Application of Physics Informed Neural Networks for solving Piston Vibration Problem

Aditya Garg
Application No: MATS240
Indian Statistical Institute, Bangalore

On work carried out at the Indian Institute of Science, Bangalore under the guidance of

Prof. Dineshkumar Harursampath
NMCAD Lab, Department of Aerospace Engineering
Indian Institute of Science, Bangalore.

Indian Academy of Sciences, Bengaluru
Indian National Science Academy, New Delhi
The National Academy of Sciences India, Prayagraj

SUMMER RESEARCH FELLOWSHIPS — 2024

Format for the final Report *,^

Name of the candidate : Aditya Garg

Application Registration no. : MATS240

Date of joining : 03-05-2024

Date of completion : 31/7/2024

Total no. of days worked : 87

Name of the guide : Dr. Dineshkumar Harursampath

Guide's institution : Indian Institute of Science,Bengaluru

Project title : Application of Physics Informed Neural Networks for solving Piston Vibration Problem

Address with pin code to which the certificate could be sent:

B-10,B-type Quarters, Indian Statistical Institute, Bangalore

8th Mile, Mysore Road,RVCE Post, Bengaluru, 560059

E-mail ID: adigtyag@gmail.com

Phone No: 7879544057

TA Form attached with final report : YES _______ NO ☐ _____

If, NO, Please specify reason : Project was carried out at an institute close my own institution

Garg

Signature of the candidate

Date: 31/07/24

Signature of the guide

Date: ____________

IMPORTANT NOTES:

*** This format should be the first page of the report and should be stapled with the main report. The final report could be anywhere between 20 and 25 pages including tables, figures etc.**

^ The final report must reach the Academy office within 10 days of completion. If delayed fellowship amount will not be disbursed.

(For office use only; do not fill/tear)

Candidate's name:	Fellowship amount:
Student: Teacher:	Deduction:
Guide's name:	TA fare:
KVPY Fellow: INSPIRE Fellow:	Amount to be paid:
PFMS Unique Code:	A/c holder's name:
Others	

Acknowledgement

I would like to express my deepest gratitude to my guide, Prof. Dineshkumar Harursampath, and my mentor, Dr. Rajnish Mallick, for their invaluable guidance, continuous support, and encouragement throughout my research. Their profound knowledge and expertise have been instrumental in shaping this work, and their unwavering belief in my abilities has been a constant source of motivation.

I am also grateful for their insightful suggestions, constructive feedback, and always being available to discuss my progress. Their collaborative spirit and dedication have significantly contributed to the success of this project.

I would also like to extend my heartfelt thanks to my team members, Mr. Krishna Kant Mishra, an MTech student at IISc, Bangalore, and Mr.Sri Sai Charan S, a BS-MS student at IISER, Tirupati, for their cooperation, and camaraderie. Working with such a talented and supportive team has made this journey enjoyable and rewarding. Their contributions and enthusiasm have been vital to the completion of this research.

I am deeply grateful to the Indian Academy of Sciences and the Indian Institute of Science for providing excellent facilities and resources that have made this research possible. The opportunity to work in such a conducive environment has greatly enhanced the quality of this work.

Name: Aditya Garg
Date: 30 /07/ 2024
Place: Indian Institute of Science, Bangalore and at Tata Institute of Fundamental Research

Bayesian PINNs-based computation mechanics framework for the Piston Vibration Problem in Python

1 Aim of the project

- To understand different neural network architectures and adaptations suitable for application to differential equations.
- Applying these adaptations on the Piston Vibration Problem

2 Introduction

Neural networks are a class of deep learning methods that take inspiration from biological learning methods. Neural networks are well suited for prediction, pattern recognition, and various other applications. Neural networks might not be the best approach for solving differential equations as they do not account for initial and/or boundary conditions associated with these problems.
Physics Informed Neural Networks (PINNs) were introduced in 2019 with the aim of equipping neural networks with essential information so that their application for predicting solutions of differential equations is possible. Considering the need for uncertainty quantification in most engineering problems, a Bayesian framework was adopted along with PINNs in a work by Karniadakis, Meng, and Yang [8], to provide uncertainty quantification along with a solution to differential equations. This method was called the Bayesian-Physics Informed Neural Networks. These neural networks are efficient in predicting solutions even when noise-free data is unavailable.
The following sections review the working of these neural network architectures. We then work on the application of these neural networks for solving the piston vibration problem and finally we compare the results and performance of each neural network architecture.

3 Mathematical Preliminaries

3.1 Bayes' Theorem [5]

The Bayes' theorem is one of the most widely used results of probability.It provides a method to calculate the probability of an event based on the prior knowledge of events.It can be stated as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Bayes' theorem is used in statistics to write down posterior models based on prior beliefs.

3.2 Gaussian Processes [5]

A Gaussian process is a collection of random variables whose any finite subset follows a multivariate Gaussian distribution. A gaussian process is characterized by a mean and a covariance function.The distribution of a Gaussian process is the joint distribution of all random variables in the collection.

4 Neural Networks [1]

Neural networks is a deep learning technique that takes inspiration from biological neural networks. Neural networks have the following characteristics:

- All processing is done at logical nodes called neurons
- Neurons are interconnected to each other via links. The input to a neuron is the weighted sum of inputs through all links
- Typically, Neurons are arranged in layered structure.
- each neuron has an associated activation function which is a function of the weighted sum of the input

Any intermediate processing neuron layer between input and output is called a hidden layer. A neural network is characterized by the pattern of connection between neurons, link weights and activation functions. A few examples of neural network architectures are the Convolutional Neural Netowrks(CNN) and Recurrent Neural Network(RNN). Activation functions are generally chosen so that their derivative is easily expressible in terms of the function. Such a choice helps in optimization methods.
Common activation Functions:

- Identity function
- Step functions
- Rectified Linear function
- Sigmoid
- Hyperbolic Tan(tanh)

Working of a neural network [1]

With notations consistent with the diagram given below of a neural network (The numbers on the links denote the weight of the links and and the function in neurons represents the activation function of that neuron), the process of neural network training is given as below:
Step 0: Initialize weights
Step 1: For each training pair(Feed Forward Step):
— Step 2 : Each input unit receives an input signal and, broadcasts this signal to all units in the next layer.
— Step 3: Each hidden unit sums its weighted input signals, and applies activation function

