ECSE 458N2 - Capstone Design Project
Physics Informed Neural Network for electromagnetic field prediction

Design Document – PINN for Electromagnetic Prediction

Cedric Djossou, ID: 260927373, cedric.djossou@mail.mcgill.ca

Karim Atoui, ID: 260986223, karim.atoui@mail.mcgill.ca

Bo Wang, ID: 260961031, bo.wang6@mail.mcgill.ca

Supervisor: David Alister Lowther

Affiliation: Computational Electromagnetics

Email: david.lowther@mcgill.ca

Professor: Marwan Kanaan

McGill University

December 9th, 2024

Abstract

As part of our Capstone project, our team aimed to develop a Physics-Informed Neural Network (PINN) for predicting electromagnetic fields. Using Python libraries such as NumPy and Magpylib, alongside the Magnet software, we generated the training and validation datasets necessary to train different models. While we were unable to fully achieve our original objectives, this report outlines the progress made, evaluates the strengths and weaknesses of the models, and presents recommendations for future development. It serves as a guide for any subsequent design team to build upon our work, highlighting the remaining tasks and areas for improvement.

Table of Content

List of abbreviations

PINN	Physics-Informed Neural Network
ANN	Artificial Neural Network
MAPE	Mean Absolute Percentage Error
MSE	Mean Square Error
PDE	Partial Differential Equation

1. Purpose

The project is to explore the challenge of accurately predicting the electromagnetic behavior of structures, beginning with simple coil-like devices and extending to more complex geometries. The objective is to evaluate the potential of these predictions in optimizing the design process for such devices.

2. Tools

To understand the extent of our work, two design and simulation tools are crucial to understand, *Magpylib* and *Magnet*.

2.1 Magpylib

Magpylib is a Python-based library designed for the simulation, analysis, and visualization of magnetic fields. It enables the modeling of magnetic sources such as coils and magnets, while providing tools for calculating magnetic fields.

Within the library, the module plays a central role, offering essential functions and utilities for field calculations and the modeling of standard geometric configurations. By integrating these capabilities, *magpylib* provides simulation results, making it useful for projects such as ours which requiring electromagnetic analysis.

2.2 Magnet

Magnet software is an advanced computational tool used for simulating and analyzing magnetic fields in complex systems. It employs the finite element method (FEM) to solve field equations with high precision. In a 2D model, the domain is discretized into a mesh composed of triangular elements, where the magnetic field within each element is approximated using a polynomial function. The accuracy of the solution can be enhanced by increasing the polynomial order, an adjustable option in the solver. Additionally, the accuracy can be further improved by refining the mesh in critical regions of the model, a process performed automatically when the user enables the adaptive meshing feature.

3. Roadmap

At the beginning of the project, we developed a comprehensive roadmap outlining the entire workflow and detailing the steps required to achieve our objectives. As part of our Capstone project, the team aimed to ensure that a functional algorithm was delivered at each milestone. The first step involved collecting analytical data using Magpylib.lib and utilizing it to implement an initial version of a Physics-Informed Neural Network (PINN). Building on this foundation, the second step focused on generating simulated data with *Magnet* and using it to further train the model, which at this stage predicts the magnetic behavior of a coil with an embedded steep cylinder. The final step is to scale the PINN to handle more complex geometries and predict their magnetic behavior accurately.

4. Progress (Work Accomplished)

The work that has been done throughout the project can be separated in two parts: Naive PINN based on Analytical Data (*Magpylib*) and PINN based on Simulated Data (*Magnet*).

4.1 Data generation

4.1.1 Magpylib

The *Magpylib* library was utilized to perform analytical computations related to magnetic field behavior, offering built-in functions to define magnetic sources and specify their geometry, dimensions (e.g., radius), current, and spatial coordinates. To optimize various queries required by the algorithm, a custom class named *MagneticSource* was developed. This class encapsulates all the essential attributes and functions needed to create and interact with magnetic sources. It includes two key helper functions: *add_source*(), which facilitates the creation of a magnetic source by specifying its dimensions (diameter and current) and spatial position, and *get_induced_B_field*(), which computes and returns the magnetic field vector at specified (x, y, z) positions. This implementation enhances the process of managing magnetic sources and retrieving relevant magnetic field information. This class can be found in section 2 of the code.

