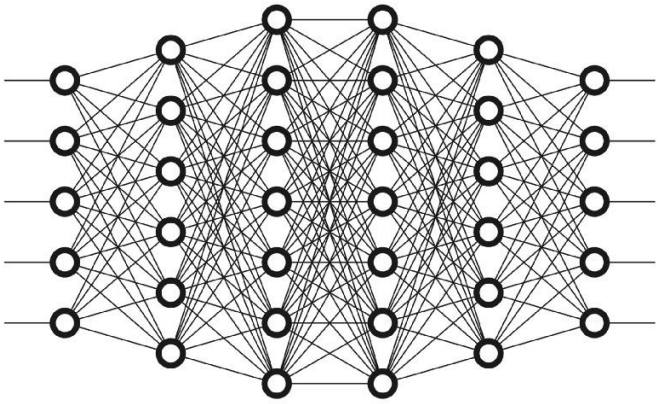
Department of Mechanical Engineering
Faculty of Engineering and Design

Final Year BEng Project Report (ME30227)

Use of Neural Networks to Solve Transient Heat Transfer in Film Cooling Experiments

Alen Abdrakhmanov



"I certify that I have read and understood the entry in the Student Handbook for the Department of Mechanical Engineering on Cheating and Plagiarism and that all material in this assignment is my own work, except where I have indicated with appropriate references."

Supervisor: Hui Tang
Assessor: Andrew Rees
Word Count: 4520
Author's Signature: *Abdrakhmanov*

SUMMARY

The aim of this project was to investigate the application of Artificial Neural Networks (ANNs), and particularly Physics-Informed Neural Networks (PINNs) for modelling transient heat transfer in film cooling applications. The primary objective of the research was to evaluate the performance of the PINN framework in comparison to a traditional Crank-Nicolson MATLAB solver and assess its potential for accurately capturing the complex behaviour of film cooling heat transfer. To achieve this objective, the project employed a multi-stage methodology, which involved a comprehensive review of the literature on neural network-based solutions for partial differential equations, the development of a custom Python code to implement the PINN framework, and the verification of the achieved result via comparison with an established finite differencing method. The results of the investigation demonstrated that the PINN framework is an effective tool for modelling the general behaviour of transient heat transfer in film cooling contexts, despite certain inaccuracies observed at the internal boundary surface of the model. Despite these discrepancies, the PINN model achieved a low RMS error of 0.003729, and a maximum percentage error of 1.03%, indicating its potential for approximating complex heat transfer processes. The limitations of the PINN framework were also identified, including a potential suboptimal network architecture and and inherent limitations of PINNs when applied to intricate boundary conditions. In conclusion, this investigation has demonstrated the promise and limitations of using PINNs for film cooling applications, further advancing the state-of-the-art in data-driven modelling and simulation techniques.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude and appreciation to Dr Hui Tang, whose expertise, guidance, and encouragement have been invaluable throughout the journey of completing this project. Dr Tang's exceptional knowledge of heat transfer and numerical modelling have been instrumental in shaping the direction of this investigation.

I would also like to extend my gratitude to Julio Stefano Nastasi, whose paper on *"A Deep Learning Approach to Efficiently Solve the Fourier Equation"* [1] has been a backbone for this project, and provided a solid platform for me to take the concepts from the paper and apply it to the practical applications of gas turbine film cooling.

NOMENCLATURE

ANN	Artificial Neural Network
PINN	Physics-Informed Neural Network
L	Turbine Blade Thickness
T_c	Temperature at the internal surface *(x=0)*
T_h	Temperature at the external surface *(x=L)*
h_i	Heat Transfer Coefficient at the internal surface *(x=0)*
h_e	Heat Transfer Coefficient at the external surface *(x=L)*
t	Time
Bi_i	Biot Number at the internal surface *(x=0)*
Bi_e	Biot Number at the external surface *(x=L)*
F_0	Fourier Number

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Film Cooling

Gas turbine blades operate in extremely high-temperature environments, with the temperature depending on the specific application and engine design. In modern gas turbines, the temperatures of the combustion gases can exceed 2000°C. The ability to operate at higher temperatures is a key factor in improving the efficiency of gas turbines [2], as it leads to an increase in the thermodynamic efficiency of the engine. Therefore, ongoing research and development efforts focus on enhancing the temperature capabilities of turbine blade materials and cooling techniques to achieve better performance in gas turbines.

One method of addressing this problem is film cooling. Film cooling is a crucial technique employed in various high-temperature applications, including gas turbine engines, industrial furnaces, and rocket propulsion systems, to protect critical components from thermal damage. By introducing a layer of cooler, less dense fluid - typically air, or coolant between the hot gases and the surface *(see Figure 1)*, film cooling effectively reduces heat transfer into the turbine surface and enhances the durability and performance of components exposed to extreme heat. This technique has significant relevance to industries such as aerospace, power generation, and automotive, where maintaining thermal stability and mitigating downtime and additional cost is paramount for safe and efficient operation.

The optimisation of film cooling systems has become a priority for engineers and researchers, as it directly impacts the efficiency and longevity of the components it serves. Traditional analytical methods for studying transient heat transfer in film cooling applications often struggle to capture the complex flow behaviour and thermal gradients involved. In recent years, neural networks have emerged as a promising alternative, offering the potential for more accurate and efficient simulations. By leveraging the inherent capabilities of neural networks to learn and adapt from data, this project aims to develop a reliable and efficient computational model to simulate transient heat transfer phenomena in film cooling applications. Implementing such a model could result in significant improvements in the design and performance of film cooling systems, ultimately benefiting the industries that rely on these systems for safe and effective operation.

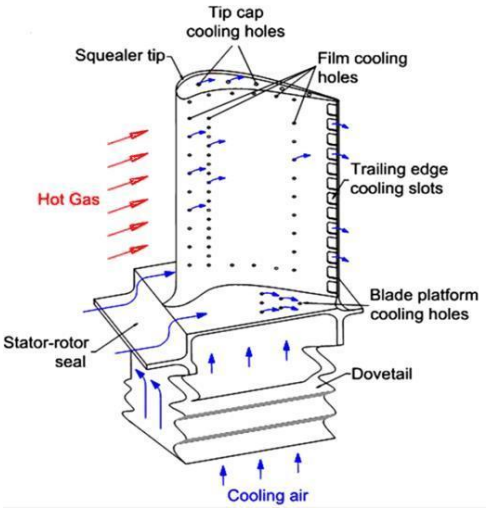


Figure 1: Diagram of Film Cooling in a Gas Turbine Blade [3]

1.2 Problem Definition

In order to investigate the application of artificial neural networks in modelling 1-dimensional heat transfer, the problem and associated parameters have to be clearly defined in the context of film cooling application as follows.

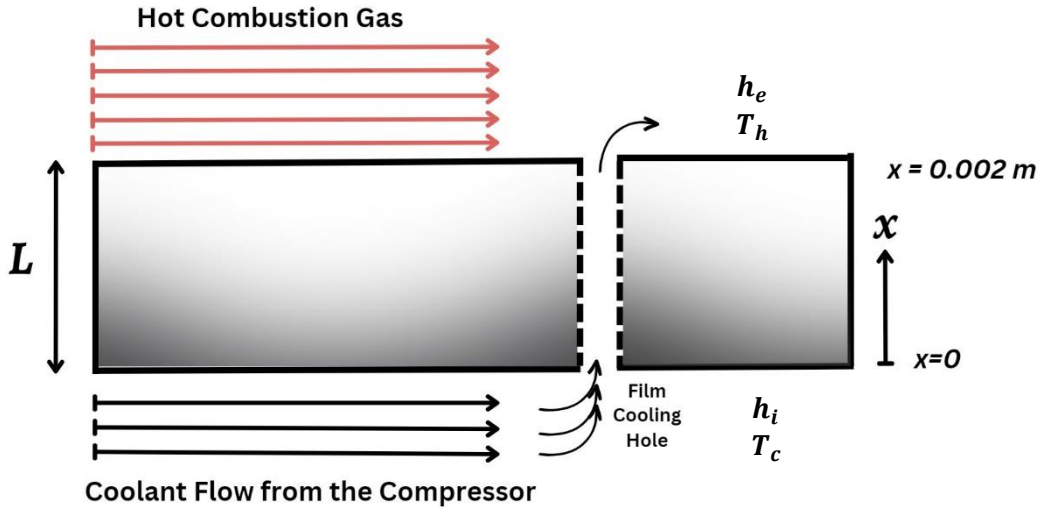


Figure 2: Schematic for the Considered Heat Transfer Problem

The spatial domain for the problem spans the thickness of a gas turbine blade, between $x = 0$ and $x = L$, where the thickness of the blade is defined as $L = 2mm$. The artificial neural network *(introduced in section 1.3)* will aim to solve 1-dimensional Fourier's Equation and model thermal evolution along the thickness of the blade, considering the temperature variations between the inner surface in contact with the coolant flow and the outer surface exposed to the hot combustion gases, for a time period of 5 seconds.

