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 Hui[1] and Derek Shah

[1]University 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

List of Figures

PART 1: High-Prandtl Turbulent Film Analysis

1 Introduction and Theoretical Background

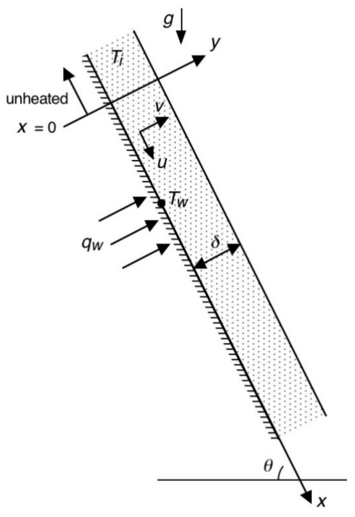

Figure 1: Turbulent Falling Liquid Film on Inclined Flat Surface

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

$$\frac{\partial}{\partial y}\left[\alpha\frac{\partial \overline{T}}{\partial y} - \overline{\nu' T'}\right] = 0 \tag{1}$$

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

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

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

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

where q_w is the wall heat flux and $F_I(\mathrm{Pr}, \nu)$ 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

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

$$\int \frac{\partial}{\partial y}\left[(\alpha + \epsilon_H)\frac{\partial \overline{T}}{\partial y}\right] dy = \int 0 dy$$

$$(\alpha + \epsilon_H)\frac{\partial \overline{T}}{\partial y} = C_1$$

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

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

Then substituting this into our axial heat flux expression yields

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

Therefore, the region near the wall obeys

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

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

$$\frac{\partial \overline{T}}{\partial y} = -\frac{\alpha}{\alpha + \epsilon_H}\cdot\frac{q_w}{k}$$

$$\int_{T_w}^{T_i} \frac{\partial \overline{T}}{\partial y} dy = -\frac{\alpha q_w}{k}\int_0^{\delta_{nw}} \frac{1}{\alpha + \epsilon_H} dy$$

$$T_i - T_w = -\frac{\alpha q_w}{k}\int_0^{\delta_{nw}} \frac{1}{\alpha + \epsilon_H} dy$$

$$T_w - T_i = \frac{\alpha q_w}{k}\int_0^{\delta_{nw}} \frac{1}{\alpha + \epsilon_H} dy$$

We introduce the following non-dimensional variables

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

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

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

Recall that the turbulent Prandtl number is defined as

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

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

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

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

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

Substituting the expression in yields

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

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

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

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

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

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

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

where $F_I(\mathrm{Pr},\nu) = \frac{\mathrm{Pr}}{\nu}$ 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 $\mathrm{Pr}\epsilon_M/\nu = \mathcal{O}(1)$. 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

$$\epsilon_M/\nu = \gamma(y^+)^3 \tag{4}$$

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: $\mathrm{Pr}_t = 1.0$. Equation (3) can then be written

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

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 $\eta = \gamma(y^+)^3\mathrm{Pr}$, 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

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

where

$$I = \frac{1}{C_{1.2}}\int_0^{\gamma(\delta^+_{nw})^3\mathrm{Pr}} \frac{\eta^{-2/3}}{1 + \eta/\mathrm{Pr}_t}d\eta \tag{7}$$

where $\mathrm{Pr}_t = 1.0$, 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

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

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

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

changing the bounds of integration as well, we get

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

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

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

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

$$\frac{(T_w - T_i)k\sqrt{\tau_w/\rho_l}}{q_w/\nu} = \int_0^{\delta^+_{nw}} \frac{dy^+}{1 + F_I(\mathrm{Pr},\nu)\frac{\epsilon_M}{\mathrm{Pr}_t}}, \quad \mathrm{Pr}_t \simeq 1 \text{ Kato, et al. (1968)}$$
$$\frac{k}{h}\left(\frac{\sqrt{\tau_w/\rho_l}}{\nu}\right) = \frac{(\gamma \mathrm{Pr})^{-1/3}}{3}\int_0^{\gamma(\delta^+_{nw})^3\mathrm{Pr}} \frac{\eta^{-2/3}}{1+\eta/\mathrm{Pr}_t}d\eta$$

Taking the inverse on both sides yields the Eq. 6

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

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

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

□

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:

$$I = \frac{1}{C_{1.2}}\int_0^{\infty} \frac{\eta^{-2/3}}{1+\eta/\mathrm{Pr}_t}d\eta \tag{8}$$

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

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 $\eta^{-5/3}$.

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

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

$$\begin{aligned} I_1 &= \frac{1}{C_{1.2}}\int_0^{0.01} \frac{\eta^{-2/3}}{1.005}d\eta \\ &= \frac{1}{3}\int_0^{0.01} \frac{\eta^{-2/3}}{1.005}d\eta \\ &= \frac{1}{3.015}\left[3\eta^{1/3}\right]_0^{0.01} \Rightarrow I_1 = 0.2144 \end{aligned}$$

The third integral is evaluated as follows

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

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,

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

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.

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

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:

$$\Gamma = \rho_l \int_0^\delta U dy \tag{13a}$$

Using the power-law universal velocity profile we obtain

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

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:

$$\Gamma = f_\Gamma(\tau_w, \rho_l, \nu, \delta) \tag{14}$$

This result has the form

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

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

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

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

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

$\square$

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.

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

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

$$\mathrm{Re} = \frac{4\Gamma}{\rho_l\nu} \tag{17}$$

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

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

We can generate an expression for u_τ as follows

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

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

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

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

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

Multiplying both sides by a factor gets us

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

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

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

Now, substituting in the known values of $\gamma = 5.10\times10^{-4}$ and our calculate integral $I = 1.208826$, we have

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

□

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

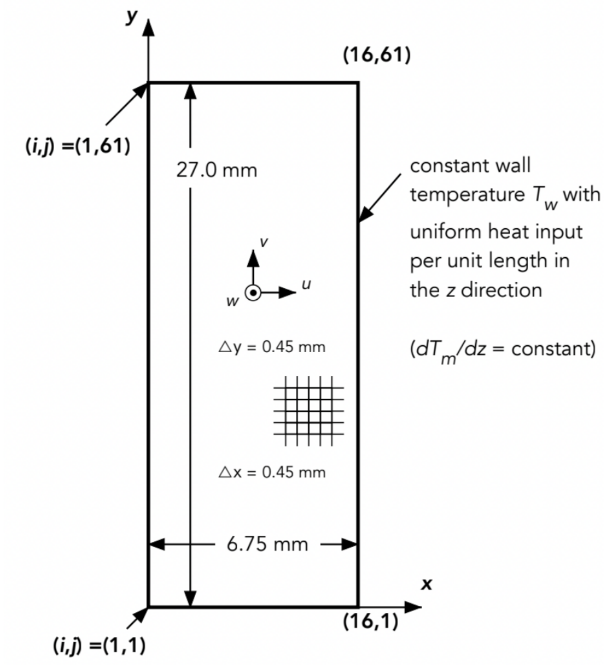

Figure 2: Constant Wall Temperature T_w with Uniform Heat Input

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:

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

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

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,

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

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

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

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

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/mm^2 (1 Pa = 1 N/m^2 = 10^{-6}N/mm^2), 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: $\alpha = 0.146\text{mm}^2/\text{s}$, $\rho = 9.97 \times 10^{-7}\text{kg/mm}^3$, $\nu = 0.826\text{mm}^2/\text{s}$, $c_p = 4164\text{J/kg}^\circ\text{C}$, $k = 6.06 \times 10^{-4}\text{W/mm}^\circ\text{C}$. 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

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

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.

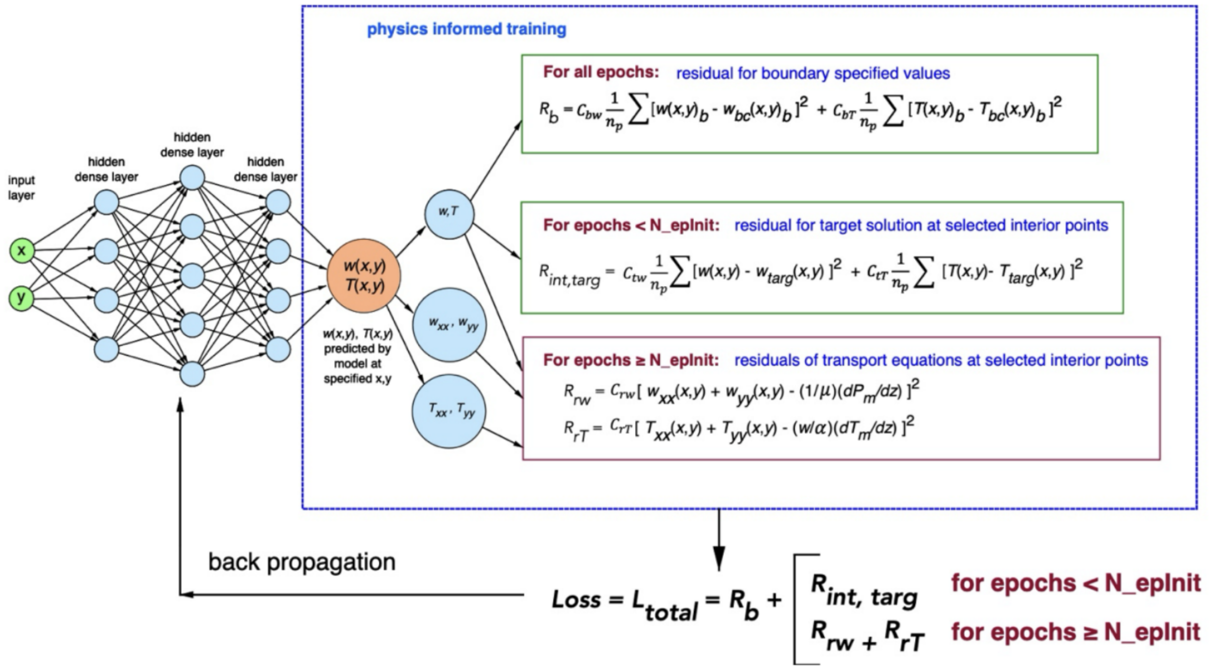
physics informed training
For all epochs: residual for boundary specified values
$R_b = C_{bw}\frac{1}{n_p}\sum[w(x,y)_b - w_{bc}(x,y)_b]^2 + C_{bT}\frac{1}{n_p}\sum[T(x,y)_b - T_{bc}(x,y)_b]^2$
For epochs < N_epInit: residual for target solution at selected interior points
$R_{int,targ} = C_{tw}\frac{1}{n_p}\sum[w(x,y) - w_{targ}(x,y)]^2 + C_{tT}\frac{1}{n_p}\sum[T(x,y) - T_{targ}(x,y)]^2$
For epochs ≥ N_epInit: residuals of transport equations at selected interior points
$R_{rw} = C_{rw}[w_{xx}(x,y) + w_{yy}(x,y) - (1/\mu)(dP_m/dz)]^2$
$R_{rT} = C_{rT}[T_{xx}(x,y) + T_{yy}(x,y) - (w/\alpha)(dT_m/dz)]^2$
input layer
hidden dense layer
hidden dense layer
hidden dense layer
x
y
w(x,y) T(x,y)
w(x,y), T(x,y) predicted by model at specified x,y
w,T
w_{xx}, w_{yy}
T_{xx}, T_{yy}
back propagation
$Loss = L_{total} = R_b + \begin{cases} R_{int,\,targ} & \text{for epochs < N\_epInit} \\ R_{rw} + R_{rT} & \text{for epochs ≥ N\_epInit} \end{cases}$

