

MEC ENG 151B/250B: Convective Transport and Computational Methods

Final Project: Turbulent Falling Film Heat Transfer
and PINNs for Convective Transport

Professor Van P. Carey
UC Berkeley, Department of Mechanical Engineering
May 14, 2025

Larry Hui¹ and Derek Shah

¹University of California at Berkeley, College of Engineering, Department of Mechanical Engineering. Author to whom any correspondence should be addressed. email: larryhui7@berkeley.edu

Preface

Before presenting our report, we would like to briefly reflect on MEC ENG 151B/250B. The course built on our earlier heat-transfer studies and offered project-based assessments that pushed us to apply theory to practical problems. These projects sharpened our analytical skills and fostered productive collaboration.

We appreciate the support and guidance provided by Professor Carey and Anisa throughout the semester.

Larry Hui

Department of Mechanical Engineering
University of California, Berkeley
larryhui7@berkeley.edu

Derek Shah

Department of Aerospace Engineering
University of California, Berkeley
derekshah@berkeley.edu

Abstract

This project was completed as a requirement for MEC ENG 151B at the University of California at Berkeley. This work presents a two part investigation of convective heat transfer in thin film and internal channel configurations that combines turbulence theory with a modern approach in the physical sciences: physics informed neural networks (PINNs). In the first part, we revisit a high Prandtl number, turbulent falling film that is relevant to molten salt solar receivers. Starting from the time-averaged energy and momentum balances, an eddy diffusivity model is integrated twice to relate wall heat flux, wall shear stress, and film thickness. By eliminating auxiliary variables, we obtain an explicit correlation showing that the non dimensional heat transfer coefficient grows with the Prandtl number raised to the one third power and the Reynolds number raised to $7/24$. The numerical pre-factor is fixed through a mixed analytic numeric evaluation of a single near wall integral, demonstrating that compact, closed form design rules can still emerge from classical analysis even in fully turbulent, high Pr flows.

In the second part a PINN is trained on fully developed laminar flow and heat transfer in rectangular and filleted channels. On a coarse, one millimeter collocation grid, the network reproduces bulk velocity but overestimates wall heat flux by roughly an order of magnitude when compared with a high resolution finite difference model that resolves the thermal boundary layer. Rounding the channel corners shifts heat flux peaks into the fillets, enhancing local cooling but reducing the area averaged flux by about two percent, leaving the sharp cornered rectangle the more effective overall heat sink for a fixed footprint.

Taken together, the study shows that classical boundary layer methods still yield fast, reliable engineering correlations when turbulence dominates and that PINNs offer mesh free flexibility for complex geometries but demand finer or adaptive wall sampling to achieve quantitative accuracy. The hybrid approach lays a foundation for multiscale optimization of large scale solar receivers and microscale cooling passages alike.

Contents

PART 1: High-Prandtl Turbulent Film Analysis	5
1 Introduction and Theoretical Background	6
1.1 Work Division	6
1.2 Task Descriptions	7
2 Results and Discussion	8
2.1 Task I Results and Discussion	8
2.2 Task II Results and Discussion	10
2.3 Task III Results and Discussion	12
2.4 Task IV Results and Discussion	13
2.5 Task V Results and Discussion	14
3 Implementation and Development of Code	16
3.1 Task II	17
4 Conclusion	17
PART 2: PINN Solution for Convective Channel Flow	18
5 Introduction and Theoretical Background	19
5.1 Work Division	21
5.2 Task Descriptions	21
6 Implementation and Development of Code	21
6.1 Custom Loss Function Structure	22
7 Results and Discussion	24
7.1 Task I Results and Discussion	24
7.2 Task II Results and Discussion	26
7.3 Task III Results and Discussion	28
8 Conclusion	30
9 Code Appendix	31
9.1 Part 1 Code	31
9.2 Part 2 PINN Code Rectangular	31
9.3 Part 2 PINN Code Rounded	40
9.4 Part 2 FDM Code (Project 1)	50

List of Figures

1 Turbulent Falling Liquid Film on Inclined Flat Surface	6
2 Constant Wall Temperature T_w with Uniform Heat Input	19
3 Schematic of PINN model for fully developed flow and convection in a channel	20
4 Domain of convective flow and heat transport with specified $w = 0$ and $T = 90^\circ\text{C}$ at the boundary. The coordinates x and y are in millimeters.	23
5 Domain of convective flow and heat transport for a rounded-corner tube with specified w velocity and T at the boundary. The coordinates x and y are in millimeters.	24
6 PINN Velocity Profile for Rectangular Channel	25
7 PINN Temperature Profile for Rectangular Channel	25
8 Parity Plot for Rectangular Channel	26
9 PINN Velocity Profile for Rounded Channel	27
10 PINN Temperature Profile for Rounded Channel	27

11 Parity Plot for Rounded Channel	28
--	----

PART 1: High-Prandtl Turbulent Film Analysis

1 Introduction and Theoretical Background

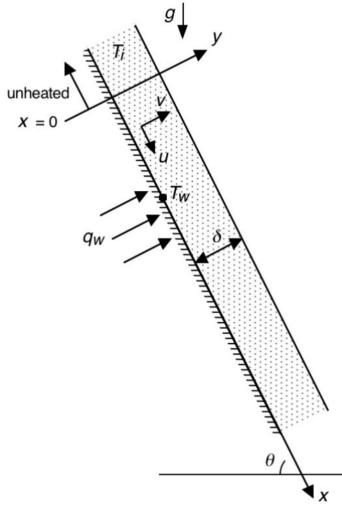


Figure 1: Turbulent Falling Liquid Film on Inclined Flat Surface

In this first part of the project, we will be analyzing heat transfer to a falling liquid film on a slightly inclined flat surface, similar to the circumstances considered in Homework 2. For the analysis here, however, the flow will have two key differences. One is that the flow will be turbulent, and the other is that the Prandtl number will be large compared to 1. These additional circumstances could be encountered in a large-scale falling film solar central receiver using molten salt as a working fluid.

For turbulent flow at high Prandtl number, a steep temperature gradient is expected to exist close to the solid surface. (This near wall region encompasses the viscous sublayer and a portion of the flow just outside it.) The outer region of the film is expected to be virtually isothermal. Heat transfer to the surrounding gas is assumed to be negligible. In the region very near the wall convection of thermal energy is assumed to be negligible compared to molecular diffusion and turbulent transport normal to the wall. The time-averaged boundary layer form of the energy equation in this region is therefore

$$\frac{\partial}{\partial y} \left[\alpha \frac{\partial \bar{T}}{\partial y} - \bar{\nu'} \bar{T}' \right] = 0 \quad (1)$$

Adopting the usual definition of eddy diffusivity, $\epsilon_H = -\bar{\nu'} \bar{T}' / \partial \bar{T} / \partial y$, this becomes

$$\frac{\partial}{\partial y} \left[(\alpha + \epsilon_H) \frac{\partial \bar{T}}{\partial y} \right] = 0 \quad (2)$$

1.1 Work Division

The tasks for this project were distributed among both team members to ensure efficient collaboration and thorough completion of all required components. Each member contributed to different aspects of the project as follows:

1. The derivation of the integral form of the wall-heat flux realtion (i.e. the derivation of equation XX) by twice integrating the near-wall energy equation, adopting the eddy-diffusivity model, and framing the governing integral that links the wall heat flux q_w to $T_w - T_i$ was done by **Larry Hui**. All of this work is outlined in [Section 2.1](#).
2. The similarity transformation of equation XX into the heat-transfer-coefficient correlation, the change of variables to η , analytically splitting of the large-Pr integral was done by **Larry Hui** and numerical

evaluation of the integral I was done by **Derek Shah**. All relevant details are presented in [Section 2.2](#).

3. The work on the neglection of the convection terms, integration of the boundary-layer u -momentum equation across the film, and obtaining the explicit expression for the wall shear stress $\tau_w = f(\rho_l, \rho_g, g, \theta, \delta)$ were done by **Larry Hui** and reviewed by **Derek Shah**. Full documentation is given in [Section 2.3](#).
4. The evaluation of the mass-flow integral Γ with the velocity profile u^+ , and the resulting power-law form of Γ were complete by **Larry Hui** and reviewed by **Derek Shah**. Full documentation is given in [Section 2.4](#).
5. The combination of the results from the two previous tasks to eliminate τ_w and obtain the implicit relation $\Gamma = f(\delta, g, \theta, \rho_l, \rho_g, \nu)$; the further combination with Task II to remove δ and yield the non-dimensional heat-transfer correlation with explicit constants were executed by **Larry Hui** and reviewed by **Derek Shah**. The algebra and numerical evaluation are laid out in [Section 2.5](#).

1.2 Task Descriptions

This project is split into 5 tasks which are briefly described below.

1. **Task I:** This task allows us to establish the integral wall heat flux relation. Firstly, we start with the time-averaged energy equation for turbulent liquid film at a high Prandtl number, this formulation assumes an eddy-diffusivity model that grows rapidly with distance from the wall. We then integrate twice—first in the normal direction of the wall and then in its dimensionless version—we then collapse the differential form into integral form that connects the imposed wall heat flux to the temperature difference between the wall and the nearly-isothermal film core. This step isolates the near-wall physics and defines an integral whose numerical value controls all later heat-transfer predictions.
2. **Task II:** This task builds on the integral form from the previous part; we choose a similarity variable, η , that deals with both the eddy-diffusivity constant and the Prandtl number. The energy-balance integral is then rearranged into a correlation for the local heat-transfer coefficient. Since the integrand goes to 0 as η gets large we can set the upper limit as infinity with some error and then split the integral into 3 sub-domains: we then compute 2 of the integrals analytically and one numerically in MATLAB for which the code is described in [Section 3.1](#). The process yields a single numerical constant that pins down the correlation so we can benchmark with it.
3. **Task III:** We then focus on the momentum transport by simplifying the boundary-layer momentum equation for the film. By neglecting streamwise convection terms we can integrate this equation across the film thickness to get an explicit expression for the wall shear stress in terms of fluid densities, gravity, surface incline angle, and the unknown film thickness. This helps us in later tasks.
4. **Task IV:** With the wall shear now characterized, we are able to figure out the mass-flow rate. A 1/7 power-law universal velocity profile is used to represent turbulent flow. Integrating that profile across the film gets us mass-flow rate per unit width. The final expression is a power law that connects mass-flow, wall shear stress, film thickness, and fluid properties through two numerical constants fixed by the chosen profile.
5. **Task V:** This task removes wall shear from our expression. First, the wall-shear expression from **Task III** is combined with the mass-flow relation from **Task IV**, eliminating shear stress and giving a single equation that predicts film thickness directly from the imposed mass-flow rate and the angle of incline. Next, we also eliminate the thickness by combining the energy result of **Task II** with the momentum and mass-flow relations. The product is a fully non-dimensional heat-transfer correlation that depends only on Reynolds and Prandtl numbers; its coefficient and exponent emerge from the preceding algebra and numerical integration.

2 Results and Discussion

This section aims to give an overview of the quantitative outcomes of the turbulent falling-film analysis and interprets their physical significance. We also present all the mathematical formulation and analytical approaches used throughout this part of the project to model and understand high-Prandtl turbulent film analysis. Solutions to **Tasks I to V** are shown here.

2.1 Task I Results and Discussion

For this task, by integrating Equation (2) twice, changing the second integration variable to y^+ , we obtain a relation of the form

$$\frac{(T_w - T_i)k\sqrt{\tau_w/\rho_l}}{q_w\nu} = \int_0^{\delta_{nw}^+} \frac{dy^+}{1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M(y^+)}{\text{Pr}_t}} \quad (3)$$

where q_w is the wall heat flux and $F_I(\text{Pr}, \nu)$ is a function of Pr and ν . Note hear that δ_{nw} and δ_{nw}^+ correspond to a thermal boundary layer thickness that is smaller than the film thickness delta δ .

Task I. Integral form of the wall-heat-flux relation: Firstly, we start from the time-averaged boundary layer form of the energy equation in the near-wall region.

Solution. We integrate Eq. 2 with respect to y to get the axial heat flux

$$\begin{aligned} \frac{\partial}{\partial y} \left[(\alpha + \epsilon_H) \frac{\partial \bar{T}}{\partial y} \right] &= 0 \\ \int \frac{\partial}{\partial y} \left[(\alpha + \epsilon_H) \frac{\partial \bar{T}}{\partial y} \right] dy &= \int 0 dy \\ (\alpha + \epsilon_H) \frac{\partial \bar{T}}{\partial y} &= C_1 \end{aligned}$$

At the wall, the no slip condition implies that the velocity fluctuations are gone $\Rightarrow \epsilon_H = 0$. Therefore, if we apply Fourier's law of heat conduction to enforce the constant wall heat flux

$$q_w = -k \frac{\partial \bar{T}}{\partial y} \Big|_{y=0} \Rightarrow -\frac{q_w}{k} = \frac{\partial \bar{T}}{\partial y} \Big|_{y=0}$$

Then substituting this into our axial heat flux expression yields

$$\alpha \frac{\partial \bar{T}}{\partial y} \Big|_{y=0} = C_1 \Rightarrow C_1 = -\alpha \frac{q_w}{k}$$

Therefore, the region near the wall obeys

$$(\alpha + \epsilon_H) \frac{\partial \bar{T}}{\partial y} = -\alpha \frac{q_w}{k}$$

We rearrange and integrate again w.r.t. y

$$\begin{aligned} \frac{\partial \bar{T}}{\partial y} &= -\frac{\alpha}{\alpha + \epsilon_H} \cdot \frac{q_w}{k} \\ \int_{T_w}^{T_i} \frac{\partial \bar{T}}{\partial y} dy &= -\frac{\alpha q_w}{k} \int_0^{\delta_{nw}} \frac{1}{\alpha + \epsilon_H} dy \\ T_i - T_w &= -\frac{\alpha q_w}{k} \int_0^{\delta_{nw}} \frac{1}{\alpha + \epsilon_H} dy \\ T_w - T_i &= \frac{\alpha q_w}{k} \int_0^{\delta_{nw}} \frac{1}{\alpha + \epsilon_H} dy \end{aligned}$$

We introduce the following non-dimensional variables

$$u^+ \triangleq \frac{u}{u_\tau} = \frac{u}{\sqrt{\tau_w/\rho_l}} , \quad y \triangleq \frac{yu_\tau}{\nu} = \frac{y\sqrt{\tau_w/\rho_l}}{\nu}$$

Note that the characteristic velocity $u_\tau \triangleq \sqrt{\tau_w/\rho_l}$. By the definition of kinematic viscosity as well as the definition of the Prandtl number, we also have the following expressions

$$\nu \triangleq \mu/\rho_l , \quad \text{Pr} = \frac{\nu}{\alpha} \Rightarrow \alpha = \frac{\nu}{\text{Pr}}$$

Recall that the turbulent Prandtl number is defined as

$$\text{Pr}_t \triangleq \frac{\epsilon_M}{\epsilon_H} \Rightarrow \epsilon_H = \frac{\epsilon_M}{\text{Pr}_t}$$

Combining this all, we can write the denominator o the integrand as

$$\begin{aligned} \alpha + \epsilon_H &= \frac{\nu}{\text{Pr}} + \frac{\epsilon_M}{\text{Pr}_t} \\ &= \frac{\nu}{\text{Pr}} + \frac{\nu}{\text{Pr}} \cdot \frac{\text{Pr}}{\nu} \frac{\epsilon_M}{\text{Pr}_t} \\ &= \frac{\nu}{\text{Pr}_t} \left(1 + \frac{\text{Pr}}{\nu} \frac{\epsilon_M}{\text{Pr}_t} \right) \end{aligned}$$

We set $F_I \triangleq \text{Pr}/\nu$, then simplifying we get

$$\alpha + \epsilon_H = \frac{\nu}{\text{Pr}} \left(1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M}{\text{Pr}_t} \right)$$

Substituting the expression in yields

$$T_w - T_i = \frac{\alpha q_w}{k} \int_0^{\delta_{nw}} \frac{1}{\frac{\nu}{\text{Pr}} \left(1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M}{\text{Pr}_t} \right)} dy$$

From the definition of y^+ , we can write the differential dy in terms of dy^+ by differentiating the non-dimensional variables

$$y^+ \triangleq \frac{yu_\tau}{\nu} \Rightarrow y = \frac{y^+ \nu}{u_\tau} \Rightarrow \frac{dy}{dy^+} = \frac{\nu}{u_\tau} \Rightarrow dy = \frac{\nu}{u_\tau} dy^+$$

The bounds of integration are changed using

$$y^+ \triangleq \frac{yu_\tau}{\nu} \Rightarrow \begin{cases} \text{lower limit: } y = 0 \Rightarrow y^+ = 0 \\ \text{upper limit: } y = \delta_{nw} \Rightarrow \frac{\delta_{nw} u_\tau}{\nu} = \delta_{nw}^+ \end{cases}$$

Substituting this differential in, multiplying both sides by $ku_\tau/q_w\nu$ and changing the bounds and using the relation shown by Tien (1964) and Simonek (1983), we write ϵ_M as a function y^+ and get

$$\begin{aligned} T_w - T_i &= \frac{\alpha q_w}{k} \int_0^{\delta_{nw}^+} \frac{(\psi/u_\tau)}{\frac{\nu}{\text{Pr}} \left(1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M(y^+)}{\text{Pr}_t} \right)} dy^+ \\ &= \frac{\nu q_w}{\text{Pr} k} \int_0^{\delta_{nw}^+} \frac{\text{Pr} dy^+}{\sqrt{\tau_w/\rho_l} \left(1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M(y^+)}{\text{Pr}_t} \right)} \end{aligned}$$

Finally, rearranging yields a relation of the form in Eq. 3

$$\frac{(T_w - T_i)k\sqrt{\tau_w/\rho_l}}{q_w \nu} = \int_0^{\delta_{nw}^+} \frac{dy^+}{1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M(y^+)}{\text{Pr}_t}}$$

where $F_I(\text{Pr}, \nu) = \frac{\text{Pr}}{\nu}$ and $\delta_{nw}^+ = \frac{\delta_{nw} u_\tau}{\nu}$

□

To evaluate the integral on the right side of the above Equation (3), the variation of ϵ_M and turbulent Prandtl number Pr_t must be known. In the relation we obtained, it is clear that for large Pr , the integrand is small except when ϵ_M is sufficiently small that $\text{Pr}\epsilon_M/\nu = \mathcal{O}(1)$. Thus, to evaluate the integral, an accurate relation for ϵ_M is only needed in the region close to the wall (i.e. for small y^+), where ϵ_M is small. Studies by Tien (1964) and Simonek (1983) indicated that ϵ_M/ν is proportional to $(y^+)^3$ for small y^+ . For this analysis, we therefore take

$$\epsilon_M/\nu = \gamma(y^+)^3 \quad (4)$$

Based on the study by Kato, et al. (1968), we will use the recommended value of 5.10×10^{-4} for γ . Here the turbulent Prandtl number is taken to be one: $\text{Pr}_t = 1.0$. Equation (3) can then be written

$$\frac{(T_w - T_i)k\sqrt{\tau_w/\rho_l}}{q_w\nu} = \int_0^{\delta_{nw}^+} \frac{dy^+}{1 + F_l(\text{Pr}, \nu)\epsilon_M} \quad (5)$$

2.2 Task II Results and Discussion

We continue from the first task. We designate the integral on the right side of Equation (3) as I and defining $\eta = \gamma(y^+)^3\text{Pr}$, change the variable of integration in I to η , and recognize the resulting modified form of Equation (5) to a relation for the heat transfer coefficient $h = q_w/(T_w - T_i)$ having the form

$$\frac{h}{k} \left(\frac{\nu}{\sqrt{\tau_w/\rho_l}} \right) = \frac{(\gamma\text{Pr})^{1/3}}{I} \quad (6)$$

where

$$I = \frac{1}{C_{1.2}} \int_0^{\gamma(\delta_{nw}^+)^3\text{Pr}} \frac{\eta^{-2/3}}{1 + \eta/\text{Pr}_t} d\eta \quad (7)$$

where $\text{Pr}_t = 1.0$, and $C_{1.2}$ is an integer numerical constant.