For this case, Fourier's heat equation is defined as:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \text{ [eq. 1]}$$

Where:

- T = Temperature as a function of time t and space x
- $\alpha = \frac{k}{\rho c}$ is the thermal diffusivity (m^2/s)
- k = 15 W/mK (Thermal conductivity)
- ρ = 8000 kg/m^3 (Typical nickel-based superalloy density)
- c = Specific heat capacity (J/kgK)

Prior to defining the initial and boundary conditions for this problem, dimensional analysis is carried out by creating dimensionless parameters that relate to the relevant physical quantities involved in the system. Converting the governing equation, the initial and boundary condition into non-dimensional form will generalise the behavior of the system and eliminate the need to incorporate thermal diffusivity into the neural network model.

In order to express the heat equation in dimensionless form, the following relationships are used:

$\tilde{T} = \frac{T - T_c}{T_h - T_c}$ *[eq. 2]* and $\tilde{x} = \frac{x}{L}$ *[eq. 3]*

Where:

- $\tilde{T}$ = Non-dimensional temperature
- T_c = 1000℃ [Temperature at the cold boundary of the considered domain (K)]
- T_h = 1200℃ [Temperature at the hot boundary of the considered domain (K)]
- $\tilde{x}$ = Non-dimensional position

The process is outlined below:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \Rightarrow \frac{\partial \tilde{T}}{\partial t} = \alpha \frac{\partial^2 \tilde{T}}{\partial x^2} \Rightarrow \frac{\partial \tilde{T}}{\partial \tilde{t}} = \frac{\alpha}{L^2} \frac{\partial^2 \tilde{T}}{\partial \tilde{x}^2} \Rightarrow \frac{\partial \tilde{T}}{\partial \left[\frac{\alpha t}{L^2}\right]} = \frac{\partial^2 \tilde{T}}{\partial x^2}$$

$$\Rightarrow \frac{\partial \tilde{T}}{\partial \tilde{t}} = \frac{\partial^2 \tilde{T}}{\partial \tilde{x}^2} \quad \textit{[eq. 4]}$$

$$\frac{\alpha t}{L^2} = F_0$$

Fourier Number *(F_0)* relates the time scale of the process to its corresponding length scale.

Hence, the following system of equations will govern the behavior of heat transfer:

$$\begin{cases} \frac{\partial \tilde{T}}{\partial \tilde{t}} = \frac{\partial^2 \tilde{T}}{\partial \tilde{x}^2} & [eq.5] \\ \frac{\partial \tilde{T}}{\partial \tilde{x}} = \widetilde{h_i}\left(\tilde{T} - \widetilde{T_c}\right) & [eq.6] \quad at\ x = 0 \\ \frac{\partial \tilde{T}}{\partial \tilde{x}} = -\widetilde{h_e}\left(\tilde{T} - \widetilde{T_h}\right) & [eq.\ 7] \quad at\ x = 1 \\ \tilde{T}(x, 0) = 1 & [eq.\ 8] \end{cases}$$

Where:

- $\widetilde{h_i}$ = Non-dimensional heat transfer coefficient at the internal boundary
- $\widetilde{h_e}$ = Non-dimensional heat transfer coefficient at the external boundary

Prior to incorporating the boundary conditions into the artificial neural network model *(See Section 3)*, it is necessary to evaluate the governing equations in terms of the dimensional heat transfer coefficients h_i and h_e, and obtain expressions for the equivalent non-dimensional parameters.

Starting with the non-dimensional governing equation for the internal boundary ($x = 0$):

$$k\frac{\partial T}{\partial x} = h_i\,(T - T_c) \quad \textit{[eq. 9]}$$

Dividing *[eq. 9]* by $h_i(T_h - T_c)$ produces:

$$\frac{k}{h_i(T_h - T_c)}\,\frac{\partial T}{\partial x} = \frac{h_i\,(T - T_c)}{h_i(T_h - T_c)} \Rightarrow \frac{k}{h_i(T_h - T_c)}\,\frac{\partial T}{\partial x} = \tilde{T} \textit{ [eq. 10]}$$

Using the relationships from *[eq. 2]* and *[eq. 3]* the equation is simplified to:

$$\frac{k}{h_i}\,\frac{\partial \bar{T}}{\partial \bar{x}} = \bar{T} \Rightarrow \frac{\partial \bar{T}}{\partial \bar{x}} = \frac{h_i L}{k}\,\bar{T}$$

$$\Rightarrow \frac{\partial \tilde{T}}{\partial \tilde{x}} = \widetilde{h_i}\,\tilde{T} \textit{ [eq. 11]}$$

Where $\widetilde{h_i} = \frac{h_i L}{k} = Bi_i$ (Biot Number at the internal boundary)

The Biot number is a key non-dimensional parameter in transient heat transfer analysis. It is defined as the ratio of the internal thermal resistance to the external thermal resistance in a system [4].

Non-dimensional governing equation for the external boundary ($x = 1$):

$$k\frac{\partial T}{\partial x} = -h_e\,(T - T_h) \textit{ [eq. 12]}$$

Dividing *[eq. 12]* by $h_e(T - T_h)$ produces:

$$\frac{k}{h_e(T - T_h)}\,\frac{\partial T}{\partial x} = -1 \Rightarrow \frac{k(T_h - T_c)}{L h_i(T_h - T_c)}\,\frac{\partial \tilde{T}}{\partial \tilde{x}} = -1 \Rightarrow \frac{\partial \tilde{T}}{\partial \tilde{x}} = \frac{-h_e L}{k}\,\frac{T - T_h}{T_h - T_c} \textit{ [eq. 13]}$$

Using the relationships from [1] the equation is simplified to:

$$\frac{\partial \tilde{T}}{\partial \tilde{x}} = \frac{-h_e L}{k}\,\frac{\tilde{T}(T_h - T_c) + T_c - T_h}{T_h - T_c} \Rightarrow \frac{\partial \tilde{T}}{\partial \tilde{x}} = \frac{-h_e L}{k}\left(\frac{\tilde{T}(T_h - T_c)}{T_h - T_c} + \frac{T_c - T_h}{T_h - T_c}\right)$$

$$\Rightarrow \frac{\partial \tilde{T}}{\partial \tilde{x}} = \frac{-h_e L}{k}\left(\tilde{T} + \frac{T_c - T_h}{T_h - T_c}\right)$$

$$\Rightarrow \frac{\partial \tilde{T}}{\partial \tilde{x}} = \frac{-h_e L}{k}\left(\tilde{T} - 1\right) \textit{ [eq. 14]}$$

Where $\widetilde{h_e} = -\frac{h_i L}{k} = Bi_e$ (Biot Number at the external boundary)

1.3 Introduction to Artificial Neural Network Theory

Artificial Neural Networks (ANNs) are a class of computational models inspired by the structure and functionality of biological neural networks. These models have gained immense popularity in recent years due to their ability to learn complex patterns and relationships within data, making them particularly suited for a wide range of applications, including image recognition, natural language processing, and prediction tasks.

The most basic building block of ANNs is the perceptron *(see Figure 3)*, a type of artificial neuron introduced by Frank Rosenblatt in 1958 [5]. A perceptron receives input signals, processes them through a simple linear function, and produces an output based on an activation function, indicated as $g()$ in *Figure 3*. While individual perceptrons are limited in their representational capacity, they can be combined in layers to create more powerful ANNs.