$$z\_in_j = v_{0j} + \sum_{i=1}^{n} x_i v_{ij} \qquad z_j = f(z\_in_j) \tag{1}$$

— Step 4: Output unit calculates output signal

$$y\_in_k = w_{0k} + \sum_{j=1}^{p} z_j w_{jk} \qquad y_k = f(y\_in_k) \tag{2}$$

Backpropagation of error:
Step 5:For each output unit calculate the error as

$$\delta_k = (t_k - y_k) f'(y\_in_k) \tag{3}$$

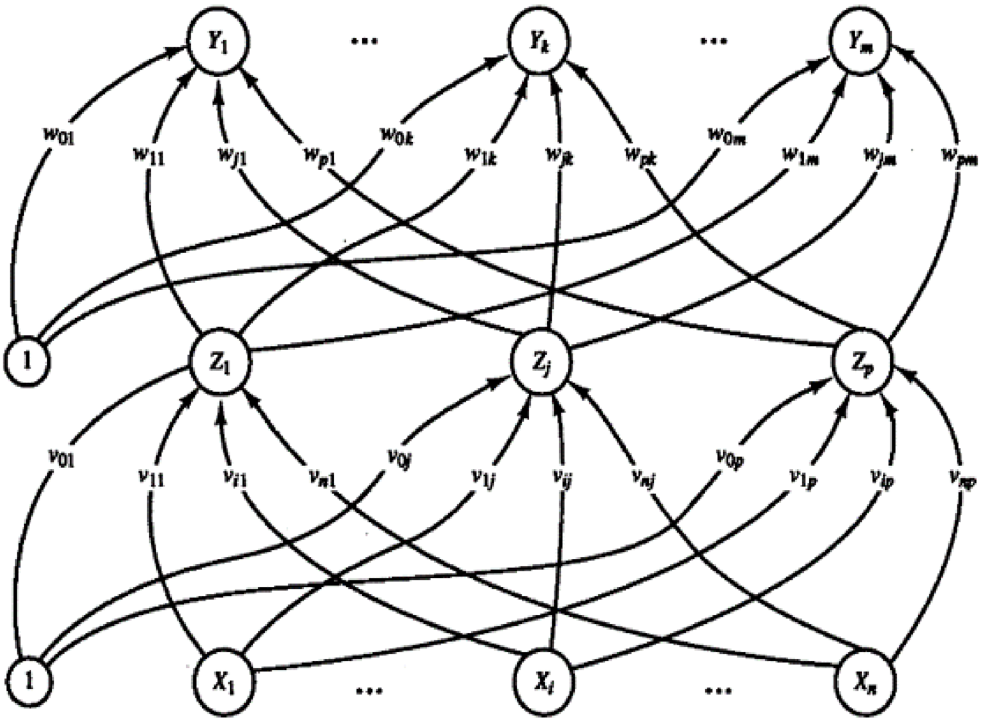


Figure 1: A depiction of 1 hidden layer neural network from [1]

Similarly, Calculate weight correction as $\Delta\omega_j k = \alpha\delta_k z_j$ and bias correction as $\Delta\omega_{0k} = \alpha\delta_k$
Step 6:For each hidden unit calculate its reverse input as :$\delta\_in_j = \sum_{k=1}^{m} \delta_k \omega_{jk}$
Hence the back propagated error is:$\delta_j = \delta\_in_j f'(z\_in_j)$,
Similarly, Calculate weight correction as $\Delta\nu_j k = \alpha\delta_j x_i$ and bias correction as $\Delta\nu_{0j} = \alpha\delta_j$
Step 7: Change the weights and repeat for all training points:

— $\omega_{jk}(new) = \omega_{jk}(old) + \Delta\omega_{jk}$ $\qquad$ $\nu_{ij}(new) = \nu_{ij}(old) + \Delta\nu_{ij}$

5 Physics Informed Neural Networks(PINNs) [4]

5.1 Need for a different approach

Neural networks are efficient in predicting various functions accurately. But when applied to the problem of predicting the solution of a physical problem, they are not enough. A neural network fits a function to given training data, however, it doesn't ensure whether the fitted function is a valid solution to the specified physical problem or not. A solution to a given physical problem should satisfy all the initial and boundary conditions. While neural networks don't take into account the information about boundary and initial conditions, Physics Informed Neural Networks use this information while at the same time attempting to fit a function to the labelled data.

5.2 Working of PINNs

Consider solving the differential equation

$$\mathcal{N}_{\mathbf{x}}(u) = f(\mathbf{x}), x \in \mathcal{D}_x \qquad \mathcal{B}_{\mathbf{x}}(u) = b, \mathbf{x} \in \Gamma \tag{4}$$

where $\mathcal{N}$ is a differential Operator acting on u;
u is the vector repressenting the state of the system
x is the input to the system;
f is the forcing term;
$\mathcal{B}_{\mathbf{x}}$ is boundary condition operator

The goal of PINNs is to build a model to approximate the state of the system,i.e., to approximate **x**. That is we have $g(t;\Theta) = \hat{u}(t;\Theta) \approx u(x;u;\mathcal{B}_x)$ where x_0 are the initial conditions the system is subjected to. A standard neural network solver with rms error as following aims to fit a function to the dataset $\mathcal{S}$ without caring whether the fit wil be a valid.

$$\underset{\Theta}{min} \; \frac{1}{N} \sum_{i=1}^{N} \sqrt{\hat{u}^{(i)}(\Theta) - (u^{(i)}_{true} + \eta^{(i)})^2} \tag{5}$$

The basic difference between a normal neural network and a PINN is in the error term. The PINNs account for the physical conditions by introducing an error term derived from the initial conditions. Thus, making the approximation by a PINNs a more accurate and a valid solution of the physical system, i.e, PINNs consider the error term (h represents error from the differential equation and h1 represents the initial conditions error) given by

$$h(u,x) = \mathcal{N}(\hat{u}) - f(\hat{u};x) \quad h1(u,x) = \mathcal{B}(\hat{u}) - b(\hat{u};x) \tag{6}$$

along with the rms error term and thus,ensuring that physical conditions are not violated. Thus, the total error to consider while training is

$$w_1 \frac{1}{N} \sum_{i=1}^{N} \sqrt{\hat{u}^{(i)}(\Theta) - (u^{(i)}_{true} + \eta^{(i)})^2} + w_2 h(u,x) + w_3 h(u,x) \tag{7}$$

where w_1, w_2 and w_3 are training weights.

6 Bayesian Neural Network[2]

Bayesian Neural networks are a probabilistic approach to neural networks where weights and biases are assumed from a stochastic process. A commonly used special case is where the assumed stochastic process is a Gaussian Process. The above can be stated as

$$\theta \sim p(\theta) \tag{8}$$

$$y = \Phi_\theta(x) + \epsilon \tag{9}$$

where θ is the vector of model parameters (weights and biases) and is from a random process p. y is the function we want to approximate. Φ represents the prediction of the neural network which depends on θ. ϵ represents the error in the fit. The likelihood can be calculated as

$$p(D|\theta) = P(\mathcal{D}_y|\mathcal{D}_x) \tag{10}$$

where $\mathcal{D}_y$ represents the response training data and $\mathcal{D}_x$ represents the input training data. The posterior then can be calculated from the Bayes' Theorem as

$$p(\theta|D) = \frac{P(\mathcal{D}_y|\mathcal{D}_x)p(\theta)}{\int_\theta P(\mathcal{D}_y|\mathcal{D}_x)p(\theta')d\theta'} \tag{11}$$

The resultant probability distribution after training with the data can then be used to calculate uncertainities releted to the model. For large datasets calculating the posterior as above is not always feasible. Thus, We make use of techniques like Markov Chain Monte Carlo and Variational Inference.

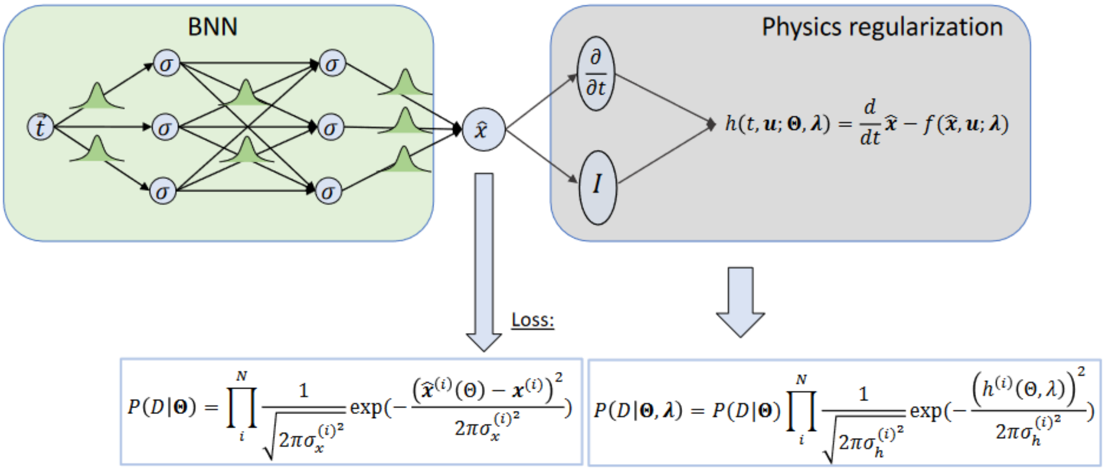


Figure 2: A Pictorial Depiction of working of Bayesian Physics Informed Neural Network from [6]

7 Bayesian Physics Informed Neural Networks(B-PINNs) [8]

7.1 Importance of Uncertainty Quantification

Although Physics Informed Neural Networks(PINNs) can estimate physical functions to good enough accuracies, it is always desirable to have information of the variability of the prediction. For example , In a problem of predicting the displacement of one end of a rod clamped at the other end, we always want to ensure that the rod doesn't break in the process, hence knowing information about the maximum variance in the predicted displacement can be used to ensure that the rod doesn't break. PINNs can be fused with another variation of neural networks called "Bayesian Neural Networks" to get a method known as "Bayesian-Physics Informed Neural Networks(B-PINNs)" which enables us to predict the uncertainty in our prediction.

