UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria Industriale DII

Corso di Laurea Magistrale in Ingegneria dell'Energia Elettrica

Physics-informed neural networks: potential and limitations of a new approach to computational electromagnetism

Relatore: prof. Federico Moro

Laureando

Tomà Sartori – 2022779

Anno Accademico 2022/2023

Sommario

Questa tesi si propone di analizzare le possibili applicazioni delle reti neurali artificiali, in particolare delle cosiddette "Physics-informed neural networks" (PINN), un tipo di reti neurali artificiali che sfrutta la conoscenza delle leggi fisiche per fronteggiare la mancanza di dati e accelerare il processo di "training", nell'ambito dell'elettromagnetismo computazionale, sviluppando un codice che utilizzi le PINN per risolvere problemi elettromagnetici, in particolare problemi elettrostatici 1D e 2D. Viene proposto un esempio, preso dal sito di *Mathworks*®, nel quale si risolve l'equazione di Poisson in un dominio di forma circolare, impiegando una rete neurale artificiale. A partire dal codice visto nell'esempio, inizia poi il lavoro di sviluppo di un algoritmo MATLAB che sia in grado di risolvere il problema elettrostatico del condensatore piano, in varie configurazioni: 1D con permittività omogenea, 1D con permittività non omogenea, 2D con permittività non omogenea. I risultati sono poi confrontati con le soluzioni degli stessi problemi ottenute per via analitica o tramite un codice FEM. Il risultato finale è un codice MATLAB utilizzabile per risolvere il problema elettrostatico del condensatore piano nelle configurazioni sopra elencate e che fornisce soluzioni la cui accuratezza è comparabile a quella delle soluzioni ottenute tramite il metodo degli elementi finiti. Sono inoltre analizzate le limitazioni che caratterizzano questo tipo di algoritmi e che ne ostacolano l'applicazione a problemi più complessi.

Abstract

The goal of this thesis is to analyze the possible applications of artificial neural networks, in particular the so-called "Physics-informed neural networks" (PINN), a type of artificial neural networks that exploits the knowledge of the physical laws to overcome the lack of data and speed up the training process, in the field of computational electromagnetism, developing a code that uses PINNs to solve electromagnetic problems described by partial differential equations, in particular 1D and 2D electrostatic problems. An example, taken from the *Mathworks*® website, is proposed. In this example a PINN is used to solve the Poisson's problem on a unit disk. From this example, we start the development of a MATLAB algorithm that is able to solve the electrostatic problem of the parallel plate capacitor, in various configurations: 1D with homogeneous permittivity, 1D with non-homogeneous permittivity, 2D with non-homogeneous permittivity. The results are then compared to the analytical or FEM solutions of the same problems. The final result is a MATLAB code that can be used to solve the parallel-plate capacitor problem in the aforementioned configurations and provides solutions with accuracy comparable to those obtained via the finite element method. The limitations that characterize this type of algorithms and limit their application to more complex problems are also analyzed.

Contents

List of Figures

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
DL	**D**eep **L**earning
ANN	**A**rtificial **N**eural **N**etwork
PINN	**P**hysics-**I**nformed **N**eural **N**etwork
AD	**A**utomatic **D**ifferentiation
MSE	**M**ean **S**quared **E**rror
FEM	**F**inite **E**lement **M**ethod
BVP	**B**oundary **V**alue **P**roblem
IBVP	**I**nitial-**B**oundary **V**alue **P**roblem

Chapter 1

An introduction to artificial neural networks

1.1 What is an artificial neural network

Artificial Intelligence (AI) has been drawing the attention of the scientific community in the past few decades, and has seen a quite steep increase in consideration in these last years. As its capabilities are being unveiled, more and more research is focusing on AI, even in sectors that, so far, have seemed indifferent to it, such as computational engineering [1].

Among all the different AI branches, one that has received particular regard in computational engineering is Machine Learning (ML). Machine Learning is a broad term that refers to the ability of a machine to learn and imitate behaviors that are peculiar to humans. In particular, it aims at creating machines that are able to do so "without being explicitly programmed" (Samuel, n.d., as cited in [1]). ML is based on data, and ML algorithms use data to train themselves in order to perform tasks so complex that developing a *conventional* algorithm to perform these same tasks would be almost impossible or would take too much time and effort.

Artificial neural networks (ANNs) are part of the broader family of ML algorithms. More specifically, ANNs can be classified as Deep Learning (DL) algorithms. The term Deep Learning refers to algorithms that use multiple layers (hence the adjective "deep") of nonlinear processing units to learn representations of data [2] and to extract features from them [3]. The idea behind ANNs is pretty simple. They replicate the neural connections of a human brain in order to learn from data and perform tasks accordingly. ANNs undergo a training process, done on a set of input data, named *training dataset*, that aims at minimizing the discrepancy between the target output (which is known for the training dataset) and the actual network output, by varying the network parameters. Once the network has been successfully trained, it can be fed a different set of input data, with unknown output, and, without the need to readjust its parameters, it can perform the required tasks. This means that training the network on a finite set of data makes it possible to process a potentially infinite number of input datasets.

We will get into more details about ANNs structure and training later, but we can already grasp the potentialities of such a kind of algorithms. It is easy to understand why their use is so popular in science, medicine and engineering [4]. Among ANNs applications, we find feature extraction, feature reduction, classification problems [5], function approximation, data processing, filtering, clustering, compression, robotics, regulations, decision making [6] and many more.

1.2 ANNs structure

An artificial neural network is composed by nodes, each node representing the artificial counterpart of a neuron, and nodes are connected to each other. Connections (edges) between nodes are intended to artificially replicate the brain synapses. Each node receives signals (real numbers) from other neurons as inputs, and its output is a non-linear function of the sum of its inputs, called *activation function*, which can be for example a hard limiter or some kind of sigmoid function. Each connection to a node has a weight, meaning that each node performs a weighted sum of its inputs. A bias might also be added to the sum of the inputs. The weights and biases represent the adjustable parameters of the network. A node (or neuron) of a network can be mathematically represented as a function:

$$Y = \theta(\sum_{i=1}^{n} W_i X_i + b), \tag{1.1}$$

where θ is the activation function, W_i are the weights and b is the bias [5].

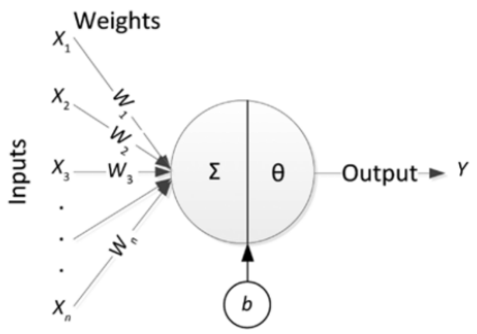


FIGURE 1.1: Graphic representation of a neuron [5]

The nodes of an ANN are organized in layers. There is usually an input layer, which takes the input data (for example, some features of sample objects) and feeds them to the rest of the network, and an output layer, which performs the required task (for example, cataloguing the objects according to certain criteria). In between them there can be one or more layers, called *hidden* layers, whose nodes receive their inputs from the nodes connected to them in the previous layer, and send their outputs to the nodes connected to them in the next one. Networks organized in this way are called *feedforward* networks. If the neurons in one layer are all connected to every neuron in the next layer, the feedforward network is called *fully connected*. Conversely, if the some nodes in a layer also send their output to other nodes in the same layer or in previous layers, the network is called *recurrent*. If a layer has N neurons, whose outputs are described by (1.1), the mathematical representation of the output of the layer is:

$$\boldsymbol{Y} = \begin{bmatrix} Y_1 \\ \vdots \\ Y_i \\ \vdots \\ Y_N \end{bmatrix} = \begin{bmatrix} \theta(\sum_{j=1}^{M} W_{1j} X_j + b_1) \\ \vdots \\ \theta(\sum_{j=1}^{M} W_{ij} X_j + b_i) \\ \vdots \\ \theta(\sum_{j=1}^{M} W_{Nj} X_j + b_N) \end{bmatrix} \tag{1.2}$$

If we represent the weights $W_{i,j}$ as a matrix

$$\boldsymbol{W} = \begin{bmatrix} W_{11} & \cdots & W_{1M} \\ \vdots & \vdots & \vdots \\ W_{N1} & \cdots & W_{NM} \end{bmatrix}, \tag{1.3}$$

we can write (1.2) in matrix form:

$$\boldsymbol{Y} = \theta(\boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}), \tag{1.4}$$

where $\boldsymbol{X}$ is the vector of the outputs of the previous layer and $\boldsymbol{b}$ is the vector of the biases. If we take a look at the network in Figure 1.2, its output, according to (1.4) can be mathematically formulated as:

$$\begin{aligned}\boldsymbol{Y}^3 &= \theta(\boldsymbol{W}^3\boldsymbol{Y}^2 + \boldsymbol{b}^3)\\ &= \theta(\boldsymbol{W}^3\theta(\boldsymbol{W}^2\boldsymbol{Y}^1 + \boldsymbol{b}^2) + \boldsymbol{b}^3)\\ &= \theta(\boldsymbol{W}^3\theta(\boldsymbol{W}^2\theta(\boldsymbol{W}^1\boldsymbol{X} + \boldsymbol{b}^1) + \boldsymbol{b}^2) + \boldsymbol{b}^3).\end{aligned} \tag{1.5}$$

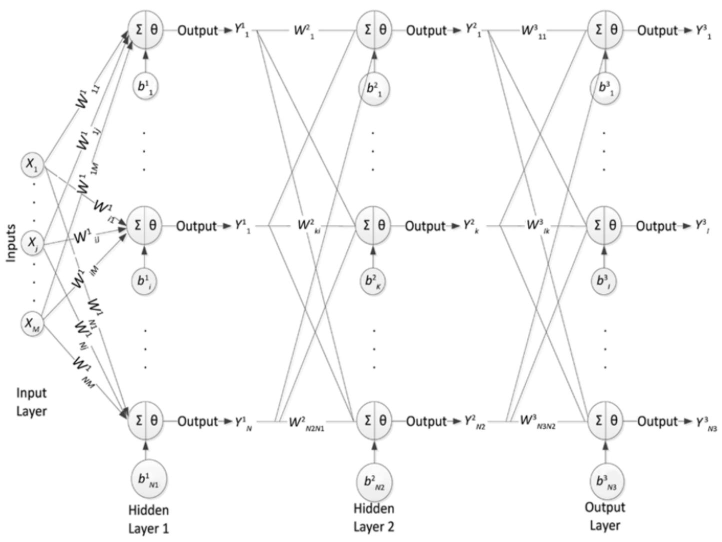


FIGURE 1.2: Graphic representation of a three-layer ANN [5]

1.3 Network training

Once the network structure has been defined, we need to adjust the weights and biases of the neurons in order to achieve satisfactory performances of the network. This is done by training the network using a training dataset. If the network has M neurons in the input layer and N neurons in the output layer, the training dataset will be a set of P pairs *(I, O)* where $\boldsymbol{I}$ is an M-dimensional vector representing the input data and $\boldsymbol{O}$ is an N-dimensional vector representing the expected output. The training problem is then reduced to an optimization problem. If $\boldsymbol{Y}$ is the vector of the neural network outputs generated by input $\boldsymbol{I}$, we can define the cost function as:

$$C = \frac{1}{P}\sum_{i=1}^{P} \|Y_i - O_i\|^2. \tag{1.6}$$

As (1.1) shows, the output of the network, which is a combination of the outputs of the single neurons, is a function of the network parameters. This means that also C is a function of network parameters. In order to achieve the desired network output, we need to find a set of these parameters (weights and biases) that minimizes the function C. This task can be handled by several different optimization algorithms, usually gradient-based, such as backpropagation or Adam, but which will be discussed in detail in Section 2.4. Once the optimal set of parameters has been found, the network is ready to perform the tasks required on different sets of input data. This type of training approach is called *supervised training* [6]. Other types of approach can be used, such as *unsupervised training* or *reinforcement training*, which, however, will not be discussed in this thesis.

1.4 Application of ANNs to computational electromagnetics and limitations of the data-driven approach

ANNs can be used as an alternative to FEM or other conventional numerical methods to solve EM boundary value problems. Conventional methods are typically fast and efficient when solving field problems, but in the case of large domains, nonlinear media and time dependent problems may lead to computationally expensive solutions [7]. Moreover, solving differential equations with ANNs offers a big advantage over conventional solvers. Since the solution obtained using ANNs is a differentiable analytic function (see (1.5)), it will be easy to use it subsequent computations [8].

An artificial neural network can be trained using space and time coordinates as inputs [9], and the respective known potential values as expected outputs in the cost function. Once the network is trained, it can be used to compute the potential in the whole space/time domain by feeding it the coordinates and evaluating the corresponding output. This is another great advantage over FEM and other computational tools, because time-dependent problems can be dealt without the need for a time-stepping scheme [10].

This approach, however, requires the knowledge of a large number of *input-output* pairs, something that is usually not available in the case of EM field problems. A way to overcome this problem could be using the FEM solution on a coarse mesh as the training dataset, but this would result in an overcomplicated algorithm, a larger computational cost and would still require the use of conventional solvers, making ANNs not a real alternative to them.

An idea to overcome this problem comes from the fact that electromagnetic problems are usually described by a set of partial differential equations and boundary and initial conditions. Embedding the knowledge of the governing PDE in the network training process could be a solution to the problem of data availability. In the next chapter we will see how this can be implemented and what are the advantages of this approach over the purely data driven application of ANNs described in this chapter.

Chapter 2

Physics-informed neural networks

2.1 What is a Physics-informed neural network

In the case of problems described by partial differential equations, an idea to overcome the lack of training data and speed up the convergence of the neural network (that is, the minimization of its cost function), is to embed the knowledge of the governing PDE inside the cost function. From this idea Physics-informed neural networks (PINNs), a particular type pf ANNs, were derived [11].

PINNs are universal function approximators [12] that can be used to solve problems involving partial differential equations. The name comes from the fact that the physical laws that govern the studied phenomena are used to drive the training process of the network, resulting in a solution that is physically consistent without needing a huge amount of training data [11]. This feature makes PINNs very interesting for the solution of electromagnetic field problems.

Maxwell's equations, which describe the laws that govern electromagnetic interactions, involve differential operators such as divergence and curl, making PINNs a suitable tool to solve the problems described by them. Maxwell's equations read:

$$\nabla \cdot \boldsymbol{D} = \rho, \tag{2.1}$$

$$\nabla \cdot \boldsymbol{B} = 0, \tag{2.2}$$

$$\nabla \times \boldsymbol{E} = -\frac{\partial \mathbf{B}}{\partial t}, \tag{2.3}$$

$$\nabla \times \boldsymbol{H} = \boldsymbol{J} + \frac{\partial \boldsymbol{D}}{\partial t}, \tag{2.4}$$

where $\boldsymbol{D}$ is the electric displacement field, ρ is the charge density, $\boldsymbol{B}$ is the magnetic flux density, $\boldsymbol{E}$ is the electric field, $\boldsymbol{H}$ is the magnetic field strength and $\boldsymbol{J}$ is the current density. These equations can be rearranged and combined depending on the particular conditions of the problem (e.g. electrostatics, magnetostatics etc.). Under specific conditions, discussed in Section 2.6, they can be reduced to a particular form of the equation:

$$\nabla \cdot k_1 \nabla u + k_2 \frac{\partial u}{\partial t} = F, \tag{2.5}$$

which describes field problems characterized by the presence of a scalar potential u. Together with boundary and initial conditions, (2.5) fully describes the problem to be solved. In (2.5), u is the scalar potential (electric or magnetic), F is the source term (e.g. current density) and k_1 and k_2 are material properties (such as permittivity and reluctivity). Although the application of PINNs is not limited to this type of PDEs, in this thesis we are going to deal only with equations like (2.5). A PINN can be trained so that its output approximates the potential distribution u in a domain. This is done by making it fulfill the PDE together with boundary and initial conditions. In the next paragraphs we will see how the knowledge of the physical laws can be embedded in the cost function and how the boundary and initial conditions can be used as additional training data in order to make the network output convergent to the exact solution.

2.2 Cost function definition

We consider, for the sake of simplicity, static problems, for which the time derivative in (2.5) can be dropped. The most straightforward way to approach the PINN training process is to divide the space domain into two parts: the boundary and the internal domain. We then sample the two subdomains to obtain points to train our network. This can be done in many different ways, but the easiest one is to mesh the domain and use the mesh nodes as sample points. The nodes belonging to the internal domain are called *collocation points* and on those points the network output must fulfill the governing PDE. On the boundary nodes instead, the output must match the boundary conditions. To achieve this, a two-term cost (or loss) function is defined. The first term is related to boundary conditions and the second term to the PDE. In the case of Dirichlet boundary conditions, the first term is defined like in (1.6):

$$L_b = \frac{1}{B}\sum_{i=1}^{B} ||Y_{b,i} - BC_i||^2, \tag{2.6}$$

where B is the number of sample points on the boundary, $Y_{b,i}$ is the network output on the i-th boundary point and BC_i is the value of the potential on the i-th boundary point prescribed by Dirichlet boundary conditions. In the case of Neumann conditions, (2.6) doesn't change much. The only thing is that we need to compute the derivative of the network output on Neumann boundary points. This can be easily done by a technique called *automatic differentiation* (AD), which will be discussed in Section 2.3. The second term of the cost function is directly derived form the differential equation. Equation (2.5), in the case of static problems, reads:

$$\nabla \cdot k\nabla u = F. \tag{2.7}$$

From (2.7), we can compute the residual of the equation as:

$$res = F - \nabla \cdot k\nabla u = 0. \tag{2.8}$$

In order to obtain a solution consistent with the equation, the identity (2.8) must be fulfilled at collocation points, thus obtaining the second term of our loss function:

$$L_{PDE} = \frac{1}{C}\sum_{i=1}^{C} res_i^2, \tag{2.9}$$

$$res_i = ||F_i - \nabla \cdot k\nabla u||, \tag{2.10}$$

where C is the number of collocation points, k and u are respectively the vector of the values of the material coefficient and the vector of the values of the network output on the collocation points and F_i is the value of the source term on the i-th collocation point. Now we can combine the two terms to obtain the complete cost function:

$$L = \lambda_1 L_b + \lambda_2 L_{PDE}, \tag{2.11}$$

$$\lambda_1 + \lambda_2 = 1. \tag{2.12}$$

By minimizing the cost function L, an approximate solution of the field problem is sought.

The two main aspects of the training process are the computation of the cost function, which involves the differentiation of the network output with respect to the input coordinates, and its optimization, which is gradient-based and involves the differentiation of the loss function with respect to the network parameters. Differentiation and optimization are discussed in the next sections.

2.3 Automatic Differentiation

As it can be observed in (2.10) and (2.11), in order to train the network we need to compute the derivatives of its output. There are several methods to compute derivatives of functions, the most commonly used being the numerical ones, who rely on *finite difference* approximation. Given a function $f : R^n \to R$, its gradient $\nabla f = (\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_i}, ..., \frac{\partial f}{\partial x_n})$ can be approximated using

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \frac{f(\boldsymbol{x} - h\boldsymbol{e_i}) - f(\boldsymbol{x})}{h}, \tag{2.13}$$

where $\boldsymbol{e_i}$ is the unit vector along the x_i-axis and h is the step size. (2.13) is obtained from the definition of the derivative of a function, i.e. when $h \to 0$. Numerical differentiation is widely used because of its simplicity of implementation, but, as Baydin et al. remark in [13], can result in low accuracy due to truncation errors (due to h being finite) and round-off errors (due to a finite number of digits in computations). The accuracy is also dependent on the choice of the step size h. Moreover, from (2.13), if we need to compute the gradient of a function f in n dimensions, we need to make n evaluations of f at each sample point. This makes numerical differentiation not suitable for neural networks training, in which we need to compute the gradients of the network output with respect to a huge number of variables.

This leads us to the introduction of a different method for differentiating functions, the so-called *automatic differentiation* (AD). As Baydin et al. explain in [13], AD is based on the fact that every numerical computation is, ultimately, a combination of elementary operations, whose derivatives are known. This means that the derivative of the original computation can be obtained combining the derivatives of its constituent operations. These operations include arithmetic operations and trigonometric, logarithmic and exponential functions. A function $f : R^n \to R^m$ can be reconstructed according to a three-part notation [14], using intermediate variables v_i. The v_i are defined as follows:

- $v_{i-n} = x_i, \quad i = 1, ..., n$ are the input variables
- $v_i, \quad i = 1, ..., l$ are the working (intermediate) variables
- $y_{m-i} = v_{l-i}, \quad i = m-1, ..., 0$ are the output variables.

AD makes use of this notation to compute the derivative of the function f. For the reason mentioned above, this method is suitable for differentiating functions expressed not only in closed form but also through an algorithm (numerical algorithms perform a combination of elementary operations). This is of particular interest since, in the practical implementation of PINNs, we will need to compute the gradient of functions expressed as algorithms.

AD can be performed in two different modes: *forward mode* and *reverse mode*. Here we only present AD in reverse mode, since it is the mode that is commonly used in ML applications. An in-depth investigation on this topic is presented in [13]. The following example, proposed by Baydin et al. in [13] to introduce AD, is provided. If we consider a function $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$, evaluated at $(x_1, x_2) = (2, 5)$, we can compute its gradient $\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2})$ in the following way. We split the process into two phases. An initial

Intermediate variables		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln(v_{-1})$	$= \ln(2)$
v_2	$= v_{-1}v_0$	$= 2 \cdot 5$
v_3	$= \sin(v_0)$	$= \sin(5)$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

TABLE 2.1: AD, intermediate variables (forward phase).

forward phase, in which we compute all the intermediate variables v_i (see table 2.1), and a second *reverse* phase, in which we compute the desired derivative combining the derivatives of the intermediate variables. Once the v_i have been defined, we proceed by assigning to each intermediate variable its complement

$$\bar{v}_i = \frac{\partial y}{\partial v_i}, \tag{2.14}$$

which represents the sensitivity of y with respect to a variation of v_i. Our goal is to compute $\bar{v}_{-1}$ and $\bar{v}_0$. From Table 2.1, we see that v_0 affects y through v_2 and v_3. This means that ve can compute $\bar{v}_0$ as:

$$\bar{v}_0 = \frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0} = \bar{v}_2\frac{\partial v_2}{\partial v_0} + \bar{v}_3\frac{\partial v_3}{\partial v_0}. \tag{2.15}$$

At this point, it is clear how we can compute all the derivatives just by looking at how the variables interact with each other.
Table 2.2, shows how the derivatives of f with respect to the input (x_1, x_2) are computed. Note that $\bar{v}_0$ and $\bar{v}_{-1}$ are computed in two incremental steps. In the case of $R^n \rightarrow R$ functions, like the one in the example, only one run of the reverse mode is needed to compute the gradient of the function $\nabla f = (\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_i}, ..., \frac{\partial f}{\partial x_n})$. This is a big advantage in network training applications, since gradients of scalar functions with a large number of variables need to be computed.