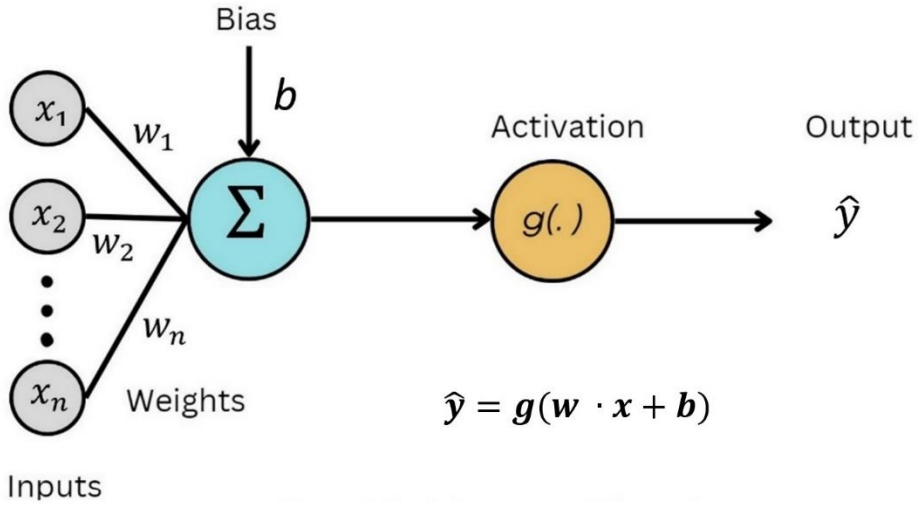


Figure 3: Schematic of a Basic Perceptron

A single neuron, equivalent to the illustration in *Figure 3,* will perform the following operation:

$$\sum_{i=0}^{n} w_i x_i + b = w_1 x_1 + w_2 x_2 + \ldots + w_n b_n + b = w \cdot x + b \quad [eq. 15]$$

This operation computes the linear combination of inputs and weights, along with the addition of a bias term. The connections between neurons are characterised by weights (*w*) and biases *(b)*, which are adjusted during the learning process *(see Section 3.2)* to minimise the error between predicted and actual outputs. The output of *Equation 15* is passed through an activation function to introduce nonlinearity and generate the final output of the neuron.

Weights and biases are crucial elements of ANNs as they determine the strength of the connections between neurons [6]. Weights represent the influence of one neuron on another, while biases allow for shifting the activation function *(see Figure 4)* along the input axis. During the learning process, weights and biases are adjusted iteratively to optimise the network's performance and minimise the error using a loss function.

The weighted sum of inputs and biases *[eq. 15]* form the input for the activation function. The activation function then processes this input and produces an output that is consequently used as the input for the neurons in the next layer. The introduction of nonlinearity by activation functions allows ANNs to overcome the limitations of linear models and represent nonlinear patterns in the data. The two most common types of activation functions are the sigmoid function and the hyperbolic tangent (tanh) function.

The sigmoid function squashes input values into the range [0, 1], making it well-suited for binary classification tasks or when the output should represent a probability. However, the sigmoid function can suffer from the vanishing gradient problem [7], leading to poor convergence, which hampers learning in deep networks. Hyperbolic tangent function is similar to the sigmoid function but maps input values into the range [-1, 1], providing a balanced output around zero. While it also suffers from the vanishing gradient problem, it generally performs better than the sigmoid function in hidden layers of ANNs.

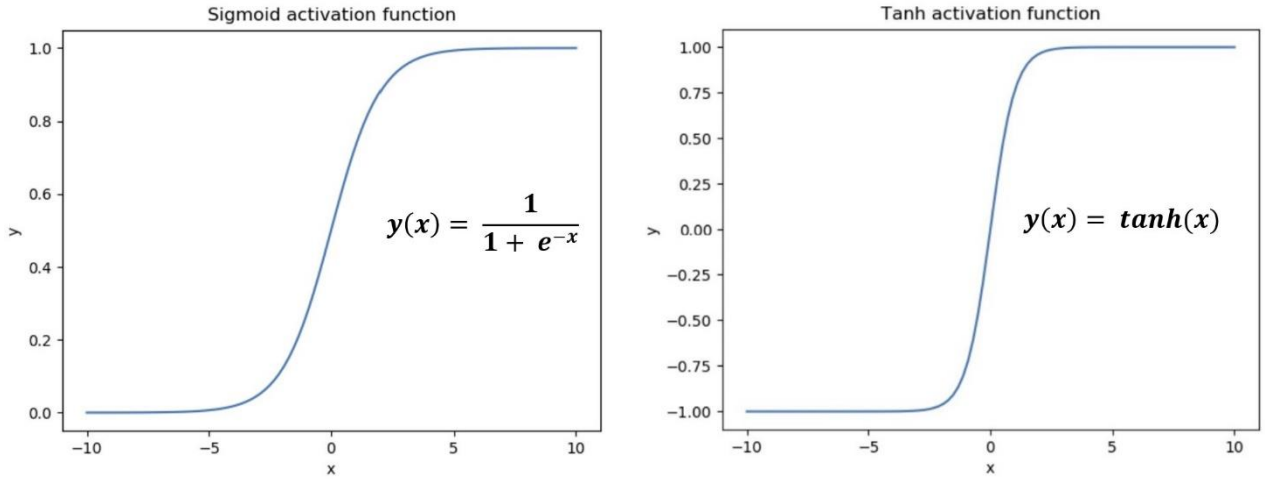


Figure 4: Sigmoid and Tanh Activation Functions

The goal of training a neural network is to find the optimal set of weights and biases that minimise the error between the predicted output and the actual target values, ultimately enabling the network to learn and represent complex patterns. Loss functions serve as the basis for this optimisation process. By evaluating the network's output against the true target values, the loss function provides a measure of how well the network is performing. During the training process *(see Section 3.2)*, gradient-based optimisation algorithms, such as gradient descent and its variants, are used to iteratively adjust the weights and biases of the network to minimise the value of the loss function. This iterative process, known as backpropagation [8], subsequently updates the values of weights and biases, allowing the neural network to learn.

In the context of transient heat transfer, ANNs offer promising solutions by leveraging their ability to model complex physical phenomena with high accuracy and efficiency. By training on historical data and incorporating physical principles and constraints during the learning process, ANNs can enable efficient modelling of transient heat transfer behaviour in various engineering applications.

2. LITERATURE REVIEW

The objective of this literature review is to provide a concise overview of the development and application of ANNs in solving ordinary and partial differential equations, with a particular focus on the use of physics-informed neural networks *(introduced in Section 3)* for modelling transient heat transfer in film cooling applications.

The history of ANNs dates back to the 1940s, when McCulloch and Pitts first proposed a mathematical model of an artificial neuron [9]. Since then, ANNs have undergone significant advancements, and as this area of research began to gain more attention in late 1980s, successful development of backpropagation algorithms gave birth to multi-layer perceptrons, making it possible to train neural network models with multiple hidden layers and improved output accuracy. In 1990s scientists and researchers began to explore the application of this new methodology to practical problems, particularly exploring the avenue of solving differential equations.

This led to the pioneering work of Lagaris, Likas, and Fotiadis in their 1998 paper, "Artificial neural networks for solving ordinary and partial differential equations," [10]. Lagaris et al. (1998) presented a novel approach to solving boundary value problems involving ordinary and partial differential equations (ODEs and PDEs). The authors showed that ANNs could be effectively trained to satisfy differential equations and boundary conditions by minimising a loss function that incorporated both components. The introduced concept of training a feedforward ANN architecture using backpropagation forms the foundation for the structure of the chosen method *(introduced in Section 3)* to solve the film cooling problem specified in *Section 1.2*. Despite developing a flexible and adaptable framework to solve a wide range of ODE and PDE problems, the authors did not consider employing ANN architectures that consist of more than one hidden layer. The ANN developed in *Section 3* elaborates this concept further, by introducing an architecture with numerous hidden layers.

Building upon this pioneering work, Raissi and Perdikaris introduced a further advancement in their 2018 paper, "Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations" [11]. The authors proposed a novel method called physics-informed neural networks (PINNs) that incorporates the governing equations of the physical system directly into the neural network's loss function. This approach ensures that the learned solutions not only fit the available data but also satisfy the underlying physical laws. Similar to the 1998 paper, limitations of the research uncovered challenges related to computational cost, particularly for high-dimensional or complex problems. Due to the one-dimensional nature of the film cooling problem this project is aiming to tackle, the computational cost consideration is not an immediate issue, but is worth considering when exploring future work and advancements. In summary, the PINN framework presented by Raissi et al. forms the foundation for the network architecture and optimisation introduced in *Section 3*.

My investigation on applying PINNs for solving transient heat transfer problems in film cooling applications builds upon the foundational work of Lagaris et al. (1998) and the advancements made by Raissi and Perdikaris (2018). While the 1998 paper pioneered the use of artificial neural networks (ANNs) for solving differential equations, the 2018 paper introduced PINNs as a deep learning framework that accounts for physical governing laws. My project contributes to the existing body of knowledge by specifically focusing on the phenomena involved in film cooling, which has significant implications for enhancing the efficiency and lifespan of gas turbines and other high-temperature applications.

