4. Code Overview

In this chapter, we first present and motivate the coding choices in terms of environment and libraries. Then, we devote a section to Object-Oriented Programming in Python, focusing on the features implemented in the project. The final section presents the structure of the repository and the content of the configuration, data and outputs files.

4.1. Working Environment

The code implemented in the project - starting from scratch - is contained in the GitLab repository `PACS-BPINNS`, that can be found here, accompanied by a `README`.
In Appendix A, we present the procedure to follow, for the different operating systems, to set-up our working environment and download all the required packages, that are listed in the file `requirements.txt`[6] and presented more in detail in subsection 4.2.

4.1.1. Interpreter

The code is written in Python, and in order to run the scripts contained in the executable repository, the user needs `python` version 3.10.* (download from here).
The constraint on the version is due to the presence of the `match-case` statement, introduced in Python 3.10, that is particularly suitable to mimic the behaviour of `switch` in Java or C++ and guarantees more readability than chains of conditional statements when we need to order a different instruction for each case.

4.1.2. Virtual Environment

Our choice of work setting was to code inside a Python Virtual Environment, because this programming language relies a lot on external dependencies and version updates are frequent. Indeed, with the choice of creating and managing separate environments for different projects we enable a safe management of versions: each environment can use different versions of package dependencies and/or Python, and this ensures that projects are not able to cause dependency conflicts for one another.

Moreover, this choice can also simplify the reproduction of the results from other programmers, because they can install the required packages without creating conflicts with the ones already present on their machines.
This tools are provided by the module `virtualenv` (version 20.14.*, downloadable from here).

Another possibility could have been to set up a Conda environment, but being interested in an environment for this single project, we preferred to rely on this Python's native feature (available after Python 3.3), without requiring any third-part distribution.

4.2. Libraries

In the code, several popular Python libraries are exploited; we now list them all, deepening more into the ones that are relevant for the implementation.

4.2.1. Built-in Packages

In this project we also relied on few common Python modules for simpler and side tasks.
All these modules are among the *Python Runtime Services* or belong to the *Python Standard Library*; therefore, they do not appear in `requirements.txt` and they are directly distributed with Python.

`os`, module providing a portable way of using operating system dependent functionality. We mainly exploit the submodule `os.path`, that is used to manipulate file paths.

`shutil`, module offering a number of high-level operations on files and collections of files.

`json`, module used to work with `JSON` data, that consist in text, written with JavaScript object notation.

`argparse`, module that enables to write user-friendly command-line interfaces.

`warnings`, module for printing user-defined warnings.

`time, datetime`, modules supplying time-related functions and classes for manipulating dates and times.

[6] This file presents more libraries than the ones installed and described in subsubsection 4.2.2, because it already contains all the related dependencies of the presented ones.

For debugging purposes we relied on `pdb`, that is a module defining an interactive source code debugger for Python programs. By inserting in the desired point of the code `import pdb; pdb.set_trace()`, the user can break into the debugger, and then continue running until the next breakpoint by typing `continue`.

4.2.2. External Packages

The massive presence of tailor-made libraries provided both by important open source projects and big tech companies such as Google and Meta is one of the strenghts of Python's languange; in this section, we present the main external libraries for Data Science and ML tasks exploited in the code.

Numpy

`numpy` is one of the most popular Python packages for scientific computing; it provides a multidimensional array object (the `numpy.array`), various derived objects such as masked arrays and matrices, and an assortment of routines for fast operations (especially of linear algebra) on arrays.
We underline some features of the `numpy.array` type compared to Python native `lists`:

- differently than Python's `list` type, it has a strict requirement on the homogeneity of the objects stored and supports element-wise operations;
- with respect to a `list`, a `numpy.array` is much faster and its elements are stored contiguously in memory;
- a `numpy.array` supports item assignment; this feature will not be shared by another structure used in the project: Tensorflow's `Tensor` (subsubsection 4.2.3).

Matplotlib

`matplotlib.pyplot` is a user-friendly interface to `matplotlib`, which is the comprehensive library for visualization in Python. It consists of a collection of functions that make `matplotlib` work similarily to `MATLAB`. Each `pyplot` function makes some action on a figure: for example, it creates a figure, plots some lines in a plotting area, decorates the plot with labels, titles and legends...