2.4 The Adam optimizer

The minimization of the cost function of the problem is the key step in the training of a neural network. In order for the network output to be an accurate approximation of the required solution, we need to adjust the network parameters through an optimization process [8]. Among the family of gradient based optimization algorithms, a fairly recent one has emerged in ML applications, *Adam* [15], which is an algorithm based on the adaptive estimation of first and second moments of the gradient. It combines two gradient descent methods, i.e. the *Momentum* method, which uses only the first moment, and *Root Mean Square Propagation* (RMSProp) method, which uses only the second moment. Adam, like every optimization algorithm, is iterative, and the first and second moments of the gradient at iteration k can be computed as:

$$m_k = \beta_1 \cdot m_{k-1} + (1 - \beta_1) \cdot \nabla_\theta C(\theta_{k-1}), \tag{2.16}$$

$$v_k = \beta_2 \cdot v_{k-1} + (1 - \beta_2) \cdot (\nabla_\theta C(\theta_{k-1}))^2, \tag{2.17}$$

Derivatives calculation		
from y expression:		
$\bar{v}_5$	$=\bar{y}$	$=1$
from v_5 expression:		
$\bar{v}_4$	$=\bar{v}_5\frac{\partial v_5}{\partial v_4}=\bar{v}_5\cdot 1$	$=1$
$\bar{v}_3$	$=\bar{v}_5\frac{\partial v_5}{\partial v_3}=\bar{v}_5\cdot(-1)$	$=-1$
from v_4 expression:		
$\bar{v}_1$	$=\bar{v}_4\frac{\partial v_4}{\partial v_1}=\bar{v}_4\cdot 1$	$=1$
$\bar{v}_2$	$=\bar{v}_4\frac{\partial v_4}{\partial v_2}=\bar{v}_4\cdot 1$	$=1$
from v_3 expression:		
$\bar{v}_0$	$=\bar{v}_3\frac{\partial v_3}{\partial v_0}=\bar{v}_3\cos v_0$	$=-0.284$
from v_2 expression:		
$\bar{v}_{-1}$	$=\bar{v}_2\frac{\partial v_2}{\partial v_{-1}}=\bar{v}_2 v_0$	$=5$
$\bar{v}_0$	$=\bar{v}_0+\bar{v}_2\frac{\partial v_2}{\partial v_0}=\bar{v}_0+\bar{v}_2 v_{-1}$	$=1.716$
form v_1 expression:		
$\bar{v}_{-1}$	$=\bar{v}_{-1}+\bar{v}_1\frac{\partial v_1}{\partial v_{-1}}=\bar{v}_{-1}+\frac{\bar{v}_1}{v_{-1}}$	$=5.5$
$\bar{x}_1=\frac{\partial y}{\partial x_1}$	$=\bar{v}_{-1}$	$=5.5$
$\bar{x}_2=\frac{\partial f}{\partial x_2}$	$=\bar{v}_0$	$=1.716$

TABLE 2.2: AD, calculation of the desired derivatives (reverse phase).

where m_k and v_k are the first and second moment at iteration k, β_1 and β_2 are two hyperparameters, valued between 0 and 1 (usually close to 1), $\nabla_\theta C(\theta_{k-1})$ is the gradient of the cost function computed after $(k-1)$ iterations and θ_{k-1} is the set of network parameters at iteration $(k-1)$. As we said, Adam is a combination of two algorithms. The update rules for Momentum and RMSProp are:

$$Momentum: \ \theta_k = \theta_{k-1} - \alpha \cdot m_k, \tag{2.18}$$

$$RMSProp: \ \theta_k = \theta_{k-1} - \frac{\alpha}{(\sqrt{v_k}+\varepsilon)}, \tag{2.19}$$

where α is the learning rate, which is set *a priori* and controls the amplitude of the movement in the search space [16], and ε is a small coefficient to avoid division by zero. As we can see from (2.18) and (2.19), Momentum uses the previous gradient to smooth out fluctuations in the optimization process, while RMSProp scales the learning rate based on the magnitude of recent gradients [17]. Adam combines the two update rules to obtain an adaptive learning rate for each parameter. Before proceeding, we note that at the beginning of the process, m_k and v_k are set to zero, and, since β_1 and β_2 are close to 1, they both tend to be biased towards zero. A solution implemented in Adam to avoid this bias is the following:

$$\hat{m}_k = \frac{m_k}{(1-\beta_1^k)}, \tag{2.20}$$

$$\hat{v}_k = \frac{v_k}{(1-\beta_2^k)}. \tag{2.21}$$

We can now use $\hat{m}_k$ and $\hat{v}_k$ to update the network parameters in the following way:

$$\theta_k = \theta_{k-1} - \alpha \cdot \frac{\hat{m}_k}{(\sqrt{\hat{v}_k} + \varepsilon)}. \tag{2.22}$$

Adam offers a few advantages compared to other optimization algorithms [17], such as adaptive learning rates for each parameter, which allows a faster convergence and higher accuracy in high-dimensional parameter spaces (which is the case of neural networks), the bias correction of the moments estimates and an overall robustness to hyperparameter choices.

In the next paragraph we combine all the knowledge acquired so far about cost function definition, automatic differentiation and Adam optimization to explain the training process of a PINN more in-depth.

2.5 Network training

At this point, we have all the elements to understand how the training of a neural network works. Once we have defined the collocation points, the boundary points and the network structure (number of layers, number of neurons in each layer and activation function type), and we have initialized the network parameters, we can set the training process hyperparameters. The first one is the number of *training epochs*, which can range anywhere from less than a hundred to several hundreds of thousands, depending on the number of sampling points and the complexity of the problem. The training can be performed simultaneously on all the collocation points or can be performed on batches of points at each iteration, so the next hyperparameter to set is the *batch size*. If the batch size is equal to the number collocation points, each iteration will correspond to one epoch, otherwise we will have more than one iteration in each epoch.

Then we have to choose the optimization algorithm hyperparameters. The learning rate, usually in the order of $10^{-1} \div 10^{-5}$ [15], can be kept constant or can decay at each iteration as:

$$\alpha_k = \frac{\alpha_0}{1 + \beta \cdot k}, \tag{2.23}$$

where α_0 is the initial learning rate, β is the decay rate and k is the current iteration. β_1 and β_2 are usually kept at their default value ($\beta_1 = 0.9, \beta_2 = 0.999$) and ε is set in the order of 10^{-8} [17].

Once all the hyperpameters have been set, we can start the optimization process. The collocation points are divided in batches, and a batch queue is formed. The cost function C is computed on the batch collocation points and on the boundary points, as described in Section 2.2, and then its gradient with respect to the network parameters is obtained through AD. Once the gradient is computed, we pass it to the Adam algorithm, which updates the network parameters. The process is repeated on the next batches of points until the batch queue is empty. Then the algorithm proceeds to the next training epoch, until all epochs are performed and the training process is over. Algorithm 1 shows a pseudocode describing the network training process.

Algorithm 1 A pseudocode for network training

```
Require: $PDEcoefficients$,$BoundaryConditions$
Require: $CollocationPoints$, $BoundaryPoints$
Require: $NetworkStructure$, $\theta$                                  ▷ $\theta$=network parameters
Require: $NumEpochs$, $BatchSize$
Require: $\alpha, \beta, \beta_1, \beta_2, \varepsilon$          ▷ leaning rate, decay rate, adam hyperparameters
  form $BatchQueue$
  $Epoch \leftarrow 0$                                                  ▷ initialize Epoch number
  while $Epoch \neq NumEpochs$ do
      $Epoch \leftarrow Epoch + 1$
      while BatchQueue hasdata do
          $XY \leftarrow$ next($BatchQueue$)
          $\mathbf{u} \leftarrow NetworkOutput(XY)$                     ▷ network output on batch points
          $\mathbf{Y}_b \leftarrow NetworkOutput(BoundaryPoints)$       ▷ network output on boundary points
          $C \leftarrow C_b(\mathbf{Y}_b) + C_{PDE}(\mathbf{u})$        ▷ compute the cost function
          $gradient \leftarrow \nabla_\theta C$                         ▷ compute the cost function gradient via AD
          $\theta \leftarrow AdamUpdate(gradient, \alpha, \beta_1, \beta_2, \varepsilon)$ ▷ use Adam to update network parameters
      end while
      $\alpha \leftarrow \alpha/(1 + \beta \cdot Epoch)$                ▷ update learning rate
      reset($BatchQueue$)
  end while
  return $\theta$                                                       ▷ optimal network parameters
```

Once the training is successfully completed, i.e. the cost function has reached a value very close to zero, we can evaluate the network output on a set of domain points different from collocation points, and we will obtain the solution of the given problem on those points. If we have a solution obtained analytically or through a conventional solver (for example FEM) on the same set of points, we can compare it to the solution obtained through the PINN to see if the training process was actually done properly. This phase is called *network testing* and serves as validation of the whole algorithm.

2.6 Application of PINNs to computational electromagnetics

Electromagnetic phenomena are generally described by Maxwell's equations. Maxwell's equations can be combined into a single equation that, together with boundary conditions, fully describes the electromagnetic problem. Only electrostatic and magnetostatic problems are examined here.

2.6.1 Electrostatic formulation

The electromagnetic problem is static, when quantities are constant in time. In particular, we are in *electrostatic* conditions when the time derivative of the electric field is equal to zero. In electrostatic conditions, we can describe our problem using a set of three equations:

$$\nabla \times \boldsymbol{E} = 0 \tag{2.24}$$

$$\nabla \cdot \boldsymbol{D} = \rho \tag{2.25}$$

$$\boldsymbol{D} = \varepsilon \boldsymbol{E}. \tag{2.26}$$

Since the electric field is curl-free, there exists an electric scalar potential V such that:

$$\boldsymbol{E} = -\nabla V. \tag{2.27}$$

Combining (2.24), (2.25), (2.26) and (2.27) we obtain a single equation involving only the scalar potential V:

$$\nabla \cdot \varepsilon \nabla V = -\rho, \tag{2.28}$$

which is a particular form of (2.5) and can be used to train a PINN as described in Sections 2.2 and 2.5.

2.6.2 Magnetostatic formulation

In *magnetostatic* conditions the time derivative of the magnetic flux density is equal to zero. From Maxwell's equations ((2.1)-(2.4)):

$$\nabla \times \boldsymbol{H} = \boldsymbol{J} \tag{2.29}$$

$$\nabla \cdot \boldsymbol{B} = 0 \tag{2.30}$$

$$\boldsymbol{H} = \nu \boldsymbol{B}, \tag{2.31}$$

the magnetic flux density is divergence free, there exists a magnetic vector potential $\boldsymbol{A}$ such that:

$$\boldsymbol{B} = \nabla \times \boldsymbol{A}. \tag{2.32}$$

To ensure the uniqueness of the solution, we need to impose a gauge on $\boldsymbol{A}$. For magnetostatics, the Coulomb gauge is typically adopted:

$$\nabla \cdot \boldsymbol{A} = 0, \tag{2.33}$$

Combining (2.29), (2.30) (2.31) and (2.32) we obtain a single equation involving only the vector potential $\boldsymbol{A}$:

$$\nabla \times \nu \nabla \times \boldsymbol{A} = \boldsymbol{J}. \tag{2.34}$$

For 2D magnetostatics, if plane symmetry is present (i.e. the x and y components of $\boldsymbol{B}$ vary only on the (x,y) plane), we can rearrange (2.34) to obtain an equation like (2.5). If $\boldsymbol{B}$ only has components on the (x,y) plane, i.e. $\boldsymbol{B} = (B_x, B_y, 0), \boldsymbol{A}$ only has one component along the z-axis ($\boldsymbol{A} = (0,0,A_z)$). If we formally compute the curl of $\boldsymbol{A}$, we obtain:

$$\boldsymbol{B} = \nabla \times \boldsymbol{A} = (\frac{\partial A_z}{\partial x}, -\frac{\partial A_z}{\partial y}, 0) \tag{2.35}$$

By using $\boldsymbol{H} = \nu \boldsymbol{B} = (\nu B_x, \nu B_y, 0)$ and (2.29) we obtain:

$$-\nabla \cdot \nu \nabla A_z = J_z, \tag{2.36}$$

which is a particular form of (2.5) and can be used to train a PINN as described in Section 2.2 and 2.5.

2.7 PINNs for the analysis of transient and non-linear problems

From what we have seen in this chapter, PINNs seem to be a promising alternative to conventional PDE solvers like FEM. They can approximate potential distributions in a bounded space domain in static conditions and, thanks to the introduction of the governing equation into the training process, the solution will be consistent with the physics of the problem. Moreover, a small adjustment in the cost function definition could allow PINNs to also treat problems in time domain without the need for time-stepping [10]. It would be sufficient to add time as an input of the network and take collocation points in space and time [9]. The cost function can be modified by adding the time-dependent term of the PDE to (2.10) and add a term to (2.11) to take initial conditions into consideration.

PINNs could also be a great tool for non-linear electromagnetic problems. When we treat magnetic media with non-linear *B-H* characteristics with FEM, we need to perform an iterative procedure [18] to account for the variations of the magnetic permeability, which can be very time-consuming. If we take the variations of μ into consideration during the network training process, we could use PINNs to treat non-linear problems just as we do with linear ones.

The goal of this thesis is to find out if it is actually possible to practically implement what is presented in this chapter at the current stage of PINNs development. We will investigate PINNs capabilities and limitations when it comes to computational electromagnetism, starting from a simple problem like the solution of Poisson's equation on a circular domain and then moving on to more complex settings and tasks. A simple *Matlab* algorithm is presented in the next chapter as a starting point, and then, in the following part of the thesis, we will try to modify it to obtain the solution of increasingly more challenging electromagnetic problems.

Chapter 3

PINN solution of Poisson's equation on a unit disk

3.1 Problem description

One of the simplest examples of partial differential equations is the so-called *Poisson's equation*. In its generic form it can be written as:

$$-\nabla^2 u = f, \tag{3.1}$$

where u is the potential distribution, f is the source function and ∇^2 is the *Laplacian*. The Laplace operator can be written in carthesian coordinates as:

$$\nabla^2 = (\frac{\partial^2}{\partial x^2}, \frac{\partial^2}{\partial y^2}, \frac{\partial^2}{\partial z^2}). \tag{3.2}$$

In order to introduce the practical implementation of PINNs, the example of PINN solver present in the MATLAB library [19] is here examined. In this example, the Poisson's equation, with $f = 1$, is solved on a unit disk (i.e. a circle with $r = 1$), with $u = 0$ on the boundary (Dirichlet condition).The collocation points and boundary points are obtained from the triangle mesh of the disk and then the potential distribution is approximated by training a PINN. To test its accuracy, the solution is compared with the analytical solution, which in this case is:

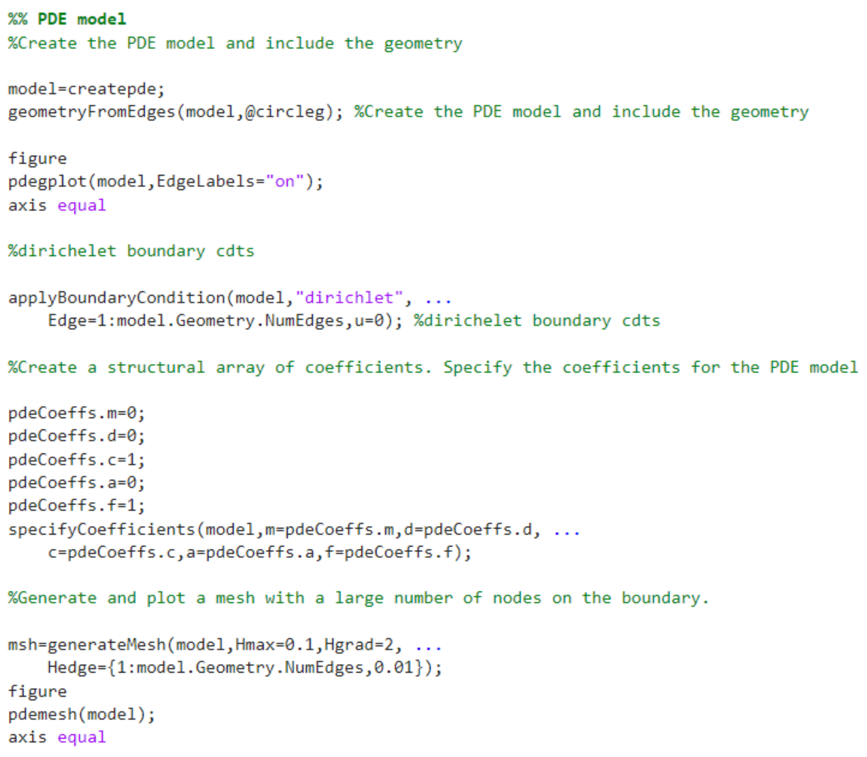
$$u(x,y) = \frac{1 - x^2 - y^2}{4}. \tag{3.3}$$

3.2 Data and geometry

The boundary value problem to solve, as already mentioned in the previous Section, is:

$$\begin{aligned} -\nabla^2 u = 1 \text{ on } \Omega \\ \text{with } u = 0 \text{ on } \partial\Omega. \end{aligned} \tag{3.4}$$

The domain Ω is a circle with unitary radius. The MATLAB *PDE toolbox* [20] is used to set up the model and generate the mesh. Figure 3.1 shows how to generate the model and the mesh using the functions provided by the PDE toolbox. The function *createpde* generates a blank PDE model, then the geometry of the domain is defined by the function *geometryFromEdges* (*circleg* is the function that describes a unitary disk center at the origin). Dirichlet boundary condition is imposed using *applyBoundaryCondition*, where *"Edge"* defines the edges on which the condition is imposed. After that, the function *specifyCoefficients* is used to assign to each



```
%% PDE model
%Create the PDE model and include the geometry

model=createpde;
geometryFromEdges(model,@circleg); %Create the PDE model and include the geometry

figure
pdegplot(model,EdgeLabels="on");
axis equal

%dirichelet boundary cdts

applyBoundaryCondition(model,"dirichlet", ...
    Edge=1:model.Geometry.NumEdges,u=0); %dirichelet boundary cdts

%Create a structural array of coefficients. Specify the coefficients for the PDE model

pdeCoeffs.m=0;
pdeCoeffs.d=0;
pdeCoeffs.c=1;
pdeCoeffs.a=0;
pdeCoeffs.f=1;
specifyCoefficients(model,m=pdeCoeffs.m,d=pdeCoeffs.d, ...
    c=pdeCoeffs.c,a=pdeCoeffs.a,f=pdeCoeffs.f);

%Generate and plot a mesh with a large number of nodes on the boundary.

msh=generateMesh(model,Hmax=0.1,Hgrad=2, ...
    Hedge={1:model.Geometry.NumEdges,0.01});
figure
pdemesh(model);
axis equal
```

FIGURE 3.1: Model setup and mesh generation in MATLAB using the PDE toolbox (source: Mathworks website [19])

coefficient of the equation its prescribed value. Coefficients *m, d, c, a, f* are defined as:

$$m\frac{\partial^2 u}{\partial t^2} + d\frac{\partial u}{\partial t} - \nabla(c\nabla u) + au = f, \tag{3.5}$$

so in this specific case the coefficients are imposed as m=0, d=0, c=1, a=0, f=1 in order to obtain a PDE equal to the one defined in (3.4). The domain is then meshed using the function *generateMesh*, through which the target maximum length of the mesh edges (H_{max}), the mesh growth rate (H_{grad}) and the target edge size around selected domain edges (H_{edge}) can be set. In this case a smaller edge length around the boundary edges is chosen to obtain a large number of sampled nodes on the boundary. In this way the boundary conditions can be accurately enforced on all $\partial\Omega$. Figure 3.2 shows the meshed domain.

3.3 Network definition and training process

3.3.1 Network definition

A fully connected network with three hidden layers, with 50 neurons each, is defined. The activation function is *tanh* for all the neurons. The input layer has two neurons: one receives the x-coordinate of the sample points as input and the other receives the y-coordinate. The output layer has one neuron, that will yield the value of the potential as its output. The function *featureInputLayer* is used to define the input layer, *fullyConnectedLayer* to define the hidden layers and the output layer and *tanhLayer* to define the activation function for all the neurons in one layer. Then using the function *dlnetwork* all the information about the network structure are gathered to generate the neural network. All these functions are part of MATLAB *Deep Learning Toolbox* [21].

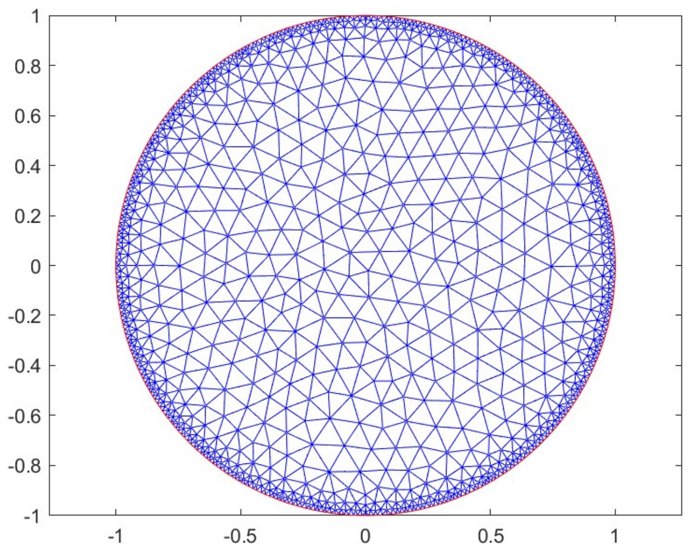
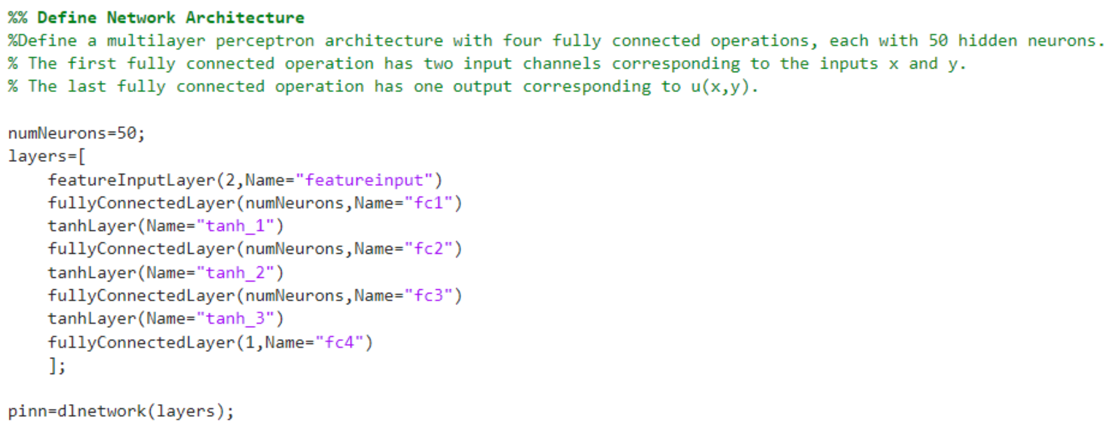


FIGURE 3.2: Geometry and mesh

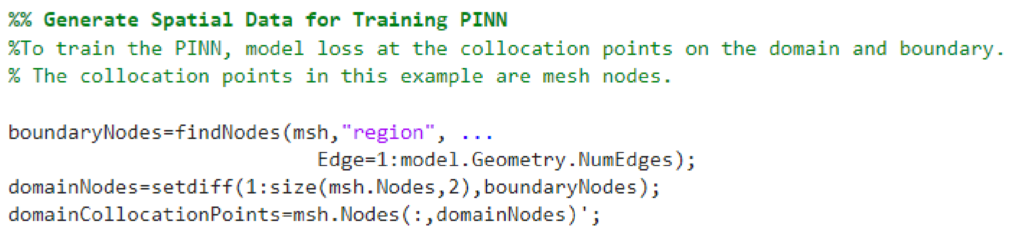


```
%% Define Network Architecture
%Define a multilayer perceptron architecture with four fully connected operations, each with 50 hidden neurons.
% The first fully connected operation has two input channels corresponding to the inputs x and y.
% The last fully connected operation has one output corresponding to u(x,y).

numNeurons=50;
layers=[
    featureInputLayer(2,Name="featureinput")
    fullyConnectedLayer(numNeurons,Name="fc1")
    tanhLayer(Name="tanh_1")
    fullyConnectedLayer(numNeurons,Name="fc2")
    tanhLayer(Name="tanh_2")
    fullyConnectedLayer(numNeurons,Name="fc3")
    tanhLayer(Name="tanh_3")
    fullyConnectedLayer(1,Name="fc4")
    ];

pinn=dlnetwork(layers);
```

FIGURE 3.3: Definition of the network structure (source: Mathworks website [19])

3.3.2 Training points choice and training options

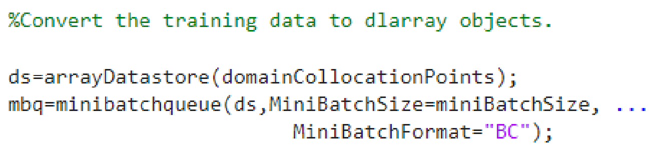
Now the sample points for the training process need to be defined. The easiest choice is to simply take the boundary nodes of the mesh as boundary points and the inner nodes as collocation points. The boundary nodes are identified using the function *findNodes*, which finds the indexes of mesh nodes belonging to a certain region (in this case the boundary). The number of collocation points obtained in this way is 3853, the number of boundary nodes 728. The collocation points are fed to the network in mini-batches. Mini-batches are sets of points of predefined size that are formatted in order to be suitable inputs for the neural network [24]. These mini-batches of points are generated by the function *minibatchqueue*, which receives collocation points coordinates (in the form of an *arrayDatastore* object [25]), batch size and data format as inputs, and outputs a series of batches containing the collocation points coordinates in the form of formatted *deep learning arrays*. These arrays store data with format information, and allow the computation of derivatives through automatic differentiation [23].



```
%% Generate Spatial Data for Training PINN
%To train the PINN, model loss at the collocation points on the domain and boundary.
% The collocation points in this example are mesh nodes.

boundaryNodes=findNodes(msh,"region", ...
                          Edge=1:model.Geometry.NumEdges);
domainNodes=setdiff(1:size(msh.Nodes,2),boundaryNodes);
domainCollocationPoints=msh.Nodes(:,domainNodes)';
```

FIGURE 3.4: Identification of the training points (source: Mathworks website [19])

The network that we defined earlier can receive as input 1×2 vectors representing the coordinates of each sample point. If we have a batch of N points on which the network output has to be evaluated, we need to store it as a deep learning array with 2 channels (corresponding to the two dimensions x, y) of size N. This is done using *"BC"* as the format input (B=batch size, C=number of channels) in *minibatchqueue*. This way the network knows that each row of the input batch corresponds to a point and each column corresponds to a dimension. The size of



```
%Convert the training data to dlarray objects.

ds=arrayDatastore(domainCollocationPoints);
mbq=minibatchqueue(ds,MiniBatchSize=miniBatchSize, ...
                        MiniBatchFormat="BC");
```

FIGURE 3.5: Generation of the mini-batches of collocation points (source: Mathworks website [19])

each batch, the number of epochs, the initial learning rate of the optimization algorithm and its decay rate represent the *training options* of the problem, i.e. the hyperparameters that we need to set a priori. In this example, the hyperparamters are chosen as:

- $n_{epochs} = 50$: number of training epochs,
- $batchsize = 500$: number of points in each batch,
- $\alpha_i = 0.01$: initial learning rate,
- $\beta = 0.005$: learning rate decay.

