Neural network preconditioning of large linear systems

Maksym Shpakovych

Neural network preconditioning of large linear systems

Maksym Shpakovych

TECHNICAL REPORT
N° 0518
October 2023
Project-Teams Concace

ISSN 0249-0803 ISRN INRIA/RT--0518--FR+ENG

Neural network preconditioning of large linear systems

Maksym Shpakovych

Project-Teams Concace

Technical Report n° 0518 — October 2023 — 37 pages

Inria centre
at the University of Bordeaux

200 avenue de la Vieille Tour
33405 Talence Cedex

Abstract: We report research results on the training and use of an artificial neural network for the preconditioning of large linear systems. The aim is to solve a large linear system, which is a discretisation of a Helmholtz differential equation, as quickly as possible using a preconditioned Krylov-like iterative method such as Flexible GMRES. The implementations are written in Python and are divided into two main repositories: one for the Flexible GMRES algorithm available at `https://gitlab.inria.fr/mshpakov/axb`, and another for learning neural networks available at `https://gitlab.inria.fr/mshpakov/maxb`. Currently, a preconditioning matrix is built column by column by solving a set of large sparse block structured least squares problems. This method is relatively easy to implement, but suffers from high computational cost and consequently high energy consumption. Furthermore, although we are considering a parametrized family of linear systems, we have to recalculate a preconditioner each time the parameters change. Since a neural network aims to approximate a complex functional dependence of any kind, it is a natural idea to use it to build a parameter invariant preconditioner to speed up convergence. We will present in detail a different approach to learning the neural network, along with several architectures that have been tested in this work. We will show with some examples that the naive approach of building a preconditioner in an unsupervised way suffers from the lack of knowledge of the data distribution of the Krylov basis vectors that appear during the FGMRES iterations. As a result, this paper demonstrates a particular iterative algorithm for learning a network. The main difficulty in implementing this type of algorithm stems from the fact that the differentiation of several iterative steps becomes numerically unstable and leads to the explosion of gradients. To overcome this difficulty, we use a replay buffer during training, which is a known approach from the reinforcement learning domain where it is used to collect experience for an agent parameters update. We will detail the different learning strategies where the replay buffer is used and explicitly show its advantages and any numerical challenges that arise. We will also present the implementation details of different learning strategies and neural network architectures.

Key-words: Learning, numerical linear algebra, neural networks, preconditioning, subspace methods, linear systems

Préconditionnement par réseaux de neurones pour des systèmes linéaires de grande taille

Résumé : Nous présentons des résultats de recherche sur la formation et l'utilisation d'un réseau neuronal artificiel pour le préconditionnement de grands systèmes linéaires. L'objectif est de résoudre un grand système linéaire, qui est une discrétisation d'une équation différentielle de Helmholtz, aussi rapidement que possible en utilisant une méthode itérative préconditionnée de type Krylov telle que Flexible GMRES. Les implémentations sont écrites en Python et sont divisées en deux dépôts principaux : l'un pour l'algorithme Flexible GMRES et l'autre pour l'apprentissage des réseaux neuronaux. Actuellement, une matrice de préconditionnement est construite colonne par colonne en résolvant un ensemble de problèmes de moindres carrés structurés par blocs. Cette méthode est relativement facile à mettre en œuvre, mais elle souffre d'un coût de calcul élevé et, par conséquent, d'une forte consommation d'énergie. En outre, bien que nous considérions une famille paramétrée de systèmes linéaires, nous devons recalculer un préconditionneur chaque fois que les paramètres changent. Étant donné qu'un réseau neuronal vise à approximer une dépendance fonctionnelle complexe de tout type, il est naturel de l'utiliser pour construire un préconditionneur invariant par rapport aux paramètres afin d'accélérer la convergence. Nous présenterons en détail une approche différente de l'apprentissage du réseau neuronal, ainsi que plusieurs architectures qui ont été testées dans le cadre de ce travail. En conséquence, ce document démontre un algorithme itératif particulier pour l'apprentissage d'un réseau. La principale difficulté dans la mise en œuvre de ce type d'algorithme provient du fait que la différenciation de plusieurs étapes itératives devient numériquement instable et conduit à l'explosion des gradients. Pour surmonter cette difficulté, nous utilisons un tampon de relecture pendant l'apprentissage, qui est une approche connue dans le domaine de l'apprentissage par renforcement où il est utilisé pour collecter de l'expérience pour une mise à jour des paramètres de l'agent. Nous détaillerons les différentes stratégies d'apprentissage dans lesquelles le tampon de relecture est utilisé et nous montrerons explicitement ses avantages et les défis numériques qui se posent. Nous présenterons également les détails de la mise en œuvre des différentes stratégies d'apprentissage et architectures de réseaux neuronaux.

Mots-clés : Apprentissage, Algèbre linéaire numérique, réseaux de neurones, préconditionnement, méthode de sous-espaces, systèmes linéaires

Contents

1 Introduction

The problem of solving a large linear system arises in many applied problems, the main source of which is the discretisation of a differential equation [15]. In this paper we aim to simulate the propagation of electromagnetic waves to model a radar cross section of a plane by solving a discretised version of the Helmholtz differential equation with an absorption boundary condition using an iterative linear system solver such as the Flexible GMRES algorithm [17, 15]. It is known that a discretised linear system becomes badly conditioned as the wavenumber in the Helmholtz equation increases, which affects the convergence rate of iterative solvers [5]. The introduction of a preconditioned matrix, whose aim is to reduce the conditioning number of the system, is usually the solution to this difficulty [15]. Preconditioner depends on the shape of the domain and on equation parameters such as a wavenumber, meaning that it has to be recalculated every time some of them change.

To compute a preconditioner numerically, a large number of methods have been proposed in the numerical linear algebra community, covering different physics modelling tasks [4]. For instance, Airbus, the company from which the task originated, computes each column of a sparse preconditioned matrix by solving a large sparse block structured least squares problem for the discretised Helmholtz equation [2, 3]. This approach requires large computational resources due to frequent matrix recalculations. A natural question that arises is whether it is possible to avoid these recalculations by exploiting the structure of the equation. Since a neural network aims to approximate a complex functional dependence of any kind, it is a natural idea to use it for our purpose. Thus, we propose to utilise a neural network as a preconditioner in the Flexible GMRES iterative solver, where the main goal is to train it only once and then use it without retraining for different domain sizes and equation parameters.

In this paper, we try different neural network architectures to solve the task and satisfy the generalisation constraints mentioned above. One of the most popular architectures that satisfies the requirements is the U-Net [14]. This is a convolutional neural network with a special structure, which means that it can be used for different domain sizes and can naturally plug additional inputs, such as the wavenumbers, as additional channels in the input tensor. For example, in [17] the authors train the U-Net model as a fixed-point iterative solver for the two-dimensional Helmholtz equation modelling the propagation of ultrasound waves inside the human skull with different sound speed distributions inside the bone. Another recent novel architecture that matches the requirements is a Fourier Neural Operator (FNO) [9]. This is also a convolutional neural network, but the convolutional operation is computed in the frequency domain using direct and inverse Fast Fourier Transforms. It takes a subset of all frequencies as they are and set others to zero, on the hypothesis that the solution can be represented as a combination of low frequencies.

Since the goal is to solve the differential equation, we were also interested in physics informed neural networks (PINN), which aim to directly approximate a solution to the equation. Although this approach does not satisfy our generalisation requirements, because each time when domain or parameters change we need to retrain a neural network, it gave us an idea of a loss function for our applications. One possible solution is to create a dataset of solved problems and compute a simple mean squared error between the model prediction and the true solution, which puts us in the context of supervised learning. This approach is simple to implement but requires additional computational effort to create a dataset. However, the PINN framework proposes to minimise the norm of the residual of a differential equation, where the differential operators are evaluated using the automatic differentiation algorithm [12, Section 8.2]. This approach does not require a dataset of solved problems, which makes learning unsupervised, but increases the cost of loss computation. The compromise used in this paper is to compute a discretised version of the

residual using the Fourier spectral method [16], which allows the calculation of derivatives. This is faster than automatic differentiation and still allows unsupervised learning. The disadvantage is that we are forced to compute derivatives at every point, even if we only need some of them, which is the case for classical PINN learning.

The outline and the contribution of the article are as follows. In Section 2 we define the problem to be solved and the step-by-step high-level explanation of the solution. In Section 3 we explain the evolution of the research to give a clear picture of the starting point and the results obtained. Section 4, contains the main result of this work, which is a preconditioner constructed by a neural network using a direct approximation of the inverse operator. In Section 6, we give ideas for future research directions that may lead to better results. The details of the numerical implementation of the algorithms are given in Section 5.

Notation

Let f be a function from $\mathbb{R}^n$ to $\mathbb{R}$, let $\{x_0, \dots, x_{N-1}\} \in \mathbb{R}^{n\times N}$ be a set of points where f is evaluated for some $N > 0$. Then $\hat{\mathbf{f}} = \{\hat{f}_0, \dots, \hat{f}_{N-1}\} \in \mathbb{R}^N$ is the discretisation of f where $\hat{f}_k = f(x_k)$. The bold symbols $\hat{\mathbf{f}}$, $\mathbf{x}$, $\mathbf{u}$, ... are always reserved to denote vectors of real or complex numbers. The neural network model is defined by letter $\mathcal{N}_\theta : \mathbb{R}^n(\text{or } \mathbb{C}^n) \to \mathbb{R}^n(\text{or } \mathbb{C}^n)$, where θ denotes the trainable parameters.

2 Problem definition

Let us consider the Helmholtz differential equation

$$\big(\nabla^2 + k(\mathbf{x})^2\big)u(\mathbf{x}) = \rho(\mathbf{x}), \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^d$ is a point in the d-dimensional domain, $\nabla^2 := \sum_{j=1}^d \frac{\partial}{\partial x_j}$ is the Laplace operator, $k : \mathbb{R}^d \to \mathbb{R}$ is the wavenumber function, $\rho : \mathbb{R}^d \to \mathbb{R}$ is the source function, and $u : \mathbb{R}^d \to \mathbb{C}$ is the complex acoustic wavefield. The wavenumber is defined as $k(\mathbf{x}) = \frac{\omega}{c(\mathbf{x})}$, where $c : \mathbb{R}^d \to \mathbb{R}_+$ is a speed of sound distribution function and $\omega \in \mathbb{R}_+$ is an angular frequency of the source. In our work we solve (1) subject to the Sommerfeld radiation condition at infinity

$$\lim_{|\mathbf{x}|\to\infty} |\mathbf{x}|^{\frac{d-1}{2}} \left(\frac{\partial}{\partial|\mathbf{x}|} - i\frac{w}{c_0}\right) u(\mathbf{x}) = 0. \tag{2}$$

It is considered that $c(x)$ is heterogeneous in a bounded region of the domain $\Omega \subset \mathbb{R}^d$, while it is uniform and equal to c_0 outside of it. Sommerfeld radiation condition is satisfied by using the Perfectly Matching Layer [7] approach where the idea is to change derivatives as

$$\frac{\partial}{\partial \tilde{x}_j} \to \frac{1}{1 + i\frac{\sigma_j(x_j)}{\omega}} \frac{\partial}{\partial x_j}, \tag{3}$$

where $\sigma_j : \mathbb{R}^d \to \mathbb{R}$ for $j \in \{1, \dots, d\}$. Then, we consider only differential equation

$$\big(\tilde{\nabla}^2 + k(\mathbf{x})^2\big)u(\mathbf{x}) = \rho(\mathbf{x}), \tag{4}$$

where $\tilde{\nabla}^2 := \sum_{j=1}^d \frac{\partial}{\partial \tilde{x}_j}$ is the Laplace operator with PML. To solve it numerically we need to discretise the equation. According to [17], we use the Fast Fourier Transform to discretise the

Helmholtz operator by applying the differentiation theorem [16], which gives a relation between the Fourier Transform of a function $f : \mathbb{R} \to \mathbb{R}(or\ \mathbb{C})$ and its derivative f' as

$$\mathcal{F}\{f'\}(\omega) = i\omega\mathcal{F}\{f\}(\omega),$$

where $\mathcal{F}\{f\}$ is the continuous Fourier Transform of a function f. After discretisation, the problem (4) is transformed into the problem of solving the linear system

$$A(\hat{\mathbf{c}})\hat{\mathbf{u}} = \hat{\boldsymbol{\rho}}, \tag{5}$$

where $A(\hat{\mathbf{c}}) \in \mathbb{C}^{n\times n}$ is the discretised Helmholtz operator, which depends on the discretisation of the sound velocity distribution $\hat{\mathbf{c}} \in \mathbb{R}^n_+$, $\hat{\mathbf{u}} \in \mathbb{C}^n$ is the discretisation of the solution, $\hat{\boldsymbol{\rho}}$ is the discretisation of the source, $n \in \mathbb{N}$ is the number of points in the discretisation. It is known that the conditioning of the system (5) degradates with the increase of the wavenumbers $k(\mathbf{x})$ [...]. Thus, it is proposed to solve the system (5) by Krylow-like iterative linear system solvers such as GMRES [15].