7.2 Working of Bayesian PINNs [6]

Bayesian PINNs emerge as a fusion of PINNs and Bayesian neural networks.Again the physics error term is the difference between BNNs and B-PINNs. The calculation for likelihood changes for B-PINNs and is given by

$$p(D|\theta) = P(\mathcal{D}_y|\mathcal{D}_x) \prod_{j=1}^{2} \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma_{h_j^{(i)2}}}} exp(-\frac{(h_j^{(i)}(u,x))^2}{2\sigma_{h_j^{(i)2}}}) \tag{12}$$

and hence the posterior is now calculated as:

$$p(\theta|D) = \frac{P(\mathcal{D}_y|\mathcal{D}_x)p(\theta) \prod_{j=1}^{2} \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma_{h_j^{(i)2}}}} exp(-\frac{(h_j^{(i)}(u,x))^2}{2\sigma_{h_j^{(i)2}}})}{\int_\theta P(\mathcal{D}_y|\mathcal{D}_x)p(\theta')d\theta'} \tag{13}$$

8 Bayesian Optimization[9]

Bayesian Optimization is an optimization technique used to approximate the maximization(or minimization) of an unknown costly to evaluate function (Black Box Function) in a small number of evaluations. Bayesian Optimization assumes a surrogate model as a prior for the function and updates it after every evaluation. The most commonly used surrogate model is Gaussian Process. The next point of evaluation is chosen through an acquisition function. Acquisition Function is a function of the surrogate model whose maxima gives the next evaluation point.

8.1 Types of Acquisition Functions

The most popular acquisition functions are:

- Probability of Improvement
- Expectation of Improvement
- Upper Confidence Bound/Lower Confidence Bound

Probability of Improvement

Probability of Improvement acquisition function computes the probability of Improvement towards the maximization(or minimization) of our black box function at each point.Then the point with the maximum probability is chosen as the next point to evaluate.The formal expression for the probability of Improvement is given as:

$$PI(x) = \Phi(\frac{(\mu(x) - x^*)}{\sigma(x)}) \tag{14}$$

where Φ=Standard Normal CDF; μ=mean function of the surrogate model ; σ=Covariance function of the surrogate model; x^*=Maximum of surrogate model prior to evaluation

Expectation of Improvement

Expectation of Improvement(EI) maximizes the expected improvement after evaluation at that point. If at any point the expectation value of the function is less than the previous evaluation point(the previous maxima) we assign the expectation as zero. It is one of the most frequently used acquisition functions. The formal expression is given as:

$$EI(x) = ((x^* - \mu)\Phi(\frac{(x^* - \mu)}{\sigma(x)}) + \sigma(x)\phi(\frac{(x^* - \mu)}{\sigma(x)}))$$

(15)

where Φ=Standard Normal CDF; μ=mean function of the surrogate model; σ=Covariance function of the surrogate model; x^*=Maximum of surrogate model prior to evaluation

Upper Confidence Bound

The upper confidence bound acquisition function is given as:

$$UCB(x) = mu(x) + \lambda\sigma(x) \tag{16}$$

where μ=mean function of the surrogate model ; σ=Covariance function of the surrogate model ; λ=some prefixed constant ; The value of λ determines whether our acquisition function focuses on exploration or exploitation.Higher values of λ result in more exploration and lower values of λ result in more exploitation

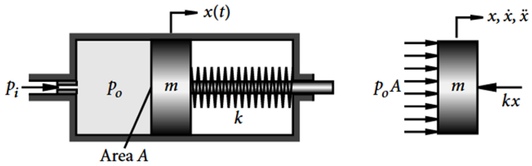


Figure 3: Depiction of the problem from [7]

9 Piston Vibration Problem [7]

Consider oil (incompressible in nature) entering a cylinder through a constriction such that the flow rate is given as $Q = \alpha(p_i - p_0)$ where p_i is the supply pressure and p_0 is the pressure in the cylinder. The cylinder contains a piston of mass m and area A backed by a spring of stiffness k. If $p_i$is of the form $p_i(t) = P_0 + P_1 sin\Omega t$ where P_1, Ω, P_0 are constants We wish to determine $x_p(t)$ which represents the displacement of the piston at time t. Using the Newton's laws and incompressibiliy of the oil yields the following equations

$$m\ddot{x} = p_0 A - kx \tag{17}$$

$$p_0 = p_i - \frac{A}{\alpha}\frac{dx}{dt} \tag{18}$$

The Governing differential equation can be then written using the conditions as:

$$\ddot{x} + \frac{A}{\alpha m}\dot{x} + \frac{k}{m}x = \frac{A}{m}p_i \tag{19}$$

Substituting p_i and rearranging,We get:

$$\ddot{x} + c\dot{x} + \omega_0^2 x = \frac{A}{m}(P_0 + P_1 sin\Omega t) \tag{20}$$

where $c = \frac{A^2}{\alpha m}$ and $\omega_0^2 = \frac{k}{m}$. A particular solution of the system can then be derived as(from [7]) :

$$x_p(t) = \frac{AP_0}{m\omega_0^2} + \frac{AP_1}{m}\frac{(\omega_0^2 - \Omega^2)sin(\Omega t) - c\Omega cos(\Omega t)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2} \tag{21}$$

Once we have a particular solution of the differential equation $\ddot{x} + c\dot{x} + \omega_0^2 x = \frac{A}{m}(P_0 + P_1 sin\Omega t)$.
we can find the general solution of the equation in the following form,

$$x_{gen} = x_1 + x_p \tag{22}$$

where x_1 is the general solution of the equation $\ddot{x} + c\dot{x} + \omega_0^2 x = 0$. Looking at the auxiliary equation $m^2 + cm + w_0^2 = 0$. We can from our experience write the general solution in the case where $c > 2\omega_0$ as $c_1 e^{s_1} + c_2 e^{s_2}$ where s_1 and s_2 are solutions of the auxiliary equation and the constants c_1 and c_2 are determined by the initial conditions. As initial conditions we will have the initial position of the piston x_0 and the initial velocity of the piston v_0. To calculate c_1 and c_2, we carry out the following calculation. We have

$$x_{gen}(t) = c_1 e^{s_1 t} + c_2 e^{s_2 t} + \frac{AP_0}{m\omega_0^2} + \frac{AP_1}{m}\frac{(\omega_0^2 - \Omega^2)sin(\Omega t) - c\Omega cos(\Omega t)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2} \tag{23}$$

9.1 Case 1: Over Damped Case

$$x_{gen}(0) = c_1 + c_2 + \frac{AP_0}{m\omega_0^2} - \frac{AP_1}{m}\frac{c\Omega}{(\omega_0^2 - \Omega_0^2)^2 + c^2\Omega^2} \qquad = x_0$$
$$\Rightarrow c_2 = x_0 + \frac{AP_1}{m}(\frac{c\Omega}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}) - \frac{AP_0}{m\omega_0^2} - c_1$$