Task II.a. Similarity transformation

Solution. Starting with Eq. 3., we define the right hand side as I

$$I \triangleq \int_0^{\delta_{nw}^+} \frac{dy^+}{1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M(y^+)}{\text{Pr}_t}}$$

We define a change of integration variable η s.t.

$$\eta = \gamma(y^+)^3\text{Pr} \Rightarrow y^+ = (\frac{\eta}{\text{Pr}\gamma})^{1/3} \Rightarrow dy^+ = \frac{1}{3}(\gamma\text{Pr})^{-1/3}\eta^{-2/3}d\eta$$

changing the bounds of integration as well, we get

$$\begin{cases} \text{lower bound: } y^+ = 0 \Rightarrow \eta = 0 \\ \text{upper bound: } y^+ = \delta_{nw}^+ \Rightarrow \eta = (\delta_{nw}^+)^3\text{Pr} \end{cases}$$

Substituting all of these expressions into I as well as taking $\epsilon_M/\nu = \gamma(y^+)^3$ yields

$$\begin{aligned} I &= \int_0^{\gamma(\delta_{nw}^+)^3\text{Pr}} \frac{\frac{1}{3}(\gamma\text{Pr})^{-1/3}\eta^{-2/3}}{1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M}{\text{Pr}_t}} d\eta \\ &= \int_0^{\gamma(\delta_{nw}^+)^3\text{Pr}} \frac{\frac{1}{3}(\gamma\text{Pr})^{-1/3}\eta^{-2/3}}{1 + \frac{\text{Pr}}{\nu} \frac{\epsilon_M}{\text{Pr}_t}} d\eta \\ &= \frac{(\gamma\text{Pr})^{-1/3}}{3} \int_0^{\gamma(\delta_{nw}^+)^3\text{Pr}} \frac{\eta^{-2/3}}{1 + \frac{\text{Pr}}{\nu} \cdot \frac{\gamma(y^+)^3}{\text{Pr}_t}} d\eta \\ &= \frac{(\gamma\text{Pr})^{-1/3}}{3} \int_0^{\gamma(\delta_{nw}^+)^3\text{Pr}} \frac{\eta^{-2/3}}{1 + \eta/\text{Pr}_t} d\eta \end{aligned}$$

Reorganizing the left hand side of the Eq. 5 to a relation for the heat transfer coefficient $h = q_w/(T_w - T_i)$

$$\frac{(T_w - T_i)k\sqrt{\tau_w/\rho_l}}{q_w/\nu} = \int_0^{\delta_{nw}^+} \frac{dy^+}{1 + F_I(\text{Pr}, \nu) \frac{\epsilon_M}{\text{Pr}_t}}, \quad \text{Pr}_t \simeq 1 \text{ Kato, et al. (1968)}$$

$$\frac{k}{h} \left(\frac{\sqrt{\tau_w/\rho_l}}{\nu} \right) = \frac{(\gamma \text{Pr})^{-1/3}}{3} \int_0^{\gamma(\delta_{nw}^+)^3 \text{Pr}} \frac{\eta^{-2/3}}{1 + \eta/\text{Pr}_t} d\eta$$

Taking the inverse on both sides yields the Eq. 6

$$\frac{h}{k} \left(\frac{\nu}{\sqrt{\tau_w/\rho_l}} \right) = (\gamma \text{Pr})^{1/3} \left(\frac{1}{3} \int_0^{\gamma(\delta_{nw}^+)^3 \text{Pr}} \frac{\eta^{-2/3}}{1 + \eta/\text{Pr}_t} d\eta \right)^{-1}$$

$$\frac{h}{k} \left(\frac{\nu}{\sqrt{\tau_w/\rho_l}} \right) = \frac{(\gamma \text{Pr})^{1/3}}{I}$$

Where we define I to be the following with $C_{1.2} = 3$ being an integer numerical constant.

$$I = \frac{1}{C_{1.2}} \int_0^{\gamma(\delta_{nw}^+)^3 \text{Pr}} \frac{\eta^{-2/3}}{1 + \eta/\text{Pr}_t} d\eta$$

□

To evaluate the integral I , you can proceed as follows. First, we adopt the idealization the Prandtl number is high. Since the integrand goes to zero as η gets large, we assume we can set the upper limit to infinity with little error. Then, break the integral up into three parts:

$$I = \frac{1}{C_{1.2}} \int_0^\infty \frac{\eta^{-2/3}}{1 + \eta/\text{Pr}_t} d\eta \quad (8)$$

$$I = \frac{1}{C_{1.2}} \int_0^{0.01} \frac{\eta^{-2/3}}{1 + \eta} d\eta + \frac{1}{C_{1.2}} \int_{0.01}^{100} \frac{\eta^{-2/3}}{1 + \eta} d\eta + \frac{1}{C_{1.2}} \int_{100}^\infty \frac{\eta^{-2/3}}{1 + \eta} d\eta \quad (9)$$

In the first integral, the denominator of the integrand only varies between 1 and 1.01, so set the denominator to its mean value 1.005. For the third integral, η is much larger than one over its η range, so neglect one compared to η in the denominator, making the integrand $\eta^{-5/3}$.

$$I \simeq \frac{1}{C_{1.2}} \int_0^{0.01} \frac{\eta^{-2/3}}{1.005} d\eta + \frac{1}{C_{1.2}} \int_{0.01}^{100} \frac{\eta^{-2/3}}{1 + \eta} d\eta + \frac{1}{C_{1.2}} \int_{100}^\infty \eta^{-5/3} d\eta \quad (10)$$

To determine the numerical (constant) value of I , we compute the first and third integral analytically, and numerically compute the second integral. Using an integration tool in Matlab, we set up a simple trapezoidal rule method to compute the second integral, for which the code description is found in [Section 3.1](#).

Task II.b. Evaluation of the integral I

Solution. Splitting the integral into 3 parts, we evaluate the first two integrals analytically

$$I_1 = \frac{1}{C_{1.2}} \int_0^{0.01} \frac{\eta^{-2/3}}{1.005} d\eta$$

$$= \frac{1}{3} \int_0^{0.01} \frac{\eta^{-2/3}}{1.005} d\eta$$

$$= \frac{1}{3.015} \left[3\eta^{1/3} \right]_0^{0.01} \Rightarrow I_1 = 0.2144$$

The third integral is evaluated as follows

$$\begin{aligned}
 I_1 &= \frac{1}{C_{1.2}} \int_{100}^{\infty} \eta^{-5/3} d\eta \\
 &= \frac{1}{3} \lim_{t \rightarrow \infty} \int_{100}^t \eta^{-5/3} d\eta \\
 &= \frac{1}{3} \lim_{t \rightarrow \infty} \left[\frac{\eta^{-2/3}}{-2/3} \right]_{100}^t \\
 &= -\frac{1}{3} \cdot \frac{3}{2} \lim_{t \rightarrow \infty} [t^{-2/3} - 100^{-2/3}] \Rightarrow I_3 = 0.02321
 \end{aligned}$$

The second integral was evaluated using the trapezoidal rule method in MATLAB and we got the integral to be

$$I_2 = 0.971216$$

Therefore, summing up all the individual integrals, we get a final value of

$$I = \sum_{i=1}^3 I_i = 1.208826$$

□

2.3 Task III Results and Discussion

For this next task, we shift focus to the momentum transport. For the boundary layer U -momentum equation,

$$U \frac{\partial U}{\partial x} + V \frac{\partial U}{\partial y} = \frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l} + \frac{1}{\rho_l} \frac{\partial}{\partial y} (\tau) = \frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l} + \frac{\partial}{\partial y} \left((\nu + \epsilon_M) \frac{\partial U}{\partial y} \right) \quad (11)$$

Neglecting the momentum convection terms and integrating this equation across the film to obtain a relation for the wall shear as a function of the film thickness and the other parameters in the equation.

$$\tau_w = f_{\tau_w}(g, \theta, \rho_l, \rho_g, \delta) \quad (12)$$

Task III. Wall-shear relation from the momentum equation

Solution. Neglecting the convective terms in Eq. 11., which is consistent with the assumption of a thin-film liquid, we get

$$\begin{aligned}
 0 &= \frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l} + \frac{\partial}{\partial y} \left((\nu + \epsilon_M) \frac{\partial U}{\partial y} \right) \\
 \frac{\partial}{\partial y} \left((\nu + \epsilon_M) \frac{\partial U}{\partial y} \right) &= -\frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l}
 \end{aligned}$$

Also, assuming turbulence is isotropic, we can use the relation: $\tau_w = \rho_l (\nu + \epsilon_M) \frac{dU}{dy}$ to get

$$\begin{aligned}
 \frac{\partial}{\partial y} \left(\frac{\tau_w}{\rho_l} \right) &= -\frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l} \\
 \frac{1}{\rho_l} \frac{\partial \tau_w}{\partial y} &= -\frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l} \\
 \frac{\partial \tau_w}{\partial y} &= -g \sin \theta (\rho_l - \rho_g)
 \end{aligned}$$

Now integrating from the wall $y = 0$ to the free surface at $y = \delta$

$$\int_0^\delta \frac{\partial \tau_w}{\partial y} dy = - \int_0^\delta g \sin \theta (\rho_l - \rho_g) dy$$

$$\tau_w(\delta) - \tau_w(0) = -g \sin \theta (\rho_l - \rho_g) \int_0^\delta dy$$

$$\tau_w(\delta) - \tau_w(0) = -g \sin \theta (\rho_l - \rho_g) \delta$$

Since at the free surface, there is no shear force, it must be true that $\tau_w(\delta) = 0$, and at the wall $\tau_w(0) = \tau_w$, hence

$$-\tau_w = -g \sin \theta (\rho_l - \rho_g) \delta \Rightarrow \tau_w = g \sin \theta (\rho_l - \rho_g) \delta$$

So compactly, our relation for the wall shear as a function of the film thickness and the other parameters in the equation can be written as

$$\tau_w = f_{\tau_w}(g, \theta, \rho_l, \rho_g, \delta) = g \sin \theta (\rho_l - \rho_g) \delta$$

□

2.4 Task IV Results and Discussion

By characterizing the wall shear, we are able to determine a mass-flow relation. The mass-flow rate per unit width of the surface (in the z direction) is computed by integrating the U velocity across the film Γ :

$$\Gamma = \rho_l \int_0^\delta U dy \quad (13a)$$

Using the power-law universal velocity profile we obtain

$$u^+ = 8.75(y^+)^{1/7} \quad (13b)$$

to evaluate the integral in Equation (13a) and obtain a relation for mass flow Γ (per unit width of surface) as a function of wall shear, film thickness and properties:

$$\Gamma = f_\Gamma(\tau_w, \rho_l, \nu, \delta) \quad (14)$$

This result has the form

$$\Gamma = C_{1.4} \rho \nu \left(\frac{\delta}{\nu / \sqrt{\tau_w / \rho_l}} \right)^{n_{1.4}} \quad (15)$$

where $C_{1.4}$ and $n_{1.4}$ are numerical constants.

Task IV. Mass-flow rate relation using the power-law universal velocity profile

Solution. We compute the mass flow rate per unit width of the surface (in the z -direction) as follows.

$$\Gamma = \rho_l \int_0^\delta U dy$$

Recall from **Task I**, we introduce the following non-dimensional variables

$$u^+ \triangleq \frac{U}{u_\tau} \Rightarrow U = u_\tau u^+ \quad , \quad y^+ \triangleq \frac{y u_\tau}{\nu} \text{ where } u_\tau = \sqrt{\tau_w / \rho_l}$$

Now, using the power-law universal velocity profile

$$U = u_\tau 8.75(y^+)^{1/7} = u_\tau 8.75 \left(\frac{y u_\tau}{\nu} \right)^{1/7}$$

Substituting into our mass flow integral yields

$$\begin{aligned}
 \Gamma &= \rho_l \int_0^\delta u_\tau 8.75 \left(\frac{yu_\tau}{\nu} \right)^{1/7} dy \\
 &= 8.75 \rho_l u_\tau^{8/7} \nu^{-1/7} \int_0^\delta y^{1/7} dy \\
 &= 8.75 \rho_l u_\tau^{8/7} \nu^{-1/7} \left[\frac{y^{8/7}}{8/7} \right]_0^\delta \\
 &= 8.75 \rho_l u_\tau^{8/7} \nu^{-1/7} \cdot \frac{7}{8} \delta^{8/7} \\
 &= \frac{35}{4} \cdot \frac{7}{8} \rho_l u_\tau^{8/7} \nu^{-1/7} \delta \Rightarrow \Gamma = 7.65625 \rho_l u_\tau^{8/7} \nu^{-1/7} \delta^{8/7}
 \end{aligned}$$

We now need to rearrange it into the canonical form described in Eq. 15 as required.

$$\begin{aligned}
 7.65625 \rho_l u_\tau^{8/7} \nu^{-1/7} \delta^{8/7} &= 7.65625 \rho_l \nu \left(\frac{u_\tau}{\nu} \right)^{8/7} \delta^{8/7} \\
 &= 7.65625 \rho_l \nu \left(\frac{\delta}{\nu/u_\tau} \right)^{8/7}
 \end{aligned}$$

Therefore, the mass flow rate per unit width of the surface in the z -direction is

$$\Gamma = 7.65625 \rho_l \nu \left(\frac{\delta}{\nu/\sqrt{\tau_w/\rho_l}} \right)^{8/7}$$

where $C_{1.4} = 7.65625$ and $n_{1.4} = 8/7$. □

2.5 Task V Results and Discussion

This task is split into two parts (a) and (b). For part (a), we combine Equation (12) and Equation (15) to eliminate the wall shear and obtain a relation between mass flow per unit width of the surface Γ , film thickness δ , with ρ_l , ρ_g , ν , g , θ appearing as parameters. We derive and document our work below.

Task V.a. Elimination of shear stress

Solution. Modifying the equation for wall shear and using $u_\tau = \sqrt{\tau_w/\rho_l}$, we get

$$\begin{aligned}
 \tau_w &= g \sin \theta (\rho_l - \rho_g) \delta \\
 \sqrt{\frac{\tau_w}{\rho_l}} &= \sqrt{\frac{g \sin \theta (\rho_l - \rho_g) \delta}{\rho_l}} \\
 u_\tau &= \sqrt{\frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l}} \delta^{1/2}
 \end{aligned}$$

We want to form the ratio $\delta/(\nu/u_\tau)$. We do this as follows

$$\frac{\delta u_\tau}{\nu} = \frac{\delta}{\nu/u_\tau} = \sqrt{\frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l}} \frac{\delta^{3/2}}{\nu}$$

We now substitute $\delta u_\tau / \nu$ into the Γ equation

$$\begin{aligned}\Gamma &= 7.65625\rho_l\nu \left(\sqrt{\frac{g \sin \theta (\rho_l - \rho_g)}{\rho_l}} \frac{\delta^{3/2}}{\nu} \right)^{8/7} \\ &= 7.65625\rho_l\nu ((\rho_l - \rho_g)g \sin \theta)^{4/7} \rho_l^{-4/7} \frac{\delta^{12/7}}{\nu^{8/7}} \\ &= 7.65625\rho_l^{3/7} ((\rho_l - \rho_g)g \sin \theta)^{4/7} \nu^{-1/7} \delta^{12/7}\end{aligned}$$

Rearranging for δ , we get our desired equation with no shear stress

$$\delta = \left[\frac{\Gamma}{7.65625\rho_l^{3/7} ((\rho_l - \rho_g)g \sin \theta)^{4/7} \nu^{-1/7}} \right]^{7/12}$$

□

Note that if a value of mass flow per unit width Γ is specified and turbulent flow results, this indicates that the system will have the film thickness δ dictated by this equation as a consequence of the physics. Part (b) of this tasks requires us to combine Equations (6), (12), and (15) to eliminate δ and obtain a non-dimensional relation for the heat transfer coefficient having the form.

$$\frac{h}{k} \left(\frac{\nu^2 \rho_l}{g \sin \theta (\rho_l - \rho_g)} \right)^{1/3} = C_{1.5} \text{Pr}^{1/3} \text{Re}^{n_{1.5}} \quad (16)$$

Where $C_{1.5}$ and $n_{1.5}$ are numerical constants and the Reynolds number Re is defined as

$$\text{Re} = \frac{4\Gamma}{\rho_l \nu} \quad (17)$$

We document our derivation and report the value of $n_{1.5}$ as a fraction, and the value of $C_{1.5}$ to three significant figures.

Task V.b. Elimination of δ

Solution. Starting with Eq. 6, we rearrange for h/k

$$\frac{h}{k} \cdot \frac{\nu}{u_\tau} = \frac{(\gamma \text{Pr})^{1/3}}{I} \Rightarrow \frac{h}{k} = \frac{(\gamma \text{Pr})^{1/3}}{I} \cdot \frac{u_\tau}{\nu}$$

We can generate an expression for u_τ as follows

$$\begin{aligned}\tau_w &= (\rho_l - \rho_g)g \sin \theta \delta \\ \sqrt{\frac{\tau_w}{\rho_l}} &= u_\tau = \sqrt{\frac{(\rho_l - \rho_g)g \sin \theta}{\rho_k}} \delta^{1/2}\end{aligned}$$

Then, we modify the mass flow rate equation and solve for $\frac{\delta}{\nu/u_\tau}$

$$\begin{aligned}\Gamma &= 7.65625\rho_l\nu \left(\frac{\delta}{\nu/u_\tau} \right)^{8/7} \\ \frac{\delta}{\nu/u_\tau} &= \left(\frac{\Gamma}{7.65625\rho_l\nu} \right)^{7/8} \\ \delta &= \frac{\nu}{u_\tau} \left(\frac{\Gamma}{7.65625\rho_l\nu} \right)^{7/8}\end{aligned}$$

Substituting δ into the expression for u_τ

$$\begin{aligned} u_\tau &= \sqrt{\frac{(\rho_l - \rho_g)g \sin \theta}{\rho_l}} \left(\frac{\nu}{u_\tau} \left(\frac{\Gamma}{7.65625 \rho_l \nu} \right)^{7/8} \right)^{1/2} \\ u_\tau^{3/2} &= \sqrt{\frac{(\rho_l - \rho_g)g \sin \theta}{\rho_l}} \nu^{1/2} \left(\frac{\Gamma}{7.65625 \rho_l \nu} \right)^{7/16} \\ u_\tau &= \left(\frac{(\rho_l - \rho_g)g \sin \theta}{\rho_l} \right)^{1/3} \nu^{1/3} \left(\frac{\Gamma}{7.65625 \rho_l \nu} \right)^{7/24} \\ \frac{u_\tau}{\nu} &= \nu^{-2/3} \left(\frac{(\rho_l - \rho_g)g \sin \theta}{\rho_l} \right)^{1/3} \left(\frac{\Gamma}{7.65625 \rho_l \nu} \right)^{7/24} \end{aligned}$$

Now, inserting this expression into Eq. 6, we get

$$\begin{aligned} \frac{h}{k} &= \frac{(\gamma \text{Pr})^{1/3}}{I} \cdot \frac{u_\tau}{\nu} \\ \frac{h}{k} &= \frac{\gamma^{1/3}}{I} \text{Pr}^{1/3} \nu^{-2/3} \left(\frac{(\rho_l - \rho_g)g \sin \theta}{\rho_l} \right)^{1/3} \left(\frac{\Gamma}{7.65625 \rho_l \nu} \right)^{7/24} \end{aligned}$$

Multiplying both sides by a factor gets us

$$\begin{aligned} \frac{h}{k} \left(\frac{\nu^2 \rho_l}{g \sin \theta (\rho_l - \rho_g)} \right)^{1/3} &= \frac{\gamma^{1/3}}{I} \text{Pr}^{1/3} \nu^{-2/3} \left(\frac{\nu^2 \rho_l}{g \sin \theta (\rho_l - \rho_g)} \right)^{1/3} \left(\frac{(\rho_l - \rho_g)g \sin \theta}{\rho_l} \right)^{1/3} \left(\frac{\Gamma}{7.65625 \rho_l \nu} \right)^{7/24} \\ &= \frac{\gamma^{1/3}}{I} \text{Pr}^{1/3} \nu^{-2/3} \frac{\nu^{2/3} (\rho_l)^{1/3}}{(g \sin \theta (\rho_l - \rho_g))^{1/3}} \cdot \frac{(g \sin \theta (\rho_l - \rho_g))^{1/3}}{\rho_l^{1/3}} \left(\frac{\Gamma}{7.65625 \rho_l \nu} \right)^{7/24} \\ &= \frac{\gamma^{1/3}}{I (7.65625)^{7/24}} \text{Pr}^{1/3} \cdot \left(\frac{\Gamma}{\rho_l \nu} \right)^{7/24} \end{aligned}$$

Using the Reynolds number, we can rewrite the term inside the bracket

$$\begin{aligned} \frac{h}{k} \left(\frac{\nu^2 \rho_l}{g \sin \theta (\rho_l - \rho_g)} \right)^{1/3} &= \frac{\gamma^{1/3}}{I (7.65625)^{7/24}} \text{Pr}^{1/3} \cdot \left(\frac{\text{Re}}{4} \right)^{7/24} \\ &= \frac{\gamma^{1/3}}{I (7.65625)^{7/24} 4^{7/24}} \text{Pr}^{1/3} \text{Re}^{7/24} \end{aligned}$$

Now, substituting in the known values of $\gamma = 5.10 \times 10^{-4}$ and our calculate integral $I = 1.208826$, we have

$$\frac{h}{k} \left(\frac{\nu^2 \rho_l}{g \sin \theta (\rho_l - \rho_g)} \right)^{1/3} = 0.0244 \text{Pr}^{1/3} \text{Re}^{7/24}$$

□

3 Implementation and Development of Code

The following section will be a summary of our analysis organization including a description of our code organization for each task as well as different cases within each task that were considered. For each major block of the script, we include the relevant code snippet followed by an explanation that explains how that section fits into the overall analysis. A complete copy of our code for each question, respectively, can be found in [Section 5](#) at the end of the report. Also note that running the code for different configurations and material properties are not included since the change is minimal.

3.1 Task II

The only necessary piece of code in this part of the project was a numerical integration scheme to calculate I_2 in [Section 2.2](#).

We first discretizes the interval $[a, b]$ into N equally space sub-intervals with a width of $\delta = \frac{b-a}{N}$. We then evaluate the integrand $f(\eta) = \frac{\eta^{-2/3}}{1+\eta}$ at all $N + 1$ nodes, and we apply the trapezoid rule in a single pass.

