')

  # Zoom augmentation to create different length scale
  # Length scale X2

  for i in range(uc_solid_ls2.shape[0]):

    solid_zoom_temp = zoom( uc_solid_ls2[i,:,:,:], (1/2, 1/2, 1/2) )
    vel_zoom_temp = zoom( uc_vel_norm_ls2[i,:,:,:], (1/2, 1/2, 1/2) )

    uc_solid_ls2[i,:,:,:] = np.tile( solid_zoom_temp, (2, 2, 2) )
    uc_vel_norm_ls2[i,:,:,:] = np.tile( vel_zoom_temp, (2, 2, 2) )

  uc_solid_ls2 = np.around(uc_solid_ls2)

  print(f'Solid data : {uc_solid_ls2.shape}\nVelocity data : {uc_vel_norm_ls2.shape}')
```

Augmentation

```
  # Combine Length scale data

  uc_solid_data = np.zeros( (1,data_size, data_size, data_size) )
  uc_vel_data = np.zeros( (1,data_size, data_size, data_size) )
```

```
try:
  uc_solid_data = np.append(uc_solid_data, uc_solid_ls4, axis=0)
  uc_vel_data = np.append(uc_vel_data, uc_vel_norm_ls4, axis=0)
except NameError:
  print('No Length scale X4 data')

try:
  uc_solid_data = np.append(uc_solid_data, uc_solid_ls2, axis=0)
  uc_vel_data = np.append(uc_vel_data, uc_vel_norm_ls2, axis=0)
except NameError:
  print('No Length scale X2 data')

try:
  uc_solid_data = np.append(uc_solid_data, uc_solid_ls1, axis=0)
  uc_vel_data = np.append(uc_vel_data, uc_vel_norm_ls1, axis=0)
except NameError:
  print('No Length scale X1 data')

uc_solid_data = uc_solid_data[1:,:,:,:]
uc_vel_data = uc_vel_data[1:,:,:,:]

print(f'\nSolid data : {uc_solid_data.shape}\nVelocity data : {uc_vel_data.shape}')

# AUGMENTATION WITH IMAGE DATA GENERATOR

aug_Generator = ImageDataGenerator(rotation_range = rotation,
                   width_shift_range = shift,
                   height_shift_range = shift,
                   fill_mode='wrap')

solid_iterator = aug_Generator.flow(uc_solid_data, seed=seed, batch_size = uc_solid_data.shape[0])
vel_iterator = aug_Generator.flow(uc_vel_data, seed=seed, batch_size = uc_vel_data.shape[0])

aug_solid = np.zeros( (1, data_size, data_size, data_size) )
aug_vel = np.zeros( (1, data_size, data_size, data_size) )

while aug_solid.shape[0] < aug_sample_total:
  aug_solid = np.append(aug_solid, solid_iterator.next(), axis=0)
  aug_vel = np.append(aug_vel, vel_iterator.next(), axis=0)

aug_solid = aug_solid[1:,:,:,:]
aug_vel = aug_vel[1:,:,:,:]

aug_solid = np.expand_dims(aug_solid, axis=-1)
aug_vel = np.expand_dims(aug_vel, axis=-1)

print(f'Solid data : {aug_solid.shape}\nVelocity data : {aug_vel.shape}')

total_number = aug_solid.shape[0]
train_index, val_test_index = train_test_split(np.arange(total_number), test_size = val_perc
+ test_perc, random_state=seed)
val_index, test_index = train_test_split(val_test_index, test_size =
test_perc/(val_perc+test_perc), random_state=seed)

print(f'Number of training samples : {len(train_index)}')
print(f'Number of validation samples : {len(val_index)}')
print(f'Number of test samples : {len(test_index)}')

train_data_solid = aug_solid[train_index]
val_data_solid = aug_solid[val_index]
test_data_solid = aug_solid[test_index]

train_data_vel = aug_vel[train_index]
val_data_vel = aug_vel[val_index]
test_data_vel = aug_vel[test_index]
```

Data Check

```
  # Raw data - XY Plane
  sample_number = 1

  slice = np.array( [31, 63, 95] )
  fig_title = ['Quarter inlet', 'Midpoint', 'Quarter outlet']

  fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )
  vel_mag = (0,3)

  for j in np.array( [0, 1, 2] ):
    im=axs[0,j].imshow(uc_solid_ls1[sample_number,:,:,slice[j]], clim=(0,1), cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[0,j],fraction=0.05)
    axs[0,j].axis('off')

    im=axs[1,j].imshow(uc_vel_norm_ls1[sample_number,:,:,slice[j]],          clim=vel_mag,
cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[1,j],fraction=0.05)
    axs[1,j].axis('off')

  # Raw data - YZ Plane
  sample_number = 1

  slice = np.array( [0, 63, 95] )
  fig_title = ['Quarter inlet', 'Midpoint', 'Quarter outlet']

  fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )

  vel_mag = (0,3)

  for j in np.array( [0, 1, 2] ):
    im=axs[0,j].imshow(np.squeeze(uc_solid_ls1[sample_number,slice[j],:,:]),     clim=(0,1),
cmap=plt.cm.hot)
```

```
    fig.colorbar(im,ax=axs[0,j],fraction=0.05)
    axs[0,j].axis('off')

    im=axs[1,j].imshow(np.squeeze(uc_vel_norm_ls1[sample_number,slice[j],:,:]),
clim=vel_mag, cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[1,j],fraction=0.05)
    axs[1,j].axis('off')

  # Training data
  sample_number = 21

  slice = np.array( [0, 59, -1] )
  fig_title = ['Inlet', 'Midpoint', 'Outlet']

  fig, axs = plt.subplots( nrows=2, ncols=3,figsize=(10,10) )

  vel_mag = (-1,1)

  for j in np.array( [0, 1, 2] ):
    im=axs[0,j].imshow(test_data_solid[sample_number,:,:,slice[j],0],                    clim=(0,1),
cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[0,j],fraction=0.05)
    axs[0,j].axis('off')
    axs[0,j].set_title('%s Solid' % (fig_title[j]))
    im=axs[1,j].imshow(test_data_vel[sample_number,:,:,slice[j],0],
clim=(vel_mag[0],vel_mag[1]), cmap=plt.cm.hot)
    fig.colorbar(im,ax=axs[1,j],fraction=0.05)
    axs[1,j].axis('off')
    axs[1,j].set_title('%s Velocity' % (fig_title[j]))
```

Darcy Unet Training

Unet Model

```
  from tensorflow.keras.models import *
  from tensorflow.keras.layers import Input, Conv3D, Conv3DTranspose,
BatchNormalization, Activation, Concatenate, Dropout, Multiply

  from numpy import floor, ceil

  def encoder_block(inputs, strides, filter_num, filter_size, activation, momentum, rate):

    path = Conv3D(filter_num, filter_size, padding='same', strides=strides)(inputs)
    path = BatchNormalization(momentum=momentum)(path)
    path = Activation(activation=activation)(path)
    path = Dropout(rate)(path)

    return path
  def decoder_block(inputs, strides, filter_num, filter_size, activation, momentum, rate):

    path = Conv3DTranspose(filter_num, filter_size, padding='same', strides=strides)(inputs)
    path = BatchNormalization(momentum=momentum)(path)
    path = Activation(activation=activation)(path)
    path = Dropout(rate)(path)

    return path

  # U-Net
  def UnetV1(input_shape, filter_num = 5, filter_size = 3, activation = 'selu', momentum = 0.99,
rate = 0.2, output_dim=3):

    inputs = Input(shape = input_shape)

    skip_connection = []

    for i in range(8):

      if(i <= 0):
        path_encoder = encoder_block(inputs, 1, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
      else:

        if(i % 2 == 1):
          path_encoder = encoder_block(path_encoder, 1, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
          skip_connection.append(path_encoder)
        else:
          path_encoder = encoder_block(path_encoder, 2, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)

    for i in reversed(range(6)):

      if(i >= 5):
        path_decoder = decoder_block(path_encoder, 2, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
        path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])
      else:

        if(i % 2 == 0):
          path_decoder = decoder_block(path_decoder, 1, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
        else:
          path_decoder = decoder_block(path_decoder, 2, filter_num*(2**floor(i/2)), filter_size,
activation, momentum, rate)
          path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])



    path = Conv3D(filter_num, filter_size, padding='same')(path_decoder)
    path = Conv3D(output_dim, 1, padding='same')(path)
    return Model(inputs=inputs, outputs=path)