Before going into details, let us quickly give the example of Python code to solve one-dimensional problem using implemented libraries `axb` and `maxb`.

```
1  # Example 1
2  from maxb import xde
3  from axb import gmres
4  from scipy.sparse.linalg import LinearOperator
5
6  # Make 1D domain.
7  domain = xde.Domain(256, (-40, 40), dtype='float64')
8
9  # Make Fourier gradient with PML.
10 grad = xde.grad.FourierGradient(domain)
11 grad = xde.PML(grad)
12
13 # Make linear operator.
14 oper = xde.operators.Helmholtz(grad)
15 A = LinearOperator(shape=(256, 256), dtype='complex128', matvec=oper.make_matvec())
16
17 # Make source.
18 source = xde.source.Gaussian(domain)
19 b = source(domain.grid)
20
21 # Solve system.
22 u, info = gmres(A, b)
```

In line 6 we create a one-dimensional domain in range from -40 to 40 with 256 points inside. The data type for generated grid is set to float64 explicitly. In line 9 we create a Fourier gradient on the defined domain and in line 10 we modify gradients calculations with respect to formula (3). In lines 13 and 14 we create the discretised Helmholtz operator and pass it to the linear operator interface from `scipy` library. In lines 17 and 18 we create the continious and discretised source. In line 21 we call GMRES to solve the system. Let us visualise the solution u and the source b on the Figure 1, the speed of sound is constant and equals 1 by default. Now we can introduce the main goal of this work in the context what is presented above.

The objective of this research is to speed up the Krylov-like iterative solver of a linear system by using the predonditioner $M : \mathbb{C}^n \to \mathbb{C}^n$ **constructed by means of a neural network**, where the operator M somehow approximates $A(\hat{\mathbf{c}})^{-1}$.

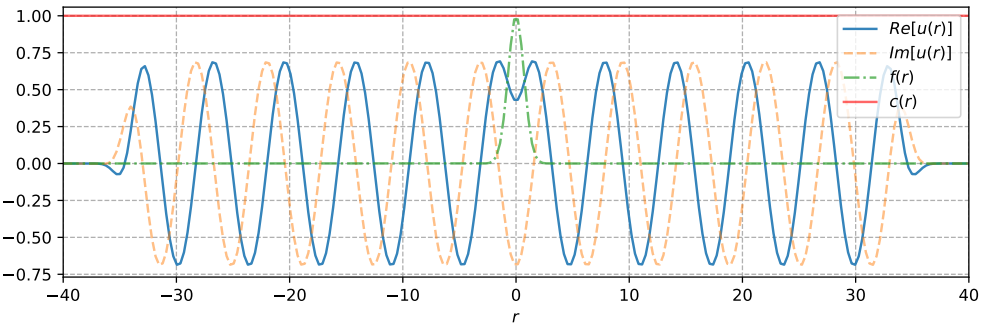


Figure 1: Solution $u \in \mathbb{C}^{256}$ and source $b \in \mathbb{R}^{256}$ for one-dimensional problem with constant $c = 1$

To provide a complete view we need to detail the process of discretisation with the discrete Fourier Transform, the use of PML, and the use of GMRES with a preconditioner.

2.1 Differentiation with Fourier Transform

In this section we consider the differentiation operation performed by the Fourier Transform. Its analogy in discrete space will be useful for discretising the Helmholtz operator. Let us start with the definitions of the Fourier Transform and its inverse.

Definition 2.1 ([16]). *Let f be an absolutely integrable function defined for all $t \in \mathbb{R}$, i.e.*

$$\int_{-\infty}^{\infty} |f(t)|dt < \infty.$$

Then there exists a mapping $\mathcal{F}\{f\} : \mathbb{R} \to \mathbb{C}$ which is called the Fourier Transform of f and is defined by

$$\mathcal{F}\{f\}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt, \tag{FT}$$

and its inverse is given by

$$\mathcal{F}^{-1}\{f\}(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} f(\omega)e^{i\omega t}d\omega. \tag{IFT}$$

Although we are working with discrete signals in this paper, we need Definition 2.1 to formulate the differentiation theorem, the discrete variant of which will be useful further on. But before, let us formulate a useful lemma for the following theorem.

Lemma 1. *Let $f(t)$ be an absolutely integrable function at each $t \in \mathbb{R}$ and limit $\lim_{t\to\pm\infty} f(t) = \mathcal{A}$ exists. Then we have that*

$$\lim_{t\to\pm\infty} f(t) = 0,$$

or equivalently $f(\pm\infty) = 0$.

Proof. By contradiction, let $\lim_{t\to\infty} |f(t)| = \mathcal{A} \neq 0$. Then, by definition of the limit we have that $\forall\varepsilon > 0, \varepsilon < \mathcal{A}$ exists $t_0 > 0$ such that for $t > t_0$ inequality $||f(t)| - \mathcal{A}| < \varepsilon$ holds. From where we obtain that $|f(t)| > \mathcal{A} - \varepsilon$, and thus $\int_{t_0}^{t} f'(\tau)d\tau > (\mathcal{A} - \varepsilon)(t - t_0)$ for $t \to \infty$ which contradicts the initial condition. The case for $t \to -\infty$ is proved similarly. □

Theorem 1 (Differentiation theorem)**.** *Let $f(t)$ be a differentiable function for all $t \in \mathbb{R}$ and $f'(t)$ its derivative such that they are absolutely integrable. Then we have*

$$\mathcal{F}\{f'\}(\omega) = i\omega\mathcal{F}\{f\}(\omega). \tag{6}$$

Proof. By definition, the functions $f(t)$ and $f'(t)$ are absolutely integrable for all $t \in \mathbb{R}$ and $f'(t)$ is continuous or piecewise continuous. Then the function $f(t)$ can be written as

$$f(t) = f(0) + \int_0^t f'(\tau)d\tau.$$

Since $\int_{-\infty}^{\infty} f'(t)dt$ converges by definition, then limits $\lim_{t\to\pm\infty}\int_0^t f'(\tau)d\tau$ exist and thus the limits $\lim_{t\to\pm\infty} f(t)$ also exist. Then, applying Lemma 1, the proof follows from integration by parts:

$$\begin{aligned}\mathcal{F}\{f'\}(\omega) &= \int_{-\infty}^{\infty} f'(t)e^{-i\omega t}dt \\ &= f(t)e^{-i\omega t}\Big|_{-\infty}^{\infty} - \int_{-\infty}^{\infty}(-i\omega)f(t)e^{-i\omega t}dt \\ &= i\omega\mathcal{F}\{f\}(\omega).\end{aligned}$$

□

Let us move to the discrete case and explain the relationship between the Fourier Transform and its discrete variant, and then adapt the result of the Theorem 1. But first, let us give a definition to direct and inverse discrete Fourier Transform.

Definition 2.2. *Let $\hat{\mathbf{f}} = \{\hat{f}_0, \ldots, \hat{f}_{N-1}\}$ be a sequence of real or complex numbers. Then the discrete Fourier Transform of $\hat{f}$ is defined by*

$$F\{\hat{\mathbf{f}}\}(n) = \sum_{k=0}^{N-1} \hat{f}_k e^{-i2\pi nk/N},\ n \in \{0, \ldots, N-1\}, \tag{DFT}$$

and its inverse is given by

$$F^{-1}\{\hat{\mathbf{f}}\}(k) = \frac{1}{N}\sum_{n=0}^{N-1} \hat{f}_n e^{i2\pi nk/N},\ k \in \{0, \ldots, N-1\}, \tag{IDFT}$$

Let us give a sample of Python code for these transforms.

```python
# Example 2
import numpy as np

def dft(f: np.ndarray) -> np.ndarray:
    """ Discrete Fourier Transform """
    N = len(f)
    k = np.arange(N)
    F = np.zeros(f.shape, dtype=complex)
    for n in range(N):
        F[n] = np.sum(f * np.exp(-1j * 2 * np.pi * n * k / N))
    return F

```

```python
def idft(F: np.ndarray) -> np.ndarray:
    """ Inverse Discrete Fourier Transform. """
    N = len(F)
    n = np.arange(N)
    f = np.zeros(F.shape, dtype=complex)
    for k in range(N):
        f[k] = np.sum(F * np.exp(1j * 2 * np.pi * n * k / N)) / N
    return f

# Test DFT and IDFT functions.
f = np.random.randn(100)
assert np.allclose(f, idft(dft(f)).real)
```

The discrete case is related to the Fourier Transform of an initial signal $f(t)$ which is zero for $t > T$ as follows [6]:

1. For window size T and number of sampling points N define the sample spacing $T_s = \frac{T}{N}$.

2. Then define the sample points $t_k = kT_s$ for $k \in \{0, \dots, N-1\}$ and evaluations $\hat{f}_k = f(t_k)$.

3. Associated with this we define the frequency sampling points $\omega_n = \frac{2\pi n}{T}$ where the number $\frac{2\pi}{T}$ is the fundamental frequency. Now we consider the problem of approximating the Fourier Transform of f at the points $\omega_n = \frac{2\pi n}{T}$. The exact answer is

$$\mathcal{F}\{f\}(\omega_n) = \int_{-\infty}^{\infty} f(t) e^{-i\omega_n t} dt, \; n \in \{0, \dots, N-1\}.$$

4. For $f(t) = 0$ for $t \notin [0, T]$

$$\mathcal{F}\{f\}(\omega_n) = \int_{0}^{T} f(t) e^{-i\omega_n t} dt, \; n \in \{0, \dots, N-1\}.$$

5. Let us approximate this integral by a left-endpoint Riemann sum approximation using the points t_k defined above:

$$\mathcal{F}\{f\}(\omega_n) \approx T_s \sum_{k=0}^{N-1} f(t_k) e^{-i\omega_n t_k}, \; n \in \{0, \dots, N-1\}.$$

6. Substituting the definitions of ω_n, t_k and T_s in terms of T and N we have

$$\mathcal{F}\{f\}(2\pi n/T) \approx \frac{T}{N} \sum_{k=0}^{N-1} \hat{f}_k e^{-i(2\pi n/T)(kT/N)} \tag{7}$$

$$= \frac{T}{N} \sum_{k=0}^{N-1} \hat{f}_k e^{-i2\pi nk/N}, \; n \in \{0, \dots, N-1\}. \tag{8}$$

   One can see that it is equivalent to (DFT) up to a normalisation constant.

Now we can use the result of the Theorem 1 together with (7) to write the equivalent of (6) in the discrete case.

Proposition 1. *Let* $\hat{\mathbf{f}} = \{f_0, \ldots, f_{N-1}\}$ *be a discretisation of a differentiable* $f(t)$ *on* $t \in [0, T]$ *such that* $f(t)$ *and* $f'(t)$ *are absolutely integrable, where* $\hat{f}_k = f(kT/N)$ *for* $k \in \{0, \ldots, N-1\}$. *Then we have*

$$\mathcal{F}\{f'\}(\omega_n) \approx i\omega_n F\{\hat{\mathbf{f}}\}(n), \tag{9}$$

where $\omega_n = 2\pi n/T$ *for* $n \in \{0, \ldots, N-1\}$.

Proof. The proof follows by replacing the right-hand side in (6) by its discrete variant and specifying the discrete frequencies ω_n as in (7). □

Proposition 2. *Let* $\hat{\mathbf{f}} = \{f_0, \ldots, f_{N-1}\}$ *be a discretisation of a differentiable* $f(t)$ *on* $t \in [0, T]$ *such that* $f(t)$ *and* $f'(t)$ *are absolutely integrable, where* $\hat{f}_k = f(kT/N)$. *Let* $\hat{\mathbf{f}}' = \{\hat{f}'_0, \ldots, \hat{f}'_{N-1}\}$ *be a discretisation of the derivative* $f'(t)$, *where* $\hat{f}'_k = f'(kT/N)$. *Then we can approximate* $\hat{\mathbf{f}}'$ *by the direct and inverse Fourier Transform on* $\hat{\mathbf{f}}$ *as*

$$\hat{\mathbf{f}}' \approx F^{-1}\{i\boldsymbol{\omega} F\{\hat{\mathbf{f}}\}\}, \tag{10}$$

where $\boldsymbol{\omega} = \{\omega_0, \ldots, \omega_{N-1}\}$, $\omega_n = 2\pi n/T$.