3. METHODOLOGY

3.1 Physics-Informed Neural Network

In traditional finite differencing approaches, physics-based models are used to describe the governing equations of a system. However, these models can be limited by simplifying assumptions, uncertainty in parameters, and errors in initial/boundary conditions. This section will elaborate on the concept of Physics-Informed Neural Networks (PINNs) and demonstrate how the framework introduced by Raissi and Perdikaris (2018) can be expanded and applied to transient heat transfer in film cooling contexts [11].

Physics-Informed Neural Networks (PINNs) are trained to solve supervised learning tasks whilst conforming to the physical governing laws described by nonlinear partial differential equations (PDEs). This is achieved by adding a loss term to the training function that penalises deviations from the governing equations. Often, low data availability of engineering systems, such as the film cooling case makes it challenging to train traditional ANNs effectively. PINNs can leverage the underlying physical laws to guide the training process, allowing them to make accurate predictions with less empirical data. Therefore, PINN approach is highly applicable to film cooling scenarios, including the case described by the governing equations in *Section 1.2.*

The PINN developed to solve the thermal evolution across the turbine blade thickness consists of a single input layer of 2 nodes (time and spatial dimension input), 4 hidden layers with 20 nodes each, and the final single output layer, producing the output temperature as a function of time and space.

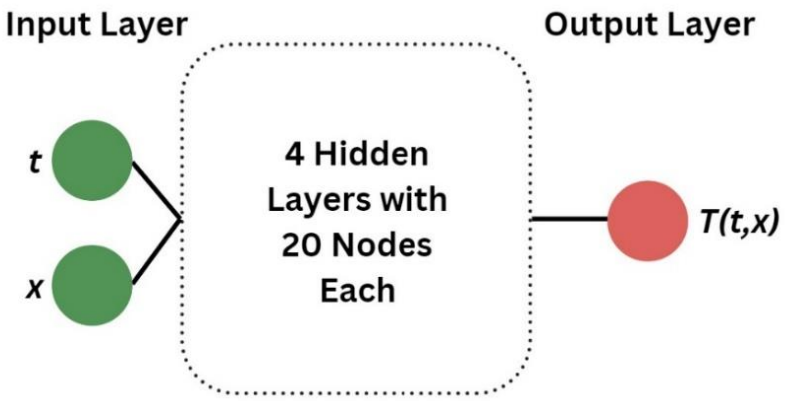


Figure 5: Simplified PINN Architecture Schematic

In python, the multi-layer neural network is defined using PyTorch's *nn.Module* class. The forward method specifies how data flows through the network. In this case, the input data is first passed through the input layer and then through the hidden layer four times, with each pass applying a hyperbolic tangent activation function to the output. The solution of the final hidden layer is then passed through the output layer to produce the final *T(t, x)* value *(Figure 6)*.

```python
class Net(nn.Module):                    #Set up neural net (fully connected - 4 hidden layers)
    def __init__(self):
      super(Net, self).__init__()
      self.input = nn.Linear(2, 20)    #One input layer with two nodes, followed by four hidden layers with 20 nodes each.
      self.hidden = nn.Linear(20, 20)
      self.output = nn.Linear(20, 1)   #The output layer has one node, which represents the predicted temperature T(t,x)

    # x represents our data
    def forward(self, x):
        # Pass data through conv1
        x = torch.sigmoid(self.input(x))
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.hidden(x))
        output = self.output(x)

        return output
```

Figure 6: Python Implementation of the PINN Structure

3.2 Loss Function & Network Optimisation

The loss function portion of the script generates two 50-element tensors for time and spatial dimension. These tensors are then further processed via matrix manipulations *(see Appendix),* to form two [2500x2] tensors *inputsb0* and *inputsb1,* corresponding to the boundary conditions at *x=0* and *x=L* respectively, as seen below:

$$\text{inputsb0} = \begin{matrix} tt & xboundary0 \\ \begin{bmatrix} 0 & 0 \\ \downarrow & \downarrow \\ . & 0 \\ . & 0 \\ . & 0 \\ 5 & 0 \end{bmatrix} \end{matrix} \quad \begin{matrix} (2500 \times 2) \\ Boundary\ at\ x{=}0 \end{matrix}$$

$$\text{inputsb1} = \begin{matrix} tt & xboundary1 \\ \begin{bmatrix} 0 & L \\ \downarrow & \downarrow \\ . & L \\ . & L \\ . & L \\ 5 & L \end{bmatrix} \end{matrix} \quad \begin{matrix} (2500 \times 2) \\ Boundary\ at\ x{=}L \end{matrix}$$

Similarly, two more [2500x2] tensors *inputs0* and *inputs* are computed to account for the initial condition, and the temperature distribution, to be fed into the neural net function defined in *Figure 6.* The neural net is evaluated with the aforementioned tensors as inputs *(See Figure 7)*, producing a temperature distribution solution.

$$\text{inputs} = \begin{matrix} tt & xx \\ \begin{bmatrix} 0 & 0 \\ \downarrow & \downarrow \\ . & . \\ . & . \\ . & . \\ 5 & L \end{bmatrix} \end{matrix} \quad \begin{matrix} (2500 \times 2) \\ Temperature \\ Distribution \end{matrix}$$

$$\text{inputs0} = \begin{matrix} tt & x0 \\ \begin{bmatrix} 0 & 0 \\ \downarrow & \downarrow \\ 0 & . \\ 0 & . \\ 0 & . \\ 0 & L \end{bmatrix} \end{matrix} \quad \begin{matrix} (2500 \times 2) \\ Initial\ Condition \\ at\ t{=}0 \end{matrix}$$

```
#Evaluate network on these inputs
u  = net.forward(inputs)            #inputs corresponds to the temperature solution tensor
u0 = net.forward(inputs0)           #inputs0 corresponds to the initial condition tensor
ub0 = net.forward(inputsb0)         #inputsb0 corresponds to the inner boundary condition tensor
ub1 = net.forward(inputsb1)         #inputsb1 corresponds to the outer boundary condition tensor
```

Figure 7: Neural Network Evaluated in Python

As mentioned in *Section 1.3, page 10*: "The goal of training a neural network is to minimise the error between the predicted output and the actual target values". The PINN computes the error between the solution obtained by *nn.Module* class *(Figure 6)*, and the differential operator terms that correspond to the governing partial differential equation, as seen in *Figure 8.* The full loss function is constructed as the sum of the four residual terms. Each of the four residual term corresponds to the difference between the predicted solution obtained from the neural network and the true solution of the partial differential equation, where:

- u is the unknown temperature solution
- u_xx is 2^{nd} order spatial derivative
- u_t is the 1^{st} order time derivative
- ub0 is the Neural Net output at the inner boundary domain
- ub1 is the Neural Net output at the outer boundary domain
- ut is the initial condition domain

```
#Calculate differential operator terms
u_t  = grad(u,tt,grad_outputs=torch.ones_like(u),retain_graph=True,create_graph=True)[0]   #first order time derivative
u_x  = grad(u,xx,grad_outputs=torch.ones_like(u),retain_graph=True,create_graph=True)[0]   #first order spatial derivative
u_xx = grad(u_x,xx,grad_outputs=torch.ones_like(u),retain_graph=True,create_graph=True)[0] #second order spatial derivative

#Non-Dimensionalising the heat transfer coefficients
hi = (hi_dim*L)/k                 #Biot number at internal boundary
he = (he_dim*L)/k                 #Biot number at external boundary

# Using computed terms, construct loss function and return this
res = (u_xx - u_t)**2             #Residual of the governing differential equation
bc0 = (hi*(u-ub0)-u_t)**2         #Boundary at x=0
bc1 = (-he*(u-ub1)-u_t)**2        #Boundary at x=1
ic  = (u0 - ut)**2                #Initial condition

# The loss function constructed as the sum of the means of the three defined vectors
#loss=torch.mean(res) + torch.mean(bc) + torch.mean(ic)
loss = torch.mean(res) + torch.mean(bc0) + torch.mean(bc1) + torch.mean(ic)
```

Figure 8: Construction of the Loss Function in Python

Once the loss function is computed, the network is trained with a built-in Adam optimiser class. The optimisation consists of two iteration loops:

- First loop trains the network for 2000 iterations at a learning rate of 0.002
- Second loop trains the network for an additional 2000 iterations at a finer learning rate of 0.0002

```
# Define the optimizer and learning rate
opt = torch.optim.Adam(net.parameters(), lr=0.002)

# Train the network for 2000 iterations using a learning rate of 0.002
for i in range(2000):
    opt.zero_grad()
    loss = loss_fn(net)
    loss.backward()
    opt.step()
    if i % 10 == 0:
        print('Iteration:', i, '. Loss:', loss)

    # Append the loss and iteration values to the respective lists
    stored_losses.append(loss.item())
    stored_iterations.append(i)

# Redefine the optimizer with a smaller learning rate (0.0002) and continue training
opt = torch.optim.Adam(net.parameters(), lr=0.0002)
```

Figure 9: Implementation of Adam Optimiser in Python

The network is trained to minimise the loss function value, resulting in a full convergence of the loss function over 4000 iterations, as demonstrated in *Figure 10*. The convergence of the loss function is significant, as it means optimisation algorithm has found a set of parameters (weights and biases) that minimise the loss function to the best of its ability.