3.3.3 Training of the network

Once the collocation points have been chosen and the training options have been set, the training process can be started. In each training epoch the algorithm has to evaluate the network output on every batch of training points, then compute the loss function value, its gradient with respect to the network parameters and update the parameters using Adam optimizer. Once this is done for all the batches of points, the algorithm updates the *training progress monitor* (a tool used to visualize the loss function progress in real time [27]) with the latest loss function value, adjusts the learning rate according to its decay rate, and then proceeds to the next epoch. Practically, this results in two nested *while* loops: the inner one does all the computations for every batch of points and the outer one performs the loss function and learning rate updates. As shown in Figure 3.6, the outer *while* loop goes on until all the training epochs have been performed, or the training process is stopped by the user. It updates the epoch number and resets the batch queue and then moves on to the inner loop, which goes on until there are no more batches in the queue. The inner loop updates the iteration number, takes the next batch

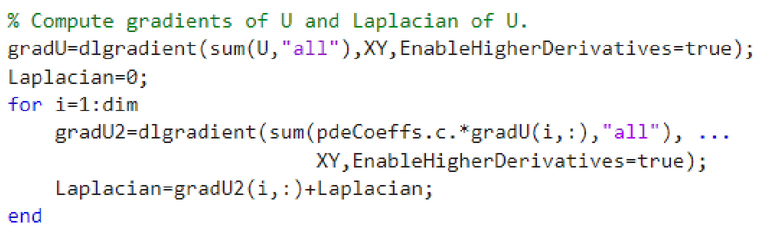
```
%% Train PINN

%Train the model using a custom training loop.
%Update the network parameters using the adamupdate function.
%At the end of each iteration, display the training progress.
%This training code uses the modelLoss helper function.
%For more information, see Model Loss Function.

iteration=0;
epoch=0;
learningRate=initialLearnRate;
while epoch<numEpochs && ~monitor.Stop
    epoch=epoch+1;
    reset(mbq);
    while hasdata(mbq) && ~monitor.Stop
        iteration=iteration+1;
        XY=next(mbq);
        % Evaluate the model loss and gradients using dlfeval.
        [loss,gradients]=dlfeval(@modelLoss,model,pinn,XY,pdeCoeffs);
        % Update the network parameters using the adamupdate function.
        [pinn,averageGrad,averageSqGrad]=...
            adamupdate(pinn,gradients,averageGrad, ...
                       averageSqGrad,iteration,learningRate);
    end
    % Update learning rate.
    learningRate=initialLearnRate/(1+learnRateDecay*iteration);
    % Update the training progress monitor.
    recordMetrics(monitor,iteration,Loss=loss);
    updateInfo(monitor,Epoch=epoch+" of "+numEpochs);
    monitor.Progress=100*iteration/numIterations;
end
```

FIGURE 3.6: The training algorithm (source: Mathworks website [19])

of points in the queue and passes it, together with all the other required inputs, to the function that computes the loss function value and its gradient. The command *dlfeval* [26] is used to evaluate the function output at each iteration. This command is used to evaluate user defined deep learning models, such as that implemented in the function *modelLoss*, proposed in the example. The loss function defined by the user takes the PDE model, the network structure, the current batch of collocation points and the coefficients of the PDE as inputs, and yields the value of the loss function and its gradient with respect to the network parameters.

The first step is to compute the network output on the collocation points. This is done using the command *forward*, which takes the network structure and the coordinates as inputs and gives a vector with values of the network output on the input coordinates as a result. After that, the Laplacian of the output is computed. This is carried out by evaluating the gradient of the output with respect to the collocation points coordinates and then the gradient of the x and y components of the gradient, again with respect to the collocation points coordinates. To compute gradients, we use automatic differentiation. In MATLAB, automatic differentiation is performed using the command *dlgradient* [23], which takes the sum of the network output vector components and the coordinates as inputs, and gives the gradient of the network output with respect to the input coordinates as output. In order to be able to compute the Laplacian, we need to enable the computation of the derivatives of the gradient via the command *Enable-HigherDerivatives*, which ensures the differentiability of the gradient. At this point we can compute the residuals of the PDE on each collocation point of the batch and thus the first term of the loss function. The residual on the i-th collocation point is computed as:

$$res_i = -f - \nabla^2 u_{PINN,i}, \tag{3.6}$$



```
% Compute gradients of U and Laplacian of U.
gradU=dlgradient(sum(U,"all"),XY,EnableHigherDerivatives=true);
Laplacian=0;
for i=1:dim
    gradU2=dlgradient(sum(pdeCoeffs.c.*gradU(i,:),"all"), ...
                      XY,EnableHigherDerivatives=true);
    Laplacian=gradU2(i,:)+Laplacian;
end
```

FIGURE 3.7: Loss function initialization and gradient computation (source: Mathworks website [19])

where $u_{PINN,i}$ is the neural network output on the i-th collocation point. A vector called ***res***, containing the residuals res_i computed on all the collocation points, is defined. For the training to be considered successful, the residuals should all be zero. We define a target vector ***zeroTarget*** of the same dimension as ***res***, composed of only zeros. The first term of the loss function is:

$$lossF = MSE(\boldsymbol{res}, \boldsymbol{zeroTarget}). \tag{3.7}$$

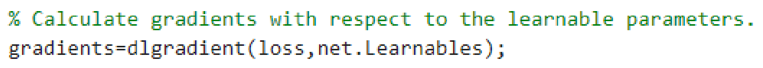
Now the second term of the loss function, the one related to boundary conditions, needs to be computed. First, the mesh nodes belonging to each boundary edge are found with the command *findNodes* and the respective boundary condition with the command *findBoundaryConditions* (in this case it is $u = 0$ on all the boundary). After that, the coordinates of the boundary nodes are formatted using the function *dlarray*, which transforms the matrix of coordinates in a formatted deep learning array (again, we need to set the format command to *"BC"*, see Section 3.2 for the details). Then the output of the network on the boundary nodes is evaluated. If we call the boundary condition imposed on the j-th boundary node as $actualBC_j$ and the network output on the same node as $predictedBC_j$, and we define two vectors ***actualBC*** and ***predictedBC*** containing all the $actualBC_j$ and $predictedBC_j$ terms, the second term of the loss function is computed as:

$$lossU = MSE(\boldsymbol{predictedBC}, \boldsymbol{actualBC}). \tag{3.8}$$

Now we have all the elements to calculate the value of the total loss function. We choose $\lambda_1 = 0.6$ and $\lambda_1 = 0.4$ (see (2.11)), and the the loss function is computed as:

$$loss = 0.6 lossU + 0.4 lossF. \tag{3.9}$$

Now, the loss function gradient with respect to the network parameters (also called *learnables*) is computed with, again, automatic differentiation (see figure 3.8).



```
% Calculate gradients with respect to the learnable parameters.
gradients=dlgradient(loss,net.Learnables);
```

FIGURE 3.8: Calculation of the loss function gradient (source: Mathworks website [19])

After the computation of the loss function value and its gradient, we can proceed with the optimization. The Adam algorithm (see Section 2.4 for details) is implemented in MATLAB with the function *adamupdate* [28]. This function takes the network structure, the loss function gradient, the previous first and second moments of the gradient (*averageGrad* and *averageSqGrad* in figure 3.9, they need to be initialized to zero before the first epoch of training), the current iteration number and the learning rate. As outputs, it gives the updated network structure and the value of the two moments to be used in the next iteration.

```
% Update the network parameters using the adamupdate function.
[pinn,averageGrad,averageSqGrad]=...
    adamupdate(pinn,gradients,averageGrad, ...
               averageSqGrad,iteration,learningRate);
```

FIGURE 3.9: Implementation of the *adamupdate* function (source: Mathworks website [19])

After the process has been repeated with all the batches in the queue, the algorithm updates the learning rate as:

$$\alpha = \frac{\alpha_i}{1+\beta \cdot Epoch}, \tag{3.10}$$

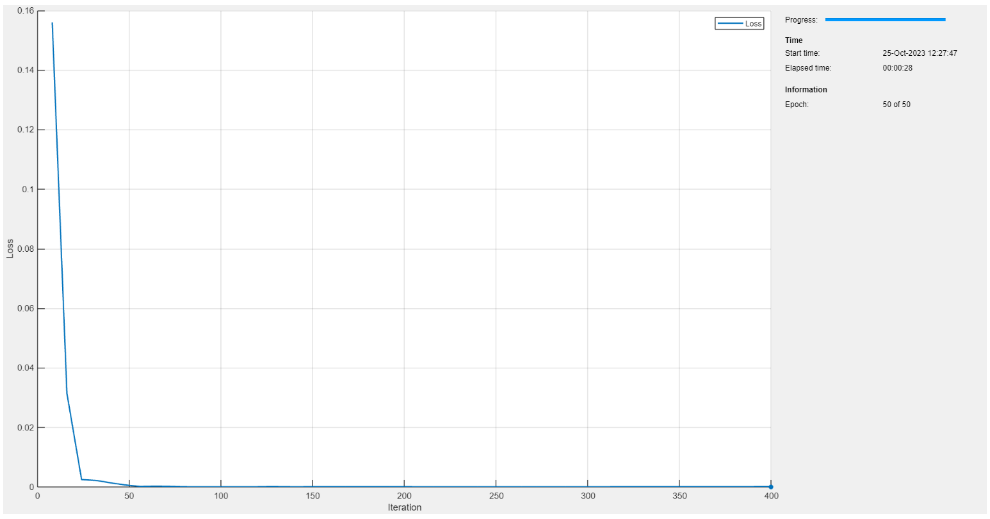
communicates the new loss function value to the training progress monitor and updates its parameters (see last three rows in figure 3.6). Figure 3.10 shows how the training progress monitor is initialized in MATLAB.

```
%Initialize the TrainingProgressMonitor object.

monitor=trainingProgressMonitor(Metrics="Loss", ...
                                  Info="Epoch", ...
                                  XLabel="Iteration");
```

FIGURE 3.10: Initialization of the training progress monitor (source: Mathworks website [19])

3.4 Training results and network testing

After 50 epochs of training, the loss function has reached a value of $1.1686 \cdot 10^{-4}$. By looking at the graph in figure 3.11, we observe that this is three orders of magnitude smaller than the initial loss function value (between 0.16 and 0.15). We can also note that it took 28 seconds to the algorithm to perform the 50 epochs of training.



Time
Start time: 25-Oct-2023 12:27:47
Elapsed time: 00:00:28

Information
Epoch: 50 of 50

FIGURE 3.11: Training progress monitor

To test the accuracy of the solution, $\boldsymbol{U}_{pinn}$, which is the array of potentials evaluated on mesh nodes, can be compared with the analytical solution (3.3) computed on the same nodes ($\boldsymbol{U}_{true}$). To do so we plot the two solutions and we compute the L_2 norm of the error vector as:

$$L_2(e) = \|\boldsymbol{U}_{pinn} - \boldsymbol{U}_{true}\| \tag{3.11}$$

Figure 3.12 shows the analytical solution and the solution obtained through the training of the PINN. We can see that the potential distribution is approximated quite well by the PINN, and the norm of the error is $4.3 \cdot 10^{-1}$. It means that with just 28 seconds of training the network is able to solve the Poisson's problem with good accuracy. To further improve the accuracy of the solution, one can use a finer mesh and a higher number of training epochs. This simple example is useful to understand how the training of a PINN can be implemented in MATLAB and how PINN algorithms can be applied to electromagnetic problems. It also represents a starting point for the development of codes discussed in the next chapters.

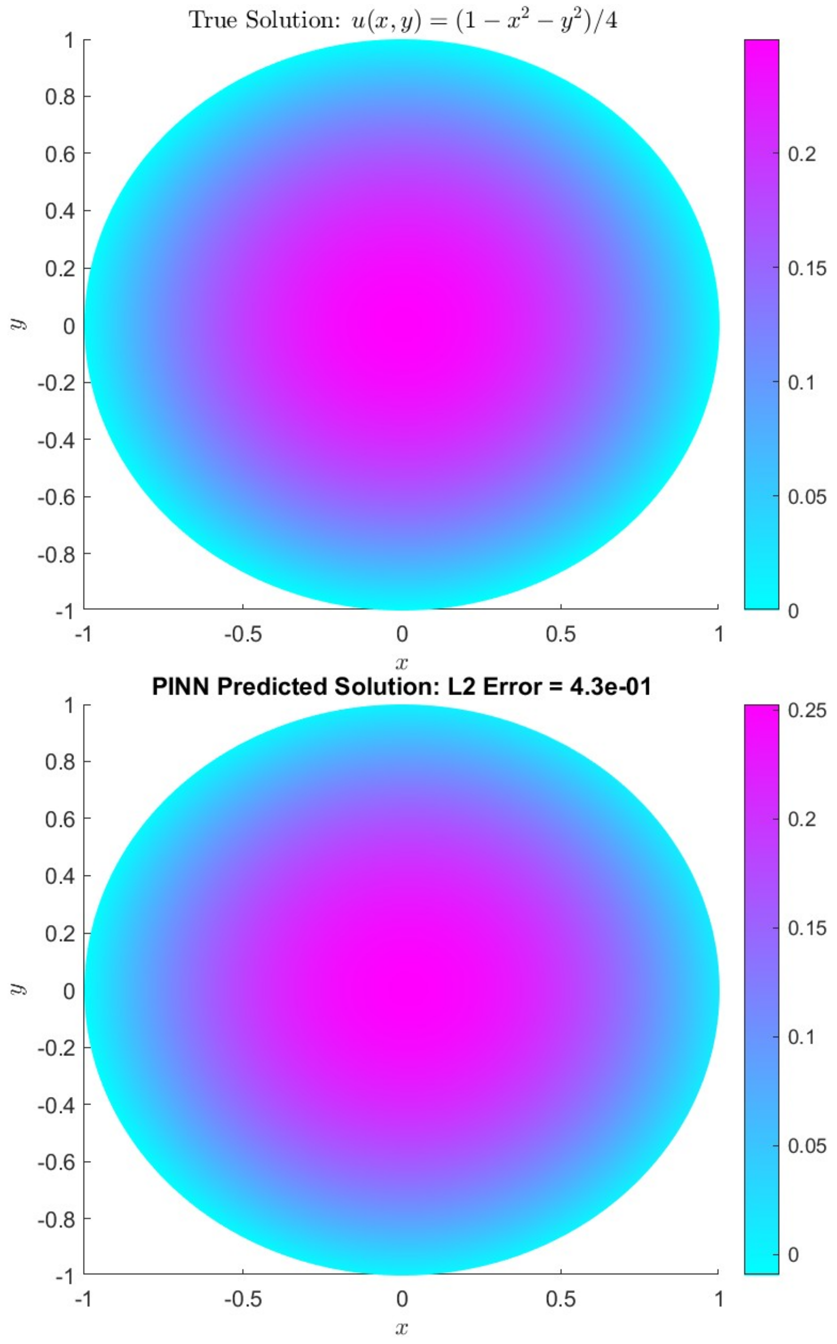


FIGURE 3.12: Analytical solution (above) and PINN solution (below) (source: Mathworks website [19])

Chapter 4

PINN solution of the parallel-plate capacitor problem

4.1 The parallel plate capacitor

The example presented in the previous chapter is the starting point for the development of an original code that is able to solve 1D electrostatic problems using physics-informed neural networks. The parallel plate capacitor problem with analytical solution is considered. It consists of two parallel conductive plates and a dielectric material between them (Figure 4.1).

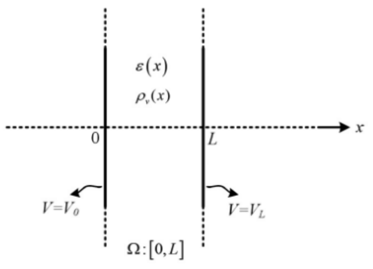


FIGURE 4.1: Parallel plate capacitor

The goal is to compute the potential distribution inside the dielectric region knowing the value of the potential at the terminals (V_0 and V_L), the distance between them (L) and the charge density (ρ_v) and permittivity (ε) between the two plates.

In electrostatic conditions, the potential distribution inside the capacitor is the solution of (2.27). The Dirichlet conditions on the two terminals ensure the uniqueness of the solution. We can use PINNs to solve the equation by modifying the MATLAB code presented in the previous chapter. The accuracy of the PINN solutions was tested by comparing them to analytical solutions or to solutions obtained through a FEM algorithm developed in MATLAB. The FEM algorithm was validated by comparing its results to solutions obtained through *COMSOL Multiphysics*®.

4.2 One-dimensional capacitor with constant permittivity

4.2.1 Problem description

We can analyze a capacitor in one dimension when all the parameters vary only in the direction orthogonal to the two terminals and if we neglect the edge effects. We consider a capacitor on the *xy*-plane with the plates parallel to the *y*-axis and with $\rho_v = 0$ and $\varepsilon = \varepsilon_0 = const$. This means that the electrical potential v will vary only along x-axis and (2.27) becomes:

$$\frac{\partial^2 u}{\partial x^2} = 0. \tag{4.1}$$

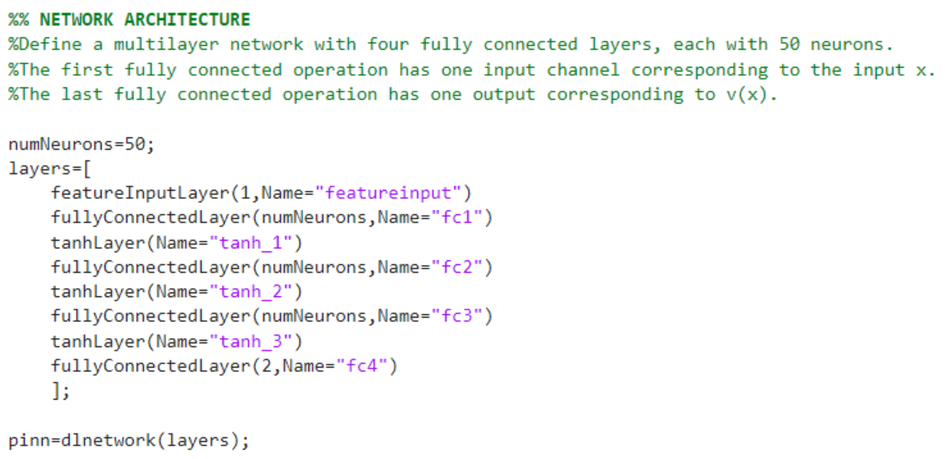
The boundary conditions are $u(x=0) = 1$ V and $u(x=L) = 0$ V. The distance L between the two terminals is 1m.

We start from the domain definition and the mesh. In this case the domain is simply a segment of length L, and can be subdivided in n subsegments of equal length, that will be the elements of the mesh. The higher n, the finer the mesh. In this case we choose $n = 50$, so we have 50 elements and 51 nodes. For a 1D problem it is not necessary to use the MATLAB PDE toolbox to to generate the mesh of the geometry. The equation coefficients are all zero but c, which is equal to the air permittivity ε_0 and can be directly plugged in the loss function formula without using the *specifyCoefficients* function (like we did in Section 3.2). The same applies to the boundary conditions. They can be defined as a two-component vector *BdCond* and passed directly to the loss function input without using the *applyBoundaryCondition* function.

4.2.2 Training process

4.2.2.1 Network definition

Having defined the geometry and data of the problem, we can now create a trainable neural network. We choose a structure with three hidden layers, with 50 neurons each, and *tanh* as activation function for all the layers. The only difference with the example discussed in Chapter 3 is that, since we have a one dimensional domain, we only have one neuron in the input layer, because the input data will only have one component (x-coordinate).