Proof. The proof follows by applying the continuous inverse Fourier Transform on the left and its discrete variand on the right in (9). □

Remark 2.1. *Practical implementation requires correcting the value of* T *for a given* N. *Consider the range* $[0, T]$ *for* $T > 0$. *Then the last point where the function* f *is evaluated is* $\frac{(N-1)T}{N}$, *which means that we do not include* $f(T)$. *To fix this, it is enough to change* T *to* $T = \frac{NT}{N-1}$.

Remark 2.2. *In Proposition 2 we considered* N *different frequencies* $\boldsymbol{\omega} = \{\omega_0, \ldots, \omega_{N-1}\}$ *to use for differentiation. This comes from naively replacing the continuous Fourier Transform with its discrete variant in (6). In practice, however, all frequencies* ω_n *for which* $n > N/2$ *for* N *even and* $n > (N-1)/2$ *for* N *odd do not exist, since* $\omega_{N/2}$ *(or* $\omega_{(N-1)/2}$*) is the highest frequency we can detect (Nyquist frequency). This comes from the Nyquist-Shannon sampling theorem.*

Let us give the Python code for the derivative calculation. We choose a Gaussian function as a test function for the derivative calculation, since it satisfies the conditions under which the Theorem 1 can be applied, i.e. it is an absolutely integrable function.

```python
# Example 3
import numpy as np
from numpy.fft import fft, ifft


def f(x):
    """ Gaussian function centered at 5. """
    return np.exp(-np.square(x - 5.))

def df(x):
    """ Derivative of the Gaussian function centered at 5. """
    return - 2. * (x - 5.) * np.exp(- np.square(x - 5.))

T = 10
N = 100

# Discretisation time points (including T).
t = np.linspace(0, T, N)
f_hat, df_hat = f(t), df(t)
```

```
20
21 # Correction of T to include f(T).
22 T = (N * T) / (N - 1)
23
24 # Discretisation frequency points.
25 n = np.arange(N)
26 w = 2 * np.pi * n / T
27
28 # Make all frequencies above Nyquist to be negative.
29 w -= w[N // 2] if N % 2 == 0 else w[N // 2 + 1]
30 w = np.fft.fftshift(w)
31
32 # Apply the Differentiation theorem.
33 df_approx = ifft(1j * w * fft(f_hat)).real
34
35 # Test the correctnes of differentiation.
36 assert np.allclose(df_approx, df_hat)
```

Let us visualise the obtained result from this example.

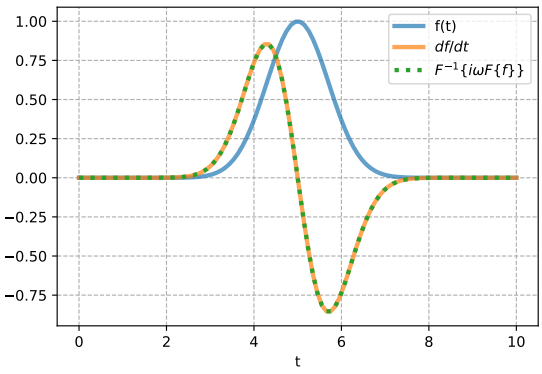


Figure 2: Comparison of the approximated derivatives using (10) and the exact derivatives.

Now we can use this tool to compute the Helmholtz operator on a sequence of discretised points $\hat{\mathbf{f}}$ that satisfy the conditions of Theorem 1. Example 3 was implemented in the class `FourierGradient` of the library `maxb`, which supports multidimensional derivatives. Also, the definition of the domain has been generalised in the class `Domain` to any range $[a, b]$ and for many dimensions. Let us give an example of Python code for the Helmholtz operator discretisation.

```
1  # Example 4
2  import numpy as np
3  from maxb import xde
4
5
6  def f(x):
7      """ Gaussian function centered at 5. """
8      return np.exp(-np.square(x - 5.))
9
10 def d2f(x):
11     """ Second derivative of the Gaussian function centered at 5. """
12     return 2. * (2. * np.square(5. - x) - 1.) * np.exp(- np.square(5. - x))
13
14 def helm(x, k=1.0):
```

```
    return d2f(x) + k * f(x)


# Domain initialization
domain = xde.Domain(128, [0, 10], dtype='float64')

# Fourier gradient initialization
grad = xde.grad.FourierGradient(domain, fft_axes=(-1,))

# Grid points
t = domain.grid[0]

k = 1.0  # wavenumber

# Helmholtz operator evaluation
helm_hat = helm(t, k=k)

# Helmholtz operator approximation
helm_approx = grad(grad(f(t))).real + k * f(t)

# Test the correctnes of approximation.
assert np.allclose(helm_approx, helm_hat)
```

Remark 2.3. *In this section we have tried to explain the differentiation approach using the Fourier Transform. The theoretical justification is given in the Proposition 2. Note that this is not a mathematical theorem with a complete proof, but rather an intuitive sketch to explain the Python implementation. The reason for this is that although the differentiation theorem is well studied and is presented in many books on Fourier Transforms, its adaptation to the discrete case is hard to find in the literature.*

2.2 Perfectly Matched Layer

In this section we explain the Perfectly Matched Layer or PML and give the code examples. The main source for this section is [7].

PML is the approach to truncate the computational domain in numerical methods for simulating open boundary problems. In other words, we want to simulate the absorbing boundary condition and avoid the direct use of the Sommerfeld radiation condition at infinity (2). It was introduced by Berenger in [1] where the idea was to replace an absorbing boundary condition by an absorbing boundary layer as shown in the Figure 3.
The problem with this approach is that waves are reflected at a transition between materials, which is also the case for the absorbing material. Instead of reflections from the grid boundary, there are reflections from the absorber boundary. However, Berenger showed that a special absorbing medium can be constructed so that waves are not reflected at the interface.

Let us start the explanation by considering the solution $u(\mathbf{x})$ of the one-dimensional Helmholtz equation (1) in infinite space. We will concentrate on truncating the problem in the positive direction (the negative direction will follow the same technique). The truncation can be explained in the following three steps:

1. *Analytic continuation*: In infinite space, the solutions and equations are analytically extended to a complex $\mathbf{x} \in \mathbb{C}^n$, which transforms the oscillating waves into exponentially decaying waves without reflections outside the region of interest.

2. *Coordinate transformation*: In infinite space, perform a coordinate transformation to express the complex $\mathbf{x} \in \mathbb{C}^n$ as a function of a real coordinate.

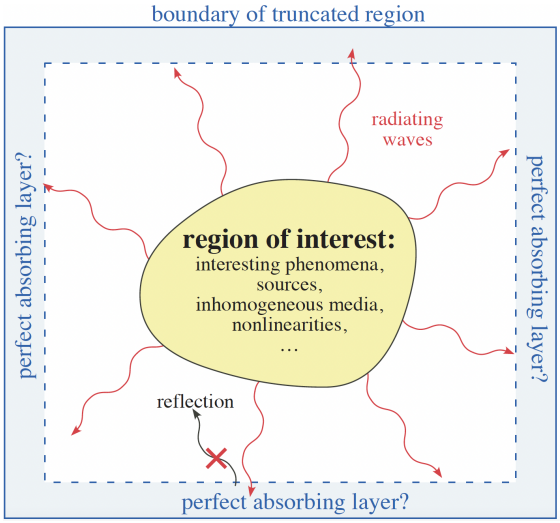


Figure 3: Absorbing layer absorbs outgoing waves without reflections from the edge of the absorber.

3. *Domain truncation*: Truncate the domain in this new real coordinate within the region where the solution decays. Since the solution decays there, as long as we truncate it after a long enough distance, it does not matter which boundary condition is used.

Under certain assumptions the analytic solution in infinite space is the superposition of the plane waves e^{ikx} for $x \in \mathbb{R}$, $k > 0$. Since it is analytic, we can continue by evaluating the solution at $x \in \mathbb{C}$. In this case, if the imaginary part grows linearly, then the solution will decay exponentially, as shown in Figure 4. The reason is that $e^{ik\left(\mathrm{Re}\{x\}+i\mathrm{Im}\{x\}\right)} = e^{ik\mathrm{Re}\{x\}}e^{-k\mathrm{Im}\{x\}}$ is exponentially decaying for $k > 0$ as $\mathrm{Im}\{x\}$ increases. For example, in Figure 4 one can see that for $\mathrm{Re}\{x\} > 0$ the imaginary part increases linearly, leading to an exponential decay of the solution. Note that the solution for $\mathrm{Re}\{x\} < 0$ is unchanged, which means that PML acts not only like an absorbing material, but also like a non-reflecting absorbing material.

Let us also provide a small Python example to reproduce the results in Figure 4.

```python
# Example 5
import numpy as np

def sigma(x, thresh=0., slp=0.2):
    s = np.zeros_like(x)
    mask = x > thresh
    s[mask] = slp * (x[mask] - thresh)
    return s

x = np.linspace(-3 * np.pi, 3 * np.pi, 100)

# Plane wave for real x.
x = x + 1j * 0
u = np.exp(1j * 2 * x)

# Plane wave for complex x with an absorbing layer for Re[x] > 0.
x_sgm = x + 1j * sigma(x)
u_sgm = np.exp(1j * 2 * x_sgm)
```

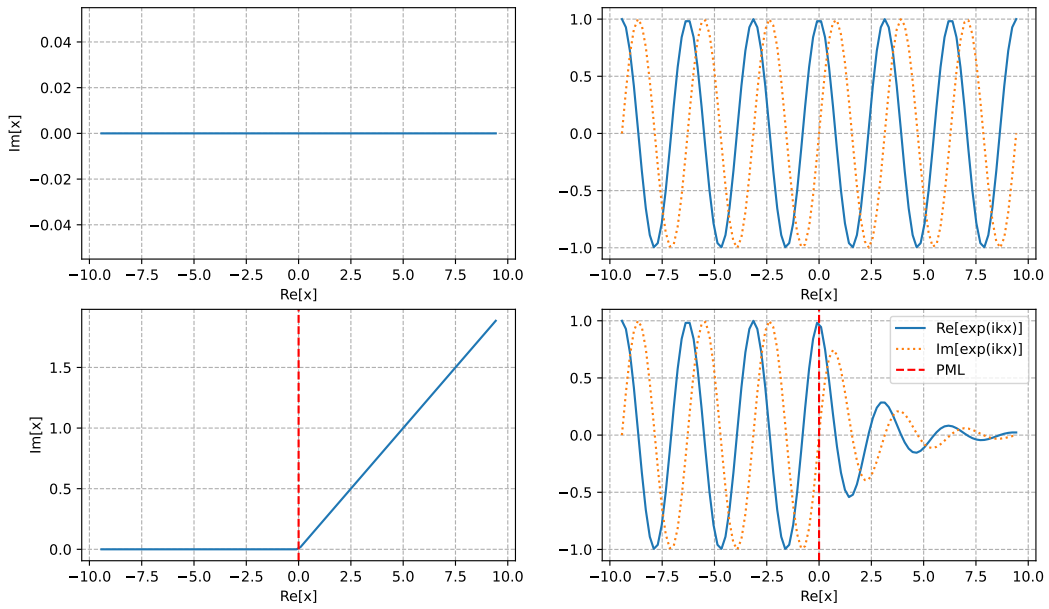


Figure 4: *Top*: real and imaginary part of original oscillating solution e^{ikx} (right) corresponds to x along the real axis in the complex plane (left). *Bottom*: We can instead evaluate this analytic function along a deformed contour in the complex plane: here (left) we deform it to increase along the imaginary axis for $\mathrm{Re}\{x\} > 0$ The e^{ikx} solution (right) is unchanged for Re $x < 0$, but is exponentially decaying for $\mathrm{Re}\{x\} > 0$ where the contour is deformed, corresponding to an "absorbing" region.

To provide the coordinate transformation let us denote the complex $x \in \mathbb{R}$ extension as $\tilde{x} \in \mathbb{C}$ and reserve letter x for the real part of $\tilde{x}$. Thus, we have that

$$\tilde{x}(x) = x + if(x), \tag{11}$$

where $f : \mathbb{R} \to \mathbb{R}$ is how we transform the imaginary part. The direct use of a complex coordinate $\tilde{x}$ is inconvenient, therefore we make a change of variables to return to the real coordinates. To get back, it is sufficient to change the derivatives as $\partial\tilde{x} = (1 + i\frac{df}{dx})\partial x$. Following [7], it is convenient to denote $\frac{df}{dx} = \frac{\sigma_x(x)}{\omega}$, for some function $\sigma_x : \mathbb{R}^d \to \mathbb{R}$, to obtain a frequency independent absorption. In terms of σ_x, the entire process of the PML can be summed up conceptually by a single transformation of our original differential equation:

$$\frac{\partial}{\partial\tilde{x}} \to \frac{1}{1 + i\frac{\sigma_x(x)}{\omega}} \frac{\partial}{\partial x}, \tag{12}$$

After performing the PML transformation (12) of the original problem, the solutions are unchanged in the region of interest and exponentially decaying in the outer regions. This implies that we can truncate the computational domain at any sufficiently large x.

