Tentative title: PINNs approaches to simulation solutions of differential equations

3. Simulation in Python

Python is a popular programming language known for its versatility in areas like web development, automation, and, notably, machine learning and data science. Extensively adopted within academia and industry alike, Python boasts a vast library ecosystem that encompasses modules, datasets, and comprehensive documentation. The Python Package Index (PyPI) is the principal hub for Python packages, hosting over 300,000 libraries ready for deployment.

Python and its extensive library support present many opportunities to teach a recent innovation in machine learning: **Physics Informed Neural Networks** (PINNs). Along with its more human-legible syntax, Python also furnishes specialized libraries—like Pandas, Numpy, Matplotlib, and PyTorch—that are optimized for the entire machine-learning workflow, from data pipelines to model training to deployment. By leveraging Python, educators can readily teach students both applied and theoretical aspects of topics in mathematics or computer science, easily visualize results through its extensive graphing capabilities, and even quickly abstract computer science processes into beginner-friendly functions. Educators can pursue any level of granularity with Python code, and such versatility coupled with its robust and widespread use, make Python an ideal choice for teaching topics in deep learning such as PINNs.

Along with the selection of the language, Ben Moseley's "harmonic-oscillator-pinn" example notebook, Karniadakis et. al.'s "Physics-informed machine learning", and Raissi's et. al. "PINNs: A deep learning framework for solving problems" serve as great case studies for PINNs in an applied setting. This paper will reimplement Moseley's original codebase while remarking on the various educational teaching points in this exercise, __, and __.

In this section, we will discuss the installation of Python in Section 3.1, code a simulation of the harmonic oscillation in Section 3.2 by using PyTorch, __ in Section 3.3, and __ in Section 3.4. Then, <conclusion> __ for Section 3.5.

3.1. Setting Up Python and Key Libraries

3.1.1 Setting Up Python on Local Machine

Setting up Python and virtual environments on a local machine affords you full control over your development environment, ensuring privacy, persistent storage, and seamless integration with local files and hardware. It is particularly beneficial for long-term projects requiring significant customization, as you can maintain, modify, and optimize your environment without restrictions. However, the downside includes the manual management of dependencies, potential difficulty in replicating environments, and the need for your hardware to support certain computational tasks, which can be costly if dealing with high-performance requirements, as in machine learning or data analysis with large datasets. If you prefer setting up your workspace on remote servers, then skip onto Section 3.1.2.

Installing Python

Visit the official Python website to download and then run the integrated installer. Depending on your operating system (Windows, macOS, or Linux), you'll see a recommended version for download. During installation, check the box that says "Add Python 3.x to PATH". This will allow you to run Python from the command prompt. To verify a successful installation, open your terminal and run the following command. If you're operating on a Windows, your terminal is called either “Command Prompt” or “GitBash”; otherwise, it is called “Terminal”.

Windows:

```
python --version
```

macOS and Linux:

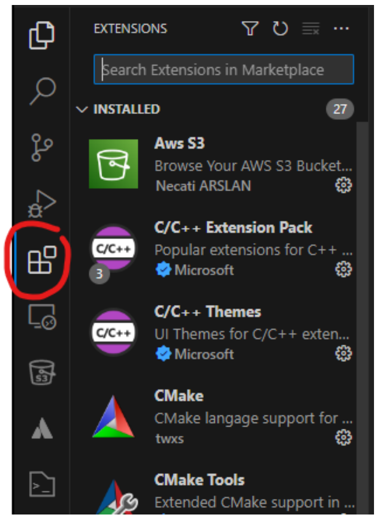
```
python3 --version
```

If a version number is returned, then the installation was successful. As part of this installation, you should have also automatically installed pip, Python's package manager, which will let you install additional libraries and tools for Python with ease.

Setting up Visual Studio Code (VS Code)

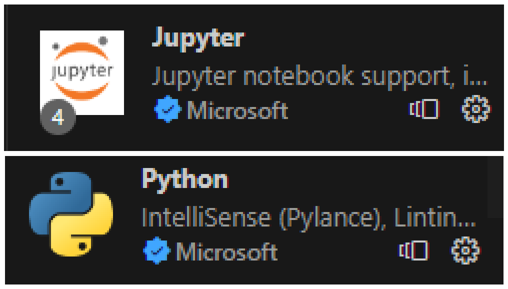
For code development and debugging purposes, the utilization of an integrated development environment (IDE) is recommended. Visual Studio Code is a contemporary, extensible IDE that caters to multiple programming languages. Download the systems-appropriate version of installer from its official website and proceed with its installation.