Scipy

`scipy.stats` is the module devoted to Statistics within SciPy (another common library for scientific computing); it offers a large number of probability distributions, summary and frequency statistics, correlation functions and statistical tests. We exploit in particular the submodule `qmc`, providing Quasi-Monte Carlo generators and associated helper functions.

Tqdm

`tqdm` is a module for displaying progress bars during iterative procedures. The computational overhead added by this feature is low (about 60ns per iteration), according to the tests (see the documentation), and lower than other similar tools such as *ProgressBar*. Moreover, `tqdm` uses smart algorithms to predict the remaining time and to skip unnecessary iteration displays, which allows for a negligible overhead in most cases.

4.2.3. TensorFlow

`tensorflow` (`tf`) is the most relevant requirement for the NN implementation and it is a free and open-source software library for machine learning developed by the Google Brain Team in 2015.
Multidimensional arrays of elements are handled in TensorFlow within `tf.Tensor` objects.
This data structure is characterized by the following properties:

- a single data type (`float32`, `int32`, or `string`, for example);
- a shape, representing the length of each `Tensor` axes.

We also remark two fundamental differences with respect to the `numpy.array` type:

- `Tensor`s have accelerator support, like GPU and TPU;
- `Tensor`s do not support item assignment.

In the code, we work with the TensorFlow version 2.9.1, and, more specifically, we download the `tensorflow-cpu` library to avoid unnecessary warnings due to the absence of GPU on the machines that we used to run the code, but if a CUDA-compatible GPU is available, by choosing `tensorflow-gpu` or `tensorflow` (that will then detect the presence of GPU), one is able to run the code on GPU (see here for more details). However, for this project the computational time required was always feasible for laptops without GPU, given the limited size of the neural network employed.

Automatic Differentiation

Automatic differentiation was already introduced in subsection 2.2. TensorFlow can automatically compute the gradients for the parameters in a model, which are then used in algorithms such as backpropagation. To do so, the framework must keep track of the order of operations done to the input `Tensors` in a model, and then compute the gradients with respect to the appropriate parameters.

To keep track of the operations, we perform automatic differentiation tasks inside a `tf.GradientTape`; this is a *context manager* that enables to record the operations inside it and to calculate gradients with respect to given variables. In this context, it is required that the variables are *watched* by the `tape`, and this happens by default only with trainable variables. Therefore, for an arbitrary tensor as the network input, we need to call the `tape.watch()` function.

Other major features

The TensorFlow framework contains other useful features for tasks involved in the training of NNs that are built for the `Tensor` data structure and automatic differentiation purposes, such as:

- Eager execution: a mode in which operations are evaluated immediately as opposed to being added to a computational graph which is executed later. With this type of execution, the code can be examined step-by step-through a debugger, since data is augmented at each line of code rather than later in a computational graph.
- API access to the most common losses, metrics, optimizers and math operations for `Tensor` objects, implemented to be compatible with automatic differentiation and gradient tapes.
- Keras submodule: `tensorflow.keras` is a high-level API of TensorFlow providing many tools for neural network implementation that are designed to be user-friendly. Keras indeed offers consistent and simple APIs which minimize the number of user actions required for common use cases. We rely on it for an easy access to built-in activation and loss functions, parameter initializers, layers and models.

4.3. Object Oriented Features

Python belongs to the family of Object Oriented Programming (OOP) Languages, representing a paradigm based on the concept of "objects" which can contain data and code, rather than on "procedures" which contain a series of computational steps to be carried out.
Python provides a variety of typical OOP features (such as encapsulation, inheritance, polymorphism. . .), but clearly, as other OOP languages, it can be used even for programming in a procedural style.
In this section, we present some modules and decorators[7] exploited to give intuitive properties to objects and data structures that appear in the B-PINNs framework. Later in section 5 we will specify the classes in which we have introduced them and the reasons behind the choice, while here we propose a general overview on the main functionalities exploited.

4.3.1. Inheritance

In the code, you can find several examples of the OOP concept of inheritance, having sub-classes which are built on parent classes: we have examples of both *single inheritance*, where subclasses inherit the features of one superclass, and *multiple inheritance*, where one class has more than one superclass and inherits from all.