  # U-Net Comprehensive model
  def UnetV2(input_shape, x_model, y_model, z_model, v_scale, filter_num = 5, filter_size = 3,
activation = 'selu', momentum = 0.99, rate = 0.2, output_dim=3):

    inputs = Input(shape = input_shape)

    x_input = tf.math.multiply( x_model(inputs, training=False), v_scale )
    y_input = tf.math.multiply( y_model(inputs, training=False), v_scale )
    z_input = z_model(inputs, training=False)

    path = Concatenate()( [x_input, y_input, z_input] )

    skip_connection = []

    for i in range(8):

      if(i <= 0):
        path_encoder = encoder_block(path, 1, filter_num, filter_size, activation, momentum,
rate)
      else:

        if(i % 2 == 1):
            path_encoder = encoder_block(path_encoder, 1, filter_num, filter_size, activation, 
momentum, rate)
            skip_connection.append(path_encoder)
          else:
            path_encoder = encoder_block(path_encoder, 2, filter_num, filter_size, activation, 
momentum, rate)

    for i in reversed(range(6)):

      if(i >= 5):
        path_decoder = decoder_block(path_encoder, 2, filter_num, filter_size, activation, 
momentum, rate)
        path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])
      else:

        if(i % 2 == 0):
          path_decoder = decoder_block(path_decoder, 1, filter_num, filter_size, activation, 
momentum, rate)
        else:
          path_decoder = decoder_block(path_decoder, 2, filter_num, filter_size, activation, 
momentum, rate)
          path_decoder = Concatenate()([ path_decoder, skip_connection[int(floor(i/2))] ])


    path = Conv3D(filter_num, filter_size, padding='same')(path_decoder)
    path = Conv3D(output_dim, 1, padding='same')(path)
    return Model(inputs=inputs, outputs=path)


  # 1 layer convolution
  def UnetV3(input_shape, x_model, y_model, z_model, v_scale, filter_num = 5, filter_size = 3,
activation = 'selu', momentum = 0.99, rate = 0.2, output_dim=3):

    inputs = Input(shape = input_shape)


    x_input = tf.math.multiply( x_model(inputs, training=False), v_scale )
    y_input = tf.math.multiply( y_model(inputs, training=False), v_scale )
    z_input = z_model(inputs, training=False)


    path = Concatenate()( [x_input, y_input, z_input] )


    for i in range(3):
      path = encoder_block(path, 1, filter_num, filter_size, activation, momentum, rate)


    path = Conv3D(output_dim, 1, padding='same')(path)


    return Model(inputs=inputs, outputs=path)


  # Multi-scale Model
  def UnetV4(input_shape, x_model, y_model, z_model, v_scale, filter_num = 5, filter_size =
(3,5,7), num_layers=3, activation = 'selu', momentum = 0.99, rate = 0.2, output_dim=3):
    inputs = Input(shape = input_shape)

    x_input = tf.math.multiply( x_model(inputs, training=False), v_scale )
    y_input = tf.math.multiply( y_model(inputs, training=False), v_scale )
    z_input = z_model(inputs, training=False)

    path = Concatenate()( [x_input, y_input, z_input] )

    path_small = path
    path_medium = path
    path_large = path

    for i in range(num_layers):
      path_small = encoder_block(path_small, 1, filter_num, filter_size[0], activation, momentum, rate)
      path_medium = encoder_block(path_medium, 1, filter_num, filter_size[1], activation, momentum, rate)
      path_large = encoder_block(path_large, 1, filter_num, filter_size[2], activation, momentum, rate)

    path = Concatenate()( [path_small, path_medium, path_large] )
    path = Conv3D(output_dim, 1, padding='same')(path)

    return Model(inputs=inputs, outputs=path)
```

Model Training

```
version = 'C-Z2'


model_name = f'UnetUC_{version}'

dir_save   = f'UnitCell SubModel/{model_name}'


try:

  os.mkdir(dir_save)

except OSError:

  print('Path already exists')


filter_size = 4

num_filters = 10


learning_rate = 0.00007

batch_size = 4


momentum = 0.99

rate = 0.05


epochs = 1000

patience_training = 50


activation = 'relu'


metrics=['MAE', 'MSE']
```

```
    optimizer = tf.keras.optimizers.Adam(learning_rate = learning_rate)

    model = UnetV1( input_shape = (data_size, data_size, data_size, 1), filter_num = num_filters,
filter_size = filter_size, activation = activation,
                momentum = momentum, rate = rate, output_dim=1)

    model.compile(     loss     =     tf.keras.losses.mean_squared_error,     optimizer=optimizer,
metrics=metrics[:] )

    nan_terminate = tf.keras.callbacks.TerminateOnNaN()
    early_stop    = tf.keras.callbacks.EarlyStopping(monitor ='val_loss', min_delta = 0,
                                      patience = patience_training,
                                      verbose = True, mode = 'auto', baseline = None)

    csv_logger                                                                          =
tf.keras.callbacks.CSVLogger("{}/training_log_{}.csv".format(dir_save,model_name))

    checkpoint = ModelCheckpoint('{}/{}.ckpt'.format(dir_save,model_name),
                        monitor = 'val_loss',
                        verbose = 1,
                        save_best_only = True,
                        mode = 'min', save_weights_only = False)

    callbacks_list = [nan_terminate, early_stop, checkpoint, csv_logger]

    print('Model ready')
```

```
model.summary(line_length = 250)

from keras.utils.vis_utils import plot_model

imgtype = ('LR','TB')
for i in imgtype:
  plot_model(model,                 to_file='UnitCell                 SubModel/Model
Image/{}_{}.jpg'.format(model_name, i), rankdir = i, show_layer_names = False)

from timeit import default_timer as timer
from datetime import timedelta

start = timer()

model.fit( x = train_data_solid, y = train_data_vel,
        epochs = epochs, batch_size = batch_size,
        validation_data = (val_data_solid, val_data_vel),
        validation_freq = 1,
        verbose = 1,
        callbacks = callbacks_list)

end = timer()
print('Elapsed time for training : {}'.format(timedelta(seconds=end-start)))

best_model = tf.keras.models.load_model( '{}/{}.ckpt'.format(dir_save, model_name) )
```

```
    best_model.save(dir_save)

    print('Training complete : {}'.format(version))

    best_model = tf.keras.models.load_model( '{}/{}.ckpt'.format(dir_save, model_name) )
    best_model.save(dir_save)
```

Model Evaluation

```
    version = 'A-Z1'

    eval_model_name = f'UnetUC_{version}'
    eval_model = tf.keras.models.load_model( f'UnitCell SubModel/{eval_model_name}/' )
    #eval_model = tf.keras.models.load_model( f'UnitCell SubModel/{eval_model_name}/',
custom_objects = {'div_loss2': div_loss2} )

    metric_evaluate = eval_model.evaluate(test_data_solid, test_data_vel)

    vz_test_pred = np.float32( eval_model.predict( x=[test_data_solid] ) )

    print(vz_test_pred.shape)

    # Overall Permeability
    channel = 2

    perm_true = test_data_vel[:,:,:,:,channel].mean(axis=(1,2,3))
```

```
perm_pred = vz_test_pred[:,:,:,:,channel].mean(axis=(1,2,3))

perm_error = abs( perm_true - perm_pred )/abs(perm_true)*100

print( 'Overall permeability error : {:.3f}\n'.format(perm_error.mean()) )

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(np.arange(perm_error.shape[0]), perm_error)
plt.title('Permeability Error')
plt.show()

# Velocity PDF

channel = 2

min = 0.001
max = 120
numbin = 400

vel_true_flat = np.abs( (test_data_vel[:,:,:,:,channel]).flatten() )
vel_pred_flat = np.abs( (vz_test_pred[:,:,:,:,channel]).flatten() )

vel_true_flat_nz = vel_true_flat[ (vel_true_flat > min) & (vel_pred_flat > min) ]
vel_pred_flat_nz = vel_pred_flat[ (vel_true_flat > min) & (vel_pred_flat > min) ]
vel_true_flat_nz_mean = vel_true_flat_nz.mean()
```

```
    #log_bins = np.logspace(np.log10(min), np.log10(max), numbin)
    log_bins = np.linspace(min, max, numbin)

    true_hist = plt.hist( vel_true_flat_nz, bins=log_bins, density=True, histtype='step', 
label='true' )
    pred_hist = plt.hist( vel_pred_flat_nz, bins=log_bins, density=True, histtype='step', 
label='pred' )
    plt.axvline(vel_true_flat_nz.mean(), color='k', linestyle='dashed', linewidth=1)
    plt.legend()

    #plt.xscale('log')
    plt.yscale('log')

    plt.show()

    th_value = 0.1
    threshold = vel_true_flat_nz_mean*th_value
    vel_true_flat_nz_th = vel_true_flat_nz[ vel_true_flat_nz > threshold]
    vel_pred_flat_nz_th = vel_pred_flat_nz[ vel_true_flat_nz > threshold]

    mape_th = np.divide( np.abs(vel_true_flat_nz_th - vel_pred_flat_nz_th), 
vel_true_flat_nz_th).mean()*100

    print(mape_th)
```

```
th_value = 1
threshold = vel_true_flat_nz_mean*th_value
vel_true_flat_nz_th = vel_true_flat_nz[ vel_true_flat_nz > threshold]
vel_pred_flat_nz_th = vel_pred_flat_nz[ vel_true_flat_nz > threshold]

mape_th = np.divide( np.abs(vel_true_flat_nz_th - vel_pred_flat_nz_th), vel_true_flat_nz_th).mean()*100

print(mape_th)

channel = 2

min = 0.001
max = 20

vel_true_indflat = np.abs( test_data_vel[:,:,:,:,channel].reshape(len(test_index), 128**3) )
vel_pred_indflat = np.abs( vz_test_pred[:,:,:,:,channel].reshape(len(test_index), 128**3) )

vel_true_indflat_nz = np.ma.masked_less(vel_true_indflat, min)
vel_pred_indflat_nz = np.ma.masked_less(vel_pred_indflat, min)
vel_true_indflat_nz_mean = vel_true_indflat_nz.mean()
vel_true_indflat_nz_indmean = vel_true_indflat_nz.mean(axis=1)
print(vel_true_indflat_nz_indmean)