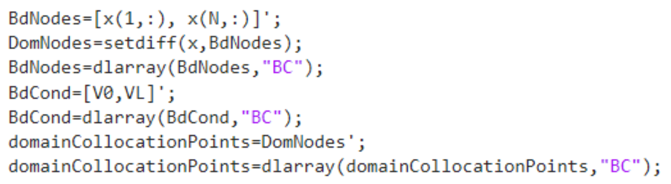


```
%% NETWORK ARCHITECTURE
%Define a multilayer network with four fully connected layers, each with 50 neurons.
%The first fully connected operation has one input channel corresponding to the input x.
%The last fully connected operation has one output corresponding to v(x).

numNeurons=50;
layers=[
    featureInputLayer(1,Name="featureinput")
    fullyConnectedLayer(numNeurons,Name="fc1")
    tanhLayer(Name="tanh_1")
    fullyConnectedLayer(numNeurons,Name="fc2")
    tanhLayer(Name="tanh_2")
    fullyConnectedLayer(numNeurons,Name="fc3")
    tanhLayer(Name="tanh_3")
    fullyConnectedLayer(2,Name="fc4")
    ];

pinn=dlnetwork(layers);
```

FIGURE 4.2: Definition of the neural network

4.2.2.2 Training points choice and training options

The easiest choice for collocation points is to take the mesh nodes, so we will have 49 collocation points. The boundary nodes will be the first and the last node of the mesh. Once the training points have been chosen, we can move on to the training options setting. Since the problem geometry is really simple and we have a much smaller number of training points with respect to the Poisson's problem that we presented in the previous chapter, we can set a higher number of training epochs. For the same reason, the batch size will be equal to the number of



```
BdNodes=[x(1,:), x(N,:)]';
DomNodes=setdiff(x,BdNodes);
BdNodes=dlarray(BdNodes,"BC");
BdCond=[V0,VL]';
BdCond=dlarray(BdCond,"BC");
domainCollocationPoints=DomNodes';
domainCollocationPoints=dlarray(domainCollocationPoints,"BC");
```

FIGURE 4.3: Identification of boundary and collocation points

collocation points, since there is no need to divide such a small number of points in batches. The training process hyperparamters are:

- $n_{epochs} = 100$: number of training epochs,
- $batchsize = 49$: number of points in each batch,
- $\alpha_i = 0.01$: initial learning rate,
- $\beta = 0.005$: learning rate decay.

4.2.2.3 Training of the network

At this point we can start the training process. We proceed in the same way as we did with Poisson's problem, but this time we need to slightly change the MATLAB function to compute the loss function value and gradient. In this case the equation describing the problem is (4.1) and so the residual of the PDE on the i-th collocation point is:

$$res_{1,i} = \frac{\partial^2 u_{PINN,i}}{\partial x^2}, \tag{4.2}$$

where $u_{PINN,i}$ is the network output on the i-th collocation point. We use *dlgradient* to compute the derivative in (4.2). We define a vector $\boldsymbol{res}_1$ containing all the res_i terms. We then define a target vector $\boldsymbol{target}_1$ of zeros with a number of entries equal to the number of collocation points (the residual of the PDE must be zero on every collocation point) and compute the first term of the loss function as:

$$lossF = MSE(\boldsymbol{res}_1, \boldsymbol{target}_1). \tag{4.3}$$

The MSE is computed using the MATLAB function *l2loss*. The second part of the loss function is simply constructed by evaluating the network output on the two boundary nodes, and computing the residual on the j-th boundary point as:

$$res_{2,j} = u_{bd,j} - BdCond_j, \tag{4.4}$$

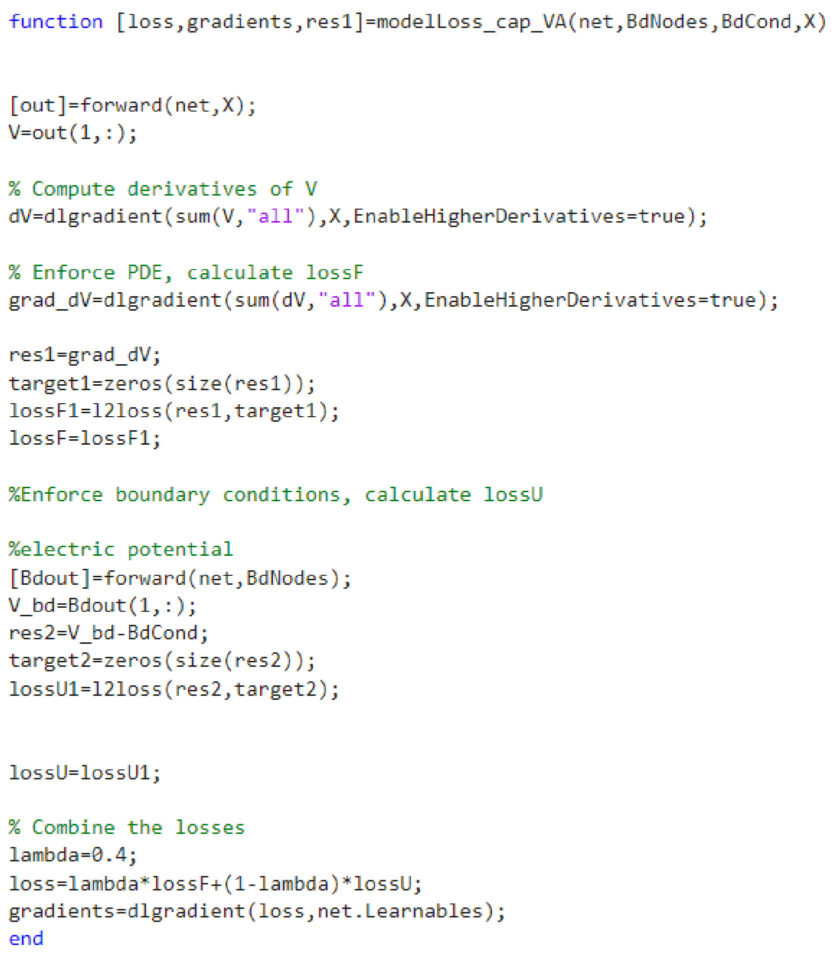
where $u_{bd,j}$ is the network output on the j-th boundary point and $BdCond_j$ is the boundary condition imposed on the j -th boundary point. We define a vector $\boldsymbol{res}_2$ containing all the $res_{2,j}$ terms and a vector $\boldsymbol{target}_2$ of zeros of the same size as $\boldsymbol{res}_2$. Then the loss function term related to the boundary conditions is computed as:

$$lossU = MSE(\boldsymbol{res}_2, \boldsymbol{target}_2), \tag{4.5}$$

where $target_2$ is a 2-component vector of zeros. The final loss function expression (constructed as (2.11)), with $\lambda_1 = 0.4$ and $\lambda_2 = 0.6$ is:

$$loss = 0.4lossF + 0.6lossU. \tag{4.6}$$

The loss function gradient with respect to the network parameters is computed using *dlgradient*. Figure 4.4 shows the MATLAB function for the loss function calculation.



```
function [loss,gradients,res1]=modelLoss_cap_VA(net,BdNodes,BdCond,X)


[out]=forward(net,X);
V=out(1,:);

% Compute derivatives of V
dV=dlgradient(sum(V,"all"),X,EnableHigherDerivatives=true);

% Enforce PDE, calculate lossF
grad_dV=dlgradient(sum(dV,"all"),X,EnableHigherDerivatives=true);

res1=grad_dV;
target1=zeros(size(res1));
lossF1=l2loss(res1,target1);
lossF=lossF1;

%Enforce boundary conditions, calculate lossU

%electric potential
[Bdout]=forward(net,BdNodes);
V_bd=Bdout(1,:);
res2=V_bd-BdCond;
target2=zeros(size(res2));
lossU1=l2loss(res2,target2);


lossU=lossU1;

% Combine the losses
lambda=0.4;
loss=lambda*lossF+(1-lambda)*lossU;
gradients=dlgradient(loss,net.Learnables);
end
```

FIGURE 4.4: Loss function

4.2.3 Training results and network testing

Once the training is completed, we can plot the network output on the mesh nodes and compare it with the analytical solution of the problem. For a one-dimensional capacitor with constant permittivity, the potential distribution is:

$$v(x) = V_0 + \frac{x}{L}(V_L - V_0). \tag{4.7}$$

After 100 training epochs, the loss function reached a value of $3.8224 \cdot 10^{-5}$ after 9s of training. The small runtime is due to the fact that the training points were few and so the calculations took little time to be performed. The L_2 norm of the error vector between the PINN and the analytical solution is $0.0154V$ (see (3.11) for the formula). In Figure 4.6 we can see that the PINN solution is very accurate.

The next step of our work will be the introduction of a space-varying permittivity coefficient. This small modification will pose a series of problems that will need to be properly addressed.

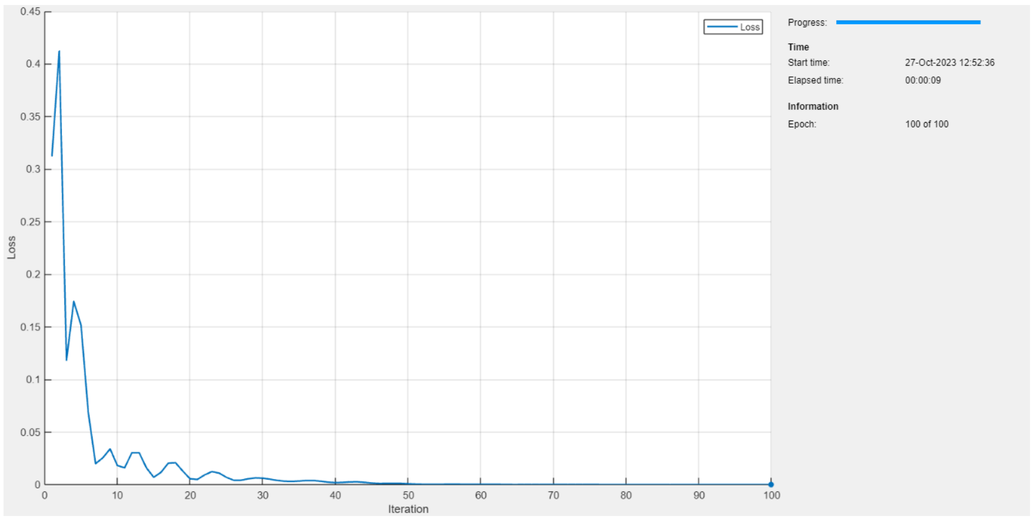


FIGURE 4.5: One-dimensional capacitor, training progress

4.3 One-dimensional capacitor with non-homogeneous permittivity

4.3.1 Problem description

In Section 4.2 we have seen PINNs applied to a very simple one dimensional electrostatic problem. To make a step further, we introduce a variation to the settings of that problem. Now we consider a relative permittivity inside the dielectric region that varies according to the following law:

$$\varepsilon_r = 1 + \frac{x}{L}. \tag{4.8}$$

The first difference is that now the permittivity coefficient in the PDE needs to be defined as a vector, that we will name $\boldsymbol{p}$, of the same length as the number of collocation points. Each component of this vector will be the value of the permittivity on the corresponding collocation point according to (4.8). All the other data remain the same as in Section 4.2.

4.3.2 The problem of non-homogeneous media

4.3.2.1 Interfaces

The first challenge posed by a non-homogeneous medium, i.e. a medium whose parameters vary in space, is how to deal with interfaces between the regions that present different values of the material parameter (permittivity, in the case of the parallel plate capacitor). Until now, we have taken the nodes of the mesh as collocation points. However, if interfaces are present, the nodes on them would belong to two (or more) regions at the same time, creating ambiguity in the coefficient value assignment. To avoid this ambiguity, we can use the centres of the elements of the mesh, instead of the nodes, as collocation points. Since elements of a mesh, constructed using the FEM prescriptions, cannot belong to two regions at the same time, we have eliminated the problem of one collocation point belonging to different regions and, if the mesh is fine enough, we will also have a good approximation of the solution at interfaces. In Section 4.3.3.2 we will see how this change is implemented in practice.

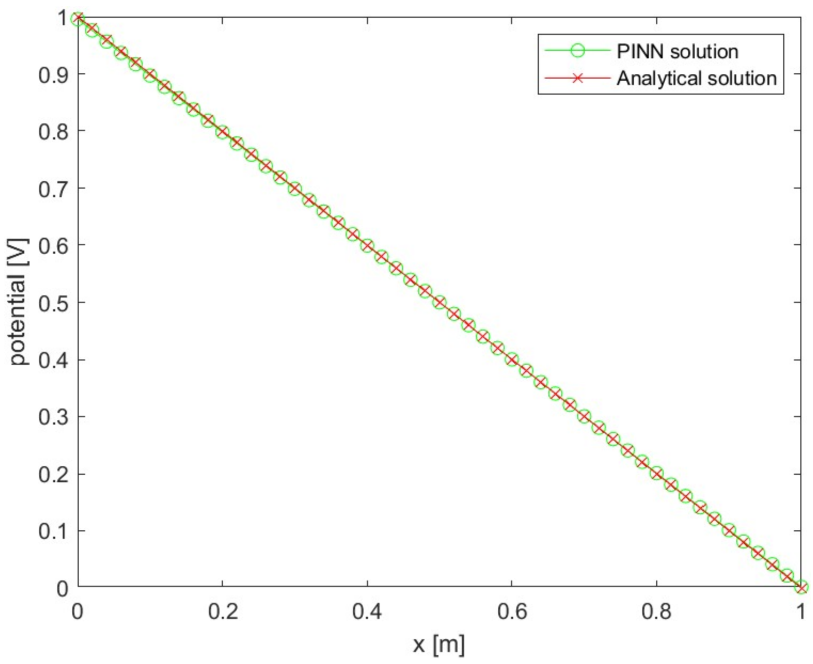


FIGURE 4.6: One-dimensional capacitor, potential distribution, PINN vs. analytical

4.3.2.2 Differentiation

Another problem concerning non-homogeneous media is that, if the loss function of the problem is constructed as in (2.9), one has to compute the spacial derivative of the material parameter. Unless the parameter variation is described by a function in closed form, AD is not able to compute its derivatives (see Section 2.3 for details). The only alternative is using *finite differences*, but, when moving from one region to another, we may encounter a sharp change in the value of the material parameter. This leads to large derivatives and has a detrimental impact on the stability of the network training [10] and subsequently on the accuracy of the final solution.

To solve the aforementioned problem, Gong et al., in [10], propose a different way to define the loss function. They consider a magnetostatic problem, but their approach can be easily transposed to electrostatics. In Chapter 2, we have seen the PDE formulations of electrostatic and magnetostatic problems. Both can be described by a Poisson's equation (2.28) in terms of scalar potential. In the approach proposed in [10] Maxwell's equations are not combined together: for instance, for electrostatics, Gauss' equation is not combined to the constitutive equation for dielectrics; in this way, the field problem is formulated in terms of electric potential and electric displacement field:

$$-\varepsilon\nabla V = \boldsymbol{D} \tag{4.9}$$

$$\nabla\cdot\boldsymbol{D} = \rho. \tag{4.10}$$

In one dimension, (4.9) and (4.10) become:

$$-\varepsilon\frac{\partial V}{\partial x} = D \tag{4.11}$$

$$\frac{\partial D}{\partial x} = \rho. \tag{4.12}$$

We see that, in (4.11), the permittivity is not differentiated. If we construct a neural network with two outputs (approximating V and D) instead of just one, we can train it using (4.11) and (4.12) to define the loss function. This way, instead of one residual for $lossF$ and one residual for $lossU$, we will have multiple residuals to combine. On the i-th collocation point we will have:

$$res_{1,i} = -\varepsilon_i \frac{\partial V_{PINN,i}}{\partial x} - D_{PINN,i} \tag{4.13}$$

$$res_{2,i} = \frac{\partial D_{PINN,i}}{\partial x} - \rho \tag{4.14}$$

for $lossF$, where $V_{PINN,i}$, $D_{PINN,i}$ and ε_i are the network outputs and the permittivity value on the i-th collocation point, and, on the j-th boundary point,

$$res_{3,j} = V_{bd,j} - V_{Dir,j} \tag{4.15}$$

for $lossU$, where $V_{bd,j}$ is the first network output on the j-th boundary point and $V_{Dir,j}$ are the corresponding Dirichlet condition. We can now build the loss function as:

$$lossF = \lambda_1 lossF_1 + \lambda_2 lossF_2 \tag{4.16}$$

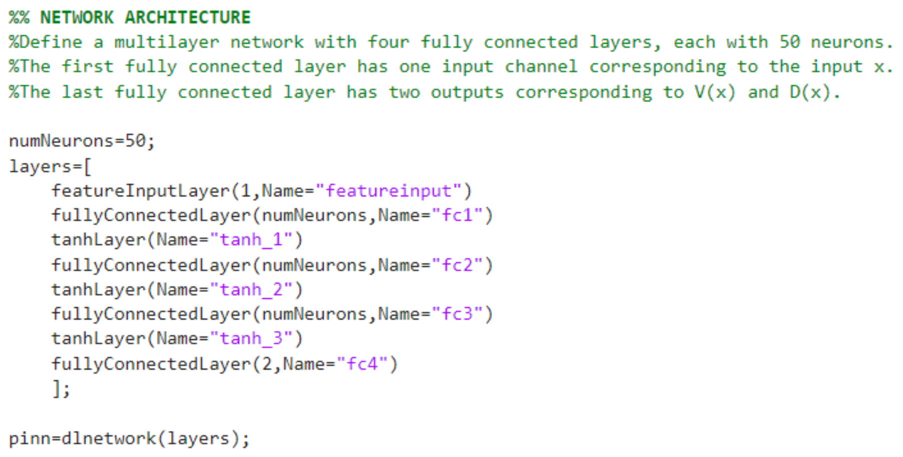
$$loss = \gamma_1 lossF + \gamma_2 lossU, \tag{4.17}$$

where the $lossF_k$ in (4.16) are computed as in (4.3) and the $lossU$ in (4.17) as in (4.5).

4.3.3 Training process

4.3.3.1 Network definition

In Section 4.3.2 we have discussed a new approach to the loss function definition, which requires more than one network output. For a 1D problem, we need a network with two output channels, one corresponding to the electric potential V and the other to the electric displacement field D, which in this case has only the x-component. To do this, we need to add one neuron to the output layer of the network. Figure 4.7 shows this change implemented in MATLAB.



```
%% NETWORK ARCHITECTURE
%Define a multilayer network with four fully connected layers, each with 50 neurons.
%The first fully connected layer has one input channel corresponding to the input x.
%The last fully connected layer has two outputs corresponding to V(x) and D(x).

numNeurons=50;
layers=[
    featureInputLayer(1,Name="featureinput")
    fullyConnectedLayer(numNeurons,Name="fc1")
    tanhLayer(Name="tanh_1")
    fullyConnectedLayer(numNeurons,Name="fc2")
    tanhLayer(Name="tanh_2")
    fullyConnectedLayer(numNeurons,Name="fc3")
    tanhLayer(Name="tanh_3")
    fullyConnectedLayer(2,Name="fc4")
    ];

pinn=dlnetwork(layers);
```

FIGURE 4.7: Definition of a neural network with two output channels

4.3.3.2 Training points choice and training options

The first important modification to the algorithm, besides the two-output network, is in the choice of collocation points. To implement what we discussed in Section 4.3.2, we take the centres of the mesh elements as collocation points. This is done by looping through all the elements, computing the mean between the two nodes of each element and putting it into a vector of the same size as the number of elements, which is 50. The vector is then converted and formatted using *dlarray*. We take the first and the last node of the mesh as boundary points.

Once the training points have been chosen, we can move on to the training options setting. Since we introduced the space varying permittivity, the complexity of the problem is a bit higher and thus it is best to perform a bigger number of training epochs with respect to the case with constant permittivity. The training process hyperparamters are:

- $n_{epochs} = 300$: number of training epochs,
- $batchsize = 50$: number of points in each batch,
- $\alpha_i = 0.01$: initial learning rate,
- $\beta = 0.005$: learning rate decay.

4.3.3.3 Training of the network

Having set the training options, we can proceed with the network training. To implement what Section 4.3.2 suggests, we need to modify the MATLAB function that computes the loss function value and its gradient. This time we have two network outputs on collocation points, V_{PINN} and D_{PINN}, so we need to define three loss function terms, the first two to enforce the PDEs on the collocation points and the other one to enforce the boundary conditions. The first one comes from (4.9):

$$res_{1,i} = -p_i \frac{\partial V_{PINN,i}}{\partial x} - D_{PINN,i} \tag{4.18}$$

$$lossF_1 = MSE(\boldsymbol{res}_1, \boldsymbol{target}_1), \tag{4.19}$$

where p_i is the component of the vector $\boldsymbol{p}$ (see Section 4.3.1) corresponding to the i-th collocation point, $\boldsymbol{res}_1$ is a vector containing all the $res_{1,i}$ terms and $\boldsymbol{target}_1$ is a vector of zeros of the same size as the number of collocation points. The second term of the loss function is defined from (4.10). In this case $\rho = 0$, so we end up with

$$res_{2,i} = \frac{\partial D_{PINN,i}}{\partial x} \tag{4.20}$$

$$lossF_2 = MSE(\boldsymbol{res}_2, \boldsymbol{target}_2), \tag{4.21}$$

where $\boldsymbol{res}_2$ is a vector containing all the $res_{2,i}$ terms and $\boldsymbol{target}_2$ is a vector of zeros of the same size as the number of collocation points. The derivatives in (4.18) and (4.20) are computed using *dlgradient*. The loss function term related to the PDE is then computed as:

$$lossF = 0.5 lossF_1 + 0.5 lossF_2. \tag{4.22}$$

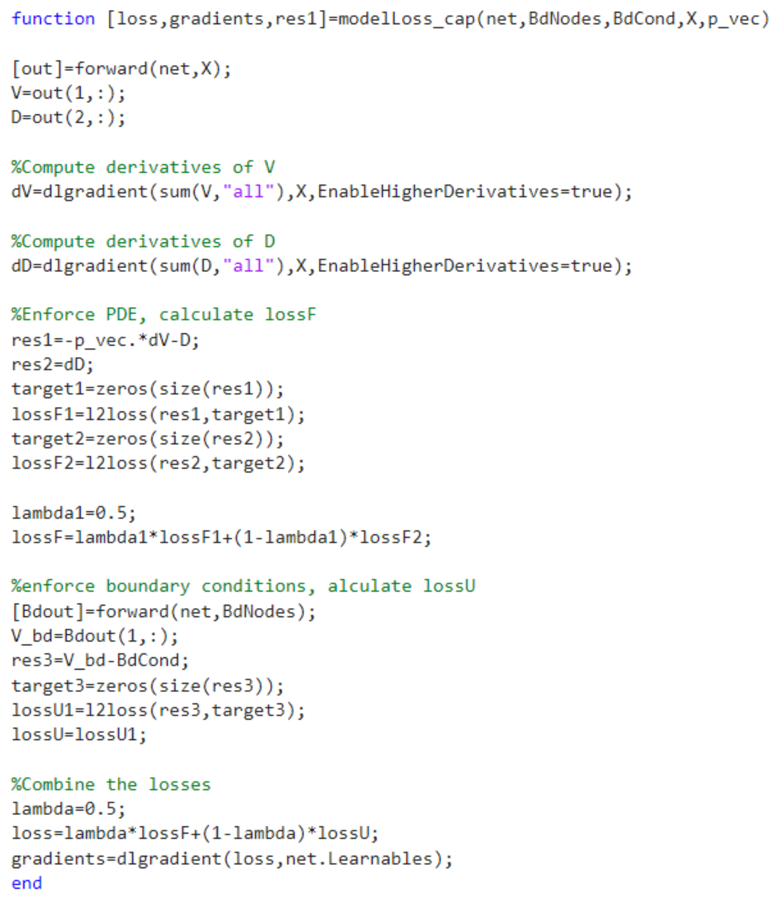
Now we need to define the two terms pertaining the boundary conditions on V and D. The conditions on V are simply $V(x = 0) = 1$ and $V(x = L) = 0$. The boundary conditions on D come from (4.9). In a 2D domain, if we have Dirichlet conditions on V, it means that $\frac{\partial V}{\partial t} = 0$ on the boundary, leading to $D_t = \varepsilon \frac{\partial V}{\partial t} = 0$. In this case the domain is 1D, so we only have the normal component $D_n = D_x$, meaning that we actually don't need to enforce any boundary condition on D. This leaves us with only one loss function term for the BCs, defined as in (4.4):

$$res_{3,j} = V_{bd,j} - BdCond_j \tag{4.23}$$
$$lossU = MSE(\boldsymbol{res}_3, \boldsymbol{target}_3), \tag{4.24}$$

where $V_{bd,j}$ and $BdCond_j$ are the network output and the boundary condition on the j-th boundary point and $\boldsymbol{target}_3$ is a vector of zeros of the same size as the number of boundary points. The total loss function is then obtained combining $lossF$ and $lossU$:

$$loss = 0.5lossF + 0.5lossU. \tag{4.25}$$

Its gradient with respect to the network parameters is computed using *dlgradient*.



```
function [loss,gradients,res1]=modelLoss_cap(net,BdNodes,BdCond,X,p_vec)

[out]=forward(net,X);
V=out(1,:);
D=out(2,:);

%Compute derivatives of V
dV=dlgradient(sum(V,"all"),X,EnableHigherDerivatives=true);

%Compute derivatives of D
dD=dlgradient(sum(D,"all"),X,EnableHigherDerivatives=true);

%Enforce PDE, calculate lossF
res1=-p_vec.*dV-D;
res2=dD;
target1=zeros(size(res1));
lossF1=l2loss(res1,target1);
target2=zeros(size(res2));
lossF2=l2loss(res2,target2);

lambda1=0.5;
lossF=lambda1*lossF1+(1-lambda1)*lossF2;

%enforce boundary conditions, alculate lossU
[Bdout]=forward(net,BdNodes);
V_bd=Bdout(1,:);
res3=V_bd-BdCond;
target3=zeros(size(res3));
lossU1=l2loss(res3,target3);
lossU=lossU1;

%Combine the losses
lambda=0.5;
loss=lambda*lossF+(1-lambda)*lossU;
gradients=dlgradient(loss,net.Learnables);
end
```

