The Finite-Element-based Solution of the Helmholtz Equation

Author: Jiayu Yang Supervisor: Matthias Heil

2023-01-15

Contents

1 Introduction

The wave equation is widely used in the study of vibrating membranes, propagating sound waves, etc. In this paper, we study the Helmholtz equation, which is the time-independent version of the wave equation, proposed by Hermann von Helmholtz, a German physicist especially known in theories of vision and thermodynamics.

Furthermore, in the study of the 2D vibrating strings, i.e. vibrating membranes, Siméon Denis Poisson solved the problem in a rectangular boundary and Gabriel Lamé solved it in an equilateral triangle boundary. However, the Helmholtz equation is applicable in studying any basic shapes of the membrane.

However, it is not always possible to obtain an analytical solution of the Helmholtz equation. Therefore, we will implement numerical methods and compare them with exact solutions in order to investigate their accuracy and complexity.

In physical mathematics, there are many numerical methods such as:

- Finite Difference Method
- Finite Volume Method
- Finite Element Method
- Method of lines
- Spectral method
- Emerging modern techniques, such as Deep Learning method

In particular, the Finite Element Method (FEM) is currently the most popular numerical method in engineering mathematics, including the modelling of mechanical, thermal and other complex systems. Except for that, FEM is also used in medical modelling, such as evaluating stresses in human bones.

In this paper, we will focus on the implementation of the finite element method (FEM) to solve the one-dimensional Helmholtz equation. In addition, we will discuss the performance of the different methods, both in terms of accuracy and complexity, so that the most appropriate method can be selected based on the desired accuracy and complexity of the modelling problem.

2 Model Problem

2.1 Wave equation & Helmholtz equation

Since the Helmholtz equation is a time-independent wave equation, we start with the wave equation, written as:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

where c is a non-negative constant. By separation of variables, we could write u(x, t) as a combination of a function $X(x)$ on x and a function $T(t)$ on t: $u(x,t) = X(x)T(t)$. Inserting back into the wave equation, we obtain:

$$XT'' = c^2 X''T \quad \Rightarrow \quad \frac{X''}{X} = \frac{T''}{c^2 T} = -k^2 \quad \Rightarrow \quad X'' + k^2 X = 0$$

Thus, this is the Helmholtz equation derived from the wave equation, where k is a constant representing the number of waves.

2.2 General form of the Helmholtz equation

The general form of the Helmholtz equation is expressed as:

$$\nabla^2 u(\boldsymbol{r}) + k^2 u(\boldsymbol{r}) = 0$$

The Helmholtz equation is also considered as an eigenvalue problem, where ∇^2 is the Laplace operator, k^2 is the eigenvalue, and $u(\mathbf{r})$ is the eigenfunction.

- In 1D, the Helmholtz equation reduces to:

$$\frac{d^2 u(x)}{dx^2} + k^2 u(x) = 0$$

- In 2D, the Helmholtz equation reduces to:

$$\frac{\partial^2 u(x,y)}{dx^2} + \frac{\partial^2 u(x,y)}{dy^2} + k^2 u(x,y) = 0$$

In this paper, we will focus on solving the 1D Helmholtz equation analytically and numerically.

2.3 Model Helmholtz equation

We will start with a simple form of the 1D Helmholtz equation on the domain $x \in [0, 1]$ with Neumann boundary conditions (BCs) on $x = 0$ and $x = 1$:

$$\begin{cases} \frac{d^2 u}{dx^2} + k^2 u(x) = 0, \quad x \in (0,1) \\ \\ \frac{du}{dx}(0) = 0, \quad \frac{du}{dx}(1) = 1 \end{cases}$$

where k doesn't depend on x, standing for the number of waves in wave problems.

We will first solve the model Helmholtz equation analytically and then take the following approaches to solve it numerically:

1. Finite Difference Method
2. Finite Element Method
3. Deep Learning Method

3 Analytical Method

To solve the two-point Boundary Value Problem:

$$\begin{cases} \frac{d^2u}{dx^2} + k^2u = 0, \quad x \in (0,1) \\ \\ \frac{du}{dx}(0) = 0, \quad \frac{du}{dx}(1) = 1 \end{cases}$$

We need to discuss the value of k:

1. $k^2 = 0$

   The equation becomes $\frac{d^2u}{dx^2} = 0$, then we have $u(x) = c_1x + c_2$, where c_1, c_2 are two constants.

   Insert into BC, we have: $u(0) = c_2 = 0, u(1) = c_1 + c_2 = 0, \Rightarrow c_1 = c_2 = 0$.

   Thus, $u(x) = 0$ is a trivial solution (zero solution).

2. $k^2 < 0$

   Write k as $k = -\beta^2$, the equation becomes: $\frac{d^2u}{dx^2} - \beta^2 u = 0$, then we have $u(x) = c_1 \sinh(\beta x) + c_2 \cosh(\beta x)$, where c_1, c_2 are two constants.