Again, Using the other initial condition we can write

$$x'_{gen}(0) = v_0 \quad = c_1 s_1 + c_2 s_2 + \frac{AP_1}{m}\frac{\Omega(\omega_0^2 - \Omega^2)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}$$

Substituting c_2 we get

$$c_1 = \frac{1}{s_1 - s_2}[v_0 - s_2 x_0 + \frac{AP_1}{m}(\frac{\Omega(\Omega^2 - \omega_0^2) - c\Omega s_2}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}) + \frac{AP_0 s_2}{m\omega_0^2}] \tag{24}$$

Back substituting we get c_1 as

$$c_2 = x_0 + \frac{AP_1}{m}(\frac{c\Omega}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}) - \frac{AP_0}{m\omega_0^2} - [\frac{1}{s_1 - s_2}[v_0 - s_2 x_0 + \frac{AP_1}{m}(\frac{\Omega(\Omega^2 - \omega_0^2) - c\Omega s_2}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}) + \frac{AP_0 s_2}{m\omega_0^2}]] \tag{25}$$

Hence, we have the general solution in this case.

9.2 Case 2: Critically Damped Case

The auxiliary equation has equal roots in this case and hence the general solution is given as

$$x_{gen}(t) = c_1 e^{st} + c_2 t e^{st} + \frac{AP_0}{m\omega_0^2} + \frac{AP_1}{m}\frac{(\omega_0^2 - \Omega^2)sin(\Omega t) - c\Omega cos(\Omega t)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2} \tag{26}$$

where s is the root of the auxiliary equation, To determine c_1 and c_2, we have

$$x_{gen}(0) = c_1 + \frac{AP_0}{m\omega_0^2} - \frac{AP_1}{m}(\frac{c\Omega}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}) \quad = x_0$$
$$\Rightarrow c_1 = x_0 + \frac{AP_1}{m}(\frac{c\Omega}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}) - \frac{AP_1}{m\omega_0^2}$$
$$x'_{gen}(0) = c_1 s + c_2 + \frac{AP_1}{m}\frac{\Omega(\omega_0^2 - \Omega^2)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2} \qquad = v_0$$
$$\Rightarrow c_2 = v_0 - c_1 s + \frac{AP_1}{m}\frac{\Omega(\Omega^2 - \omega_0^2)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}$$

9.3 Case 3: Under Damped Case

Instead of approaching this case directly from the roots of auxilliary equation, we simplify calculations by assuming that the general solution in this case will be given as

$$x_{gen}(t) = e^{at}(c_1 cosbt + c_2 sinbt) + \frac{AP_0}{m\omega_0^2} + \frac{AP_1}{m}\frac{(\omega_0^2 - \Omega^2)sin(\Omega t) - c\Omega cos(\Omega t)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2} \tag{27}$$

where a+ib and a-ib are the solutions of our auxiliary equation.

$$\begin{aligned}
x_{gen}(0) = c_1 + \frac{AP_0}{m\omega_0^2} - \frac{AP_1}{m}\frac{c\Omega cos(\Omega t)}{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2} && = x_0 \\
\Rightarrow c_1 = x_0 + \frac{AP_0}{m}\left(\frac{sin\phi}{\sqrt{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}}\right) - \frac{AP_1}{m\omega_0^2} && \\
x'_{gen}(0) = c_1 a + c_2 b + \frac{AP_1\Omega}{m}\frac{cos\phi}{\sqrt{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}} && = v_0 \\
\Rightarrow c_2 = \frac{1}{b}\left[v_0 - c_1 a - \frac{AP_1\Omega}{m}\frac{cos\phi}{\sqrt{(\omega_0^2 - \Omega^2)^2 + c^2\Omega^2}}\right] &&
\end{aligned}$$

10 Code and Implementation

Implementation of PINNs was done in the DeepXde library of Python.

10.1 Implementation of PINNs

```
#importing necessary Libraries
import deepxde as xd
import numpy as np
import math
import tensorflow as tf
import matplotlib.pyplot as plt
#Declaration fo Physical Constants
#k=float(input("Spring constant="))
k=5
#A=float(input("Area="))
A=10
#m=float(input("mass"))
m=5
#alpha=float(input("alpha"))
alpha=10
#P0=float(input("P0="))
P0=1
#P1=float(input("P1="))
P1=1/2
#Omega=float(input("Omega"))
Omega=3
#y0=float(input("y(0)"))
y0=2
#ydash0=float(input("y'(0)"))
ydash0=1
#Calculation of other constants
c=(A**2)/(alpha*m)
omega2=k/m
denominator=((omega2-(Omega)**(2))**(2))+((c*Omega)**(2))
#Function to calculate a particular solution
def particular_solution(t):
```

```
    return ((A*P0)/(m*omega2))+(A*P1*(((omega2-(Omega)**2)*np
        .sin(Omega*t))-c*Omega*np.cos(Omega*t)))/(m*
        denominator)
e=math.exp(1)
#Function to calculate general solution from the particular
   solution
def solution(t):
    if c**2<4*omega2:
        a=-c/2
        b=math.sqrt(4*omega2-c**2)/2
        c1=y0+((A*P1*Omega*c)/(m*(denominator)))-((A*P0)/(m*
           omega2))
        c2=(1/b)*((ydash0)-(c1*a)-((A*P1*Omega*(omega2-Omega
           **2))/(m*(denominator))))
        return (e**(a*t)*c1*np.cos(b*t))+(e**(a*t)*c2*np.sin(
           b*t))+(particular_solution(t))
    elif c**2==4*omega2:
        s=-c/2
        c1=y0-((A*P0)/(m*omega2))+((A*P1*c*Omega)/(m*
           denominator))
        c2=ydash0-c1*s+(A*P1*(Omega**2-omega2))/(m*
           denominator)
        return (c1*e**(s*t))+(c2*t*e**(s*t))+
           particular_solution(t)
    else:
        s1=(-c-np.sqrt(c**2-4*omega2))/2
        s2=(-c+np.sqrt(c**2-4*omega2))/2
        c1=1/(s1-s2)*(ydash0+((A*P0*s2)/(m*omega2))-(s2*y0)
           +((A*P1*(Omega*(Omega**2-omega2)-c*Omega*s2))/(m*
           denominator)))
        c2=y0-c1+((A*P1*c*Omega)/(m*denominator))-((A*P0)/(m*
           omega2))
        return (c1*e**(s1*t))+(c2*e**(s2*t))+
           particular_solution(t)
#Definition of the ODE
def ode(x,y):
    y_der=xd.grad.jacobian(y,x,i=0)
    y_dder=xd.grad.hessian(y,x,i=0)
    return (y_dder) + (c*y_der) + (y*omega2) - ((A*P0/m)+((A*
        P1*tf.sin(Omega*x))/m))
#Domain where time would vary i.e. t=0 to t=25
geom=xd.geometry.TimeDomain(0,25)
#defining the boundary conditions
def boundary(x,on_initial):
    return xd.utils.isclose(x[0],0)and on_initial
ic1=xd.icbc.IC(geom,lambda x:y0,boundary)
def error(inputs,outputs,X):
    return xd.grad.jacobian(outputs,inputs,i=0)-ydash0
ic2=xd.icbc.OperatorBC(geom,error, boundary)
#generation of training and validation data
```

```
data=xd.data.TimePDE(geom,ode,[ic1,ic2],50,2,solution=
    solution,num_test=500)
#Hyper Parameters
layer=[1]+5*[50]+[1]
activation="sin"
initializer="Glorot normal"
net=xd.nn.FNN(layer,activation,initializer)
#Definition of the neural network model and training
model=xd.Model(data,net)
model.compile("adam",lr=0.0005,metrics=["l2 relative error"])
for i in range(0,100):
    losshistory,train_state=model.train(iterations=10)
    xd.utils.plot_best_state(train_state)
    plt.title('Equation:x"+'+str(c)+"x'"+"+"+str(omega2)+"x="
        +str(A*P0/m)+"+"+str(A*P1/m)+"Sin("+str(Omega)+"t)")
    plt.show()
```