4.1.2 Magnet

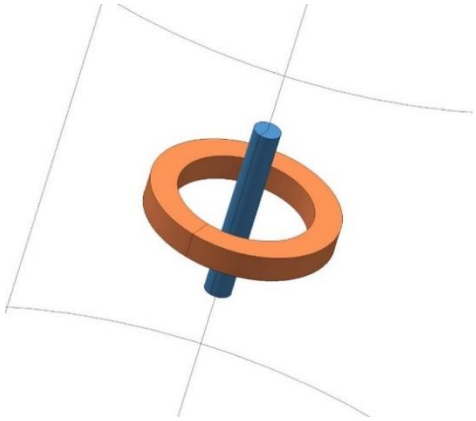

Figure 1. Model built in Magnet

For this project, Magnet was utilized to generate data for training and validating Physics-Informed Neural Network (PINN) predictions. The simulation process begins with the creation of a 3D model of the electromagnetic configuration as shown in Figure 1. After constructing the model, the solver was configured with precise parameters to ensure accurate and efficient simulations. All the parameters used during the data generation are compiled in Table 1.

Table 1. Data Generation Parameters

Parameters		Value	Unit
Coil	Inner radius	20	mm
	Outer radius	30	mm
	Thickness	10	mm
	Material	Copper	
	Turns	100	
	Current	100	mA

Rod	Radius	5	mm
	Length	70	mm
	Material	Cold Rolled 1010 Steel	
Boundary Condition	Shape	Cylindrical	
	Radius	170	mm
	Height	340	mm
Solver	Polynomial Order	3	
	Newton Tolerance	0.01	%
	CG tolerance	0.01	%
	Max Newton iterations	50	
	Source frequency	120	Hz

4.2 Algorithm explained

4.2.1 Requirements to run the code

To ensure the code runs correctly, the .csv files for the training and validation sets—namely, *trainingSet1_v2.csv*, *trainingSet2_v2.csv*, and *validationSet_v2.csv*—must be located in the directories specified in the code. These directories are stored as string variables named *trainingSet1_url*, *trainingSet2_url*, and *validationSet_url*. Each variable contains the path starting with '*/content/gdrive/MyDrive/*', which corresponds to the *MyDrive* directory in Google Drive. The related piece of code is shown below in Figure 2 and can be found in Section 6 of the code on our GitHub page.

```
# Get Training Set

trainingSet1_url = '/content/gdrive/MyDrive/trainingSet1_v2.csv'
trainingSet2_url ='/content/gdrive/MyDrive/trainingSet2_v2.csv'
validationSet_url = '/content/gdrive/MyDrive/validationSet_v2.csv'


ts1_df = pd.read_csv(trainingSet1_url,names = ['x','y','Hx','Hy'])
ts2_df = pd.read_csv(trainingSet2_url,names = ['x','y','Hx','Hy'])
training_set_df = pd.concat([ts1_df, ts2_df], ignore_index=True)
training_set_df['Hz'] = np.zeros(len(training_set_df))
training_set_df['z'] = np.zeros(len(training_set_df))

training_datapoints = training_set_df[['x','y','z']].to_numpy()
training_B =  training_set_df[['Hx','Hy','Hz']].to_numpy()*const.mu_0

# Get Validation Set
val_and_test_set_df = pd.read_csv(validationSet_url,names = ['x','y','Hx','Hy'])
val_and_test_set_df['Hz'] = np.zeros(len(val_and_test_set_df))
val_and_test_set_df['z'] = np.zeros(len(val_and_test_set_df))
validation_set_df = val_and_test_set_df.iloc[:len(val_and_test_set_df) // 2]

validation_datapoints = validation_set_df[['x','y','z']].to_numpy()
validation_B = validation_set_df[['Hx','Hy','Hz']].to_numpy()*const.mu_0


# Get Test Set
test_set_df = val_and_test_set_df.iloc[len(val_and_test_set_df) // 2:]
test_set_df = test_set_df.iloc[:].reset_index(drop=True)

test_datapoints = test_set_df[['x','y','z']].to_numpy()
test_B = test_set_df[['Hx','Hy','Hz']].to_numpy()*const.mu_0
```

Figure 2. Data collection (Magnet)

4.2.2 Magnetic Source

As outlined in Section 4.1.1, the dataset for our first model was generated using a Python library called *Magpylib*. This library provides built-in functions to create magnetic source objects, enabling users to define the shape of a magnetic source (in our case, a circular coil), its diameter, current intensity, and position in space (x, y, z). Once the source is created, the *getB* function can be used to calculate the magnetic flux density at specified points. This function takes a *Magpylib* source object (a coil in our case) and a set of (x, y, z) data points as input to compute the B-field.