2.3 Preconditioned GMRES

In this section we present the GMRES algorithm with preconditioner. Despite the fact that it is easily found in [15], we include it here without detailed proof to make this report self-contained

and to explicitly show where the preconditioner is used. Let us consider the system (5) for fixed $\hat{\mathbf{c}}$ to denote $A = A(\hat{\mathbf{c}})$. Then the right preconditioned GMRES algorithm is based on solving

$$AM(\mathbf{y}) = \hat{\boldsymbol{\rho}}, \quad \hat{\mathbf{u}} = M(\mathbf{y}), \tag{13}$$

where $M : \mathbb{C}^n \to \mathbb{C}^n$ is the preconditioner operator, i.e. it approximates A^{-1} somehow. The new variable $\mathbf{y}$ never needs to be explicitly called, which can be observed in the Algorithm 1. The preconditioner is applied to the vectors of the Krylov basis V_m in lines 3 and 11 of the algorithm.

Remark 2.4. *The operator $M : \mathbb{C}^n \to \mathbb{C}^n$ in (13) is used in the generalised form, assuming that it can be linear (then $M(\mathbf{y}) = M\mathbf{y}$ for $M \in \mathbb{C}^{n\times n}$) or nonlinear, which is not a classical way presented in [15]. The first reason is that in this work we use a nonlinear neural network as preconditioner, which does not change the Algorithm 1 except for lines 3 and 11 where we write a general operator application instead of matrix multiplication. The second reason is to avoid confusion with Flexible GMRES, which occurs when one tries to analyse the reasons for using Flexible GMRES when M is fixed but nonlinear. In Flexible GMRES, the behaviour of the preconditioner changes between iterations and the algorithm requires an additional matrix Z_m to store the modified basis of the search space. Since we do not need Z_m because M is fixed (but possibly nonlinear), we do not use Flexible GMRES in our context.*

Algorithm 1 GMRES with Right Preconditioning [15, Algorithm 9.5]

1: Compute $\mathbf{r}_0 = \hat{\boldsymbol{\rho}} - A\mathbf{u}_0$, $\beta = \|\mathbf{r}_0\|^2$, and $\mathbf{v}_1 = \mathbf{r}_0/\beta$
2: **for** $j = 1, \ldots, m$ **do**
3: Compute $\mathbf{w} := AM(\mathbf{v}_j)$
4: **for** $i = 1, \ldots, j$ **do**
5: $h_{i,j} := \langle \mathbf{w}, \mathbf{v}_i \rangle$
6: $\mathbf{w} := \mathbf{w} - h_{i,j}\mathbf{v}_i$
7: **end for**
8: Compute $h_{j+1,j} = \|\mathbf{w}\|_2$ and $\mathbf{v}_{j+1} = \mathbf{w}/h_{j+1,j}$
9: Define $V_m := [\mathbf{v}_1, \ldots, \mathbf{v}_m]$, $\bar{H}_m = \{h_{i,j}\}_{1\le i\le j+1; 1\le j\le m}$
10: **end for**
11: Compute $\mathbf{q}_m = \text{argmin}_{\mathbf{q}} \|\beta\mathbf{e}_1 - \bar{H}\mathbf{q}\|_2$, and $\mathbf{u}_m = \mathbf{u}_0 + M(V_m\mathbf{q}_m)$
12: **if** stopping test is satisfied **then** Stop
13: **else** Set $\mathbf{u}_0 = \mathbf{u}_m$ and GoTo 1
14: **end if**

The Algorithm 1 is implemented in `axb` which supports operators A, M located on CPU or GPU. Let us also give a Python implementation of the algorithm which is useful for learning.

```
# Example 6
import numpy as np
from numpy.linalg import norm, lstsq
from scipy.sparse.linalg import LinearOperator

# Linear system generation
n = 10
A = np.random.randn(n, n)
x = np.random.randn(n, 1)
b = A @ x

```

```
# Identity preconditioner
M = LinearOperator(shape=(n, n), dtype=float, matvec=lambda x: x)

# GMRES with Right Preconditioning
K_dim = 10  # Dimension of the Krylov subspace (m in Algorithm 1)
restarts = 1

b_norm = norm(b)
V = np.zeros(shape=(n, K_dim))
Z = np.zeros(shape=(n, K_dim))
H = np.zeros(shape=(K_dim + 1, K_dim))
e = np.zeros(shape=(K_dim + 1, 1))

resids = []
xk = np.zeros_like(x)

for _ in range(restarts):
    r = _r = b - A @ xk
    _r_norm = r_norm = norm(r)

    resids.append(r_norm / b_norm)

    V[:, [0]] = r / r_norm
    e[0] = r_norm

    for j in range(K_dim):
        Z[:, [j]] = M(V[:, [j]])
        w = A @ Z[:, [j]]

        # Arnoldi iterations
        for i in range(j + 1):
            H[i, j] = w.T.dot(V[:, [i]])
            w = w - H[i, j] * V[:, [i]]

        H[j + 1, j] = norm(w)
        if j + 1 < K_dim:
            V[:, [j + 1]] = w / H[j + 1, j]

        y = lstsq(H[:j + 2, :j + 1], e[:j + 2], rcond=None)[0]
        _xk = xk + M(V[:, :j + 1]) @ y
        _r = b - A @ _xk
        _r_norm = norm(_r)
        resids.append(_r_norm / b_norm)

    y = lstsq(H[:j + 2, :j + 1], e[:j + 2], rcond=None)[0]
    xk = xk + M(V) @ y

# Test the solution
assert np.allclose(xk, x)
```

3 Research path

In this section we describe the evolution of the current research. The starting point was a paper written by Stanziola et. al. [17], where the authors train the U-net neural network to solve the two-dimensional Helmholtz problem. The idea of the authors is to obtain a neural network that can be used in a fixed-point iteration scheme

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \text{U-net}_\theta(\mathbf{u}_k, \mathbf{r}_k), \tag{14}$$

where $\mathbf{u}_k \in \mathbb{C}^n$ is the kth approximation of the discretised solution, $\mathbf{r}_k = A(\mathbf{c})\mathbf{u}_k - \boldsymbol{\rho}$ is the discretised residual for some $\mathbf{c} \in \mathbb{R}^n_+$, θ are the parameters of the neural network, and $\boldsymbol{\rho} \in \mathbb{R}^n$. The authors show that it is possible for $\mathbf{c} \in [1, 2]^n$, $n = 96^2$. They also show that the neural network trained on the grid 96 by 96 can be successively applied to the larger domain, which is an empirical proof of the model generalisation.

Considering that, the first question we ask is whether we can use the U-net as a preconditioner. The results obtained in [18] show that this is possible, which is also confirmed in Section 6. Both variants give approximately the same result, which are $M(\mathbf{v}_k) := \text{U-net}_\theta(\mathbf{u}_k, \mathbf{v}_k)$ and $M(\mathbf{v}_k) := \text{U-net}_\theta(\mathbf{0}, \mathbf{v}_k)$. This preconditioner significantly reduces the number of GMRES iterations, but also increases the computational cost per iteration. Despite the fact that the evaluation of M is highly parallelizable, which reduces the computational time, the comparison of the absolute power consumption remains questionable.

The second question we ask is whether we can simplify both the learning process of the U-net and its architecture. The training process of the U-net in [17] is highly unstable due to the differentiation of multiple fixed-point iterations and requires some "magic" constants that are not explained in the paper. The U-net architecture also contains the hidden state, which theoretically helps to capture the dependence between iterations, but makes the architecture more difficult to understand and implement. To solve these issues we tried to simplify all parts and to approximate A^{-1} directly by

$$\min_\theta \|X - A\text{U-net}_\theta(X)\|_2^2, \tag{15}$$

where $X \in \mathbb{C}^{N \times n}$, N is the size of the dataset. The U-net discussed in Section 4.4 is not the only type of architecture that can be used here. In sections 4.2 and 4.3 we also considered DeepONet and FNO, which are the neural operators [8]. Although the idea (15) is simple and easy to implement, the question of the input distribution of X is not obvious because we do not know in advance the structure of $\mathbf{v}_j$ in step 3 of the Algorithm 1.

4 Unsupervised inverse operator learning

In this section we try to approximate the inverse of the linear operator $A(\mathbf{c}) \in \mathbb{C}^{n \times n}$ for $\mathbf{c} \in \mathbb{R}^n_+$ using a neural network. The aim is to find the architecture that is adaptive in terms of the domian size n and is generalisable in terms of the varying $\mathbf{c} \in \mathbb{R}^n_+$. Ideally, once trained on a fixed domain, the neural network could be applied to a larger domain, taking $\mathbf{c}$ as input.

We started our research with the physics-informed neural network (PINN) [13], which is designed to approximate the solution of any differential equation and is trained in an unsupervised way. The result is the trained neural network that satisfies a differential equation and can be evaluated at any point in the domain. Although the neural network has to be retrained every time the equation parameters or boundary conditions change, this was the first step that gave us useful insights that are presented in Section 4.1.

We continued our research with an extension of PINN called DeepONet [10], which aims to add equation parameters to the model input. However, this approach is still discretisation dependent, which means that we have to retrain the model when the domain changes. We give details about it in Section 4.2.

The next step was the neural operator [8]. The idea, as with DeepONet, is to use the equation parameters, boundary conditions and other variable objects as inputs to the neural network. However, the main difference is that the neural operator is a descretisation invariant model that computes a discretised solution of a differential equation. There are several well-

known architectures, such as the Fourier Neural Operator [9] and the U-net [14], which are discussed in the sections 4.3 and 4.4 respectively.

4.1 Physics-informed neural network

Let us consider the general differential equation

$$L(u) = f,$$

where L is a differential operator, then the neural network u_θ aims to approximate u such that the norm of the residual $\|L(u_\theta) - f\|$ is minimised with respect to model parameters θ. Thus, the neural network becomes physics-informed when the differential equation based loss is minimised. In our case we work with the Helmholtz operator $L = \nabla^2 + k^2$, where L can be considered as

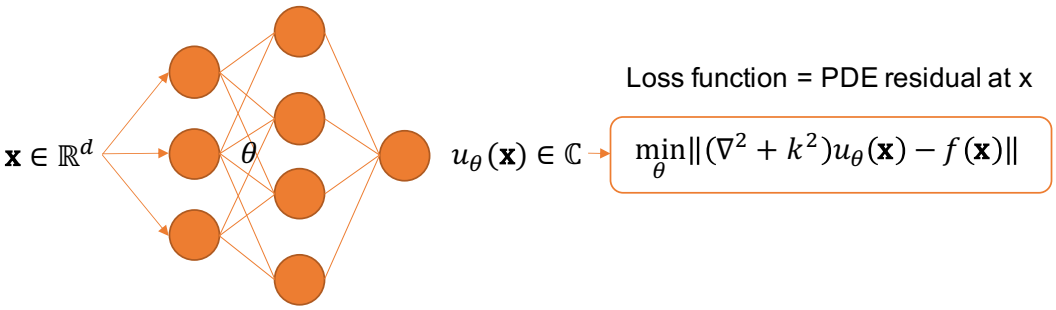


Figure 5: Physics-informed neural network concept applied to the Helmholtz operator.

a continuous or discretised operator. To compute the norm of a residual in the first case, we can use the automatic differentiation algorithm, which allows us to evaluate $L(u_\theta)(x)$ for any x. Another possibility is to evaluate u_θ on a fixed grid and obtain $\hat{\mathbf{u}}_\theta$, which is then used to approximate the Helmholtz operator by (10). Let us give a code example for the last possibility.

```
# Example 7
import maxb
import torch
from maxb import xde

device = "cuda:0"  # "cpu"

# Domain initialization
domain = xde.Domain(256, [-40, 40], device=device, backend=torch)
grid = domain.grid.reshape(domain.dims, -1).T

# Source initialization
source = xde.source.Gaussian(domain, axis=1)
src = source(grid).reshape(*domain.num_points)

# Fourier gradient with PML initialization
grad = xde.grad.FourierGradientModule(domain).to(device)
grad = xde.PMLModule(grad)

# Helmholtz operator initialization
helm = xde.operators.HelmholtzModule(grad)

# PINN initialization
```

```
net = maxb.nn.MMLP(in_features=domain.dims, out_features=1, hidden_features=64,
                   input_activation="sin", num_layers=2).to(device)

# Optimizer initialization
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)

# Training loop
for epoch in range(250):
    optimizer.zero_grad()
    # Discretised residual computation
    u = net(grid).reshape(*domain.num_points)
    r = helm(u) - src
    # Backpropagation
    loss = torch.norm(r)
    loss.backward()
    # Parameters update
    optimizer.step()
    # Plot solution
    if (epoch + 1) % 50 == 0 or epoch == 0:
        with torch.no_grad():
            u = net(grid).reshape(*domain.num_points).cpu()
        maxb.utils.plot_solution_1d(grid.T.cpu()[0], u)
        print(f"Epoch: {epoch+1:3d}/{250} - Loss: {loss:4f}")
```