10.2 Implementation of Bayesian PINNs

A modified version of the xPINNs library .[3]

BPINN_HMC.py

```
 #!/usr/bin/env python
# coding: utf-8

# #### Bayesian Physics-Informed Neural Network (with ODE)

# In[1]:

#importing necessary files
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"

import tensorflow as tf
import tensorflow_probability as tfp
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sio

import load_data
import bnn
import bayesian
import hmc
import time
from sklearn.metrics import mean_squared_error
wd = os.path.abspath(os.getcwd())
filename = os.path.basename(__file__)[:-3]

path2saveResults = 'Results/'+filename
```

```
if not os.path.exists(path2saveResults):
    os.makedirs(path2saveResults)
#Physical constants
#k=float(input("Spring constant="))
k=5
#A=float(input("Area="))
A=10
#m=float(input("mass"))
m=5
#alpha=float(input("alpha"))
alpha=10
#P0=float(input("P0="))
P0=1
#P1=float(input("P1="))
P1=1/2
#Omega=float(input("Omega"))
Omega=3
#y0=float(input("y(0)"))
y0=2
#ydash0=float(input("y'(0)"))
ydash0=1
c=(A**2)/(alpha*m)
omega2=k/m


# nmax = 2000 b
#%% Plotting loading
ColorS = [0.5, 0.00, 0.0]
ColorE = [0.8, 0.00, 0.0]
ColorI = [1.0, 0.65, 0.0]
ColorR = [0.0, 1.00, 0.7]

color_mu = 'tab:blue'
color_k = 'tab:red'
color_b = 'tab:green'
#Function for plotting the results
def plotting(N, T_data, X_data, T_exact, X_exact, T_r, mu,
    std):
    plt.figure(figsize=(1100/72,400/72))

    plt.scatter(T_data[:N, :].numpy().flatten(), X_data[:N,
        :].numpy().flatten(),edgecolors=(0.0, 0.0, 0.7),marker
        ='.',facecolors='none',s=100, lw=1.0,zorder=3, alpha
        =1.0, label=r'Training data')

    plt.scatter(T_exact[::2], X_exact[::2], s=100,linewidth
        =1.0, marker='x',color=ColorI, label='True')


```

```
    plt.fill_between(T_r.numpy().flatten(), (mu+1.96*std).
       flatten(), (mu-1.96*std).flatten(), zorder=1, alpha
       =0.8, color=ColorR)
    plt.plot(T_r.numpy().flatten(), mu.flatten(), color=(0.7,
        0.0, 0.0),linewidth=3.0,linestyle="--", zorder=3,
       alpha=1.0,label=r'B-PINN (mu +- 2std)')


    #plt.yticks([0.3,0.9],fontsize=16)
    #plt.xticks([0.0,0.2,0.6,0.8],fontsize=16)
    plt.legend(loc='upper right',ncol=1, fancybox=True,
       framealpha=0.,fontsize=24)
    #plt.ylim([0.25,1])
    plt.xlabel("t",fontsize=24)
    plt.ylabel("x(t)",fontsize=24)
    plt.tight_layout()
    plt.savefig(path2saveResults+'/PINN_data_BPINN_.pdf')
    plt.show()


#%% load data


#
T_data, X_data, T_r, T_exact, X_exact = load_data.load_data()
plt.plot(T_data,X_data)
print('Tensorflow version: ', tf.__version__)
print('Tensorflow-probability version: ', tfp.__version__)

# end
# In[2]:


# define ODE
def ode_fn(t, fn, additional_variables):
    mu, k, b = additional_variables
    with tf.GradientTape() as g_tt:
        g_tt.watch(t)
        with tf.GradientTape() as g_t:
            g_t.watch(t)
            y = fn(t)
        y_der = g_t.gradient(y, t)
    y_dder = g_tt.gradient(y_der, t)
    f = (y_dder) + (c*y_der) + (y*omega2) - ((A*P0/m)+((A*P1*
       tf.sin(Omega*t))/m))

    return f
```

```
# #### Prior distributions and change of variables
#
# We define the prior distributions for $\mu, k, b$ to be
    independent LogNormal distributions:
# $$\log \mu \sim N(\log 2.2, 0.5) \\\log k \sim N(\log 350,
    0.5), \\\log b \sim N(\log 0.56, 0.5)$$
#
# In practice, we instead sample $\log\mu - \log 2.2, \log k
    - \log 350$ and $\log b -\log 0.56$, so that those three
    quantities are independent and identically standard normal
     random variables. Suppose we have one posterior sample of
     those three quntities, $\epsilon_\mu, \epsilon_k, \
    epsilon_b$, then to obtain posterior sample of $\mu, k, b$
    , we do the following:
# $$\mu = e^{\log 2.2 + \epsilon_\mu} = 2.2 e^{\epsilon_\mu},
     k = 350e^{\epsilon_k}, b = 0.56e^{\epsilon_b}.$$
#
# In this way, re-scaling is done and positivities are
    guaranteed.

# In[3]:
# N = 150 # 225, 200, 150, 100

# X_data = X_un_tf


def main(N):

    # create Bayesian neural network
    BNN = bnn.BNN(layers=[1,50,50,50, 1])
    # specify number of observations

    # specify noise level for PDE
    noise_ode = (0.25,0.25)
    # specify noise level for observations
    noise_u = 0.25
    # create Bayesian model
    model = bayesian.PI_Bayesian(
        x_u=T_data[:N,:],
        y_u=X_data[:N,:],
        # y_u=X_un_tf[:N, :],
        x_pde=T_r,
        pde_fn=ode_fn,
        L=4,
        noise_u=noise_u,
        noise_pde=noise_ode,
        prior_sigma=1,y0=y0,ydash0=ydash0
    )
    # compute log posterior density function
    log_posterior = model.build_posterior(BNN.bnn_fn)
```

```python
    # create HMC (Hamiltonian Monte Carlo) sampler
    hmc_kernel = hmc.AdaptiveHMC(
        target_log_prob_fn=log_posterior,
        init_state=BNN.variables+model.additional_inits,
        num_results=4000,
        num_burnin=4000,
        num_leapfrog_steps=50,
        step_size=0.001,
    )
    # In[4]:


    # sampling
    start_time = time.perf_counter()
    samples, results = hmc_kernel.run_chain()
    Acc_rate = np.mean(results.inner_results.is_accepted.numpy())
    print('Accepted rate: ', Acc_rate)
    print(results.inner_results.accepted_results.step_size[0].numpy())
    stop_time = time.perf_counter()
    print('Duration time is %.3f seconds'%(stop_time - start_time))

    u_pred = BNN.bnn_infer_fn(T_r, samples[:8])

    mu = tf.reduce_mean(u_pred, axis=0).numpy()
    std = tf.math.reduce_std(u_pred, axis=0).numpy()

    return model, samples, u_pred, mu, std, Acc_rate

# In[5]:


# compute posterior samples of x and store the results
# x_samples = BNN.bnn_infer_fn(T_r, samples[:2*model.L]).numpy()
# sio.savemat(
#     'results/out_{}.mat'.format(str(N)), {'x_samples': x_samples, 't': T_r.numpy()}
# )


# #### Posterior estimate on function



if __name__ == '__main__':
    N=80
    model, samples, u_pred, mu, std, Acc_rate = main(N)
```

```
    plotting(N, T_data, X_data, T_exact, X_exact, T_r, mu,
       std)



# In[6]:

"""


log_mu, log_k, log_b = samples[-3:]
mu, k, b = tf.exp(log_mu+model.log_mu_init), tf.exp(log_k+
    model.log_k_init), tf.exp(log_b+model.log_b_init)

"""



# In[7]:
"""
    def PlotHist(ax, prior, post, var, color, limY, limX,
       limX0):

    ax.hist(post, bins=num_bins, density=True, label='
       posterior of '+var, color=color, alpha=0.7)
    mean = np.round(np.mean(post),3)
    std = np.round(np.std(post),2)
    ax.set_ylim([0,limY])
    ax.set_xlim([limX0,limX])
    plt.title(var+' = '+str(mean)+'$\pm$'+str(std),fontsize
       =24, color=color)
    legend = plt.legend(loc='upper right',fontsize=24,ncol=1,
        fancybox=True, framealpha=0.)
    plt.setp(legend.get_texts(), color=color)






num_bins = 30



fig = plt.figure(figsize=(1200/72,800/72))
gs = fig.add_gridspec(2, 2)



s = model.additional_priors[0].sample(3000)
```

```
prior = (tf.exp(s + model.log_mu_init)).numpy()
post = mu.numpy()
var = '$c$'
ax1 = plt.subplot(gs[0, 0])


PlotHist(ax1, prior, post, var, color_mu,20, 20, 0.0)



s = model.additional_priors[1].sample(3000)
prior = (tf.exp(s + model.log_k_init)).numpy()
post = k.numpy()
var = '$k$'
ax2 = plt.subplot(gs[0, 1])


PlotHist(ax2, prior, post, var, color_k, 20, 20, 100)




s = model.additional_priors[2].sample(3000)
prior = (tf.exp(s + model.log_b_init)).numpy()
post = b.numpy()
var = '$x_0$'
ax3 = plt.subplot(gs[1, 0])


PlotHist(ax3, prior, post, var, color_b, 23, 20, 0.4)

plt.savefig(path2saveResults+'/BPINN_Para_v2.pdf')
"""
```