To handle scenarios involving multiple magnetic sources in space, we developed a custom class called *MagneticSource*. This class encapsulates the essential attributes and functions required to create and interact with magnetic sources. When creating a *MagneticSource* object, users must provide three arguments: diameters, currents, and positions, all as lists—even if there is only one source. This design facilitates the simultaneous creation of multiple sources.

The class includes two custom methods: *get_induced_B_field*() and *add_source*(). The *get_induced_B_field*() method, similar to the *getB*() function in *Magpylib*, calculates the

B-field components (x, y, and z) while accounting for the presence of multiple sources in space through the principle of superposition. The *add_source*() method allows additional magnetic sources to be added to an existing *MagneticSource* object. This design enables the class to manage multiple underlying magnetic sources and treat them collectively as a single equivalent source. This class can be found in Section 2 of the code.

4.2.3 Neural Network Structure

To enable customization of the neural network's architecture, we created a class named *MagneticFieldNN*. This class allows users to define the network's structure by specifying parameters such as the input size, the hidden layer size, the output size, and the number of layers when instantiating an object. The implementation details for this class can be found in Section 3 of the code.

4.2.4 Model Implementation

We developed a class designed to manage the training and predictions of the Physics-Informed Neural Network (PINN). The attributes of the arguments passed when creating an instance of this class are detailed in Table 2 . This class supports the training of two models:

1. **The first model**, which uses *Magpylib* for generating datasets. In this case, a coil object and the coordinate points (x, y, z) for training (training points) are provided as arguments. The B-fields for these points are obtained using the *coil.get_induced_B_field*() method. The same approach applies to the test points. Since *Magpylib* is used to generate B-fields, the arguments *training_B* and *test_B* are left as empty arrays.

2. **The second model**, which uses Magnet to generate training and test datasets. Here, the coil argument is set to None, and the *training_B* and *test_B* arguments contain the B-field data read from the CSV files, as discussed in Section 4.2.1. All other arguments (*model*, *criterion*, *optimizer*, *training_points*, *num_epochs*, *lambda_divergence*, and *testPoints*) remain the same for both models.

The training process is initiated by calling the *fit*() method, which executes a training loop involving forward passes, loss calculations, and backward passes. While the loop resembles a standard training routine, our loss function incorporates a divergence penalty to account for the error on the physics of the model (Gauss's Magnetic Law). This penalty is computed using the gradient of each B-field component (Bx, By, Bz), calculated with a custom method called *compute_divergence*(). The gradient is determined using finite differences—forward and backward differences for edge points and central differences for interior points.

To address the issue of exploding gradients, which can result in Not-a-Number (NaN) values, we implemented *compute_divergence_NAN_handling*(). This function removes problematic points when NaN values are detected and is called automatically by *compute_divergence*() in such cases. The penalty is calculated as the mean absolute divergence. This value is then scaled by a lambda factor. The total loss is a combination of the data error and the divergence penalty.

Once the training is complete, the *predict*() method is used for model predictions during testing. The algorithm can be found in section 4 of the case. The instantiation details for the first and second models can be found in Sections 5 and 7 of the code, respectively.

Table 2. Attributes of the class MagneticFieldNN_SIM

Attributes	Type	Expected Value	Description
Coil	MagneticSource	Instance of MagneticSource if data was generated with magpylib else None	Magpylib coil (Circle) object
Model	MagneticFieldNN	Instance of MagneticFieldNN	Object containing the model structure
Criterion	nn.MSELoss()	Reference to the loss function	Loss function used during training
Optimizer	optim.Adam()	Instance of adam optimizer	Type of optimizer used during training
Training_points	[[ ] , [ ] , [ ]]	A 2-dimension array with 3 elements	Array of 3D data points (x y, z) used for training
Num_epoch	Int	Any positive integer	Max. number of epochs during training
Lambda_divergence	Float/Double	Any positive floating point number	Divergence term that specifies the weight the divergence error has on the total error
TestPoints	[[ ] , [ ] , [ ]]	A 2-dimension array with 3 elements	Array of 3D data points (x y, z) used for testing
Training_B	[[ ] , [ ] , [ ]]	[ ] if data was generated with magpylib else the list contains the corresponding B-fields	Corresponding training B-field components (Bx, By, Bz)
Test_B	[[ ] , [ ] , [ ]]	[ ] if data was generated with magpylib else the list contains the corresponding B-fields	Corresponding testing B-field

4.2.5 Accuracy

To evaluate the accuracy of our model, we utilized the *Mean Absolute Percentage Error*. As the sample size is in the order of thousands, we computed the proportions of predictions under 5%, 10%, 15%, 20% and 25% error. The piece of code that implements the accuracy measurements for Model 1 can be found at the end of Section 5 in the code and at the end of Section 7 for Model 2.