When dealing with inheritance, we often override methods and we need to choose which method to refer to between the child's and the parent's one. This holds for all methods, also for special ones like constructors.
Indeed, given one parent class, when building a subclass upon it we may want to define a custom constructor for the child class, that will be called when instantiating the class; otherwise, the constructor of the parent class will be called. In other cases, like ours, we may want to define a constructor for the child and use also the constructor of the parent class. In this case, we rely on the Python built-in function `super()`.

MRO and `super()` function

While dealing with inheritance, `super()` allows us to call methods of the superclass in our subclass, therefore it becomes fundamental when accessing inherited methods that have been overridden. The primary use case of this is to extend the functionality of the inherited method. Technically speaking, `super()` returns a proxy object that delegates method calls to the parent class.

[7] In Python, a decorator is a design pattern that allows to modify the functionality of a function by wrapping it in another function.

Having to deal with calls to parents' methods, when the inheritance tree is a complex structure it becomes determinant to understand which is the family tree, namely what is the *Method Resolution Order* (MRO). `super()` provides the next method according to the MRO, on which you can find information in each class's private class attribute `__mro__`. The MRO consists in ordering parent classes, following the order indicated in the child class declaration and completing each branch; when common ancestors are found, they are always delayed to the end of the branch. The process stops at the "object" class.

In our context, we use `super()` to retrieve the constructor of the parent class, to which we pass the arguments required to instantiate an object of the parent class. Basically, for the constructors, when we call `super().__init__()` it provides the next `__init__()` method according to the used MRO algorithm in the context of the complete inheritance hierarchy.

`super()` can also take two parameters, which determine from which point in the tree to start looking for methods: the first is the subclass, and the second parameter is an object that is an instance of that subclass.

```python
class First():
    def __init__(self):
        print("Building First....")
        self.x = "Attribute First"

class Second(First):
    def __init__(self):
        super(Second, self).__init__()
        print("Building Second...")
        self.y = "Attribute Second"

class Third(First):
    def __init__(self):
        super(Third, self).__init__()
        print("Building Third...")
        self.z = "Attribute Third"

class Fourth(Second,Third):
    def __init__(self):
        super(Fourth, self).__init__()
        print("Building Fourth...")
```

Figure 7: Single and Multiple Inheritance in Python

Diamond inheritance

In our implementation for the classes in subsection 5.6, we rely on a family tree that is similar to the so-called *diamond shape*: this consists of a class having two parents that share their parent, as the ones considered in Figure 7, indeed, the class `Fourth` inherits both from `Second` and `Third` which inherit both from `First`.

With the code shown above, an instance of `Fourth` has the attributes `y` and `z` from the direct parents and `x` from the "grandparent" class. An instance of `Second` has only the attributes `y` and `x`, while an instance of `Third` has only the attributes `z` and `x`.

The chain of calls to `super()` are used to activate all the ancestors' `__init()__`, and in our case are also used to pass the arguments to the ancestors' constructors, coming from the structure defined in subsubsection 5.2.3.

4.3.2. Abstract Base Class

Abstract classes are used to represent general concepts, which can be used as base classes for concrete classes. In our case, the abstract concept will be for example the one of a training algorithm, where the training pipeline is common to all the methods, but the recipe to sample new parameters changes according to the specific

algorithm. So, we want to store all the common features in an abstract class that can't work without being used in a specific children (more details in subsection 5.7).

With this inheritance feature, we separate, in a certain sense, the interface from the implementation, because abstract base classes only define generic methods and properties, while a part of their implementation is then handled by the concrete subclasses, of which we can instantiate objects that can handle tasks.

In Python, the module `abc` provides the infrastructure for defining them. From this module, we import specifically `ABC`, the built-in class that is passed in the class declaration as parent class to the classes that we want to make abstract. This forbids the instantiation of the abstract class.

From `abc`, we import as well the decorator `@abstractmethod`, that can be applied to methods of an abstract class if we want to force their overriding in children classes. This prevents us to create instances of classes with abstract parents if not all their abstract methods (also called virtual methods) have been overridden.

This features help to avoid bugs and make the class hierarchies easier to maintain by providing strict recipes to follow when coding methods in the subclasses.