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2} \sum_{k=1}^N (f(x_{k-1}) + f(x_k)) = \Delta x \left(\frac{f(x_N) + f(x_0)}{2} + \sum_{k=1}^{N-1} f(x_k) \right)$$

Since, we just summed all the terms exactly once, the time complexity scales linearly as $\mathcal{O}(N)$. We chose $N = 100\,000$ since it gave us good accuracy without taking too much computing power. If higher accuracy was required, we could have always increased N at the cost of proportionally increased runtime. We include pseudo-code of the algorithm for reference. Actual code can be found in the [Code Appendix](#).

Algorithm 1 Trapezoidal Rule for Numerical Integration

1. Initialize integration parameters and number of subdivisions:

$$N \leftarrow 100000, \quad lb \leftarrow 0.01, \quad ub \leftarrow 100, \quad \delta \leftarrow \frac{ub - lb}{N}.$$

2. Compute sample points:

$$\text{for } i = 0, 1, \dots, N : \quad \eta_i \leftarrow lb + i \delta.$$

3. Evaluate integrand:

$$\text{for } i = 0, 1, \dots, N : \quad f_i \leftarrow \frac{\eta_i^{-2/3}}{1 + \eta_i}.$$

4. Apply trapezoidal rule:

$$I \leftarrow \frac{1}{3} \delta \left(\frac{1}{2} f_0 + \sum_{i=1}^{N-1} f_i + \frac{1}{2} f_N \right).$$

Output: I (approximate value of the integral).

4 Conclusion

This first part of the project allowed us to develop and compare various of analytical and numerical approaches to predict heat transfer in a turbulent, high-Prandtl-number falling liquid film. Starting with the eddy-diffusivity model for the near-wall energy balance, we derived an integral expression for the wall heat flux and we were able to evaluate it using some approximations, analytically, and numerically via the Trapezoid rule for numerical integration. A similarity transformation then allowed us to simplify the governing integral into a universal heat-transfer correlation, isolating the integral non-dimensionless group I .

We also looked at simplifying the turbulent boundary layer momentum equation to obtain an explicit wall-shear relation, and—using the power-law universal velocity profile—derived the mass-flow rate per unit width. By eliminating the shear and film thickness, we ended up successfully with a non-dimensional correlation for the heat-transfer coefficient as a function of Reynolds and Prandtl numbers. Each step in this portion allowed us to reinforce the others which gave us good predictions demonstrating that classical methods, despite their age, are robust and are of great practical value for rapid engineering estimation when dealing with large-scale falling film applications. The next part of our project will be a quick introduction to a modern technique in the world of physical sciences: Physics-informed Neural Networks and their application to convective transport.

PART 2: PINN Solution for Convective Channel Flow

5 Introduction and Theoretical Background

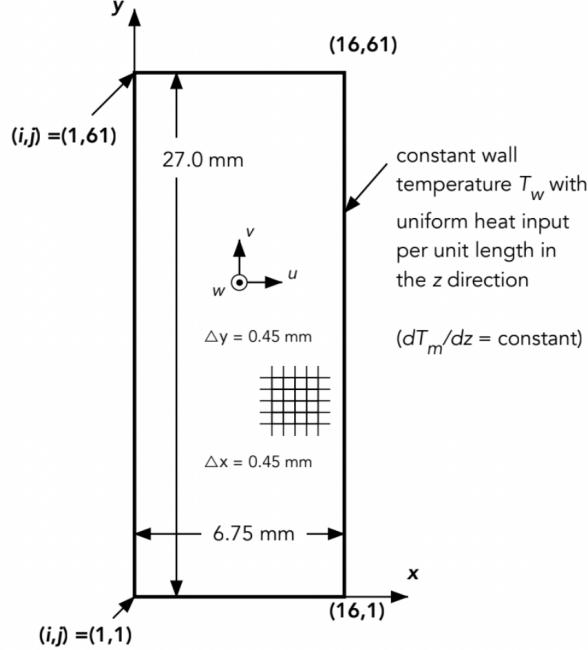


Figure 2: Constant Wall Temperature T_w with Uniform Heat Input

The 2nd part of this project introduces us to the use of a physics-inspired neural network (PINN) to solve a convective transport problem for which there are governing partial differential equations and associated boundary conditions. Specifically, this project focuses on the type of fully-developed flow and heat transfer considered in Project 1, but with a completely different approach to determining a solution. The project aims to model fully-developed flow and heat transfer in a channel with a specified cross section, uniform temperature around the perimeter at each axial (\$z\$) location, and uniform heat transfer input per unit length of the channel.

Here the governing equations and boundary conditions discussed in Project 1 apply. The equations are:

$$0 = \frac{\mu}{\rho} \left[\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} \right] - \frac{1}{\rho} \frac{\partial P}{\partial z} \quad (18)$$

$$w \frac{\partial T}{\partial z} = \alpha \left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right] \quad (19)$$

Note that \$w\$ is the velocity component in the axial direction of the passage. For fully developed flow, \$u\$ and \$v\$ are zero and \$z\$ derivatives are zero, except for \$\partial T/\partial z\$, which is constant for constant heat addition at the walls per unit length, and the imposed pressure gradient \$\partial P/\partial z\$. The continuity equation is satisfied for these conditions since all terms in it are zero. Recall that for fully developed heat transfer with uniform heat addition at the walls,

$$\frac{\partial T}{\partial z} = \frac{dT_m}{dz} \text{ at all } (x, y) \text{ locations.} \quad (20)$$

Also, the \$u\$ and \$v\$ momentum equations reduce to the pressure gradient in each direction equaling zero, which means the pressure is uniform across any cross section of the tube and \$\partial P/\partial z\$ in the flow is equal to the imposed constant pressure gradient \$dP/dz\$. The governing equations can therefore be written as

$$\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} = \frac{1}{\rho \nu} \frac{dP}{dz} \quad (21)$$

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{w}{\alpha} \left(\frac{dT_m}{dz} \right) \quad (22)$$

Here we will consider flow of water in which the right side of the above equations contains a fixed constant or a fixed constant multiplying w . In addition, here the parameter units will be expressed in mm instead of meters. In equations 21 and 22, pressure is in N/mm² (1 Pa = 1 N/m² = 10⁻⁶N/mm²), velocity w is in mm/s, x , y , and z are in units of mm and T is in deg C. Here, the following properties for water are used: $\alpha = 0.146\text{mm}^2/\text{s}$, $\rho = 9.97 \times 10^{-7}\text{kg/mm}^3$, $\nu = 0.826\text{mm}^2/\text{s}$, $c_p = 4164\text{J/kg°C}$, $k = 6.06 \times 10^{-4}\text{W/mm°C}$. And, in mm-based units, the values of the constants in 21 and 22 are:

$$\frac{1}{\mu} \left(\frac{dP}{dz} \right) = -80.0\text{mm}^{-1}\text{s}^{-1}, \quad \frac{1}{\alpha} \left(\frac{dT}{dz} \right) = 0.103^\circ\text{Cs/mm}$$

Also the boundary conditions that apply at the passage wall are

$$w = 0(\text{mm/s}) \text{ and } T = 90^\circ\text{C} \text{ (at the passage wall)}$$

Two key ideas are foundations for PINN approaches to solving governing PDE's for physical systems. One is the use of a neural network as a powerful means of fitting performance trends in the data or the way the system behaves. PINN methods can be designed to fit multiple features of system behavior. In a system governed by a PDE and boundary or initial conditions, both satisfying the equation and satisfying the boundary or initial conditions can be captured.

In training the neural network, feedback on the goodness of fit is provided as the value of a loss function (sometimes called an error function), so that the best possible fit corresponds to a minimum in the loss function value returned. The neural network typically has a large number of adjustable parameters, and the objective is to find the combination of those parameters that minimizes the loss function. The task is therefore to find the minimum in a very high dimensional hyperspace. The PINN solution scheme for the system of interest here is depicted in the figure below.

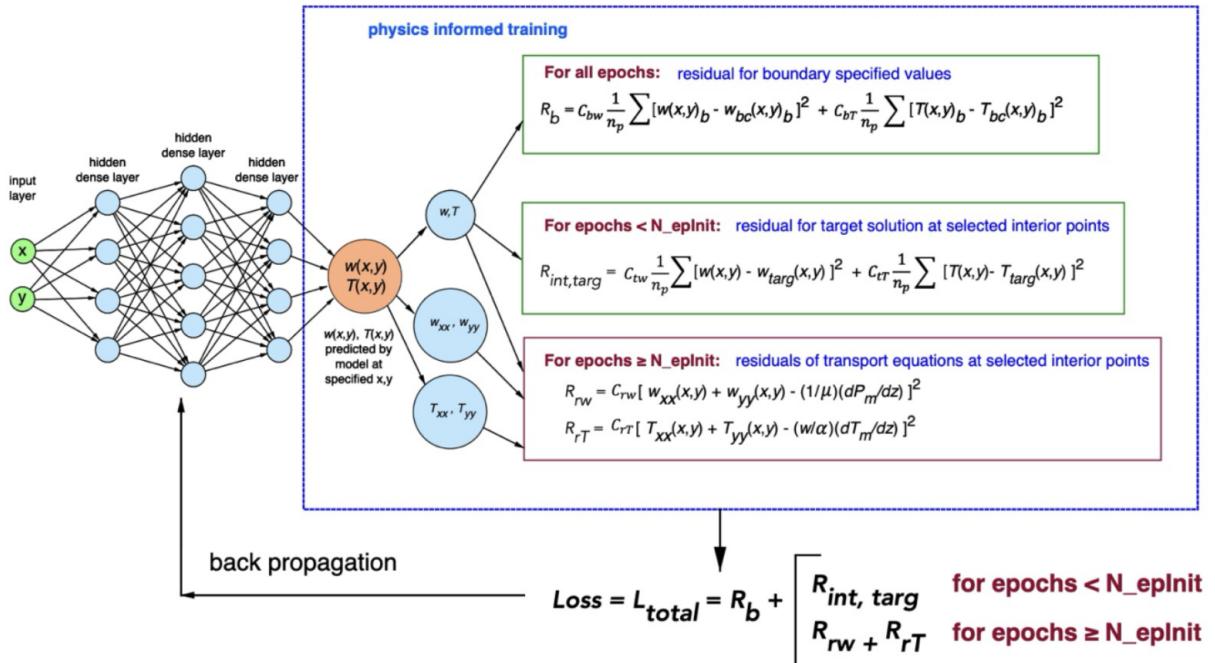


Figure 3: Schematic of PINN model for fully developed flow and convection in a channel

5.1 Work Division

The tasks for this project were distributed among both team members to ensure efficient collaboration and thorough completion of all required components. Each member contributed to different aspects of the project as follows:

1. The setup of the PINN notebook, full training to convergence, and extraction of benchmark wall-heat-flux and velocity data were completed by **Derek Shah** and debugging as well as revision for accuracy was done by **Larry Hui**.
2. All of the geometry/mesh edits for the rounded corner variant, the subsequent retraining of the network, and the comparison plots were implemented by **Derek Shah** with some technical validation, debugging, and analysis provided by **Larry Hui**.
3. The calculation of local temperature gradients, assembly of comparative heat-flux tables, and final performance assessment were carried out by **Derek Shah** and **Larry Hui** to get both our input on optimal performance geometry.

5.2 Task Descriptions

This project is split into 3 tasks which are briefly described below.

1. **Task I:** This task focuses on preparing the Jupyter Notebook to accurately and fully represent the developed flow and heat transfer for a rectangular channel. We populate the boundary-point tensor with the wall coordinates provided; we verify the data arrays and boundary checker contains identical point pairs, and then we run the following cells sequentially. We train the network in two long epoch blocks until the loss hits our tolerance of about 0.3. When it converges, we use the field data and the plots to compute the temperature gradient and heat flux to the fluid at the mid-span of both the short and long walls of the channel; we also determine the peak flow velocity and minimum temperature in the flow and tabulate them next to results from the Poisson-equation Gauss-Siedel code developed in Project 1, correcting for the shift to mm-based units. Further discussion and results can be found in [Section 3.1](#).
2. **Task II:** We now consider a new geometry. Modifying the code to model the flow and heat transfer in the rounded corner channel shown in [Fig. 5](#). We remove 12 grid points closest to each sharp corner and add the 12 new arc points. Like above, we retrain our network, plot the field data, and discuss how they differ from the corresponding plots for the sharp rectangular domain. [Section 3.2](#).
3. **Task III:** Lastly, explore the variation of the heat flux along the wall of the passage in the near corner region. Extracting the temperature gradients at the wall nodes closest to each corner (and if we are doing it for the rounded tube, we extract them at the bend midpoint) we compute the heat flux and discuss how the heat flux behaves at the corner itself as well as how it varies along the wall surface as we approach the corner. Finally, we state which geometry offers the more uniform and better thermal performance backing it up with the following two temperature gradient trends. Discussion can be found [Section 3.2](#).

6 Implementation and Development of Code

The following section will be a summary of our analysis organization including a description of our code organization for each task as well as different cases within each task that were considered. For each major block of the script, we include the relevant code snippet followed by an explanation that explains how that section fits into the overall analysis. A complete copy of our code for each question, respectively, can be found in the [Appendix](#) at the end of the report. Also note that running the code for different configurations and material properties are not included since the change is minimal. An overview of the associated program for the PINN approach is provided below.

For this project, a computer code will be provided that implements the PINN scheme for solving the governing equations with their boundary conditions. A version of the provided program is in Python, set-up as a Jupyter notebook file. A pdf listing of the code, as well as the Jupyter notebook file (`CodeP3Pt2B.1PINNconvSp25.ipynb`) are provided. The program implements a Physics Inspired Neural Network solution of the steady, fully-developed flow and convective heat transfer in a channel with a specified cross section. Here is a summary description of the computer program structure:

1. Cell 1: Training Data Preparation
2. Cell 2: Definition of the Neural Network Structure
3. Cell 3: definition of the custom loss function used in the network training which computes a loss metric that combines metrics of the fit to the boundary condition and how well the solution obeys the governing momentum and heat transport equations.
4. Cell 4: contains code to execute the fit routine that trains the neural network to fit trends in the physical behavior in the available data and reflected in the governing equations.
5. Cells 5, 6, & 7: post-processing and visualization—these create plots of the w and T fields predicted by the trained model.

Ensure that the following dependencies (`TensorFlow`, `TensorFlow`, `matplotlib`, `scikit-learn`) as well as **Python 3.10.14**, or a version close to that is downloaded. In Python, special programming in the `keras` and `tensorflow` packages handle the neural network training minimization problem. Other environments for neural network analysis and modeling similarly have packages to handle the elements of the training.

As depicted in **Fig. 3**, during the training process, when x, y data are input to the network, output values of the downstream velocity w and temperature T are computed. The code also produces values of the second derivatives of w and T at (x, y) , based on the current condition of the model in the training process. In Python, `tensorflow` can provide derivative information because it generates that type of information during the (backpropagation) training process. A metric for the goodness of the fit at that point in the training is computed by a section of the code referred to as the loss function (or sometimes the error function). The average of this loss function value over many sets of input data is interpreted as the indicator of the goodness of the fit, with a smaller loss value being better. In neural network training aimed at fitting the data so outputs match output values in the input data set, calculation of the loss is often computed simply as the mean squared error between the predicted and data output values for a batch of data. In any case, the objective is to train the model to find a set of model adjustable parameters that minimize the loss function value.

6.1 Custom Loss Function Structure

Custom loss functions provide the means to customize the PINN modeling scheme to the specific system of interest. Use of simple rms error between data output values and model predictions for training set data as a loss function can be accomplished using a function call from the `keras` python package. For PINNs, specialized features are added to combine multiple loss mechanisms. As seen in **Fig. 3**, in this project three loss mechanisms are incorporated into the custom loss function. The first (in the red rectangle) is the sum of the square of the deviations between the model predictions of w and T at the boundary (wall) and the specified boundary conditions ($w = 0$ and $T = 90^\circ\text{C}$). These loss components are computed throughout the training process.

In the early portion of the simulation (number of epochs up to about 2000), the boundary loss calculation is followed by calculation of a loss component that sums the squared error between the model predictions of w and T and the corresponding value for an approximate target solution. The target solutions for w and T were constructed as approximate curve fits between the boundary values of w and T and estimates of their values in the middle of the channel based on values of a round tube with a diameter equal to the hydraulic diameter of the channel of interest here. (This is a place where physics inspires the model

structure.) This motivates the model to evolve towards the target solution early in the training. For epochs beyond about 2000, the custom loss function switches and instead of computing the loss function component for the target solution, the squared errors of the residue errors of the momentum equation and heat transfer PDE equations are computed and averages at numerous points inside the (x, y) domain of the flow passage. The early training to fit the estimated target solution gets the model fit to develop the right shape, and the subsequent training to predict a fit that obeys the governing PDE's and boundary conditions leads to a model fit that has a behavior that is consistent with the governing equations and boundary conditions.

The code file in the [Appendix](#) contains Python code that will accomplish the following:

1. Prepare the training data.
2. Specify the design of the neural network.
3. Compile the network and specify the features of the custom loss function.
4. Execute the training of the model.
5. Use the model to predict and plot the features of the w velocity and temperature fields.

We have set up the code to model the flow passage domain shown in [Fig. 4](#). This geometry is to be used in **Task I**.

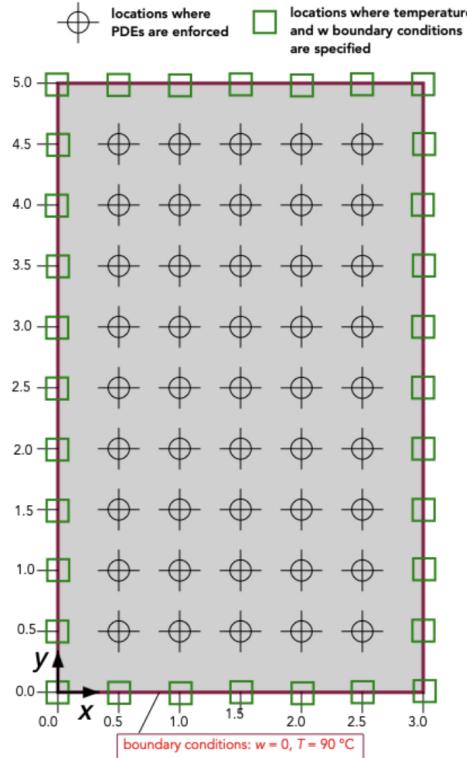


Figure 4: Domain of convective flow and heat transport with specified $w = 0$ and $T = 90^\circ\text{C}$ at the boundary. The coordinates x and y are in millimeters.

For **Task II**, we model the flow and heat transfer in the rounded-corner channel shown below in [Fig. 5](#).

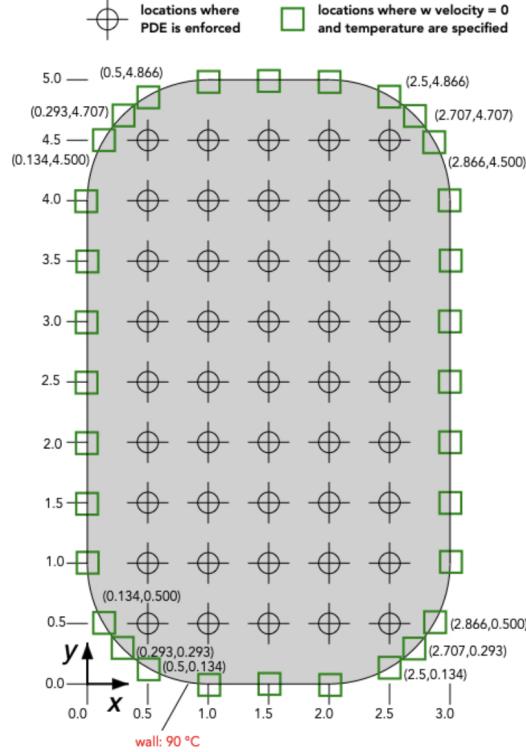


Figure 5: Domain of convective flow and heat transport for a rounded-corner tube with specified w velocity and T at the boundary. The coordinates x and y are in millimeters.

7 Results and Discussion

7.1 Task I Results and Discussion

First, the boundary conditions were modified so that all wall data points were included in the `pointsBoundaryChk` tensor. To begin, Cells 1 and 2 were run sequentially. Then, in Cell 3, `N_epInit` was set to 2000 and the cell was executed. Next, in Cell 4, `epochs` was set to 2500, initiating the training process.

During training, the loss progressively decreased with each epoch. After completing all 2500 epochs, the script reported and saved the smallest loss recorded during the batch. Following this, `N_epInit` in Cell 3 was reset to 1 and re-executed. Cell 4 was then run again for another 2500 epochs. This process was repeated in increments of 2500 to 10000 epochs to continue reducing the loss. The objective was to decrease the loss value below 0.3, indicating a well-fitted model. Training was deemed sufficient when the model achieved a loss value of 0.29. It is worth mentioning that the progress was extremely slow most likely due to the amount of trainable parameters. We also tried tuning the hyperparameters of our optimizer: Adam. In addition, we also tried several other ones including Stochastic Gradient Descent (SGD) and AdamW (which was faster thanks to the weight decay). The following plots in Fig. 6, 7 were generated once the code was run.