Once finished, navigate to “Extensions” within VS Code to unlock this IDE's full potential. Extensions are additional features created by other creators or organizations (e.g. Microsoft) that are officially supported and provide enhanced functionality for Python, such as IntelliSense, debugging, environment selection, etc. You can navigate to the “Extensions” view by clicking on the Extensions icon in the Activity Bar on the left side of the VS Code window or the View: Extensions command (Ctrl+Shift+X).


EXTENSIONS
Search Extensions in Marketplace
INSTALLED 27
Aws S3
Browse Your AWS S3 Bucket...
Necati ARSLAN
C/C++ Extension Pack
Popular extensions for C++ ...
Microsoft
C/C++ Themes
UI Themes for C/C++ exten...
Microsoft
CMake
CMake langage support for ...
twxs
CMake Tools
Extended CMake support in ...

Note: the image above is taken from a preexisting VS Code environment, so your version may not directly match this.

Two extensions to install are "Jupyter" and "Python", both of which are developed and maintained by Microsoft. Click their installation buttons and let them run to completion (note: these extensions will also install several smaller companion extensions like "Pylance", which are also useful). You may need to restart VS Code to complete this process.


Jupyter
Jupyter notebook support, i...
Microsoft
Python
IntelliSense (Pylance), Lintin...
Microsoft

Setting up virtual environments and interpreters

With an IDE set up, virtual environments can be easily set up using conda (automatically installed when you installed Python), as well. Virtual environments help isolate project-specific dependencies, which in turn allow users to swap in between depending on the task at hand. This reduces library bloat (since users only need to load the libraries they need) and maintains project consistency.

To get started, open the integrated terminal within VS Code (in VSCode, type in Ctrl+` or view > Terminal). Type the following command to create a new environment; replace 'myenv' with whatever you wish to name your environment, and '[version]' with your specific Python version (if you have a preference).

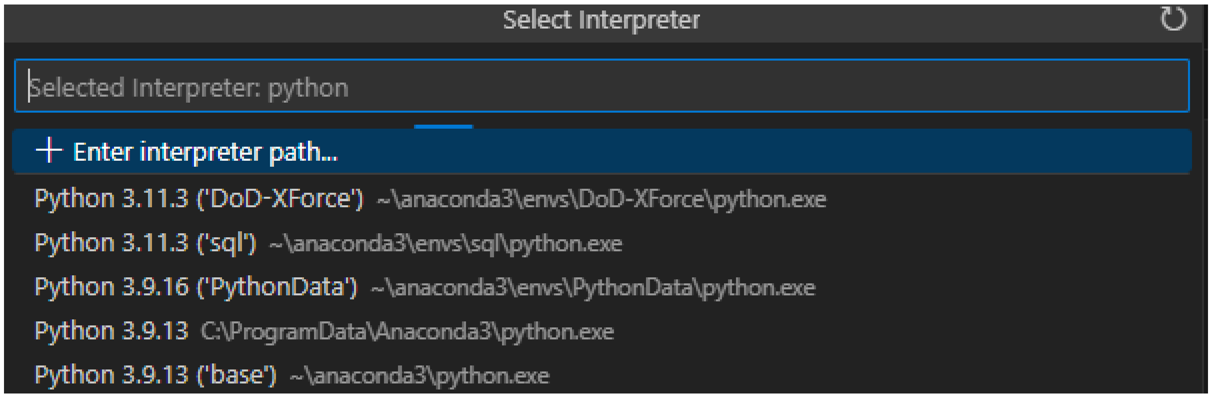
```
conda create --name myenv python=[version]
```

Then, to start using the environment you just created within the terminal, activate it with this command.

```
conda activate myenv
```

Explicitly, this allows you to run your virtual environment's libraries within the terminal interface, but it does not automatically let you run those libraries within VS Code. To also enable your environment in VS Code, we need to designate it as the interpreter. For context, VS Code works with many languages beyond just Python, and thus, needs to know when to interpret a block of text as Python code, C++ code, LaTeX, or even just plain text. By designating an interpreter (which would be our virtual environment that contains Python), we ensure that our VS Code will read and run Python.