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 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 theAppendix 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 $T = 90°C$). 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 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 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**.

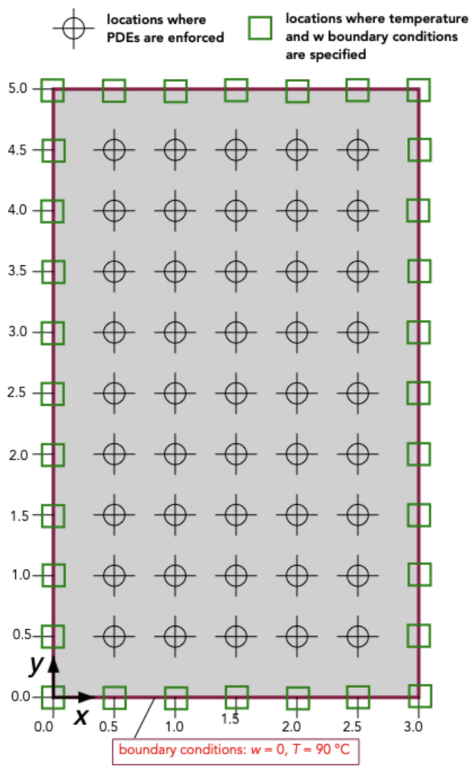
locations where PDEs are enforced
locations where temperature and w boundary conditions are specified
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
y
x
0.0
0.5
1.0
1.5
2.0
2.5
3.0
boundary conditions: w = 0, T = 90 °C

Figure 4: Domain of convective flow and heat transport with specified $w = 0$ and $T = 90°\text{C}$ 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**.

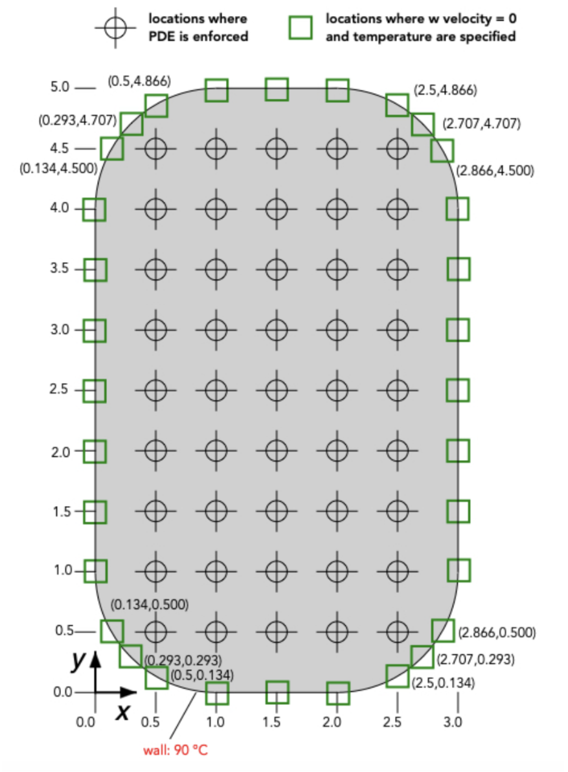

Figure 5: Domain of convective flow and heat transport for a rounded-corner tube with specified w velocity and T at the boundary. The coordinates x and y are in millimeters.

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.