   Insert into BC, we have: $u(0) = c_2 = 0, u(1) = c_1 \sinh(\beta) + c_2 \cosh(\beta) = 0$. As $sinh(x) > 0$, we have $c_1 = c_2 = 0$.

   Thus, $u(x) = 0$ is a trivial solution (zero solution).

3. $k^2 > 0$

   With the equation $\frac{d^2u}{dx^2} + k^2u = 0$, we have $u(x) = c_1 \sin(kx) + c_2 \cos(kx)$, where c_1, c_2 are two constants.

   Insert into BC, we have: $u(0) = c_2 = 0, u(1) = c_1 \sin(k) + c_2 \cos(k) = 0, \Rightarrow c_2 = 0, c_1 \sin(k) = 0$.

   To get non-trivial solution, we require $c_1 \neq 0$, thus we have $\sin(k) = 0 \Rightarrow k = n\pi, n$ is an integer.

   Thus, $u_n(x) = \sin(n\pi x)$, where n is an integer. (We write $c_1 = 1$ here as it is combined with coefficients in total solution a_n).

Thus, we obtained the analytical solution:

$$u_n(x) = -\frac{\cos(kx)}{k \cdot \sin(k)}$$

The first derivative of the solution is:

$$u_n'(x) = \frac{\sin(k \cdot x)}{\sin(k)}$$

By superposition, we combine $u_n(x)$ and get the analytical solution of the model Helmholtz problem:

$$u(x) = \sum_{n=1}^{\infty} a_n \sin(n\pi x), \quad n = 1, 2, 3, ...$$

We write the analytical solutions into the `Analytical` function:

```
def Analytical(k):
    u   = lambda x: - np.cos(k * x) / (k * np.sin(k))
    u_x = lambda x: np.sin(k * x) / np.sin(k)
    return (u, u_x)
```

4 Finite Difference Method (FDM)

The finite difference method (FDM) is one of the most traditional methods for solving differential equations numerically.

First, we need to approximate u(x) at discrete points. Define a uniform grid $\{x_i = ih | i = 0, 1, \ldots, n\}$ on $x \in (0, 1)$ with (n+1) nodes, where the grid spacing is all $h = \frac{1-0}{n} = \frac{1}{n}$. Then, the approximated solution is expressed as:

$$u = \sum_{i=0}^{n} u(x_i) = \sum_{i=0}^{n} u_i \quad \text{, where } i = 0, 1, ..., n$$

By using the Taylor's theorem, the first order and the second order derivative of $u(x)$ by

$$\frac{du_i}{dx} \approx \frac{u_{i+1} - u_{i-1}}{2h} \approx \frac{u_{i+1} - u_i}{h} \approx \frac{u_i - u_{i-1}}{h}$$

$$\frac{d^2u_i}{dx^2} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$

Insert $\frac{d^2u(x_i)}{dx^2}$ into our model Helmholtz equation, we obtain:

$$\frac{d^2u_i}{dx^2} + k^2u_i = \frac{u_{i+1} + (h^2k^2 - 2) \cdot u_i + u_{i-1}}{h^2} = 0$$

$$\Rightarrow u_{i+1} + (h^2k^2 - 2) \cdot u_i + u_{i-1} = 0$$

When $i - 1 = 0$, the first point of the domain u_0 satisfies the equation:

$$u_0 + (h^2k^2 - 2) \cdot u_1 + u_2 = 0$$

When $i + 1 = n$, the last point of the domain u_n satisfies the equation:

$$u_{n-2} + (h^2k^2 - 2) \cdot u_{n-1} + u_n = 0$$

We will then insert the first derivative $\frac{du_i}{dx}$ into the BCs $\frac{du}{dx}(0) = 0, \quad \frac{du}{dx}(1) = 1$ to get:

$$\frac{du}{dx}(0) = \frac{du_0}{dx} = 0 \quad \Rightarrow \quad \frac{du_0}{dx} = \frac{u_1 - u_0}{h} = 0 \quad \Rightarrow \quad u_0 = u_1$$

$$\frac{du}{dx}(1) = \frac{du_n}{dx} = 1 \quad \Rightarrow \frac{du_n}{dx} = \frac{u_n - u_{n-1}}{h} = 1 \quad \Rightarrow \quad u_n = u_{n-1} + h$$

Combine the above equations of u_0 and u_n together, we deduce the following relationship:

$$\begin{cases} u_0 = u_1 \\ u_0 + (h^2k^2 - 2) \cdot u_1 + u_2 = 0 \end{cases} \qquad \Rightarrow (h^2k^2 - 1) \cdot u_1 + u_2 = 0$$

$$\begin{cases} u_n = u_{n-1} + h \\ u_{N-2} + (h^2k^2 - 2) \cdot u_{n-1} = 0 \end{cases} \qquad \Rightarrow u_{n-2} + (h^2k^2 - 1) \cdot u_{n-1} = -h$$

In summary, we obtain the functions on each point of the domain:

$$\begin{cases} (h^2k^2 - 1) \cdot u_1 + u_2 = 0 & \text{, if } i = 0 \\ u_{i+1} + (h^2k^2 - 2) \cdot u_i + u_{i-1} = 0 & \text{, if } i = 1, ..., (n-1) \\ u_{n-2} + (h^2k^2 - 1) \cdot u_{n-1} = -h & \text{, if } i = n \end{cases}$$