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
```

```
    ax.bar(np.arange(vel_true_indflat_nz_indmean.shape[0]), vel_true_indflat_nz_indmean)
    plt.title('Permeability Error')
    plt.show()

    th_value = 0.01
    threshold = vel_true_indflat_nz_mean*th_value
    print(threshold)

    vel_true_indflat_nz_th = np.ma.masked_less(vel_true_indflat_nz, threshold)
    vel_pred_indflat_nz_th = np.ma.masked_less(vel_pred_indflat_nz, threshold)
    mape_th = np.divide( np.abs(vel_true_indflat_nz_th - vel_pred_indflat_nz_th),
vel_true_indflat_nz_th).mean(axis=1)*100

    print(mape_th)

    test_sample = 0
    channel = 2

    print( 'Maximum Sample Number : {}'.format(test_data_solid.shape[0] - 1) )

    mySolid = test_data_solid[test_sample,:,:,:,0]
    mySolid_mask = np.ma.masked_less(mySolid, 0.5)

    myVel_true = test_data_vel[test_sample,:,:,:,channel]
    myVel_true = np.ma.array( myVel_true, mask=np.ma.getmask(mySolid_mask) )
```

```
myVel_pred = vz_test_pred[test_sample,:,:,:,channel]
myVel_pred = np.ma.array( myVel_pred, mask=np.ma.getmask(mySolid_mask) )


test_slice = np.array( [0, data_size//2, 72] )
fig_title = ['Inlet', 'Midpoint', 'Outlet']

fig, axs = plt.subplots( nrows=3, ncols=3,figsize=(20,20) )

vel_range = (0,30)

for j in range(3):

  im=axs[j,0].imshow(mySolid[:,:,test_slice[j]], clim=(0,1), cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,0],fraction=0.05)
  axs[j,0].axis('off')
  #axs[j,0].set_title('{} Solid'.format(fig_title[j]))

  im=axs[j,1].imshow(myVel_true[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,1],fraction=0.05)
  axs[j,1].axis('off')
  #axs[j,1].set_title('{} Simulation'.format(fig_title[j]))

  im=axs[j,2].imshow(myVel_pred[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,2],fraction=0.05)
  axs[j,2].axis('off')
```

```
    #axs[j,2].set_title('{} Prediction'.format(fig_title[j]))
```

PolySphere Data

```
  dir_data = 'Data/UnitCell/PolySphere validation'

  domainRange = [1,2,3,4,5,6,7,8,9]

  rs_solid = np.zeros( (1, data_size, data_size, data_size) )
  rs_vel = np.zeros( (1, data_size, data_size, data_size) )

  for i in domainRange:

    rs_solid_load = loadmat( '{}/PolySphere_domain{}_solid.mat'.format(dir_data,
i) )['solid'].astype('int')
    rs_solid = np.append( rs_solid, np.expand_dims(rs_solid_load, axis=0), axis=0 )
    del rs_solid_load

    rs_vel_load = loadmat( '{}/PolySphere_domain{}_vfield.mat'.format(dir_data,
i) )['vfield'].astype('float32')
    rs_vel = np.append( rs_vel, np.expand_dims(rs_vel_load[:,:,:,channel], axis=0), axis=0 )
    del rs_vel_load

  rs_solid = rs_solid[1:,:,:,:]
  rs_vel = rs_vel[1:,:,:,:]
```

```
rs_solid = np.expand_dims(rs_solid,axis=-1)
rs_vel = np.expand_dims(rs_vel,axis=-1)

print(f'Solid data : {rs_solid.shape}\nVelocity data : {rs_vel.shape}')

# Raw Velocity Normalization

rs_vel_norm = rs_vel*0.333/9270

rs_vel_norm = minmax_transform(rs_vel_norm, x_min=x_min, x_range=x_max-x_min)

print(f'Mean velocity Random Sohere : {rs_vel_norm.mean()}')

# Predict velocity
version = 'D-Z1'

eval_model_name = f'UnetUC_{version}'
eval_model = tf.keras.models.load_model( f'UnitCell SubModel/{eval_model_name}' )

metric_evaluate = eval_model.evaluate(rs_solid, rs_vel_norm)

vz_test_pred = np.float32( eval_model.predict( x=[rs_solid] ) )

print(vz_test_pred.shape)

# Predict Y velocity using X model
```

```
version = 'F-X1'

rs_solid_xtoy = np.rot90(rs_solid, axes=(1,2))

eval_model_name = f'UnetUC_{version}'
eval_model = tf.keras.models.load_model( f'UnitCell SubModel/{eval_model_name}' )

vz_test_pred_xtoy = np.float32( eval_model.predict( x=[rs_solid_xtoy] ) )

vz_test_pred = np.rot90(vz_test_pred_xtoy*-1, 3, axes=(1,2))

print(vz_test_pred.shape)

# Overall Permeability

perm_true = rs_vel_norm.mean(axis=(1,2,3,4))
perm_pred = vz_test_pred.mean(axis=(1,2,3,4))

perm_error = abs( perm_true - perm_pred )/abs(perm_true)*100

print( 'Overall permeability error : {:.3f}\n'.format(perm_error.mean()) )

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(np.arange(perm_error.shape[0]), perm_error)
```

```
plt.title('Permeability Error')
plt.show()

print(perm_true)
print(perm_pred)

# 3 Channel velocity for STAFE

dir_data = 'Data/UnitCell/PolySphere validation'

domainRange = [1,2,3,4,5,6,7,8,9]

rs_vel_stafe = np.zeros( (1,data_size,data_size,data_size, 3) )

for i in domainRange:
  rs_vel_load = loadmat( '{}/PolySphere_domain{}_vfield.mat'.format(dir_data,
i) )['vfield'].astype('float32')
  rs_vel_stafe = np.append( rs_vel_stafe, np.expand_dims(rs_vel_load[:,:,:,:], axis=0), axis=0)
  del rs_vel_load

rs_vel_stafe = rs_vel_stafe[1:,:,:,:]

print(f'Velocity data : {rs_vel_stafe.shape}')

rs_vel_norm_stafe_x = minmax_transform( rs_vel_stafe[:,:,:,:,0], x_min=0, x_range=1.6e-
9 )*0.333/9270
```

```
  rs_vel_norm_stafe_y = minmax_transform( rs_vel_stafe[:,:,:,:,1], x_min=0, x_range=1.6e-9 )*0.333/9270
  rs_vel_norm_stafe_z = minmax_transform( rs_vel_stafe[:,:,:,:,2], x_min=0, x_range=3e-9 )*0.333/9270

  version_span = 'F-X1'
  version_stream = 'D-Z1'

  eval_model_stream = tf.keras.models.load_model( f'UnitCell SubModel/UnetUC_{version_stream}')
  eval_model_span= tf.keras.models.load_model( f'UnitCell SubModel/UnetUC_{version_span}')

  vx_pred = np.float32( eval_model_span.predict( x=[rs_solid] ) )
  vy_pred = np.rot90( -1*np.float32( eval_model_span.predict( x=np.rot90(rs_solid, axes=(1,2)) ) ), 3, axes=(1,2) )
  vz_pred = np.float32( eval_model_stream.predict( x=[rs_solid] ) )

  # STAFE

  for i in range(rs_vel_stafe.shape[0]):
   q_x_true = rs_vel_norm_stafe_x[i,:,:,:].mean(axis=(1,2))
   q_x_pred = vx_pred[i,:,:,:,0].mean(axis=(1,2))

   q_y_true = rs_vel_norm_stafe_y[i,:,:,:].mean(axis=(0,2))
   q_y_pred = vy_pred[i,:,:,:,0].mean(axis=(0,2))
  q_z_true = rs_vel_norm_stafe_z[i,:,:,:].mean(axis=(0,1))
  q_z_pred = vz_pred[i,:,:,:,0].mean(axis=(0,1))

  stafe_denom = np.sum( np.abs(q_z_true) )

  stafe_x = np.sum( np.abs( q_x_true - q_x_pred ) )/stafe_denom
  stafe_y = np.sum( np.abs( q_y_true - q_y_pred ) )/stafe_denom
  stafe_z = np.sum( np.abs( q_z_true - q_z_pred ) )/stafe_denom

  stafe = stafe_x + stafe_y + stafe_z

  print(stafe)

# Velocity PDF
vel_true_flat = np.abs( rs_vel_norm.flatten() )
vel_pred_flat = np.abs( vz_test_pred.flatten() )

min = 0.001
max = 8
numbin = 500

vel_true_flat_nz = vel_true_flat[ (vel_true_flat > min) & (vel_pred_flat > min) ]
vel_pred_flat_nz = vel_pred_flat[ (vel_true_flat > min) & (vel_pred_flat > min) ]
```

```
    lin_bins = np.linspace(min, max, numbin)
    true_hist = plt.hist( vel_true_flat_nz, bins=lin_bins, density=True, histtype='step', 
label='true' )
    pred_hist = plt.hist( vel_pred_flat_nz, bins=lin_bins, density=True, histtype='step', 
label='pred' )

    plt.legend()
    plt.xlabel('Velocity')
    plt.ylabel('Probability')
    plt.show()