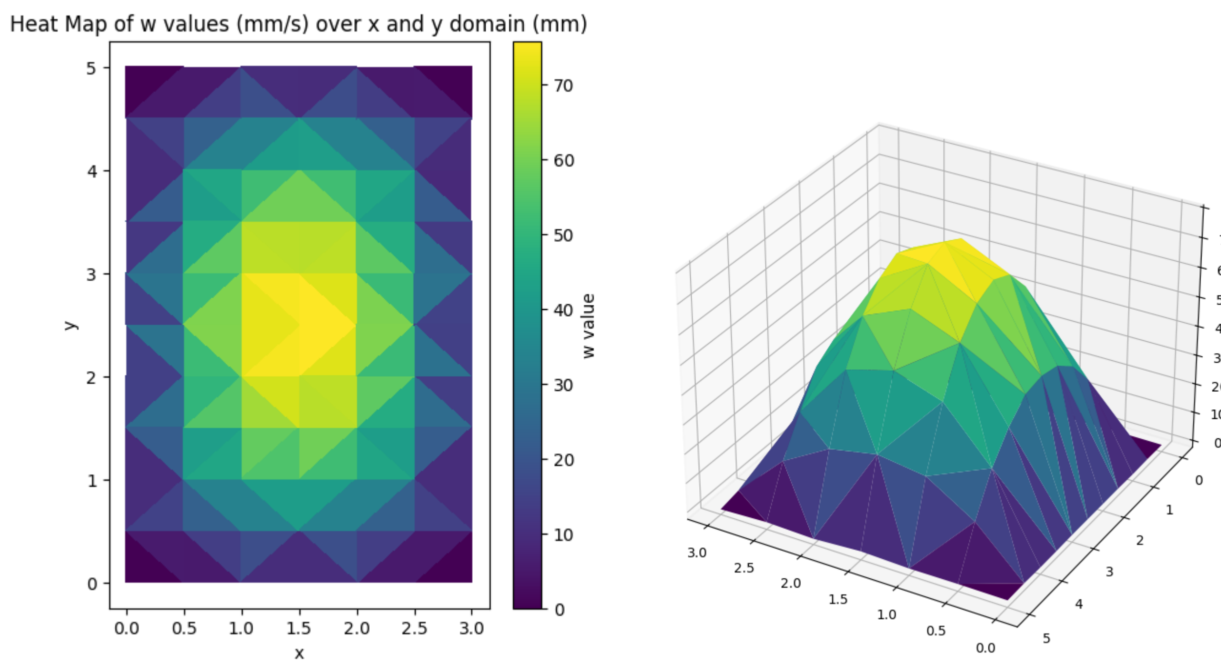

Figure 6: PINN Velocity Profile for Rectangular Channel

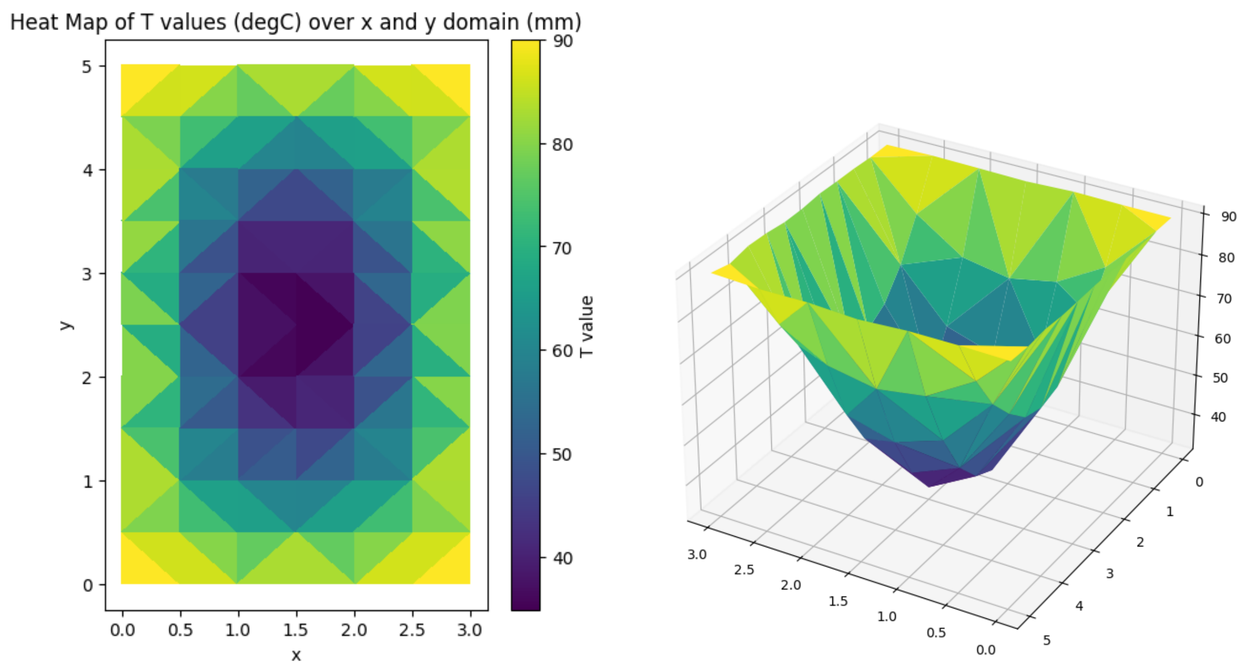

Figure 7: PINN Temperature Profile for Rectangular Channel

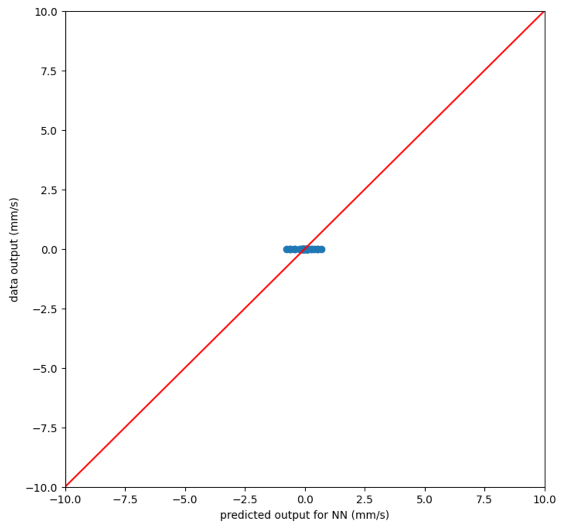

Figure 8: Parity Plot for Rectangular Channel

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

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

Wall location	T_{wall} (°C)	T_{int} (°C)	$\partial T/\partial n$ ($\mathrm{{}^\circ C\,m^{-1}}$)	q'' ($\mathrm{kW\,m^{-2}}$)
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 $w_{\max} = 79.192\ \mathrm{mm\,s^{-1}} = 0.0792\ \mathrm{m\,s^{-1}}$, occurring at the point (1.5 mm, 2.5 mm). The coldest fluid temperature in the dataset is $T_{\min} = 32.267$°C, found at the same location.

Model	q''_{long} ($\mathrm{kW\,m^{-2}}$)	q''_{short} ($\mathrm{kW\,m^{-2}}$)	Peak $\lvert w\rvert$ ($\mathrm{m\,s^{-1}}$)	$T_{\min}$ (°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:

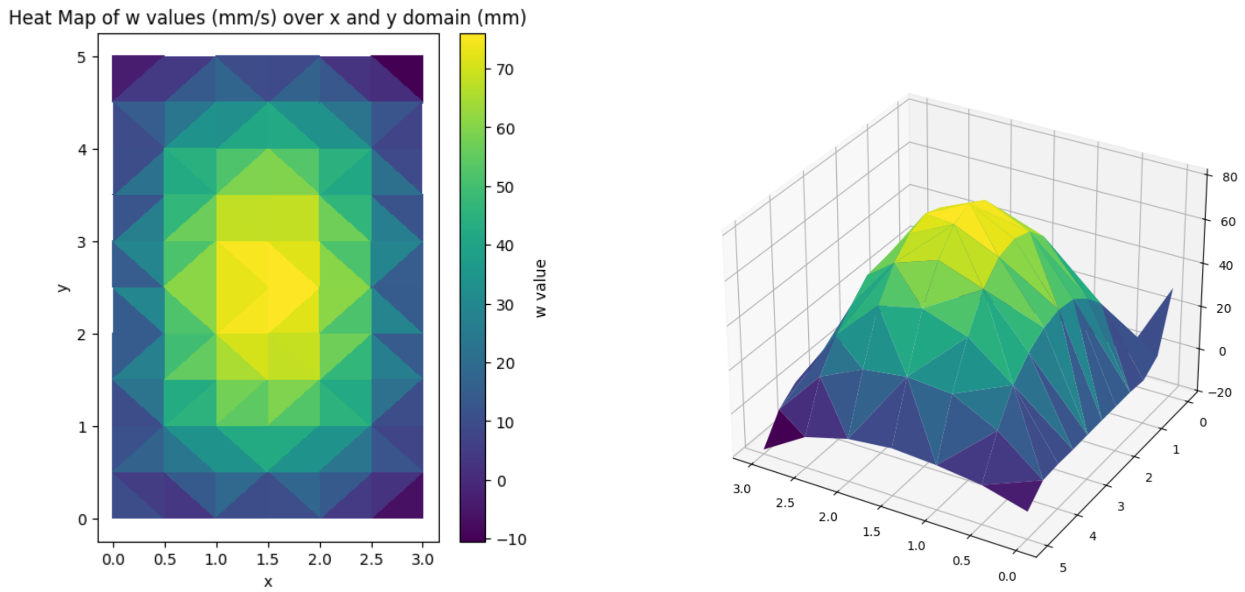

Figure 9: PINN Velocity Profile for Rounded Channel

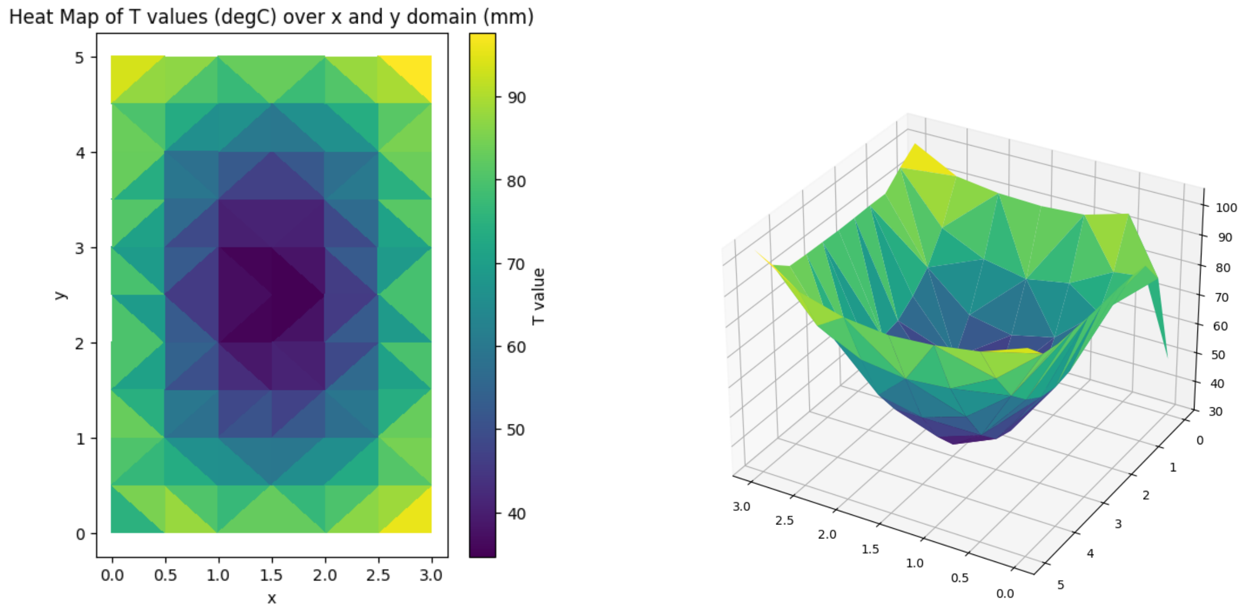

Figure 10: PINN Temperature Profile for Rounded Channel

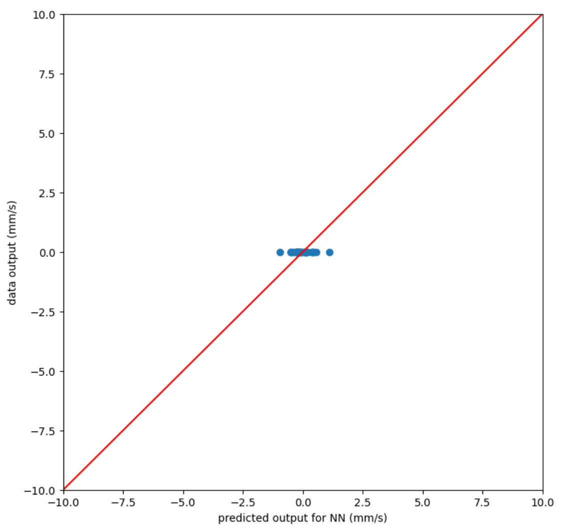

Figure 11: Parity Plot for Rounded Channel

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 $\lvert w\rvert$ (m s^{-1})	T_{min} (°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 ($\Delta n = 5.0 \times 10^{-4}$ m) to approximate the normal temperature gradient,

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

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

Location (m)	Δn (m)	T_{wall} (°C)	T_{int} (°C)	$\partial T/\partial n$ (°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 $\sim 6.5\times10^4$ °C m^{-1}, long wall $\sim 4.1\times10^4$ °C m^{-1}), producing heat-flux densities of 39 and 25 kW m^{-2}, respectively. Approaching the corner the gradient drops to $\sim 2.4 \times 10^4$ °C m^{-1}, so the flux is already down to ≈ 14 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} (°C)	T_{int} (°C)	$\partial T/\partial n$ (°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 $T_{\text{wall}}(3.0\text{ mm}, 5.0\text{ mm}) = 103.86°\text{C}$ and the first interior point $T_{\text{int}}(2.5\text{ mm}, 4.5\text{ mm}) = 79.05°\text{C}$ 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 $\mathrm{kW\,m^{-2}}$; however, the overall wall-integrated flux fell by $\sim 2\%$ because the straight-wall area dominates. Peak axial velocity ($\approx 0.08\ \mathrm{m\,s^{-1}}$) and minimum core temperature ($\approx 32^\circ\mathrm{C}$) 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

```
% Number of blocks
N = 100000;

% Limits of integration
lb = 0.01;
ub = 100;

% Step size and sample points
delta = (ub-lb) / N;
eta = lb:delta:up;

% Integrat evaluated at each sample pt
f = eta.^(-2/3) ./ (1 + eta);

% Trapezoid Rule
I = 1/3 * (delta * (0.5*f(1) + sum(f(2:end-1)) + 0.5*f(end)));
fprintf('I approx %.6f\n', I);
```