To get started, use the command Ctrl+Shift+P to open the Command Palette. Start typing "Python: Select Interpreter" in the Command Palette. A list will pop up showing all the Python interpreters VSCode can find, including those from your conda environments. Select the interpreter that corresponds to the conda environment you created.


Select Interpreter

Selected Interpreter: python

+ Enter interpreter path...

Python 3.11.3 ('DoD-XForce') ~\anaconda3\envs\DoD-XForce\python.exe
Python 3.11.3 ('sql') ~\anaconda3\envs\sql\python.exe
Python 3.9.16 ('PythonData') ~\anaconda3\envs\PythonData\python.exe
Python 3.9.13 C:\ProgramData\Anaconda3\python.exe
Python 3.9.13 ('base') ~\anaconda3\python.exe

Note: the image above is taken from a preexisting VS Code environment, so your version may not directly match this.

Installing Pip and Essential Packages

Pip, the package installer for Python, is typically included in modern distributions of Python. It simplifies the process of installing and managing additional libraries or dependencies. With Pip in place, one can then effortlessly incorporate essential deep-learning libraries:

- Numpy for numerical computations
- Matplotlib for data visualization

- PyTorch for general supervised, unsupervised, and deep learning

To get started, activate the virtual environment within your terminal. This virtual environment should be the location in which you want to install the libraries.

```
conda activate myenv
```

If this activation step is not completed, Pip will install the packages to the "base" environment, which represents the default overall working environment. It is generally a good practice to keep your packages within specific environments rather than all in "base", as it makes it easier to update old packages and troubleshoot inter-package conflicts should they arise in the future.

Next, input the following command, where the text following "install" and delimited by a space represents the names of the various libraries being installed.

```
pip install numpy matplotlib torch torchvision
```

Let the installations run to completion, and you may be required to restart your terminal to see the effects take place. Remember to activate your virtual environment within the terminal or select it as your interpreter within VS Code to allow working with the packages.

At this point, you are ready to start programming in Python on your local machine.

3.1.2 Setting Up Python on Remote Servers

Leveraging servers for computational processes poses advantages over working on your local machine. Google Colab is a streamlined process because Colab provides a host of Google computing resources (like GPUs) for free, which are particularly useful for intensive computations required by ML processes. Google Colab simplifies the setup by providing a pre-configured environment, while simultaneously enhancing collaboration, version control, and environment sharing (everything is cloud-based). However, it's not without its limitations: you are subjected to usage quotas, limited session times, and potential internet dependency issues. Moreover, there's less control over the environment's specifics, and concerns regarding data security and privacy, especially with sensitive information, as your data is uploaded to the cloud. The temporary nature of the storage means additional steps to ensure your data and environment settings are consistently backed up if needed for future sessions.

Connecting to a server

Visit Google Colab and sign in with a Google account. If you use Gmail, this should be the same account. Once you're signed in, you can create a new notebook by clicking on "File" > "New notebook". You'll see a new tab with a fresh notebook, where you can write and execute Python code in code cells.

If you need high-intensity computational resources (like a GPU or TPU): Click on "Runtime" in the menu, choose "Change runtime type", and then in the pop-up, select "GPU" or "TPU" from the "Hardware accelerator" dropdown menu. Click "Save". This step is optional if you anticipate large computational workloads.

You are now ready to work in Colab as if it were a VS Code IDE!

Installing Essential Packages

Google Colab comes with many popular libraries installed (like numpy, pandas, matplotlib). However, you might need other libraries for ML (like TensorFlow, PyTorch, etc.). To install them, you use !pip install command. For instance:

```
!pip install tensorflow
```

Setting up Virtual Environments

Every time you run a Colab notebook, Google creates a new virtual environment for your run instance, which gets recycled when the session ends. This means there is no virtual environment setup needed (like the local machine case), but the tradeoff is that you will need to reinstall libraries like Tensorflow every time you want to run the code again.

3.2 Harmonic Oscillator Simulation with Traditional Neural Networks

Process overview

Initially, we'll teach a conventional neural network to estimate a segment of the solution using certain known data points from that solution. Subsequently, we'll instruct a PINN to predict the entire solution beyond these data points by incorporating the foundational differential equation into its loss criteria. This reimplementation will be done as if Python was locally installed on your machine.