hmc.py

```
    import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp

tfd = tfp.distributions


class AdaptiveHMC:
    def __init__(
        self,
        target_log_prob_fn,
        init_state,
        num_results=1000,
        num_burnin=1000,
```

```
        num_leapfrog_steps=30,
        step_size=0.1,
    ):
        self.target_log_prob_fn = target_log_prob_fn
        self.init_state = init_state
        self.kernel = tfp.mcmc.SimpleStepSizeAdaptation(
            inner_kernel=tfp.mcmc.HamiltonianMonteCarlo(
                target_log_prob_fn=self.target_log_prob_fn,
                num_leapfrog_steps=num_leapfrog_steps,
                step_size=step_size,
            ),
            num_adaptation_steps=int(0.8 * num_burnin),
            target_accept_prob=0.75,
        )
        self.num_results = num_results
        self.num_burnin = num_burnin

    @tf.function
    def run_chain(self):
        samples, results = tfp.mcmc.sample_chain(
            num_results=self.num_results,
            num_burnin_steps=self.num_burnin,
            current_state=self.init_state,
            kernel=self.kernel,
        )
        return samples, results
```

generate_data.py

```

import deepxde as xd
import tensorflow as tf
import math
import numpy as np

from matplotlib import pyplot as plt

def generate_data(t_min,t_max,domain_pts,bnd_pts):
    #Physical Constants

    #w_n=float(input("W_n:"))
    #zeta=float(input("Zeta:"))
    #Defining the differential Equation

    """def ode_fn(t, fn, additional_variables):
        mu, k, b = additional_variables
        with tf.GradientTape() as g_tt:
            g_tt.watch(t)
            with tf.GradientTape() as g_t:
                g_t.watch(t)
```

```
            x = fn(t)
        x_t = g_t.gradient(x, t)
    x_tt = g_tt.gradient(x_t, t)
    f = 1/k*x_tt + mu/k*x_t + x - b

    return f
"""
#Initial Conditions
#k=float(input("Spring constant="))
k=5
#A=float(input("Area="))
A=10
#m=float(input("mass"))
m=5
#alpha=float(input("alpha"))
alpha=10
#P0=float(input("P0="))
P0=1
#P1=float(input("P1="))
P1=1/2
#Omega=float(input("Omega"))
Omega=3
#y0=float(input("y(0)"))
y0=2
#ydash0=float(input("y'(0)"))
ydash0=1
c=(A**2)/(alpha*m)
omega2=k/m
denominator=((omega2-(Omega)**(2))**(2))+((c*Omega)**(2))
def particular_solution(t):
    return ((A*P0)/(m*omega2))+(A*P1*(((omega2-(Omega)**2)*np.sin(Omega*t))-c*Omega*np.cos(Omega*t)))/(m*denominator)
e=math.exp(1)
def solution(t):
    if c**2<4*omega2:
        a=-c/2
        b=math.sqrt(4*omega2-c**2)/2
        c1=y0+((A*P1*Omega*c)/(m*(denominator)))-((A*P0)/(m*omega2))
        c2=(1/b)*((ydash0)-(c1*a)-((A*P1*Omega*(omega2-Omega**2))/(m*(denominator))))
        return (e**(a*t)*c1*np.cos(b*t))+(e**(a*t)*c2*np.sin(b*t))+(particular_solution(t))
    elif c**2==4*omega2:
        s=-c/2
        c1=y0-((A*P0)/(m*omega2))+((A*P1*c*Omega)/(m*denominator))
        c2=ydash0-c1*s+(A*P1*(Omega**2-omega2))/(m*denominator)
```

```
            return (c1*e**(s*t))+(c2*t*e**(s*t))+particular_solution(t)
        else:
            s1=(-c-np.sqrt(c**2-4*omega2))/2
            s2=(-c+np.sqrt(c**2-4*omega2))/2
            c1=1/(s1-s2)*(ydash0+((A*P0*s2)/(m*omega2))-(s2*y0)+((A*P1*(Omega*(Omega**2-omega2)-c*Omega*s2))/(m*denominator)))
            c2=y0-c1+((A*P1*c*Omega)/(m*denominator))-((A*P0)/(m*omega2))
            return (c1*e**(s1*t))+(c2*e**(s2*t))+particular_solution(t)

    def ode(x,y):
        y_der=xd.grad.jacobian(y,x,i=0)
        y_dder=xd.grad.hessian(y,x,i=0)
        return (y_dder) + (c*y_der) + (y*omega2) - ((A*P0/m)+((A*P1*tf.sin(Omega*x))/m))
    #y0=float(input("y(0)"))
    #ydash0=float(input("y'(0)"))
    geom=xd.geometry.TimeDomain(0,25)
    def boundary(x,on_initial):
        return xd.utils.isclose(x[0],0)and on_initial
    ic1=xd.icbc.IC(geom,lambda x:y0,boundary)
    def error(inputs,outputs,X):
        return xd.grad.jacobian(outputs,inputs,i=0)-ydash0
    ic2=xd.icbc.OperatorBC(geom,error, boundary)
    data=xd.data.TimePDE(geom,ode,[ic1,ic2],domain_pts,bnd_pts,solution=solution,num_test=500)
    #x=np.array(data.train_points())
    x=np.array(data.train_x)
    y=np.array(data.train_y)
    j=np.sort(x,axis=0)
    k=y[x.argsort(axis=0)]
    k_new=k.reshape((int(len(k)),1))
    print(k)
    print(j)
    filo=open("new_data.txt","w")
    for i in range(0,len(j)):
        filo.write(f"{float(j[i])}  {float(k_new[i])}\n",)
    together=np.array([j,k_new])
    filo.close()
    tog_new=np.reshape(together,(domain_pts+bnd_pts+2,2))
    plt.plot(j,k_new)
    file =open("deep_data.txt","w")
    np.savetxt(file,tog_new)
    return j,k_new
if __name__=="__main__":
    generate_data(0, 25, 3000, 2)
```