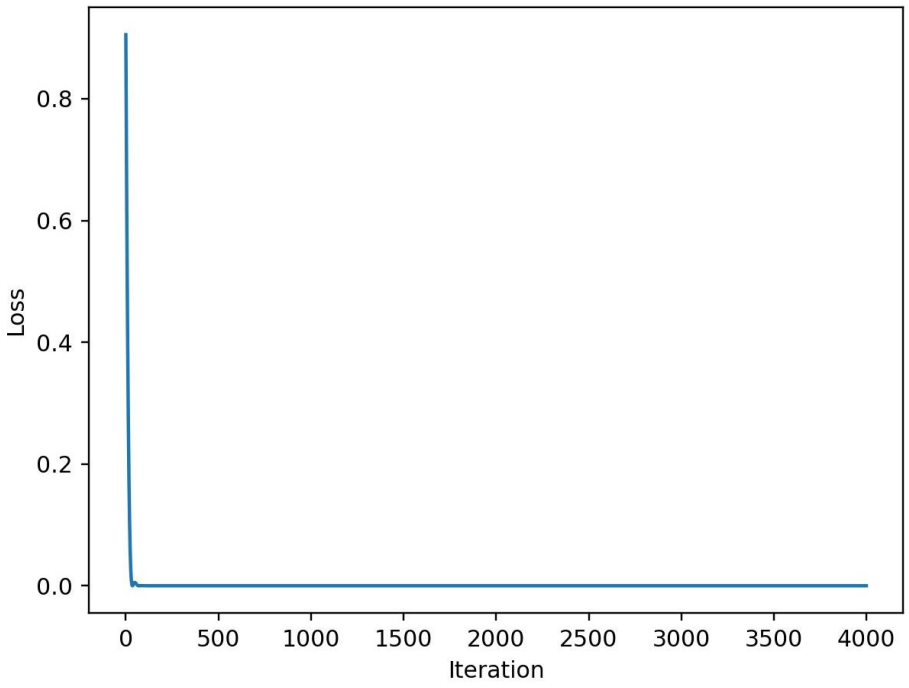


Figure 10: Loss Function Convergence Study

4. RESULTS & ANALYSIS

4.1 Neural Network Approximation

The temperature distribution output produced by the PINN described in *Section 2* is demonstrated in *Figure 11* & *Figure 12.*

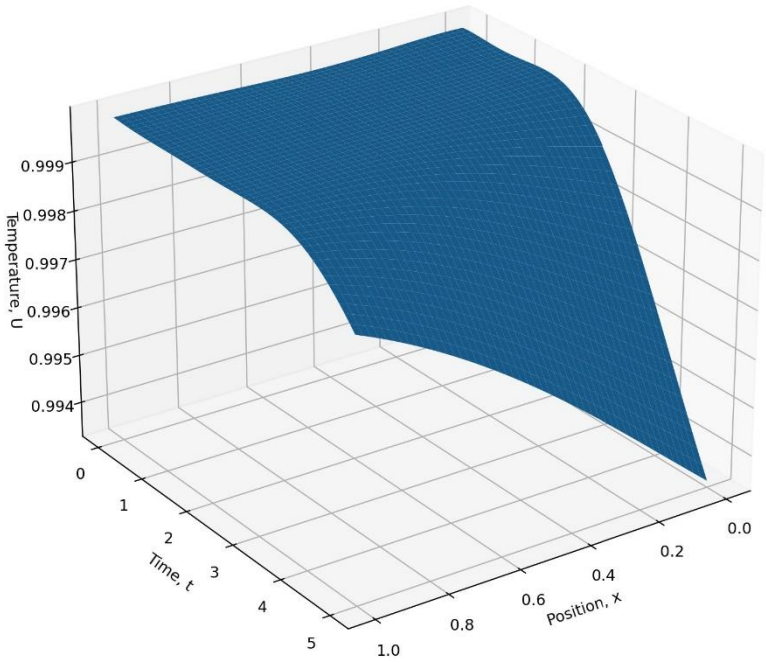


Figure 11: PINN Output - Temperature Distribution

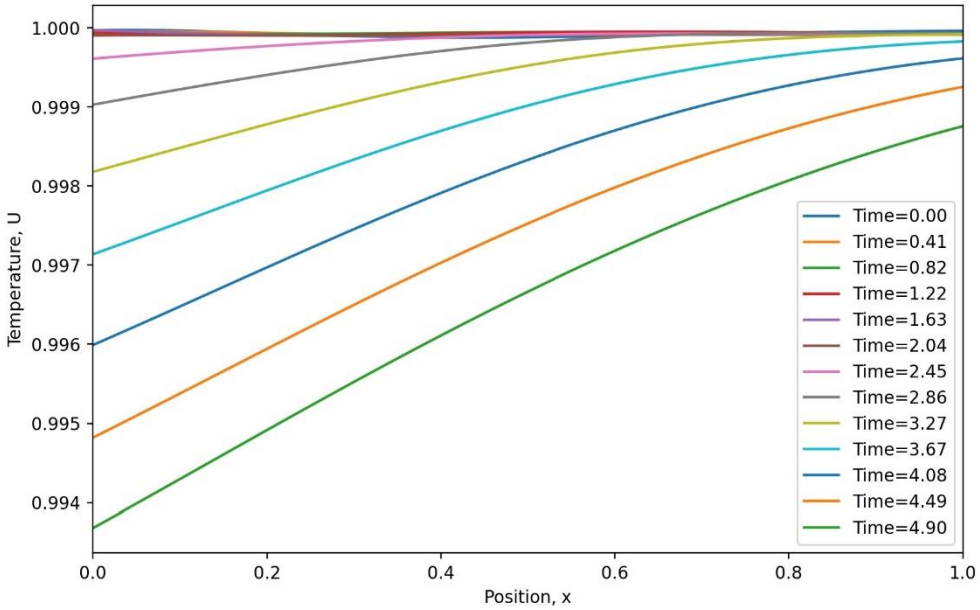


Figure 12: PINN Output - Temperature Distribution at different time locations

The temperature behaviour illustrated in the figures on page 16 follows the expected temperature evolution in the context of film cooling, with applied convective boundary conditions.

The initial conditions specify that at the start of the time domain, the temperature distribution across the thickness of the turbine blade is uniform and at a maximum of $\tilde{T}(0, x) = 1$. This is reflected by the surface plot, as it takes approximately 2.15 seconds for the effect of film cooling to have an impact, initially at the internal boundary. The surface at $x=0$ is in direct contact with the coolant flow. The steep temperature gradient between the coolant fluid and the inner blade surface $\tilde{T}(t, 0)$, provides a strong platform for convective heat transfer, resulting in a rapid drop in temperature, reaching the final temperature of $\tilde{T}(5, 0) \approx 0.993$.

Contrarily, at the external surface, the film cooling effect takes longer to have an impact, due to direct contact of the surface with high temperature combustion gases from the engine. As the film cooling process progresses, the coolant fluid forms a protective film of cool gas between the hot combustion gases and the outer blade surface, reducing heat transfer by conduction and enhancing convective heat transfer specified in [*Eq. 7*]. This results in a temperature drop observed at an approximate 3.1 seconds, and a final temperature of $\tilde{T}(5, 1) \approx 0.998$.

4.2 Verification & Analysis

Whilst the temperature distribution obtained by the PINN looks promising and satisfies the expected heat transfer behaviour under convective boundary conditions, it is paramount to verify the model with an established and reliable numerical modelling technique. Due to the one-dimensional characteristic of the heat transfer problem specified in *Section 1.2*, an implicit finite differencing method modelled in MATLAB software would provide a high degree of accuracy and stability. Crank-Nicolson scheme's unconditional stable nature and second order accuracy makes it the best numerical method to perform the task of verifying the PINN solution.