4.2.6 Cross validation

This function, *cross_validate_model*, performs k-fold cross-validation on a model, specifically the *MagneticFieldNN_SIM* model. It takes several parameters, including the

number of folds (*num_folds*), the model class, the training data points, the coil object for magnetic field data, the loss criterion, the optimizer class, the number of epochs, the divergence penalty factor, and the optional training/validation labels. The function begins by splitting the training data into *num_folds* subsets using KFold, then iterates over each fold. For each fold, it generates training and validation sets, initializes the model and optimizer, and trains the model on the training data. After training, it evaluates the model on the validation set by calculating the Mean Absolute Percentage Error (MAPE) between predicted and actual values. The MAPE scores for each fold are stored and averaged to provide an overall performance metric for the model across all folds. This process helps evaluate the model's generalization ability by testing it on multiple, different splits of the data.

5. Analysis of the model

The primary issue with our model at the moment is its inability to accurately recover the initial magnetic field magnitudes after normalizing them. To address this, our team attempted to develop another ANN solely to predict the magnitude. However, this approach has been proven ineffective as it was significantly reducing accuracy to around 5-10%. Thus, the field magnitudes were not considered in our model.

Additionally, the cross-validation function is only implemented for Model 1, limiting its applicability to other models.

Furthermore, the optimal number of folds for cross-validation has not yet been determined, leaving room for improvement in model evaluation.

Moreover, the input parameters currently only account for fixed parameters, as shown in **Error! Reference source not found.** and **Error! Reference source not found.**, and do not consider characteristics of the magnetic source, such as radius, dimension, and current as part of the input layer.

Also, the models currently only incorporate Gauss's law for magnetism, and extending the models to include other Maxwell's equations could enhance their performance.

Finally, as noted in Section 4.2.2, the hyperparameters for Model 2 have not been optimized, which should be prioritized to improve its accuracy and generalization.

On the positive side, one of the strengths of the models is their speed as the processing time for a sample size of 100 000 for model 1(10 000 for model 2) is around 2 minutes (30 sec). In addition, the algorithm for model 1 is easily extendable for more than one magnetic source.

6. Vision for the future

The vision for the future of this project is to develop a highly accurate and versatile Physics-Informed Neural Network (PINN) capable of modeling complex electromagnetic field interactions in real-world scenarios. Future teams working on the project will have the opportunity to build on the foundational work by refining the model to ensure precise predictions across various coil geometries and configurations, while integrating diverse material properties and electromagnetic environments.

6.1. Expanding the Model with Additional Input Parameters

A key area of growth will be the incorporation of additional input parameters to account for a wider range of real-world conditions. These could include geometric characteristics

such as coil radius, length, and positioning of metallic structures, as well as advanced material properties like conductivity, permeability, and temperature-dependent effects. Future teams will need to perform sensitivity analysis and optimization to prioritize parameters that significantly influence the model's predictions, ensuring both computational efficiency and accurate representation of physical phenomena.

6.2. Repository Maintenance and Collaboration

The repository we made on GitHub will be a central hub for future teams. It houses all versions of the PINN and provide detailed documentation for further development. Future teams will be responsible for ensuring that the repository remains well-maintained. By adhering to best practices and maintaining clear documentation, the GitHub page will foster a collaborative and efficient development process.

6.3. Promoting Open Collaboration

Future teams will be encouraged to document their findings, modifications, and improvements in the repository. They will contribute new reports that demonstrate the effectiveness of the model in different scenarios. This will ensure the project advances the state of electromagnetic modeling through machine learning and other innovative techniques. Also, it will allow other teams to continue the project and push its boundaries forward.

7. References

[1] G. Turnbull, “Maxwell’s Equations”, Vol. 101, No. 7, July 2013. [Online]

Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6532357

[Accessed Sep. 18, 2024]

[2] Y. Kumar, K. Kaur, G. Singh, “Machine Learning aspects and its applications towards different research areas”, ICCAKM, Amity University, 2020. [Online]

Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9051502

[Accessed Sep. 18, 2024]

[3] Michael Ortner, Lucas Gabriel Coliado Bandeira, “Magpylib Documentation”, magpylib.readthedocs.io, 2018 [Online]

Available:

https://magpylib.readthedocs.io/en/latest/_pages/user_guide/guide_index.html

[Accessed Nov. 7, 2024]