bnn.py

```python
    import numpy as np
import tensorflow as tf


class BNN:
    def __init__(self, layers, activation=tf.tanh):
        self.L = len(layers) - 1
        self.variables = self.init_network(layers)
        self.bnn_fn = self.build_bnn()
        self.bnn_infer_fn = self.build_infer()
        self.activation = activation

    def init_network(self, layers):
        W, b = [], []
        init = tf.zeros
        # init = tf.keras.initializers.glorot_normal()
        for i in range(self.L):
            W += [init(shape=[layers[i], layers[i + 1]], dtype=tf.float32)]
            b += [tf.zeros(shape=[1, layers[i + 1]], dtype=tf.float32)]
        return W + b

    def build_bnn(self):
        def _fn(x, variables):
            """
            BNN function, for one realization of the neural network, used for MCMC

            Args:
            -----
            x: input,
                tensor, with shape [None, input_dim]
            variables: weights and bias,
                list of tensors, each one of which has dimension [:, :]

            Returns:
            --------
            y: output,
                tensor, with shape [None, output_dim]
            """
            W = variables[: len(variables) // 2]
            b = variables[len(variables) // 2 :]
            y = x
            for i in range(self.L - 1):
                y = self.activation(tf.matmul(y, W[i]) + b[i])
```

```python
            return tf.matmul(y, W[-1]) + b[-1]

        return _fn

    def build_infer(self):
        def _fn(x, variables):
            """
            BNN function, for batch of realizations of the
               neural network, used for inference

            Args:
            -----
            x: input,
                tensor, with shape [None, input_dim]
            variables: weights and bias,
                list of tensors, each one of which has
                   dimension [batch_size, :, :]

            Returns:
            --------
            y: output,
                tensor, with shape [batch_size, None,
                   output_dim]
            """
            W = variables[: len(variables) // 2]
            b = variables[len(variables) // 2 :]
            batch_size = W[0].shape[0]
            y = tf.tile(x[None, :, :], [batch_size, 1, 1])
            for i in range(self.L - 1):
                y = self.activation(tf.einsum("Nij,Njk->Nik",
                    y, W[i]) + b[i])
            return tf.einsum("Nij,Njk->Nik", y, W[-1]) + b
               [-1]

        return _fn
```

bayesian.py

```python
    import tensorflow as tf
import tensorflow_probability as tfp


tfd = tfp.distributions


class PI_Bayesian:
    def __init__(
        self,
        x_u,
        y_u,
```

```
        x_pde,
        pde_fn,
        y0,ydash0,
        L=6,
        noise_u=0.05,
        noise_pde=0.05,
        prior_sigma=1.0,

    ):
        self.x_u = x_u
        self.y_u = y_u
        self.x_pde = x_pde
        self.y0=y0
        self.ydash0=ydash0
        self.pde_fn = pde_fn
        self.L = L
        self.noise_u = noise_u
        self.noise_pde = noise_pde
        self.prior_sigma = prior_sigma

        self.log_mu_init = tf.math.log(2.2)
        self.log_k_init = tf.math.log(350.0)
        self.log_b_init = tf.math.log(0.56)

        # self.log_mu_init = tf.math.log(-5.0)
        # self.log_k_init = tf.math.log(1.0)
        # self.log_b_init = tf.math.log(0.56)

        self.additional_inits = [self.log_mu_init, self.log_k_init, self.log_b_init]
        self.additional_priors = [
            tfd.Normal(0, scale=0.5),
            tfd.Normal(0, scale=0.5),
            tfd.Normal(0, scale=0.5),
        ]
    def build_posterior(self, bnn_fn):
        y_u = tf.constant(self.y_u, dtype=tf.float32)

        def _fn(*variables):
            """
            log posterior function, which takes neural network's parameters input, and outputs (probably unnormalized) density probability
            """
            # split the input list into variables for neural networks, and additional variables
            variables_nn = variables[: 2 * self.L]
            log_mu, log_k, log_b = variables[2 * self.L :]
            mu, k, b = (
                tf.exp(log_mu + self.log_mu_init),
```

```
            tf.exp(log_k + self.log_k_init),
            tf.exp(log_b + self.log_b_init),
        )
        # explicitly create a tf.Tensor here, for input to neural networks, to avoid bugs
        x_u = tf.constant(self.x_u, dtype=tf.float32)
        x_pde = tf.constant(self.x_pde, dtype=tf.float32)

        # make inference
        _fn = lambda x: bnn_fn(x, variables_nn)
       # y_u_pred = _fn(x_u)
        pde_pred = self.pde_fn(x_pde, _fn, [mu, k, b])

        # construct prior distributions, likelihood distributions
        u_likeli = tfd.Normal(loc=y_u, scale=self.noise_u * tf.ones_like(y_u))
        bnd_likeli_1=tfd.Normal(loc=self.y0, scale=self.noise_u * 1)
        with tf.GradientTape() as g_t:
            g_t.watch(x_u)
            y_u_pred=_fn(x_u)
        u_t = g_t.gradient(y_u_pred, x_u)
        bnd_likeli_2=tfd.Normal(loc=self.ydash0, scale=self.noise_u * 1)
        noise_pde1, noise_pde2 = self.noise_pde
        N1, N2 = y_u_pred.shape[0], pde_pred.shape[0]
        pde_likeli_1 = tfd.Normal(
            loc=tf.zeros([N1, 1]), scale=noise_pde1 * tf.ones([N1, 1])
        )
        pde_likeli_2 = tfd.Normal(
            loc=tf.zeros([N2 - N1, 1]), scale=noise_pde2 * tf.ones([N2 - N1, 1])
        )
        # pde_likeli = tfd.Normal(loc=tf.zeros_like(pde_pred), scale=self.noise_pde*tf.ones_like(pde_pred))

        prior = tfd.Normal(loc=0, scale=self.prior_sigma)

        # compute unnormalized log posterior, by adding log prior and log likelihood
        log_prior = tf.reduce_sum(
            [tf.reduce_sum(prior.log_prob(var)) for var in variables_nn]
        ) + tf.reduce_sum(
            [
                dist.log_prob(v)
                for v, dist in zip([log_mu, log_k, log_b
```

```
                    ], self.additional_priors)
                ]
            )
            # log_prior += tf.reduce_sum([dist.log_prob(v)
               for v, dist in zip([(mu-2.2)/2.2, (k-370.0)
               /370.0, (b-0.56)/0.56], self.additional_priors
               )])
            log_likeli = (
                tf.reduce_sum(u_likeli.log_prob(u_t))
                +1000*tf.reduce_sum(bnd_likeli_1.log_prob(
                   y_u_pred[0]))
                +1000*tf.reduce_sum(bnd_likeli_2.log_prob(u_t
                   [0]))
                + tf.reduce_sum(pde_likeli_1.log_prob(
                   pde_pred[:N1, :]))
                + tf.reduce_sum(pde_likeli_2.log_prob(
                   pde_pred[N1:N2, :]))
            )
            return log_prior + log_likeli

        return _fn
```

11 Results and inferences

The plots below show the results obtained by the above codes. The y-axis in the plot corresponds to the displacement of the piston and the x-axis represents time. The set of hyper-parameters that were chosen are as in the tables below

Learning Rate	No. of Training points	Iterations	Optimizer	Training Interval(sec)
0.001	50	1000	Adam	0-25