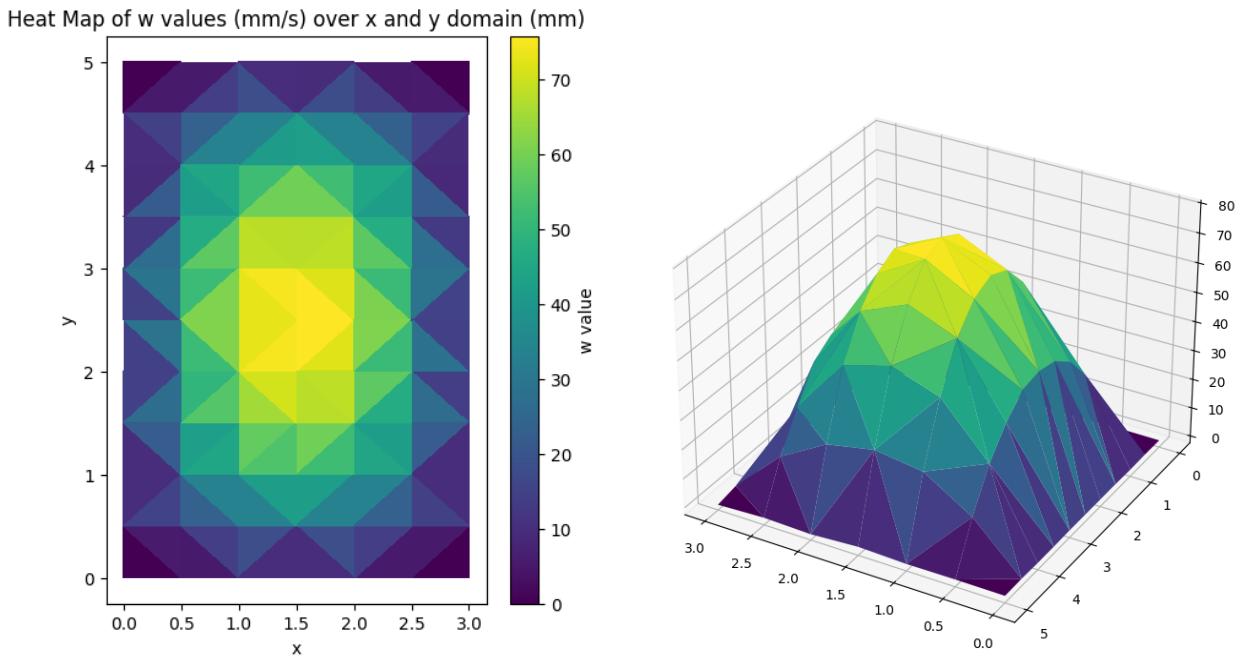


Figure 6: PINN Velocity Profile for Rectangular Channel

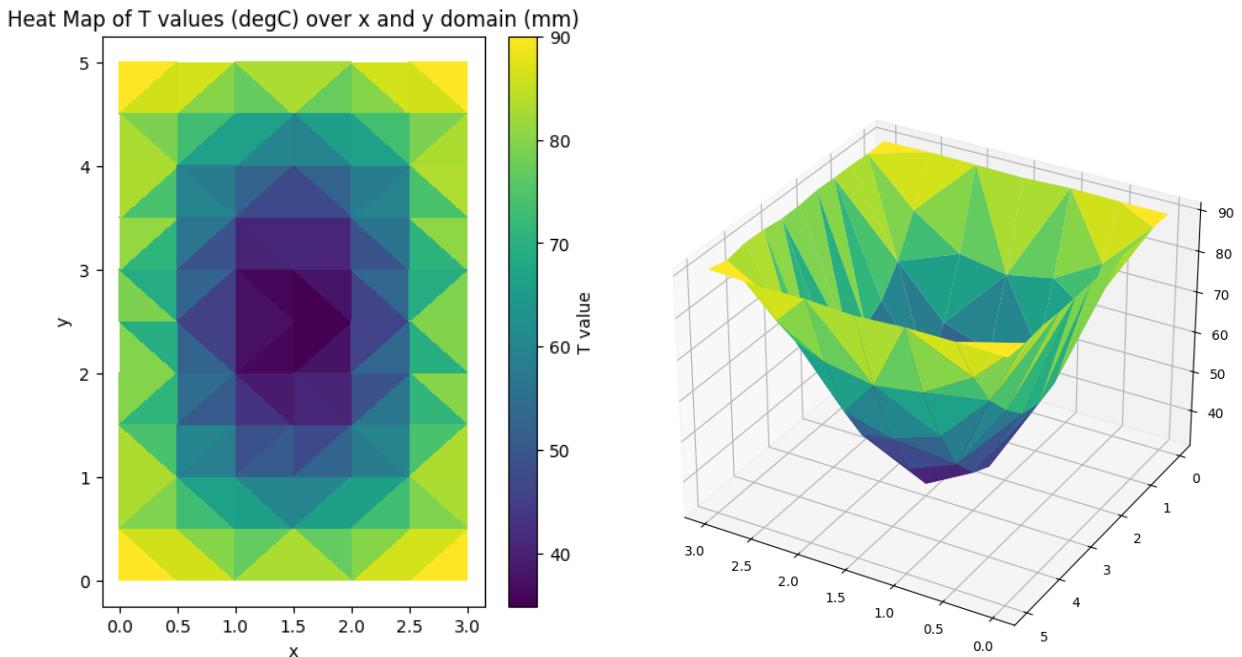


Figure 7: PINN Temperature Profile for Rectangular Channel

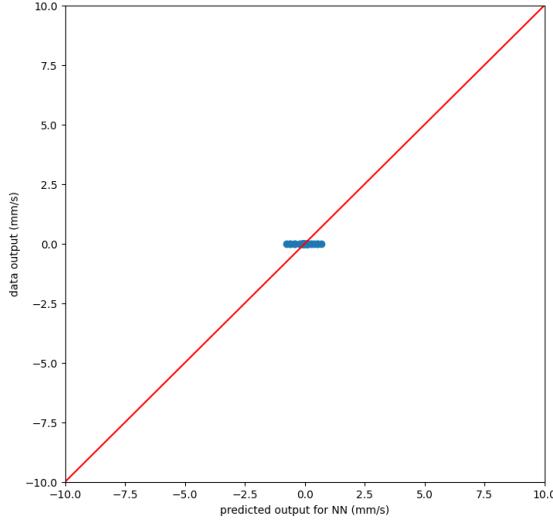


Figure 8: Parity Plot for Rectangular Channel

The PINN outputs are reported in millimetres and mm s^{-1} . All coordinates and velocities were therefore multiplied by 10^{-3} to convert them to metres and m s^{-1} . The thermal conductivity supplied for the PINN model is $k = 6.06 \times 10^{-4} \text{ W mm}^{-1} \text{ }^{\circ}\text{C}^{-1}$, which corresponds to $k = 0.606 \text{ W m}^{-1} \text{ }^{\circ}\text{C}^{-1}$.

$$\frac{\partial T}{\partial n} = \frac{T_{\text{int}} - T_{\text{wall}}}{\Delta n}, \quad q'' = -k \frac{\partial T}{\partial n}, \quad \Delta n = 0.5 \text{ mm} = 5 \times 10^{-4} \text{ m}.$$

Wall location	T_{wall} ($^{\circ}\text{C}$)	T_{int} ($^{\circ}\text{C}$)	$\partial T / \partial n$ ($^{\circ}\text{C m}^{-1}$)	q'' (kW m^{-2})
Long wall center (0 mm, 2.5 mm)	90.100	57.638	-6.49×10^4	39.3
Short wall center (1.5 mm, 0 mm)	89.755	69.047	-4.14×10^4	25.1

The maximum w -velocity predicted by the PINN is $w_{\text{max}} = 79.192 \text{ mm s}^{-1} = 0.0792 \text{ m s}^{-1}$, occurring at the point (1.5 mm, 2.5 mm). The coldest fluid temperature in the dataset is $T_{\text{min}} = 32.267^{\circ}\text{C}$, found at the same location.

Model	q''_{long} (kW m^{-2})	q''_{short} (kW m^{-2})	Peak $ w $ (m s^{-1})	T_{min} ($^{\circ}\text{C}$)
PINN	39.3	25.1	0.0792	32.27
Finite difference	0.58	1.5	0.0593	79.5

This large discrepancy between the PINN and finite-difference model is likely because the PINN's mesh is so coarse that it skips most of the thermal boundary layer region adjacent to the walls. Lacking interior nodes there, the network fits the sparse wall data by steepening its temperature profile over the few points it does see, which over-estimates the normal gradients and thus the wall heat flux. The finite-difference model, with 0.1 mm spacing, explicitly resolves that boundary layer and spreads the same temperature across the correct physical thickness, yielding lower and more realistic heat-flux values.

7.2 Task II Results and Discussion

To modify the model for the channel with rounded corners, the code was updated to reflect the geometry shown in **Fig 5**. First, the three coordinate points nearest to each of the four corners were removed from the `xdata` and `ydata` arrays in Cell 1, eliminating a total of 12 entries adjacent to the corners. Consequently, the same 12 (x, y) coordinate pairs were removed from the `pointsBoundaryChk tf.constant` tensor in Cell 4 to ensure consistency. Next, new entries were added to both the `xdata/ydata` arrays and

the `pointsBoundaryChk` tensor to include the coordinates of the 12 rounded corner points defined in **Fig. 5**. With the geometry correctly modified, the training procedure was repeated exactly as it was for the rectangular domain: Cells 1 and 2 were run sequentially, Cell 3 was initialized with `N_epInit = 2000`, followed by Cell 4 with `epochs = 2500`. Afterward, `N_epInit` was reset to 1, and Cell 4 was run iteratively until the loss dropped below 0.3. The resulting plots generated by the PINN model were saved:

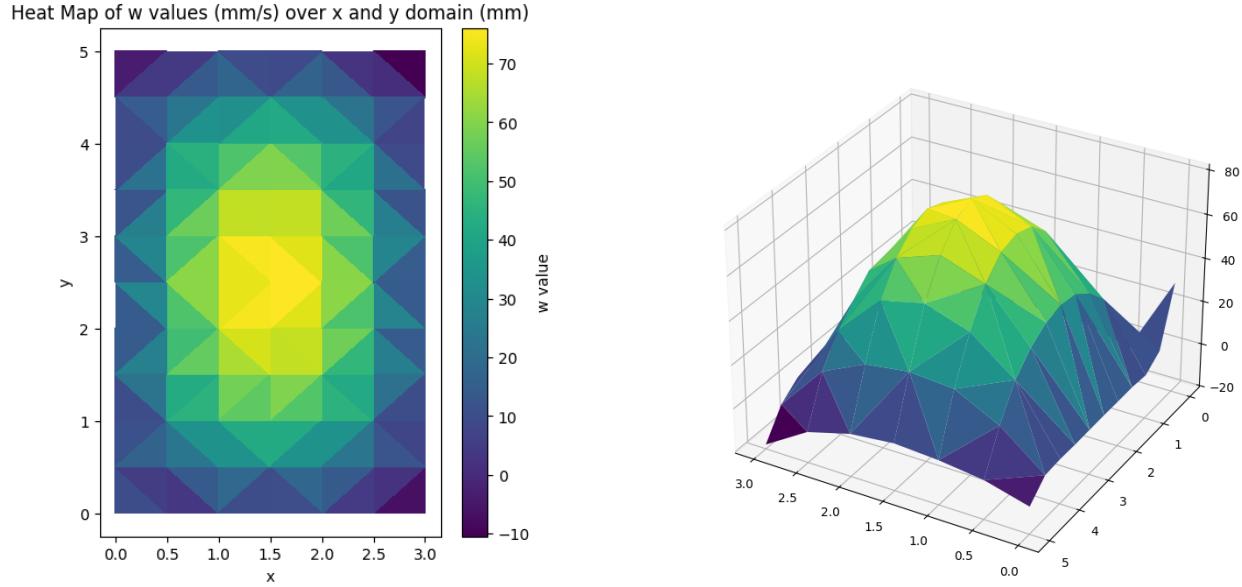


Figure 9: PINN Velocity Profile for Rounded Channel

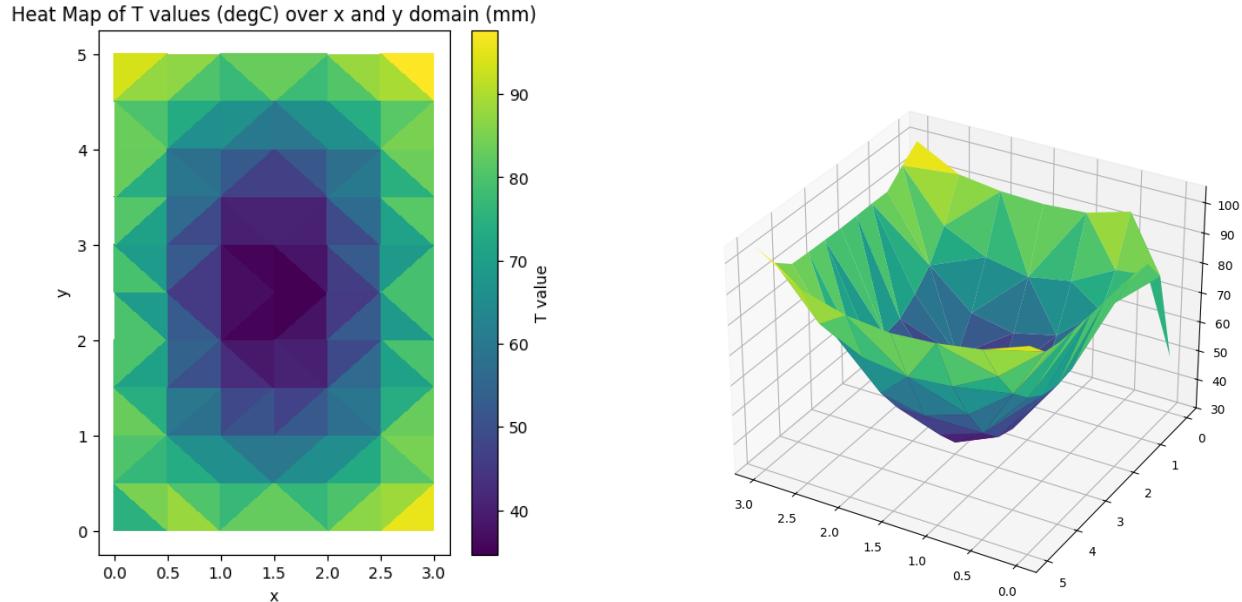


Figure 10: PINN Temperature Profile for Rounded Channel

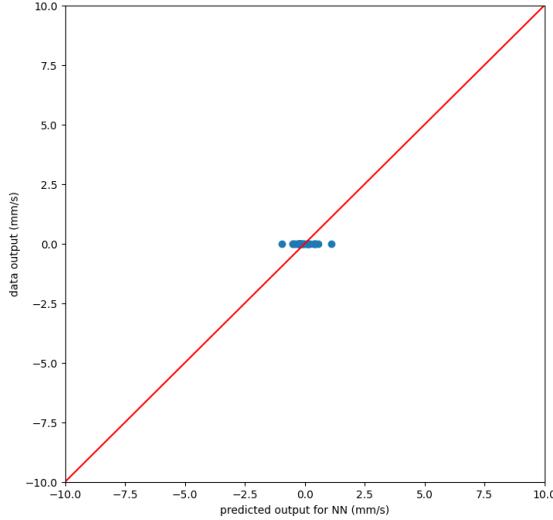


Figure 11: Parity Plot for Rounded Channel

The plots seem similar to the rectangular plots; however, on this plot there are noticeable discontinuities at the corners. This makes sense since the corners don't physically exist, so when trying to solve them, the model does not know what to do and diverges at the corners.

Model	q''_{long} (kW m^{-2})	q''_{short} (kW m^{-2})	Peak $ w $ (m s^{-1})	T_{\min} ($^{\circ}\text{C}$)
PINN – rectangular	39.3	25.1	0.0792	32.27
PINN – round	38.8	25.0	0.0804	31.39

It can be seen that the mid-wall heat flux values along with the max and min velocity and temperature seem to relatively agree between the two models, with the rectangular channels showing slightly better heat transfer characteristics based on the slightly large heat flux values.

7.3 Task III Results and Discussion

We use a finite one-sided first-order difference between each wall node and the first interior node ($\Delta n = 5.0 \times 10^{-4} \text{ m}$) to approximate the normal temperature gradient,

$$\left. \frac{\partial T}{\partial n} \right|_{\text{wall}} = \frac{T_{\text{int}} - T_{\text{wall}}}{\Delta n}, \quad q'' = -k, \frac{\partial T}{\partial n}, \quad k = 0.606 \text{ W m}^{-1} \text{ } ^{\circ}\text{C}^{-1}.$$

The resulting heat-flux densities at the two wall nodes closest to the corner are listed in Table ??.

Location (m)	Δn (m)	T_{wall} ($^{\circ}\text{C}$)	T_{int} ($^{\circ}\text{C}$)	$\partial T / \partial n$ ($^{\circ}\text{C m}^{-1}$)	q'' (kW m^{-2})
(0.000, 0.0005)	5.0×10^{-4}	89.984	78.127	-2.37×10^4	14.4
(0.0005, 0.000)	5.0×10^{-4}	90.010	78.127	-2.38×10^4	14.4

Away from the corner, mid-wall regions exhibit much steeper normal gradients (short wall $\sim 6.5 \times 10^4 \text{ } ^{\circ}\text{C m}^{-1}$, long wall $\sim 4.1 \times 10^4 \text{ } ^{\circ}\text{C m}^{-1}$), producing heat-flux densities of 39 and 25 kW m^{-2} , respectively. Approaching the corner the gradient drops to $\sim 2.4 \times 10^4 \text{ } ^{\circ}\text{C m}^{-1}$, so the flux is already down to $\approx 14 \text{ kW m}^{-2}$ one cell away. At the corner itself the wall normals are perpendicular; a vector normal to both faces must be the zero vector, forcing the normal component of the temperature gradient and thus the heat flux to vanish exactly at the corner. Thus the heat-flux density decreases smoothly along each wall as the corner is approached, tending to zero at the intersection.

As for the rounded channel, the heat flux and temperature gradient were calculated the same way above. However, for the bended section in order to get the normal component of heat flux the temperature at the previously corner node and interior node closest to the bend were taken, accounting for the different distance due to the diagonally points. The final values are tabulated below:

Location (m)	Δn (m)	T_{wall} ($^{\circ}\text{C}$)	T_{int} ($^{\circ}\text{C}$)	$\partial T / \partial n$ ($^{\circ}\text{C m}^{-1}$)	q'' (kW m^{-2})
(0.0025, 0.005)	5.0×10^{-4}	94.442	79.049	-3.08×10^4	18.7
(0.003, 0.0045)	5.0×10^{-4}	94.821	79.049	-3.15×10^4	19.1
(0.003, 0.005)	7.071×10^{-4}	103.858	79.049	-3.51×10^4	21.3

Note: The normal gradient at the bend was evaluated with $T_{\text{wall}}(3.0 \text{ mm}, 5.0 \text{ mm}) = 103.86^{\circ}\text{C}$ and the first interior point $T_{\text{int}}(2.5 \text{ mm}, 4.5 \text{ mm}) = 79.05^{\circ}\text{C}$ over a normal spacing $\Delta n = \sqrt{2} \times 0.5 \text{ mm} = 0.7071 \text{ mm} = 7.071 \times 10^{-4} \text{ m}$.

Based on these results, for two channels of the same nominal length and width, it can be reasonably determined that the rectangular channel is better at transferring heat. This can be physically seen by the table in **Task II**, where the rectangular channel has a greater heat flux and greater minimum interior temperature, meaning it has absorbed more heat from its surroundings. The reasons for this difference is because the rounded channels are actually better at transferring heat in the corners. It can be seen in the two tables in **Task III** that all values of the rounded geometry near the corner are greater than the rectangular, as the rectangular heat flux at the corner approaches 0. Because of this higher corner heat flux, the rounded geometry has a slightly less heat flux in the straight/middle sections of the walls when compared to rectangular. And because there are significantly more walls than there are corners, the rectangular geometry wins out in terms of being the better channel for facilitating heat transfer.

8 Conclusion

The physics informed neural network (PINN) and a high resolution finite-difference model (FDM) were analyzed for fully developed laminar flow and conjugate heat transfer in three channel geometries. The PINN, trained on a coarse 1 mm collocation grid, reproduced the bulk velocity field but over-predicted wall heat flux by roughly an order of magnitude because it under-resolved the thermal boundary layer that the 0.1 mm FDM grid captured. Filleting the rectangular corners by 1 mm redistributed heat-flux load from the long walls to the bends, raising local q'' near the fillet to about 21 kW m^{-2} ; however, the overall wall-integrated flux fell by $\sim 2\%$ because the straight-wall area dominates. Peak axial velocity ($\approx 0.08 \text{ m s}^{-1}$) and minimum core temperature ($\approx 32^\circ\text{C}$) changed negligibly, indicating that bulk flow is insensitive to modest geometric smoothing.