The discretisation process of the method begins with approximating the term $U_i^{n+1/2}$ using the average of the second centered differences for U_i^{n+1} and U_i^n, as follows:

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \frac{\alpha}{2}\left(\frac{U_{i-1}^{n+1} - 2U_i^{n+1} + U_{i+1}^{n+1}}{\Delta x^2} + \frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{\Delta x^2}\right) [eq. 16]$$

$$where\ p = \frac{\alpha \Delta t}{\Delta x^2}$$

Reducing the scheme to approximate the Heat Equation as:

$$-\frac{p}{2}U_{i-1}^{n+1} + (1+p)U_i^{n+1} - \frac{p}{2}U_{i+1}^{n+1} = \frac{p}{2}U_{i-1}^n + (1-p)U_i^n + \frac{p}{2}U_{i+1}^n\ [eq. 17]$$

Implementing into tridiagonal matrix:

$$a(2: nx - 1) = -\frac{p}{2}$$

$$b(2: nx - 1) = 1 + p$$

$$c(2{:}\,nx-1) = -\frac{p}{2}$$

$$d(2{:}\,nx-1) = \frac{p}{2}U_{i-1}^{n} + (1-p)U_{i}^{n} + \frac{p}{2}U_{i+1}^{n}$$

Due to the convective nature of the internal and external surface conditions, further manipulation is required to obtain tridiagonal matrix inputs at the two boundaries. Full derivation is accessible in the *Appendix*, and the final matrix structure is as follows:

Internal Boundary at $x = 0$

$$\begin{cases} a(1) = 0 \; or \; NaN \\ b(1) = 1 + p + \frac{h\Delta xp}{k} \\ c(1) = -p \\ d(1) = \left(1 - p - \frac{h\Delta xp}{k}\right)U_1^n + pU_2^n + \frac{2h\Delta xp}{k}T_c \end{cases}$$

External Boundary at $x = L$

$$\begin{cases} a(nx) = -p \\ b(nx) = 1 + p + \frac{h\Delta xp}{k} \\ c(nx) = \; 0 \; or \; NaN \\ d(nx) = pU_{nx-1}^n + \left(1 - p - \frac{h\Delta xp}{k}\right)U_{nx}^n + \frac{2h\Delta xp}{k}T_h \end{cases}$$

```
ivec = 2:nx-1; % set up index vector

%now loop through time
for n=1:nt-1
    % Refresh graph
    drawnow

    % wait one timestep to give roughly real time display
    pause(dt)

    %calculate internal values using Crank-Nicolson method
    b(1)    = 1 + p + (hc*dx*p)/k;
    c(1)    = -p;
    d(1)    = (1-p-(hc*dx*p)/k)*u(n,1) + p*u(n,2) + ((2*hc*dx*p)/k)*Tc;
    a(ivec) = -p/2;
    b(ivec) = 1 + p;
    c(ivec) = -p/2;
    d(ivec) = (p/2)*u(n,ivec-1) + (1-p)*u(n,ivec) + (p/2)*u(n,ivec+1);
    a(nx)   = -p;
    b(nx)   = 1 + p + (hh*dx*p)/k;
    d(nx)   = p*u(n,nx-1) + (1-p-(hh*dx*p)/k)*u(n,nx) + Th*(2*hh*dx*p)/k;

    u(n+1,:) = tdm(a,b,c,d);

    % update graph with new values
    set(h,'YData', u(n+1,:));
end
```

Figure 13: MATLAB Implementation of Crank-Nicolson TDM

The temperature distribution produced by a Crank-Nicolson scheme is demonstrated in *Figure 14.*

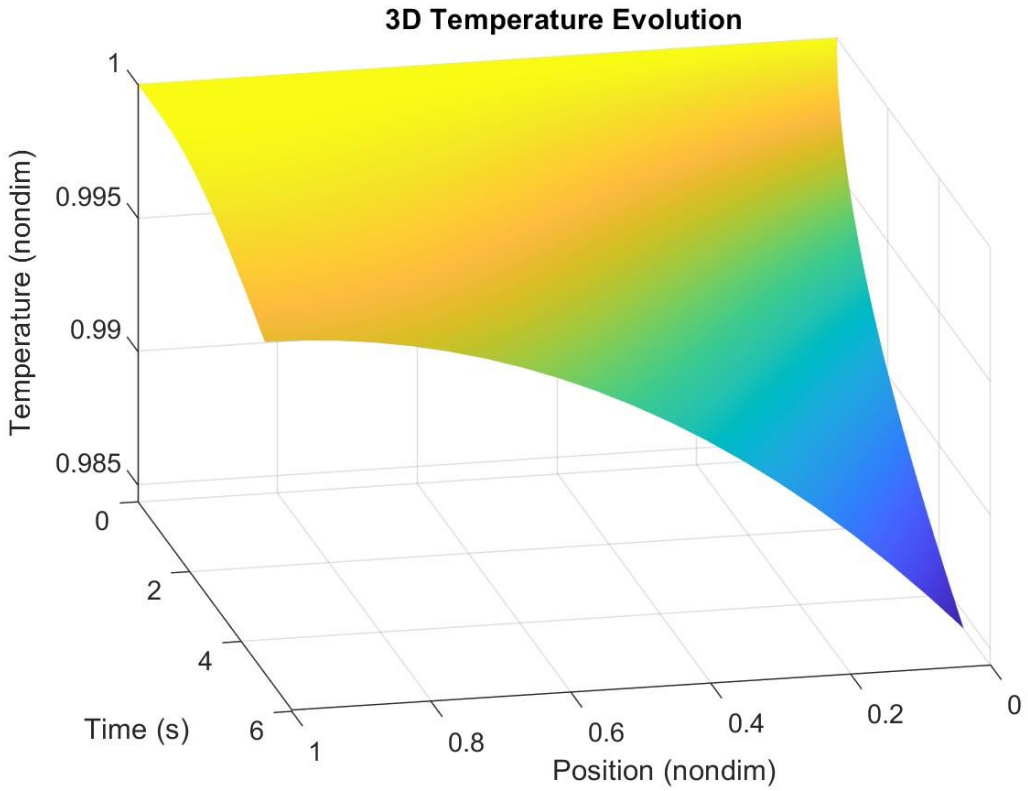


Figure 14: Temperature Distribution Output via Crank-Nicolson

The solutions from *Figure 11* and *Figure 14* are numerically compared using a Python script *(see Appendix).* Root Mean Square Error (RMSE) and Maximum Error % metrics are defined and computed as follows:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{T}_i - T_i)^2}{n}} = 0.003729$$

$$Maximum\ Error\ \% = \max\left(100 \times \left|\frac{\hat{T} - T}{\hat{T}}\right|\right) = 1.03\%$$

Where $\hat{T}$ and T denote the temperatures computed by the Crank-Nicolson scheme via MATLAB and the Neural Network respectively.