4.1 Matrix System

Now we write them in a matrix system:

$$\boldsymbol{Au} = \boldsymbol{d}$$

$$\begin{bmatrix} h^2k^2-1 & 1 & 0 & 0 & 0 & ... & 0 \\ 1 & h^2k^2-2 & 1 & 0 & 0 & ... & 0 \\ 0 & 1 & h^2k^2-2 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & h^2k^2-2 & 1 & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 1 & h^2k^2-2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & h^2k^2-1 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \dots \\ u_{n-1} \\ u_n \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ ... \\ 0 \\ -h \end{Bmatrix}$$

Therefore, we can obtain the solution $\{u_i\}_{i=0}^{N}$ by solving this matrix system.

We will now write the matrix system in codes. The matrix $\mathbf{A}$ is written in the `MatrixA` function:

```
def MatrixA(self, i: int, j: int):
    if i == j:
        if i == 0 or i == self.N:
            return -1 + (self.h**2) * (self.k**2)
        else:
            return -2 + (self.h**2) * (self.k**2)
    elif abs(i - j) == 1:
        return 1
    else:
        return 0
```

Then, we insert the `MatrixA` into the matrix system, and implement the built-in function `linalg.solve` to solve the linear system in our function `solve`:

```
def solve(self):
    for i in range(self.N + 1):
        self.A[i, i] = self.MatrixA(i, i)
        if i == 0:
            self.A[i, i + 1] = self.MatrixA(i, i + 1)
        elif i == self.N:
            self.A[i, i - 1] = self.MatrixA(i, i + 1)
        else:
            self.A[i, i - 1] = self.MatrixA(i, i - 1)
            self.A[i, i + 1] = self.MatrixA(i, i + 1)
    self.d[-1] = - self.h
    self.u = np.linalg.solve(self.A, self.d)
    return self.u
```

The solve function gives us the solution values $\{u_i\}_{i=0}^{n}$ on each discrete point of the domain $\{x_i\}_{i=0}^{n}$.

4.2 Outputs

We take an example to check whether this method works. Suppose the value of k is 4, then our model problem becomes:

$$\begin{cases} \frac{d^2u}{dx^2}(x) + 4^2u(x) = 0, & x \in (0,1) \\ \frac{du}{dx}(0) = 0, \frac{du}{dx}(1) = 1 \end{cases}$$

Take the number of partitions as $n = 4$, the domain $x \in [0,1]$ is then equally partitioned to 4 sub-domains with the length $h = \frac{1}{4}$. Thus, our trial solution is expressed as:

$$\tilde{u}(x) = \sum_{i=0}^{4} u_i(x) \quad \text{, where } i = 0,1,2,3,4$$

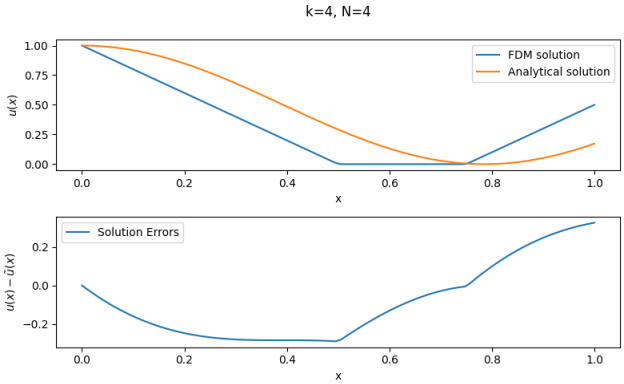
The matrix system becomes:

$$\Rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 \\ 0 & 1 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -0.25 \end{Bmatrix}$$

We will then solve this linear system by solving the following five equations:

$$\begin{cases} 0 \cdot u_0 + 1 \cdot u_1 = 0 \\ 1 \cdot u_0 - 1 \cdot u_1 + 1 \cdot u_2 = 0 \\ 1 \cdot u_1 - 1 \cdot u_2 + 1 \cdot u_3 = 0 \\ 1 \cdot u_2 - 1 \cdot u_3 + 1 \cdot u_4 = 0 \\ 1 \cdot u_3 + 0 \cdot u_4 = -0.25 \end{cases} \Rightarrow \begin{cases} u_0 = 0.25 \\ u_1 = 0 \\ u_2 = -0.25 \\ u_3 = -0.25 \\ u_4 = 0 \end{cases}$$

With the discrete point solutions, we link them and plot in blue as shown below, and plot the previously calculated analytical solutions in orange:

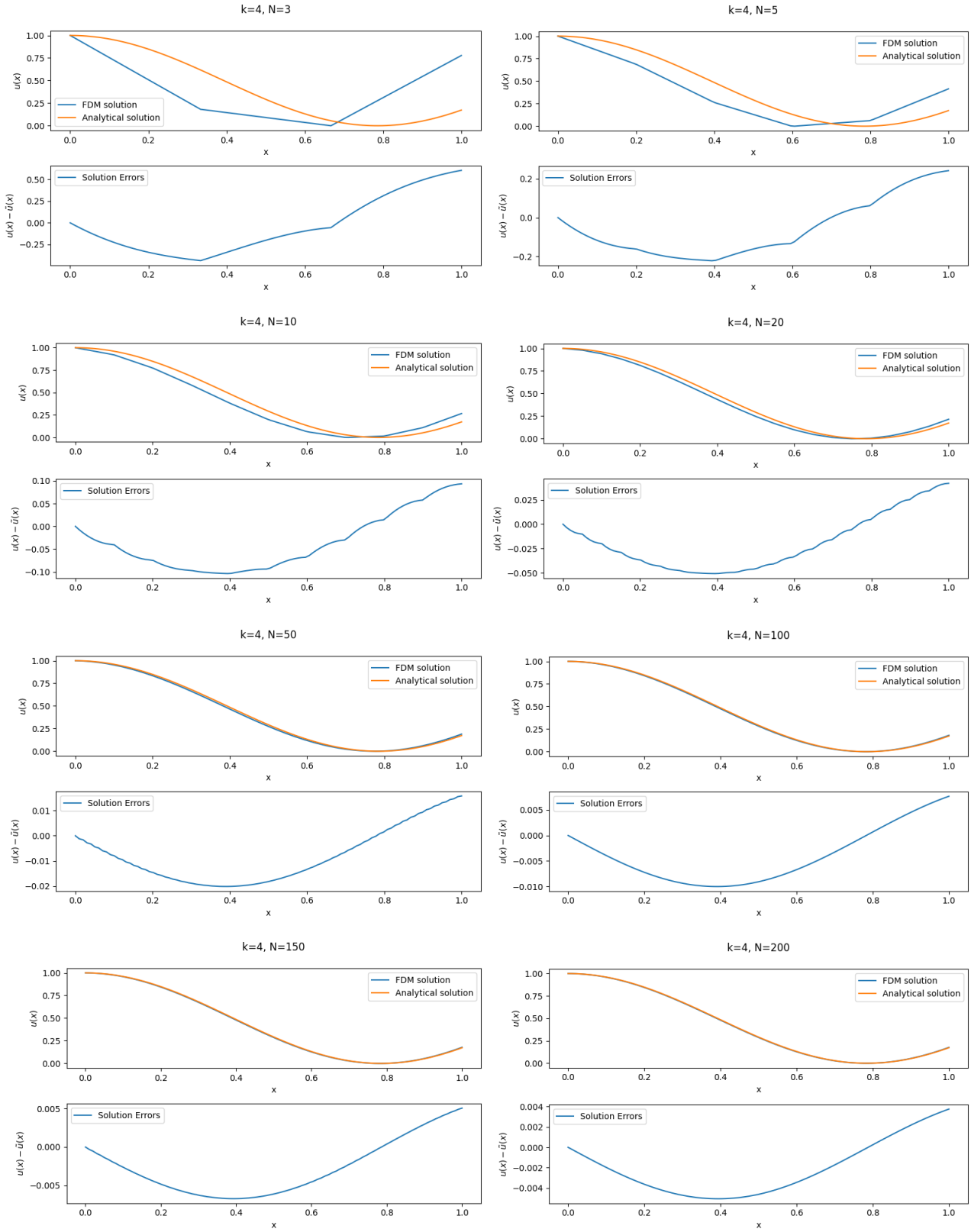


The shape of the FDM solution approximately matches the analytical solution. To obtain a more accurate FDM solution, we can divide the domain into more sub-domains to obtain more discrete solutions.