In practical terms, a sharp cornered rectangular channel maximizes total heat transfer for a given footprint: rounded corners slightly enhance local cooling at the bends but reduce flux along the much larger straight walls. Boosting PINN fidelity will require denser or adaptive collocation near walls, promising mesh-free flexibility without sacrificing quantitative accuracy.

9 Code Appendix

This appendix will include all the code relevant for **Part 1 Task I**, **Part 2 Task 1**, **Part 2 Task 2**. The code is also commented to ease your understanding and to display our thought process while writing.

9.1 Part 1 Code

```

1 % Number of blocks
2 N = 100000;
3
4 % Limits of integration
5 lb = 0.01;
6 ub = 100;
7
8 % Step size and sample points
9 delta = (ub-lb) / N;
10 eta = lb:delta:ub;
11
12 % Integrat evaluated at each sample pt
13 f = eta.^(-2/3) ./ (1 + eta);
14
15 % Trapezoid Rule
16 I = 1/3 * (delta * (0.5*f(1) + sum(f(2:end-1)) + 0.5*f(end)));
17 fprintf('I approx %.6f\n', I);

```

9.2 Part 2 PINN Code Rectangular

```

1 # CodeP3Pt2B.1PINNconvSp25.ipynb - V.P. Carey, ME250B Sp25
2
3 #import useful packages
4 import keras
5 import pandas as pd
6 from keras.models import Sequential
7 import numpy as np
8 import keras.backend as kb
9 import tensorflow as tf
10 #the following 2 lines are only needed for Mac OS machines
11 import os
12 os.environ['KMP_DUPLICATE_LIB_OK']=True'
13
14 #create input data array
15
16 # x(mm), y(mm) These are (x,y) coordinates of boundary location where w velocity and T are
17 # specified
18 xdata = []
19 ND = 32
20 xdata = [ [ 0.0, 0.0 ], [ 0.0, 0.5 ], [ 0.0, 1.0 ], [ 0.0, 1.5 ], [ 0.0, 2.0 ], [ 0.0, 2.5
21 ], [ 0.0, 3.0 ],
22 [ 0.0, 3.5 ], [ 0.0, 4.0 ], [ 0.0, 4.5 ], [ 0.0, 5.0 ],
23 [ 3.0, 0.0 ], [ 3.0, 0.5 ], [ 3.0, 1.0 ], [ 3.0, 1.5 ], [ 3.0, 2.0 ], [ 3.0, 2.5
24 ], [ 3.0, 3.0 ],
25 [ 3.0, 3.5 ], [ 3.0, 4.0 ], [ 3.0, 4.5 ], [ 3.0, 5.0 ] ]
26
27 xdata.append([ 0.5, 0.0 ])
28 xdata.append([ 1.0, 0.0 ])
29 xdata.append([ 1.5, 0.0 ])
30 xdata.append([ 2.0, 0.0 ])
31 xdata.append([ 2.5, 0.0 ])
32
33 xdata.append([ 0.5, 5.0 ])
34 xdata.append([ 1.0, 5.0 ])
35 xdata.append([ 1.5, 5.0 ])
36 xdata.append([ 2.0, 5.0 ])
37 xdata.append([ 2.5, 5.0 ])

```

```

35
36 xarray= np.array(xdata)
37 print (xdata)
38 print (xarray)
39
40 # w (mm/s) These are specified velocity values (mm/s) and temperature T (deg C) for each
   boundary pair of (x,y) above
41
42 ydata = []
43
44 ydata = [ [ 0.0, 90.], [ 0.0, 90.], [ 0., 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [
   0.0, 90.], [ 0.0, 90.], [ 0.0, 90.],
   [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.],
   [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [
   0.0, 90. ], [ 0.0, 90. ] ]
45
46
47 ydata.append([ 0.0, 90. ])
48 ydata.append([ 0.0, 90. ])
49 ydata.append([ 0.0, 90. ])
50 ydata.append([ 0.0, 90. ])
51 ydata.append([ 0.0, 90. ])
52 ydata.append([ 0.0, 90. ])
53
54 ydata.append([ 0.0, 90. ])
55 ydata.append([ 0.0, 90. ])
56 ydata.append([ 0.0, 90. ])
57 ydata.append([ 0.0, 90. ])
58 ydata.append([ 0.0, 90. ])
59
60 yarray= np.array(ydata)
61 print (ydata)
62 print (yarray)
63
64 data_inputs = np.array(xdata)
65 data_outputs = np.array(ydata)
66
67 # Convert to TensorFlow tensors
68 tf_tensor_inputs = tf.convert_to_tensor(data_inputs, dtype=tf.float32)
69 tf_tensor_outputs = tf.convert_to_tensor(data_outputs, dtype=tf.float32)
70 shape_tensor = tf.shape(tf_tensor_inputs)
71 print("Shape of tf_tensor_inputs:", shape_tensor)
72 shape_tensor = tf.shape(tf_tensor_outputs)
73 print("Shape of tf_tensor_outputs:", shape_tensor)
74
75 # define neural network model
76
77 #As seen below, we have created four dense layers.
78 #A dense layer is a layer in neural network that's fully connected.
79 #In other words, all the neurons in one layer are connected to all other neurons in the
   next layer.
80 #In the first layer, we need to provide the input shape, which is 1 in our case.
81 #The activation function we have chosen is elu, which stands for exponential linear unit
   .
82
83 from tensorflow.keras.models import Sequential
84 from tensorflow.keras.layers import Dense
85
86 import tensorflow as tf
87 # Set the seed when creating the initializer
88 initializer = tf.keras.initializers.RandomUniform(minval=-2.9, maxval=2.9, seed=42)
89
90 # Create your model (for example, a simple fully connected feedforward NN)
91 # Assume 2D input (x, y), Output: z-direction velocity w
92 model = Sequential([
93     Dense(5, input_dim=2, activation='elu', kernel_initializer=initializer),
94     Dense(12, activation='elu', kernel_initializer=initializer),
95     Dense(8, activation='elu', kernel_initializer=initializer),
96     Dense(2)
97 ])

```

```

98
99 # Set the kappa value (this can be adjusted as per your needs)
100 kappa = 1.3
101
102 # Set the x_max and y_max values of your domain in mm (can adjust these according to your
103 # problem)
104 x_max = 3.0
105 y_max = 5.0
106
107 #print summary of model features
108 model.summary()
109
110 # PINN CUSTOM LOSS FUNCTION - V.P. Carey, ME250B Sp25
111
112 import tensorflow as tf
113 from tensorflow.keras import backend as K
114
115 def custom_loss_function(kappa, x_max, y_max, epoch_tracker):
116     # Custom loss function for the neural network model.
117     #- kappa: weight for the Poisson residual term.
118     #- x_max, y_max: maximum x and y coordinates of the domain in mm.
119     #- epoch_tracker: callback object to keep track of the current epoch.
120
121     def loss(y_true, y_pred):
122         #ALWAYS MATCH BOUNDARY VALUES HERE
123         # Define pointsBoundaryChk as a constant tensor
124
125         pointsBoundaryChk = tf.constant([
126             [ 3.0,  0.0 ],
127             [ 3.0,  0.5 ],
128             [ 3.0,  1.0 ],
129             [ 3.0,  1.5 ],
130             [ 3.0,  2.0 ],
131             [ 3.0,  2.5 ],
132             [ 3.0,  3.0 ],
133             [ 3.0,  3.5 ],
134             [ 3.0,  4.0 ],
135             [ 3.0,  4.5 ],
136             [ 3.0,  5.0 ],
137             [ 0.0,  0.0 ],
138             [ 0.0,  0.5 ],
139             [ 0.0,  1.0 ],
140             [ 0.0,  1.5 ],
141             [ 0.0,  2.0 ],
142             [ 0.0,  2.5 ],
143             [ 0.0,  3.0 ],
144             [ 0.0,  3.5 ],
145             [ 0.0,  4.0 ],
146             [ 0.0,  4.5 ],
147             [ 0.0,  5.0 ],
148             [ 0.5,  0.0 ],
149             [ 1.0,  0.0 ],
150             #ADD x,y PAIRS DATA HERE SO TENSOR LOOKS LIKE Fig. 3pt2B.3 IN WRITEUP - SHOULD BE 32
151             # PAIRS TOTAL
152             [ 1.5,  0.0 ],
153             [ 2.0,  0.0 ],
154             [ 2.5,  0.0 ],
155             [ 0.5,  5.0 ],
156             [ 1.0,  5.0 ],
157             [ 1.5,  5.0 ],
158             [ 2.0,  5.0 ],
159             [ 2.5,  5.0 ]
160         ], dtype=tf.float32)
161
162         # Boundary loss: Calculate squared error for the boundary condition
163         boundary_loss = 2.0*K.mean(K.square(y_pred - y_true))

```

```

164
165     #####
166
167     # 2 SECOND STEP OPTIONS DEPENDING ON epoch
168
169     # Check epoch from the callback
170     epoch = epoch_tracker.epochs_trained
171     # FOR EARLY EPOCHS
172     if epoch < epoch_tracker.N_epInit:
173
174         #NEXT FIND LOSS DUE TO MATCH OF INTERIOR POINTS TO TARGET w VELOCITY FIELD
175         # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit
176         points = tf.constant([
177             [0.5, 0.5, 16., 78.40579710144928],
178             [0.5, 1.0, 28.444445, 69.38808373590982],
179             [0.5, 1.5, 37.333332, 62.94685990338164],
180             [0.5, 2., 42.666668, 59.08212560386473],
181             [0.5, 2.5, 44.444443, 57.7938808373591],
182             [0.5, 3., 42.666668, 59.08212560386473],
183             [0.5, 3.5, 37.333332, 62.94685990338164],
184             [0.5, 4., 28.444445, 69.38808373590982],
185             [0.5, 4.5, 16., 78.40579710144928],
186             [1.0, 0.5, 25.6, 71.44927536231884],
187             [1.0, 1.0, 45.511112, 57.02093397745571],
188             [1.0, 1.5, 59.733334, 46.71497584541062],
189             [1.0, 2., 68.266667, 40.53140096618357],
190             [1.0, 2.5, 71.111115, 38.47020933977455],
191             [1.0, 3., 68.266667, 40.53140096618357],
192             [1.0, 3.5, 59.733334, 46.71497584541062],
193             [1.0, 4., 45.511112, 57.02093397745571],
194             [1.0, 4.5, 25.6, 71.44927536231884],
195             [1.5, 0.5, 28.8, 69.13043478260869],
196             [1.5, 1.0, 51.2, 52.89855072463767],
197             [1.5, 1.5, 67.2, 41.30434782608695],
198             [1.5, 2., 76.8, 34.347826086956516],
199             [1.5, 2.5, 80., 32.028985507246375],
200             [1.5, 3., 76.8, 34.347826086956516],
201             [1.5, 3.5, 67.2, 41.30434782608695],
202             [1.5, 4., 51.2, 52.89855072463767],
203             [1.5, 4.5, 28.8, 69.13043478260869],
204             [2., 0.5, 25.6, 71.44927536231884 ],
205             [2., 1.0, 45.511112, 57.02093397745571],
206             [2., 1.5, 59.733334, 46.71497584541062],
207             [2., 2., 68.266667, 40.53140096618357],
208             [2., 2.5, 71.111115, 38.47020933977455],
209             [2., 3., 68.266667, 40.53140096618357],
210             [2., 3.5, 59.733334, 46.71497584541062],
211             [2., 4., 45.511112, 57.02093397745571],
212             [2., 4.5, 25.6, 71.44927536231884],
213             [2.5, 0.5, 16., 78.40579710144928],
214             [2.5, 1.0, 28.444445, 69.38808373590982],
215             [2.5, 1.5, 37.333332, 62.94685990338164],
216             [2.5, 2., 42.666668, 59.08212560386473],
217             [2.5, 2.5, 44.444443, 57.7938808373591],
218             [2.5, 3., 42.666668, 59.08212560386473],
219             [2.5, 3.5, 37.333332, 62.94685990338164],
220             [2.5, 4., 28.444445, 69.38808373590982],
221             [2.5, 4.5, 16., 78.40579710144928]
222         ], dtype=tf.float32)
223
224         meanIntTarg_res = 0.
225         for _ in range(45):
226             # does this list 45 times not keeping track of index
227             # THIS COMPUTES INTERIOR POINT TARGET SOLUTION LOSS FUNCTION
228             # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit
229
230             # Randomly select one point
231             idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)

```

```

232     selected_point = tf.gather(points, idx)  # shape: [3]
233
234     # Slice to get only the first two columns: x and y
235     xy_input = selected_point[:2]  # shape: [2]
236
237     # Prepare input for the model: [1, 2]
238     model_input = tf.reshape(xy_input, shape=(1, 2))
239
240     #retreive w target value
241     w_targInt = selected_point[2]
242     #retreive T target value
243     T_targInt = selected_point[3]
244
245     # Compute the output
246     Predoutput = model(model_input)
247     wpred = Predoutput[0, 0]
248     Tpred = Predoutput[0, 1]
249     # add loss contributions
250     meanIntTarg_res += (wpred - w_targInt)*(wpred - w_targInt)/45.
251     meanIntTarg_res += (Tpred - T_targInt)*(Tpred - T_targInt)/45.
252
253
254     #Combine mean boundary and mean interior point target losses
255     total_loss = boundary_loss + meanIntTarg_res
256
257
258 else:
259     # DO THIS IF CURRENT epoch IS GREATER THAN N_epInit,
260     # THIS COMPUTES THE PDE RESIDUAL LOSS FUNCTION at 45 INTERIOR POINTS AND
261     # SKIPS ERROR CALC FOR TARGET GUESSED w FIELD AT INTERIOR POINTS
262     # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch > epoch_tracker.N_epInit
263
264     #>>> Randomly select an interior point from this list using TensorFlow's random
265     #      uniform
266     points2 = tf.constant([
267         [0.5, 0.5],
268         [0.5, 1.0],
269         [0.5, 1.5],
270         [0.5, 2.],
271         [0.5, 2.5],
272         [0.5, 3.],
273         [0.5, 3.5],
274         [0.5, 4.],
275         [0.5, 4.5],
276         [1.0, 0.5],
277         [1.0, 1.0],
278         [1.0, 1.5],
279         [1.0, 2.],
280         [1.0, 2.5],
281         [1.0, 3.],
282         [1.0, 3.5],
283         [1.0, 4.],
284         [1.0, 4.5],
285         [1.5, 0.5],
286         [1.5, 1.0],
287         [1.5, 1.5],
288         [1.5, 2.],
289         [1.5, 2.5],
290         [1.5, 3.],
291         [1.5, 3.5],
292         [1.5, 4.],
293         [1.5, 4.5],
294         [2., 0.5],
295         [2., 1.0],
296         [2., 1.5],
297         [2., 2.],
298         [2., 2.5],
299         [2., 3.],
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589

```

```

299         [2.,  3.5],
300         [2.,  4.],
301         [2.,  4.5],
302         [2.5, 0.5],
303         [2.5, 1.0],
304         [2.5, 1.5],
305         [2.5, 2.],
306         [2.5, 2.5],
307         [2.5, 3.],
308         [2.5, 3.5],
309         [2.5, 4.],
310         [2.5, 4.5]
311     ], dtype=tf.float32)
# THIS LIST INCLUDES ALL 45 [X,Y] POINTS IN points ARRAY IN PROJECT WRITEUP
312
313
314
315 meanPois_res_w = 0.
316 meanPois_res_T = 0.
317 #meanwlimPen = 0. #ASSOCIATED WITH PENALTY CALC COMMENTED_OUT HERE BELOW
318 for _ in range(45):
319     ##compute Laplacian/Poisson residual here - THIS LIST IS DONE ONCE PER epoch
320
321     # Use tf.random.uniform to sample an index for the interior point
322     idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)
323     selected_point = tf.gather(points2, idx)
324     #x_interior, y_interior = selected_point[0], selected_point[1]
325     #=====
326     input_point = tf.stack([selected_point[0], selected_point[1]])      # shape:
327     (2,)           input_point = tf.reshape(input_point, (1, 2))  # shape: (1, 2) for batch
328
329     # First and second derivatives - extraction from model have output of shape
330     (2)
331     with tf.GradientTape(persistent=True) as tape2:
332         tape2.watch(input_point)
333         with tf.GradientTape(persistent=True) as tape1:
334             tape1.watch(input_point)
335             output = model(input_point)      # shape: (1, 2)
336             wpred = output[0, 0]
337             Tpred = output[0, 1]
338
339             # First-order derivatives
340             dw_dx, dw_dy = tape1.gradient(wpred, input_point)[0]
341             dT_dx, dT_dy = tape1.gradient(Tpred, input_point)[0]
342
343             # Second-order derivatives
344             d2w_dx2, d2w_dy2 = tape2.gradient([dw_dx, dw_dy], input_point)[0]
345             d2T_dx2, d2T_dy2 = tape2.gradient([dT_dx, dT_dy], input_point)[0]
346
347             # Clean up
348             del tape1
349             del tape2
350             #=====
351
352
353     # Calculate the Laplacian (sum of second derivatives)
354     laplacian_w = d2w_dx2 + d2w_dy2
355     laplacian_T = d2T_dx2 + d2T_dy2
356
357     #shape_tensor = tf.shape(dw_dx)
358     #tf.print("Shape of dw_dx:", shape_tensor)
359     #shape_tensor = tf.shape(dw_dy)
360     #tf.print("Shape of dw_dy:", shape_tensor)
361
362     #ADD PRESSURE GRADIENT TERM dPmdzomu = dPdz over mu (POISSON EQ for w)
363     dPmdzomu = -80.0

```

```

364         # Compute the mean w Poisson residual for the 45 points( sum of squared (
365         # Laplacian - dPmdzomu )
366         meanPois_res_w += K.square((laplacian_w - dPmdzomu))/45.
367
368         #ADD TEMPERATURE GRADIENT TERM w*dTmdzoalpha = (w*dTdz over alpha) (POISSON
369         # EQ)
370         # dTmdzoalpha ~ sdegC/mm**3
371         dTmdzoalpha = 0.103
372         # Compute the mean T Poisson residual for the 45 points( sum of squared (
373         # Laplacian - w_pred*dTmdzoalpha )
374         meanPois_res_T += K.square((laplacian_T - wpred*dTmdzoalpha))/45.
375
376
377
378     return total_loss
379
380
381 # ====== END CUSTOM LOSS FUNCTION DEFINITION ======
382
383
384 from tensorflow.keras.callbacks import Callback
385
386 class EpochTracker(Callback):
387     def __init__(self, N_epInit):
388         super().__init__()
389         self.N_epInit = N_epInit
390         self.epochs_trained = 0
391
392     def on_epoch_end(self, epoch, logs=None):
393         self.epochs_trained = epoch
394
395 # Set the loss function with the required arguments
396 # Set N_epInit=2000 for first run of fit routine in next cell,
397 # then set it then set N_epInit=1 for all runs of fit routine beyond that
398
399 epoch_tracker = EpochTracker(N_epInit=1)
400
401 custom_loss = custom_loss_function(kappa=1.3, x_max=3.0, y_max=5.0, epoch_tracker=
402                                         epoch_tracker)
403
404
405 # Compile the model with the custom loss
406 model.compile(optimizer='adam', loss=custom_loss)
407
408 ##### After the compilation of the model, run the fit method with some number of epochs (can
409 # be run iteratively.
410
411
412 #The fit method takes three parameters; namely, x, y, and number of epochs.
413 #During model training, if all the batches of data are seen by the model once,
414 #we say that one epoch has been completed.
415
416 ###### RUN FIT
417
418 # Assuming 'boundary_points' and 'boundary_w' are TensorFlow tensors
419
420 # Now, fit the model and pass the callback to 'fit()'
421 #historyData = model.fit(boundary_points_xtrain, boundary_w_ytrain,
422 ##             epochs=2, batch_size=32,
423 ##             callbacks=[epoch_tracker])
424 historyData = model.fit(tf_tensor_inputs, tf_tensor_outputs,
425                         epochs=2500, batch_size=32,
426                         callbacks=[epoch_tracker])

```

```

427
428
429 loss_hist = historyData.history['loss']
430 #The above line will return a dictionary, access it's info like this:
431 best_epoch = np.argmin(historyData.history['loss']) + 1
432 print ('best epoch = ', best_epoch)
433 print ('smallest loss = ', np.min(loss_hist))
434
435 predictions = model.predict(data_inputs)
436
437 predictions = model.predict(data_inputs)
438
439 #SETTING UP PLOT
440 %matplotlib inline
441 # importing the required module
442 import matplotlib.pyplot as plt
443 plt.rcParams['figure.figsize'] = [8, 8] # for square canvas
444 =====
445
446 '''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''
447
448 #CAN'T USE THIS LOG-LOG FOR W BECUASE DATA VALUES ARE ALL ZERO
449
450 plt.scatter(predictions[:, 0], data_outputs[:, 0])
451 #plt.title('w,T PINN training of Neural Network ==> y = w')
452 plt.xlabel('predicted output for NN (mm/s)')
453 plt.ylabel('data output (mm/s)')
454 #plt.loglog()
455 #plt.xlim(xmax = 100, xmin = 1.)
456 #plt.ylim(ymax = 100, ymin = 1.)
457 # Generate red y=x line
458 #x_data = np.linspace(1.0, 100.0, num=3)
459
460 plt.xlim(xmax = 10, xmin = -10.)
461 plt.ylim(ymax = 10, ymin = -10.)
462 # Generate red y=x line
463 #x_data = np.linspace(1.0, 100.0, num=3)
464 x_data = np.linspace(-10., 10.0, num=3)
465 y_data = x_data
466 plt.plot(x_data, y_data, color='red')
467 plt.show()
468 #PLOTS VELOCITY BOUNDARY DATA ONLY
469
470 # w velocity field
471 test = []
472 outpt=[]
473 predictions = model.predict(data_inputs)
474
475 #first point (row [0])comparison of data and prediction
476 # put in a loop to print comparion for all data points
477
478 xa=[]
479 ya=[]
480 za=[]
481
482 xao = np.linspace(0, 3, 7) # 1D array for x values
483 yao = np.linspace(0, 5, 11) # 1D array for y values
484 xa = np.linspace(0, 10, 77) # 1D array for x values
485 ya = np.linspace(0, 10, 77) # 1D array for y values
486 za = np.linspace(0, 10, 77) # 1D array for z values
487
488 x = -0.50
489 ia=-1
490 while (x < 2.51):
491     x = x + 0.5
492     y = -0.5
493     while (y < 4.51):
494         ia=ia+1