    # Save pdf

    scipy.io.savemat( f'Prediction Results/UnitCell 
Model/{version}_vel{channel}_RS_true.mat', {'data': vel_true_flat_nz} )
    scipy.io.savemat( f'Prediction Results/UnitCell 
Model/{version}_vel{channel}_RS_pred.mat', {'data': vel_pred_flat_nz} )

    vel_true_mape = np.abs( sic_vel_norm.flatten() )
    vel_pred_mape = np.abs( vz_test_pred.flatten() )

    vel_true_mape_nz = vel_true_mape[ (vel_true_mape > min) & (vel_pred_mape > min) ]
    vel_pred_mape_nz = vel_pred_mape[ (vel_true_mape > min) & (vel_pred_mape > min) ]

    vel_true_mape_nz_mean = vel_true_mape_nz.mean()
    th1 = vel_true_mape_nz_mean*0.1
    vel_true_mape_nz_th1 = vel_true_mape_nz[ vel_true_mape_nz > th1 ]
    vel_pred_mape_nz_th1 = vel_pred_mape_nz[ vel_true_mape_nz > th1 ]

    mape_th1 = np.divide( np.abs(vel_true_mape_nz_th1 - vel_pred_mape_nz_th1),
vel_true_mape_nz_th1).mean()*100

    th2 = vel_true_mape_nz_mean
    vel_true_mape_nz_th2 = vel_true_mape_nz[ vel_true_mape_nz > th2 ]
    vel_pred_mape_nz_th2 = vel_pred_mape_nz[ vel_true_mape_nz > th2 ]

    mape_th2 = np.divide( np.abs(vel_true_mape_nz_th2 - vel_pred_mape_nz_th2),
vel_true_mape_nz_th2).mean()*100

    print( f'Threshold 10 % : {mape_th1}\nThreshold 100 % : {mape_th2}' )

    test_sample = 6
    channel = 0

    print( 'Maximum Sample Number : {}'.format(rs_solid.shape[0] - 1) )

    mySolid = rs_solid[test_sample,:,:,:,0]
    mySolid_mask = np.ma.masked_less(mySolid,1)

    myVel_true = rs_vel_minmax[test_sample,:,:,:,channel]
    myVel_true = np.ma.array( myVel_true, mask=np.ma.getmask(mySolid_mask) )
```

```
myVel_pred = vz_test_pred[test_sample,:,:,:,channel]
myVel_pred = np.ma.array( myVel_pred, mask=np.ma.getmask(mySolid_mask) )



test_slice = np.array( [data_size//4, data_size//2, 3*data_size//4] )
fig_title = ['Quarterpoint', 'Halfpoint', 'Three Quarterpoint']

fig, axs = plt.subplots( nrows=3, ncols=3,figsize=(20,20) )

vel_range = (0,8)

for j in range(3):

  im=axs[j,0].imshow(mySolid[:,:,test_slice[j]], clim=(0,1), cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,0],fraction=0.05)
  axs[j,0].axis('off')
  #axs[j,0].set_title('{} Solid'.format(fig_title[j]))

  im=axs[j,1].imshow(myVel_true[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,1],fraction=0.05)
  axs[j,1].axis('off')
  #axs[j,1].set_title('{} Simulation'.format(fig_title[j]))

  im=axs[j,2].imshow(myVel_pred[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,2],fraction=0.05)
```

```
    axs[j,2].axis('off')
    #axs[j,2].set_title('{} Prediction'.format(fig_title[j]))
```

SiC Foam Data

```
  dir_data = 'Data/UnitCell/SiCFoam validation'

  densityRange = [45, 65, 80]
  domainRange = [1, 2]

  sic_solid = np.zeros( (1,data_size,data_size,data_size) )
  sic_vel = np.zeros( (1,data_size,data_size,data_size) )

  for i in densityRange:
    for j in domainRange:

      sic_solid_load = loadmat( f"{dir_data}/{i}PPI_domain{j}_solid.mat" )['solid'].astype('int')
      sic_solid = np.append( sic_solid, np.expand_dims(sic_solid_load, axis=0), axis=0 )
      del sic_solid_load

      sic_vel_load                                                                    =
loadmat( f"{dir_data}/{i}PPI_domain{j}_vfield.mat" )['vfield'].astype('float32')
      sic_vel = np.append( sic_vel, np.expand_dims(sic_vel_load[:,:,:,channel], axis=0), axis=0)
      del sic_vel_load

  sic_solid = sic_solid[1:,:,:,:]
```

```
sic_vel = sic_vel[1:,:,:,:]

print(f'Solid data : {sic_solid.shape}\nVelocity data : {sic_vel.shape}')

Solid data : (6, 128, 128, 128)
Velocity data : (6, 128, 128, 128)

# Raw Velocity Normalization
res_sic = 26.708e-6
res_rs = 20e-6

sic_vel_norm_high = sic_vel[:4,:,:,:]*(res_rs**2)/(res_sic**2)*0.333/9270
sic_vel_norm_low = sic_vel[4:,:,:,:]*(res_rs**2)/((res_sic/2)**2)*0.333/9270

sic_vel_norm = minmax_transform( np.append(sic_vel_norm_high, sic_vel_norm_low,
axis=0), x_min=x_min, x_range=x_max-x_min )

sic_solid = np.expand_dims(sic_solid,axis=-1)
sic_vel_norm = np.expand_dims(sic_vel_norm,axis=-1)

print(f'Mean velocity SiC Foam : {sic_vel_norm.mean()}')

Mean velocity SiC Foam : 1.1074862054299068

# Predict velocity
version = 'D-Z1'
```

```
eval_model_name = f'UnetUC_{version}'
eval_model = tf.keras.models.load_model( f'UnitCell SubModel/{eval_model_name}' )

metric_evaluate = eval_model.evaluate(sic_solid, sic_vel_norm)

vz_test_pred = np.float32( eval_model.predict( x=[sic_solid] ) )

print(vz_test_pred.shape)

1/1 [==============================] - 17s 17s/step - loss: 0.3111 - MAE: 0.3536 - MSE: 0.3111
1/1 [==============================] - 1s 1s/step
(6, 128, 128, 128, 1)

# Predict Y velocity using X model

version = 'F-X1'

sic_solid_xtoy = np.rot90(sic_solid, axes=(1,2))

eval_model_name = f'UnetUC_{version}'
eval_model = tf.keras.models.load_model( f'UnitCell SubModel/{eval_model_name}' )

vz_test_pred_xtoy = np.float32( eval_model.predict( x=[sic_solid_xtoy] ) )
```

```
vz_test_pred = np.rot90(vz_test_pred_xtoy*-1, 3, axes=(1,2))

print(vz_test_pred.shape)

# MASK PREDICTIONS

sic_solid_mask = np.ma.masked_less(sic_solid, 1)

sic_vel_mask = np.ma.array( sic_vel_norm, mask=np.ma.getmask(sic_solid_mask) )
vz_test_pred_mask = np.ma.array( vz_test_pred, mask=np.ma.getmask(sic_solid_mask) )

print(f'Solid size : {sic_solid_mask.shape}')

# Overall Permeability
#perm_true = sic_vel_mask.mean(axis=(1,2,3,4))
#perm_pred = vz_test_pred_mask.mean(axis=(1,2,3,4))

perm_true = sic_vel_norm.mean(axis=(1,2,3,4))
perm_pred = vz_test_pred.mean(axis=(1,2,3,4))

perm_error = abs( perm_true - perm_pred )/abs(perm_true)*100

print( f'Overall permeability error : {perm_error.mean():.3f}' )

sic_perm = [8.15, 11.48, 6.6, 7.76, 9.14, 10.43]
plt.scatter(perm_true, perm_error)
```

```
  plt.xlabel('Normalized Permeability')
  plt.ylabel('Permebaility Error (%)')
  plt.show()

  print(perm_true)
  print(perm_pred)

  # 3 Channel velocity for STAFE

  dir_data = 'Data/UnitCell/SiCFoam validation'

  densityRange = [45, 65, 80]
  domainRange = [1, 2]

  sic_vel_stafe = np.zeros( (1,data_size,data_size,data_size, 3) )

  for i in densityRange:
    for j in domainRange:

      sic_vel_load = 
loadmat( f"{dir_data}/{i}PPI_domain{j}_vfield.mat" )['vfield'].astype('float32')
      sic_vel_stafe = np.append( sic_vel_stafe, np.expand_dims(sic_vel_load[:,:,:,:], axis=0), 
axis=0)
      del sic_vel_load

  sic_vel_stafe = sic_vel_stafe[1:,:,:,:]
```

```
print(f'Velocity data : {sic_vel_stafe.shape}')

res_sic = 26.708e-6
res_rs = 20e-6

sic_vel_stafe_high = sic_vel_stafe[:4,:,:,:]*(res_rs**2)/(res_sic**2)*0.333/9270
sic_vel_stafe_low = sic_vel_stafe[4:,:,:,:]*(res_rs**2)/((res_sic/2)**2)*0.333/9270

sic_vel_stafe = np.append(sic_vel_stafe_high, sic_vel_stafe_low, axis=0)

sic_vel_stafe_x = sic_vel_stafe[:,:,:,:,0]
sic_vel_stafe_y = sic_vel_stafe[:,:,:,:,1]
sic_vel_stafe_z = sic_vel_stafe[:,:,:,:,2]

sic_vel_norm_stafe_x = minmax_transform( sic_vel_stafe_x, x_min=0, x_range=1.6e-9 )
sic_vel_norm_stafe_y = minmax_transform( sic_vel_stafe_y, x_min=0, x_range=1.6e-9 )
sic_vel_norm_stafe_z = minmax_transform( sic_vel_stafe_z, x_min=0, x_range=3e-9 )

version_span = 'B-X1'
version_stream = 'B-Z1'

eval_model_stream = tf.keras.models.load_model( f'UnitCell SubModel/UnetUC_{version_stream}')
eval_model_span= tf.keras.models.load_model( f'UnitCell SubModel/UnetUC_{version_span}')
```