Environment setup

Following the Moseley code, we first create our virtual environment, specifying that we want to use Python 3.

```
conda create -n pinn python=3
conda activate pinn
```

Next, we install the following library packages. Note, Moseley uses conda instead of pip to do the installation. Both are package managers that come with the default installation of Python, so in practice, they pose little difference.

```
conda install jupyter numpy matplotlib
conda install pytorch torchvision torchaudio -c pytorch
```

Imports

Next, we want the following imports called in our code.

```python
from PIL import Image # import Pillow

import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

Custom Functions

Next, we'll implement some custom functions and classes to assist with modeling later down the road. These functions will be called 'save_gif_PIL' and 'oscillator', while the custom class will be called 'FCN'.

```python
def save_gif_PIL(outfile, files, fps=5, loop=0):
    "Helper function for saving GIFs"
    imgs = [Image.open(file) for file in files]
    imgs[0].save(fp=outfile, format='GIF', append_images=imgs[1:],
save_all=True, duration=int(1000/fps), loop=loop)
```

The function 'save_gif_PIL' is designed to create an animated GIF from a sequence of images using Python, specifically employing the PIL library, often known for handling various image processing tasks. The function accepts four parameters: 'outfile', which specifies the name and path for the resulting GIF; 'files', a list containing the paths to the image files to be used in the sequence; 'fps', or frames per second, indicating how many images should be displayed per second in the animated GIF (defaulting to 5); and 'loop', determining how many times the GIF will repeat (with the default 0 meaning it loops indefinitely).

Within the function, it first reads and opens the series of image files specified in 'files', storing them in a list called 'imgs'. The 'save' method is then called on the first image in this sequence, and it is instructed to compile an animated GIF. The 'append_images' parameter is particularly noteworthy: it includes all subsequent images in 'imgs' beyond the first one, indicating these should follow in the animation sequence. The 'save_all' parameter ensures all images in 'imgs' are included in the output file. 'Duration' controls the display time for each image frame in the GIF, calculated based on the 'fps' value, converted to the time each image should be shown in milliseconds. The 'loop' parameter is passed as-is, dictating the looping behavior of the animation. The process culminates with the creation of an animated GIF file saved to the location specified by 'outfile'.

```
def oscillator(d, w0, x):
    """Defines the analytical solution to the 1D underdamped harmonic 
oscillator problem.
    Equations taken from: 
https://beltoforion.de/en/harmonic_oscillator/"""
    assert d < w0
    w = np.sqrt(w0**2-d**2)
    phi = np.arctan(-d/w)
    A = 1/(2*np.cos(phi))
    cos = torch.cos(phi+w*x)
    sin = torch.sin(phi+w*x)
    exp = torch.exp(-d*x)
    y  = exp*2*A*cos
    return y
```

This Python function, 'oscillator', computes the analytical solution for a one-dimensional underdamped harmonic oscillator's motion. The oscillator is described in terms of its damping ratio 'd', natural frequency 'w0', and a sequence of time points 'x'. It's important that the system is underdamped, which implies that the damping ratio 'd' is less than the natural frequency 'w0'; this is enforced at the start of the function with an assertion check.

The function calculates the actual frequency 'w' of the damped oscillation using the formula derived from the system's parameters, essentially adjusting the natural frequency by the effect of damping. It then computes the phase shift 'phi' caused by the damping using the arctangent function, ensuring the accurate representation of the oscillatory motion under non-ideal conditions.

The amplitude 'A' of the oscillator is calculated considering the phase shift, compensating for changes in magnitude due to the non-zero damping. Subsequently, the function calculates the cosine and sine of the sum of 'phi' and the product of 'w' and 'x', representing the periodic nature of the motion. These values are multiplied by the exponential decay factor, 'exp', which models the reduction of amplitude over time due to the damping effect.

The final position 'y' of the oscillator at each point in time is obtained by combining these calculations, specifically by multiplying 'exp' with '2*A' and 'cos', the latter representing the harmonic motion's cyclical pattern. This comprehensive mathematical model allows for the precise depiction of an underdamped harmonic oscillator's behavior over time, reflecting foundational principles of classical physics. The function returns 'y', providing a series of positional values corresponding to the input times in 'x', fully characterizing the oscillator's motion trajectory.