```

```

495     y = y + 0.50
496     xa[ia]=x
497     ya[ia]=y
498
499     test = [[ x , y ]]
500     testarray = np.array(test)
501     outpt = model.predict(testarray)
502     za[ia] = outpt[0][0]
503     print ('x= ', testarray[0][0], ', y= ', testarray[0][1], 'w= ', outpt[0][0])
504     #Predoutput[0, 0] ???
505
506
507 X, Y, Z = xa, ya, za
508
509 import sys
510 #import csv
511 import numpy as np
512 import matplotlib.pyplot as plt
513 from mpl_toolkits.mplot3d import axes3d
514
515 # Plot X,Y,Z
516
517 # Creating figure
518 #fig = plt.figure(figsize =(16, 9))
519 #ax = plt.axes(projection ='3d')
520
521 fig = plt.figure()
522 #ax = fig.add_subplot(111, projection='3d')
523 ax = plt.axes(projection ='3d')
524 #ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
525 ax.plot_trisurf(X, Y, Z,
526                  linewidth = 0.2,
527                  antialiased = True
528                  , cmap='viridis');
529 ax.invert_yaxis()
530 ax.invert_xaxis()
531 plt.show()
532
533
534
535 # Plot the data
536 plt.figure(figsize=(5, 6))
537 plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
538 plt.colorbar(label='w value')
539 plt.xlabel('x')
540 plt.ylabel('y')
541 plt.title('Heat Map of w values (mm/s) over x and y domain (mm)')
542 plt.show()
543
544 #temperature field T
545 test = []
546 outpt=[]
547 predictions = model.predict(data_inputs)
548
549 #first point (row [0])comparison of data and prediction
550 # put in a loop to print comparion for all data points
551
552 xa=[]
553 ya=[]
554 za=[]
555
556 xao = np.linspace(0, 3, 7) # 1D array for x values
557 yao = np.linspace(0, 5, 11) # 1D array for y values
558 xa = np.linspace(0, 10, 77) # 1D array for x values
559 ya = np.linspace(0, 10, 77) # 1D array for y values
560 za = np.linspace(0, 10, 77) # 1D array for z values
561
562 x = -0.50

```

```

563 ia=-1
564 while (x < 2.51):
565     x = x + 0.5
566     y = -0.5
567     while (y < 4.51):
568         ia=ia+1
569         y = y + 0.50
570         xa[ia]=x
571         ya[ia]=y
572
573     test = [[ x , y ]]
574     testarray = np.array(test)
575     outpt = model.predict(testarray)
576     za[ia] = outpt[0][1]
577     print ('x (mm) = ', testarray[0][0], ', y (mm) = ', testarray[0][1], 'T (degC) = ',
578           outpt[0][1])
579     #Predoutput[0, 0] ???
580
581 X, Y, Z = xa, ya, za
582
583 import sys
584 #import csv
585 import numpy as np
586 import matplotlib.pyplot as plt
587 from mpl_toolkits.mplot3d import axes3d
588
589 # Plot X,Y,Z
590
591 # Creating figure
592 #fig = plt.figure(figsize =(16, 9))
593 #ax = plt.axes(projection ='3d')
594
595 fig = plt.figure()
596 #ax = fig.add_subplot(111, projection='3d')
597 ax = plt.axes(projection ='3d')
598 #ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
599 ax.plot_trisurf(X, Y, Z,
600                  linewidth = 0.2,
601                  antialiased = True
602                  , cmap='viridis');
603 ax.invert_yaxis()
604 ax.invert_xaxis()
605 plt.show()
606
607
608
609 # Plot the data
610 plt.figure(figsize=(5, 6))
611 plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
612 plt.colorbar(label='T value')
613 plt.xlabel('x')
614 plt.ylabel('y')
615 plt.title('Heat Map of T values (degC) over x and y domain (mm)')
616 plt.show()

```

9.3 Part 2 PINN Code Rounded

```

1 # CodeP3Pt2B.1PINNconvSp25.ipynb - V.P. Carey, ME250B Sp25
2
3 #import useful packages
4 import keras
5 import pandas as pd
6 from keras.models import Sequential
7 import numpy as np
8 import keras.backend as kb

```

```

9 import tensorflow as tf
10 #the following 2 lines are only needed for Mac OS machines
11 import os
12 os.environ['KMP_DUPLICATE_LIB_OK']= 'True'
13
14 #create input data array
15
16 # x(mm), y(mm) These are (x,y) coordinates of boundary location where w velocity and T are
17 # specified
18 xdata = []
19 ND = 32
20 xdata = [ [0.293, 0.293], [0.134, 0.500], [ 0.0, 1.0 ], [ 0.0, 1.5 ],[ 0.0, 2.0 ], [ 0.0,
21 2.5 ], [ 0.0, 3.0 ],
22 [ 0.0, 3.5 ], [ 0.0, 4.0 ], [0.134, 4.500], [0.293, 4.707],
23 [2.707, 0.293], [2.866, 0.500], [ 3.0, 1.0 ], [ 3.0, 1.5 ],[ 3.0, 2.0 ], [ 3.0,
2.5 ], [ 3.0, 3.0 ],
24 [ 3.0, 3.5 ], [ 3.0, 4.0 ], [2.866, 4.500], [2.707, 4.707]]
25
26 xdata.append([0.500, 0.134])
27 xdata.append([ 1.0, 0.0 ])
28 xdata.append([ 1.5, 0.0 ])
29 xdata.append([ 2.0, 0.0 ])
30 xdata.append([2.500, 0.134])
31
32 xdata.append([0.500, 4.866])
33 xdata.append([ 1.0, 5.0 ])
34 xdata.append([ 1.5, 5.0 ])
35 xdata.append([ 2.0, 5.0 ])
36 xdata.append([2.500, 4.866])
37
38 xarray= np.array(xdata)
39 print (xdata)
40 print (xarray)
41
42 # w (mm/s) These are specified velocity values (mm/s) and temperature T (deg C) for each
43 # boundary pair of (x,y) above
44
45 ydata = []
46
47 ydata = [ [ 0.0, 90.], [ 0.0, 90.], [ 0., 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [
48 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.],
49 [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0 , 90.],
50 [ 0.0, 90.], [ 0.0, 90.], [ 0.0 , 90.], [ 0.0, 90.], [ 0.0, 90.], [ 0.0, 90.], [
51 0.0, 90. ], [ 0.0, 90. ] ]
52
53 ydata.append([ 0.0, 90. ])
54 ydata.append([ 0.0, 90. ])
55 ydata.append([ 0.0, 90. ])
56 ydata.append([ 0.0, 90. ])
57 ydata.append([ 0.0, 90. ])
58 ydata.append([ 0.0, 90. ])
59 ydata.append([ 0.0, 90. ])
60
61
62 yarray= np.array(ydata)
63 print (ydata)
64 print (yarray)
65
66 data_inputs = np.array(xdata)
67 data_outputs = np.array(ydata)
68
69
70 # Convert to TensorFlow tensors