```
  vx_pred = np.float32( eval_model_span.predict( x=[sic_solid] ) )
  vy_pred = np.rot90( -1*np.float32( eval_model_span.predict( x=np.rot90(sic_solid,
axes=(1,2)) ) ), 3, axes=(1,2) )
  vz_pred = np.float32( eval_model_stream.predict( x=[sic_solid] ) )

  # STAFE

  for i in range(sic_vel_stafe.shape[0]):
   q_x_true = sic_vel_norm_stafe_x[i,:,:,:].mean(axis=(1,2))
   q_x_pred = vx_pred[i,:,:,:,0].mean(axis=(1,2))

   q_y_true = sic_vel_norm_stafe_y[i,:,:,:].mean(axis=(0,2))
   q_y_pred = vy_pred[i,:,:,:,0].mean(axis=(0,2))

   q_z_true = sic_vel_norm_stafe_z[i,:,:,:].mean(axis=(0,1))
   q_z_pred = vz_pred[i,:,:,:,0].mean(axis=(0,1))

   stafe_denom = np.sum( np.abs(q_z_true) )

   stafe_x = np.sum( np.abs( q_x_true - q_x_pred ) )/stafe_denom
   stafe_y = np.sum( np.abs( q_y_true - q_y_pred ) )/stafe_denom
   stafe_z = np.sum( np.abs( q_z_true - q_z_pred ) )/stafe_denom

   stafe = stafe_x + stafe_y + stafe_z
```

```
    print(stafe)

   # Velocity PDF
   vel_true_flat = np.abs( sic_vel_norm.flatten() )
   vel_pred_flat = np.abs( vz_test_pred.flatten() )

   min = 0.001
   max = 8
   numbin = 500

   vel_true_flat_nz = vel_true_flat[ (vel_true_flat > min) & (vel_pred_flat > min) ]
   vel_pred_flat_nz = vel_pred_flat[ (vel_true_flat > min) & (vel_pred_flat > min) ]



   lin_bins = np.linspace(min, max, numbin)
   true_hist = plt.hist( vel_true_flat_nz, bins=lin_bins, density=True, histtype='step',
label='true' )
   pred_hist = plt.hist( vel_pred_flat_nz, bins=lin_bins, density=True, histtype='step',
label='pred' )

   plt.legend()
   plt.xlabel('Velocity')
   plt.ylabel('Probability')
   plt.show()

   ks_stats_total = ks_2samp(true_hist[0], pred_hist[0])
```

```
    print( f'KS Statistic : {ks_stats_total[0]}\nP value : {ks_stats_total[1]}')

    # Save pdf

    scipy.io.savemat( f'Prediction Results/UnitCell Model/{version}_vel{channel}_true.mat',
{'data': vel_true_flat_nz} )
    scipy.io.savemat( f'Prediction Results/UnitCell Model/{version}_vel{channel}_pred.mat',
{'data': vel_pred_flat_nz} )

    vel_true_mape = np.abs( sic_vel_norm.flatten() )
    vel_pred_mape = np.abs( vz_test_pred.flatten() )

    vel_true_mape_nz = vel_true_mape[ (vel_true_mape > min) & (vel_pred_mape > min) ]
    vel_pred_mape_nz = vel_pred_mape[ (vel_true_mape > min) & (vel_pred_mape > min) ]

    vel_true_mape_nz_mean = vel_true_mape_nz.mean()

    th1 = vel_true_mape_nz_mean*0.1
    vel_true_mape_nz_th1 = vel_true_mape_nz[ vel_true_mape_nz > th1 ]
    vel_pred_mape_nz_th1 = vel_pred_mape_nz[ vel_true_mape_nz > th1 ]

    mape_th1 = np.divide( np.abs(vel_true_mape_nz_th1 - vel_pred_mape_nz_th1),
vel_true_mape_nz_th1).mean()*100

    th2 = vel_true_mape_nz_mean
    vel_true_mape_nz_th2 = vel_true_mape_nz[ vel_true_mape_nz > th2 ]
    vel_pred_mape_nz_th2 = vel_pred_mape_nz[ vel_true_mape_nz > th2 ]

    mape_th2 = np.divide( np.abs(vel_true_mape_nz_th2 - vel_pred_mape_nz_th2),
vel_true_mape_nz_th2).mean()*100

    print( f'Threshold 10 % : {mape_th1}\nThreshold 100 % : {mape_th2}' )

    # Quantitative Visualization of velocity profile (REVISION)
    test_sample = 4

    mySolid = sic_solid[test_sample,:,:,:,0]
    mySolid_mask = np.ma.masked_less(mySolid,1)

    myVel_true = sic_vel_norm[test_sample,:,:,:,0]
    myVel_true = np.ma.array( myVel_true, mask=np.ma.getmask(mySolid_mask) )

    myVel_pred = vz_test_pred[test_sample,:,:,:,0]
    myVel_pred = np.ma.array( myVel_pred, mask=np.ma.getmask(mySolid_mask) )

    threshold = myVel_true.mean()

    #myVel_error = np.abs(np.divide((myVel_true - myVel_pred),myVel_true))*100
    myVel_error = np.abs(myVel_true - myVel_pred)

    test_slice = np.array( [data_size//4, data_size//2, 3*data_size//4] )
```

```
 vel_range = (0,6)

 fig, axs = plt.subplots( nrows=3, ncols=3,figsize=(20,20) )
 for j in range(3):

  im=axs[j,0].imshow(myVel_true[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,0],fraction=0.05)
  axs[j,0].axis('off')

  im=axs[j,1].imshow(myVel_pred[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,1],fraction=0.05)
  axs[j,1].axis('off')

  im=axs[j,2].imshow(myVel_error[:,:,test_slice[j]], clim=(0,3), cmap=plt.cm.hot)
  fig.colorbar(im,ax=axs[j,2],fraction=0.05)
  axs[j,2].axis('off')

[]

 test_sample = 4

 mySolid = sic_solid[test_sample,:,:,:,0]
 mySolid_mask = np.ma.masked_less(mySolid,1)

 myVel_true = sic_vel_norm[test_sample,:,:,:,0]
myVel_true = np.ma.array( myVel_true, mask=np.ma.getmask(mySolid_mask) )


myVel_pred = vz_test_pred[test_sample,:,:,:,0]

myVel_pred = np.ma.array( myVel_pred, mask=np.ma.getmask(mySolid_mask) )


test_slice = np.array( [data_size//4, data_size//2, 3*data_size//4] )

fig_title = ['Quarterpoint', 'Halfpoint', 'Three Quarterpoint']


fig, axs = plt.subplots( nrows=3, ncols=3,figsize=(20,20) )


vel_range = (-4,4)


for j in range(3):


  im=axs[j,0].imshow(mySolid[:,:,test_slice[j]], clim=(0,1), cmap=plt.cm.hot)

  fig.colorbar(im,ax=axs[j,0],fraction=0.05)

  axs[j,0].axis('off')

  axs[j,0].set_title(f'{fig_title[j]} Solid')


  im=axs[j,1].imshow(myVel_true[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)

  fig.colorbar(im,ax=axs[j,1],fraction=0.05)

  axs[j,1].axis('off')

  axs[j,1].set_title(f'{fig_title[j]} Simulation')


  im=axs[j,2].imshow(myVel_pred[:,:,test_slice[j]], clim=vel_range, cmap=plt.cm.hot)

  fig.colorbar(im,ax=axs[j,2],fraction=0.05)
```

```
    axs[j,2].axis('off')
    axs[j,2].set_title(f'{fig_title[j]} Prediction')

  # SAVE X VELOCITY

  case_name = (45, 65, 80)

  for i in range(6):
      save_name = f'{case_name[i//2]}PPI_domain{i%2+1}_vfieldx'
      scipy.io.savemat(    f'Prediction    Results/UnitCell    Model/{version}/{save_name}.mat',
{'vfield_ML': np.squeeze(vz_test_pred[i,:,:,:,0])} )

  # SAVE Y VELOCITY

  case_name = (45, 65, 80)

  for i in range(6):
      save_name = f'{case_name[i//2]}PPI_domain{i%2+1}_vfieldy'
      scipy.io.savemat(    f'Prediction    Results/UnitCell    Model/{version}/{save_name}.mat',
{'vfield_ML': np.squeeze(vz_test_pred[i,:,:,:,0])} )

  # SAVE Z VELOCITY

  case_name = (45, 65, 80)

  for i in range(6):
      save_name = f'{case_name[i//2]}PPI_domain{i%2+1}_vfield'
      scipy.io.savemat( f'Prediction Results/UnitCell Model/{version}/{save_name}.mat', {'vfield_ML': np.squeeze(vz_test_pred[i,:,:,:,0])} )
```

Data Analysis