9.2 Part 2 PINN Code Rectangular

```
# CodeP3Pt2B.1PINNconvSp25.ipynb - V.P. Carey, ME250B Sp25

#import useful packages
import keras
import pandas as pd
from keras.models import Sequential
import numpy as np
import keras.backend as kb
import tensorflow as tf
#the following 2 lines are only needed for Mac OS machines
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'

#create input data array

# x(mm), y(mm)   These are (x,y) coordinates of boundary location where w velocity and T are
     specified
xdata = []
ND = 32
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
    ], [ 0.0, 3.0 ],
            [ 0.0, 3.5 ], [ 0.0, 4.0 ], [ 0.0, 4.5 ], [ 0.0, 5.0 ],
            [ 3.0, 0.0 ], [ 3.0, 0.5 ], [ 3.0, 1.0 ], [ 3.0, 1.5 ],[ 3.0, 2.0 ], [ 3.0, 2.5
                ], [ 3.0, 3.0 ],
            [ 3.0, 3.5 ], [ 3.0, 4.0 ], [ 3.0, 4.5 ], [ 3.0, 5.0 ]]

xdata.append([  0.5, 0.0 ])
xdata.append([  1.0, 0.0 ])
xdata.append([  1.5, 0.0 ])
xdata.append([  2.0, 0.0 ])
xdata.append([  2.5, 0.0 ])

xdata.append([  0.5, 5.0 ])
xdata.append([  1.0, 5.0 ])
xdata.append([  1.5, 5.0 ])
xdata.append([  2.0, 5.0 ])
xdata.append([  2.5, 5.0 ])
```

```

xarray= np.array(xdata)
print (xdata)
print (xarray)

# w (mm/s)   These are specified velocity values (mm/s) and temperature T (deg C) for each boundary pair of (x,y) above

ydata = []

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.  ] ]

ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90. ])
ydata.append([ 0.0, 90.  ])

ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])

yarray= np.array(ydata)
print (ydata)
print (yarray)

data_inputs = np.array(xdata)
data_outputs = np.array(ydata)

# Convert to TensorFlow tensors
tf_tensor_inputs = tf.convert_to_tensor(data_inputs, dtype=tf.float32)
tf_tensor_outputs = tf.convert_to_tensor(data_outputs, dtype=tf.float32)
shape_tensor = tf.shape(tf_tensor_inputs)
print("Shape of tf_tensor_inputs:", shape_tensor)
shape_tensor = tf.shape(tf_tensor_outputs)
print("Shape of tf_tensor_outputs:", shape_tensor)

# define neural network model

    #As seen below, we have created four dense layers.
    #A dense layer is a layer in neural network thats fully connected.
    #In other words, all the neurons in one layer are connected to all other neurons in the next layer.
    #In the first layer, we need to provide the input shape, which is 1 in our case.
    #The activation function we have chosen is elu, which stands for exponential linear unit.

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

import tensorflow as tf
# Set the seed when creating the initializer
initializer = tf.keras.initializers.RandomUniform(minval=-2.9, maxval=2.9, seed=42)

# Create your model (for example, a simple fully connected feedforward NN)
# Assume 2D input (x, y),  Output: z-direction velocity w
model = Sequential([
        Dense(5, input_dim=2, activation='elu', kernel_initializer=initializer),
        Dense(12, activation='elu',kernel_initializer=initializer),
        Dense(8, activation='elu',kernel_initializer=initializer),
        Dense(2)
    ])
```

```

# Set the kappa value (this can be adjusted as per your needs)
kappa = 1.3

# Set the x_max and y_max values of your domain in mm (can adjust these according to your
    problem)
x_max = 3.0
y_max = 5.0

#print summary of model features
model.summary()

# PINN CUSTOM LOSS FUNCTION - V.P. Carey, ME250B Sp25

import tensorflow as tf
from tensorflow.keras import backend as K

def custom_loss_function(kappa, x_max, y_max, epoch_tracker):
        # Custom loss function for the neural network model.
        #- kappa: weight for the Poisson residual term.
        #- x_max, y_max: maximum x and y coordinates of the domain in mm.
        #- epoch_tracker: callback object to keep track of the current epoch.

    def loss(y_true, y_pred):
        #ALWAYS MATCH BOUNDARY VALUES HERE
        # Define pointsBoundaryChk as a constant tensor

        pointsBoundaryChk = tf.constant([
            [ 3.0, 0.0 ],
            [ 3.0, 0.5 ],
            [ 3.0, 1.0 ],
            [ 3.0, 1.5 ],
            [ 3.0, 2.0 ],
            [ 3.0, 2.5 ],
            [ 3.0, 3.0 ],
            [ 3.0, 3.5 ],
            [ 3.0, 4.0 ],
            [ 3.0, 4.5 ],
            [ 3.0, 5.0 ],
            [ 0.0, 0.0 ],
            [ 0.0, 0.5 ],
            [ 0.0, 1.0 ],
            [ 0.0, 1.5 ],
            [ 0.0, 2.0 ],
            [ 0.0, 2.5 ],
            [ 0.0, 3.0 ],
            [ 0.0, 3.5 ],
            [ 0.0, 4.0 ],
            [ 0.0, 4.5 ],
            [ 0.0, 5.0 ],
            [ 0.5, 0.0 ],
            [ 1.0, 0.0 ],
        #ADD x,y PAIRS DATA HERE SO TENSOR LOOKS LIKE Fig. 3pt2B.3 IN WRITEUP - SHOULD BE 32
            PAIRS TOTAL
            [ 1.5, 0.0 ],
            [ 2.0, 0.0 ],
            [ 2.5, 0.0 ],
            [ 0.5, 5.0 ],
            [ 1.0, 5.0 ],
            [ 1.5, 5.0 ],
            [ 2.0, 5.0 ],
            [ 2.5, 5.0 ]
        ], dtype=tf.float32)


        # Boundary loss: Calculate squared error for the boundary condition
        boundary_loss = 2.0*K.mean(K.square(y_pred - y_true))


        ################

        # 2 SECOND STEP OPTIONS DEPENDING ON epoch

        # Check epoch from the callback
        epoch = epoch_tracker.epochs_trained
        # FOR EARLY EPOCHS
        if epoch < epoch_tracker.N_epInit:

            #NEXT FIND LOSS DUE TO MATCH OF INTERIOR POINTS TO TARGET w VELOCITY FIELD
            # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit
            points = tf.constant([
                [0.5, 0.5, 16., 78.40579710144928],
                [0.5, 1.0, 28.444445, 69.38808373590982],
                [0.5, 1.5, 37.333332, 62.94685990338164],
                [0.5, 2., 42.666668, 59.08212560386473],
                [0.5, 2.5, 44.444443, 57.7938808373591],
                [0.5, 3., 42.666668, 59.08212560386473],
                [0.5, 3.5, 37.333332, 62.94685990338164],
                [0.5, 4., 28.444445, 69.38808373590982],
                [0.5, 4.5, 16., 78.40579710144928],
                [1.0, 0.5, 25.6, 71.44927536231884],
                [1.0, 1.0, 45.511112, 57.02093397745571],
                [1.0, 1.5, 59.733334, 46.71497584541062],
                [1.0, 2., 68.26667, 40.53140096618357],
                [1.0, 2.5, 71.1111155, 38.47020933977455],
                [1.0, 3., 68.26667, 40.53140096618357],
                [1.0, 3.5, 59.733334, 46.71497584541062],
                [1.0, 4., 45.511112, 57.02093397745571],
                [1.0, 4.5, 25.6, 71.44927536231884],
                [1.5, 0.5, 28.8, 69.13043478260869],
                [1.5, 1.0, 51.2, 52.89855072463767],
                [1.5, 1.5, 67.2, 41.30434782608695],
                [1.5, 2., 76.8, 34.347826086956516],
                [1.5, 2.5,  80., 32.028985507246375],
                [1.5, 3., 76.8, 34.347826086956516],
                [1.5, 3.5, 67.2, 41.30434782608695],
                [1.5, 4., 51.2, 52.89855072463767],
                [1.5, 4.5, 28.8, 69.13043478260869],
                [2., 0.5, 25.6, 71.44927536231884 ],
                [2., 1.0, 45.511112, 57.02093397745571],
                [2., 1.5, 59.733334, 46.71497584541062],
                [2., 2., 68.2666, 40.53140096618357],
                [2., 2.5, 71.111115, 38.47020933977455],
                [2., 3., 68.26667, 40.53140096618357],
                [2., 3.5, 59.733334, 46.71497584541062],
                [2., 4., 45.511112, 57.02093397745571],
                [2., 4.5, 25.6, 71.44927536231884],
                [2.5, 0.5, 16., 78.40579710144928],
                [2.5, 1.0, 28.444445, 69.38808373590982],
                [2.5, 1.5, 37.333332, 62.94685990338164],
                [2.5, 2., 42.666668, 59.08212560386473],
                [2.5, 2.5, 44.444443, 57.7938808373591],
                [2.5, 3., 42.666668, 59.08212560386473],
                [2.5, 3.5, 37.333332, 62.94685990338164],
                [2.5, 4., 28.444445, 69.38808373590982],
                [2.5, 4.5, 16., 78.40579710144928]
            ], dtype=tf.float32)

            meanIntTarg_res = 0.
            for _ in range(45):
                # does this list 45 times not keeping track of index
                # THIS COMPUTES INTERIOR POINT TARGET SOLUTION LOSS FUNCTION
                # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit

                # Randomly select one point
                idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)
                selected_point = tf.gather(points, idx)  # shape: [3]

                # Slice to get only the first two columns: x and y
                xy_input = selected_point[:2]  # shape: [2]

                # Prepare input for the model: [1, 2]
                model_input = tf.reshape(xy_input, shape=(1, 2))

                #retreive w target value
                w_targInt = selected_point[2]
                #retreive T target value
                T_targInt = selected_point[3]

                # Compute the output
                Predoutput = model(model_input)
                wpred = Predoutput[0, 0]
                Tpred = Predoutput[0, 1]
                # add loss contributions
                meanIntTarg_res += (wpred - w_targInt)*(wpred - w_targInt)/45.
                meanIntTarg_res += (Tpred - T_targInt)*(Tpred - T_targInt)/45.


            #Combine mean boundary and mean interior point target losses
            total_loss = boundary_loss + meanIntTarg_res


        else:
            # DO THIS IF CURRENT epoch IS GREATER THAN N_epInit,
            # THIS COMPUTES THE PDE RESIDUAL LOSS FUNCTION at 45 INTERIOR POINTS AND
            # SKIPS ERROR CALC FOR TARGET GUESSED w FIELD AT INTERIOR POINTS
            # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch > epoch_tracker.N_epInit

            #>>> Randomly select an interior point from this list using TensorFlow's random
                uniform
            points2 = tf.constant([
                [0.5, 0.5],
                [0.5, 1.0],
                [0.5, 1.5],
                [0.5, 2.],
                [0.5, 2.5],
                [0.5, 3.],
                [0.5, 3.5],
                [0.5, 4.],
                [0.5, 4.5],
                [1.0, 0.5],
                [1.0, 1.0],
                [1.0, 1.5],
                [1.0, 2.],
                [1.0, 2.5],