```python
class FCN(nn.Module):
    "Defines a connected network"

    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS):
        super().__init__()
        activation = nn.Tanh
        self.fcs = nn.Sequential(*[
                        nn.Linear(N_INPUT, N_HIDDEN),
                        activation()])
        self.fch = nn.Sequential(*[
                        nn.Sequential(*[
                            nn.Linear(N_HIDDEN, N_HIDDEN),
                            activation()]) for _ in range(N_LAYERS-
1)])
        self.fce = nn.Linear(N_HIDDEN, N_OUTPUT)

    def forward(self, x):
        x = self.fcs(x)
        x = self.fch(x)
        x = self.fce(x)
        return x
```

The code defines a Python class 'FCN', representing a fully connected neural network (also known as a dense network), typically used in deep learning for various pattern recognition tasks. This class is a blueprint for creating an object that embodies a neural network model. It is built upon the 'nn.Module' base class provided by PyTorch, which we had previously installed.

In the 'init' method, the network's architecture is established. The parameters 'N_INPUT', 'N_OUTPUT', 'N_HIDDEN', and 'N_LAYERS' specify the number of input features, the number of expected outputs, the number of hidden units in each layer, and the number of layers in the network, respectively. The 'activation' variable defines the type of activation function to be used in the network's neurons, in this case, 'Tanh' (hyperbolic tangent), a common choice for promoting non-linearity in learning patterns.

The 'self.fcs' attribute creates the network's initial layer, transforming the input data to a higher-dimensional hidden space ('N_HIDDEN'). 'self.fch' defines a sequence of intermediary fully connected layers (excluding the first and last) coupled with the chosen activation function, with the number of layers being 'N_LAYERS-1' as the first layer is already defined by 'self.fcs'. These layers serve to allow the network to learn more complex representations of the input data. Finally, 'self.fce' establishes the end layer of the network, mapping the representation from the hidden layers down to the output space dimension ('N_OUTPUT').

The 'forward' method defines the sequence of operations the model performs to generate predictions from input data 'x'. Here, the data is passed through the initial layer 'fcs', propagating through the intermediary hidden layers 'fch', and finally through the end layer 'fce'. Each stage applies its transformations and activation functions, distilling the raw input into a prediction output through the network's learned mappings. The method returns 'x', which, at this point, holds the network's predictions corresponding to the provided inputs.

Generate Training Data

```
d, w0 = 2, 20

# get the analytical solution over the full domain
x = torch.linspace(0,1,500).view(-1,1)
y = oscillator(d, w0, x).view(-1,1)
print(x.shape, y.shape)

# slice out a small number of points from the LHS of the domain
x_data = x[0:200:20]
y_data = y[0:200:20]
print(x_data.shape, y_data.shape)
```