4.3.3. Dataclass

It may happen that we are interested in storing data and exploiting at the same time some features of OOP, but we would like to have an object more specific and light than a traditional class (e.g. subsubsection 5.3.4). From Python 3.7, for this task we can rely on Data Classes that are characterized by the `@dataclass` decorator, provided by the `dataclasses` module and they consist in Python classes with some limitations, thought mainly for containing data (although there is no strict restriction about that).

They allow the user to define classes with less code, because for example, methods such as `__init__()` and `__repr__()` are automatically added without requiring their implementation and all methods are class methods so we do not require the typically redundant `self` syntax of classes in this simpler context.
They also provide more functionalities out of the box; for example, with the option `@dataclass(frozen=True)` we can define a class whose instances are immutable, and with methods such as `asdict()`, `astuple()` an object of a dataclass can be converted to a dictionary or a tuple, other natural data structures for the storage of data.

4.3.4. Iterators

In Python, there is the possibility to create classes as iterators, which are objects that you can traverse through all its values; in our project, they are useful for defining training batches (subsubsection 5.4.3).
An example of implementation of iterators is shown in Figure 8:

- `__init__()` initializes the data attribute that is expected to be an iterable;
- `__iter__()` returns the iterator object. In our example, this is the Data object on which it is called itself using `iter()` on the instance. We initialize the `current_index` with zero, to start with the first datum;
- `__next__()` returns the next value after one iteration. We increment the `current_index` attribute to keep track of the current index of the element in data. This special method is called when `next()` is invoked on the instance.

```
1  class Data:
2      def __init__(self, data):
3          self.data = data
4      def __iter__(self):
5          self.current_index = 0
6          return self
7      def __next__(self):
8          if self.current_index < len(self.data):
9              x = self.data[self.current_index]
10             self.current_index += 1
11             return x
12         raise StopIteration
```

Figure 8: `__iter__()` and `__next__()` special methods

4.3.5. Property

Properties can be considered the "Pythonic" way of working with attributes, which are massively present in the data management (subsubsection 5.3.4) because as showed in subsection 2.2 datasets for PINNs are articulated.

Properties' main strengths are the following:

- you can access instance attributes exactly as if they were public attributes while using intermediaries (getters and setters) to validate new values and to avoid accessing or modifying the data directly;
- by using `@property`, you can reuse the name of a property to avoid creating new names for the getters, setters, and deleters, hence the syntax is concise and readable.

In Figure 9 we show with a basic example the syntax to define the custom functions with this decorator.

```
1  class Person:
2      def __init__(self, firstname, lastname):
3          self.first = firstname
4          self.last = lastname
5
6      @property
7      def fullname(self):
8          return self.first + ' ' + self.last
9
10     @fullname.setter
11     def fullname(self, name):
12         firstname, lastname = name.split()
13         self.first = firstname
14         self.last = lastname
15
```

Figure 9: Example with `@property` decorator

4.4. Repository Structure

The detailed description of the repository starts in this section, with a deepening into the purpose and content of the folders `config`, `data` and `outs`.
The presentation of the folder `src`, containing the actual implementation, is postponed to section 5.

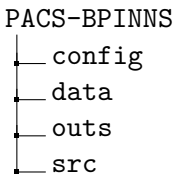

Figure 10: `PACS-BPINNS` repository

4.4.1. Configuration - `config` Folder

This folder contains the `.json` files for the configuration of the options of test cases, which are the first instructions read by the main executable script and contain information about algorithm parameters, network architecture, dataset choices or generation and various utilities. The files are divided into two main subfolders:

- `test_models`, whose files are work-in-progress test cases;
- `best_models`, containing options of test cases that gave rise to interesting results.

Apart from the distinction in the folder of membership, all configuration files have the same structure: they store `json` dictionaries of dictionaries where the first level of keys represents different categories of settings to be chosen, while the second level of keys consists of the names of the variables to be set.

The first level-keys describing clusters of options are:

1. `general`, storing:
   - `problem`, string with the name of the problem to solve (eg. `Regression` for function interpolation, or the name of the PDE for Physics-Informed tasks);
   - `case_name`, string with a name identifying the specific dataset for the experiment;
   - `method`, string containing the training algorithm to choose;
   - `init`, string that is empty to indicate that there is no pre-training, or contains the name of the desired pre-training method.