                [1.0, 3.],
                [1.0, 3.5],
                [1.0, 4.],
                [1.0, 4.5],
                [1.5, 0.5],
                [1.5, 1.0],
                [1.5, 1.5],
                [1.5, 2.],
                [1.5, 2.5],
                [1.5, 3.],
                [1.5, 3.5],
                [1.5, 4.],
                [1.5, 4.5],
                [2., 0.5],
                [2., 1.0],
                [2., 1.5],
                [2., 2.],
                [2., 2.5],
                [2., 3.],
                [2., 3.5],
                [2., 4.],
                [2., 4.5],
                [2.5, 0.5],
                [2.5, 1.0],
                [2.5, 1.5],
                [2.5, 2.],
                [2.5, 2.5],
                [2.5, 3.],
                [2.5, 3.5],
                [2.5, 4.],
                [2.5, 4.5]
            ], dtype=tf.float32)
               # THIS LIST INCLUDES ALL 45 [X,Y] POINTS IN points ARRAY IN PROJECT WRITEUP


            meanPois_res_w = 0.
            meanPois_res_T = 0.
            #meanwlimPen = 0.  #ASSOCIATED WITH PENALTY CALC COMMENTED_OUT HERE BELOW
            for _ in range(45):
                ##compute Laplacian/Poisson residual here - THIS LIST IS DONE ONCE PER epoch

                # Use tf.random.uniform to sample an index for the interior point
                idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)
                selected_point = tf.gather(points2, idx)
                #x_interior, y_interior = selected_point[0], selected_point[1]
                #=======================================
                input_point = tf.stack([selected_point[0], selected_point[1]])    # shape:
                    (2,)
                input_point = tf.reshape(input_point, (1, 2))  # shape: (1, 2) for batch

                # First and second derivatives - extraction from model have output of shape
                    (2)
                with tf.GradientTape(persistent=True) as tape2:
                    tape2.watch(input_point)
                    with tf.GradientTape(persistent=True) as tape1:
                        tape1.watch(input_point)
                        output = model(input_point)    # shape: (1, 2)
                        wpred = output[0, 0]
                        Tpred = output[0, 1]

                    # First-order derivatives
                    dw_dx, dw_dy = tape1.gradient(wpred, input_point)[0]
                    dT_dx, dT_dy = tape1.gradient(Tpred, input_point)[0]

                # Second-order derivatives
                d2w_dx2, d2w_dy2 = tape2.gradient([dw_dx, dw_dy], input_point)[0]
                d2T_dx2, d2T_dy2 = tape2.gradient([dT_dx, dT_dy], input_point)[0]

                # Clean up
                del tape1
                del tape2
                #=======================================



                # Calculate the Laplacian (sum of second derivatives)
                laplacian_w = d2w_dx2 + d2w_dy2
                laplacian_T = d2T_dx2 + d2T_dy2

                #shape_tensor = tf.shape(dw_dx)
                #tf.print("Shape of dw_dx:", shape_tensor)
                #shape_tensor = tf.shape(dw_dy)
                #tf.print("Shape of dw_dy:", shape_tensor)

                #ADD PRESSURE GRADIENT TERM dPmdzomu = dPdz over mu (POISSON EQ for w)
                dPmdzomu = -80.0
                # Compute the mean w Poisson residual for the 45 points( sum of squared (
                    Laplacian - dPmdzomu) )
                meanPois_res_w +=  K.square((laplacian_w - dPmdzomu))/45.

                #ADD TEMPERATURE GRADIENT TERM w*dTmdzoalpha = (w*dTdz over alpha) (POISSON
                    EQ)
                #  dTmdzoalpha ~ sdegC/mm**3
                dTmdzoalpha = 0.103
                # Compute the mean T Poisson residual for the 45 points( sum of squared (
                    Laplacian - w_pred*dTmdzoalpha) )
                meanPois_res_T +=  K.square((laplacian_T - wpred*dTmdzoalpha))/45.


            #Combine boundary and interior Poisson Eq. residual losses
            total_loss = boundary_loss + kappa * meanPois_res_w + kappa * meanPois_res_T


        return total_loss
    return loss

# ======== END CUSTOM LOSS FUNCTION DEFINITION ==========


from tensorflow.keras.callbacks import Callback

class EpochTracker(Callback):
    def __init__(self, N_epInit):
        super().__init__()
        self.N_epInit = N_epInit
        self.epochs_trained = 0

    def on_epoch_end(self, epoch, logs=None):
        self.epochs_trained = epoch

# Set the loss function with the required arguments
#    Set N_epInit=2000 for first run of fit routine in next cell,
#    then set it then set N_epInit=1 for all runs of fit routine  beyond that

epoch_tracker = EpochTracker(N_epInit=1)

custom_loss = custom_loss_function(kappa=1.3, x_max=3.0, y_max=5.0, epoch_tracker=
    epoch_tracker)



# Compile the model with the custom loss
model.compile(optimizer='adam', loss=custom_loss)

#### After the compilation of the model, run the fit method with some number of epochs (can
    be run iteratively.


#The fit method takes three parameters; namely, x, y, and number of epochs.
#During model training, if all the batches of data are seen by the model once,
#we say that one epoch has been completed.


##### RUN FIT

# Assuming ‘boundary_points‘ and ‘boundary_w‘ are TensorFlow tensors

# Now, fit the model and pass the callback to ‘fit()‘
#historyData = model.fit(boundary_points_xtrain, boundary_w_ytrain,
#                 epochs=2, batch_size=32,
#                 callbacks=[epoch_tracker])
historyData = model.fit(tf_tensor_inputs, tf_tensor_outputs,
                epochs=2500, batch_size=32,
                callbacks=[epoch_tracker])
```

```python


loss_hist = historyData.history['loss']
#The above line will return a dictionary, access it's info like this:
best_epoch = np.argmin(historyData.history['loss']) + 1
print ('best epoch = ', best_epoch)
print('smallest loss =', np.min(loss_hist))

predictions = model.predict(data_inputs)

predictions = model.predict(data_inputs)

#SETTING UP PLOT
%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 8] # for square canvas
#========

'''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''

#CAN'T USE THIS LOG-LOG FOR W BECUASE DATA VALUES ARE ALL ZERO

plt.scatter(predictions[:, 0], data_outputs[:, 0])
#plt.title('w,T PINN training of Neural Network ==> y = w')
plt.xlabel('predicted output for NN (mm/s)')
plt.ylabel('data output (mm/s)')
#plt.loglog()
#plt.xlim(xmax = 100, xmin = 1.)
#plt.ylim(ymax = 100, ymin = 1.)
# Generate red y=x line
#x_data = np.linspace(1.0, 100.0, num=3)

plt.xlim(xmax = 10, xmin = -10.)
plt.ylim(ymax = 10, ymin = -10.)
# Generate red y=x line
#x_data = np.linspace(1.0, 100.0, num=3)
x_data = np.linspace(-10., 10.0, num=3)
y_data = x_data
plt.plot(x_data, y_data, color='red')
plt.show()
#PLOTS VELOCITY BOUNDARY DATA ONLY

# w velocity field
test = []
outpt=[]
predictions = model.predict(data_inputs)

#first point (row [0])comparison of data and prediction
# put in a loop to print comparion for all data points

xa=[]
ya=[]
za=[]

xao = np.linspace(0, 3, 7)  # 1D array for x values
yao = np.linspace(0, 5, 11)    # 1D array for y values
xa = np.linspace(0, 10, 77)    # 1D array for x values
ya = np.linspace(0, 10, 77)    # 1D array for y values
za = np.linspace(0, 10, 77)    # 1D array for z values

x =-0.50
ia=-1
while (x < 2.51):
    x = x + 0.5
    y = -0.5
    while (y < 4.51):
        ia=ia+1
        y = y + 0.50
        xa[ia]=x
        ya[ia]=y

        test = [[ x , y ]]
        testarray = np.array(test)
        outpt = model.predict(testarray)
        za[ia] = outpt[0][0]
        print ('x= ', testarray[0][0], ', y= ', testarray[0][1],'w= ', outpt[0][0])
        #Predoutput[0, 0] ??


X, Y, Z = xa, ya, za

import sys
#import csv
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

# Plot X,Y,Z

# Creating figure
#fig = plt.figure(figsize =(16, 9))
#ax = plt.axes(projection ='3d')

fig = plt.figure()
#ax = fig.add_subplot(111, projection='3d')
ax = plt.axes(projection ='3d')
#ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
ax.plot_trisurf(X, Y, Z,
                linewidth = 0.2,
                antialiased = True
                , cmap='viridis');
ax.invert_yaxis()
ax.invert_xaxis()
plt.show()



# Plot the data
plt.figure(figsize=(5, 6))
plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
plt.colorbar(label='w value')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Heat Map of w values (mm/s) over x and y domain (mm)')
plt.show()

#temperature field T
test = []
outpt=[]
predictions = model.predict(data_inputs)

#first point (row [0])comparison of data and prediction
# put in a loop to print comparion for all data points

xa=[]
ya=[]
za=[]

xao = np.linspace(0, 3, 7)  # 1D array for x values
yao = np.linspace(0, 5, 11)    # 1D array for y values
xa = np.linspace(0, 10, 77)    # 1D array for x values
ya = np.linspace(0, 10, 77)    # 1D array for y values
za = np.linspace(0, 10, 77)    # 1D array for z values

x =-0.50
```

```
ia=-1
while (x < 2.51):
    x = x + 0.5
    y = -0.5
    while (y < 4.51):
        ia=ia+1
        y = y + 0.50
        xa[ia]=x
        ya[ia]=y

        test = [[ x , y ]]
        testarray = np.array(test)
        outpt = model.predict(testarray)
        za[ia] = outpt[0][1]
        print ('x (mm) = ', testarray[0][0], ', y (mm) = ', testarray[0][1],'T (degC) = ',
            outpt[0][1])
        #Predoutput[0, 0] ??


X, Y, Z = xa, ya, za

import sys
#import csv
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

# Plot X,Y,Z

# Creating figure
#fig = plt.figure(figsize =(16, 9))
#ax = plt.axes(projection ='3d')

fig = plt.figure()
#ax = fig.add_subplot(111, projection='3d')
ax = plt.axes(projection ='3d')
#ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
ax.plot_trisurf(X, Y, Z,
                linewidth = 0.2,
                antialiased = True
                , cmap='viridis');
ax.invert_yaxis()
ax.invert_xaxis()
plt.show()



# Plot the data
plt.figure(figsize=(5, 6))
plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
plt.colorbar(label='T value')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Heat Map of T values (degC) over x and y domain (mm)')
plt.show()
```