```
torch.Size([500, 1]) torch.Size([500, 1])
torch.Size([10, 1]) torch.Size([10, 1])
```

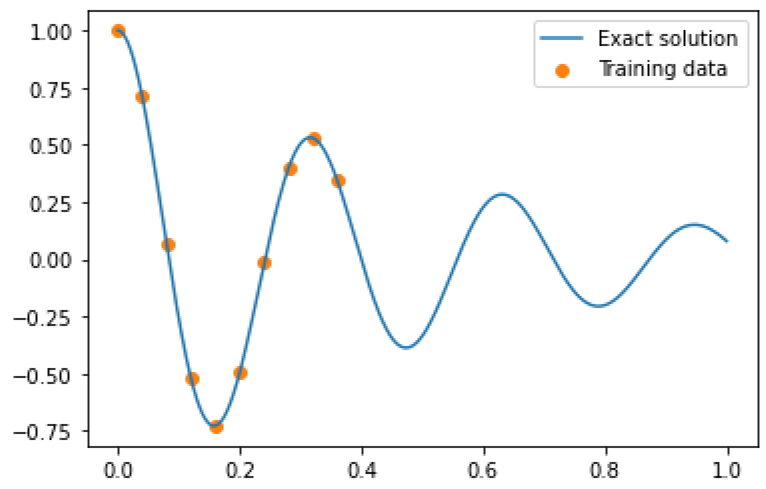
This snippet of Python code analyzes the behavior of a one-dimensional underdamped harmonic oscillator, as defined by the 'oscillator' function previously described. The system's damping ratio and natural frequency are initialized with the variables 'd' and 'w0', respectively, set to 2 and 20.

The 'torch.linspace' function generates a sequence of 500 points evenly spaced between 0 and 1, representing time intervals. These points are then reshaped into a two-dimensional tensor (or matrix) 'x' with 500 rows and 1 column, serving as the input for the 'oscillator' function to solve the system over the specified domain. The 'oscillator' function is called with 'd', 'w0', and 'x' as arguments, and the resulting positional values 'y' are similarly reshaped into a 500x1 tensor. The shapes of these tensors are printed to the console, confirming their dimensions.

Next, the code extracts a subset of the data points to simulate having a sparse dataset, often a realistic scenario where complete continuous data is not available. It selects every 20th point from the first 200 points in the 'x' and 'y' tensors, using slicing syntax. This operation significantly reduces the dataset size, resulting in 'x_data' and 'y_data' tensors with only 10 points each (effectively a 95% reduction from the original set). The shapes of these smaller tensors are printed, confirming their dimensions as 10x1.

This approach allows one to work with a more manageable number of data points, possibly emulating a situation where data is sampled at specific intervals. The smaller dataset can be particularly useful for various purposes such as testing, training with quicker computations, or situations where data is naturally scarce or expensive to collect. The printed output at the end confirms the shapes of the original and sliced data tensors.

```
plt.figure()
plt.plot(x, y, label="Exact solution")
plt.scatter(x_data, y_data, color="tab:orange", label="Training
data")
plt.legend()
plt.show()
```



The above code block visualizes the training data we have generated.

Function for saving visualizations

Before training our model, we can first make a custom function to visualize progress at every training step.

```
def plot_result(x,y,x_data,y_data,yh,xp=None):
    "Pretty plot training results"
    plt.figure(figsize=(8,4))
    plt.plot(x,y, color="grey", linewidth=2, alpha=0.8, label="Exact 
solution")
    plt.plot(x,yh, color="tab:blue", linewidth=4, alpha=0.8, 
label="Neural network prediction")
    plt.scatter(x_data, y_data, s=60, color="tab:orange", alpha=0.4, 
label='Training data')
    if xp is not None:
        plt.scatter(xp, -0*torch.ones_like(xp), s=60, 
color="tab:green", alpha=0.4, 
                    label='Physics loss training locations')
    l = plt.legend(loc=(1.01,0.34), frameon=False, fontsize="large")
    plt.setp(l.get_texts(), color="k")
    plt.xlim(-0.05, 1.05)
    plt.ylim(-1.1, 1.1)
    plt.text(1.065,0.7,"Training step: %i"%(i+1),fontsize="xx-
large",color="k")
    plt.axis("off")
```

This function is designed to visually present the outcomes of a neural network's training process, particularly one that has been trained to approximate solutions to a mathematical or physical system, like a differential equation.

Upon invocation, plot_result initiates a new figure with specified dimensions. It then proceeds to plot the exact solution of the system (presumably obtained through analytical methods) as a continuous line and the neural network's predictions as another continuous line. These are represented in different colors for clear distinction, with the exact solution typically being the ground truth the network is trying to approximate.

Additionally, the function plots individual data points used during training ('Training data'), highlighting them as scattered points. These points represent the dataset on which the neural network was trained - a subset of the full domain, as is often the case in machine learning for efficiency or due to limited data availability.

An optional set of points, represented by xp, may be included, indicating specific locations in the input domain where an additional physics-informed loss function was applied during training (if such an approach was used). This advanced technique helps steer the neural network's learning process using the underlying system's known physical laws, improving prediction accuracy.

The plot's aesthetics, such as legends, colors, transparency (alpha), and text annotations, are carefully controlled to deliver a presentation-ready visualization. Labels are used to identify the different datasets, and the legend is customized and positioned outside of the plot area to avoid obscuring the data.

Plotting NN

Now, we can finally plot the normal neural network is see its performance.