```
    # Training data set permeability

    train_data_vel_perm = train_data_vel.mean(axis=(1,2,3,4))

    train_bins = np.linspace(0, 8, 9)
    train_hist = plt.hist( train_data_vel_perm, bins=train_bins, density=False, histtype='step', label='true' )

    plt.legend()
    plt.xlabel('Permeability')
    plt.ylabel('Probability')
    plt.show()

    print(train_hist[0])
    print(train_hist[1])

    # Save histogram
    save_name = 'Model A training data'
    scipy.io.savemat( f'Prediction Results/UnitCell Model/{save_name}.mat', {'data': train_data_vel_perm} )
```

```
# Training Data PDF


# Velocity PDF
vel_train_flat = np.abs( train_data_vel.flatten() )


min = 0.001
max = 10
numbin = 1000


vel_train_flat_nz = vel_train_flat[ (vel_train_flat > min) ]



lin_bins = np.linspace(min, max, numbin)
train_hist = plt.hist( vel_train_flat_nz, bins=lin_bins, density=True, histtype='step', label='pred' )


plt.legend()
plt.xlabel('Velocity')
plt.ylabel('Probability')
plt.show()


print(train_hist[0])


# Save histogram
save_name = 'ModelD_training'
```

```
    scipy.io.savemat( f'Prediction Results/UnitCell Model/{save_name}_bin_vel{channel}.mat',
{'data': train_hist[1]} )
    scipy.io.savemat( f'Prediction Results/UnitCell Model/{save_name}_pdf_vel{channel}.mat',
{'data': train_hist[0]} )

    # SiC PDF
    vel_true_flat = np.abs( sic_vel_norm.flatten() )

    min = 0.001
    max = 8
    numbin = 1000

    vel_true_flat_nz = vel_true_flat[ (vel_true_flat > min) ]

    lin_bins = np.linspace(min, max, numbin)
    true_hist = plt.hist( vel_true_flat_nz, bins=lin_bins, density=True, histtype='step',
label='true' )

    plt.legend()
    plt.xlabel('Velocity')
    plt.ylabel('Probability')
    plt.show()

    # Save histogram
    save_name = 'SiCFoam'
```

```
  scipy.io.savemat( f'Prediction Results/UnitCell Model/{save_name}_bin_vel{channel}.mat',
{'data': true_hist[1]} )
  scipy.io.savemat( f'Prediction Results/UnitCell Model/{save_name}_pdf_vel{channel}.mat',
{'data': true_hist[0]} )

  !pip install porespy

  import porespy as ps

  uc_solid_mis = np.zeros_like(uc_solid_data)

  for i in range(uc_solid_data.shape[0]):
    print(i)
    uc_solid_mis[i,:,:,:] = ps.filters.local_thickness(uc_solid_data[i,:,:,:])

  scipy.io.savemat( f'Prediction Results/UnitCell_modelB_mis.mat', {'data': uc_solid_mis} )

  sic_solid_mis = np.zeros_like(sic_solid)

  for i in range(sic_solid.shape[0]):
    sic_solid_mis[i,:,:,:] = ps.filters.local_thickness(sic_solid[i,:,:,:])