FIGURE 4.8: Modified loss function

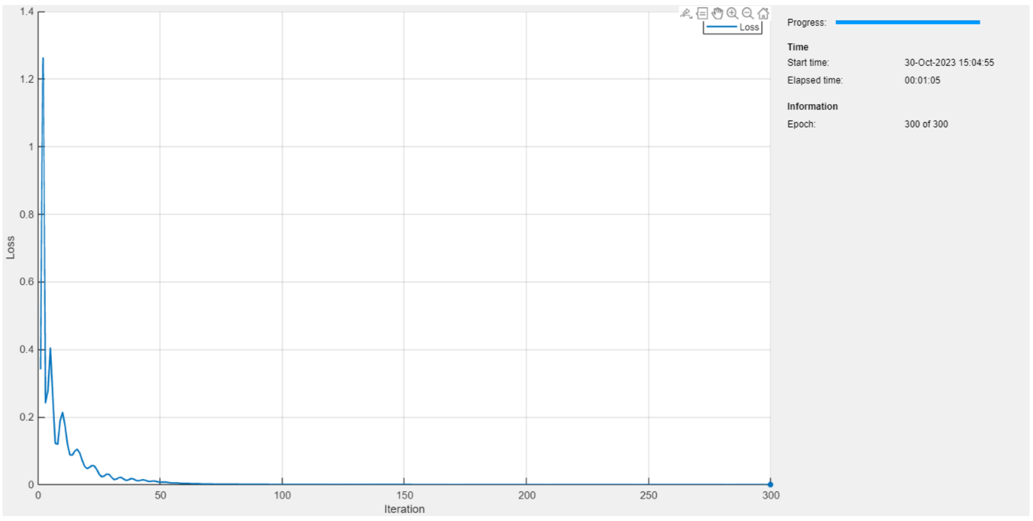
Figure 4.8 shows the modified MATLAB function that computes the loss function and its gradient. The rest of the training process is the same as described in Section 3.3.3.

4.3.4 Training results and network testing

Once the training is complete, we can test the results. After 300 epochs the loss function reached a value of $1.1975 \cdot 10^{-4}$. The training process took a total of 1 minute and 5 seconds. We note that the runtime is much higher with respect to the case with constant permittivity, due the higher number of epochs and the slightly more complex loss function. The PINN solution can be compared to the analytical one:

$$V(x) = V_0 + \frac{V_L - V_0}{\log(2)} \cdot \log(1 + \frac{x}{L}) \tag{4.26}$$

The L_2 norm of the error vector between the analytical and the PINN solution is 0.0075. Figure 4.9 shows the loss function evolution in time while Figure 4.10 shows the comparison between the two solutions evaluated on the mesh nodes. We can see that the PINN manages to approximate the real solution quite well with a fairly small number of training epochs.



Progress:

Time
Start time: 30-Oct-2023 15:04:55
Elapsed time: 00:01:05

Information
Epoch: 300 of 300

FIGURE 4.9: One dimensional capacitor with varying permittivity, training progress

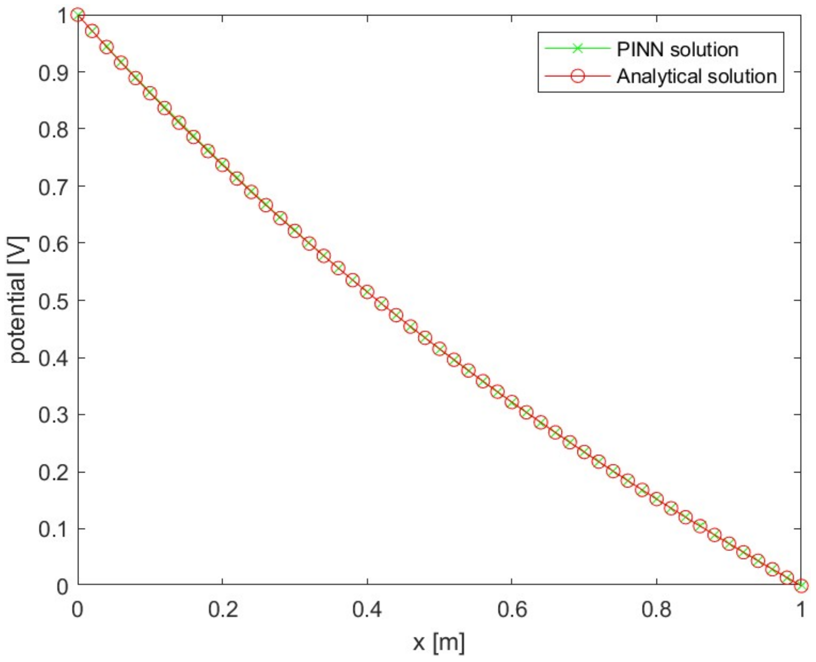


FIGURE 4.10: One dimensional capacitor with varying permittivity, potential distribution, PINN vs. analytical

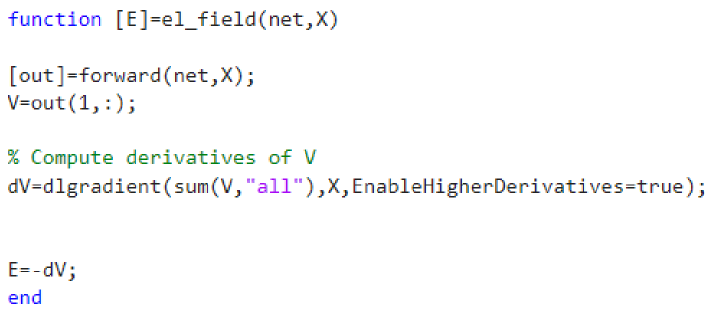
To further test the accuracy of the solution, we can exploit the differentiability of the PINN output (see Section 1.4) to compute the electric field inside the capacitor. At this purpose a MATLAB function that exploits *dlgradient* for automatic differentiation has been developed. It allows to compute the electric field from the scalar potential as *dlgradient* do compute E as:

$$E = -\frac{\partial V}{\partial x}. \tag{4.27}$$

If we compute E on the mesh nodes, we can compare it with the analytical solution derived from (4.26):

$$E(x) = \frac{V_0 - V_L}{\log(2)} \cdot \frac{1}{x+L}. \tag{4.28}$$

In figure 4.11 we see the function used to compute the electric field, while figure 4.12 shows the comparison between the electric field computed via (4.28) and the one computed differentiating the PINN solution via AD. The L_2 norm of the error vector between the two solutions, computed as in (3.11), is 0.1365, which confirms the good accuracy of the PINN solution.



```
function [E]=el_field(net,X)

[out]=forward(net,X);
V=out(1,:);

% Compute derivatives of V
dV=dlgradient(sum(V,"all"),X,EnableHigherDerivatives=true);


E=-dV;
end
```

FIGURE 4.11: Function to compute the electric field

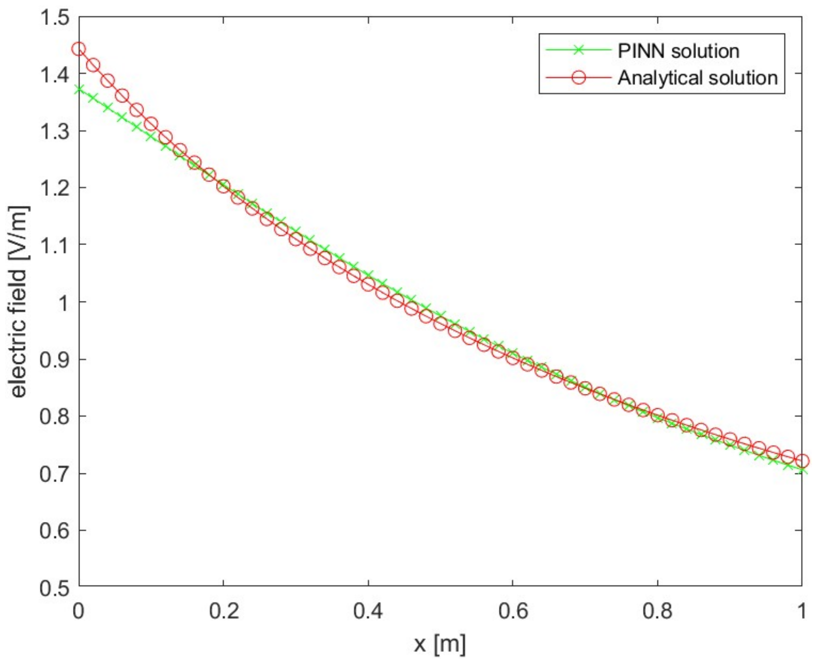


FIGURE 4.12: One dimensional capacitor with varying permittivity, electric field, PINN vs. analytical

4.3.5 Old vs. new approach

To check if the new approach to the loss function definition actually improved the performance of the algorithm, we can redo the training keeping the same settings but using the "old" approach, i.e. the one introduced in the previous chapter and adapted to the 1D capacitor in Section 4.2.2.3, with the only difference that instead of a constant permittivity value we need to consider a varying permittivity in (4.2). Figure 4.13 shows the results compared to the an-

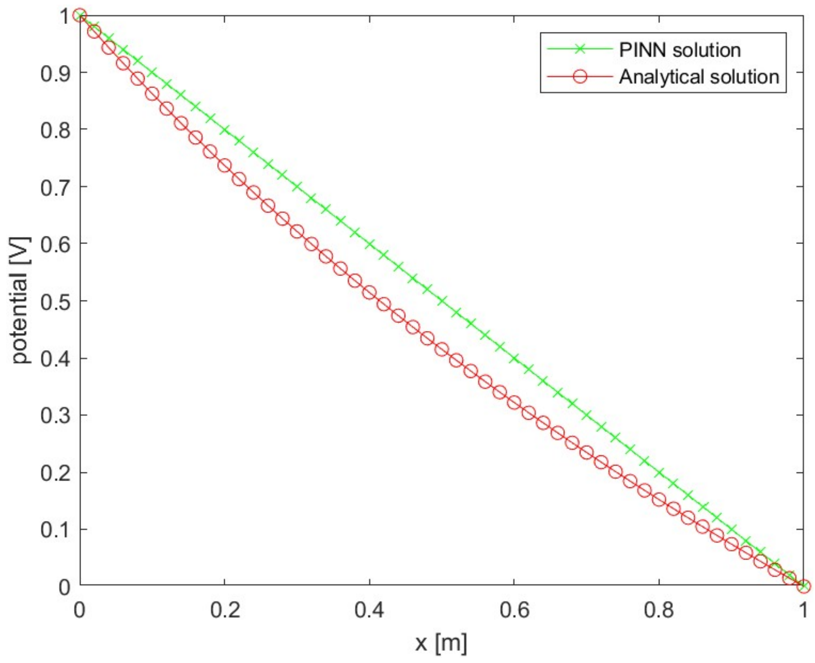
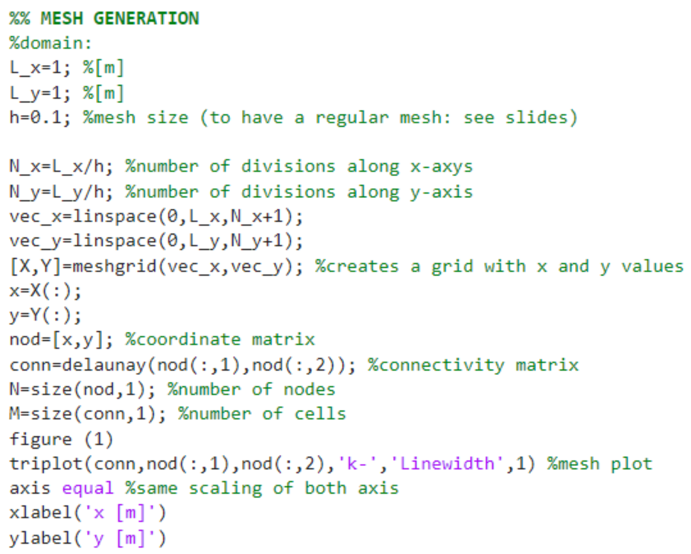


FIGURE 4.13: One dimensional capacitor with varying permittivity, PINN vs. analytical, old approach

alytical solution. We can see that the PINN was not able to approximate the variation of the permittivity and the solution looks like the one obtained with a constant permittivity value. This is due to the fact that AD, as already mentioned in Section 4.3.2.2, is not able to compute the derivatives of the permittivity, because the permittivity is passed to the *modelLoss* function as a vector of constant coefficients, while AD can only differentiate functions expressed in closed form or by an algorithm (see Section 2.3). In this particular case, since ε is defined by an expression in closed form (see (4.8)), we could solve this problem by defining the permittivity vector inside the *modelLoss* MATLAB function instead of passing it to the function as an input, allowing AD to compute its derivatives. However, in the next Sections, we will present some cases in which this is not possible. This means that the approach proposed in Section 4.3 allows us to treat problems that would be impossible to solve using the algorithm discussed in the previous chapter, improving the generality of the code.

4.4 Extension to a two dimensional domain

Until now, we have considered a one-dimensional domain. To see if the approach proposed in Section 4.3 can be applied to a two-dimensional domain, we can treat the problem proposed in Section 4.3 as a 2D problem. Figure 4.15 shows the 2D domain configuration. We neglect the edge effects in this case too. If the approach works, we should obtain the same solution as in the 1D case.



```
%% MESH GENERATION
%domain:
L_x=1; %[m]
L_y=1; %[m]
h=0.1; %mesh size (to have a regular mesh: see slides)

N_x=L_x/h; %number of divisions along x-axys
N_y=L_y/h; %number of divisions along y-axis
vec_x=linspace(0,L_x,N_x+1);
vec_y=linspace(0,L_y,N_y+1);
[X,Y]=meshgrid(vec_x,vec_y); %creates a grid with x and y values
x=X(:);
y=Y(:);
nod=[x,y]; %coordinate matrix
conn=delaunay(nod(:,1),nod(:,2)); %connectivity matrix
N=size(nod,1); %number of nodes
M=size(conn,1); %number of cells
figure (1)
triplot(conn,nod(:,1),nod(:,2),'k-','Linewidth',1) %mesh plot
axis equal %same scaling of both axis
xlabel('x [m]')
ylabel('y [m]')
```

FIGURE 4.14: 2D mesh generation

4.4.1 Domain meshing and training points choice

The first step is to mesh the domain to obtain collocation and boundary points. We define two vectors of equally spaced points, between 0 and 1, that represent the divisions along the x and y axes. We choose 0.1 m as the edge size for the mesh. We use the function *meshgrid* to build a grid whose nodes correspond to the nodes of the mesh, then the function *delaunay* (a function that creates a Delaunay triangulation from the points in the grid [29]) to build the connectivity matrix. Figure 4.14 shows the code used to create the mesh and Figure 4.15 shows the meshed domain.

As suggested in Section 4.3, we choose the centres of the mesh elements as collocation points. To do so, we loop through all the elements and we take the mean value of the x and y coordinates of their nodes, putting them inside a $M \times 2$ matrix, where M is the number of elements. The number of collocation points that we obtain is $M = 200$. On the plates of the capacitor, where the potential is constant, Dirichlet conditions are imposed, while on the sides of the dielectric region Neumann conditions are imposed (see Figure 4.15), so that the electric field is tangent to the Neumann boundary. To find the indexes of the elements of the coordinate matrix corresponding to the boundary nodes, we use the MATLAB function *convhull*, that computes the convex hull of the matrix. Once we have the indexes, we can find the coordinates of the nodes belonging to each edge of the boundary. Figure 4.17 shows the details of the procedure.

4.4.2 Loss function definition

Since we are now in a 2D setting, we need to make some changes to the MATLAB function that computes the loss function value. Now we have two components of the electric displacement field, so we need to add one term to *lossF*. Moreover, since we also have Dirichlet and Neumann conditions, we need to add three terms to *lossU*, one for the Neumann conditions on the electric potential and two for the Dirichlet and Neumann conditions on the electric displacement field. The three-component network output $(V_{PINN}, D_{x,PINN}, D_{y,PINN})$ must fulfill (4.9) and (4.10) on collocation points. In our case, since $\rho = 0$, the residuals on the i-th collocation

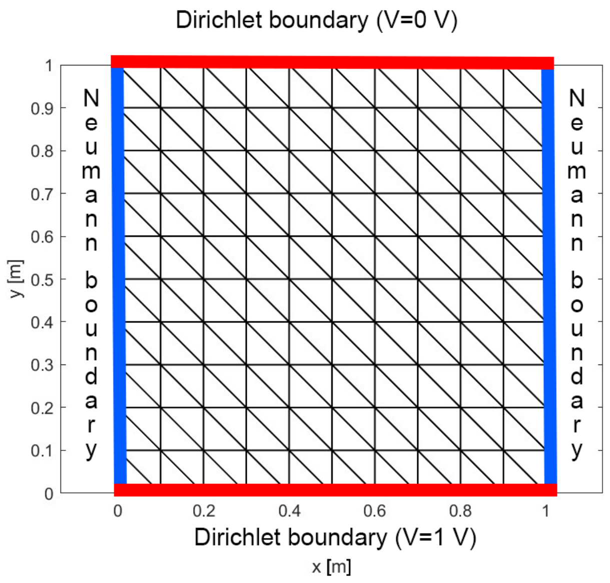
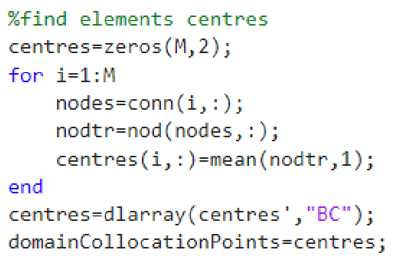
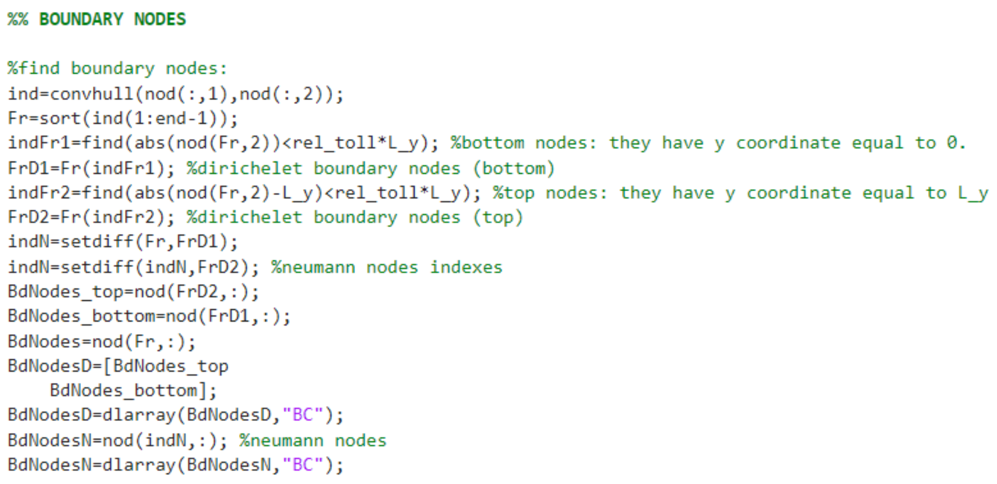


FIGURE 4.15: 2D meshed domain



```
%find elements centres
centres=zeros(M,2);
for i=1:M
    nodes=conn(i,:);
    nodtr=nod(nodes,:);
    centres(i,:)=mean(nodtr,1);
end
centres=dlarray(centres',"BC");
domainCollocationPoints=centres;
```

FIGURE 4.16: Definition of the collocation points



```
%% BOUNDARY NODES

%find boundary nodes:
ind=convhull(nod(:,1),nod(:,2));
Fr=sort(ind(1:end-1));
indFr1=find(abs(nod(Fr,2))<rel_toll*L_y); %bottom nodes: they have y coordinate equal to 0.
FrD1=Fr(indFr1); %dirichelet boundary nodes (bottom)
indFr2=find(abs(nod(Fr,2)-L_y)<rel_toll*L_y); %top nodes: they have y coordinate equal to L_y
FrD2=Fr(indFr2); %dirichelet boundary nodes (top)
indN=setdiff(Fr,FrD1);
indN=setdiff(indN,FrD2); %neumann nodes indexes
BdNodes_top=nod(FrD2,:);
BdNodes_bottom=nod(FrD1,:);
BdNodes=nod(Fr,:);
BdNodesD=[BdNodes_top
    BdNodes_bottom];
BdNodesD=dlarray(BdNodesD,"BC");
BdNodesN=nod(indN,:); %neumann nodes
BdNodesN=dlarray(BdNodesN,"BC");
```

FIGURE 4.17: Identification of boundary nodes

point are computed as

$$res_{1,i} = -p_i \frac{\partial V_{PINN,i}}{\partial x} - D_{x,PINN,i} \tag{4.29}$$

$$res_{2,i} = -p_i \frac{\partial V_{PINN,i}}{\partial y} - D_{y,PINN,i} \tag{4.30}$$

$$res_{3,i} = (\frac{\partial D_{x,PINN,i}}{\partial x} + \frac{\partial D_{y,PINN,i}}{\partial y}). \tag{4.31}$$