```
torch.manual_seed(123)
model = FCN(1,1,32,3)
optimizer = torch.optim.Adam(model.parameters(),lr=1e-3)
files = []
for i in range(1000):
    optimizer.zero_grad()
    yh = model(x_data)
    loss = torch.mean((yh-y_data)**2)# use mean squared error
    loss.backward()
    optimizer.step()
    # plot the result as training progresses
    if (i+1) % 10 == 0:

        yh = model(x).detach()

        plot_result(x,y,x_data,y_data,yh)

        file = "plots/nn_%.8i.png"%(i+1)
        plt.savefig(file, bbox_inches='tight', pad_inches=0.1,
dpi=100, facecolor="white")
        files.append(file)

        if (i+1) % 500 == 0: plt.show()
        else: plt.close("all")

save_gif_PIL("nn.gif", files, fps=20, loop=0)
```

It starts by setting a specific starting point for the random number generator to ensure consistency in results across different runs, essential for reproducibility. A neural network model is then created with specified architecture parameters, and an optimization strategy (Adam optimizer) is set up for adjusting the network's internal parameters based on its performance, with a defined learning rate.

In the training phase, the script enters a loop to incrementally improve the model. During each iteration, the model makes predictions, and the discrepancy between these predictions and the actual data is calculated using Mean Squared Error, a common measure of prediction accuracy. This 'loss' is used to adjust the model's internal workings slightly, aiming to reduce prediction error in subsequent iterations.

Simultaneously, the training process is visually documented. Periodically, the current state of the model's predictions is plotted against the actual data, providing a graphical representation of the model's performance. These plots are saved as individual image files. Additionally, at certain milestones, the script directly displays the plot to provide a real-time view of the progress. After the training loop concludes, all these saved images are compiled into an animated GIF, providing a visual recap of the entire training process. This approach not only helps in monitoring and debugging but also serves as an educational tool, illustrating the gradual improvement of the neural network model as it learns from the data.

Plotting PINN

With the traditional neural network outputs saved, we can do test the novel PINNs.

```
x_physics = torch.linspace(0,1,30).view(-1,1).requires_grad_(True)# sample locations over the problem domain
mu, k = 2*d, w0**2

torch.manual_seed(123)
model = FCN(1,1,32,3)
optimizer = torch.optim.Adam(model.parameters(),lr=1e-4)
files = []
for i in range(20000):
    optimizer.zero_grad()

    # compute the "data loss"
    yh = model(x_data)
    loss1 = torch.mean((yh-y_data)**2)# use mean squared error

    # compute the "physics loss"
    yhp = model(x_physics)
    dx  = torch.autograd.grad(yhp, x_physics, torch.ones_like(yhp), create_graph=True)[0]# computes dy/dx
    dx2 = torch.autograd.grad(dx,  x_physics, torch.ones_like(dx), create_graph=True)[0]# computes d^2y/dx^2
    physics = dx2 + mu*dx + k*yhp# computes the residual of the 1D harmonic oscillator differential equation
    loss2 = (1e-4)*torch.mean(physics**2)

    # backpropagate joint loss
    loss = loss1 + loss2# add two loss terms together
    loss.backward()
    optimizer.step()


    # plot the result as training progresses
    if (i+1) % 150 == 0:

        yh = model(x).detach()
        xp = x_physics.detach()
        plot_result(x,y,x_data,y_data,yh,xp)

        file = "plots/pinn_%.8i.png"%(i+1)
        plt.savefig(file, bbox_inches='tight', pad_inches=0.1,
dpi=100, facecolor="white")
        files.append(file)

        if (i+1) % 6000 == 0: plt.show()
        else: plt.close("all")