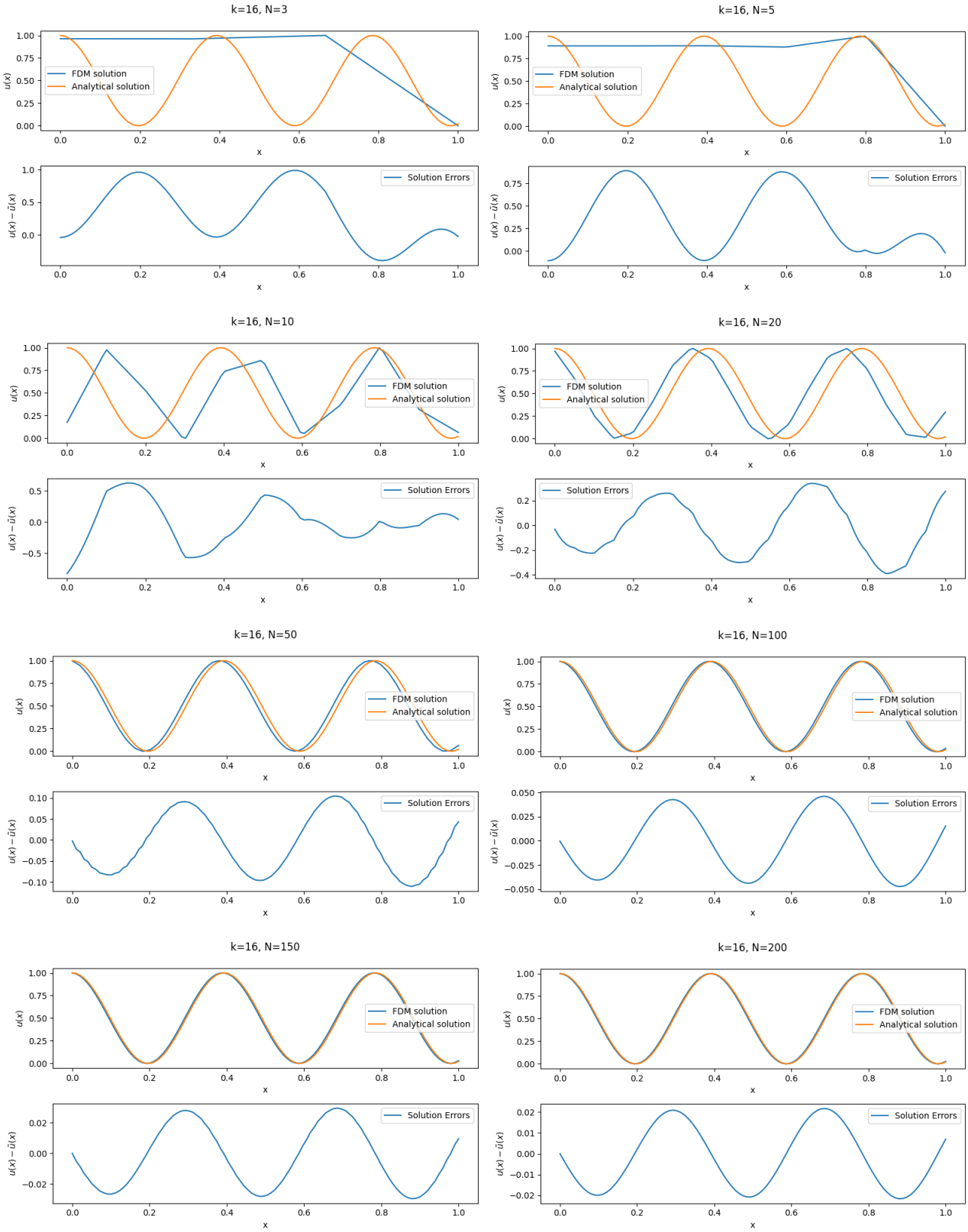
Finally, we will solve the model Helmholtz equation respectively for the cases when k = 4, k = 16, and k = 32. We choose the number of partitions as N = 3, 5, 10, 20, 50, 100, 150, 200.

In the following plots, we show both the FDM solution $\tilde{u}(x)$ in blue compared to the analytical solution $u(x)$ in orange, and also the solution errors calculated from the difference between them: $u(x) - \tilde{u}(x)$.

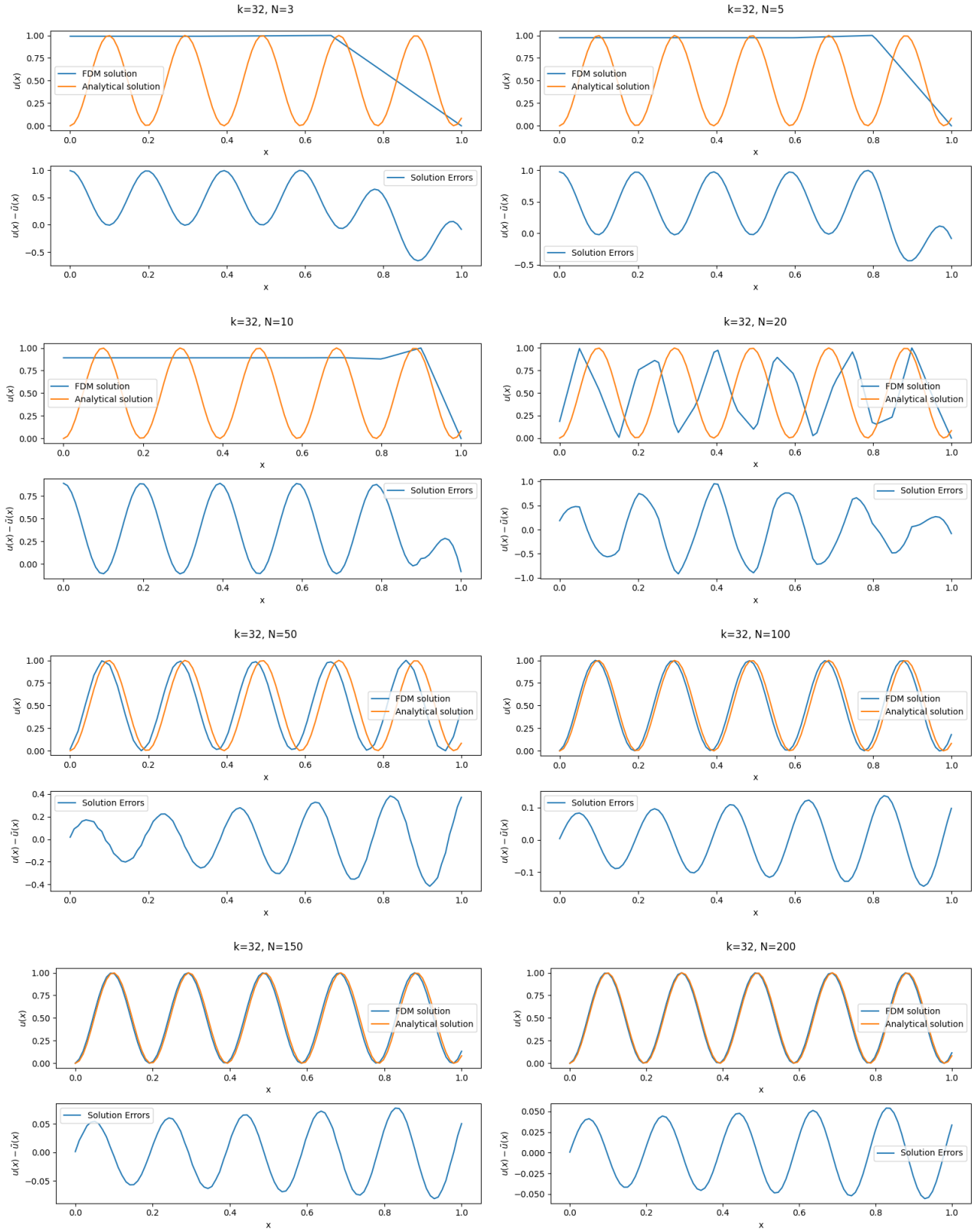
For the case when k = 4:



For the case when k = 16:



For the case when k = 32:



We can roughly observe that the greater the value of k, the more partitions n are required to obtain an approximately accurate trial solution.

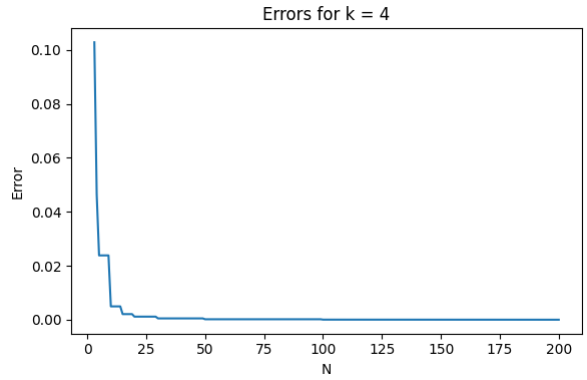
To delve into the performance of the FDM, we will analyze its error and time consumption specifically.

4.3 Performance Analysis

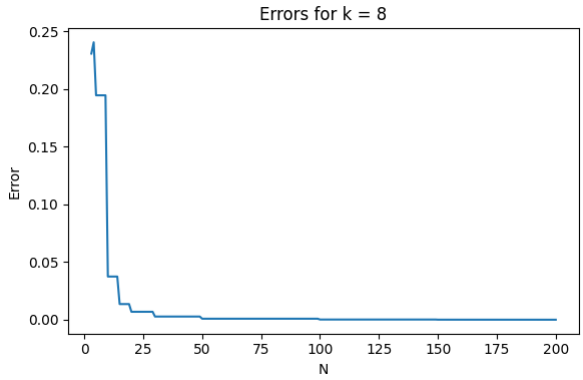
4.3.1 Errors

Then we plot the errors of the cases with k = 4, k = 8, k = 16, and k = 32.

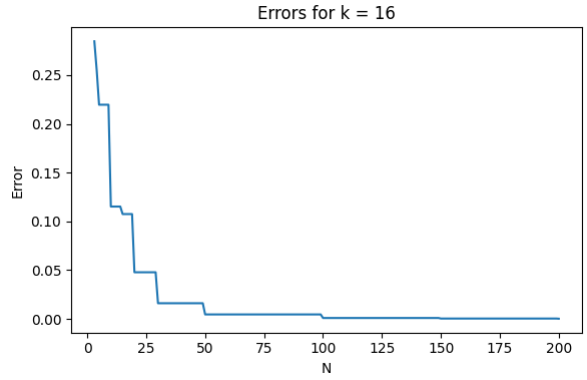
For the case when k = 4:



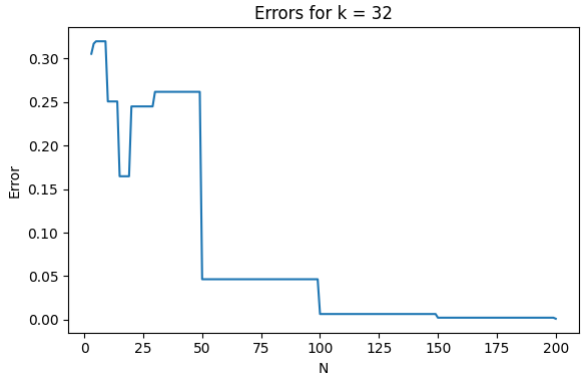
For the case when k = 8:



For the case when k = 16:



For the case when k = 32:



For k = 4, the first eligible number of segmentation of errors less than or equal to 10^{-3} is N = 30.