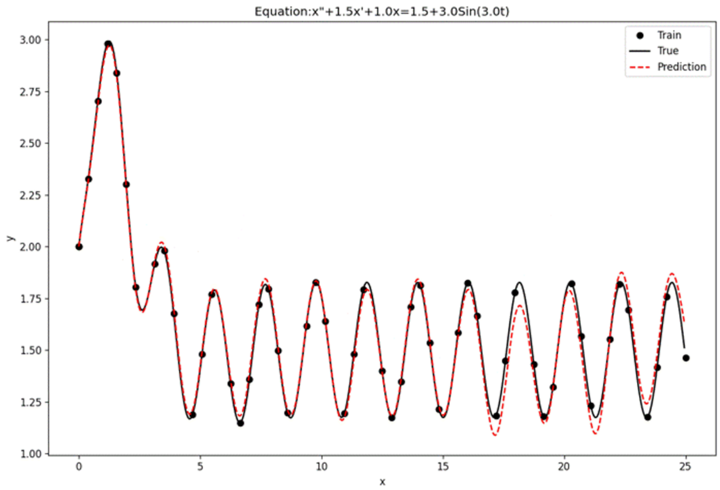
Table 1: Hyper-Parameters for PINNs

Step Size(HMC)	Training points	Iterations (Burn in)	Iterations (Total)	leap frog steps
0.001	70	4000	8000	50

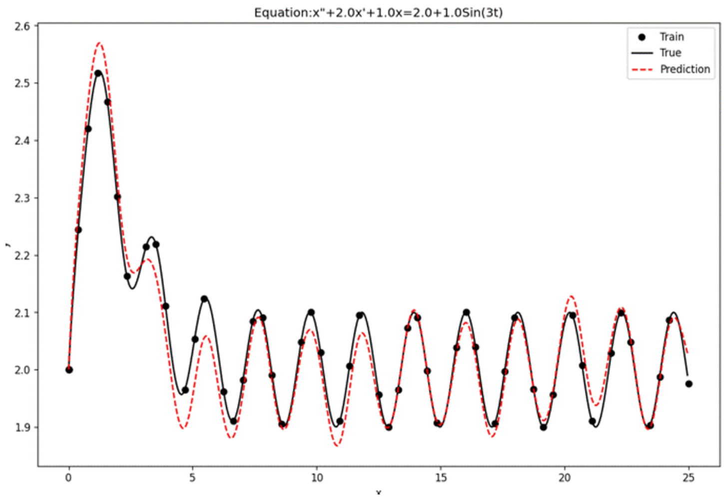
Table 2: Hyper-Parameters for Bayesian PINNs

12 Further work/ Suggestions of the research project

- Working on Integration of Bayesian PINNs model to DeepXde to make a more universal module.
- Working on application of Bayesian PINNs and PINNs on string vibration PDE.
- Working with other neural network architectures suitable for solving physical problems.

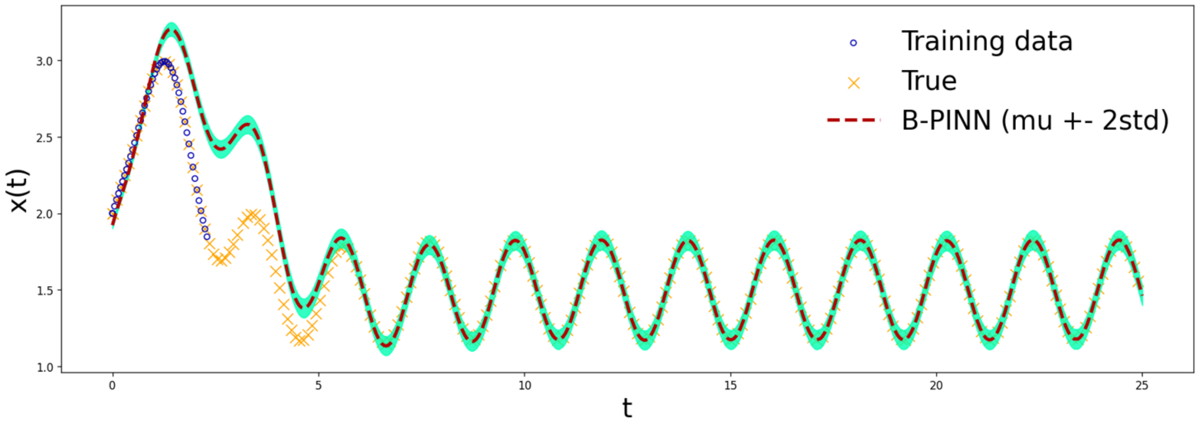


(a) Under Damped Case(PINNs)

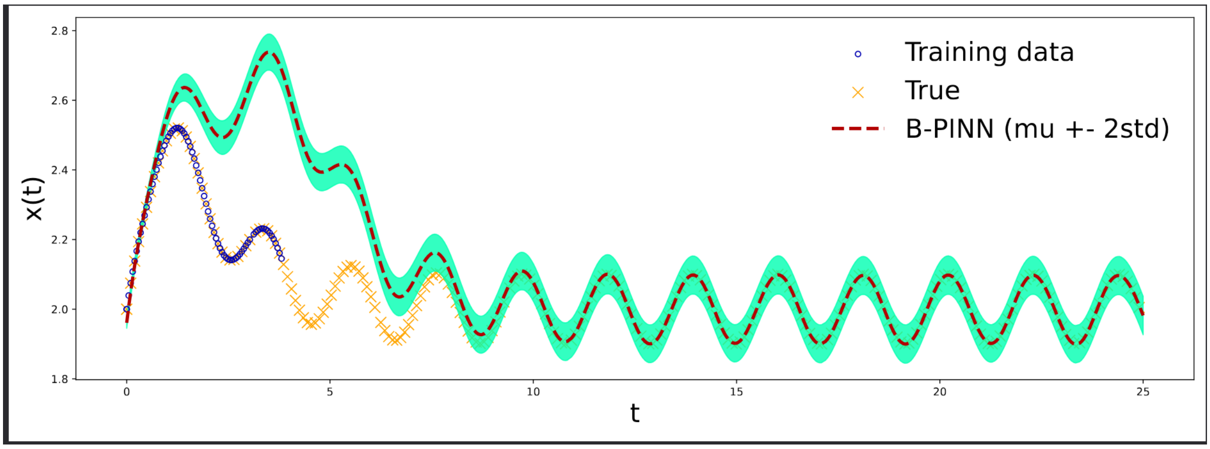


(b) Critically Damped Case(PINNs)

Figure 4: Output results for PINNs



(a) Under Damped Case (PINNs)



(b) Critically Damped Case (PINNs)

Figure 5: Output results for Bayesian PINNs

Bibliography

[1] Laurene V Fausett. *Fundamentals of neural networks: architectures, algorithms and applications.* Pearson Education India, 2006.

[2] Laurent Valentin Jospin, Hamid Laga, Farid Boussaid, Wray Buntine, and Mohammed Bennamoun. Hands-on bayesian neural networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48, 2022.

[3] Kevin Linka, Amelie Schäfer, Xuhui Meng, Zongren Zou, George Em Karniadakis, and Ellen Kuhl. Bayesian physics informed neural networks for real-world nonlinear dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, 402:115346, 2022.

[4] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[5] Sheldon M Ross. *Introduction to probability models.* Academic press, 2014.

[6] Simon Stock, Jochen Stiasny, Davood Babazadeh, Christian Becker, and Spyros Chatzivasileiadis. Bayesian physics-informed neural networks for robust system identification of power systems. In *2023 IEEE Belgrade PowerTech*, pages 1–6. IEEE, 2023.

[7] Wei-Chau Xie. *Differential equations for engineers.* Cambridge university press, 2010.

[8] Liu Yang, Xuhui Meng, and George Em Karniadakis. B-pinns: Bayesian physics-informed neural networks for forward and inverse pde problems with noisy data. *Journal of Computational Physics*, 425:109913, 2021.

[9] Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*, 2020.