On the j-th Dirichlet node we have two residuals to minimize:

$$res_{4,j} = V_{bd,j} - BdCond_j \tag{4.32}$$

$$res_{5,j} = D_{bd,x,j}, \tag{4.33}$$

where$V_{bd,j}$ and $BdCond_j$ are the network output corresponding to the electric potential and the boundary condition on the j-th Dirichlet node and $D_{bd,x,j}$ is the network output corresponding to the electric displacement field component tangent to the Dirichlet boundary on the j-th Dirichlet node. On the k-th Neumann node we have two residuals to minimize, related to Neumann conditions on V and D:

$$res_{6,k} = \frac{\partial V_{bdN,k}}{\partial x} \tag{4.34}$$

$$res_{7,k} = D_{bdN,x,k}, \tag{4.35}$$

$$\tag{4.36}$$

where $V_{bdN,k}$ and $D_{bdN,x,k}$ are the network output components on k-th Neumann node. Each loss function component is then defined as in (4.3):

$$loss_n = MSE(\boldsymbol{res}_n, \boldsymbol{target}_n). \tag{4.37}$$

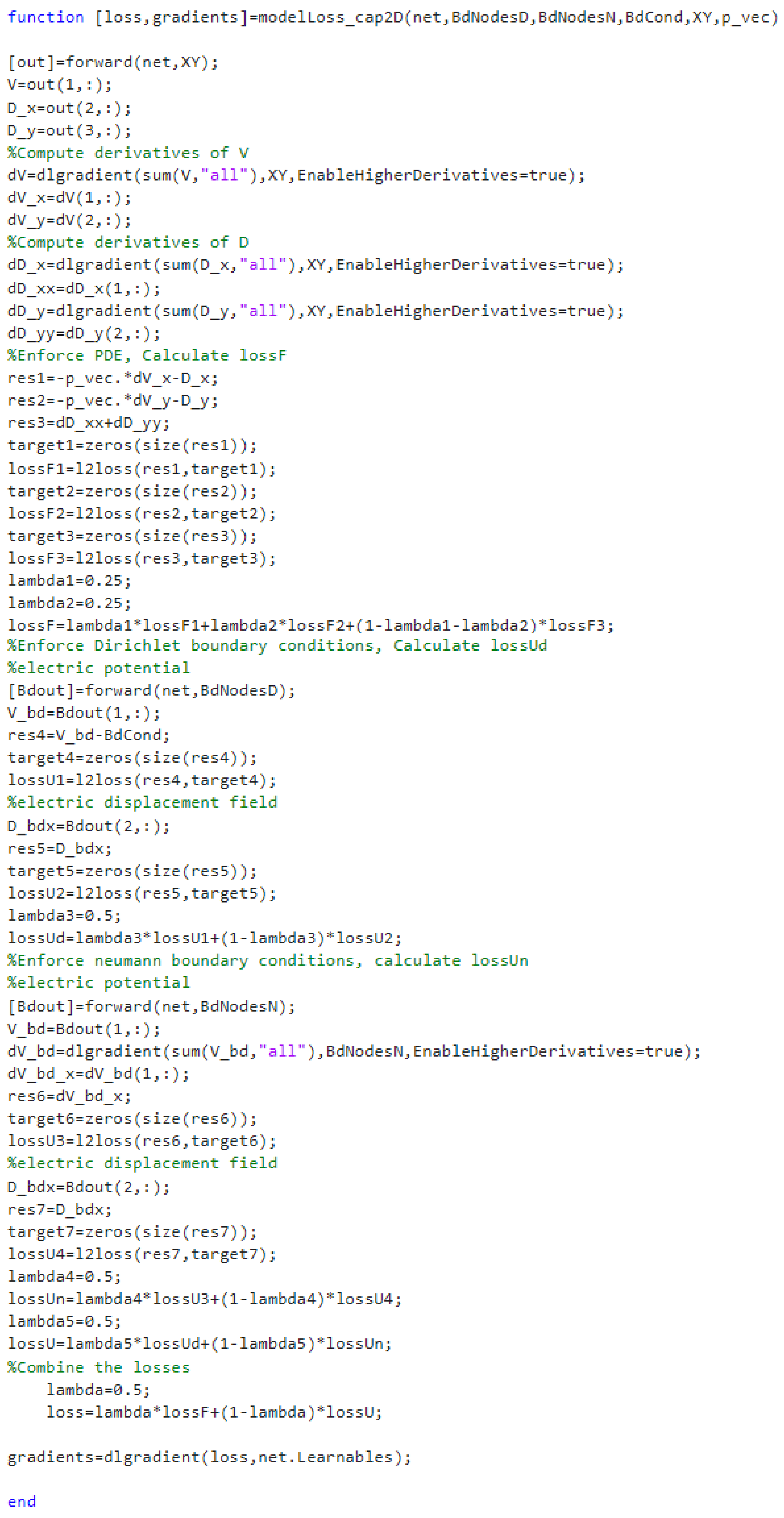
The complete loss function is then a linear combination of all the $loss_n$ terms. Figure 4.18 shows the MATLAB function used to compute the loss function and its gradient.

4.4.3 Training process and results

We choose the following training options:

- $n_{epochs} = 300$: number of training epochs,
- $batchsize = 200$: number of points in each batch,
- $\alpha_i = 0.01$: initial learning rate,
- $\beta = 0.005$: learning rate decay.

After 300 epochs of training, for a total training time of 1 minute and 28 seconds, the loss function has reached a value of $1.9226 \cdot 10^{-04}$. We can test if the approach was successful by comparing the potential distribution along the y-axis with the one obtained in the 1D case. To test the accuracy of the solution, we can plot the potential distribution along the y-axis on 50 equally spaced points for three different values of the x-coordinate ($x = 0$ m, $x = 0.5$ m and $x = 1$ m). If the training is carried out properly,the potential distributions obtained from these three x-coordinates must be approximately the same. We can also compare them with the analytical solution of the 1D case to see if they are accurate. Figure 4.19 shows the results.



```
function [loss,gradients]=modelLoss_cap2D(net,BdNodesD,BdNodesN,BdCond,XY,p_vec)

[out]=forward(net,XY);
V=out(1,:);
D_x=out(2,:);
D_y=out(3,:);
%Compute derivatives of V
dV=dlgradient(sum(V,"all"),XY,EnableHigherDerivatives=true);
dV_x=dV(1,:);
dV_y=dV(2,:);
%Compute derivatives of D
dD_x=dlgradient(sum(D_x,"all"),XY,EnableHigherDerivatives=true);
dD_xx=dD_x(1,:);
dD_y=dlgradient(sum(D_y,"all"),XY,EnableHigherDerivatives=true);
dD_yy=dD_y(2,:);
%Enforce PDE, Calculate lossF
res1=-p_vec.*dV_x-D_x;
res2=-p_vec.*dV_y-D_y;
res3=dD_xx+dD_yy;
target1=zeros(size(res1));
lossF1=l2loss(res1,target1);
target2=zeros(size(res2));
lossF2=l2loss(res2,target2);
target3=zeros(size(res3));
lossF3=l2loss(res3,target3);
lambda1=0.25;
lambda2=0.25;
lossF=lambda1*lossF1+lambda2*lossF2+(1-lambda1-lambda2)*lossF3;
%Enforce Dirichlet boundary conditions, Calculate lossUd
%electric potential
[Bdout]=forward(net,BdNodesD);
V_bd=Bdout(1,:);
res4=V_bd-BdCond;
target4=zeros(size(res4));
lossU1=l2loss(res4,target4);
%electric displacement field
D_bdx=Bdout(2,:);
res5=D_bdx;
target5=zeros(size(res5));
lossU2=l2loss(res5,target5);
lambda3=0.5;
lossUd=lambda3*lossU1+(1-lambda3)*lossU2;
%Enforce neumann boundary conditions, calculate lossUn
%electric potential
[Bdout]=forward(net,BdNodesN);
V_bd=Bdout(1,:);
dV_bd=dlgradient(sum(V_bd,"all"),BdNodesN,EnableHigherDerivatives=true);
dV_bd_x=dV_bd(1,:);
res6=dV_bd_x;
target6=zeros(size(res6));
lossU3=l2loss(res6,target6);
%electric displacement field
D_bdx=Bdout(2,:);
res7=D_bdx;
target7=zeros(size(res7));
lossU4=l2loss(res7,target7);
lambda4=0.5;
lossUn=lambda4*lossU3+(1-lambda4)*lossU4;
lambda5=0.5;
lossU=lambda5*lossUd+(1-lambda5)*lossUn;
%Combine the losses
    lambda=0.5;
    loss=lambda*lossF+(1-lambda)*lossU;

gradients=dlgradient(loss,net.Learnables);

end
```

FIGURE 4.18: Loss function for the 2D problem

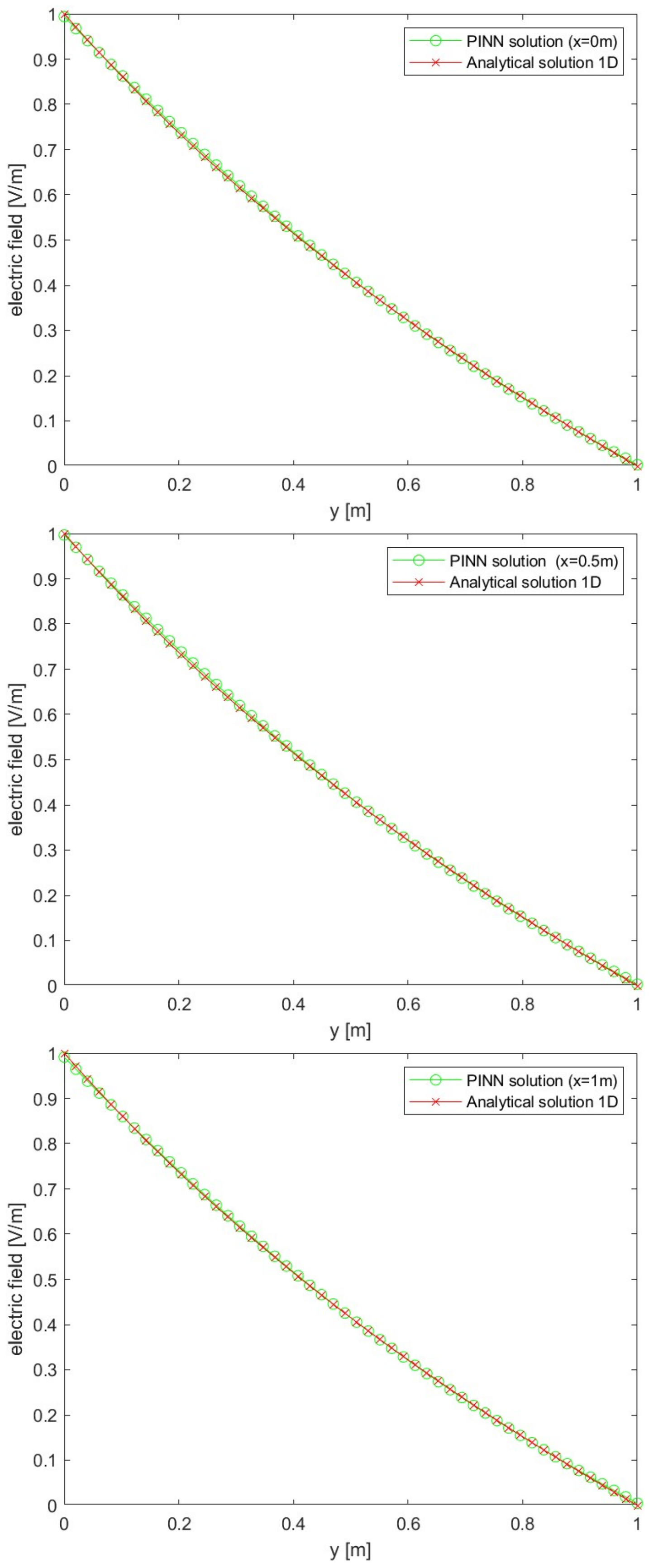


FIGURE 4.19: Two-dimensional capacitor, potential distribution, PINN vs. analytical

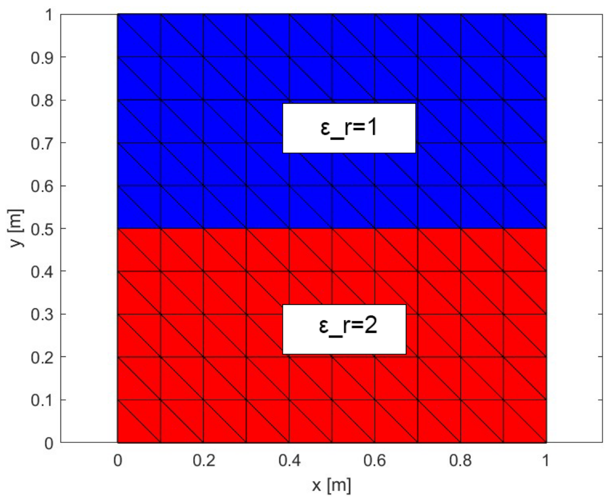


FIGURE 4.20: Two-region domain and mesh

We can see that in all the three cases the PINN solution is accurate compared to the analytical one. To validate the graphical results, we can check the L_2 norm of the error vectors. For $x = 0$ m, $L_2(E) = 0.0203$, for $x = 0.5$ m, $L_2(E) = 0.0224$ and for $x = 1$ m, $L_2(E) = 0.0183$. All the norms are small and in the same order of magnitude.

We can conclude that the proposed approach works for both 1D and 2D electrostatic problems with non-homogeneous permittivity. In the next Section we will see if it also works when we have a permittivity that is not a continuous function (unlike (4.8)), but has a constant value in one region of the dielectric and another constant value in another region.

4.5 Parallel plate capacitor with two dielectric regions

4.5.1 Problem description

To test the generality of the approach proposed in Section 4.3 and 4.4, we now consider a 2D capacitor with two dielectric regions with different values of the permittivity. The setting remains the same as in Section 4.5, but now we have $\varepsilon_r = 2$ from $y = 0$m to $y = 0.5$m and $\varepsilon_r = 1$ from $y = 0.5$m to $y = 1$m. The domain meshing can be carried out in the exact same way as we did in the previous case. Figure 4.20 shows the meshed domain and highlights the two regions.

4.5.2 Training points and loss function definition

The collocation and boundary points can be chosen exactly as in Section 4.4.1. However, we need to define two additional sets of points to account for the interface between the two regions. On these points we will enforce interface conditions. The interface points will be the centres of the elements that have at least one node tangent to the interface between the two regions. To do so we need to find those elements in the two regions whose centre is less than 0.1m (mesh edge size) from the interface. Once we have found the interface points belonging to each region (we need two separate arrays, one for the interface nodes belonging to the region with $\varepsilon_r = 1$ and one for those belonging to the region with $\varepsilon_r = 2$), we need to remove them from the collocation points array (because they are already constrained nodes). One more thing to do before proceeding to the loss function definition, is to divide the collocation points in two batches, one with the points belonging to the first region and one with the points belonging to the second region. We will call these two arrays $\boldsymbol{XY}_1$ and $\boldsymbol{XY}_2$.

Now it is time to modify our MATLAB function to compute the loss function and its gradient. The idea to solve this kind of problem is to enforce (4.9) and (4.10) on the collocation points of the two regions separately, and then combine the two solutions by imposing an interface condition. In this case we can enforce the continuity of the electric displacement field component normal to the interface (D_y in this case). This new condition requires the addition of a new term to the loss function:

$$res_{int,i} = D_{y,int1,i} - D_{y,int2,i}, \tag{4.38}$$

$$loss_{int} = MSE(\boldsymbol{res}_{int}, \boldsymbol{target}_{int}), \tag{4.39}$$

where $D_{y,int1,i}$ and $D_{y,int2,i}$ are the y-components of the D field evaluated on the i-th couple of interface points (interface points that have the same x-coordinate but belong one to region 1 and one to region 2), $\boldsymbol{res}_{int}$ is a vector containing all the $res_{int,i}$ terms and $\boldsymbol{target}_{int}$ is a vector of zeros of the same size as $\boldsymbol{res}_{int}$. Figure 4.21 shows the bit of code that enforces the interface condition in the MATLAB function. nod_int1 and nod_int2 are the interface nodes in the two regions.

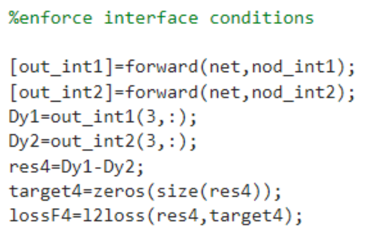

```
%enforce interface conditions

[out_int1]=forward(net,nod_int1);
[out_int2]=forward(net,nod_int2);
Dy1=out_int1(3,:);
Dy2=out_int2(3,:);
res4=Dy1-Dy2;
target4=zeros(size(res4));
lossF4=l2loss(res4,target4);
```

FIGURE 4.21: Enforcement of the interface condition in the loss function algorithm

Boundary conditions will be enforced in the same way as we did in the previous example. The final loss function will be the linear combination of the loss functions related to the PDE in the two regions, the loss function related to the interface condition and the loss functions related to the boundary conditions.

4.5.3 Training results and comparison with FEM

The increased complexity of the problem requires a higher number of training epochs to obtain an accurate solution. We choose a number of epochs equal to 1000. All the other training parameters remain the same as in the previous example. Also the PINN structure remains the same. Once the training is completed, we can test the accuracy of the solution by comparing it to the solution of the same problem obtained using a FEM algorithm. After 1000 training epochs, for a total training time of 2 minutes and 13 seconds, the loss function reached a value of $5.3374 \cdot 10^{-4}$. Figure 4.22 shows the progress of the network training. We can plot the equipotential lines using the MATLAB *contour* function to visualize the 2D potential distribution in the domain. Figure 4.23 shows the comparison between the PINN and the FEM solution. We can see that the two distributions look similar. The L_2 norm of the error vector (computed as in (3.11)) is 0.0543, which confirms the good accuracy of the solution obtained using the PINN.

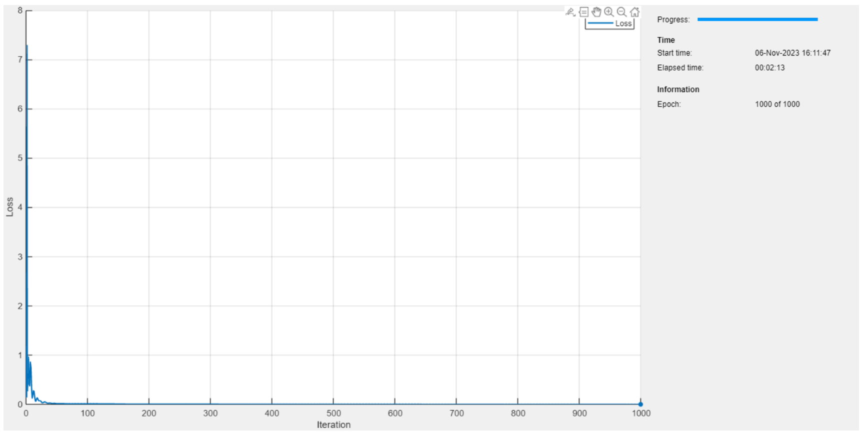


FIGURE 4.22: Two-region capacitor, training progress

4.6 Developing a solid approach to electrostatic problems

The results obtained in the examples above confirm the capabilities of the approach proposed in Section 4.3. By using the centres of the mesh elements as collocation points, we managed to solve the problem of interfaces between regions with different values of the material parameter. Moreover, using a mesh-guided approach to the choice of training points makes it easy to implement a FEM code for the testing of the results. This approach should allow us to treat problems with more complex geometry, and by varying the mesh size around specific edges of the domain, we could obtain a more precise approximation of the potential distribution in sensible areas of the domain. Gong et al., in [10], call this approach *Mesh-Assisted Non-Uniform Sampling*.

Another important result presented in this chapter was the ability to solve problems, even though very simple, that had a space-varying parameter and that the original code that we have introduced in chapter 3 was not able to solve. This was done without twisting the code too much and without the need for tools other than those introduced in the previous chapters. In particular, we were able to keep all the advantages of automatic differentiation by combining Maxwell's equations in a different way. The next examples that we are going to deal with, will put the current version of the algorithm to test. We will use these examples to see if this approach is solid enough to solve more complex problems.

4.7 Parallel plate capacitor with three dielectric regions

4.7.1 Problem description

To make a step further in the complexity of the problems that we are testing our algorithm with, we can try to solve a three-region capacitor problem. In particular, we analyze a device that has a central dielectric region with a relative permittivity $\varepsilon_r = 16$ and the upper and lower region with $\varepsilon_r = 1$. The central region is 1m large and 0.4m long. The rest of the data remains the same as in the example proposed in Section 4.5. The main difference is that we now have two interfaces, one between the central and the upper region and one between the central and the lower region. Figure 4.24 shows the meshed domain and the division between the three regions.

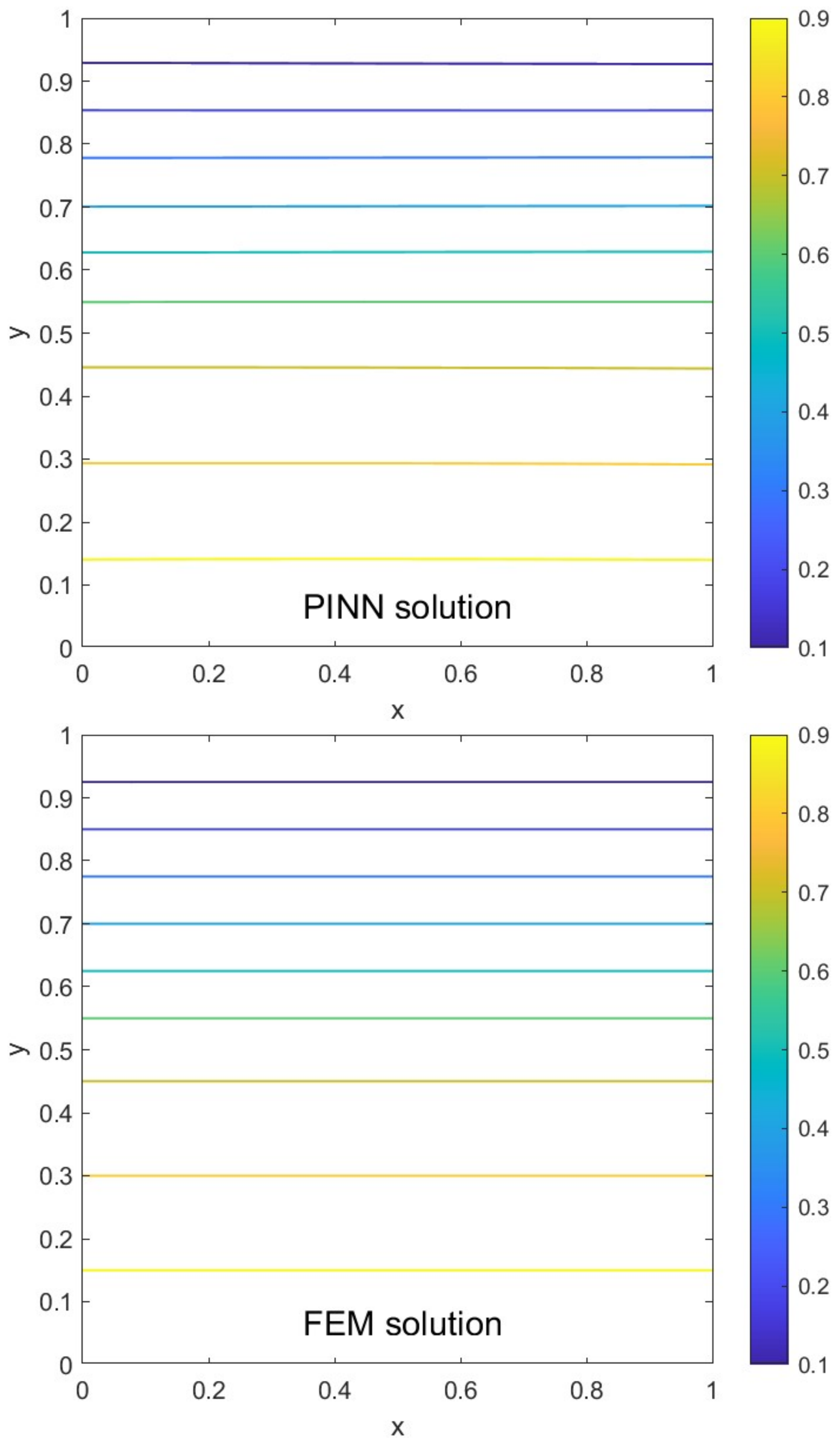


FIGURE 4.23: Two-region capacitor, contour plot of the potential distribution, PINN vs. FEM

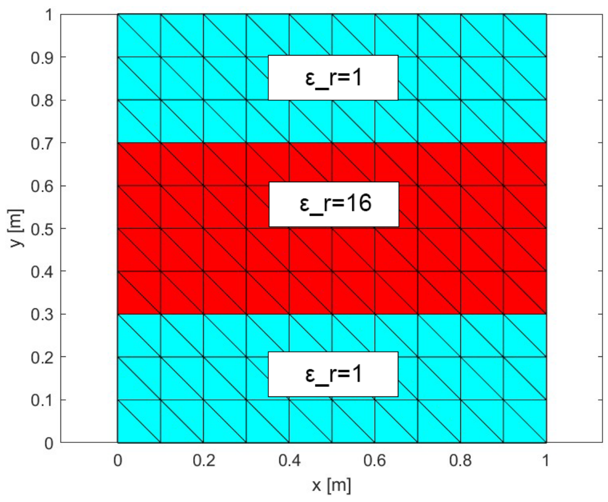


FIGURE 4.24: Three-region domain and mesh

4.7.2 Training points and loss function definition

The training points choice is done in the same way as in the previous case. The only difference is that now we need four sets of interface nodes, two for the upper interface and two for the lower interface. The interface nodes are identified in the same way as described in Section 4.6.2. Also the loss function remains almost exactly the same as described in Section 4.5.2, we only need to add two terms to it. The first one is needed because now we have three regions and not just two, and the second one is needed to account for the condition on the second interface.