  scipy.io.savemat( f'Prediction Results/sic_mis.mat', {'data': sic_solid_mis} )
```

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and others. Tensorflow: A system for large-scale machine learning. 2015.

[2] A. Aitkenhead. Converting a 3D logical array into an STL surface mesh. *MATLAB Central File Exchange*, May 2010.

[3] J. D. Anderson Jr. *Hypersonic and High-Temperature Gas Dynamics, Second Edition.* American Institute of Aeronautics and Astronautics, 2006.

[4] R. L. Baker. Graphite Sublimation Chemistry Nonequilibrium Effects. *AIAA Journal*, 15(10):1391–1397, Oct. 1977.

[5] J. Bear. *Dynamics of Fluids in Porous Media.* Courier Corporation, Feb. 2013.

[6] A. Bejan. *Convection Heat Transfer.* John Wiley & Sons, Mar. 2013.

[7] H. A. Bethe and M. C. Adams. A Theory for the Ablation of Glassy Materials. *Journal of the Aerospace Sciences*, 26(6):321–328, 1959.

[8] Z. Bhumgara. Polyhipe foam materials as filtration media. *Filtration & Separation*, 32(3):245–251, Mar. 1995.

[9] K. Boomsma, D. Poulikakos, and F. Zwick. Metal foams as compact high performance heat exchangers. *Mechanics of Materials*, 35(12):1161–1176, Dec. 2003.

[10] D. W. Breck and D. W. Breck. *Zeolite molecular sieves: structure, chemistry, and use.* John Wiley & Sons, 1973.

[11] S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis. Physics-informed neural networks (PINNs) for fluid mechanics: a review. *Acta Mechanica Sinica*, Jan. 2022.

[12] S. Cai, Z. Wang, S. Wang, P. Perdikaris, and G. E. Karniadakis. Physics-Informed Neural Networks for Heat Transfer Problems. *Journal of Heat Transfer*, 143(6), Apr. 2021.

[13] A. F. Charwat. The effect of surface-evaporation kinetics on sublimation near the leading edge. *International Journal of Heat and Mass Transfer*, 8(3):383–394, Mar. 1965.

[14] S. Chen, Q. Liu, G. He, Y. Zhou, M. Hanif, X. Peng, S. Wang, and H. Hou. Reticulated carbon foam derived from a sponge-like natural product as a high-performance anode in microbial fuel cells. *Journal of Materials Chemistry*, 22(35):18609–18613, 2012.

[15] Y.-K. Chen, E. C. Stern, and P. Agrawal. Thermal Ablation Simulations of Quartz Materials. *Journal of Spacecraft and Rockets*, 56(3), May 2019.

[16] C. Deng, Zhiwenand He, Y. Liu, and K. C. Kim. Super-resolution reconstruction of turbulent velocity fields using a generative adversarial network-based artificial intelligence framework. *Physics of Fluids*, 31(12):125111, Dec. 2019.

[17] A. Diani, K. K. Bodla, L. Rossetto, and S. V. Garimella. Numerical Analysis of Air Flow through Metal Foams. *Energy Procedia*, 45:645–652, Jan. 2014.

[18] B. Dias, A. Turchi, E. C. Stern, and T. E. Magin. A model for meteoroid ablation including melting and vaporization. *Icarus*, 345:113710, July 2020.

[19] R. Dotts, H. Battley, J. Hughes, and W. Neuenschwander. Space Shuttle Orbiter - Reusable surface insulation subsystem thermal performance. *20th Aerospace Sciences Meeting AIAA*, Jan. 1982.

[20] A. Faghri. *Heat Pipe Science And Technology.* Global Digital Press, Mar. 1995.

[21] J. A. Fay and F. R. Riddell. Theory of Stagnation Point Heat Transfer in Dissociated Air. *Journal of the Aerospace Sciences*, 25(2):73–85, 1958.

[22] K. Fukami, K. Fukagata, and K. Taira. Super-resolution reconstruction of turbulent flows with machine learning. *Journal of Fluid Mechanics*, 870:106–120, July 2019.

[23] P. Furler, J. Scheffe, D. Marxer, M. Gorbar, A. Bonk, U. Vogt, and A. Steinfeld. Thermochemical CO2 splitting via redox cycling of ceria reticulated foam structures with dual-scale porosities. *Physical Chemistry Chemical Physics*, 16(22):10503–10511, 2014.

[24] L. J. Gibson and M. F. Ashby. *Cellular Solids: Structure and Properties.* Cambridge Solid State Science Series. Cambridge University Press, Cambridge, 2 edition, 1997.

[25] J. T. Gostick, Z. A. Khan, T. G. Tranter, M. D. r. Kok, M. Agnaou, M. Sadeghi, and R. Jervis. PoreSpy: A Python Toolkit for Quantitative Analysis of Porous Media Images. *Journal of Open Source Software*, 4(37):1296, May 2019.

[26] T. B. Group. Unit Cells.

[27] R. N. Gupta, K. P. Lee, and C. D. Scott. Aerothermal Study of Mars Pathfinder Aeroshell. *Journal of Spacecraft and Rockets*, 33(1):61–69, Jan. 1996.

[28] M. S. Hassouna and A. A. Farag. MultiStencils Fast Marching Methods: A Highly Accurate Solution to the Eikonal Equation on Cartesian Domains. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(9):1563–1574, Sept. 2007.

[29] W. M. Haynes. *CRC handbook of chemistry and physics*. CRC press, 2016.

[30] F. He and J. Wang. Numerical investigation on critical heat flux and coolant volume required for transpiration cooling with phase change. *Energy Conversion and Management*, 80:591–597, Apr. 2014.

[31] Q. He and A. M. Tartakovsky. Physics-Informed Neural Network Method for Forward and Backward Advection-Dispersion Equations. *Water Resources Research*, 57(7), 2021.

[32] T. A. Heppenheimer. *Facing the Heat Barrier: A History of Hypersonics*. Courier Dover Publications, Sept. 2018.

[33] H. Hidalgo. Ablation of Glassy Material Around Blunt Bodies of Revolution. *ARS Journal*, 30(9):806–814, 1960.

[34] J. Hilsenrath and M. Klein. Tables of Thermodynamic Properties of Air in Chemical Equilibrium Including Second Viral Corrections from 1500K to 15000K. Technical report, National Bureau of Standards Gaithersburg MD, Mar. 1965.

[35] J. O. Hirschfelder, C. F. Curtiss, and R. B. Bird. *The Molecular Theory of Gases and Liquids*. Wiley, 1954.

[36] J. Hong and J. Liu. Rapid estimation of permeability from digital rock using 3D convolutional neural network. *Computational Geosciences*, 24(4):1523–1539, Aug. 2020.

[37] G. Huang, Z. Liao, R. Xu, Y. Zhu, and P.-X. Jiang. Self-pumping transpiration cooling with a protective porous armor. *Applied Thermal Engineering*, 164:114485, Jan. 2020.

[38] G. Huang, Y. Zhu, Z. Liao, and P.-X. Jiang. Experimental investigation of self-pumping internal transpiration cooling. *International Journal of Heat and Mass Transfer*, 123:514–522, Aug. 2018.

[39] X. Huang, X. Chen, A. Li, D. Atinafu, H. Gao, W. Dong, and G. Wang. Shape-stabilized phase change materials based on porous supports for thermal energy storage applications. *Chemical Engineering Journal*, 356:641–661, Jan. 2019.

[40] H. Huisseune, S. De Schampheleire, B. Ameel, and M. De Paepe. Comparison of metal foam heat exchangers to a finned heat exchanger for low Reynolds number applications. *International Journal of Heat and Mass Transfer*, 89:1–9, Oct. 2015.

[41] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167*, Mar. 2015. Preprint at https://arxiv.org/abs/1502.03167.

[42] X. Jin, S. Cai, H. Li, and G. E. Karniadakis. NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 426:109951, Feb. 2021.

[43] X. Jin, P. Cheng, W.-L. Chen, and H. Li. Prediction model of velocity field around circular cylinder over various Reynolds numbers by fusion convolutional neural networks based on pressure on the cylinder. *Physics of Fluids*, 30(4):047105, Apr. 2018.

[44] S. Kamrava, P. Tahmasebi, and M. Sahimi. Linking Morphology of Porous Media to Their Macroscopic Permeability by Deep Learning. *Transport in Porous Media*, 131(2):427–448, Jan. 2020.

[45] A. Kashefi and T. Mukerji. Point-cloud deep learning of porous media for permeability prediction. *Physics of Fluids*, 33(9):097109, Sept. 2021.

[46] A. Kashefi and T. Mukerji. Physics-informed PointNet: A deep learning solver for steady-state incompressible flows and thermal fields on multiple sets of irregular geometries. *Journal of Computational Physics*, 468:111510, Nov. 2022.

[47] A. Kashefi and T. Mukerji. Prediction of Fluid Flow in Porous Media by Sparse Observations and Physics-Informed PointNet, Aug. 2022.

[48] A. Kashefi, D. Rempe, and L. J. Guibas. A point-cloud deep learning framework for prediction of fluid flow fields on irregular geometries. *Physics of Fluids*, 33(2):027104, Feb. 2021.

[49] M. Kaviany. *Principles of Heat Transfer in Porous Media*. Springer Science & Business Media, Dec. 2012.

[50] R. Keys. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(6):1153–1160, 1981.

[51] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization, Jan. 2017. Preprint at http://arxiv.org/abs/1412.6980.

[52] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-Normalizing Neural Networks. In *Advances in Neural Information Processing Systems*, 2017. Presented as part of NeurlPS Proceedings.

[53] D.-J. Kroon. Accurate Fast Marching. *MATLAB Central File Exchange*, Jan. 2011.

[54] T. Kubota. Ablation With Ice Model at M = 5.8. *ARS Journal*, 30(12):1164–1169, 1960.

[55] M. Lacroix, P. Nguyen, D. Schweich, C. Pham Huu, S. Savin-Poncet, and D. Edouard. Pressure drop measurements and modeling on SiC foams. *Chemical Engineering Science*, 62(12):3259–3267, June 2007.

[56] R. E. Larson and J. J. L. Higdon. A periodic grain consolidation model of porous media. *Physics of Fluids A: Fluid Dynamics*, 1(1):38–46, Jan. 1989.

[57] L. Lees. Laminar Heat Transfer Over Blunt-Nosed Bodies at Hypersonic Flight Speeds. *Journal of Jet Propulsion*, 26(4):259–269, 1956.

[58] G. Lepage, F. O. Albernaz, G. Perrier, and G. Merlin. Characterization of a microbial fuel cell with reticulated carbon foam electrodes. *Bioresource Technology*, 124:199–207, Nov. 2012.

[59] M. Liu and P. Mostaghimi. Characterisation of reactive transport in pore-scale correlated porous media. *Chemical Engineering Science*, 173:121–130, Dec. 2017.

[60] D. Malakoff. Mach 12 by 2012? *Science*, 300(5621):888–889, May 2003.

[61] Z. Mao, A. D. Jagtap, and G. E. Karniadakis. Physics-informed neural networks for high-speed flows. *Computer Methods in Applied Mechanics and Engineering*, 360:112789, Mar. 2020.

[62] E. Marshall. NASA and Military Press for a Spaceplane: The aerospace plane, a possible alternative to the shuttle, is to be designed with military goals in mind more than commercial travel. *Science*, 231(4734):105–107, Jan. 1986.

[63] A. T. Mohan, N. Lubbers, D. Livescu, and M. Chertkov. Embedding Hard Physical Constraints in Neural Network Coarse-Graining of 3D Turbulence, Feb. 2020.

[64] C. H. Mortensen. *Effects of Thermochemical Nonequilibrium on Hypersonic Boundary-Layer Instability in the Presence of Surface Ablation or Isolated Two-Dimensional Roughness*. PhD thesis, University of California Los Angeles, United States – California, 2015.

[65] C. H. Mortensen and X. Zhong. Simulation of Second-Mode Instability in a Real-Gas Hypersonic Flow with Graphite Ablation. *AIAA Journal*, 52(8):1632–1652, 2014.