8. Appendices

i. Model Algorithm

```
# Class to create a magnetic source (coil)
# Diameters, currents and positions can be passed as single arguments or arrays
# If passed as arrays, equivalent of several sources
# the three arguments must have the same dimensions

class MagneticSource():
  def __init__(self,diameters, currents, positions):
    self.source = []
    try:
      iter(diameters)
      iter(currents)
      iter(positions)

      for cur, dia, pos in zip(currents, diameters, positions):
        self.source.append(magpy.current.Circle(diameter=dia, current=cur, position=pos))

    except TypeError:
      self.source = magpy.current.Circle(diameter=diameters, current=currents,
position=positions)


  # get the induced magnetic field at a specific point
  def get_induced_B_field(self,points):
    try:
      iter(self.source)
      B = 0
      for coil in self.source:
        if (isinstance(coil, magpy._src.obj_classes.class_current_Circle.Circle)):
          B = B + magpy.getB(coil,points)
      return B

    except TypeError:

      return magpy.getB(self.source,points)


  def add_source(self, dia, cur, pos):
    self.source.append(magpy.current.Circle(diameter=dia, current=cur, position=pos))


# This class can now specify the number of hidden layer in the neural network instead of
manually creating them
# Attributes:
# input size -> int: size of the input layer
# hidden_size -> int: size of the hidden layers
# output_size -> int: size of the output layer
# numOfLayers -> int: overall number of layers

class MagneticFieldNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, numOfLayers):
        super(MagneticFieldNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.layers = nn.ModuleList()  # Create a ModuleList to store layers

        # Add input layer
        self.layers.append(nn.Linear(input_size, hidden_size,bias=True))
        self.layers.append(nn.ReLU())