Here we are using the PyTorch library to initialise and train the neural network. Note that we use the class `FourierGradientModule` instead of `FourierGradient` because we need to inherit from `torch.nn.Module` to make this class differentiable. The same logic applies to `PML` and `Helmholtz`. The neural network architecture shown in the Figure 6 and named `MMLP` in the code deserves special attention. This is a well-known Multi-Layer Perceptron (MLP)

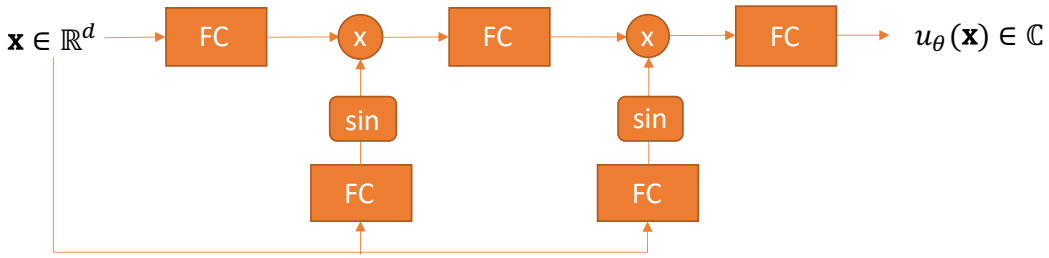


Figure 6: Physics-informed neural network architecture for the Helmholtz operator approximation.

architecture with component-by-component vector Multiplication between layers (MMLP). The multiplication is done with vectors obtained after a Fully Connected (FC) layer with a sin activation function. The periodic activation becomes crucial due to the periodic nature of the solution. One can change the activation in line 25 of the previous example and re-run the training to see the difference. The author took this architecture from an example on the Internet, but this site is no longer available, so there is no citation.

As mentioned above, this approach is not useful for building a preconditioner because of the retraining requirements when the equation parameters change, and because there is no way to send a vector to the input. However, here we have learned how to use a physics-based loss to train the neural network in an unsupervised way, which will be useful in further research.

4.2 DeepONet

In this section we discuss the next step in operator learning, which is the extension of physics-informed neural networks. The original idea of [10] where authors present DeepONet is to approximate a differential operator L with a neural network instead of approximating the solution u of a differential equation as in PINN. The concept of the DeepONet is presented in the Figure 7. Note that initially this is a supervised learning, since the evaluations of the operator L applied

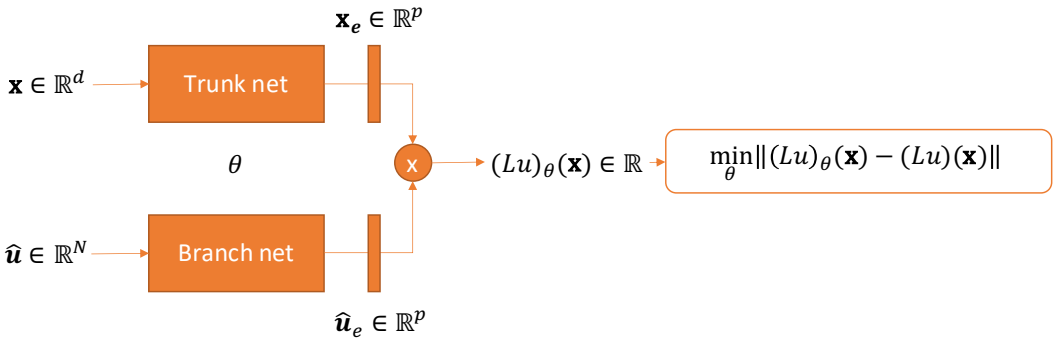


Figure 7: DeepONet concept, where $(Lu)_\theta$ is the neural network with trainable parameters θ that approximates (Lu), and p is the size of the empedding vectors. The notation of the Trunk net and the Branch net is taken from the original paper [10].

to the function u at the points x are needed for the loss calculation. Let us adapt this idea to our problem. Recall that we wish to approximate the inverse of the Helmholtz operator using a neural network to use it as a preconditioner. So we have $L = (\nabla^2 + k^2)^{-1}$ and the input to the Branch net is $\hat{\boldsymbol{\rho}}$.

We then move from supervised to unsupervised learning by applying the inverse of L^{-1} to the residual. Let us denote $(L\rho)_\theta(\mathbf{x}) = u_\theta(\mathbf{x})$. Then, we have that

$$\begin{aligned} L^{-1}\big((L\rho)_\theta(\mathbf{x}) - (L\rho)(\mathbf{x})\big) &= (L^{-1}u_\theta)(\mathbf{x}) - \rho(\mathbf{x}) \\ &= (\nabla^2 + k^2)u_\theta(\mathbf{x}) - \rho(\mathbf{x}) \end{aligned}$$

Also, since the input vectors for the preconditioner M are complex in line 3 of the Algorithm 1, we need to use $\hat{\boldsymbol{\rho}} \in \mathbb{C}^N$. The modification of the original idea is shown in the Figure 8. Before giving the code example let us discuss the limitations of the model. Let us provide the Observation 1, from where we can conclude that if we use DeepONet as a preconditioner for $p < N$ in the full GMRES iterations, then the vectors from the preonditioned Krylov basis will be linearly dependent, which is the first limitation.

Observation 1. *Let $X = \{\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(N-1)}\} \in \mathbb{R}^{d\times N}$ be a set of discretisation points in the domain and let $\hat{\boldsymbol{\rho}} \in \mathbb{C}^N$ be a fixed right hand side. Let $X_e = \{\mathbf{x}_e^{(0)}, \dots, \mathbf{x}_e^{(N-1)}\} \in \mathbb{R}^{p\times N}$ be a set of embeddings for X, let $p < N$, and let $\hat{\boldsymbol{\rho}}_e \in \mathbb{C}^p$ be an embedding for a fixed right-hand side $\hat{\boldsymbol{\rho}}$. Let $\hat{\mathbf{u}}_\theta = \{u_\theta^{(0)}, \dots, u_\theta^{(N-1)}\} \in \mathbb{C}^N$ be a set of evaluations of the neural network at X for fixed $\hat{\boldsymbol{\rho}}$. Then the dimension of the linear space to which $\hat{\mathbf{u}}_\theta$ belongs is at most p.*

Proof. By definition, $u_\theta^{(i)} = \langle \mathbf{x}_e^{(i)}, \hat{\boldsymbol{\rho}}_e \rangle$ which can also be written in a matrix form $\hat{\mathbf{u}}_\theta = X_e^\top \boldsymbol{\rho}_e$. Then $\hat{\mathbf{u}}_\theta \in \text{range}(X_e^\top)$. The proof follows. □

The second limitation is that if the number of the discretisation points N changes, then we need to retrain the model. The third limitation is that if the parameters of the operator changes (k in

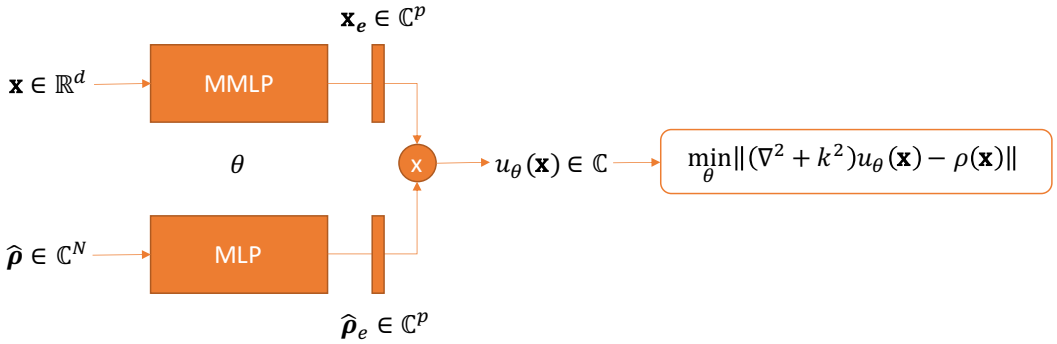


Figure 8: DeepONet modified concept. The Trunk net is proposed to be replaced by MMLP as for PINN and the Branch net is MLP for simplicity.

the case of the Helmholtz operator), then we also need to retrain the model. Let us give a code example to train the model.

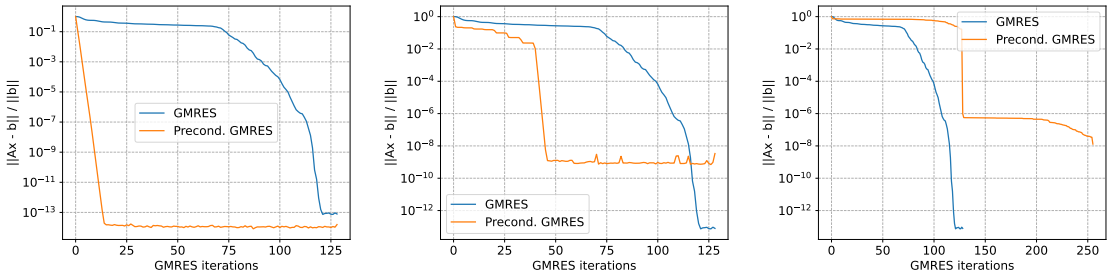
```python
# Example 8
import maxb
import torch
from maxb import xde

device = "cuda:0"  # "cpu"
# Domain initialization
num_points = 128
domain = xde.Domain(num_points, [-40, 40], device=device, backend=torch)
grid = domain.grid.reshape(domain.dims, -1).T
# Fourier gradient with PML initialization
grad = xde.grad.FourierGradientModule(domain).to(device)
grad = xde.PMLModule(grad)
# Helmholtz operator initialization
helm = xde.operators.HelmholtzModule(grad)
# DeepONet initialization
net = maxb.nn.DeepONet(domain, hidden_features=2*num_points, num_layers=4).to(device)
# Optimizer initialization
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
# Training loop
for epoch in range(5000):
    optimizer.zero_grad()
    # Random source generation
    src = torch.randn(32, domain.total_num_points, dtype=torch.cfloat).to(device)
    # Discretised residual computation
    u = net(grid, src).T
    r = helm(u) - src
    # Backpropagation
    loss = torch.norm(r, dim=-1).mean()
    loss.backward()
    # Parameters update
    optimizer.step()
    # Plot solution
    if (epoch + 1) % 500 == 0 or epoch == 0:
        source = xde.source.Gaussian(domain, loc=torch.rand(1) * 70. - 35., axis=1)
        src = source(grid).unsqueeze(0).to(device) + 1j * 0.
        with torch.no_grad():
            u = net(grid, src).reshape(*domain.num_points).cpu()
```

```
        maxb.utils.plot_solution_1d(grid.T.cpu()[0], u)
        print(f"Epoch: {epoch+1:3d}/{2000} - Loss: {loss:4f}")
```

Then we can use this model as a preconditioner for fixed k. Let us provide the code example.

```
from axb import gmres

if device != "cpu":
    import cupy as xp
    from cupyx.scipy.sparse.linalg import LinearOperator
else:
    import numpy as xp
    from scipy.sparse.linalg import LinearOperator

# GMRES
A = LinearOperator(shape=(num_points, num_points), dtype='complex128',
                   matvec=helm.make_matvec(device=device, xp=xp))
source = xde.source.Gaussian(domain)
b = source(domain.grid)

resids = []
def callback(r):
    global resids
    resids.append(r.get())
u, info = gmres(A, b, callback=callback, callback_type='pr_norm_iter')

# Preconditioned GMRES by DeepONet
M = LinearOperator(shape=(num_points, num_points), dtype='complex128',
                   matvec=net.make_matvec(grid, xp=xp))
resids_prec = []
def callback(r):
    global resids_prec
    resids_prec.append(r.get())
u, info = gmres(A, b, M=M, callback=callback, callback_type='pr_norm_iter')
```