9.3 Part 2 PINN Code Rounded

```
# CodeP3Pt2B.1PINNconvSp25.ipynb - V.P. Carey, ME250B Sp25

#import useful packages
import keras
import pandas as pd
from keras.models import Sequential
import numpy as np
import keras.backend as kb
import tensorflow as tf
#the following 2 lines are only needed for Mac OS machines
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'

#create input data array

# x(mm), y(mm)   These are (x,y) coordinates of boundary location where w velocity and T are
     specified
xdata = []
ND = 32
xdata =  [  [0.293, 0.293], [0.134, 0.500], [ 0.0, 1.0 ], [ 0.0, 1.5 ],[ 0.0, 2.0 ], [ 0.0,
    2.5 ], [ 0.0, 3.0 ],
            [ 0.0, 3.5 ], [ 0.0, 4.0 ], [0.134, 4.500], [0.293, 4.707],
            [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 ],
            [ 3.0, 3.5 ], [ 3.0, 4.0 ], [2.866, 4.500], [2.707, 4.707]]

xdata.append([0.500, 0.134])
xdata.append([  1.0, 0.0 ])
xdata.append([  1.5, 0.0 ])
xdata.append([  2.0, 0.0 ])
xdata.append([2.500, 0.134])

xdata.append([0.500, 4.866])
xdata.append([  1.0, 5.0 ])
xdata.append([  1.5, 5.0 ])
xdata.append([  2.0, 5.0 ])
xdata.append([2.500, 4.866])

xarray= np.array(xdata)
print (xdata)
print (xarray)


# w (mm/s)   These are specified velocity values (mm/s) and temperature T (deg C) for each
    boundary pair of (x,y) above

ydata = []

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.  ] ]

ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90. ])
ydata.append([ 0.0, 90.  ])

ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])
ydata.append([ 0.0, 90.  ])


yarray= np.array(ydata)
print (ydata)
print (yarray)

data_inputs = np.array(xdata)
data_outputs = np.array(ydata)


# Convert to TensorFlow tensors
```

```
tf_tensor_inputs = tf.convert_to_tensor(data_inputs, dtype=tf.float32)
tf_tensor_outputs = tf.convert_to_tensor(data_outputs, dtype=tf.float32)
shape_tensor = tf.shape(tf_tensor_inputs)
print("Shape of tf_tensor_inputs:", shape_tensor)
shape_tensor = tf.shape(tf_tensor_outputs)
print("Shape of tf_tensor_outputs:", shape_tensor)

# define neural network model

    #As seen below, we have created four dense layers.
    #A dense layer is a layer in neural network thats fully connected.
    #In other words, all the neurons in one layer are connected to all other neurons in the
        next layer.
    #In the first layer, we need to provide the input shape, which is 1 in our case.
    #The activation function we have chosen is elu, which stands for exponential linear unit
        .

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

import tensorflow as tf
# Set the seed when creating the initializer
initializer = tf.keras.initializers.RandomUniform(minval=-2.9, maxval=2.9, seed=42)

# Create your model (for example, a simple fully connected feedforward NN)
# Assume 2D input (x, y),  Output: z-direction velocity w
model = Sequential([
        Dense(5, input_dim=2, activation='elu', kernel_initializer=initializer),
        Dense(12, activation='elu',kernel_initializer=initializer),
        Dense(8, activation='elu',kernel_initializer=initializer),
        Dense(2)
    ])

# Set the kappa value (this can be adjusted as per your needs)
kappa = 1.3

# Set the x_max and y_max values of your domain in mm (can adjust these according to your
    problem)
x_max = 3.0
y_max = 5.0

#print summary of model features
model.summary()

# PINN CUSTOM LOSS FUNCTION - V.P. Carey, ME250B Sp25

import tensorflow as tf
from tensorflow.keras import backend as K

def custom_loss_function(kappa, x_max, y_max, epoch_tracker):
        # Custom loss function for the neural network model.
        #- kappa: weight for the Poisson residual term.
        #- x_max, y_max: maximum x and y coordinates of the domain in mm.
        #- epoch_tracker: callback object to keep track of the current epoch.

    def loss(y_true, y_pred):
        #ALWAYS MATCH BOUNDARY VALUES HERE
        # Define pointsBoundaryChk as a constant tensor

        pointsBoundaryChk = tf.constant([
            [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 ],
            [ 3.0, 3.5 ],
            [ 3.0, 4.0 ],
            [2.866, 4.500],
            [2.707, 4.707],
            [0.293, 0.293],
            [0.134, 0.500],
            [ 0.0, 1.0 ],
            [ 0.0, 1.5 ],
            [ 0.0, 2.0 ],
            [ 0.0, 2.5 ],
            [ 0.0, 3.0 ],
            [ 0.0, 3.5 ],
            [ 0.0, 4.0 ],
            [0.134, 4.500],
            [0.293, 4.707],
            [0.500, 0.134],
            [ 1.0, 0.0 ],
        #ADD x,y PAIRS DATA HERE SO TENSOR LOOKS LIKE Fig. 3pt2B.3 IN WRITEUP - SHOULD BE 32
             PAIRS TOTAL
            [ 1.5, 0.0 ],
            [ 2.0, 0.0 ],
            [2.500, 0.134],
            [0.500, 4.866],
            [ 1.0, 5.0 ],
            [ 1.5, 5.0 ],
            [ 2.0, 5.0 ],
            [2.500, 4.866]
        ], dtype=tf.float32)


        # Boundary loss: Calculate squared error for the boundary condition
        boundary_loss = 2.0*K.mean(K.square(y_pred - y_true))


        ################

        # 2 SECOND STEP OPTIONS DEPENDING ON epoch

        # Check epoch from the callback
        epoch = epoch_tracker.epochs_trained
        # FOR EARLY EPOCHS
        if epoch < epoch_tracker.N_epInit:

            #NEXT FIND LOSS DUE TO MATCH OF INTERIOR POINTS TO TARGET w VELOCITY FIELD
            # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit
            points = tf.constant([
                [0.5, 0.5, 16., 78.40579710144928],
                [0.5, 1.0, 28.444445, 69.38808373590982],
                [0.5, 1.5, 37.333332, 62.94685990338164],
                [0.5, 2., 42.666668, 59.08212560386473],
                [0.5, 2.5, 44.444443, 57.7938808373591],
                [0.5, 3., 42.666668, 59.08212560386473],
                [0.5, 3.5, 37.333332, 62.94685990338164],
                [0.5, 4., 28.444445, 69.38808373590982],
                [0.5, 4.5, 16., 78.40579710144928],
                [1.0, 0.5, 25.6, 71.44927536231884],
                [1.0, 1.0, 45.511112, 57.02093397745571],
                [1.0, 1.5, 59.733334, 46.71497584541062],
                [1.0, 2., 68.26667, 40.53140096618357],
                [1.0, 2.5, 71.1111155, 38.47020933977455],
                [1.0, 3., 68.26667, 40.53140096618357],
                [1.0, 3.5, 59.733334, 46.71497584541062],
                [1.0, 4., 45.511112, 57.02093397745571],
                [1.0, 4.5, 25.6, 71.44927536231884],
                [1.5, 0.5, 28.8, 69.13043478260869],
                [1.5, 1.0, 51.2, 52.89855072463767],
                [1.5, 1.5, 67.2, 41.30434782608695],
                [1.5, 2., 76.8, 34.347826086956516],
                [1.5, 2.5,  80., 32.028985507246375],
                [1.5, 3., 76.8, 34.347826086956516],
                [1.5, 3.5, 67.2, 41.30434782608695],
                [1.5, 4., 51.2, 52.89855072463767],
                [1.5, 4.5, 28.8, 69.13043478260869],
                [2., 0.5, 25.6, 71.44927536231884 ],
                [2., 1.0, 45.511112, 57.02093397745571],
                [2., 1.5, 59.733334, 46.71497584541062],
                [2., 2., 68.2666, 40.53140096618357],
                [2., 2.5, 71.111115, 38.47020933977455],
                [2., 3., 68.26667, 40.53140096618357],
                [2., 3.5, 59.733334, 46.71497584541062],
                [2., 4., 45.511112, 57.02093397745571],
                [2., 4.5, 25.6, 71.44927536231884],
                [2.5, 0.5, 16., 78.40579710144928],
                [2.5, 1.0, 28.444445, 69.38808373590982],
                [2.5, 1.5, 37.333332, 62.94685990338164],
                [2.5, 2., 42.666668, 59.08212560386473],
                [2.5, 2.5, 44.444443, 57.7938808373591],
                [2.5, 3., 42.666668, 59.08212560386473],
                [2.5, 3.5, 37.333332, 62.94685990338164],
                [2.5, 4., 28.444445, 69.38808373590982],
                [2.5, 4.5, 16., 78.40579710144928]
            ], dtype=tf.float32)

            meanIntTarg_res = 0.
            for _ in range(45):
                # does this list 45 times not keeping track of index
                # THIS COMPUTES INTERIOR POINT TARGET SOLUTION LOSS FUNCTION
                # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch < epoch_tracker.N_epInit

                # Randomly select one point
                idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)
                selected_point = tf.gather(points, idx)  # shape: [3]

                # Slice to get only the first two columns: x and y
                xy_input = selected_point[:2]  # shape: [2]

                # Prepare input for the model: [1, 2]
                model_input = tf.reshape(xy_input, shape=(1, 2))

                #retreive w target value
                w_targInt = selected_point[2]
                #retreive T target value
                T_targInt = selected_point[3]

                # Compute the output
                Predoutput = model(model_input)
                wpred = Predoutput[0, 0]
                Tpred = Predoutput[0, 1]
                # add loss contributions
                meanIntTarg_res += (wpred - w_targInt)*(wpred - w_targInt)/45.
                meanIntTarg_res += (Tpred - T_targInt)*(Tpred - T_targInt)/45.


            #Combine mean boundary and mean interior point target losses
            total_loss = boundary_loss + meanIntTarg_res


        else:
            # DO THIS IF CURRENT epoch IS GREATER THAN N_epInit,
            # THIS COMPUTES THE PDE RESIDUAL LOSS FUNCTION at 45 INTERIOR POINTS AND
            # SKIPS ERROR CALC FOR TARGET GUESSED w FIELD AT INTERIOR POINTS
            # HERE IT IS DOING THIS ONCE PER EPOCH IF epoch > epoch_tracker.N_epInit

            #>>> Randomly select an interior point from this list using TensorFlow's random uniform
            points2 = tf.constant([
                [0.5, 0.5],
                [0.5, 1.0],
                [0.5, 1.5],
                [0.5, 2.],
                [0.5, 2.5],
                [0.5, 3.],
                [0.5, 3.5],
                [0.5, 4.],
                [0.5, 4.5],
                [1.0, 0.5],
                [1.0, 1.0],
                [1.0, 1.5],
                [1.0, 2.],
                [1.0, 2.5],
                [1.0, 3.],
                [1.0, 3.5],
                [1.0, 4.],