```

```
59.
60.         # Add hidden layers
61.         for _ in range(numOfLayers - 1):  # Subtract 1 for the input layer
62.             self.layers.append(nn.Linear(hidden_size, hidden_size,bias=True))
63.             self.layers.append(nn.ReLU())
64.
65.         # Add output layer
66.         self.layers.append(nn.Linear(hidden_size, output_size,bias=True))
67.
68.     def forward(self, x):
69.         for layer in self.layers:
70.             x = layer(x)
71.         return x
72.
73. # Class that implements the model (neural network)
74. # The class trains the model and makes predictions
75. # Attributes:
76. # Coil: magpylib coil (circle) object ->  None if data was not generated by magpylib
77. # Model: MagneticFieldNN object that represents a neural network
78. # Criterion: Type of loss function
79. # Optimizer: Type of optimizer
80. # Training_points: Array of 3D  data points (x, y, z) used for training
81. # Num_epoch: Maximal number of epoch during training
82. # Lambda_divergence:divergence term that specifies the weight the divergence error has on the
total error
83. # TestPoints: Array of 3D  data points (x, y, z) used for testing
84. # Training_B: Corresponding training B-field -> [] if data was generated by magpylib
85. # Test_B: Corresponding testing B-field -> [] if data was generated by magpylib
86.
87. # Functions:
88. # compute_divergence(predicted_B, grid_points): Compute the divergence of B
89. # predicted_B: argument -> 3D array containing the (x,y,z) components of B (Bx,By,Bz)
90. # grid_points: argument -> 3D array containing the corresponding (x,y,z) components
91.
92.
93. # compute_divergence_NAN_handling(predicted_B, grid_points): Similar to compute_divergence()
but handles the NAN exception
94.
95. # fit(): trains the model
96.
97. # predict(): makes predictions
98.
99. # get_params(): returns model parameters
100.
101. # set_params(): set model's parameters
102. class MagneticFieldNN_SIM(BaseEstimator, RegressorMixin):
103.
104.   def __init__(self, coil, model, criterion, optimizer, training_points, num_epochs,
lambda_divergence,testPoints,training_B=[],test_B=[]):
105.
106.     self.training_points = training_points
107.     fieldVector = []
108.     if coil is None:
109.
110.       fieldVector = training_B
111.
112.     else:
113.
114.       fieldVector = coil.get_induced_B_field(training_points)
115.
116.     norms = np.linalg.norm(fieldVector, axis=1)
117.     fieldDirection = fieldVector/norms[:, np.newaxis]
118.
119.     self.model = model
120.     self.criterion = criterion
```

```
    self.optimizer = optimizer
    self.num_epochs = num_epochs
    self.lambda_divergence = lambda_divergence
    self.Xs = torch.tensor(training_points, dtype=torch.float32)
    self.Ys = torch.tensor(fieldDirection, dtype=torch.float32)
    self.coil = coil


    self.testPoints = testPoints

    testVector = []

    if coil is None:

      testVector = test_B

    else:

      testVector = coil.get_induced_B_field(testPoints)


    testNorms = np.linalg.norm(testVector, axis=1)
    self.testExpected = testVector/testNorms[:, np.newaxis]
    self.testXs = torch.tensor(testPoints, dtype=torch.float32)
    self.testYs = torch.tensor(self.testExpected, dtype=torch.float32)
    self.testPredictions = []


  def compute_divergence(self, predicted_B, grid_points):
    """Compute the divergence of the magnetic field."""
    B_x, B_y, B_z = predicted_B[:, 0], predicted_B[:, 1], predicted_B[:, 2]


    # Compute partial derivatives using finite differences
    dB_x_dx = np.gradient(B_x, grid_points[:, 0])
    dB_y_dy = np.gradient(B_y, grid_points[:, 1])
    dB_z_dz = 0



    if self.coil is not None:
      dB_z_dz = np.gradient(B_z, grid_points[:, 2])



    divergence = dB_x_dx + dB_y_dy + dB_z_dz

    num_nan = np.isnan(divergence).sum()

    if(num_nan>0):

      divergence = self.compute_divergence_NAN_handling(predicted_B,grid_points)


    return np.mean(np.abs(divergence))


  def compute_divergence_NAN_handling(self,predicted_B, grid_points):
    """
    Compute the divergence of the magnetic field with improved NaN handling.
    """
    # Separate components of the magnetic field
    B_x, B_y, B_z = predicted_B[:, 0], predicted_B[:, 1], predicted_B[:, 2]

    # Compute partial derivatives using finite differences
```

```
    dB_x_dx = np.gradient(B_x, grid_points[:, 0])
    dB_y_dy = np.gradient(B_y, grid_points[:, 1])
    dB_z_dz = 0


    # Compute divergence
    divergence = dB_x_dx + dB_y_dy + dB_z_dz

    cleared_divergence = divergence[~np.isnan(divergence)]
    cleared_divergence = cleared_divergence[np.isfinite(cleared_divergence)]

    return np.mean(np.abs(cleared_divergence))




  def fit(self):

    self.model.train() # Set the model in training mode
    for epoch in range(self.num_epochs):
      # Forward pass
      outputs = self.model(self.Xs)

      # Compute divergence penalty
      divergence_penalty = self.compute_divergence(outputs.detach().numpy(),
self.training_points)

      # Compute loss
      loss = self.criterion(outputs, self.Ys) + self.lambda_divergence * divergence_penalty

      if (loss <= 0.0015):
        break

      # Backward pass and optimization
      self.optimizer.zero_grad()
      loss.backward()
      self.optimizer.step()

      # Print loss for monitoring training progress
      print(f'Epoch [{epoch+1}/{self.num_epochs}], Loss: {loss.item():.4f}')

      # Print loss for monitoring training progress
      print(f'Epoch [{epoch+1}/{self.num_epochs}], Loss: {loss.item():.4f}')


  def predict(self):
    # Test set
    self.model.eval()
    with torch.no_grad():
      self.testPredictions = self.model(self.testXs)


  def get_params(self,deep=True):

    return {

            'input_size': self.model.input_size,
            'hidden_size': self.model.hidden_size,
            'output_size': self.model.output_size,
            'learning_rate': self.optimizer.param_groups[0]['lr'],
            'epochs': self.num_epochs

    }