```

```

71 tf_tensor_inputs = tf.convert_to_tensor(data_inputs, dtype=tf.float32)
72 tf_tensor_outputs = tf.convert_to_tensor(data_outputs, dtype=tf.float32)
73 shape_tensor = tf.shape(tf_tensor_inputs)
74 print("Shape of tf_tensor_inputs:", shape_tensor)
75 shape_tensor = tf.shape(tf_tensor_outputs)
76 print("Shape of tf_tensor_outputs:", shape_tensor)
77
78 # define neural network model
79
80     #As seen below, we have created four dense layers.
81     #A dense layer is a layer in neural network that's fully connected.
82     #In other words, all the neurons in one layer are connected to all other neurons in the
83     #next layer.
84     #In the first layer, we need to provide the input shape, which is 1 in our case.
85     #The activation function we have chosen is elu, which stands for exponential linear unit
86
87
88
89 from tensorflow.keras.models import Sequential
90 from tensorflow.keras.layers import Dense
91
92
93 import tensorflow as tf
94 # Set the seed when creating the initializer
95 initializer = tf.keras.initializers.RandomUniform(minval=-2.9, maxval=2.9, seed=42)
96
97
98 # Create your model (for example, a simple fully connected feedforward NN)
99 # Assume 2D input (x, y), Output: z-direction velocity w
100 model = Sequential([
101     Dense(5, input_dim=2, activation='elu', kernel_initializer=initializer),
102     Dense(12, activation='elu', kernel_initializer=initializer),
103     Dense(8, activation='elu', kernel_initializer=initializer),
104     Dense(2)
105 ])
106
107 # Set the kappa value (this can be adjusted as per your needs)
108 kappa = 1.3
109
110
111 # Set the x_max and y_max values of your domain in mm (can adjust these according to your
112 # problem)
113 x_max = 3.0
114 y_max = 5.0
115
116
117 #print summary of model features
118 model.summary()
119
120
121 # PINN CUSTOM LOSS FUNCTION - V.P. Carey, ME250B Sp25
122
123
124 import tensorflow as tf
125 from tensorflow.keras import backend as K
126
127
128 def custom_loss_function(kappa, x_max, y_max, epoch_tracker):
129     # Custom loss function for the neural network model.
130     #- kappa: weight for the Poisson residual term.
131     #- x_max, y_max: maximum x and y coordinates of the domain in mm.
132     #- epoch_tracker: callback object to keep track of the current epoch.
133
134     def loss(y_true, y_pred):
135         #ALWAYS MATCH BOUNDARY VALUES HERE
136         # Define pointsBoundaryChk as a constant tensor
137
138         pointsBoundaryChk = tf.constant([
139             [2.707, 0.293],
140             [2.866, 0.500],
141             [ 3.0,  1.0 ],
142             [ 3.0,  1.5 ],
143             [ 3.0,  2.0 ],
144             [ 3.0,  2.5 ],
145             [ 3.0,  3.0 ],
146             [ 3.0,  3.5 ],
147             [ 3.0,  4.0 ],
148             [ 3.0,  4.5 ],
149             [ 3.0,  5.0 ],
150             [ 3.0,  5.5 ],
151             [ 3.0,  6.0 ],
152             [ 3.0,  6.5 ],
153             [ 3.0,  7.0 ],
154             [ 3.0,  7.5 ],
155             [ 3.0,  8.0 ],
156             [ 3.0,  8.5 ],
157             [ 3.0,  9.0 ],
158             [ 3.0,  9.5 ],
159             [ 3.0, 10.0 ],
160             [ 3.0, 10.5 ],
161             [ 3.0, 11.0 ],
162             [ 3.0, 11.5 ],
163             [ 3.0, 12.0 ],
164             [ 3.0, 12.5 ],
165             [ 3.0, 13.0 ],
166             [ 3.0, 13.5 ],
167             [ 3.0, 14.0 ],
168             [ 3.0, 14.5 ],
169             [ 3.0, 15.0 ],
170             [ 3.0, 15.5 ],
171             [ 3.0, 16.0 ],
172             [ 3.0, 16.5 ],
173             [ 3.0, 17.0 ],
174             [ 3.0, 17.5 ],
175             [ 3.0, 18.0 ],
176             [ 3.0, 18.5 ],
177             [ 3.0, 19.0 ],
178             [ 3.0, 19.5 ],
179             [ 3.0, 20.0 ],
180             [ 3.0, 20.5 ],
181             [ 3.0, 21.0 ],
182             [ 3.0, 21.5 ],
183             [ 3.0, 22.0 ],
184             [ 3.0, 22.5 ],
185             [ 3.0, 23.0 ],
186             [ 3.0, 23.5 ],
187             [ 3.0, 24.0 ],
188             [ 3.0, 24.5 ],
189             [ 3.0, 25.0 ],
190             [ 3.0, 25.5 ],
191             [ 3.0, 26.0 ],
192             [ 3.0, 26.5 ],
193             [ 3.0, 27.0 ],
194             [ 3.0, 27.5 ],
195             [ 3.0, 28.0 ],
196             [ 3.0, 28.5 ],
197             [ 3.0, 29.0 ],
198             [ 3.0, 29.5 ],
199             [ 3.0, 30.0 ],
200             [ 3.0, 30.5 ],
201             [ 3.0, 31.0 ],
202             [ 3.0, 31.5 ],
203             [ 3.0, 32.0 ],
204             [ 3.0, 32.5 ],
205             [ 3.0, 33.0 ],
206             [ 3.0, 33.5 ],
207             [ 3.0, 34.0 ],
208             [ 3.0, 34.5 ],
209             [ 3.0, 35.0 ],
210             [ 3.0, 35.5 ],
211             [ 3.0, 36.0 ],
212             [ 3.0, 36.5 ],
213             [ 3.0, 37.0 ],
214             [ 3.0, 37.5 ],
215             [ 3.0, 38.0 ],
216             [ 3.0, 38.5 ],
217             [ 3.0, 39.0 ],
218             [ 3.0, 39.5 ],
219             [ 3.0, 40.0 ],
220             [ 3.0, 40.5 ],
221             [ 3.0, 41.0 ],
222             [ 3.0, 41.5 ],
223             [ 3.0, 42.0 ],
224             [ 3.0, 42.5 ],
225             [ 3.0, 43.0 ],
226             [ 3.0, 43.5 ],
227             [ 3.0, 44.0 ],
228             [ 3.0, 44.5 ],
229             [ 3.0, 45.0 ],
230             [ 3.0, 45.5 ],
231             [ 3.0, 46.0 ],
232             [ 3.0, 46.5 ],
233             [ 3.0, 47.0 ],
234             [ 3.0, 47.5 ],
235             [ 3.0, 48.0 ],
236             [ 3.0, 48.5 ],
237             [ 3.0, 49.0 ],
238             [ 3.0, 49.5 ],
239             [ 3.0, 50.0 ],
240             [ 3.0, 50.5 ],
241             [ 3.0, 51.0 ],
242             [ 3.0, 51.5 ],
243             [ 3.0, 52.0 ],
244             [ 3.0, 52.5 ],
245             [ 3.0, 53.0 ],
246             [ 3.0, 53.5 ],
247             [ 3.0, 54.0 ],
248             [ 3.0, 54.5 ],
249             [ 3.0, 55.0 ],
250             [ 3.0, 55.5 ],
251             [ 3.0, 56.0 ],
252             [ 3.0, 56.5 ],
253             [ 3.0, 57.0 ],
254             [ 3.0, 57.5 ],
255             [ 3.0, 58.0 ],
256             [ 3.0, 58.5 ],
257             [ 3.0, 59.0 ],
258             [ 3.0, 59.5 ],
259             [ 3.0, 60.0 ],
260             [ 3.0, 60.5 ],
261             [ 3.0, 61.0 ],
262             [ 3.0, 61.5 ],
263             [ 3.0, 62.0 ],
264             [ 3.0, 62.5 ],
265             [ 3.0, 63.0 ],
266             [ 3.0, 63.5 ],
267             [ 3.0, 64.0 ],
268             [ 3.0, 64.5 ],
269             [ 3.0, 65.0 ],
270             [ 3.0, 65.5 ],
271             [ 3.0, 66.0 ],
272             [ 3.0, 66.5 ],
273             [ 3.0, 67.0 ],
274             [ 3.0, 67.5 ],
275             [ 3.0, 68.0 ],
276             [ 3.0, 68.5 ],
277             [ 3.0, 69.0 ],
278             [ 3.0, 69.5 ],
279             [ 3.0, 70.0 ],
280             [ 3.0, 70.5 ],
281             [ 3.0, 71.0 ],
282             [ 3.0, 71.5 ],
283             [ 3.0, 72.0 ],
284             [ 3.0, 72.5 ],
285             [ 3.0, 73.0 ],
286             [ 3.0, 73.5 ],
287             [ 3.0, 74.0 ],
288             [ 3.0, 74.5 ],
289             [ 3.0, 75.0 ],
290             [ 3.0, 75.5 ],
291             [ 3.0, 76.0 ],
292             [ 3.0, 76.5 ],
293             [ 3.0, 77.0 ],
294             [ 3.0, 77.5 ],
295             [ 3.0, 78.0 ],
296             [ 3.0, 78.5 ],
297             [ 3.0, 79.0 ],
298             [ 3.0, 79.5 ],
299             [ 3.0, 80.0 ],
200             [ 3.0, 80.5 ],
201             [ 3.0, 81.0 ],
202             [ 3.0, 81.5 ],
203             [ 3.0, 82.0 ],
204             [ 3.0, 82.5 ],
205             [ 3.0, 83.0 ],
206             [ 3.0, 83.5 ],
207             [ 3.0, 84.0 ],
208             [ 3.0, 84.5 ],
209             [ 3.0, 85.0 ],
210             [ 3.0, 85.5 ],
211             [ 3.0, 86.0 ],
212             [ 3.0, 86.5 ],
213             [ 3.0, 87.0 ],
214             [ 3.0, 87.5 ],
215             [ 3.0, 88.0 ],
216             [ 3.0, 88.5 ],
217             [ 3.0, 89.0 ],
218             [ 3.0, 89.5 ],
219             [ 3.0, 90.0 ],
220             [ 3.0, 90.5 ],
221             [ 3.0, 91.0 ],
222             [ 3.0, 91.5 ],
223             [ 3.0, 92.0 ],
224             [ 3.0, 92.5 ],
225             [ 3.0, 93.0 ],
226             [ 3.0, 93.5 ],
227             [ 3.0, 94.0 ],
228             [ 3.0, 94.5 ],
229             [ 3.0, 95.0 ],
220             [ 3.0, 95.5 ],
221             [ 3.0, 96.0 ],
222             [ 3.0, 96.5 ],
223             [ 3.0, 97.0 ],
224             [ 3.0, 97.5 ],
225             [ 3.0, 98.0 ],
226             [ 3.0, 98.5 ],
227             [ 3.0, 99.0 ],
228             [ 3.0, 99.5 ],
229             [ 3.0, 100.0 ],
230             [ 3.0, 100.5 ],
231             [ 3.0, 101.0 ],
232             [ 3.0, 101.5 ],
233             [ 3.0, 102.0 ],
234             [ 3.0, 102.5 ],
235             [ 3.0, 103.0 ],
236             [ 3.0, 103.5 ],
237             [ 3.0, 104.0 ],
238             [ 3.0, 104.5 ],
239             [ 3.0, 105.0 ],
230             [ 3.0, 105.5 ],
231             [ 3.0, 106.0 ],
232             [ 3.0, 106.5 ],
233             [ 3.0, 107.0 ],
234             [ 3.0, 107.5 ],
235             [ 3.0, 108.0 ],
236             [ 3.0, 108.5 ],
237             [ 3.0, 109.0 ],
238             [ 3.0, 109.5 ],
239             [ 3.0, 110.0 ],
230             [ 3.0, 110.5 ],
231             [ 3.0, 111.0 ],
232             [ 3.0, 111.5 ],
233             [ 3.0, 112.0 ],
234             [ 3.0, 112.5 ],
235             [ 3.0, 113.0 ],
236             [ 3.0, 113.5 ],
237             [ 3.0, 114.0 ],
238             [ 3.0, 114.5 ],
239             [ 3.0, 115.0 ],
230             [ 3.0, 115.5 ],
231             [ 3.0, 116.0 ],
232             [ 3.0, 116.5 ],
233             [ 3.0, 117.0 ],
234             [ 3.0, 117.5 ],
235             [ 3.0, 118.0 ],
236             [ 3.0, 118.5 ],
237             [ 3.0, 119.0 ],
238             [ 3.0, 119.5 ],
239             [ 3.0, 120.0 ],
230             [ 3.0, 120.5 ],
231             [ 3.0, 121.0 ],
232             [ 3.0, 121.5 ],
233             [ 3.0, 122.0 ],
234             [ 3.0, 122.5 ],
235             [ 3.0, 123.0 ],
236             [ 3.0, 123.5 ],
237             [ 3.0, 124.0 ],
238             [ 3.0, 124.5 ],
239             [ 3.0, 125.0 ],
230             [ 3.0, 125.5 ],
231             [ 3.0, 126.0 ],
232             [ 3.0, 126.5 ],
233             [ 3.0, 127.0 ],
234             [ 3.0, 127.5 ],
235             [ 3.0, 128.0 ],
236             [ 3.0, 128.5 ],
237             [ 3.0, 129.0 ],
238             [ 3.0, 129.5 ],
239             [ 3.0, 130.0 ],
230             [ 3.0, 130.5 ],
231             [ 3.0, 131.0 ],
232             [ 3.0, 131.5 ],
233             [ 3.0, 132.0 ],
234             [ 3.0, 132.5 ],
235             [ 3.0, 133.0 ],
236             [ 3.0, 133.5 ],
237             [ 3.0, 134.0 ],
238             [ 3.0, 134.5 ],
239             [ 3.0, 135.0 ],
230             [ 3.0, 135.5 ],
231             [ 3.0, 136.0 ],
232             [ 3.0, 136.5 ],
233             [ 3.0, 137.0 ],
234             [ 3.0, 137.5 ],
235             [ 3.0, 138.0 ],
236             [ 3.0, 138.5 ],
237             [ 3.0, 139.0 ],
238             [ 3.0, 139.5 ],
239             [ 3.0, 140.0 ],
230             [ 3.0, 140.5 ],
231             [ 3.0, 141.0 ],
232             [ 3.0, 141.5 ],
233             [ 3.0, 142.0 ],
234             [ 3.0, 142.5 ],
235             [ 3.0, 143.0 ],
236             [ 3.0, 143.5 ],
237             [ 3.0, 144.0 ],
238             [ 3.0, 144.5 ],
239             [ 3.0, 145.0 ],
230             [ 3.0, 145.5 ],
231             [ 3.0, 146.0 ],
232             [ 3.0, 146.5 ],
233             [ 3.0, 147.0 ],
234             [ 3.0, 147.5 ],
235             [ 3.0, 148.0 ],
236             [ 3.0, 148.5 ],
237             [ 3.0, 149.0 ],
238             [ 3.0, 149.5 ],
239             [ 3.0, 150.0 ],
230             [ 3.0, 150.5 ],
231             [ 3.0, 151.0 ],
232             [ 3.0, 151.5 ],
233             [ 3.0, 152.0 ],
234             [ 3.0, 152.5 ],
235             [ 3.0, 153.0 ],
236             [ 3.0, 153.5 ],
237             [ 3.0, 154.0 ],
238             [ 3.0, 154.5 ],
239             [ 3.0, 155.0 ],
230             [ 3.0, 155.5 ],
231             [ 3.0, 156.0 ],
232             [ 3.0, 156.5 ],
233             [ 3.0, 157.0 ],
234             [ 3.0, 157.5 ],
235             [ 3.0, 158.0 ],
236             [ 3.0, 158.5 ],
237             [ 3.0, 159.0 ],
238             [ 3.0, 159.5 ],
239             [ 3.0, 160.0 ],
230             [ 3.0, 160.5 ],
231             [ 3.0, 161.0 ],
232             [ 3.0, 161.5 ],
233             [ 3.0, 162.0 ],
234             [ 3.0, 162.5 ],
235             [ 3.0, 163.0 ],
236             [ 3.0, 163.5 ],
237             [ 3.0, 164.0 ],
238             [ 3.0, 164.5 ],
239             [ 3.0, 165.0 ],
230             [ 3.0, 165.5 ],
231             [ 3.0, 166.0 ],
232             [ 3.0, 166.5 ],
233             [ 3.0, 167.0 ],
234             [ 3.0, 167.5 ],
235             [ 3.0, 168.0 ],
236             [ 3.0, 168.5 ],
237             [ 3.0, 169.0 ],
238             [ 3.0, 169.5 ],
239             [ 3.0, 170.0 ],
230             [ 3.0, 170.5 ],
231             [ 3.0, 171.0 ],
232             [ 3.0, 171.5 ],
233             [ 3.0, 172.0 ],
234             [ 3.0, 172.5 ],
235             [ 3.0, 173.0 ],
236             [ 3.0, 173.5 ],
237             [ 3.0, 174.0 ],
238             [ 3.0, 174.5 ],
239             [ 3.0, 175.0 ],
230             [ 3.0, 175.5 ],
231             [ 3.0, 176.0 ],
232             [ 3.0, 176.5 ],
233             [ 3.0, 177.0 ],
234             [ 3.0, 177.5 ],
235             [ 3.0, 178.0 ],
236             [ 3.0, 178.5 ],
237             [ 3.0, 179.0 ],
238             [ 3.0, 179.5 ],
239             [ 3.0, 180.0 ],
230             [ 3.0, 180.5 ],
231             [ 3.0, 181.0 ],
232             [ 3.0, 181.5 ],
233             [ 3.0, 182.0 ],
234             [ 3.0, 182.5 ],
235             [ 3.0, 183.0 ],
236             [ 3.0, 183.5 ],
237             [ 3.0, 184.0 ],
238             [ 3.0, 184.5 ],
239             [ 3.0, 185.0 ],
230             [ 3.0, 185.5 ],
231             [ 3.0, 186.0 ],
232             [ 3.0, 186.5 ],
233             [ 3.0, 187.0 ],
234             [ 3.0, 187.5 ],
235             [ 3.0, 188.0 ],
236             [ 3.0, 188.5 ],
237             [ 3.0, 189.0 ],
238             [ 3.0, 189.5 ],
239             [ 3.0, 190.0 ],
230             [ 3.0, 190.5 ],
231             [ 3.0, 191.0 ],
232             [ 3.0, 191.5 ],
233             [ 3.0, 192.0 ],
234             [ 3.0, 192.5 ],
235             [ 3.0, 193.0 ],
236             [ 3.0, 193.5 ],
237             [ 3.0, 194.0 ],
238             [ 3.0, 194.5 ],
239             [ 3.0, 195.0 ],
230             [ 3.0, 195.5 ],
231             [ 3.0, 196.0 ],
232             [ 3.0, 196.5 ],
233             [ 3.0, 197.0 ],
234             [ 3.0, 197.5 ],
235             [ 3.0, 198.0 ],
236             [ 3.0, 198.5 ],
237             [ 3.0, 199.0 ],
238             [ 3.0, 199.5 ],
239             [ 3.0, 200.0 ],
230             [ 3.0, 200.5 ],
231             [ 3.0, 201.0 ],
232             [ 3.0, 201.5 ],
233             [ 3.0, 202.0 ],
234             [ 3.0, 202.5 ],
235             [ 3.0, 203.0 ],
236             [ 3.0, 203.5 ],
237             [ 3.0, 204.0 ],
238             [ 3.0, 204.5 ],
239             [ 3.0, 205.0 ],
230             [ 3.0, 205.5 ],
231             [ 3.0, 206.0 ],
232             [ 3.0, 206.5 ],
233             [ 3.0, 207.0 ],
234             [ 3.0, 207.5 ],
235             [ 3.0, 208.0 ],
236             [ 3.0, 208.5 ],
237             [ 3.0, 209.0 ],
238             [ 3.0, 209.5 ],
239             [ 3.0, 210.0 ],
230             [ 3.0, 210.5 ],
231             [ 3.0, 211.0 ],
232             [ 3.0, 211.5 ],
233             [ 3.0, 212.0 ],
234             [ 3.0, 212.5 ],
235             [ 3.0, 213.0 ],
236             [ 3.0, 213.5 ],
237             [ 3.0, 214.0 ],
238             [ 3.0, 214.5 ],
239             [ 3.0, 215.0 ],
230             [ 3.0, 215.5 ],
231             [ 3.0, 216.0 ],
232             [ 3.0, 216.5 ],
233             [ 3.0, 217.0 ],
234             [ 3.0, 217.5 ],
235             [ 3.0, 218.0 ],
236             [ 3.0, 218.5 ],
237             [ 3.0, 219.0 ],
238             [ 3.0, 219.5 ],
239             [ 3.0, 220.0 ],
230             [ 3.0, 220.5 ],
231             [ 3.0, 221.0 ],
232             [ 3.0, 221.5 ],
233             [ 3.0, 222.0 ],
234             [ 3.0, 222.5 ],
235             [ 3.0, 223.0 ],
236             [ 3.0, 223.5 ],
237             [ 3.0, 224.0 ],
238             [ 3.0, 224.5 ],
239             [ 3.0, 225.0 ],
230             [ 3.0, 225.5 ],
231             [ 3.0, 226.0 ],
232             [ 3.0, 226.5 ],
233             [ 3.0, 227.0 ],
234             [ 3.0, 227.5 ],
235             [ 3.0, 228.0 ],
236             [ 3.0, 228.5 ],
237             [ 3.0, 229.0 ],
238             [ 3.0, 229.5 ],
239             [ 3.0, 230.0 ],
230             [ 3.0, 230.5 ],
231             [ 3.0, 231.0 ],
232             [ 3.0, 231.5 ],
233             [ 3.0, 232.0 ],
234             [ 3.0, 232.5 ],
235             [ 3.0, 233.0 ],
236             [ 3.0, 233.5 ],
237             [ 3.0, 234.0 ],
238             [ 3.0, 234.5 ],
239             [ 3.0, 235.0 ],
230             [ 3.0, 235.5 ],
231             [ 3.0, 236.0 ],
232             [ 3.0, 236.5 ],
233             [ 3.0, 237.0 ],
234             [ 3.0, 237.5 ],
235             [ 3.0, 238.0 ],
236             [ 3.0, 238.5 ],
237             [ 3.0, 239.0 ],
238             [ 3.0, 239.5 ],
239             [ 3.0, 240.0 ],
230             [ 3.0, 240.5 ],
231             [ 3.0, 241.0 ],
232             [ 3.0, 241.5 ],
233             [ 3.0, 242.0 ],
234             [ 3.0, 242.5 ],
235             [ 3.0, 243.0 ],
236             [ 3.0, 243.5 ],
237             [ 3.0, 244.0 ],
238             [ 3.0, 244.5 ],
239             [ 3.0, 245.0 ],
230             [ 3.0, 245.5 ],
231             [ 3.0, 246.0 ],
232             [ 3.0, 246.5 ],
233             [ 3.0, 247.0 ],
234             [ 3.0, 247.5 ],
235             [ 3.0, 248.0 ],
236             [ 3.0, 248.5 ],
237             [ 3.0, 249.0 ],
238             [ 3.0, 249.5 ],
239             [ 3.0, 250.0 ],
230             [ 3.0, 250.5 ],
231             [ 3.0, 251.0 ],
232             [ 3.0, 251.5 ],
233             [ 3.0, 252.0 ],
234             [ 3.0, 252.5 ],
235             [ 3.0, 253.0 ],
236             [ 3.0, 253.5 ],
237             [ 3.0, 254.0 ],
238             [ 3.0, 254.5 ],
239             [ 3.0, 255.0 ],
230             [ 3.0, 255.5 ],
231             [ 3.0, 256.0 ],
232             [ 3.0, 256.5 ],
233             [ 3.0, 257.0 ],
234             [ 3.0, 257.5 ],
235             [ 3.0, 258.0 ],
236             [ 3.0, 258.5 ],
237             [ 3.0, 259.0 ],
238             [ 3.0, 259.5 ],
239             [ 3.0, 260.0 ],
230             [ 3.0, 260.5 ],
231             [ 3.0, 261.0 ],
232             [ 3.0, 261.5 ],
233             [ 3.0, 262.0 ],
234             [ 3.0, 262.5 ],
235             [ 3.0, 263.0 ],
236             [ 3.0, 263.5 ],
237             [ 3.0, 264.0 ],
238             [ 3.0, 264.5 ],
239             [ 3.0, 265.0 ],
230             [ 3.0, 265.5 ],
231             [ 3.0, 266.0 ],
232             [ 3.0, 266.5 ],
233             [ 3.0, 267.0 ],
234             [ 3.0, 267.5 ],
235             [ 3.0, 268.0 ],
236             [ 3.0, 268.5 ],
237             [ 3.0, 269.0 ],
238             [ 3.0, 269.5 ],
239             [ 3.0, 270.0 ],
230             [ 3.0, 270.5 ],
231             [ 3.0, 271.0 ],
232             [ 3.0, 271.5 ],
233             [ 3.0, 272.0 ],
234             [ 3.0, 272.5 ],
235             [ 3.0, 273.0 ],
236             [ 3.0, 273.5 ],
237             [ 3.0, 274.0 ],
238             [ 3.0, 274.5 ],
239             [ 3.0, 275.0 ],
230             [ 3.0, 275.5 ],
231             [ 3.0, 276.0 ],
232             [ 3.0, 276.5 ],
233             [ 3.0, 277.0 ],
234             [ 3.0, 277.5 ],
235             [ 3.0, 278.0 ],
236             [ 3.0, 278.5 ],
237             [ 3.0, 279.0 ],
238             [ 3.0, 279.5 ],
239             [ 3.0, 280.0 ],
230             [ 3.0, 280.5 ],
231             [ 3.0, 281.0 ],
232             [ 3.0, 281.5 ],
233             [ 3.0, 282.0 ],
234             [ 3.0, 282.5 ],
235             [ 3.0, 283.0 ],
236             [ 3.0, 283.5 ],
237             [ 3.0, 284.0 ],
238             [ 3.0, 284.5 ],
239             [ 3.0, 285.0 ],
230             [ 3.0, 285.5 ],
231             [ 3.0, 286.0 ],
232             [ 3.0, 286.5 ],
233             [ 3.0, 287.0 ],
234             [ 3.0, 287.5 ],
235             [ 3.0, 288.0 ],
236             [ 3.0, 288.5 ],
237             [ 3.0, 289.0 ],
238             [ 3.0, 289.5 ],
239             [ 3.0, 290.0 ],
230             [ 3.0, 290.5 ],
231             [ 3.0, 291.0 ],
232             [ 3.0, 291.5 ],
233             [ 3.0, 292.0 ],
234             [ 3.0, 292.5 ],
235             [ 3.0, 293.0 ],
236             [ 3.0, 293.5 ],
237             [ 3.0, 294.0 ],
238             [ 3.0, 294.5 ],
239             [ 3.0, 295.0 ],
230             [ 3.0, 295.5 ],
231             [ 3.0, 296.0 ],
232             [ 3.0, 296.5 ],
233             [ 3.0, 297.0 ],
234             [ 3.0, 297.5 ],
235             [ 3.0, 298.0 ],
236             [ 3.0, 298.5 ],
237             [ 3.0, 299.0 ],
238             [ 3.0, 299.5 ],
239             [ 3.0, 300.0 ],
230             [ 3.0, 300.5 ],
231             [ 3.0, 301.0 ],
232             [ 3.0, 301.5 ],
233             [ 3.0, 302.0 ],
234             [ 3.0, 302.5 ],
235             [ 3.0, 303.0 ],
236             [ 3.0, 303.5 ],
237             [ 3.0, 304.0 ],
238             [ 3.0, 304.5 ],
239             [ 3.0, 305.0 ],
230             [ 3.0, 305.5 ],
231             [ 3.0, 306.0 ],
232             [ 3.0, 306.5 ],
233             [ 3.0, 307.0 ],
234             [ 3.0, 307.5 ],
235             [ 3.0, 308.0 ],
236             [ 3.0, 308.5 ],
237             [ 3.0, 309.0 ],
238             [ 3.0, 309.5 ],
239             [ 3.0, 310.0 ],
230             [ 3.0, 310.5 ],
231             [ 3.0, 311.0 ],
232             [ 3.0, 311.5 ],
233             [ 3.0, 312.0 ],
234             [ 3.0, 312.5 ],
235             [ 3.0, 313.0 ],
236             [ 3.0, 313.5 ],
237             [ 3.0, 314.0 ],
238             [ 3.0, 31
```

```

136      [ 3.0, 4.0 ],
137      [2.866, 4.500],
138      [2.707, 4.707],
139      [0.293, 0.293],
140      [0.134, 0.500],
141      [ 0.0, 1.0 ],
142      [ 0.0, 1.5 ],
143      [ 0.0, 2.0 ],
144      [ 0.0, 2.5 ],
145      [ 0.0, 3.0 ],
146      [ 0.0, 3.5 ],
147      [ 0.0, 4.0 ],
148      [0.134, 4.500],
149      [0.293, 4.707],
150      [0.500, 0.134],
151      [ 1.0, 0.0 ],
152      #ADD x,y PAIRS DATA HERE SO TENSOR LOOKS LIKE Fig. 3pt2B.3 IN WRITEUP - SHOULD BE 32
153      PAIRS TOTAL
154      [ 1.5, 0.0 ],
155      [ 2.0, 0.0 ],
156      [2.500, 0.134],
157      [0.500, 4.866],
158      [ 1.0, 5.0 ],
159      [ 1.5, 5.0 ],
160      [ 2.0, 5.0 ],
161      [2.500, 4.866]
162      ], dtype=tf.float32)
163
164      # Boundary loss: Calculate squared error for the boundary condition
165      boundary_loss = 2.0*K.mean(K.square(y_pred - y_true))
166
167      #####
168
169      # 2 SECOND STEP OPTIONS DEPENDING ON epoch
170
171      # Check epoch from the callback
172      epoch = epoch_tracker.epochs_trained
173      # FOR EARLY EPOCHS
174      if epoch < epoch_tracker.N_epInit:
175
176          #NEXT FIND LOSS DUE TO MATCH OF INTERIOR POINTS TO TARGET w VELOCITY FIELD
177          # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit
178          points = tf.constant([
179              [0.5, 0.5, 16., 78.40579710144928],
180              [0.5, 1.0, 28.444445, 69.38808373590982],
181              [0.5, 1.5, 37.333332, 62.94685990338164],
182              [0.5, 2., 42.666668, 59.08212560386473],
183              [0.5, 2.5, 44.444443, 57.7938808373591],
184              [0.5, 3., 42.666668, 59.08212560386473],
185              [0.5, 3.5, 37.333332, 62.94685990338164],
186              [0.5, 4., 28.444445, 69.38808373590982],
187              [0.5, 4.5, 16., 78.40579710144928],
188              [1.0, 0.5, 25.6, 71.44927536231884],
189              [1.0, 1.0, 45.511112, 57.02093397745571],
190              [1.0, 1.5, 59.733334, 46.71497584541062],
191              [1.0, 2., 68.266667, 40.53140096618357],
192              [1.0, 2.5, 71.1111155, 38.47020933977455],
193              [1.0, 3., 68.266667, 40.53140096618357],
194              [1.0, 3.5, 59.733334, 46.71497584541062],
195              [1.0, 4., 45.511112, 57.02093397745571],
196              [1.0, 4.5, 25.6, 71.44927536231884],
197              [1.5, 0.5, 28.8, 69.13043478260869],
198              [1.5, 1.0, 51.2, 52.89855072463767],
199              [1.5, 1.5, 67.2, 41.30434782608695],
200              [1.5, 2., 76.8, 34.347826086956516],
201              [1.5, 2.5, 80., 32.028985507246375],
202

```

```

203     [1.5, 3., 76.8, 34.347826086956516],  

204     [1.5, 3.5, 67.2, 41.30434782608695],  

205     [1.5, 4., 51.2, 52.89855072463767],  

206     [1.5, 4.5, 28.8, 69.13043478260869],  

207     [2., 0.5, 25.6, 71.44927536231884 ],  

208     [2., 1.0, 45.511112, 57.02093397745571],  

209     [2., 1.5, 59.733334, 46.71497584541062],  

210     [2., 2., 68.2666, 40.53140096618357],  

211     [2., 2.5, 71.111115, 38.47020933977455],  

212     [2., 3., 68.26667, 40.53140096618357],  

213     [2., 3.5, 59.733334, 46.71497584541062],  

214     [2., 4., 45.511112, 57.02093397745571],  

215     [2., 4.5, 25.6, 71.44927536231884],  

216     [2.5, 0.5, 16., 78.40579710144928],  

217     [2.5, 1.0, 28.444445, 69.38808373590982],  

218     [2.5, 1.5, 37.333332, 62.94685990338164],  

219     [2.5, 2., 42.666668, 59.08212560386473],  

220     [2.5, 2.5, 44.444443, 57.7938808373591],  

221     [2.5, 3., 42.666668, 59.08212560386473],  

222     [2.5, 3.5, 37.333332, 62.94685990338164],  

223     [2.5, 4., 28.444445, 69.38808373590982],  

224     [2.5, 4.5, 16., 78.40579710144928]  

225 ], dtype=tf.float32)  

226  

227 meanIntTarg_res = 0.  

228 for _ in range(45):  

229     # does this list 45 times not keeping track of index  

230     # THIS COMPUTES INTERIOR POINT TARGET SOLUTION LOSS FUNCTION  

231     # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit  

232  

233     # Randomly select one point  

234     idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)  

235     selected_point = tf.gather(points, idx) # shape: [3]  

236  

237     # Slice to get only the first two columns: x and y  

238     xy_input = selected_point[:2] # shape: [2]  

239  

240     # Prepare input for the model: [1, 2]  

241     model_input = tf.reshape(xy_input, shape=(1, 2))  

242  

243     #retreive w target value  

244     w_targInt = selected_point[2]  

245     #retreive T target value  

246     T_targInt = selected_point[3]  

247  

248     # Compute the output  

249     Predoutput = model(model_input)  

250     wpred = Predoutput[0, 0]  

251     Tpred = Predoutput[0, 1]  

252     # add loss contributions  

253     meanIntTarg_res += (wpred - w_targInt)*(wpred - w_targInt)/45.  

254     meanIntTarg_res += (Tpred - T_targInt)*(Tpred - T_targInt)/45.  

255  

256  

257     #Combine mean boundary and mean interior point target losses  

258     total_loss = boundary_loss + meanIntTarg_res  

259  

260  

261 else:  

262     # DO THIS IF CURRENT epoch IS GREATER THAN N_epInit,  

263     # THIS COMPUTES THE PDE RESIDUAL LOSS FUNCTION at 45 INTERIOR POINTS AND  

264     # SKIPS ERROR CALC FOR TARGET GUESSED w FIELD AT INTERIOR POINTS  

265     # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch > epoch_tracker.N_epInit  

266  

267     #>>> Randomly select an interior point from this list using TensorFlow's random  

268     # uniform  

269     points2 = tf.constant([
          [0.5, 0.5],