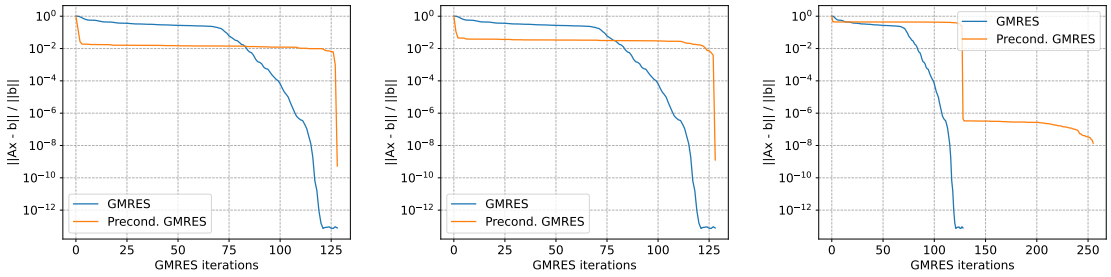
Let us visualize the norm of the residuals for GMRES with and without the preconditioner in the Figure 9. One can see the fast convergence with the DeepONet preconditioner compared to



(a) $p = 2N$ (b) $p = N$ (c) $p = N/2$

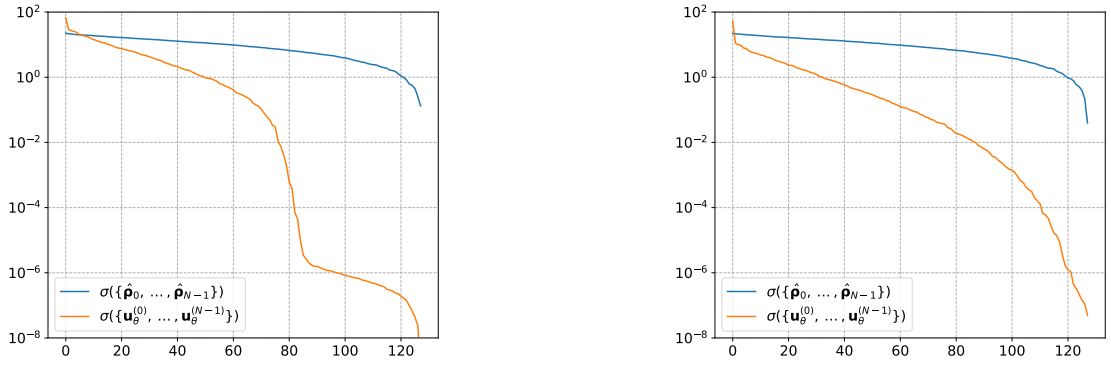
Figure 9: Non-preconditioned and DeepONet preconditioned full GMRES residuals norm during iterations for $p \in \{2N, N, N/2\}$, $N = 128$, and 4 layers in the trunk net.

the non-preconditioned full GMRES. However, this result is only obtained for $p = 2N$, where $N = 128$ is the number of discretisation points. If we try to reduce p, we observe the difficulties



(a) $p = 2N$ (b) $p = N$ (c) $p = N/2$

Figure 10: Non-preconditioned and DeepONet preconditioned full GMRES residuals norm during iterations for $p \in \{2N, N, N/2\}$, $N = 128$, and 2 layers in the trunk net.



(a) 2 hiddel layers (b) 4 hidden layers

Figure 11: Singular values distribution of $\{\mathbf{u}_\theta^{(0)}, \ldots, \mathbf{u}_\theta^{(N-1)}\}$ for fixed $\{\hat{\boldsymbol{\rho}}_0, \ldots, \hat{\boldsymbol{\rho}}_{N-1}\}$ and variable hidden layers in the MMLP.

in convergence (Figures 10c, 10b), that can be partially explained by the Observation 1. Another problem arises when we reduce the number of layers for the trunk network in DeepONet (Figure 10). Empirical observations of the singular values of a basis of a linear space processed through the DeepONet have a distribution that makes the vectors almost linearly dependent in 32-bit arithmetic (Figure 11). The explanation of this observations is not given in this report. The code to reproduce this result is given below.

```
l = 4
p = num_points
net = maxb.nn.DeepONet(domain, hidden_features=p, num_layers=l).to(device)

U, R = [], []
for _ in range(domain.total_num_points):
    R.append(torch.randn(domain.total_num_points, dtype=torch.cfloat, device=device))
    with torch.no_grad():
        U.append(net(domain.flat_grid, R[-1].unsqueeze(0)).ravel())

U = torch.vstack(U).cpu()
R = torch.vstack(R).cpu()
```

4.3 Fourier Neural Operator

In this section, based on [8, 9], we consider the concept of neural operators for learning differential operators. It represents a mapping between infinite dimensional function spaces, where in practice the mapping is considered between discrete spaces. The architecture of the neural operators, shown in Figure 12, allows them to be discretisation invariant models. This means that once trained on a fixed "grid", they can be applied to a larger or smaller grid. The well-known discretisation invariant neural network is a convolutional type, where the term "grid" means the pixels of an image. The arcitecture of neural operators consists of L layers where each

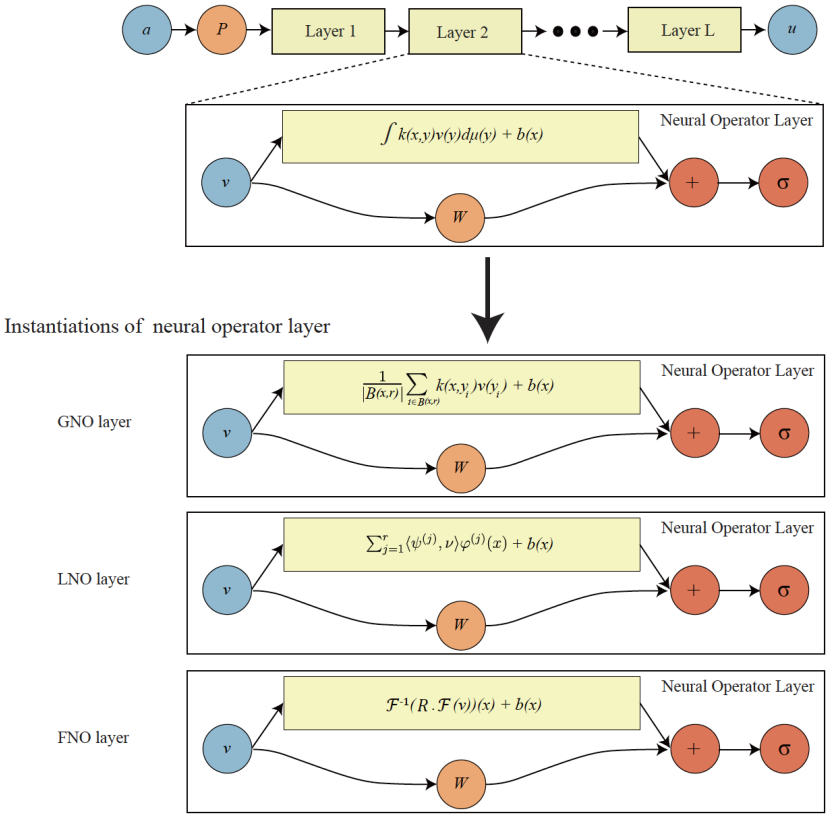


Figure 12: Neural operator architecture schematic. The input function a is passed to a pointwise lifting operator P that is followed by L layers of integral operators and pointwise non-linearity operations σ. Three instantiation of neural operator layers, GNO, LNO, and FNO are provided [8].

of them consists of convolutional part (yellow rectangular), bias (orange circle with lette W), and nonlinear activation function (red circle with letter σ). The most important part is a convolution, which provides discretisation invariance. This operation can be performed in different ways. For example, the well-known theorem states that multiplication in the frequency domain is equivalent to convolution in the spatial domain [11]. This fact is used in the last variant, which is presented in the Figure 12 and will be considered further in this section.

The Fourier Neural Operator (FNO) is a neural network architecture that combines the power of neural networks with the mathematical machinery of Fourier analysis. It is designed to address

complex mathematical modelling and simulation challenges in science and engineering.

FNO uses the representational learning capabilities of neural networks and the spectral analysis properties of Fourier transforms. Below is a description of the way it works:

1. **Representation Learning with Neural Networks:** FNO begins by encoding the underlying problem using a neural network. This encoding involves the extraction of meaningful features and patterns from the data. The neural network learns to capture complex relationships present in the data, which may represent spatio-temporal behaviour or other complex phenomena.

2. **Utilization of Fourier Transform:** Once the neural network has captured these patterns, FNO applies the Fourier Transform - a mathematical technique that decomposes a function into a set of sinusoidal basis functions. This transformation shifts the perspective from the time-space domain to the frequency domain. In this way, FNO attempts to find simpler, more structured representations of the learned patterns.

3. **Learning Interaction Rules:** In the frequency domain, FNO aims to identify the rules of interaction between the highest frequencies or modes. These interaction rules can be thought of as the underlying governing equations that dictate the evolution of the system. The neural network learns how these frequencies interact, effectively learning the mathematical operators that describe the dynamics of the system.

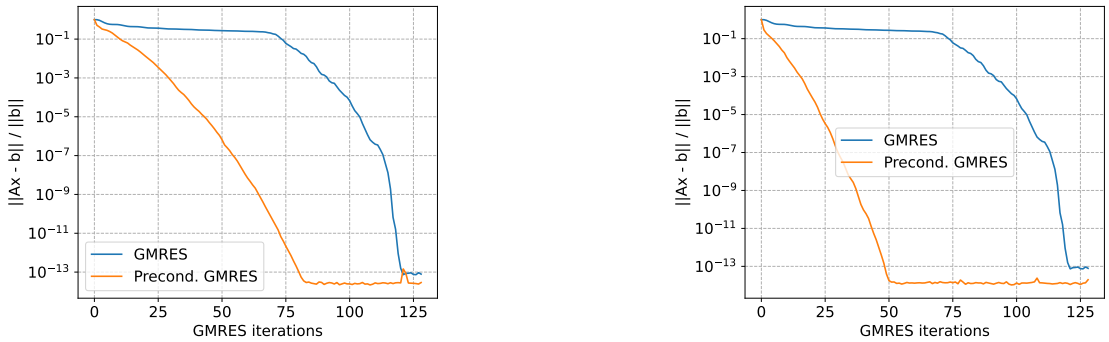
4. **Inverse Fourier Transform:** Once the interaction rules are learned in the frequency domain, FNO performs an inverse Fourier transform. This step translates the simplified, structured representations back into the original time-space domain, resulting in a predicted solution that captures the complex behaviour of the system.

In summary, the Fourier Neural Operator bridges the gap between neural networks and mathematical analysis techniques. It integrates the learning capabilities of neural networks with the spectral insights of Fourier analysis to provide accurate and computationally efficient solutions to complex mathematical modelling problems encountered in scientific research and engineering applications. As it was mentioned above, the key idea is that we truncate the high frequencies and apply the trainable linear operator to them before returning to the spatial domain by means of the inverse Fourier transform. Let us call this operation as a spectral convolution and give the code example for the one-dimensional case.

```python
import torch

class SpectralConv1d(torch.nn.Module):

    def __init__(self, in_channels, out_channels, num_modes):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.num_modes = num_modes
        # Kernel initialisation
        kernel = torch.rand(in_channels, out_channels, num_modes, 2, dtype=torch.float)
        kernel /= in_channels * out_channels
        self.kernel = torch.nn.Parameter(kernel)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """ x input shape is (batch_size, channels, num_points) """
        batch_size, _, n = x.shape
        # Direct Fourier Transform.
```

```
        x_fft = torch.fft.rfft(x)
        f = torch.zeros((batch_size, self.out_channels, x_fft.shape[-1]),
                        dtype=x_fft.dtype, device=x_fft.device)
        # Truncation and trainable linear operator application.
        f[..., :self.num_modes] = torch.einsum(
            "bix,iox->box", x_fft[..., :self.num_modes],
            torch.view_as_complex(self.kernel))
        # Inverse Fourier Transform.
        return torch.fft.irfft(f, n=n)
```

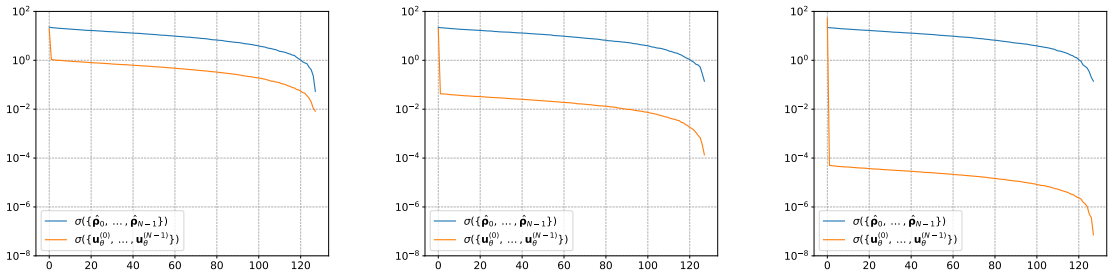
Note that it can be applied to the input tensor `x` with any last dimension, because only the fixed number of frequencies `num_modes` is taken for processing. The complete code example is the same as for DeepONet (Example #8) with the difference in a few lines, which can be found in `example_09.ipynb`.