2. `architecture`, containing information on network architecture:
    - `activation`, string with the desired activation function; we chose the *swish* function;
    - `n_layers`, number of hidden layers L;
    - `n_neurons`, number of neurons per hidden layer K_l.
3. `losses`, containing a boolean for each part of the loss to include in the total loss used for learning;
    - `data_u`, representing the component of solution fitting (Equation 16);
    - `data_f`, representing the component of parametric field fitting (Equation 18);
    - `data_b`, representing the component of boundary loss (Equation 17);
    - `pde`, representing the component of the PDE residual (Equation 23);
    - `prior`, representing the prior distribution of network parameters (Equation 19).
4. `metrics`, containing the same keys of `losses` to establish which component of the loss function we want to compute (regardless of the fact that we are using it for learning) and plot its history.
5. `num_points`, containing the number of data points to use:
    - `sol`, for solution fitting points;
    - `par`, for parametric field fitting points;
    - `bnd`, for boundary points;
    - `pde`, for collocation points.
6. `uncertainty`, containing the values of the std linked to noise and uncertainty mentioned in subsection 2.4:
    - `sol`, σ_u for the solution;
    - `par`, σ_f for the parametric field;
    - `bnd`, σ_b for the boundary data;
    - `pde`, σ_r for the PDE residual.
7. `utils`, containing:
    - `random_seed`, the random seed set;
    - `debug_flag`, boolean to print details useful for debugging tasks;
    - `save_flag`, boolean to save the on-going test in a specific folder in `outs` (see subsubsection 4.4.3);
    - `gen_flag`, boolean to generate the dataset requested; it is necessary to set it to true only when the dataset is not already present in the `data` folder.
8. {method_name}[8], dictionary that stores the parameters of the training algorithm.
9. {method_name}_0[8], dictionary that stores the parameters of the method used to pre-train (optional).

The keys of the last two dictionaries are the method names; the second-level keys of the latter two dictionaries depend on the method chosen; for the algorithms implemented in the project, we have:

1. ADAM (algorithm 3)
    - `epochs`, number of epochs N;
    - `burn_in`, delay for the physical loss in the learning process;
    - `lr`, learning rate η.
    - `beta_1`, exponential decay rate for the first moment estimates β_1;
    - `beta_2`, exponential decay rate for the second moment estimates β_2;
    - `eps`, small number to prevent any division by zero ϵ;
2. HMC (algorithm 4)
    - `epochs`, total number of samples N;
    - `burn_in`, burn-in B and delay for the physical loss in the learning process;
    - `skip`, number of samples to skip among subsequent samples S;
    - `HMC_L`, number of leap-frog steps L;
    - `HMC_dt`, temporal step Δt;
    - `HMC_eta`, mass matrix parameter α.
3. VI (algorithm 5)
    - `epochs`, number of epochs N;
    - `burn_in`, delay for the physical loss in the learning process;
    - `samples`, number of desired samples of $\boldsymbol{\theta}$;
    - `alpha`, learning rate while learning $\boldsymbol{\zeta}$.
4. SVGD (algorithm 6)
    - `epochs`, number of epochs E;
    - `burn_in`, delay for the physical loss in the learning process;
    - `N`, total number of particles N;
    - `h`, bandwidth in kernel definition h;
    - `eps`, step size ε.

[8] The name of the dictionary is one among the algorithms implemented: `ADAM`, `HMC`, `VI`, `SVGD`.

4.4.2. Datasets - `data` Folder

This folder contains the data generated for the proposed test cases, stored in `.npy` files (NumPy array files). Each time the `gen_flag` cited in subsubsection 4.4.1 is set to `True`, a folder for the current test case is generated or overwritten.

If instead the data do not need to be generated, but are given from external sources or have already been created, the user needs to set the `gen_flag` to `False`, and, in the first case, upload the files following the same name convention of the created data.

As shown in Figure 11, data are separated by type of problem in folders named after the task (for the `Regression` case), or the differential equation in the physics-informed problem (`Laplace1D`, `Oscillator`...).

Then, a second level of folders can be found where we store the specific dataset for that problem, to distinguish, for example, the Laplace 1D problem with different functions.

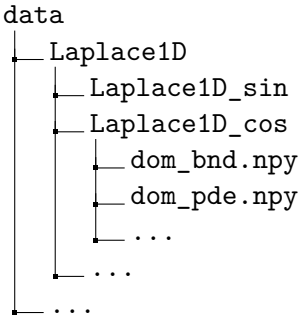

Figure 11: Hierarchy of the folder `data`.