```

```
250.
251.   def set_params(self, **params):
252.         # Set the model's parameters (hyperparameters)
253.         for param, value in params.items():
254.             setattr(self, param, value)
255.         return self
256.
257. # Model with data simulated using maglib.py (python)
258.
259. # coil
260. current = 0.1
261. coil_diameter = 0.25
262. coil = MagneticSource(current,coil_diameter,(0,0,0))
263.
264.
265. # model
266. input_size = 3
267. hidden_size = 30
268. output_size = 3
269. numOfLayers = 4
270. model = MagneticFieldNN(input_size,hidden_size,output_size,numOfLayers)
271.
272.
273. # Criterion
274. criterion = nn.MSELoss()
275.
276.
277. # Optimizer
278. learning_rate = 0.00985
279. optimizer = optim.Adam(model.parameters(), lr=learning_rate)
280.
281. # Training set
282. np.random.seed(26)
283. numberOfSamples = 100000
284. training_points = np.random.rand(numberOfSamples, 3) * 2 - 1
285.
286. # Validation Set
287. np.random.seed(21)
288. numberOfSamples = 10000
289. validation_points = np.random.rand(numberOfSamples, 3) * 2 - 1
290.
291. # Test Set
292. numberOfTests = 10000
293. np.random.seed(27)
294. testPoints = np.random.rand(numberOfTests, 3) * 2 - 1
295.
296.
297. # Training
298. num_epochs = 850
299. #num_epochs = 1
300. lambda_divergence = 0.0015
301.
302. PINN_SIM_DATA = MagneticFieldNN_SIM(coil, model, criterion, optimizer, training_points,
num_epochs, lambda_divergence,testPoints)
303. PINN_SIM_DATA.fit()
304.
305.
306. # Test set
307. testExpected = PINN_SIM_DATA.testExpected
308. testXs = PINN_SIM_DATA.testXs
309. testYs = PINN_SIM_DATA.testYs
310.
311.
312. # Prediction
313. PINN_SIM_DATA.predict()
```

```
testPredictions = PINN_SIM_DATA.testPredictions


# Test accuracy of model 1
meanTestYs = torch.mean(testYs)

tss = torch.sum((testYs - meanTestYs)**2)


rss = torch.sum((testYs - testPredictions)**2)


rSquared = 1 - (rss / tss)
absolutePercentageError = np.abs((testExpected - testPredictions.numpy()) / testExpected) * 100
absolutePercentageError[np.isnan(absolutePercentageError)] = 0
mape = np.mean(absolutePercentageError, axis=1)

errorPercentage = np.arange(5, 26, 5)
NbUnderPercentage = np.array([np.sum(mape <= x) for x in errorPercentage])

#print(f'mape: {mape}')
plt.plot(errorPercentage, NbUnderPercentage/numberOfTests * 100)
plt.xticks(errorPercentage)
plt.yticks(np.arange(20, 91, 5))

plt.xlabel('Mean Absolute Percentage Error [%]')
plt.ylabel('Proportion of predictions [%]')
plt.title('Proportion of Predictions under Mean absolute Percentage Error for Model 1 ')

plt.grid(True)
plt.show()

# Get Training Set

trainingSet1_url = '/content/gdrive/MyDrive/trainingSet1_v2.csv'
trainingSet2_url ='/content/gdrive/MyDrive/trainingSet2_v2.csv'
validationSet_url = '/content/gdrive/MyDrive/validationSet_v2.csv'


ts1_df = pd.read_csv(trainingSet1_url,names = ['x','y','Hx','Hy'])
ts2_df = pd.read_csv(trainingSet2_url,names = ['x','y','Hx','Hy'])
training_set_df = pd.concat([ts1_df, ts2_df], ignore_index=True)
training_set_df['Hz'] = np.zeros(len(training_set_df))
training_set_df['z'] = np.zeros(len(training_set_df))

training_datapoints = training_set_df[['x','y','z']].to_numpy()
training_B =  training_set_df[['Hx','Hy','Hz']].to_numpy()*const.mu_0

# Get Validation Set
val_and_test_set_df = pd.read_csv(validationSet_url,names = ['x','y','Hx','Hy'])
val_and_test_set_df['Hz'] = np.zeros(len(val_and_test_set_df))
val_and_test_set_df['z'] = np.zeros(len(val_and_test_set_df))
validation_set_df = val_and_test_set_df.iloc[:len(val_and_test_set_df) // 2]

validation_datapoints = validation_set_df[['x','y','z']].to_numpy()
validation_B = validation_set_df[['Hx','Hy','Hz']].to_numpy()*const.mu_0


# Get Test Set
test_set_df = val_and_test_set_df.iloc[len(val_and_test_set_df) // 2:]
test_set_df = test_set_df.iloc[:].reset_index(drop=True)

test_datapoints = test_set_df[['x','y','z']].to_numpy()
test_B = test_set_df[['Hx','Hy','Hz']].to_numpy()*const.mu_0