(a) 16 modes (b) 32 modes

Figure 13: Non-preconditioned and FNO preconditioned full GMRES residuals norm during iterations for different modes, $N = 128$, and 1 layer.

On the Figure 13 one can see the comparison of FNO preconditioners for different number of Fourier modes. Note, that for $N = 128$ the maximal number of Fourier modes to consider is $64+1$ (including the Nyquist frequency) due to the symmetry of the spectrum for the real-valued signal, which is our case because we represent complex numbers in the real domain ($\mathbb{C} \sim \mathbb{R}^2$) in the FNO implementation.



(a) 1 hiddel layer (b) 4 hidden layers (c) 8 hidden layers

Figure 14: Singular values distribution of $\{\mathbf{u}_\theta^{(0)}, \ldots, \mathbf{u}_\theta^{(N-1)}\}$ for fixed $\{\hat{\boldsymbol{\rho}}_0, \ldots, \hat{\boldsymbol{\rho}}_{N-1}\}$ and variable hidden layers in the FNO.

Then, we can compare the disctributions of the singular values of $\{\mathbf{u}_\theta^{(0)}, \dots, \mathbf{u}_\theta^{(N-1)}\}$ for fixed $\{\hat{\boldsymbol{\rho}}_0, \dots, \hat{\boldsymbol{\rho}}_{N-1}\}$. In the Figure 14 one can observe that with the increase of the number of layers we decrease the singular values which can lead to the linear dependance of the preconditioned Krylov basis.

Untill this moment we trained neural networks with fixed $\hat{\mathbf{c}} = 1$, however we require the model that works with variable $\hat{\mathbf{c}}$ without retraining. Let us add it as an additional channel to the input tensor as it is presented in the code example below, where $\hat{\mathbf{c}}$ is uniformly distributed on the range $[1, 2]$.

```python
# Example 10
import maxb
import torch
from maxb import xde

device = "cuda:0"  # "cpu"
# Domain initialization
num_points = 128
domain = xde.Domain(num_points, [-40, 40], device=device, backend=torch)
grid = domain.grid.reshape(domain.dims, -1).T
# Fourier gradient with PML initialization
grad = xde.grad.FourierGradientModule(domain).to(device)
grad = xde.PMLModule(grad)
# Helmholtz operator initialization
helm = xde.operators.HelmholtzModule(grad)
# DeepONet initialization
net = maxb.nn.FourierNet1d(3, 2, 16, 2, 16).to(device)
# Optimizer initialization
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)

# Training loop
for epoch in range(5000):
    optimizer.zero_grad()
    # Random source generation
    src = torch.randn(32, domain.total_num_points, dtype=torch.cfloat).to(device)
    sos = torch.rand(32, 1, domain.total_num_points, dtype=torch.float).to(device) + 1.
    # Discretised residual computation
    u = net.forward_cpx(src, sos)
    r = helm(u) - src
    # Backpropagation
    loss = torch.norm(r, dim=-1).mean()
    loss.backward()
    # Parameters update
    optimizer.step()
    # Plot solution
    if (epoch + 1) % 500 == 0 or epoch == 0:
        source = xde.source.Gaussian(domain, loc=torch.rand(1) * 70. - 35., axis=1)
        src = source(grid).unsqueeze(0).to(device) + 1j * 0.
        sos = torch.rand(1, 1, domain.total_num_points, dtype=torch.float).to(device) + 1.
        with torch.no_grad():
            u = net.forward_cpx(src, sos).reshape(*domain.num_points).cpu()
        maxb.utils.plot_solution_1d(grid.T.cpu()[0], u)
        print(f"Epoch: {epoch+1:3d}/{5000} - Loss: {loss:4f}")
```

Then, to use the trained model as a preconditioner, we can run the following code.

```python
from axb import gmres

```

```python
if device != "cpu":
    import cupy as xp
    from cupyx.scipy.sparse.linalg import LinearOperator
else:
    import numpy as xp
    from scipy.sparse.linalg import LinearOperator

# GMRES
sos = torch.rand(1, domain.total_num_points, dtype=torch.float).to(device) + 1.
A = LinearOperator(shape=(num_points, num_points), dtype='complex128',
                   matvec=helm.make_matvec(device=device, xp=xp, sos_map=sos))
source = xde.source.Gaussian(domain)
b = source(domain.grid)

resids = []
def callback(r):
    global resids
    resids.append(r.get())
u, info = gmres(A, b, callback=callback, callback_type='pr_norm_iter')

# Preconditioned GMRES by FNO
M = LinearOperator(shape=(num_points, num_points), dtype='complex128',
                   matvec=net.make_matvec(device=device, xp=xp, sos=sos.unsqueeze(0)))
resids_prec = []
def callback(r):
    global resids_prec
    resids_prec.append(r.get())
u, info = gmres(A, b, M=M, callback=callback, callback_type='pr_norm_iter')
```

Let us visualise the residuals norm during preconditioned GMRES iterations for random $\hat{\mathbf{c}}$.

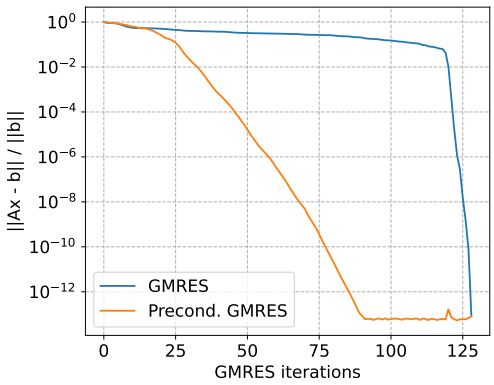


Figure 15: Non-preconditioned and FNO preconditioned full GMRES residuals norm during iterations for 16 modes, $N = 128$, and $\hat{\mathbf{c}}$ uniformly distributed on $[1, 2]$.

4.4 U-Net

In this section we consider the U-Net architecture (Figure 16). The U-Net architecture is a type of convolutional neural network (CNN) specifically designed for image segmentation tasks in computer vision [14]. The U-Net architecture is named after its U-shaped design. It is particularly effective for tasks where it is necessary to segment an image and identify specific objects or regions within that image. Image segmentation involves assigning a label to each pixel

in the image to categorise it as part of a particular object or background. The U-Net consists of the following parts:

1. **Contracting Path (Encoder):** The top part of the U-shape is called the contracting path or encoder. It's made up of a series of convolution and pooling layers. These layers reduce the spatial dimensions of the input image while increasing the number of feature channels. This helps the network to capture different levels of information and extract features at different scales.

2. **Bottleneck:** At the bottom of the U-shape there's a bottleneck layer. This is where the network captures the most abstract and consolidated features from the input image.

3. **Expansive Path (Decoder):** The bottom part of the U-shape is called the Expansive Path or Decoder. It's responsible for gradually upsampling the features back to the original image size. Each step in the decoder involves upsampling the feature map and combining it with the feature map from the corresponding layer in the contracting path through a process called skip connections.

4. **Skip Connections:** Skip connections are a crucial component of the U-Net architecture. They connect the feature maps from the contracting path to the corresponding layers in the expansive path. These connections help to preserve fine-grained spatial information that can be lost in the downsampling process. By fusing information from multiple scales, the network can better localise objects and produce more accurate segmentation results.

5. **Final Layer:** The final layer of the network typically consists of a convolutional layer that produces the segmentation map. This map assigns a label to each pixel in the image, indicating which object or category it belongs to.

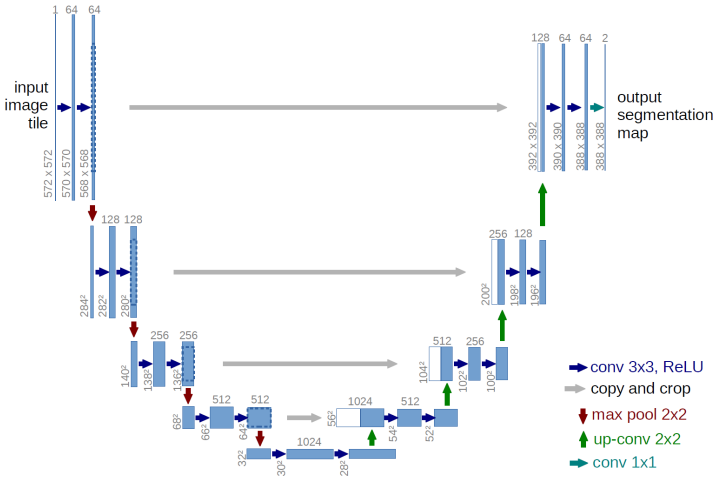
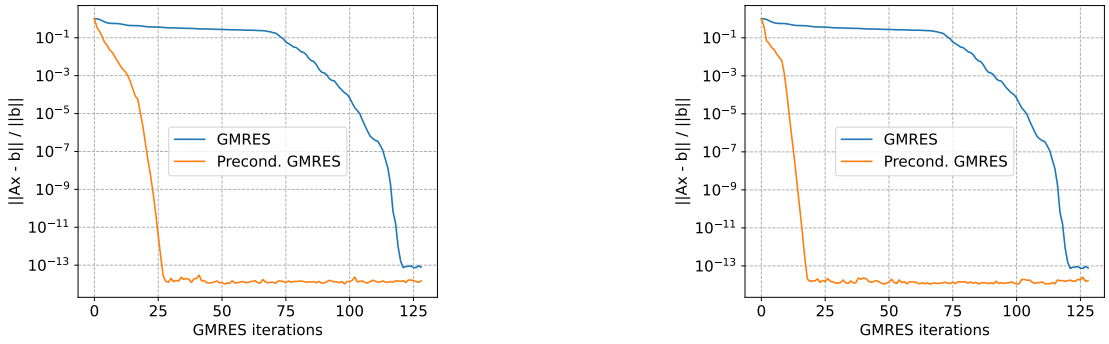


Figure 16: U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations [14].

The strength of the U-Net architecture lies in its ability to handle both contextual information (via the contraction path) and precise localisation (via the expansion path). The skip connections

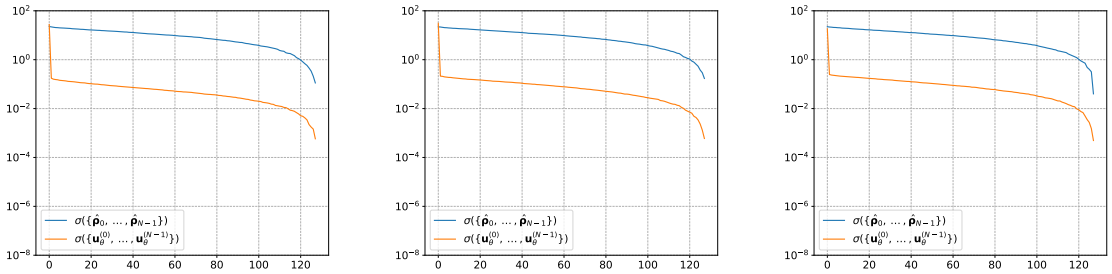
play an important role in achieving this balance. U-Net has been widely used in medical image analysis, where tasks such as segmenting organs or tumours from medical scans are critical. Its flexibility and effectiveness in handling segmentation tasks has made it a popular choice for various image analysis applications outside the medical field. This architecture can also be



(a) 2 depth (b) 4 depth

Figure 17: Non-preconditioned and U-net preconditioned full GMRES residuals norm during iterations for different depth, $N = 128$.

used to build preconditioners. Because of the convolutional nature of the model, we can use it for any size of image in the 2-dimensional case, or for any size of vector in the 1-dimensional case, which in our context is the vector from the Krylov basis. By definition of the model, the output has the same size as input. The example of code for training the model and using it as a preconditioner can be found in `example_11.ipynb`. Let us visualise the GMRES residuals norm during iterations for a different depth, which equal to the number of the skip connections (Figure 17). Then we discover the influence of depth parameter on the singular value distribution of the output. In the Figure 18 one can see that the increase of the depth does not affect the distribution.



(a) 1 depth (b) 2 depth (c) 4 depth

Figure 18: Singular values distribution of $\{\mathbf{u}_\theta^{(0)}, \dots, \mathbf{u}_\theta^{(N-1)}\}$ for fixed $\{\hat{\boldsymbol{\rho}}_0, \dots, \hat{\boldsymbol{\rho}}_{N-1}\}$ and variable depth of the U-net.