```

```

270      [0.5, 1.0],
271      [0.5, 1.5],
272      [0.5, 2.],
273      [0.5, 2.5],
274      [0.5, 3.],
275      [0.5, 3.5],
276      [0.5, 4.],
277      [0.5, 4.5],
278      [1.0, 0.5],
279      [1.0, 1.0],
280      [1.0, 1.5],
281      [1.0, 2.],
282      [1.0, 2.5],
283      [1.0, 3.],
284      [1.0, 3.5],
285      [1.0, 4.],
286      [1.0, 4.5],
287      [1.5, 0.5],
288      [1.5, 1.0],
289      [1.5, 1.5],
290      [1.5, 2.],
291      [1.5, 2.5],
292      [1.5, 3.],
293      [1.5, 3.5],
294      [1.5, 4.],
295      [1.5, 4.5],
296      [2., 0.5],
297      [2., 1.0],
298      [2., 1.5],
299      [2., 2.],
300      [2., 2.5],
301      [2., 3.],
302      [2., 3.5],
303      [2., 4.],
304      [2., 4.5],
305      [2.5, 0.5],
306      [2.5, 1.0],
307      [2.5, 1.5],
308      [2.5, 2.],
309      [2.5, 2.5],
310      [2.5, 3.],
311      [2.5, 3.5],
312      [2.5, 4.],
313      [2.5, 4.5]
314  ], dtype=tf.float32)
315  # THIS LIST INCLUDES ALL 45 [X,Y] POINTS IN points ARRAY IN PROJECT WRITEUP
316
317
318  meanPois_res_w = 0.
319  meanPois_res_T = 0.
320  #meanwlimPen = 0. #ASSOCIATED WITH PENALTY CALC COMMENTED_OUT HERE BELOW
321  for _ in range(45):
322      ##compute Laplacian/Poisson residual here - THIS LIST IS DONE ONCE PER epoch
323
324      # Use tf.random.uniform to sample an index for the interior point
325      idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)
326      selected_point = tf.gather(points2, idx)
327      #x_interior, y_interior = selected_point[0], selected_point[1]
328      =====
329      input_point = tf.stack([selected_point[0], selected_point[1]])    # shape:
330      (2, )
331      input_point = tf.reshape(input_point, (1, 2))  # shape: (1, 2) for batch
332
333      # First and second derivatives - extraction from model have output of shape
334      (2)
335      with tf.GradientTape(persistent=True) as tape2:
336          tape2.watch(input_point)
337          with tf.GradientTape(persistent=True) as tape1:

```

```

336         tape1.watch(input_point)
337         output = model(input_point)      # shape: (1, 2)
338         wpred = output[0, 0]
339         Tpred = output[0, 1]
340
341         # First-order derivatives
342         dw_dx, dw_dy = tape1.gradient(wpred, input_point)[0]
343         dT_dx, dT_dy = tape1.gradient(Tpred, input_point)[0]
344
345         # Second-order derivatives
346         d2w_dx2, d2w_dy2 = tape2.gradient([dw_dx, dw_dy], input_point)[0]
347         d2T_dx2, d2T_dy2 = tape2.gradient([dT_dx, dT_dy], input_point)[0]
348
349         # Clean up
350         del tape1
351         del tape2
352         =====
353
354
355         # Calculate the Laplacian (sum of second derivatives)
356         laplacian_w = d2w_dx2 + d2w_dy2
357         laplacian_T = d2T_dx2 + d2T_dy2
358
359         #shape_tensor = tf.shape(dw_dx)
360         #tf.print("Shape of dw_dx:", shape_tensor)
361         #shape_tensor = tf.shape(dw_dy)
362         #tf.print("Shape of dw_dy:", shape_tensor)
363
364         #ADD PRESSURE GRADIENT TERM dPmdzomu = dPdz over mu (POISSON EQ for w)
365         dPmdzomu = -80.0
366         # Compute the mean w Poisson residual for the 45 points( sum of squared (
367             Laplacian - dPmdzomu ) )
368         meanPois_res_w += K.square((laplacian_w - dPmdzomu))/45.
369
370         #ADD TEMPERATURE GRADIENT TERM w*dTmdzoalpha = (w*dTdZ over alpha) (POISSON
371             EQ)
372         # dTmdzoalpha ~ sdegC/mm**3
373         dTmdzoalpha = 0.103
374         # Compute the mean T Poisson residual for the 45 points( sum of squared (
375             Laplacian - w_pred*dTmdzoalpha ) )
376         meanPois_res_T += K.square((laplacian_T - wpred*dTmdzoalpha))/45.
377
378         #Combine boundary and interior Poisson Eq. residual losses
379         total_loss = boundary_loss + kappa * meanPois_res_w + kappa * meanPois_res_T
380
381         return total_loss
382     return loss
383
384     # ===== END CUSTOM LOSS FUNCTION DEFINITION =====
385
386
387     from tensorflow.keras.callbacks import Callback
388
389     class EpochTracker(Callback):
390         def __init__(self, N_epInit):
391             super().__init__()
392             self.N_epInit = N_epInit
393             self.epochhs_trained = 0
394
395         def on_epoch_end(self, epoch, logs=None):
396             self.epochhs_trained = epoch
397
398         # Set the loss function with the required arguments
399         # Set N_epInit=2000 for first run of fit routine in next cell,
400         # then set it then set N_epInit=1 for all runs of fit routine beyond that

```

```

401 epoch_tracker = EpochTracker(N_epInit=1)
402
403 custom_loss = custom_loss_function(kappa=1.3, x_max=3.0, y_max=5.0, epoch_tracker=
404     epoch_tracker)
405
406
407 # Compile the model with the custom loss
408 model.compile(optimizer='adam', loss=custom_loss)
409
410 ##### After the compilation of the model, run the fit method with some number of epochs (can
411     be run iteratively.
412
413
414 #The fit method takes three parameters; namely, x, y, and number of epochs.
415 #During model training, if all the batches of data are seen by the model once,
416 #we say that one epoch has been completed.
417
418
419 ##### RUN FIT
420
421 # Assuming 'boundary_points' and 'boundary_w' are TensorFlow tensors
422
423 # Now, fit the model and pass the callback to 'fit()'
424 #historyData = model.fit(boundary_points_xtrain, boundary_w_ytrain,
425 #    epochs=2, batch_size=32,
426 #    callbacks=[epoch_tracker])
427 historyData = model.fit(tf_tensor_inputs, tf_tensor_outputs,
428     epochs=2500, batch_size=32,
429     callbacks=[epoch_tracker])
430
431
432 loss_hist = historyData.history['loss']
433 #The above line will return a dictionary, access it's info like this:
434 best_epoch = np.argmin(historyData.history['loss']) + 1
435 print ('best epoch = ', best_epoch)
436 print('smallest loss = ', np.min(loss_hist))
437
438 predictions = model.predict(data_inputs)
439
440 predictions = model.predict(data_inputs)
441
442 #SETTING UP PLOT
443 %matplotlib inline
444 # importing the required module
445 import matplotlib.pyplot as plt
446 plt.rcParams['figure.figsize'] = [8, 8] # for square canvas
447 =====
448
449 '''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''
450
451 #CAN'T USE THIS LOG-LOG FOR W BECAUSE DATA VALUES ARE ALL ZERO
452
453 plt.scatter(predictions[:, 0], data_outputs[:, 0])
454 #plt.title('w,T PINN training of Neural Network ==> y = w')
455 plt.xlabel('predicted output for NN (mm/s)')
456 plt.ylabel('data output (mm/s)')
457 #plt.loglog()
458 #plt.xlim(xmax = 100, xmin = 1.)
459 #plt.ylim(ymax = 100, ymin = 1.)
460 # Generate red y=x line
461 #x_data = np.linspace(1.0, 100.0, num=3)
462
463 plt.xlim(xmax = 10, xmin = -10.)
464 plt.ylim(ymax = 10, ymin = -10.)
465 # Generate red y=x line
466 #x_data = np.linspace(1.0, 100.0, num=3)

```

```

467 x_data = np.linspace(-10., 10.0, num=3)
468 y_data = x_data
469 plt.plot(x_data, y_data, color='red')
470 plt.show()
471 #PLOTS VELOCITY BOUNDARY DATA ONLY
472
473 # w velocity field
474 test = []
475 outpt=[]
476 predictions = model.predict(data_inputs)
477
478 #first point (row [0])comparison of data and prediction
479 # put in a loop to print comparion for all data points
480
481 xa=[]
482 ya=[]
483 za=[]
484
485 xao = np.linspace(0, 3, 7) # 1D array for x values
486 yao = np.linspace(0, 5, 11) # 1D array for y values
487 xa = np.linspace(0, 10, 77) # 1D array for x values
488 ya = np.linspace(0, 10, 77) # 1D array for y values
489 za = np.linspace(0, 10, 77) # 1D array for z values
490
491 x = -0.50
492 ia=-1
493 while (x < 2.51):
494     x = x + 0.5
495     y = -0.5
496     while (y < 4.51):
497         ia=ia+1
498         y = y + 0.50
499         xa[ia]=x
500         ya[ia]=y
501
502         test = [[ x , y ]]
503         testarray = np.array(test)
504         outpt = model.predict(testarray)
505         za[ia] = outpt[0][0]
506         print ('x= ', testarray[0][0], ', y= ', testarray[0][1], 'w= ', outpt[0][0])
507         #Predoutput[0, 0] ?
508
509
510 X, Y, Z = xa, ya, za
511
512 import sys
513 #import csv
514 import numpy as np
515 import matplotlib.pyplot as plt
516 from mpl_toolkits.mplot3d import axes3d
517
518 # Plot X,Y,Z
519
520 # Creating figure
521 #fig = plt.figure(figsize =(16, 9))
522 #ax = plt.axes(projection ='3d')
523
524 fig = plt.figure()
525 #ax = fig.add_subplot(111, projection='3d')
526 ax = plt.axes(projection ='3d')
527 #ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
528 ax.plot_trisurf(X, Y, Z,
529                 linewidth = 0.2,
530                 antialiased = True
531                 , cmap='viridis');
532 ax.invert_yaxis()
533 ax.invert_xaxis()
534 plt.show()

```

```

535
536
537
538 # Plot the data
539 plt.figure(figsize=(5, 6))
540 plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
541 plt.colorbar(label='w value')
542 plt.xlabel('x')
543 plt.ylabel('y')
544 plt.title('Heat Map of w values (mm/s) over x and y domain (mm)')
545 plt.show()
546
547 #temperature field T
548 test = []
549 outpt=[]
550 predictions = model.predict(data_inputs)
551
552 #first point (row [0])comparison of data and prediction
553 # put in a loop to print comparion for all data points
554
555 xa=[]
556 ya=[]
557 za=[]
558
559 xao = np.linspace(0, 3, 7) # 1D array for x values
560 yao = np.linspace(0, 5, 11) # 1D array for y values
561 xa = np.linspace(0, 10, 77) # 1D array for x values
562 ya = np.linspace(0, 10, 77) # 1D array for y values
563 za = np.linspace(0, 10, 77) # 1D array for z values
564
565 x = -0.50
566 ia=-1
567 while (x < 2.51):
568     x = x + 0.5
569     y = -0.5
570     while (y < 4.51):
571         ia=ia+1
572         y = y + 0.50
573         xa[ia]=x
574         ya[ia]=y
575
576         test = [[ x , y ]]
577         testarray = np.array(test)
578         outpt = model.predict(testarray)
579         za[ia] = outpt[0][1]
580         print ('x (mm) = ', testarray[0][0], ', y (mm) = ', testarray[0][1], 'T (degC) = ',
581               outpt[0][1])
582         #Predoutpt[0, 0] ???
583
584 X, Y, Z = xa, ya, za
585
586 import sys
587 #import csv
588 import numpy as np
589 import matplotlib.pyplot as plt
590 from mpl_toolkits.mplot3d import axes3d
591
592 # Plot X,Y,Z
593
594 # Creating figure
595 #fig = plt.figure(figsize =(16, 9))
596 #ax = plt.axes(projection ='3d')
597
598 fig = plt.figure()
599 #ax = fig.add_subplot(111, projection='3d')
600 ax = plt.axes(projection ='3d')
601 #ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)

```

```

602 ax.plot_trisurf(X, Y,
603                 linewidth = 0.2,
604                 antialiased = True
605                 , cmap='viridis');
606 ax.invert_yaxis()
607 ax.invert_xaxis()
608 plt.show()
609
610
611
612 # Plot the data
613 plt.figure(figsize=(5, 6))
614 plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
615 plt.colorbar(label='T value')
616 plt.xlabel('x')
617 plt.ylabel('y')
618 plt.title('Heat Map of T values (degC) over x and y domain (mm)')
619 plt.show()

```

9.4 Part 2 FDM Code (Project 1)

```

1 % Gauss-Seidel Solver for Fully Developed Heat Transfer in a Rectangular Channel
2 clear
3 clc
4 close all
5 % Define constants and parameters
6 Nx = 30; % Number of nodes in x direction
7 Ny = 50; % Number of nodes in y direction
8 dx = 0.1e-3; % Grid spacing in x (m)
9 dy = 0.1e-3; % Grid spacing in y (m)
10 Tw = 90; % Wall temperature ( C )
11 nu = 8.26e-7;%9.00e-7 ; % Kinematic viscosity (m^2/s)
12 rho = 997; %1055; % Density (kg/m^3)
13 alpha = 1.46e-7; %1.08e-7; % Thermal diffusivity (m^2/s)
14 cp = 4164; %3559; % Specific heat (J/kg C)
15 k = 0.606; % Thermal conductivity (W/m C)
16 dPdz = -80 * nu * rho * 1000; % Pressure gradient (Pa/m)
17 dTmdz = alpha * .103 * 1000000000; % Mean temperature gradient ( C /m)
18
19 % Convergence criteria
20 epsilon_w = 0.0001;
21 epsilon_T = 0.05;
22
23 % Initialize fields
24 w = zeros(Nx, Ny);
25 T = 70 * ones(Nx, Ny); % Initial temperature field
26 T(1,:) = Tw;
27 T(Nx,:) = Tw;
28 T(:,1) = Tw;
29 T(:,Ny) = Tw;
30
31 % Gauss-Seidel iterations for velocity field
32 converged_w = false;
33 while ~converged_w
34     max_change_w = 0;
35     for i = 2:Nx-1
36         for j = 2:Ny-1
37             w_new = ((dy/dx)^2 * (w(i+1,j) + w(i-1,j)) + w(i,j+1) + w(i,j-1)) / ...
38                     (2*(dy/dx)^2 + 2) - (dPdz * dy^2) / (rho * nu * (2*(dy/dx)^2 + 2));
39             max_change_w = max(max_change_w, abs(w_new - w(i,j)));
40             w(i,j) = w_new;
41         end
42     end
43     if max_change_w < epsilon_w
44         converged_w = true;
45     end

```

```

46 end
47
48 % Gauss-Seidel iterations for temperature field
49 converged_T = false;
50 while ~converged_T
51     max_change_T = 0;
52     for i = 2:Nx-1
53         for j = 2:Ny-1
54             T_new = ((dy/dx)^2 * (T(i+1,j) + T(i-1,j)) + T(i,j+1) + T(i,j-1)) / ...
55             (2*(dy/dx)^2 + 2) - (w(i,j) * dTdz * dy^2) / (alpha * (2*(dy/dx)^2 + 2));
56             max_change_T = max(max_change_T, abs(T_new - T(i,j)));
57             T(i,j) = T_new;
58         end
59     end
60     if max_change_T < epsilon_T
61         converged_T = true;
62     end
63 end
64
65 %% Post-processing
66 % Numerical integration for mean temp and mean velocity
67 % all domain nodes now have one unit area
68 A = dx * dy * ones(Nx, Ny);
69 % all boundary nodes now have 0.5 unit area
70 A(1,:) = 0.5 * dx * dy; % left wall
71 A(end,:) = 0.5 * dx * dy; % right wall
72 A(:,1) = 0.5 * dx * dy; % bottom wall
73 A(:,end) = 0.5 * dx * dy; % top wall
74 A0 = Nx*Ny*dx*dy;
75 w_m = sum(sum(w .* A)) / A0;
76 % All nodes contribute dxdy
77 T_m = sum(sum(w .* T .* A)) / (w_m * A0);
78 % Compute local heat flux and heat transfer coefficient
79 q_w = zeros(Nx, Ny);
80 h = zeros(Nx, Ny);
81 for i = 2:Nx-1
82     q_w(i,1) = k * (T(i,1) - T(i,2)) / dy; % bottom wall
83     q_w(i,Ny) = k * (T(i,Ny) - T(i,Ny-1)) / dy; % top wall
84 end
85 for j = 2:Ny-1
86     q_w(1,j) = k * (T(1,j) - T(2,j)) / dx; % left wall
87     q_w(Nx,j) = k * (T(Nx,j) - T(Nx-1,j)) / dx; % right wall
88 end
89
90 for i = 2:Nx-1
91     for j = [1 Ny]
92         h(i,j) = q_w(i,j) / (T(i,j) - T_m);
93     end
94 end
95 for j = 2:Ny-1
96     for i = [1 Nx]
97         h(i,j) = q_w(i,j) / (T(i,j) - T_m);
98     end
99 end
100
101 % Total heat input rate and mean heat transfer coefficient
102
103 A_wall = .5 * dx * dy * ones(Nx, Ny);
104 Q_total = sum(q_w(:) .* A_wall(:));
105 Aw = 0.5 * dx * dy * (Nx + Nx + Ny + Ny);
106 T_wall_avg = (sum(T(1,1:Ny)) + sum(T(Nx, 1:Ny)) + sum(T(1:Nx-1,1)) + sum(T(1:Nx-1,Ny))) / ...
    (2*(Nx+Ny));
107 h_mean = Q_total / (Aw * (T_wall_avg - T_m));
108
109
110 % Generate 3D plots
111 [x_grid, y_grid] = meshgrid(linspace(0, (Nx)*dx, Nx), linspace(0, (Ny)*dy, Ny));

```

```

112 figure; mesh(x_grid, y_grid, w'); title('Velocity Field w (m/s)'); xlabel('x'); ylabel('y');
113 zlabel('w (m/s)');
114 figure; mesh(x_grid, y_grid, T'); title('Temperature Field T ( C )'); xlabel('x'); ylabel('y');
115 zlabel('T ( C )');

116 % Heat transfer coefficient plots
117 h_left_wall = h(1, 2:Ny-1);
118 h_right_wall = h(Nx, 2:Ny-1);
119 h_top_wall = h(2:Nx-1, 1);
120 h_bottom_wall = h(2:Nx-1, Ny);

121 figure; plot(2:Nx-1, h_top_wall(:, ), 'r', 2:Nx-1, h_bottom_wall(:, ), 'b');
122 title('Heat Transfer Coefficient along short walls');
123 legend('Top Wall HTC (W/m^2C)', 'Bottom Wall HTC (W/m^2C)')
124 figure; plot(2:Ny-1, h_left_wall(:, ), 'r', 2:Ny-1, h_right_wall, 'b');
125 title('Heat Transfer Coefficient along long walls');
126 legend('Left Wall HTC (W/m^2C)', 'Right Wall HTC (W/m^2C)')

127 Dh = 4 * (AO) / (2*(Nx*dy+Ny*dx));
128 Nusselt = h_mean * Dh / k;
129 Reynolds = wm * Dh / nu;

130 % Aspect ratios and Nusselt numbers for first 4 data points
131 aspect_ratios = [1, 0.5, 0.25, 0.125];
132 nusselt_numbers = [2.976, 3.391, 4.439, 5.597];
133 last_aspect_ratio = 0.262295082;
134 last_nusselt_number = 5.987893727;
135 last_aspect_ratio2 = 21/41;
136 last_nusselt_number2 = 4.98;

137 figure;
138 plot(aspect_ratios, nusselt_numbers, '-o', 'LineWidth', 2, 'MarkerSize', 8);
139 hold on;
140 plot(last_aspect_ratio, last_nusselt_number, 'rx', 'MarkerSize', 10, 'LineWidth', 2);
141 plot(last_aspect_ratio2, last_nusselt_number2, 'rx', 'MarkerSize', 10, 'LineWidth', 2);
142 xlabel('Aspect Ratio of Rectangular Channels');
143 ylabel('Average Nusselt number for uniform Wall Temperature');
144 title('Nusselt Number vs. Aspect Ratio');
145 ylim([0, 6.5]);
146 grid on;
147 hold off;

```