[66] C. H. Mortensen and X. Zhong. Real-Gas and Surface-Ablation Effects on Hypersonic Boundary-Layer Instability over a Blunt Cone. *AIAA Journal*, 54(3):980–998, 2016.

[67] D. A. Nield and A. Bejan. *Convection in Porous Media: 2nd edition*. Springer Science & Business Media, 1999.

[68] L. C. d. S. M. Ozelim and A. L. B. Cavalcante. Representative Elementary Volume Determination for Permeability and Porosity Using Numerical Three-Dimensional Experiments in Microtomography Data. *International Journal of Geomechanics*, 18(2):04017154, Feb. 2018.

[69] F. Panerai, J. C. Ferguson, J. Lachaud, A. Martin, M. J. Gasch, and N. N. Mansour. Micro-tomography based analysis of thermal conductivity, diffusivity and oxidation behavior of rigid and flexible fibrous insulators. *International Journal of Heat and Mass Transfer*, 108:801–811, May 2017.

[70] G. Plesch, M. Vargová, U. F. Vogt, M. Gorbár, and K. Jesenák. Zr doped anatase supported reticulated ceramic foams for photocatalytic water purification. *Materials Research Bulletin*, 47(7):1680–1686, July 2012.

[71] G. Popkin. U.S. defenses look to thwart missiles early. *Science*, 363(6425):327–328, Jan. 2019.

[72] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. pages 652–660, 2017.

[73] P. Raghunandan, J. B. Haskins, G. E. Palmer, B. K. Bessire, and E. C. Stern. Material Response Modeling of Melt Flow-Vapor Ablation for Iron. *AIAA Journal*, 60(4):2028–2038, 2022.

[74] P. Ranut, E. Nobile, and L. Mancini. High resolution X-ray microtomography-based CFD simulation for the characterization of flow permeability and effective thermal conductivity of aluminum metal foams. *Experimental Thermal and Fluid Science*, 67:30–36, Oct. 2015.

[75] T. Reimer, M. Kuhn, A. Gülhan, B. Esser, M. Sippel, and A. van Foreest. Transpiration Cooling Tests of Porous CMC in Hypersonic Flow. *17th AIAA International Space Planes and Hypersonic Systems and Technologies Conference*, Apr. 2011.

[76] M. D. Ribeiro, A. Rehman, S. Ahmed, and A. Dengel. DeepCFD: Efficient Steady-State Laminar Flow Approximation with Deep Convolutional Neural Networks, Nov. 2021.

[77] J. T. Richardson, Y. Peng, and D. Remue. Properties of ceramic foam catalyst supports: pressure drop. *Applied Catalysis A: General*, 204(1):19–32, Nov. 2000.

[78] J. F. Roberts, C. H. Lewis, and M. Reed. Ideal Gas Spherically Blunted Cone Flow Field Solutions At Hypersonic Conditions. Technical Report, ARO Inc., Aug. 1966.

[79] L. Roberts. Stagnation-Point Shielding by Melting and Vaporization. Technical Report NASA-TR-R-10, Jan. 1959.

[80] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, 2015.

[81] Z. Rosenberg. Hypersonic X-51 programme ends in success, May 2013. Library Catalog: www.flightglobal.com.

[82] M. Röding, Z. Ma, and S. Torquato. Predicting permeability via statistical learning on higher-order microstructural information. *Scientific Reports*, 10(1):15239, Sept. 2020.

[83] J. E. Santos, D. Xu, H. Jo, C. J. Landry, M. Prodanović, and M. J. Pyrcz. PoreFlow-Net: A 3D convolutional neural network to predict fluid flow through porous media. *Advances in Water Resources*, 138:103539, Apr. 2020.

[84] J. E. Santos, Y. Yin, H. Jo, W. Pan, Q. Kang, H. S. Viswanathan, M. Prodanović, M. J. Pyrcz, and N. Lubbers. Computationally Efficient Multiscale Neural Networks Applied to Fluid Flow in Complex 3D Porous Media. *Transport in Porous Media*, 140(1):241–272, Oct. 2021.

[85] S. M. Scala and G. L. Vidale. Vaporization processes in the hypersonic laminar boundary layer. *International Journal of Heat and Mass Transfer*, 1(1):4–22, June 1960.

[86] L. Shen, J. Wang, W. Dong, J. Pu, J. Peng, D. Qu, and L. Chen. An experimental investigation on transpiration cooling with phase change under supersonic condition. *Applied Thermal Engineering*, 105:549–556, July 2016.

[87] R. Stone. Need for speed. *Science*, 367(6474):134–138, Jan. 2020.

[88] H. Su, F. He, J. Wang, Xiaoguang Luo, and B. Ai. Numerical investigation on the effects of porous cone parameters on liquid transpiration cooling performance. *International Journal of Thermal Sciences*, 161:106743, Mar. 2021.

[89] H. Su, J. Wang, F. He, L. Chen, and B. Ai. Numerical investigation on transpiration cooling with coolant phase change under hypersonic conditions. *International Journal of Heat and Mass Transfer*, 129:480–490, Feb. 2019.

[90] O. Sudakov, E. Burnaev, and D. Koroteev. Driving digital rock towards machine learning: Predicting permeability with gradient boosting and deep neural networks. *Computers & Geosciences*, 127:91–98, June 2019.

[91] S. Takbiri, M. Kazemi, A. Takbiri-Borujeni, and J. McIlvain. A deep learning approach to predicting permeability of porous media. *Journal of Petroleum Science and Engineering*, 211:110069, Apr. 2022.

[92] H. Tran, C. Johnson, D. Rasky, F. Hui, M.-T. Hsu, and Y. Chen. Phenolic Impregnated Carbon Ablators (PICA) for Discovery class missions. *31st Thermophysics Conference AIAA*, June 1996.

[93] S. Trimble. Hypersonic airliner ”may not be as hard as people think”: Boeing CTO, Aug. 2018.

[94] B. True, W. Johnson, and S. Chen. Reducing phosphorus discharge from flow-through aquaculture: III: assessing high-rate filtration media for effluent solids and phosphorus removal. *Aquacultural Engineering*, 32(1):161–170, Dec. 2004.

[95] E. R. Van Driest. The Problem of Aerodynamic Heating. *Inst. of the Aeronautical Sciences*, 1956.

[96] A. van Foreest, M. Sippel, A. Gülhan, B. Esser, B. A. C. Ambrosius, and K. Sudmeijer. Transpiration Cooling Using Liquid Water. *Journal of Thermophysics and Heat Transfer*, 23(4):693–702, 2009.

[97] A. H. Van Tuyl. The Use of Rational Approximations in the Calculation of Flows with Detached Shocks. *Journal of the Aerospace Sciences*, 27(7):559–560, July 1960.

[98] G. Walther, B. Klöden, T. Büttner, T. Weißgärber, B. Kieback, A. Böhm, D. Naumann, S. Saberi, and L. Timberg. A New Class of High Temperature and Corrosion Resistant Nickel-Based Open-Cell Foams. *Advanced Engineering Materials*, 10(9):803–811, 2008.

[99] J. Wang, L. Zhao, X. Wang, J. Ma, and J. Lin. An experimental investigation on transpiration cooling of wedge shaped nose cone with liquid coolant. *International Journal of Heat and Mass Transfer*, 75:442–449, Aug. 2014.

[100] K. Wang, Y. Chen, M. Mehana, N. Lubbers, K. C. Bennett, Q. Kang, H. S. Viswanathan, and T. C. Germann. A physics-informed and hierarchically regularized data-driven model for predicting fluid flow through porous media. *Journal of Computational Physics*, 443:110526, Oct. 2021.

[101] Y. D. Wang, M. J. Blunt, R. T. Armstrong, and P. Mostaghimi. Deep learning in pore scale imaging and modeling. *Earth-Science Reviews*, 215:103555, Apr. 2021.

[102] Y. D. Wang, T. Chung, R. T. Armstrong, and P. Mostaghimi. ML-LBM: Predicting and Accelerating Steady State Flow Simulation in Porous Media with Convolutional Neural Networks. *Transport in Porous Media*, 138(1):49–75, May 2021.

[103] Y. D. Wang, T. Chung, A. Rabbani, R. T. Armstrong, and P. Mostaghimi. Fast direct flow simulation in porous media by coupling with pore network and Laplace models. *Advances in Water Resources*, 150:103883, Apr. 2021.

[104] K. C. Weston. The Stagnation-point Boundary Layer with Suction and Injection in Equilibrium Dissociating Air. Technical Report NASA-TN-D-3889, Dec. 1968.

[105] S. Whitaker. Flow in porous media I: A theoretical derivation of Darcy's law. *Transport in Porous Media*, 1(1):3–25, Mar. 1986.

[106] C. R. Wilke. A Viscosity Equation for Gas Mixtures. *The Journal of Chemical Physics*, 18(4):517–519, Apr. 1950.

[107] C. E. Wittliff and J. T. Curtis. Normal shock wave parameters in equilibrium air. rep. no. cal-111 (contract no. af 33 (616)-6579), cornell aeronaut. lab. *Inc., Nov*, 1961.

[108] D. Wu, D. Qu, W. Jiang, G. Chen, L. An, C. Zhuang, and Z. Sun. Self-floating nanostructured Ni–NiO x /Ni foam for solar thermal water evaporation. *Journal of Materials Chemistry A*, 7(14):8485–8490, 2019.

[109] J. Wu, X. Yin, and H. Xiao. Seeing permeability from images: fast prediction with convolutional neural networks. *Science Bulletin*, 63(18):1215–1222, Sept. 2018.

[110] J.-W. Wu, W.-F. Sung, and H.-S. Chu. Thermal conductivity of polyurethane foams. *International Journal of Heat and Mass Transfer*, 42(12):2211–2217, June 1999.

[111] W. Xu, H. Zhang, Z. Yang, and J. Zhang. Numerical investigation on the flow characteristics and permeability of three-dimensional reticulated foam materials. *Chemical Engineering Journal*, 140(1):562–569, July 2008.

[112] M. Zafari, M. Panjepour, M. Davazdah Emami, and M. Meratian. Microtomography-based numerical simulation of fluid flow and heat transfer in open cell metal foams. *Applied Thermal Engineering*, 80:347–354, Apr. 2015.

[113] E. Zermatten, S. Haussener, M. Schneebeli, and A. Steinfeld. Tomography-based determination of permeability and Dupuit–Forchheimer coefficient of characteristic snow samples. *Journal of Glaciology*, 57(205):811–816, 2011.

[114] D. Zhang, R. Zhang, S. Chen, and W. E. Soll. Pore scale study of flow in porous media: Scale dependency, REV, and statistical REV. *Geophysical Research Letters*, 27(8):1195–1198, 2000.

[115] H. Zhang, H. Yu, X. Yuan, H. Xu, M. Micheal, J. Zhang, H. Shu, G. Wang, and H. Wu. Permeability prediction of low-resolution porous media images using autoencoder-based convolutional neural network. *Journal of Petroleum Science and Engineering*, 208:109589, Jan. 2022.

[116] S. Zhang, X. Li, J. Zuo, J. Qin, K. Cheng, Y. Feng, and W. Bao. Research progress on active thermal protection for hypersonic vehicles. *Progress in Aerospace Sciences*, 119:100646, Nov. 2020.

[117] W. Zhao, D. M. France, W. Yu, T. Kim, and D. Singh. Phase change material with graphite foam for applications in high-temperature latent heat storage systems of concentrated solar power plants. *Renewable Energy*, 69:134–146, Sept. 2014.

[118] X. Zheng, X. Gao, Z. Huang, Z. Li, Y. Fang, and Z. Zhang. Form-stable paraffin/graphene aerogel/copper foam composite phase change material for solar energy conversion and storage. *Solar Energy Materials and Solar Cells*, 226:111083, July 2021.

[119] X.-H. Zhou, J. McClure, C. Chen, and H. Xiao. Neural Network Based Pore Flow Field Prediction in Porous Media Using Super Resolution. *arXiv:2109.09863 [physics]*, Sept. 2021. Preprint at http://arxiv.org/abs/2109.09863.

[120] Y. Zhu, W. Peng, R. Xu, and P. Jiang. Review on active thermal protection and its heat transfer for airbreathing hypersonic vehicles. *Chinese Journal of Aeronautics*, 31(10):1929–1953, Oct. 2018.

[121] A. A. Zick and G. M. Homsy. Stokes flow through periodic arrays of spheres. *Journal of Fluid Mechanics*, 115:13–26, Feb. 1982.