Let us visualise the residuals norm during preconditioned GMRES iterations for random $\hat{\mathbf{c}}$ (Figure 19).
This result can be improved by adding the PML layer as an aditioanl input to the U-net. To be more precise, we use a discretised sigma function $\sigma_x(x)$ for each variable $x_1, x_2, \dots, x_d$. We

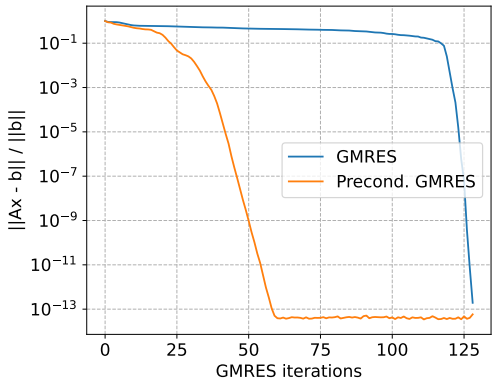
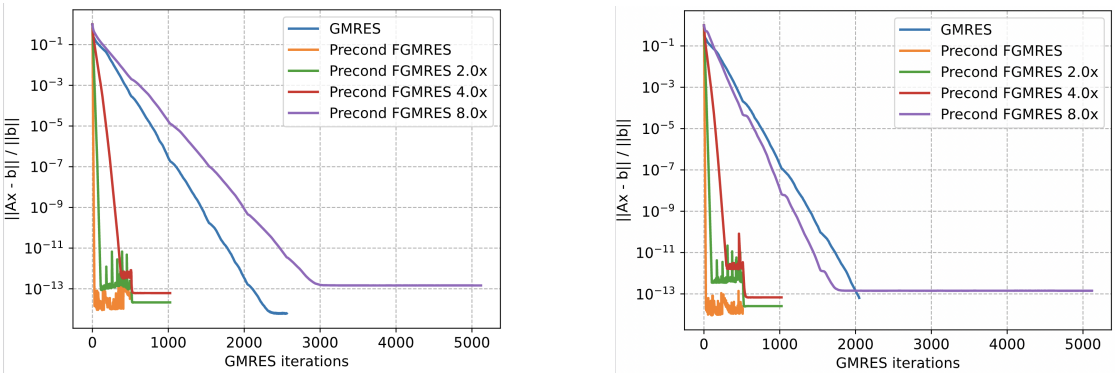


Figure 19: Non-preconditioned and U-net preconditioned full GMRES residuals norm during iterations for 2 depth, $N = 128$ and $\hat{\mathbf{c}}$ uniformly distributed on $[1, 2]$.

tested it for 2-dimensional case. The example is given in `inverse_with_pml_2d` folder. On Figure 20 one can see the difference between U-net trained on the grid 64 by 64 (orange curve) and applied as a preconditioned up to the grid 512 by 512 (purple curve) without sigma (left) and with (right).



(a) No PML (b) With PML

Figure 20: The norms of GMRES residuals preconditioned by U-Net trained without PML as an input (left) and with PML (right). The U-Net trained on the grid 64 by 64 (orange curve) and applied as a preconditioned up to the grid 512 by 512 (purple curve).

5 AXB and MAXB modules

Before explaining further results, let us present the two Python libraries developed to facilitate the research experiments. The reason is that the results in the next sections require scalability in the computational sense (training with multiple GPUs) to work with the 2-dimensional case instead of the 1-dimensional as in the previous sections. The first library `axb` is an efficient implementation of the flexible GMRES [15] in Python, supporting both CPU and GPU backends for computations with `numpy` and `cupy` Python libraries respectively. More details and examples can be found in the repository[1] and in the documentation for the `axb.gmres` method.

The `maxb` [2] library is designed to build and train the neural preconditioners, i.e. the neural networks used as preconditioners. It consists of two main submodules `maxb.nn` and `maxb.xde`. The first contains the neural network architectures discussed in the previous chapters and the training utilities for them developed with `torch` and `pytorch_lightning` libraries. The second submodule `maxb.xde` implements the discretised version of the Laplace and Helmholtz differential operators, using the discrete Fourier transform to compute derivatives, as explained in the Proposition 2. It supports two backends for computations `numpy` and `torch` to work with `numpy.ndarray` and `torch.Tensor` on both CPU and GPU devices. Note, that `maxb` uses `torch` library for computations on a GPU where `axb` uses `cupy` for the same goal. The reason is that `cupy` and `numpy` libraries have the same interface which simplifies the code that supports both devices. However, there is no problem to transform arrays from one data structure to another without uploading and downloading the data on GPU. For example, the cast from the torch to cupy looks like

```
cupy_array = cupy.asarray(torch_tensor)
```

where the inverse is

```
torch_tensor = torch.as_tensor(cupy_array, device="cuda")
```

Examples of how to use the `maxb` module to solve 1-dimensional and 2-dimensional problems can be found in the `inverse_1d` and `inverse_2d` folders. It is sufficient to run the `run.py` file with the Python interpreter version 3.8 or higher from the folders where this file is located.

```
cd example_maxb_1d/
python -m run
```

These folders contain the set of configuration files for the learning process, the model architecture, the differentiation approach, etc. They are all well commented, which should make it easier to understand the parameters they contain. There are also several types of logs that we can analyse during training. The first are the terminal logs, which are active by default. There one can observe the loss behaviour during training and when the model is saved. However, it does not give a picture of the training loss and no information about the quality of the obtained preconditioner. This information can be observed with the help of the web application TensorBoard[3] developed by Google. There one can find not only the training/validation loss plots, but also the GMRES residuals obtained during the solution of a linear system with and without a preconditioner

[1] https://gitlab.inria.fr/mshpakov/axb
[2] https://gitlab.inria.fr/mshpakov/maxb
[3] https://www.tensorflow.org/tensorboard

(Section IMAGES). To solve linear systems on GPU we require the specific version of `torch` and `cupy` libraries. Current configuration requires torch version `1.12.1` for cuda `10.2` and cupy for cuda `11.2`. This is the only configuration found by the author when both libraries work together which requires loading the module by the command

```
module load nvidia/cuda/11.2
```

This instruction is useful if you use Kraken computing cluster of Cefracs.

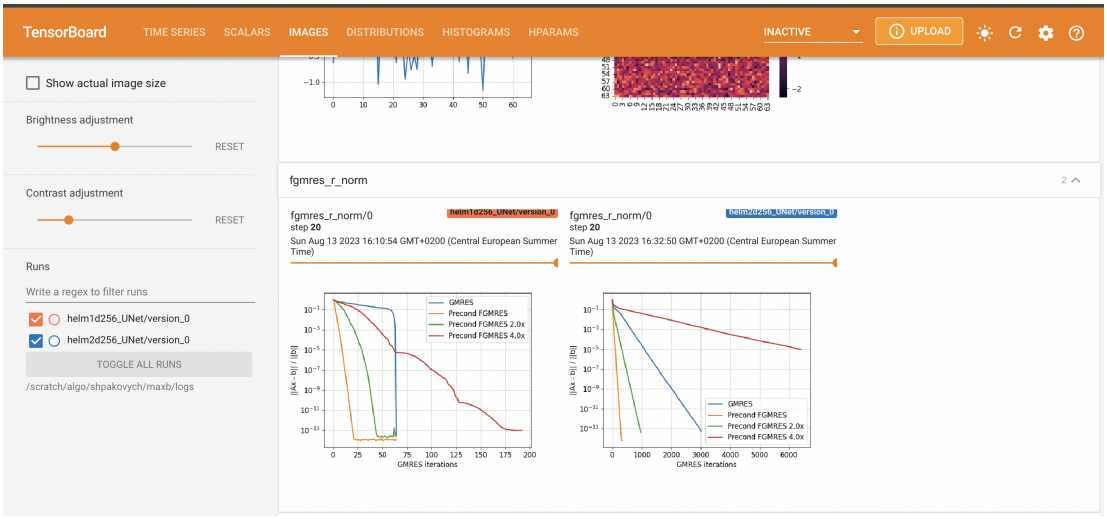


Figure 21: TensorBoard interface example

6 Unsupervised iterative learning

In the previous sections we considered the approximation of the inverse operator using a neural network $\mathcal{N}_\theta : \mathbb{C}^n \to \mathbb{C}^n$, trained to minimise $\|X - A(\hat{\mathbf{c}})\mathcal{N}_\theta(X)\|$ with respect to the parameters θ of the model for $X \in \mathbb{C}^{N\times n}$, which is normally distributed in the examples above. In this section we consider building a neural preconditioner by solving another problem that indirectly trains the model to solve the preconditioner construction problem.

6.1 Fixed-point iterations

The first indirect problem is to train a neural network to solve a fixed-point problem. It was shown in [18] that the U-Net trained to solve a fixed-point problem [17] can be used successively as a preconditioner. However, the Python implementation was difficult to read and modify, which is crucial in research. So we reimplemented the learning process [17, Algorithm 1] in our library `maxb`. The examples of training the network can be found in `fixedpoint_1d` and `fixedpoint_2d` for the U-nets without hidden state, and `fixedpoint_stateful_1d` and `fixedpoint_stateful_2d` with hidden state. These examples work, but do not exactly reproduce the results of [17]. The reason is that in the paper there is no normalisation parameter α, which is equal to 1000, and we do not give the model the discretisation of the sigma function from the PML. This should be done in the future.

6.2 GMRES iterations

Another potential possibility to train a neural network to be a preconditioner is to minimise the norm of residuals $\|\mathbf{v}_j - A(\hat{\mathbf{c}})\mathcal{N}_\theta(\mathbf{v}_j)\|$, where $\mathbf{v}_j$ is the j-th Krylov basis vector in the GMRES algorithm. It means that the learning algorithm is the GMRES with one additional step of the residuals norm minimisation. There are two possibilities of the implementation.

The first possibility is if we minimise the loss at each iteration and do not accumulate the set of losses as is done in [17, Algorithm 1, Step 14]. This approach is easy to implement and does not require differentiation of the whole GMRES algorithm. The main motivation for such an approach is that we do not need to assume the distribution of inputs to the preconditioner, as was done in the sections above.

The second possibility is to accumulate losses and differentiate them in a recurrent manner. This approach is potentially more appropriate if we use the hidden state neural network, the presence of which can improve the convergence of the preconditioned GMRES. However, the implementation becomes more difficult since all GMRES algorithm must be differentiable in terms of the machine learning framework that is used (PyTorch for example).

This idea has not implemented yet and must be tested in future.

Acknowledgements

This work was developed in collaboration with colleagues from the Concace Inria project with Airbus CR & T and Cerfacs, namely Luc Giraud, Carola Kruse and Paul Mycek.

References

[1] J P Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2), 10 1994.

[2] B. Carpentieri, I. S. Duff, L. Giraud, and G. Sylvand. Combining fast multipole techniques and an approximate inverse preconditioner for large electromagnetism calculations. *SIAM Journal on Scientific Computing*, 27(3):774–792, 2005.

[3] Bruno Carpentieri. Sparse preconditioners for dense linear systems from electromagnetic applications. 2002.

[4] I. S. Duff, L. Giraud, J. Langou, and E. Martin. Using spectral low rank preconditioners for large electromagnetic calculations. *International Journal for Numerical Methods in Engineering*, 62(3):416–434, 2005.

[5] O. G. Ernst and M. J. Gander. *Why it is Difficult to Solve Helmholtz Problems with Classical Iterative Methods*, pages 325–363. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[6] David Gilliam. Mathematics 5342 discrete fourier transform. 11 1998.

[7] Steven Johnson. Notes on perfectly matched layers (pmls). 08 2021.

[8] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. 08 2021.

[9] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Animashree Anandkumar. Fourier neural operator for parametric partial differential equations. 10 2020.

[10] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3:218–229, 03 2021.

[11] Clare D. McGillem and George R. Cooper. Continuous and discrete signal and system analysis. 1984.

[12] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*, pages 1–664. Springer Series in Operations Research and Financial Engineering. Springer Nature, 2006.

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

[14] O. Ronneberger, P.Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS*, pages 234–241. Springer, 2015. (available on arXiv:1505.04597 [cs.CV]).

[15] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003.

[16] Julius O. Smith, III. *Mathematics of the Discrete Fourier Transform (DFT): With Audio Applications*. W3K Publishing, North Charleston, 2nd edition edition, 2007.

[17] Antonio Stanziola, Simon R. Arridge, Ben T. Cox, and Bradley E. Treeby. A helmholtz equation solver using unsupervised learning: Application to transcranial ultrasound. *Journal of Computational Physics*, 441:110430, 2021.

[18] Yan-Fei Xiang. *Solution of large linear systems with a massive number of right-hand sides and machine learning*. Theses, Université de Bordeaux, December 2022.

Inria centre
at the University of Bordeaux

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803