                [1.0, 4.5],
                [1.5, 0.5],
                [1.5, 1.0],
                [1.5, 1.5],
                [1.5, 2.],
                [1.5, 2.5],
                [1.5, 3.],
                [1.5, 3.5],
                [1.5, 4.],
                [1.5, 4.5],
                [2., 0.5],
                [2., 1.0],
                [2., 1.5],
                [2., 2.],
                [2., 2.5],
                [2., 3.],
                [2., 3.5],
                [2., 4.],
                [2., 4.5],
                [2.5, 0.5],
                [2.5, 1.0],
                [2.5, 1.5],
                [2.5, 2.],
                [2.5, 2.5],
                [2.5, 3.],
                [2.5, 3.5],
                [2.5, 4.],
                [2.5, 4.5]
            ], dtype=tf.float32)
               # THIS LIST INCLUDES ALL 45 [X,Y] POINTS IN points ARRAY IN PROJECT WRITEUP


            meanPois_res_w = 0.
            meanPois_res_T = 0.
            #meanwlimPen = 0.  #ASSOCIATED WITH PENALTY CALC COMMENTED_OUT HERE BELOW
            for _ in range(45):
                ##compute Laplacian/Poisson residual here - THIS LIST IS DONE ONCE PER epoch

                # Use tf.random.uniform to sample an index for the interior point
                idx = tf.random.uniform(shape=[], minval=0, maxval=45, dtype=tf.int32)
                selected_point = tf.gather(points2, idx)
                #x_interior, y_interior = selected_point[0], selected_point[1]
                #=====================================
                input_point = tf.stack([selected_point[0], selected_point[1]])    # shape:
                    (2,)
                input_point = tf.reshape(input_point, (1, 2))  # shape: (1, 2) for batch

                # First and second derivatives - extraction from model have output of shape
                    (2)
                with tf.GradientTape(persistent=True) as tape2:
                    tape2.watch(input_point)
                    with tf.GradientTape(persistent=True) as tape1:
                        tape1.watch(input_point)
                        output = model(input_point)    # shape: (1, 2)
                        wpred = output[0, 0]
                        Tpred = output[0, 1]

                    # First-order derivatives
                    dw_dx, dw_dy = tape1.gradient(wpred, input_point)[0]
                    dT_dx, dT_dy = tape1.gradient(Tpred, input_point)[0]

                # Second-order derivatives
                d2w_dx2, d2w_dy2 = tape2.gradient([dw_dx, dw_dy], input_point)[0]
                d2T_dx2, d2T_dy2 = tape2.gradient([dT_dx, dT_dy], input_point)[0]

                # Clean up
                del tape1
                del tape2
                #=======================================



                # Calculate the Laplacian (sum of second derivatives)
                laplacian_w = d2w_dx2 + d2w_dy2
                laplacian_T = d2T_dx2 + d2T_dy2

                #shape_tensor = tf.shape(dw_dx)
                #tf.print("Shape of dw_dx:", shape_tensor)
                #shape_tensor = tf.shape(dw_dy)
                #tf.print("Shape of dw_dy:", shape_tensor)

                #ADD PRESSURE GRADIENT TERM dPmdzomu = dPdz over mu (POISSON EQ for w)
                dPmdzomu = -80.0
                # Compute the mean w Poisson residual for the 45 points( sum of squared (
                    Laplacian - dPmdzomu) )
                meanPois_res_w +=  K.square((laplacian_w - dPmdzomu))/45.

                #ADD TEMPERATURE GRADIENT TERM w*dTmdzoalpha = (w*dTdz over alpha) (POISSON
                    EQ)
                #  dTmdzoalpha ~ sdegC/mm**3
                dTmdzoalpha = 0.103
                # Compute the mean T Poisson residual for the 45 points( sum of squared (
                    Laplacian - w_pred*dTmdzoalpha) )
                meanPois_res_T +=  K.square((laplacian_T - wpred*dTmdzoalpha))/45.


            #Combine boundary and interior Poisson Eq. residual losses
            total_loss = boundary_loss + kappa * meanPois_res_w + kappa * meanPois_res_T


        return total_loss
    return loss

# ======== END CUSTOM LOSS FUNCTION DEFINITION ==========


from tensorflow.keras.callbacks import Callback

class EpochTracker(Callback):
    def __init__(self, N_epInit):
        super().__init__()
        self.N_epInit = N_epInit
        self.epochs_trained = 0

    def on_epoch_end(self, epoch, logs=None):
        self.epochs_trained = epoch

# Set the loss function with the required arguments
#    Set N_epInit=2000 for first run of fit routine in next cell,
#    then set it then set N_epInit=1 for all runs of fit routine  beyond that
```

```python

epoch_tracker = EpochTracker(N_epInit=1)

custom_loss = custom_loss_function(kappa=1.3, x_max=3.0, y_max=5.0, epoch_tracker=
    epoch_tracker)



# Compile the model with the custom loss
model.compile(optimizer='adam', loss=custom_loss)

#### After the compilation of the model, run the fit method with some number of epochs (can
    be run iteratively.


#The fit method takes three parameters; namely, x, y, and number of epochs.
#During model training, if all the batches of data are seen by the model once,
#we say that one epoch has been completed.


##### RUN FIT

# Assuming `boundary_points` and `boundary_w` are TensorFlow tensors

# Now, fit the model and pass the callback to `fit()`
#historyData = model.fit(boundary_points_xtrain, boundary_w_ytrain,
#                 epochs=2, batch_size=32,
#                 callbacks=[epoch_tracker])
historyData = model.fit(tf_tensor_inputs, tf_tensor_outputs,
                epochs=2500, batch_size=32,
                callbacks=[epoch_tracker])


loss_hist = historyData.history['loss']
#The above line will return a dictionary, access it's info like this:
best_epoch = np.argmin(historyData.history['loss']) + 1
print ('best epoch = ', best_epoch)
print('smallest loss =', np.min(loss_hist))

predictions = model.predict(data_inputs)

predictions = model.predict(data_inputs)

#SETTING UP PLOT
%matplotlib inline
# importing the required module
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 8] # for square canvas
#========

'''CALCULATE PREDICTED VALUES AND RETRIEVE DATA VALUES TO PLOT'''

#CAN'T USE THIS LOG-LOG FOR W BECUASE DATA VALUES ARE ALL ZERO

plt.scatter(predictions[:, 0], data_outputs[:, 0])
#plt.title('w,T PINN training of Neural Network ==> y = w')
plt.xlabel('predicted output for NN (mm/s)')
plt.ylabel('data output (mm/s)')
#plt.loglog()
#plt.xlim(xmax = 100, xmin = 1.)
#plt.ylim(ymax = 100, ymin = 1.)
# Generate red y=x line
#x_data = np.linspace(1.0, 100.0, num=3)

plt.xlim(xmax = 10, xmin = -10.)
plt.ylim(ymax = 10, ymin = -10.)
# Generate red y=x line
#x_data = np.linspace(1.0, 100.0, num=3)
```

```python
x_data = np.linspace(-10., 10.0, num=3)
y_data = x_data
plt.plot(x_data, y_data, color='red')
plt.show()
#PLOTS VELOCITY BOUNDARY DATA ONLY

# w velocity field
test = []
outpt=[]
predictions = model.predict(data_inputs)

#first point (row [0])comparison of data and prediction
# put in a loop to print comparion for all data points

xa=[]
ya=[]
za=[]

xao = np.linspace(0, 3, 7)  # 1D array for x values
yao = np.linspace(0, 5, 11)    # 1D array for y values
xa = np.linspace(0, 10, 77)    # 1D array for x values
ya = np.linspace(0, 10, 77)    # 1D array for y values
za = np.linspace(0, 10, 77)    # 1D array for z values

x =-0.50
ia=-1
while (x < 2.51):
    x = x + 0.5
    y = -0.5
    while (y < 4.51):
        ia=ia+1
        y = y + 0.50
        xa[ia]=x
        ya[ia]=y

        test = [[ x , y ]]
        testarray = np.array(test)
        outpt = model.predict(testarray)
        za[ia] = outpt[0][0]
        print ('x= ', testarray[0][0], ', y= ', testarray[0][1],'w= ', outpt[0][0])
        #Predoutput[0, 0] ??


X, Y, Z = xa, ya, za

import sys
#import csv
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

# Plot X,Y,Z

# Creating figure
#fig = plt.figure(figsize =(16, 9))
#ax = plt.axes(projection ='3d')

fig = plt.figure()
#ax = fig.add_subplot(111, projection='3d')
ax = plt.axes(projection ='3d')
#ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
ax.plot_trisurf(X, Y, Z,
                linewidth = 0.2,
                antialiased = True
                , cmap='viridis');
ax.invert_yaxis()
ax.invert_xaxis()
plt.show()
```

```python



# Plot the data
plt.figure(figsize=(5, 6))
plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
plt.colorbar(label='w value')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Heat Map of w values (mm/s) over x and y domain (mm)')
plt.show()

#temperature field T
test = []
outpt=[]
predictions = model.predict(data_inputs)

#first point (row [0])comparison of data and prediction
# put in a loop to print comparion for all data points

xa=[]
ya=[]
za=[]

xao = np.linspace(0, 3, 7)  # 1D array for x values
yao = np.linspace(0, 5, 11)    # 1D array for y values
xa = np.linspace(0, 10, 77)    # 1D array for x values
ya = np.linspace(0, 10, 77)    # 1D array for y values
za = np.linspace(0, 10, 77)    # 1D array for z values

x =-0.50
ia=-1
while (x < 2.51):
    x = x + 0.5
    y = -0.5
    while (y < 4.51):
        ia=ia+1
        y = y + 0.50
        xa[ia]=x
        ya[ia]=y

        test = [[ x , y ]]
        testarray = np.array(test)
        outpt = model.predict(testarray)
        za[ia] = outpt[0][1]
        print ('x (mm) = ', testarray[0][0], ', y (mm) = ', testarray[0][1],'T (degC) = ',
            outpt[0][1])
        #Predoutput[0, 0] ??


X, Y, Z = xa, ya, za

import sys
#import csv
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

# Plot X,Y,Z

# Creating figure
#fig = plt.figure(figsize =(16, 9))
#ax = plt.axes(projection ='3d')

fig = plt.figure()
#ax = fig.add_subplot(111, projection='3d')
ax = plt.axes(projection ='3d')
#ax.plot_trisurf(X, Y, Z, color='white', edgecolors='grey', alpha=0.5)
```

```
ax.plot_trisurf(X, Y, Z,
                linewidth = 0.2,
                antialiased = True
                , cmap='viridis');
ax.invert_yaxis()
ax.invert_xaxis()
plt.show()



# Plot the data
plt.figure(figsize=(5, 6))
plt.tripcolor(xa, ya, za, shading='flat', cmap='viridis')
plt.colorbar(label='T value')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Heat Map of T values (degC) over x and y domain (mm)')
plt.show()
```