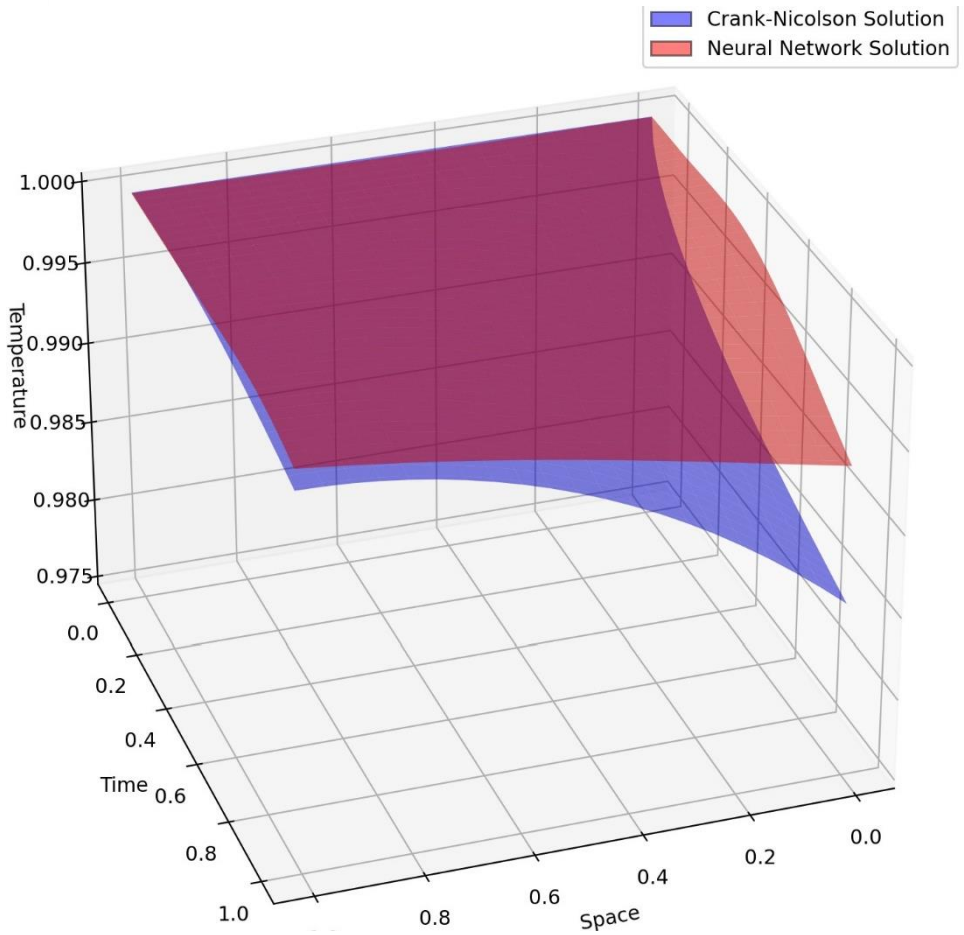


Figure 15: Surface Plot Comparison of the Modelling Methods

Figure 15 demonstrates that despite a relatively good performance based on *Root Mean Square Error (RMSE)* parameter, the solution provided by the PINN exhibits lower levels of accuracy at the internal boundary *(x=0)*. Whilst the neural network correctly models the inner surface in contact with the coolant flow experiencing a greater heat transfer rate per unit area, the magnitude of the heat flux is underestimated, which is reflected by the shallower gradient, in comparison to the Crank-Nicolson approximation. The model's inaccuracy 'at the internal boundary' is manifested by the *Maximum Error* value of 1.03%. In contrast, the neural network demonstrates a very high level of accuracy at the external boundary, with a local *Maximum Error* of 0.179% at the very end of the time domain. Neural network solution's deviation from the reference model at the inner boundary might be caused by various factors, such as neural network architecture and optimisation, which is explored in the *Discussion* section of the report. Overall, despite a relatively poor approximation at one of the boundaries, the PINN model successfully predicts the general heat transfer behaviour under convective boundary conditions in the film cooling context, and demonstrates a good performance based on the error metrics presented on *page 19*.

5. DISCUSSION

As evidenced in *Section 4.2,* the application of Physics-Informed Neural Networks (PINNs) to model transient heat transfer in film cooling contexts has demonstrated the potential of this deep learning framework for capturing the general behaviour of film cooling heat transfer phenomena. While the PINN solution aligns well with the traditional Crank-Nicolson MATLAB solver in most aspects, it exhibits inaccuracies at the internal boundary surface, which is in contact with the coolant flow and experiences a lower heat transfer rate compared to the benchmark MATLAB method. Despite these discrepancies, the model achieves a low RMS error and a maximum percentage error of only 1.03%, suggesting that the PINN framework is a promising tool for modelling complex heat transfer processes.

Potential limitations and sources of inaccuracy at the inner boundary might arise from the choice of network architecture, and hyperparameters, such as the numerous elements involved in the network training process. Insufficient or inaccurate data at the inner boundary, where the coolant flow interacts with the surface, could lead to a suboptimal representation of the heat transfer process in that region. Therefore, the nonlinearity and transient nature of the film cooling problem might require a more careful tuning of the PINN's optimisation algorithm and architecture, involving the exploration of an alternative learning rate, hidden layer structure and number of iterations in training, to capture the intricate physics at the boundary.

The outcome of this project, despite its limitations, builds upon the foundational work of Lagaris et al. (1998) and the further advancements made by Raissi and Perdikaris (2018). By applying the PINN framework to a specific and complex engineering problem, this research extends the generality and applicability of PINNs to real-world situations. Additionally, the insights gained from comparing PINN solutions to an established numerical method contributes to our understanding of the strengths, weaknesses, and challenges associated with using deep learning for solving complex physical systems.

6. CONCLUSION & FUTURE WORK

The primary objective of the research was to evaluate the performance of the PINN framework in comparison to a traditional Crank-Nicolson MATLAB solver and assess its potential for accurately capturing the complex behaviour of film cooling heat transfer.

The major findings of this investigation can be summarised as follows:

1. The PINN framework demonstrates its ability to model the general behaviour of film cooling heat transfer effectively, aligning well with the traditional Crank-Nicolson MATLAB solver in most aspects.
2. Some inaccuracies were observed at the internal boundary surface. Despite these discrepancies, the PINN model achieves a low RMS error and a maximum percentage error of 1.03%, indicating its potential for modelling complex heat transfer processes.
3. Potential sources of inaccuracy at the inner boundary include suboptimal network architecture, choice of hyperparameters, and inherent limitations of PINNs when applied to intricate boundary conditions.
4. The project builds on the foundational work, extending the generality and applicability of PINNs to real-world engineering problems and contributing to our understanding of the strengths, weaknesses, and challenges associated with using deep learning for modelling physical systems exhibiting transient heat transfer.

Considering the findings of this investigation and the limitations identified, future work may focus on:

1. Refining the network architecture and tuning the hyperparameters to optimise the model's performance and enhance its accuracy at the internal boundary surface.
2. Investigating the use of hybrid approaches that combine the strengths of traditional numerical solvers with PINNs to address the limitations of each method and improve the overall accuracy and efficiency of the solution.
3. Extending the PINN framework to model transient heat transfer in multiple spatial dimensions and applying the developed techniques to a wider range of engineering applications to further demonstrate the utility and versatility of the approach.

As Raissi et al. (2018) have highlighted in their paper, “We must note however that the proposed methods should not be viewed as replacements of classical numerical methods for solving partial differential equations. Such methods have matured over the last 50 years and, in many cases, meet the robustness and computational efficiency standards required in practice”. The conclusion of this project aligns with the perspective put forth by the authors, emphasising that, as the accelerating advancement of more robust and effective ANN algorithms persists, there is undoubtedly an opportunity for traditional numerical methods to coexist complimentarily with deep neural networks, harnessing the strengths of both approaches to advance the field of computational modelling.

REFERENCES

[1] Nastasi, G. S. (n.d). A deep learning approach to efficiently solve the Fourier equation.

[2] NASA Technology Transfer Program. (n.d.). Turbine Blade Film Cooling with Fluidic Oscillator. Retrieved from: https://technology.nasa.gov/patent/LEW-TOPS-52 [3] ResearchGate. (n.d.). Gas turbine blade cooling schematic [Figure]. Retrieved from: https://www.researchgate.net/figure/Gas-turbine-blade-cooling-schematic_fig1_315731610

[4] Thermopedia. (n.d.). Biot Number. Retrieved from: https://www.thermopedia.com/content/585/#:~:text=The%20numerical%20value%20of%20Biot,by%20convection%20at%20its%20surface.

[5] G. Ramakrishnan, "A concise history of neural networks," Towards Data Science, Mar. 2018. Retrieved from: https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec.

[6] Panagiotis Antoniadis. Baeldung. (n.d.). Sigmoid vs. tanh functions. Retrieved from: https://www.baeldung.com/cs/sigmoid-vs-tanh-functions#:~:text=and%20presents%20a%20similar%20behavior,instead%20of%201%20and%200.

[7] Wikipedia. (2021). Vanishing gradient problem. In Wikipedia. Retrieved from: https://en.wikipedia.org/wiki/Vanishing_gradient_problem

[8] Towards Data Science. (2020). Understanding backpropagation algorithm. Retrieved from: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

[9] McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5(4), 115-133.

[10] Lagaris, I. E., Likas, A., & Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. IEEE Transactions on Neural Networks, 9(5), 987-1000.

[11] Raissi, M., & Perdikaris, P. (2018). Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations.