4.7.3 Training results and comparison with FEM

The training parameters remain the same as in the previous example, except for the number of training epochs. Given the increased complexity of the problem, we choose a number of epochs equal to 6000. We also change the number of neurons in each layer of the PINN to 250, to improve the capacity of the PINN to approximate the solution. Once the training process is finished, after 50 minutes and 34 seconds (the time increase is due to the higher number of epochs and neurons to train), the loss function reached a value of $2.4754 \cdot 10^{-4}$. From figure 4.25, we can see that the solution obtained by the PINN is quite similar to the one obtained by FEM. This is also confirmed by the mean squared error between the two solutions evaluated on the mesh nodes, which is equal to $1.934 \cdot 10^{-4}$. These results confirm the generality of the approach that we adopted, even though the training time and the code complexity are starting to increase dramatically.

4.8 Parallel plate capacitor with a dielectric inclusion

4.8.1 Problem description

The last example that we are going to discuss is a full 2D electrostatic problem, i.e. a capacitor with a dielectric inclusion inside the domain. This inclusion has a permittivity value sixteen times higher with respect to the rest of the dielectric region. The inclusion is 0.4m$\times$0.4m and is at the centre of the domain. All the other data remain the same. Figure 4.26 shows the domain configuration and the elements of the mesh. We can see that now we have four interfaces, two of them parallel and two of them normal to the capacitor terminals. We can try to modify the code to account for the two additional interfaces. One way to this is to divide the domain in

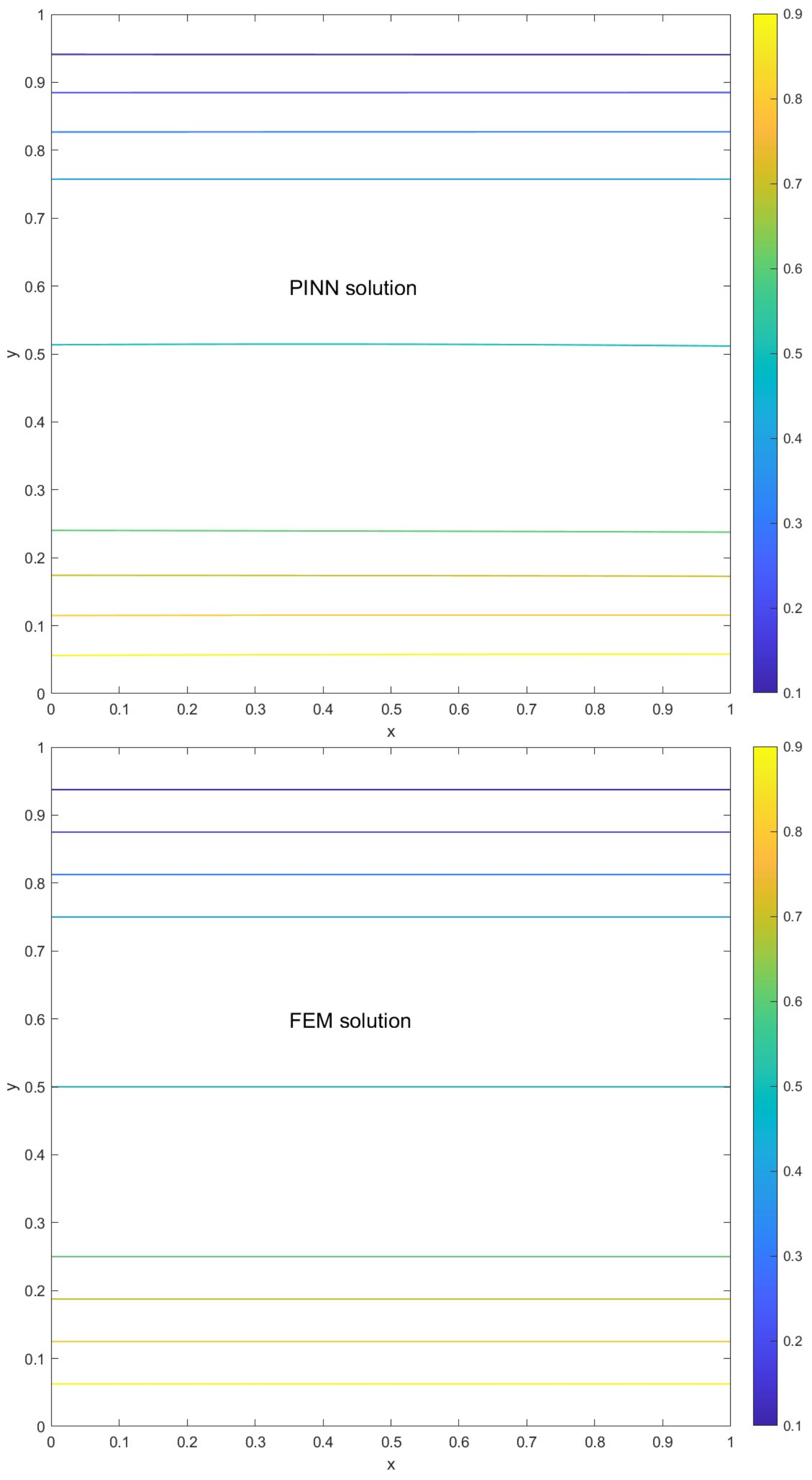


FIGURE 4.25: Three-region capacitor, contour plot of the potential distribution, PINN vs. FEM

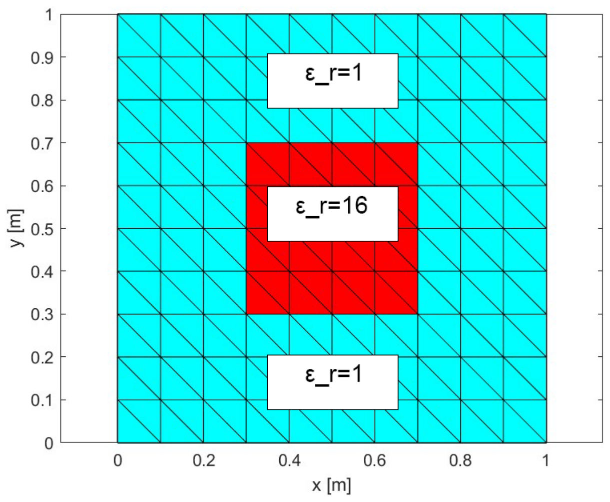


FIGURE 4.26: Capacitor with dielectric inclusion, meshed domain

two regions regions, as we did in the previous examples. The first region is the area around the inclusion (cyan area in figure 4.26), the second one is the inclusion (red area in Figure 4.26). The interface conditions to enforce are the following:

$$[D_y] = D_y^+ - D_y^- = 0 \text{ for interfaces parallel to the capacitor terminals} \tag{4.40}$$

$$[D_x] = D_x^+ - D_x^- = 0 \text{ for interfaces normal to the capacitor terminals.} \tag{4.41}$$

(4.40) and (4.41) are enforced in the same way as described in Section 4.5.2, being careful not to enforce two different conditions on the nodes that are in the corners of the inclusion. They could belong to both parallel and normal interfaces, we arbitrarily decide that they belong to the parallel ones. The training is then performed in the same way as before, with the addition of two loss function terms to account for interface conditions on D_x. The number of epochs is initially set to 10000.

4.8.2 Convergence failure

After 10000 training epochs, the training process is complete. However, this time the PINN output does not converge to a solution that represents, not even approximately, the potential distribution inside the capacitor. Even increasing the number of epochs and the number of mesh nodes doesn't seem to bring any significant improvement to the solution. This is probably due to the fact that, in this case, there is a region, the dielectric inclusion, that does not have any boundary condition imposed on its points. Conversely, in all the previous examples, all the regions had at least some points tangent to the domain boundary. This means that the inclusion doesn't have any points on which the electric potential or the electric displacement field have a fixed value, probably leading to instability in the solution.

We can try to solve this problem by changing our approach to the network training. The idea is to train the PINN as in Section 4.4, instead of dividing the domain in regions. The permittivity coefficient is defined as a vector of the same length as the number of collocation points, and its elements have a value of $\varepsilon = 16\varepsilon_0$ in correspondence of the collocation points inside the inclusion, while they are equal to ε_0 outside the inclusion. The loss function is defined as in Section 4.4.2 and all the other settings are kept the same as before. To test the robustness of the approach, we performed 200000 training epochs to see if this approach improved the accuracy of the solution. After 200000 training epochs and 7 hours and 42 minutes of training time, even though the loss function has reached a value of $2.407 \cdot 10^{-6}$, the solution obtained was not accurate compared to the one obtained with FEM. We can clearly observe from Figure 4.28 that the PINN output is not an accurate approximation of the potential inside the capacitor. The equipotential lines are straight, while they should curve in correspondence of the dielectric inclusion edges, and they are concentrated towards the lower terminal of the capacitor. Numerical results show that the algorithm is not able to treat 2D electrostatic problems in which the electric displacement has two spatial components.

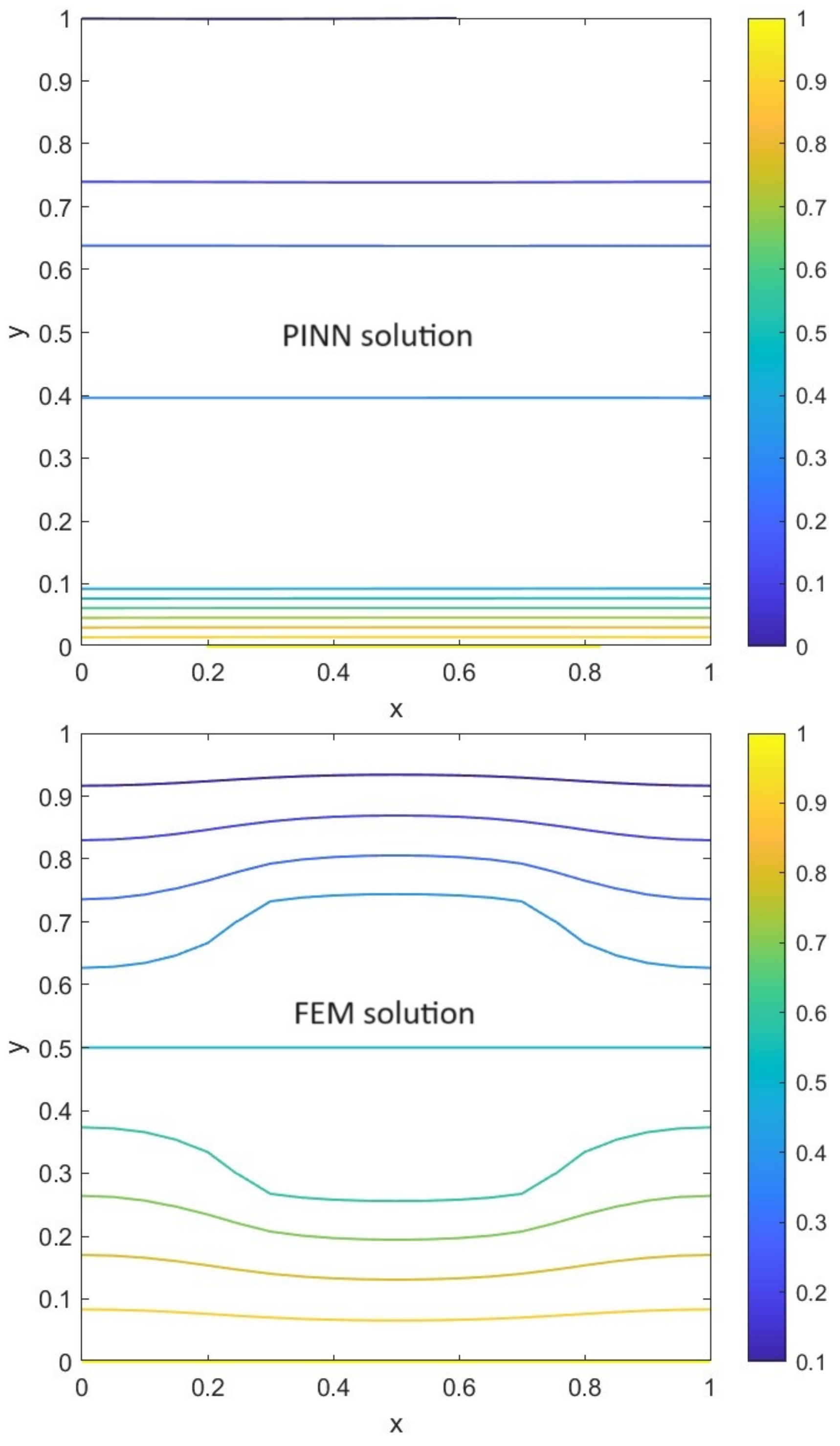


FIGURE 4.27: Capacitor with dielectric inclusion, potential distribution, PINN vs. FEM

Chapter 5

Extending the use of PINNs to general 2D electromagnetic problems

5.1 Problems with non-homogeneous media

The first limitation that we encountered in the development of our code was in the solution of problems that had a non-homogeneous medium in their domain. As long as the model geometry only featured interfaces between different regions oriented in only one direction (parallel to x-axis in our case), the algorithm achieved good accuracy, but as soon as we introduced a domain that also had interfaces oriented in another direction (parallel to y-axis), the algorithm did not provide acceptable results. The problem probably lies in the inability of the network to approximate fields that have both components (along x and y-axes) different from zero. In order to move on to more complex problems, we need to find a way to actually be able to treat this type of configurations.

5.2 Problems with sources inside the domain

The next step in the code development would be the introduction of sources inside the domain. In chapter 4 we discussed problems that featured no charge density. However, in most applications, field sources are actually present, and they play a key role in the field and potential distribution in the domain. For example in electrical transformers, it's the current density in the coils that generates the magnetic field responsible for the energy transfer [30]. Theoretically, by plugging the value of the source F term inside (2.8), we should be able to solve problems that involve field sources using PINNs. However, as already happened with a space varying material parameter, a space varying source parameter in (2.7) might cause problems in the network convergence. In fact, it would lead to the introduction of interfaces between the field source (for example the winding of a transformer) and the rest of the domain. Figure 5.1 shows an example of a domain that is non-homogeneous and has field sources inside the domain. We can see from the image that the coil, which is our field source, has interfaces with both the core and the air.

5.3 Non-linear problems

The solution of non-linear field problems is very common in electrical engineering because electric machines are made of non-linear materials. Ferromagnetic materials are characterized by a magnetic permeability that is not constant, but is dependent on the value of the magnetic field strength H and the magnetic flux density B. This is due to the fact that, during the magnetization process of the material, the B-H curve saturates. Since $\mu = B/H$, if the slope of the B-H curve varies, varies also the value of μ. Figure 5.2 shows a typical magnetization curve

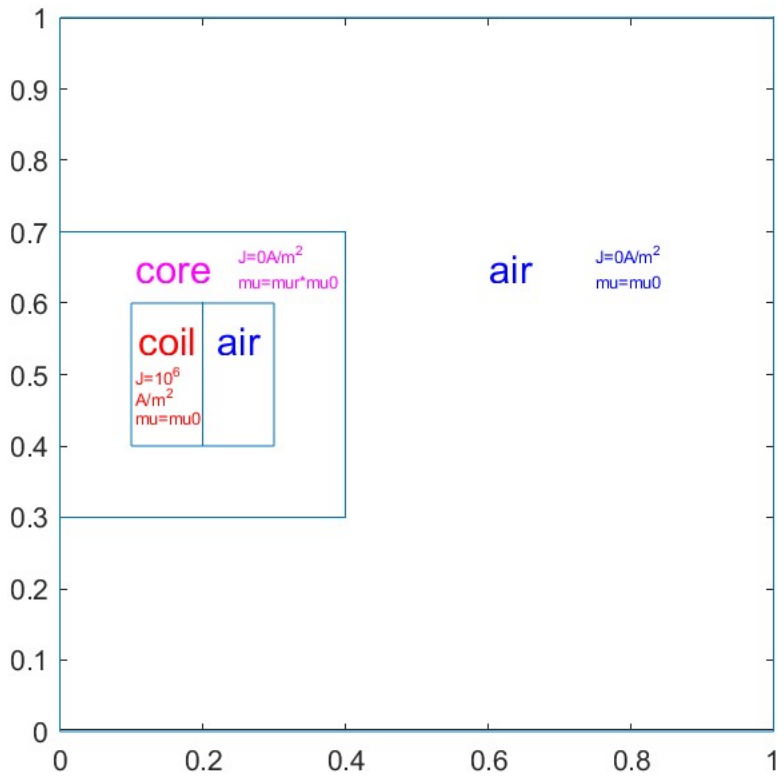


FIGURE 5.1: Non-homogeneous domain with sources

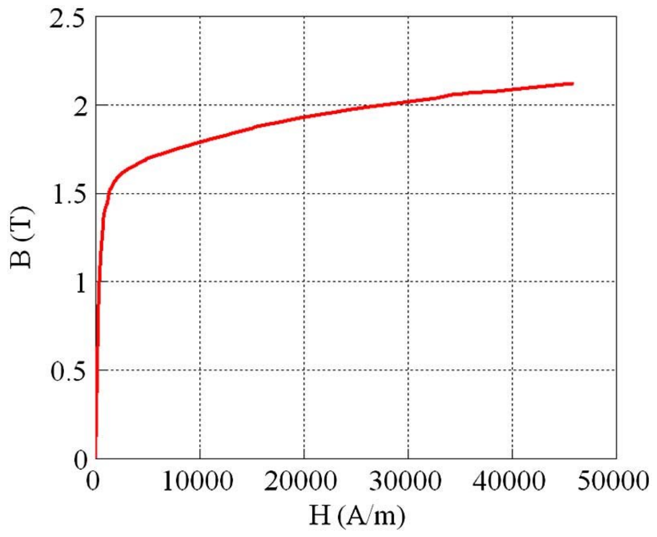


FIGURE 5.2: Magnetization curve of a ferromagnetic material [32]

of a ferromagnetic material. We can observe how the curve saturates, showing a non linear behavior. An in-depth explanation of ferromagnetic materials and the phenomenon of hysteresis is presented in [31].

As already mentioned in Section 2.7, a finite element solution of non-linear electromagnetic problems can be computationally demanding because it often requires a time-domain solver. Integrating the non-linearity into the loss function of a network can be a way to solve non-linear problems using PINNs. The value of μ can be updated at each training iteration, knowing the magnetization curve of the material and the value of H and B. B can be easily obtained from the curl of the magnetic potential A (by using automatic differentiation), we would need to add one more PINN output to obtain the magnetic field strength H. This can be done using the approach proposed in [10], that we already readapted to solve the parallel plate capacitor problems in Chapter 4. An additional term would be added to the loss function to account for the variation of μ:

$$\mu_{curve,i} = \frac{B_{PINN,i}}{H_{PINN,i}} \tag{5.1}$$

$$res_{\mu,i} = \frac{B_{PINN,i}}{H_{PINN,i}} - \mu_{curve,i}, \tag{5.2}$$

where $\mu_{curve,i}$ is the value of the permeability on the i-th collocation point obtained from the magnetization curve, and $B_{PINN,i}$ and $H_{PINN,i}$ are the values of the magnetic flux density and the magnetic field strength on the i-th collocation point obtained through the PINN training. In this way the non-linearity would be integrated in the PINN training and the solution obtained should be coherent with the physics of the problem.

5.4 Time-dependent problems

When, in addition to space derivatives, time derivatives of field quantities are present, we need to perform an analysis in space and time in order to solve our problem. Using conventional solvers like FDTD (Finite Difference Time-Domain) or FEMTD (Finite Element Time-Domain) requires the implementation of a time-stepping scheme [33]. Conversely, the use of PINNs to solve time-dependent problems could eliminate the need for time-stepping, like in the case of non linear problems. The time domain analysis could be carried out by adding time to the inputs of the network, and training it on space and time collocation points [34]. The network training process could remain almost the same, with the addition of a term to the loss function to account for initial conditions. For a 2D transient problem (2.5) becomes:

$$\nabla \cdot k_1 \nabla u(x,y,t) + k_2 \frac{\partial}{\partial t} u(x,y,t) = F. \tag{5.3}$$

The residuals that define the loss function would be:

$$res_{PDE,i} = -\nabla \cdot k_1 \nabla u_{PINN,i} - k_2 \frac{\partial u_{PINN,i}}{\partial t} - F, \tag{5.4}$$

$$res_{in,i} = u_{PINN,i,0} - u_{0,i} \tag{5.5}$$

$$res_{bd,j} = u_{PINN,bd,j} - u_{bd,j}, \tag{5.6}$$

where $u_{PINN,i}$ is the network output computed on the collocation point (x_i, y_i, t_i), $u_{PINN,i,0}$ is the network output on the collocation point $(x_i, y_i, 0)$, $u_{PINN,bd,j}$ is the network output on the boundary point $(x_{bd,j}, y_{bd,j}, t_j)$ and $u_{0,i}$ and $u_{bd,j}$ are the initial and boundary conditions on the i-th collocation point and on the j-th boundary point. All the loss function terms can be computed as in (3.7).

5.5 Developing a solid approach to electromagnetic problems

In this chapter, we have briefly analyzed the possible future steps that can be made to continue the development of a MATLAB code that can efficiently solve electromagnetic IBVPs using PINNs. Among all the aspects that electromagnetism forces us to take into account, non-homogeneity, non linearity and dependence on time seem the biggest and issues that need to be tackled in order to make PINNs a viable alternative to traditional numerical methods such as FEM. If what is discussed in this chapter is implemented successfully, it could make the PINN-based algorithm a serious competitor against conventional solvers for electromagnetic initial-boundary value problems (IBVPs). However, the issues mentioned in Section 5.1 and 5.2 must be resolved.