Inside each subfolder, we finally find the data, split into different files:

- **Boundary data:** `dom_bnd.npy`, `sol_bnd.npy` for the inputs and solution values of boundaries;
- **Collocation data:** `dom_pde.npy` for the inputs of collocation points;
- **Fitting data:** `dom_sol.npy`, `sol_train.npy` and `dom_par.npy`, `par_train.npy` for inputs and values for solution or parametric field;
- **Test data:** `dom_test.npy`, `sol_test.npy`, `par_test.npy` for inputs and values of test points.

4.4.3. Outputs - `outs` Folder

This folder contains the results and the summarizing statistics of each test case and the folder hierarchy (see Figure 12) is similar to the `data` folder: the first level denotes the type of problem, the second enables to fully characterize the test case by specifying the dataset.

There is another level of indentation: inside each subfolder, each run of the code with the option `save_flag` equal to `True`, generates a folder, whose name contains date and time of the execution.
The `trash` folder instead contains the output of the latest test case run with the `save_flag` equal to `False` and is cleaned each time.

Inside each test case folder, there are the following subfolders:

1. `log`, storing the `.txt` files:
   - `parameters.txt`, reproducing the `.json` configuration file;
   - `errors.txt`, reproducing the output on the screen after each run, with relative errors on solution (and eventually parametric field) and uncertainty quantification,
   - `keys.txt`, with the components of the loss used in the learning process;

   and the `.npy` files:
   - `loglikelihood.npy`, history of values of the log-posterior (up to a constant);
   - `posterior.npy`, history of values of the posterior (up to a constant);

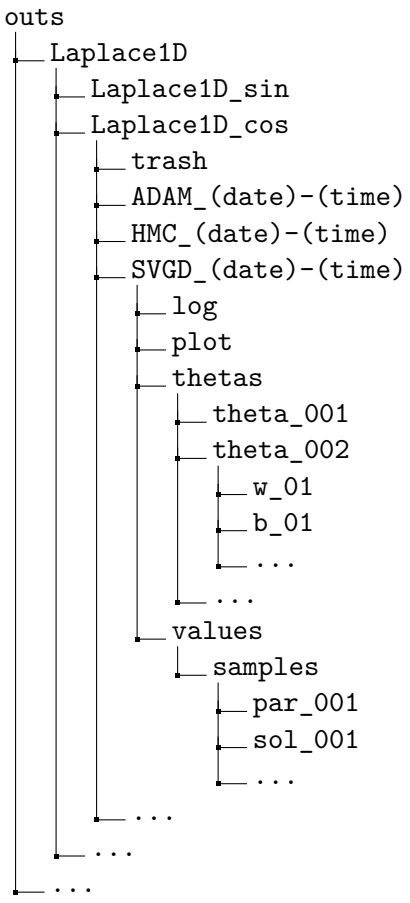

Figure 12: Hierarchy of the folder `outs`.

2. `plot` storing the graphical output. The files that can be found are:
    - `u_confidence.png` mean of the predicted distribution for the solution, with confidence interval given by mean ± standard deviation;
    - `u_nn_samples.png` plot of several samples from the predicted distribution for the solution;
    - `f_confidence.png` mean of the predicted distribution for the parametric field, with confidence interval given by mean ± standard deviation;
    - `f_nn_samples.png` plot of several samples from the predicted distribution for the parametric field;
    - `Mean Squared Error.png`, plot of the MSE;
    - `Loss (Log-Likelihood).png`, plot of the log-posterior (up to a constant);
3. `thetas`, storing all the sampled network parameters. For each sample of parameters, a subfolder named `theta_(number)` is created, containing as many `.npy` files as the double of the number of layers: each file indeed stores the values of the weights or of the biases in each layer;
4. `values`, containing:
    - `sol_NN.npy` mean of the predicted distribution for the solution;
    - `sol_std.npy` standard deviation of the predicted distribution for the solution;
    - `par_NN.npy` mean of the predicted distribution for the parametric field;
    - `par_std.npy` standard deviation of the predicted distribution for the parametric field;
    - the subfolder `samples`, contaning in `.npy` files the values of solution and parametric field corresponding to each selected sample of network parameters.