9.4 Part 2 FDM Code (Project 1)

```
% Gauss-Seidel Solver for Fully Developed Heat Transfer in a Rectangular Channel
clear
clc
close all
% Define constants and parameters
Nx = 30;  % Number of nodes in x direction
Ny = 50;  % Number of nodes in y direction
dx = 0.1e-3;  % Grid spacing in x (m)
dy = 0.1e-3;  % Grid spacing in y (m)
Tw = 90;       % Wall temperature ( C )
nu = 8.26e-7;%9.00e-7 ;  % Kinematic viscosity (m^2/s)
rho = 997; %1055;      % Density (kg/m^3)
alpha = 1.46e-7; %1.08e-7; % Thermal diffusivity (m^2/s)
cp = 4164; %3559;      % Specific heat (J/kg C)
k = 0.606;      % Thermal conductivity (W/m C)
dPdz = -80 * nu * rho * 1000;  % Pressure gradient (Pa/m)
dTmdz = alpha * .103 * 1000000000;   % Mean temperature gradient ( C /m)

% Convergence criteria
epsilon_w = 0.0001;
epsilon_T = 0.05;

% Initialize fields
w = zeros(Nx, Ny);
T = 70 * ones(Nx, Ny);  % Initial temperature field
T(1,:) = Tw;
T(Nx,:) = Tw;
T(:,1) = Tw;
T(:,Ny) = Tw;

% Gauss-Seidel iterations for velocity field
converged_w = false;
while ~converged_w
    max_change_w = 0;
    for i = 2:Nx-1
        for j = 2:Ny-1
            w_new = ((dy/dx)^2 * (w(i+1,j) + w(i-1,j)) + w(i,j+1) + w(i,j-1)) / ...
                    (2*(dy/dx)^2 + 2) - (dPdz * dy^2) / (rho * nu * (2*(dy/dx)^2 + 2));
            max_change_w = max(max_change_w, abs(w_new - w(i,j)));
            w(i,j) = w_new;
        end
    end
    if max_change_w < epsilon_w
        converged_w = true;
    end
```

```
end

% Gauss-Seidel iterations for temperature field
converged_T = false;
while ~converged_T
    max_change_T = 0;
    for i = 2:Nx-1
        for j = 2:Ny-1
            T_new = ((dy/dx)^2 * (T(i+1,j) + T(i-1,j)) + T(i,j+1) + T(i,j-1)) / ...
                    (2*(dy/dx)^2 + 2) - (w(i,j) * dTmdz * dy^2) / (alpha * (2*(dy/dx)^2 + 2)
                        );
            max_change_T = max(max_change_T, abs(T_new - T(i,j)));
            T(i,j) = T_new;
        end
    end
    if max_change_T < epsilon_T
        converged_T = true;
    end
end

%% Post-processing
% Numerical integration for mean temp and mean velocity
% all domain nodes now have one unit area
A = dx * dy * ones(Nx, Ny);
% all boundary nodes now have 0.5 unit area
A(1,:)   = 0.5 * dx * dy;   % left wall
A(end,:) = 0.5 * dx * dy;   % right wall
A(:,1)   = 0.5 * dx * dy;   % bottom wall
A(:,end) = 0.5 * dx * dy;   % top wall
A0 = Nx*Ny*dx*dy;
wm = sum(sum(w .* A)) / A0;
% All nodes contribute dxdy
Tm = sum(sum(w .* T .* A)) / (wm * A0);
% Compute local heat flux and heat transfer coefficient
qw = zeros(Nx, Ny);
h = zeros(Nx, Ny);
for i = 2:Nx-1
    qw(i,1) = k * (T(i,1) - T(i,2)) / dy; % bottom wall
    qw(i,Ny) = k * (T(i,Ny) - T(i,Ny-1)) / dy; % top wall
end
for j = 2:Ny-1
    qw(1,j) = k * (T(1,j) - T(2,j)) / dx; % left wall
    qw(Nx,j) = k * (T(Nx,j) - T(Nx-1,j)) / dx; % right wall
end

for i = 2:Nx-1
    for j = [1 Ny]
        h(i,j) = qw(i,j) / (T(i,j) - Tm);
    end
end
for j = 2:Ny-1
    for i = [1 Nx]
        h(i,j) = qw(i,j) / (T(i,j) - Tm);
    end
end

% Total heat input rate and mean heat transfer coefficient

A_wall = .5 * dx * dy * ones(Nx,Ny);
Q_total = sum(qw(:) .* A_wall(:));
Aw = 0.5 * dx * dy * (Nx + Nx + Ny + Ny);
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));
h_mean = Q_total / (Aw * (T_wall_avg - Tm));


% Generate 3D plots
[x_grid, y_grid] = meshgrid(linspace(0, (Nx)*dx, Nx), linspace(0, (Ny)*dy, Ny));
```

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

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

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

Dh = 4 * (A0) / (2*(Nx*dy+Ny*dx));
Nusselt = h_mean * Dh / k;
Reynolds =  wm * Dh / nu;

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

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