Chapter 6

Conclusion

The goal of this work was to develop a MATLAB code that was able to solve increasingly more difficult electromagnetic BVPs using PINNs. We analyzed all the essential aspects of artificial neural networks structure and training and then we tried to apply the knowledge acquired to the solution of practical examples. We started from the Poisson's equation example taken from the *MathWorks*® website and then moved on to analyze various configurations of the parallel plate capacitor, re-adapting the initial code depending on the situation. We succesfully adapted the MATLAB PINN solver to the analysis of a 1D electrostatic problem with homogeneous media. In order to extend the analysis to inhomogeneous approach, a new approach, based on field and scalar variables, was developed. The approach proposed in [10], that we rearranged in order to apply it to electrostatic problems, combined with the interface conditions method that we implemented, turned out to be successful in various applications. Numerical results showed a very good agreement with both analytical and FEM solutions.

However, the extension to 2D static problems led to a considerable increase in the code complexity. The training time has risen from 9 s all the way up to 50 min and 34 s, which is a considerable amount of time compared to a standard FEM solver for a linear, two-dimensional, electrostatic problems like the ones we presented in Section 4.7. Moreover, it was not possible to solve a full 2D electrostatic problem with a dielectric inclusion.

Even though the attempts made in this thesis were not always successful and there are still many steps to be taken in order to make PINN-based solvers competitive with FEM-based ones, we also need to highlight the positive results that are presented in this thesis. Even if the problems that the algorithm was able to solve were very simple, it must be remarked that PINNs are still at an early stage of development and the literature is still limited, especially if electromagnetic problems are considered. It is clear that developing a code that is able to deal with the vast majority of problems that electrical engineers are required to solve is not an easy task. The algorithm presented in this thesis is just at the beginning of its development and already encountered challenges that seem unsolvable with the tools and strategies that we introduced in the previous chapters. A huge amount of research will be required to bring everything that we mentioned in this thesis together into a code that is usable, solid and efficient.

The possible future developments of the code presented in this thesis, as discussed in Chapter 5, regard the possibility to solve problems that feature non-homogeneous media, field sources, non-linear materials and time-dependent equations. As this thesis highlights, PINNs offer all the tools needed to tackle this type of problems. However a way to successfully implement them in practical applications must be found.

Appendix A

MATLAB codes

In this Appendix we present the last functioning version of the algorithm, that was used to solve the problem discussed in Section 4.7. Section A.1 shows the main algorithm and Section A.2 shows the function used to compute the loss function value and its gradients.

A.1 Main algorithm

```
%% MODEL PARAMETERS

%Domain

L_x=1; %[m]
L_y=1; %[m]

h=0.1; %mesh size

rel_toll=1e-6;

eps_0=8.85418e-12; %vacuum permittivity
p1=1; %material parameter region 1
p2=16; %region 2

%Boundary conditions

V_b=1; %bottom
V_t=0; %top
V_0=V_t-V_b; %applied voltage


%% MESH GENERATION

N_x=L_x/h; %number of division along x-axys
N_y=L_y/h; %number of division along y-axis
vec_x=linspace(0,L_x,N_x+1);
vec_y=linspace(0,L_y,N_y+1);
[X,Y]=meshgrid(vec_x,vec_y); %creates a grid with x and y values
x=X(:);
y=Y(:);
nod=[x,y]; %coordinate matrix
conn=delaunay(nod(:,1),nod(:,2)); %connectivity matrix
```

```
N=size(nod,1); %number of nodes
M=size(conn,1); %number of cells
figure (1)
triplot(conn,nod(:,1),nod(:,2),'k-','Linewidth',1) %mesh plot
axis equal %same scaling of both axis
xlabel('x [m]')
ylabel('y [m]')
hold on

%Find elements centres

centres=zeros(M,2);
for i=1:M
nodes=conn(i,:);
nodtr=nod(nodes,:);
centres(i,:)=mean(nodtr,1);
end


%% BOUNDARY NODES AND CONDITIONS

%Find boundary nodes

ind=convhull(nod(:,1),nod(:,2));
Fr=sort(ind(1:end-1));
indFr1=find(abs(nod(Fr,2))<rel_toll*L_y); %bottom nodes:
%they have y coordinate equal to 0
FrD1=Fr(indFr1); %dirichlet boundary nodes (bottom)
indFr2=find(abs(nod(Fr,2)-L_y)<rel_toll*L_y); %top nodes:
%they have y coordinate equal to L_y
FrD2=Fr(indFr2); %dirichlet boundary nodes (top)
indN=setdiff(Fr,FrD1);
indN=setdiff(indN,FrD2);
BdNodes_top=nod(FrD2,:);
BdNodes_bottom=nod(FrD1,:);
BdNodes=nod(Fr,:);
BdNodesD=[BdNodes_top
BdNodes_bottom];
BdNodesD=dlarray(BdNodesD,"BC");
BdNodesN=nod(indN,:);
BdNodesN=dlarray(BdNodesN,"BC"); %neumann nodes

%Boundary conditions

BdCond=zeros(size(BdNodesD,1),1);
BdCond(1:size(BdNodes_top,1))=V_t;
BdCond(size(BdNodes_top,1)+1:size(BdNodes_top,1)+size(BdNodes_bottom,1))=V_b;
BdCond=dlarray(BdCond,"BC");

%% DIVIDE THE TWO REGIONS
```

```
ind_reg2=[];
for e=1:M
triangle=conn(e,:); %triangle nodes
nodtri=nod(triangle,:); %triangle coordinates
centre=mean(nodtri,1); %compute the centre of the triangle
%check if the triangle is inside the inclusion:
if abs(centre(2))>0.3 && abs(centre(2))<0.7
ind_reg2=[ind_reg2,e]; %#ok<AGROW>
patch(nodtri(:,1),nodtri(:,2),'red')
else
patch(nodtri(:,1),nodtri(:,2),'blue')
end
end
ind_reg3=[];
for e=1:M
triangle=conn(e,:); %triangle nodes
nodtri=nod(triangle,:); %triangle coordinates
centre=mean(nodtri,1); %compute the centre of the triangle
%check if the triangle is inside the inclusion:
if abs(centre(2))>0.7
ind_reg3=[ind_reg3,e]; %#ok<AGROW>
patch(nodtri(:,1),nodtri(:,2),'green')
end
end
ind_tot=1:M;
ind_reg1=setdiff(ind_tot,ind_reg2);
ind_reg1=setdiff(ind_reg1,ind_reg3);

%% INTERFACE POINTS

nod_int1=[]; %region 1 (blue)
ind_int1=[];
nod_int3=[]; %region3 (green)
ind_int3=[];
nod_int2_1=[]; %region 2 (red) with region 1 (blue)
ind_int2_1=[];
nod_int2_2=[]; %region 2 (red) with region 3 (green)
ind_int2_2=[];
for i=1:M
nod_y=centres(i,2);
if abs(nod_y-0.3)<h && (nod_y-0.3)<0
nod_int1=[nod_int1;
centres(i,:)];  %#ok<AGROW>
ind_int1=[ind_int1,i];   %#ok<AGROW>
end

if abs(nod_y-0.7)<h && (nod_y-0.7)>0
nod_int3=[nod_int3;
centres(i,:)];  %#ok<AGROW>
```

```
ind_int3=[ind_int3,i];   %#ok<AGROW>
end

if abs(nod_y-0.3)<h && (nod_y-0.3)>0
nod_int2_1=[nod_int2_1;
centres(i,:)];     %#ok<AGROW>
ind_int2_1=[ind_int2_1,i];  %#ok<AGROW>
end

if abs(nod_y-0.7)<h && (nod_y-0.7)<0
nod_int2_2=[nod_int2_2;
centres(i,:)];          %#ok<AGROW>
ind_int2_2=[ind_int2_2,i];  %#ok<AGROW>
end
end

nod_int1=sortrows(nod_int1);
nod_int1=dlarray(nod_int1,"BC");
nod_int2_1=sortrows(nod_int2_1);
nod_int2_1=dlarray(nod_int2_1,"BC");
nod_int2_2=sortrows(nod_int2_2);
nod_int2_2=dlarray(nod_int2_2,"BC");
nod_int3=sortrows(nod_int3);
nod_int3=dlarray(nod_int3,"BC");

%Update collocation points

centres=dlarray(centres',"BC");
domainCollocationPoints=centres;
ind_int=[ind_int1, ind_int2_1, ind_int2_2, ind_int3];
ind_reg1=setdiff(ind_reg1,ind_int1);
ind_reg2=setdiff(ind_reg2,ind_int2_1);
ind_reg2=setdiff(ind_reg2,ind_int2_2);
ind_reg3=setdiff(ind_reg3,ind_int3);
domainCollocationPoints(ind_int,:)=[];

%% COEFFICIENTS VECTOR

p=p1.*ones(M,1);
p(ind_reg2)=p2;
p=dlarray(p,"BC");

%% NETWORK ARCHITECTURE

%Define a multilayer network with four fully connected layers,
%each with 150 neurons
%The first layer has two input channels corresponding to the inputs x and y
%The last layer has three outputs corresponding to A(x,y), Hx(x,y), Hy(x,y)

numNeurons=150;
```

```
layers=[
featureInputLayer(2,Name="featureinput")
fullyConnectedLayer(numNeurons,Name="fc1")
tanhLayer(Name="tanh_1")
fullyConnectedLayer(numNeurons,Name="fc2")
tanhLayer(Name="tanh_2")
fullyConnectedLayer(numNeurons,Name="fc3")
tanhLayer(Name="tanh_3")
fullyConnectedLayer(3,Name="fc4")
];

pinn=dlnetwork(layers);

%% TRAINING OPTIONS

N_epochs=6000; %number of epochs
initialLR=0.01; %initial learning rate
LRDecay=0.005; %learning rate decay

%Initialize the average gradients and squared average gradients for Adam

average_grad=[];
average_sq_grad=[];

%Total number of iterations for the training progress monitor

N_iterations=N_epochs;

%Initialize the training progress monitor

monitor=trainingProgressMonitor(Metrics="Loss",Info="Epoch",...
...XLabel="Iteration");

%% TRAIN PINN

iteration=0;
epoch=0;
learning_rate=initialLR;

while epoch<N_epochs && ~monitor.Stop
epoch=epoch+1;
iteration=iteration + 1;
XY1=centres(ind_reg1,:);
XY2=centres(ind_reg2,:);
XY3=centres(ind_reg3,:);
XY1=dlarray(XY1,"BC");
XY2=dlarray(XY2,"BC");
XY3=dlarray(XY3,"BC");

%Compute the loss function value and its gradients
```

```
[loss,gradients]=dlfeval(@modelLoss_cap2D,pinn,BdNodesD,BdNodesN,...
...BdCond,XY1,XY2,XY3,nod_int1,nod_int2_1,nod_int2_2,nod_int3);

%Update the network parameters using Adam optimizer

[pinn,average_grad,average_sq_grad]=adamupdate(pinn,gradients,...
...average_grad,average_sq_grad,iteration,learning_rate);

%Update the learning rate

learning_rate=initialLR/(1+LRDecay*iteration);

%Update the training progress monitor

recordMetrics(monitor,iteration,Loss=loss);
updateInfo(monitor,Epoch=epoch+"of"+N_epochs);
monitor.Progress=100*iteration/N_iterations;

end

%% TEST PINN

%Contour plot

nodes_dlarray=dlarray(nod',"CB");
Out_pinn=gather(extractdata(predict(pinn,nodes_dlarray)));
Vpinn=Out_pinn(1,:);
Dxpinn=Out_pinn(2,:);
Dypinn=Out_pinn(3,:);

figure (3)
Z=reshape(Vpinn,[N_y+1,N_x+1]); %reshape the solution vector as a matrix
contour(X,Y,Z,'LineWidth',1)
colorbar
axis equal
xlabel('x');
ylabel('y');
zlabel('potential (PINN)')
xt = 0.35;
yt = 0.6;
str = 'PINN solution';
text(xt,yt,str,'Color','black','FontSize',14)
f=gcf;
exportgraphics(f,'PINN solution.jpg')
```

A.2 Loss function

```
function [loss,gradients]=modelLoss_cap2D(net,BdNodesD,BdNodesN,BdCond...
...XY1,XY2,XY3,nod_int1,nod_int2_1,nod_int2_2,nod_int3)

%Train pinn on the first region

[out1]=forward(net,XY1);
V=out1(1,:);
D_x=out1(2,:);
D_y=out1(3,:);

%Compute derivatives of V

dV=dlgradient(sum(V,"all"),XY1,EnableHigherDerivatives=true);
dV_x=dV(1,:);
dV_y=dV(2,:);

%Compute derivatives of D

dD_x=dlgradient(sum(D_x,"all"),XY1,EnableHigherDerivatives=true);
dD_xx=dD_x(1,:);
dD_y=dlgradient(sum(D_y,"all"),XY1,EnableHigherDerivatives=true);
dD_yy=dD_y(2,:);

%Enforce PDE, calculate lossF

res1=-dV_x-D_x;
res2=-dV_y-D_y;
res3=dD_xx+dD_yy;
target1=zeros(size(res1));
lossF1=l2loss(res1,target1);
target2=zeros(size(res2));
lossF2=l2loss(res2,target2);
target3=zeros(size(res3));
lossF3=l2loss(res3,target3);
lambda1=0.25;
lambda2=0.25;
lossF_reg1=lambda1*lossF1+lambda2*lossF2+(1-lambda1-lambda2)*lossF3;

%Train pinn on the second region

[out2]=forward(net,XY2);
V=out2(1,:);
D_x=out2(2,:);
D_y=out2(3,:);

%Compute derivatives of V

dV=dlgradient(sum(V,"all"),XY2,EnableHigherDerivatives=true);
```

```
dV_x=dV(1,:);
dV_y=dV(2,:);

%Compute derivatives of D

dD_x=dlgradient(sum(D_x,"all"),XY2,EnableHigherDerivatives=true);
dD_xx=dD_x(1,:);
dD_y=dlgradient(sum(D_y,"all"),XY2,EnableHigherDerivatives=true);
dD_yy=dD_y(2,:);

%Enforce PDE, calculate lossF

res1=-16.*dV_x-D_x;
res2=-16.*dV_y-D_y;
res3=dD_xx+dD_yy;
target1=zeros(size(res1));
lossF1=l2loss(res1,target1);
target2=zeros(size(res2));
lossF2=l2loss(res2,target2);
target3=zeros(size(res3));
lossF3=l2loss(res3,target3);
lambda1=0.25;
lambda2=0.25;
lossF_reg2=lambda1*lossF1+lambda2*lossF2+(1-lambda1-lambda2)*lossF3;

%Train pinn on the third region

[out3]=forward(net,XY3);
V=out3(1,:);
D_x=out3(2,:);
D_y=out3(3,:);

%Compute derivatives of V

dV=dlgradient(sum(V,"all"),XY3,EnableHigherDerivatives=true);
dV_x=dV(1,:);
dV_y=dV(2,:);

%Compute derivatives of D

dD_x=dlgradient(sum(D_x,"all"),XY3,EnableHigherDerivatives=true);
dD_xx=dD_x(1,:);
dD_y=dlgradient(sum(D_y,"all"),XY3,EnableHigherDerivatives=true);
dD_yy=dD_y(2,:);

%Enforce PDE, calculate lossF

res1=-dV_x-D_x;
res2=-dV_y-D_y;
res3=dD_xx+dD_yy;
```

```
target1=zeros(size(res1));
lossF1=l2loss(res1,target1);
target2=zeros(size(res2));
lossF2=l2loss(res2,target2);
target3=zeros(size(res3));
lossF3=l2loss(res3,target3);
lambda1=0.25;
lambda2=0.25;
lossF_reg3=lambda1*lossF1+lambda2*lossF2+(1-lambda1-lambda2)*lossF3;

%Enforce interface conditions

[out_int1]=forward(net,nod_int1);
[out_int2_1]=forward(net,nod_int2_1);
[out_int2_2]=forward(net,nod_int2_2);
[out_int3]=forward(net,nod_int3);
Dy1=out_int1(3,:);
Dy21=out_int2_1(3,:);
Dy23=out_int2_2(3,:);
Dy3=out_int3(3,:);
res4=Dy1-Dy21;
res5=Dy23-Dy3;
target4=zeros(size(res4));
target5=zeros(size(res5));
lossF4=l2loss(res4,target4);
lossF5=l2loss(res5,target5);
loss_int=0.5*lossF4+0.5*lossF5;

lossF=0.25*lossF_reg1+0.25*lossF_reg2+0.25*lossF_reg3+(1-0.75)*loss_int;

%Enforce Dirichlet boundary conditions, calculate lossUd.

%Electric potential

[Bdout]=forward(net,BdNodesD);
V_bd=Bdout(1,:);
res4=V_bd-BdCond;
target4=zeros(size(res4));
lossU1=l2loss(res4,target4);

%Electric displacement field

D_bdx=Bdout(2,:);
res5=D_bdx;
target5=zeros(size(res5));
lossU2=l2loss(res5,target5);

lambda3=1;
lossUd=lambda3*lossU1+(1-lambda3)*lossU2;
```

```
%Enforce Neumann boundary conditions, calculate lossUn

%Electric potential

[Bdout]=forward(net,BdNodesN);
V_bd=Bdout(1,:);
dV_bd=dlgradient(sum(V_bd,"all"),BdNodesN,EnableHigherDerivatives=true);
dV_bd_x=dV_bd(1,:);
res6=dV_bd_x;
target6=zeros(size(res6));
lossU3=l2loss(res6,target6);

%Electric displacement field

D_bdx=Bdout(2,:);
res7=D_bdx;
target7=zeros(size(res7));
lossU4=l2loss(res7,target7);

lambda4=0.5;
lossUn=lambda4*lossU3+(1-lambda4)*lossU4;

lambda5=0.5;
lossU=lambda5*lossUd+(1-lambda5)*lossUn;

%Combine the losses

lambda=0.5;
loss=lambda*lossF+(1-lambda)*lossU;

gradients=dlgradient(loss,net.Learnables);

end
```

Bibliography

[1] Brown, S. "Machine Learning, explained", *MIT Sloan*, 21 Apr. 2021, <https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained>.

[2] LeCun, Y., Bengio, Y., Hinton, G. "Deep learning", *Nature*, Vol. 521, pp. 436-44, May 2015.

[3] Shinde, P.P., Shah, S. "A Review of Machine Learning and Deep Learning Applications", *Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, 2018.

[4] Krogh, A. "What are artificial neural networks?", *NATURE BIOTECHNOLOGY*, Vol. 26, No. 2, Feb. 2008.

[5] Awad, M., Khanna, R. *Efficient Learning Machines, Theories, Concepts, and Applications for Engineers and System Designers*, pp. 127-142, ApressOpen, 2015.

[6] Krenker, A., Bešter, J., Kos, A. "Introduction to the Artificial Neural Networks", *Artificial Nural Networks Methodological Advances and Biomedical Applications*, InTech, 2011.

[7] Beltrán-Pulido, A., Bilionis, I., Aliprantis, D. "Physics-Informed Neural Networks for Solving Parametric Magnetostatic Problems", *IEEE Transactions on Energy Conversion*, Vol. 20, No. 10, Jun. 2022.

[8] Lagaris, I.E., Likas, A., Fotiadis, D.I. "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations", *IEEE Transactions on Neural Networks*, Vol. 9, No. 5, Sep. 1998.

[9] Lu, L., Meng, X., Mao, Z., Karniadakis, G.M. "DeepXDE: A Deep Learning Library for Solving Differential Equations", *SIAM Review*, Vol. 63, No. 1, pp. 208–228, Society for Industrial and Applied Mathematics, Feb. 2021.

[10] Gong, Z., Chu, Y., Yang, S. "Physics-Informed Neural Networks for Solving Two-Dimensional Magnetostatic Fields", *IEEE Transactions on Magnetics*, Vol. 59, No. 11, Nov. 2023.

[11] Katsikis, D., Muradova, A.D., Stavroulakis, G.E. "A Gentle Introduction to Physics-Informed Neural Networks, with Applications in Static Rod and Beam Problems", *Journal of Advances in Applied & Computational Mathematics*, Vol. 9, pp. 103-128, May 2022.

[12] Hornik, K., Stinchcombe, M., White, H. "Multilayer feedforward networks are universal approximators", *Neural Networks*, Vol. 2, No. 5, pp. 359-366, 1989.

[13] Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M. "Automatic Differentiation in Machine Learning: a Survey", *Journal of Machine Learning Research*, Vol. 18, pp. 1-43, Apr 2018.

[14] Griewank, A., Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Society for Industrial and Applied Mathematics, 2008.

[15] Kingma, D.P., Ba, J.L. "Adam: a method for stochastic optimization", *International Conference on Learning Representations*, 2015.

[16] Ajagekar, A. "Adam", *Cornell University Computational Optimization Open Textbook*, 16 Dec. 2021,
<https://optimization.cbe.cornell.edu/index.php?title=Adam>

[17] Kumar, D.K. "Getting to Know Adam Optimization: A Comprehensive Guide", *LinkedIn*, 30 Mar. 2023,
<https://www.linkedin.com/pulse/getting-know-adam-optimization-comprehensive-guide-kiran-kumar>

[18] Kim, N.H. *Introduction to Nonlinear Finite Element Analysis*, Springer, 2015.

[19] "Solve Poisson Equation on Unit Disk Using Physics-Informed Neural Networks", *MathWorks Help Center*,
<https://it.mathworks.com/help/pde/ug/solve-poisson-equation-on-unit-disk-using-pinn.html>

[20] "Partial Differential Equation Toolbox", *MathWorks Help Center*,
<https://it.mathworks.com/help/pde/>

[21] "Deep Learning Toolbox", *MathWorks Help Center*,
<https://it.mathworks.com/help/deeplearning/>

[22] "dlarray", *MathWorks Help Center*,
<https://it.mathworks.com/help/deeplearning/ref/dlarray.html>

[23] "dlgradient", *MathWorks Help Center*,
<https://it.mathworks.com/help/deeplearning/ref/dlarray.dlgradient.html>

[24] "minibatchqueue", *MathWorks Help Center*,
<https://it.mathworks.com/help/deeplearning/ref/minibatchqueue.html>

[25] "arrayDatastore", *MathWorks Help Center*,
<https://it.mathworks.com/help/matlab/ref/matlab.io.datastore.arraydatastore.html>

[26] "dlfeval", *MathWorks Help Center*,
<https://it.mathworks.com/help/deeplearning/ref/dlfeval.html>

[27] "Monitor Deep Learning Training Progress", *MathWorks Help Center*,
<https://it.mathworks.com/help/deeplearning/ug/monitor-deep-learning-training-progress.html?lang=en>

[28] "adamupdate", *MathWorks Help Center*,
<https://it.mathworks.com/help/deeplearning/ref/adamupdate.html>

[29] "delaunay", *MathWorks Help Center*,
<https://it.mathworks.com/help/matlab/ref/delaunay.html>.

[30] "Electrical Transformers, What Transformers Are and How They Work", *Maddox Transformer*,
<https://www.maddoxtransformer.com/electrical-transformers>.

[31] Hernando, A., Crespo, P., Marín, P., González, A. "Magnetic Hysteresis", *Encyclopedia of Materials: Science and Technology*, pp. 4780-4787, Elsevier Science Ltd., 2001.

[32] Wang, J., Lin, H., Fang, S., Huang, Y. "A General Analytical Model of Permanent Magnet Eddy Current Couplings", *IEEE Transactions on Magnetics*, Vol. 50, No. 1, Jan. 2014.

[33] Teixeira, F.L. "Time-Domain Finite-Difference and Finite-Element Methods for Maxwell Equations in Complex Media", *IEEE Transactions on Antennas and Propagation*, Vol. 56, No. 8, Aug. 2008.

[34] Mattey, R., Ghosh, S. "A Physics Informed Neural Network for Time–Dependent Nonlinear and Higher Order Partial Differential Equations", *Computer Methods in Applied Mechanics and Engineering*, Jun. 2021.