save_gif_PIL("pinn.gif", files, fps=20, loop=0)
```

The process begins by defining a set of points across the problem domain, requiring gradients for optimization, and setting key physical parameters (mu, k) based on known factors (d, w0) of the system being modeled, such as a harmonic oscillator.

A neural network model is then instantiated with a specific structure, and an algorithm (Adam optimizer) is prepared for its training with a specified learning rate. The script enters a rigorous training loop, aimed at dual objectives: fitting the data and adhering to the physical system's laws.

First, it calculates the 'data loss' by comparing the model's predictions to actual known data, aiming to minimize this discrepancy. Next, it calculates the 'physics loss', a novel step where the model's adherence to the governing differential equation is assessed. This involves computing the model's derivatives and evaluating the residual (the difference from zero) of the equation, capturing how closely the model's behavior mirrors the physical law.

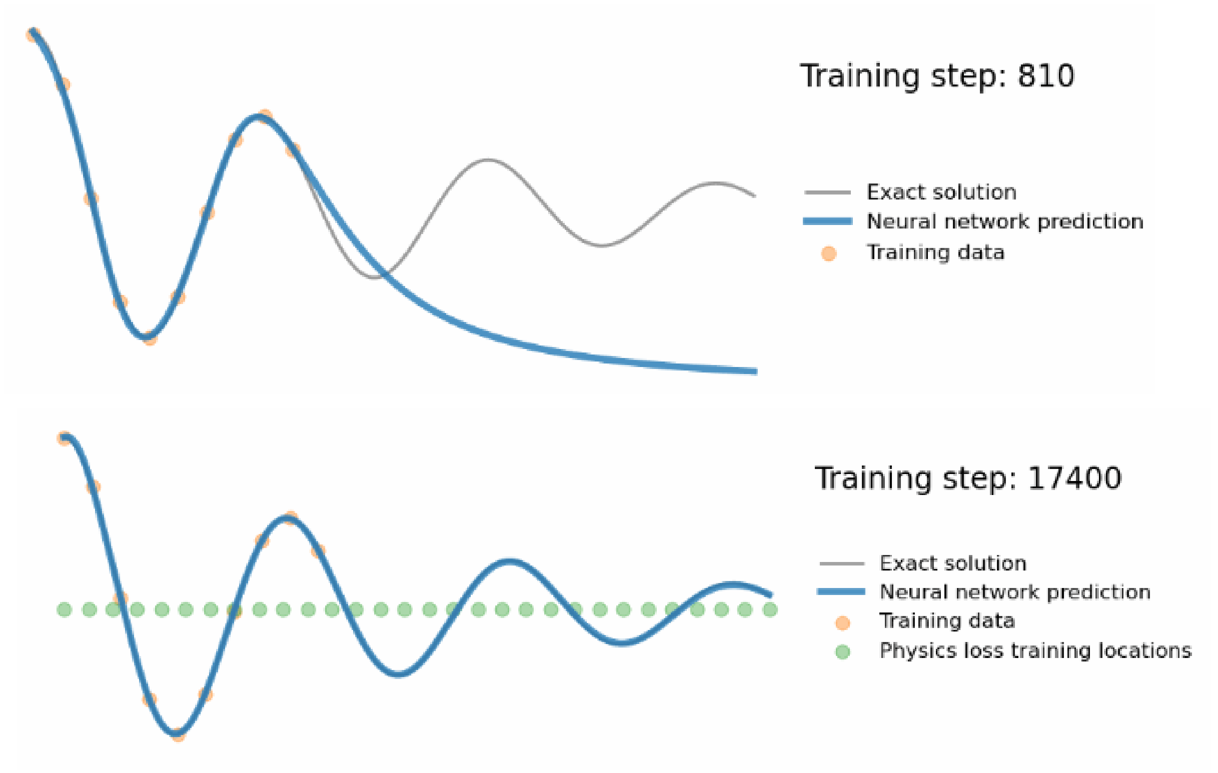
Both types of loss are combined, signifying that the model should simultaneously fit the data and respect the physical rules. This joint loss is used to adjust the model's parameters iteratively, refining its predictions.

Periodically within these iterations, the script generates and saves visual plots, contrasting the model's predictions against actual data and highlighting the points used for the physics-based training. These intermittent snapshots are stored for later compilation.

After thousands of such training cycles, marked by gradual improvements, the script compiles the saved plots into an animated GIF. This visual file succinctly demonstrates the model's journey, showcasing how it has learned from both the empirical data and the ingrained physical principles, offering a rich, multifaceted understanding of its development and capabilities.

Comparing NN to PINN

With the data graphed from both the NN and the PINN models, we can compare the results from the Moseley example below.



With traditional NNs, we see great performance in the short-run, but then the model stagnates in its predictive power as early as training step 810. With PINNs, the model continually improves itself until it approximates the actual solution almost perfectly, even predicting accurate results far away from the initial data points. PINNs thus have implications for not only model robustness but also for data efficiency, as it can get accurate results with much fewer data points.

The Moseley example is both a great way to showcase the power of PINNs compared to traditional NNs and an entry point for educating others about PINNs through a coding framework.