For k = 8, the first eligible number of segmentation of errors less than or equal to 10^{-3} is N = 50.

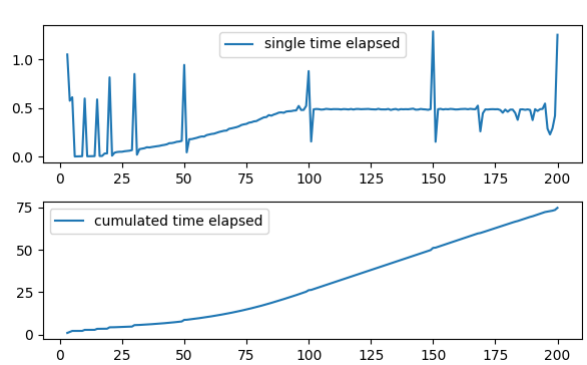
For k = 16, the first eligible number of segmentation of errors less than or equal to 10^{-3} is N = 100.

For k = 32, the max number of partitions N = 200 is not enough to achieve errors less than or equal to 10^{-3}.

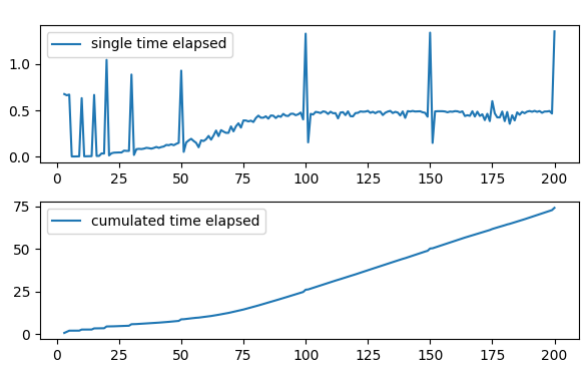
4.3.2 Time Comsumption

We will now plot the time elapsed for the cases respectively when k = 4, k = 8, k = 16, and k = 32 as below:

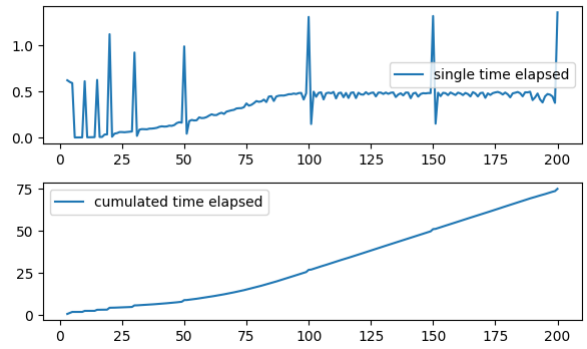
For the case when k = 4:



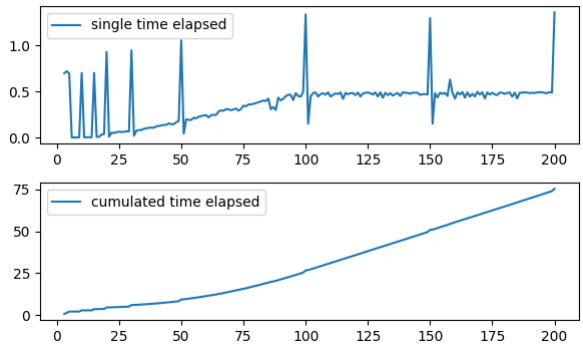
For the case when k = 8:



For the case when k = 16:



For the case when k = 32:



5 Finite Element Method (FEM)

Now we will implement the FEM to solve our model Helmholtz equation:

$$\begin{cases} \frac{d^2u}{dx^2} + k^2u(x) = 0, \quad x \in (0,1) \\ \\ \frac{du}{dx}(0) = 0, \quad \frac{du}{dx}(1) = 1 \end{cases}$$

5.1 Trial Solution

Assume the trial solution of u(x) as $\tilde{u}(x) = \sum_{i=0}^{n} c_i\varphi_i(x)$, where $\{c_i\}_{i=0}^{n}$ are the unknown constants and $\{\varphi_i(x)\}_{i=0}^{n}$ are the basis functions. In order to obtain a trial solution that is closest to the true solution, we need to choose the proper constant coefficients $\{c_i\}_{i=0}^{n}$.