APPENDIX

[A] Loss Function portion of the Python PINN architecture:

```
def loss_fn(net):

  # Creates two tensors (t,x) that are each 1D tensors with 50 elements.
  # The elements of t and x are evenly spaced values between tlim (0 to 5)
and xlim(-5 to 5) respectively.
  # The requires_grad flag is set to True, so gradients will need to be
computed with respect to these tensors during backpropagation.
  t = torch.linspace(tlim[0],tlim[1],50,requires_grad=True)      #size 1x50
  x = torch.linspace(xlim[0],xlim[1],50,requires_grad=True)      #size 1x50
  xboundary0 = torch.zeros((2500, 1))                            #2500x1
tensor of zeroes (x=0)
  xboundary1 = torch.ones((2500, 1))                             #2500x1
tensor of twos (x=1)

  # This creates a grid of points (t[i], x[j]) for all pairs of i and j
between 0 and 49, inclusive.
  # The resulting tensors tt and xx are each 2D tensors with 50 rows and 50
columns, representing the t and x values at each point on the grid.
  # The reshape function then flatten these tensors to have 2500 rows and 1
column.
  tt,xx = torch.meshgrid(t,x)     #both tensors have size 50x50
  tt = tt.reshape([-1,1])         #size 2500x1
  xx = xx.reshape([-1,1])         #size 2500x1

  # This reshapes the t and x tensors to be 2D tensors with 50 rows and 1
column, which is useful for later computations.
  t = t.reshape([-1,1]) #size 50x1
  x = x.reshape([-1,1]) #size 50x1

  # This creates two tensors t0 and x0 that are each 2D tensors with 50
rows and 1 column.
  # The elements of t0 are all equal to tlim[0], while the elements of x0
are the same as those in the x tensor.
  t0 = tlim[0]*torch.ones_like(x)       #size 50x1
  x0 = x                                #size 50x1

  # Vectors for boundary space and time point
  # This creates two tensors xb1 and xb2, each with 50 elements, and sets
them to xlim[0] and xlim[1], respectively.
  # The requires_grad=True flag is set so that gradients can be computed
with respect to these tensors during backpropagation.
  # The torch.cat function is used to concatenate xb1 and xb2 tensors
vertically into a single 100-element tensor xb.
  # Similarly, t and t tensors are concatenated vertically into a single
100-element tensor tb.
  xb1 = xlim[0]*torch.ones_like(t,requires_grad=True)      #size 50x1
  xb2 = xlim[1]*torch.ones_like(t,requires_grad=True)      #size 50x1
  #xb = torch.cat([xb1,xb2],0)                              #size 100x1
  #tb = torch.cat([t,t],0)                                  #size 100x1


  # Aggregate these points into Nx2 input tensor of (time,space) tuples
```

```
  inputs  = torch.cat([tt,xx],1)                                  #size 2500x2
  inputs0 = torch.cat([t0,x0],1)                                  #size 50x2
  inputsb0 = torch.cat([tt,xboundary0],1)                         #size 2500x2
  inputsb1 = torch.cat([tt,xboundary1],1)                         #size 2500x2


  #Evaluate network on these inputs
  u  = net.forward(inputs)             #inputs corresponds to the
temperature solution tensor
  u0 = net.forward(inputs0)            #inputs0 corresponds to the initial
condition tensor
  ub0 = net.forward(inputsb0)          #inputsb0 corresponds to the inner
boundary condition tensor
  ub1 = net.forward(inputsb1)          #inputsb1 corresponds to the outer
boundary condition tensor

  # Initial condition function (Uniform Temperature)
  ut = 1

  #Calculate differential operator terms
  u_t  =
grad(u,tt,grad_outputs=torch.ones_like(u),retain_graph=True,create_graph=Tr
ue)[0]   #first order time derivative
  u_x  =
grad(u,xx,grad_outputs=torch.ones_like(u),retain_graph=True,create_graph=Tr
ue)[0]   #first order spatial derivative
  u_xx =
grad(u_x,xx,grad_outputs=torch.ones_like(u),retain_graph=True,create_graph=
True)[0] #second order spatial derivative

  #Non-Dimensionalising the heat transfer coefficients
  hi = (hi_dim*L)/k                  #Biot number at internal boundary
  he = (he_dim*L)/k                  #Biot number at external boundary

  # Using computed terms, construct loss function and return this
  res = (u_xx - u_t)**2             #Residual of the governing differential
equation
  bc0 = (hi*(u-ub0)-u_t)**2          #Boundary at x=0
  bc1 = (-he*(u-ub1)-u_t)**2         #Boundary at x=1
  ic  = (u0 - ut)**2                #Initial condition

  # The loss function constructed as the sum of the means of the three
defined vectors
  #loss=torch.mean(res) + torch.mean(bc) + torch.mean(ic)
  loss = torch.mean(res) + torch.mean(bc0) + torch.mean(bc1) +
torch.mean(ic)

  return loss
```

[B] Tri-Diagonal Matrix Inputs Derivation

At the internal (x=0) boundary:

$$k\frac{U_2 - U_0}{2\Delta x} = h(U_1 - T_c)$$

$$-\frac{p}{2}U_0^{n+1} + (1+p)U_1^{n+1} - \frac{p}{2}U_2^{n+1} = \frac{p}{2}U_0^n + (1-p)U_1^n + \frac{p}{2}U_2^n$$

$$U = U_2 - \frac{2h\Delta x}{k}U_1 + \frac{2h\Delta x}{k}T_c$$

$$\left(1 + p + \frac{h\Delta xp}{k}\right)U_1^{n+1} - pU_2^{n+1} = \left(1 - p - \frac{h\Delta xp}{k}\right)U_1^n + pU_2^n + \frac{2h\Delta xp}{k}T_c$$

$$\begin{cases} a(1) = 0 \; or \; NaN \\ b(1) = 1 + p + \dfrac{h\Delta xp}{k} \\ c(1) = -p \\ d(1) = \left(1 - p - \dfrac{h\Delta xp}{k}\right)U_1^n + pU_2^n + \dfrac{2h\Delta xp}{k}T_c \end{cases}$$

At the external (x=L) boundary:

$$k\frac{U_{nx+1} - U_{nx-1}}{2\Delta x} = h(T_h - U_{nx})$$

$$-\frac{p}{2}U_{nx-1}^{n+1} + (1+p)U_{nx}^{n+1} - \frac{p}{2}U_{nx+1}^{n+1} = \frac{p}{2}U_{nx-1}^n + (1-p)U_{nx}^n + \frac{p}{2}U_{nx+1}^n$$

$$T_{nx+1} = U_{nx-1} - \frac{2h\Delta x}{k}U_{nx} + \frac{2h\Delta x}{k}T_h$$

$$-pU_{nx-1}^{n+1} + \left(1 + p + \frac{h\Delta xp}{k}\right)U_{nx}^{n+1} = pU_{nx-1}^n + \left(1 - p - \frac{h\Delta xp}{k}\right)U_{nx}^n + \frac{2h\Delta xp}{k}T_h$$

$$\begin{cases} a(nx) = -p \\ b(nx) = 1 + p + \dfrac{h\Delta xp}{k} \\ c(nx) = \; 0 \; or \; NaN \\ d(nx) = pU_{nx-1}^n + \left(1 - p - \dfrac{h\Delta xp}{k}\right)U_{nx}^n + \dfrac{2h\Delta xp}{k}T_h \end{cases}$$

[C] Python script for error metrics calculation

```
# Load the data from the CSV files
validationdata = np.genfromtxt('validationdata.csv', delimiter=',')
NN_temperature_evolution_fc =
np.genfromtxt('NN_temperature_evolution_fc.csv', delimiter=',')

# Align the files
x = np.linspace(0, 1, NN_temperature_evolution_fc.shape[1])
y = np.linspace(0, 1, NN_temperature_evolution_fc.shape[0])
f = RectBivariateSpline(y, x, NN_temperature_evolution_fc, kx=1, ky=1)
x_new = np.linspace(0, 1, validationdata.shape[1])
y_new = np.linspace(0, 1, validationdata.shape[0])
NN_temperature_evolution_fc_resized = f(y_new, x_new)

# Calculate the mean squared error between the two matrices
mse = np.mean((validationdata - NN_temperature_evolution_fc)**2)

# Calculate the root mean squared error between the two matrices
rmse = np.sqrt(mse)

# Calculate the maximum absolute error
max_abs_error = np.max(np.abs(validationdata -
NN_temperature_evolution_fc))

# Calculate the maximum error percentage
max_error_percentage = (max_abs_error / np.max(validationdata)) * 100

# Print the MSE, RMSE, and maximum error percentage to the console
print('Mean squared error: {:.6f}'.format(mse))
print('Root mean squared error: {:.6f}'.format(rmse))
print('Maximum error percentage: {:.2f}%'.format(max_error_percentage))
```