```

```
379.
380. # Model with data simulated using Magnet
381.
382. # model
383. input_size = 3
384. hidden_size = 30
385. output_size = 3
386. numOfLayers = 4
387. model = MagneticFieldNN(input_size,hidden_size,output_size,numOfLayers)
388.
389. # Criterion
390. criterion = nn.MSELoss()
391. # Optimizer
392. learning_rate = 0.00985
393. optimizer = optim.Adam(model.parameters(), lr=learning_rate)
394.
395. num_epochs = 850
396.
397. lambda_divergence = 0.0015
398. # Training
399. PINN_MAGNET_DATA = MagneticFieldNN_SIM(None,model, criterion,
optimizer,training_datapoints,num_epochs,lambda_divergence,test_datapoints,training_B, test_B)
400. PINN_MAGNET_DATA.fit()
401.
402. # Test Set
403. testExpected = PINN_MAGNET_DATA.testExpected
404. testXs = PINN_MAGNET_DATA.testXs
405. numberOfTests = len(testXs)
406.
407. testYs = PINN_MAGNET_DATA.testYs
408.
409. # Prediction
410. PINN_MAGNET_DATA.predict()
411. testPredictions = PINN_MAGNET_DATA.testPredictions
412. testPredictions[:,-1] = 0
413.
414. # Test accuracy (model 2)
415. meanTestYs = torch.mean(testYs)
416.
417. tss = torch.sum((testYs - meanTestYs)**2)
418.
419.
420. rss = torch.sum((testYs - testPredictions)**2)
421.
422.
423. rSquared = 1 - (rss / tss)
424. absolutePercentageError = np.abs((testExpected - testPredictions.numpy()) / testExpected) * 100
425. absolutePercentageError[np.isnan(absolutePercentageError)] = 0
426. mape = np.mean(absolutePercentageError, axis=1)
427.
428. errorPercentage = np.arange(5, 26, 5)
429. NbUnderPercentage = np.array([np.sum(mape <= x) for x in errorPercentage])
430.
431. #print(f'mape: {mape}')
432. plt.plot(errorPercentage, NbUnderPercentage/numberOfTests * 100)
433. plt.xticks(errorPercentage)
434. plt.yticks(np.arange(20, 91, 5))
435.
436. plt.xlabel('Mean Absolute Percentage Error [%]')
437. plt.ylabel('Proportion of predictions [%]')
438. plt.title('Proportion of Predictions under Mean absolute Percentage Error for Model 2')
439.
440. plt.grid(True)
441.
442. plt.show()
```

```
443.
444.
445. # Cross-validation function
446. def cross_validate_model(num_folds, model_class, training_points, coil, criterion,
optimizer_cls, num_epochs, lambda_divergence,y_train=[],y_val=[]):
447.     """
448.     Perform k-fold cross-validation on the MagneticFieldNN_SIM model.
449.     """
450.     kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)
451.     scores = []
452.
453.     for fold, (train_idx, val_idx) in enumerate(kf.split(training_points)):
454.         # Split data into training and validation sets for this fold
455.         X_train, X_val = training_points[train_idx], training_points[val_idx]
456.
457.         if(coil is not None):
458.           # Generate training and validation magnetic fields
459.           y_train = coil.get_induced_B_field(X_train)
460.           y_val = coil.get_induced_B_field(X_val)
461.
462.         # Initialize model and optimizer for this fold
463.         model = MagneticFieldNN(input_size=3, hidden_size=30, output_size=3, numOfLayers=4)
464.         optimizer = optimizer_cls(model.parameters(), lr=learning_rate)
465.
466.         # Create MagneticFieldNN_SIM instance for this fold
467.         fold_model = model_class(
468.             coil=coil,
469.             model=model,
470.             criterion=criterion,
471.             optimizer=optimizer,
472.             training_points=X_train,
473.             num_epochs=num_epochs,
474.             lambda_divergence=lambda_divergence,
475.             testPoints=X_val
476.         )
477.
478.         # Train model on this fold's training data
479.         fold_model.fit()
480.
481.         # Predict on validation set
482.         fold_model.predict()
483.         predictions = fold_model.testPredictions.detach().numpy()
484.         y_val_actual = y_val / np.linalg.norm(y_val, axis=1)[:, np.newaxis]  # Normalize
expected values
485.
486.         # Calculate MAPE for this fold
487.         fold_score = mean_absolute_percentage_error(y_val_actual, predictions)
488.         scores.append(fold_score)
489.         print(f"Fold {fold + 1} - MAPE: {fold_score:.4f}")
490.
491.     # Average score across all folds
492.     avg_score = np.mean(scores)
493.     print(f"Average MAPE across {num_folds} folds: {avg_score:.4f}")
494.     return avg_score
495.
496.
497. # Parameters for cross-validation
498. num_folds = 5
499. learning_rate = 0.00985
500. num_epochs = 850
501. lambda_divergence = 0.0015
502.
503.
504. # Perform cross-validation for model with data from maglib.py
505. average_mape = cross_validate_model(
```

```
    num_folds=num_folds,
    model_class=MagneticFieldNN_SIM,
    training_points=validation_points,
    coil=coil,
    criterion=criterion,
    optimizer_cls=optim.Adam,
    num_epochs=num_epochs,
    lambda_divergence=lambda_divergence
)

print(f"Final Average MAPE from Cross-Validation: {average_mape:.4f} for Model 1")

```