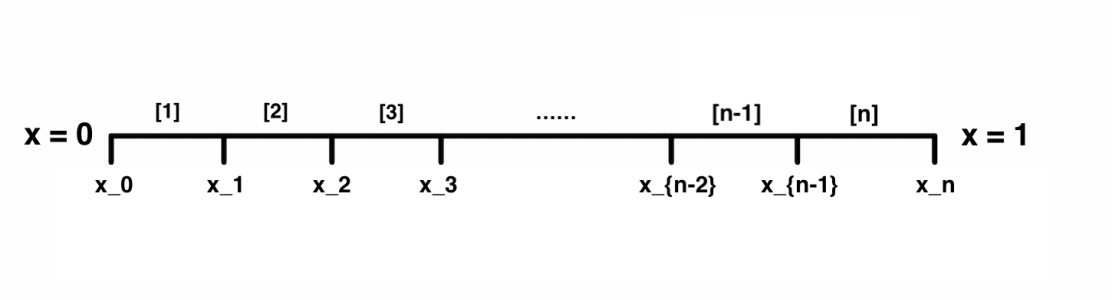
Define the measure of the difference between $u(x)$ and $\tilde{u}(x)$ as the residual $\epsilon(x)$:

$$\epsilon(x) = \frac{d^2\tilde{u}(x)}{dx^2} + k^2\tilde{u}(x)$$

If the trial solution $\tilde{u}(x)$ is exactly the true solution $u(x)$, then $\epsilon(x) = 0$.

5.2 Segmentation of the domain

By using the FEM, we will now divide the domain of interest $x \in (0,1)$ into n partitioned elements with the equal length. Denote the length of partition by h, $h = \frac{1}{n(1-0)} = \frac{1}{n}$.



As the above diagram shows, each element is enclosed by two nodes, so we have (n+1) nodes in total. The i-th node of the domain is denoted as $x_i = ih = \frac{i}{n}$, where $i \in \{0, 1, \dots n\}$. The left node and the right node of x_i are denoted by x_l and x_r, shown as:

$$x_l = \begin{cases} x_i = ih = 0 & , i = 0 \\ x_{i-1} = ih & , i = 1, ..., n \end{cases}$$

$$x_r = \begin{cases} x_{i+1} = (i+1)h & , i = 0, 1, ..., (n-1) \\ x_i = in & , i = n \end{cases}$$

Note that $x_r - x_l = h = \frac{1}{n}$. Now we write the nodes x_i, x_l, x_r into codes:

```
# the i-th node xi
x_i = self.x[i]

# the left node xl
if i == 0:
  x_l = x_i
else:
  x_l = self.x[i - 1]

# the right node xr
if i == N:
  x_r = x_i
else:
  x_r = self.x[i + 1]
```

In order to find the constant coefficients $\{c_i\}_{i=0}^{n}$, we require the trial solution $\tilde{u}(x)$ at the nodes $x_0, x_1, ..., x_n$ equal to the true solution $u(x)$. Equivalently, the residual is required to be locally minimized to zero, providing (n+1) equations to solve the unknown constants $\{c_i\}_{i=0}^{n}$.

$$\epsilon(x_i) = 0, \quad i = 0, 1, ..., n$$

5.3 Galerkin's Method

In order to minimize the residual $\epsilon(x)$ to zero at nodes $x_0, x_1, \ldots, x_n$, we will implement the Galerkin's method to make the residual being orthogonal to the basis functions $\{\varphi_j(x)\}_{j=0}^{n}$. Write the orthogonality as the inner product:

$$< \epsilon(x), \varphi_j(x) >= 0, \quad \text{where } j = 0, 1, ..., n$$

$$\Rightarrow \int_0^1 \epsilon(x)\varphi_j(x)dx = 0$$

Substitute into the residual function $\epsilon(x) = \frac{d^2\tilde{u}(x)}{dx^2} + k^2\tilde{u}(x)$, we get the standard Galerkin equation:

$$\Rightarrow \int_0^1 (\frac{d^2\tilde{u}}{dx^2} + k^2\tilde{u}) \cdot \varphi_j(x)dx = 0$$

By using the integration by parts, we obtain the modified Galerkin equation:

$$[\frac{d\tilde{u}}{dx} \cdot \varphi_j(x)]_0^1 - \int_0^1 [\frac{d\tilde{u}}{dx} \cdot \frac{d\varphi_j(x)}{dx} - k^2\tilde{u} \cdot \varphi_j(x)]dx = 0$$

This integral formulation after integration by parts is the variational formulation of the Helmholtz equation. Note that this is a weak formulation, as we have a weaker requirement for the differentiability of u(x) than the original formulation.

Substitute into the trial solution $\tilde{u}(x) = \sum_{i=0}^{n} c_i \varphi_i(x)$ and the BCs $\frac{d\tilde{u}}{dx}(0) = 0$, $\frac{d\tilde{u}}{dx}(1) = 1$, the equation is then reduced to:

$$\int_0^1 [-c_i \cdot \frac{d\varphi_i(x)}{dx} \frac{d\varphi_j(x)}{dx} + k^2 c_i \cdot \varphi_i(x) \cdot \varphi_j(x)]dx = -[\frac{d\tilde{u}}{dx} \cdot \varphi_j(x)]_0^1$$

$$\Rightarrow \int_0^1 c_i \cdot [-\frac{d\varphi_i(x)}{dx} \frac{d\varphi_j(x)}{dx} + k^2 \varphi_i(x) \cdot \varphi_j(x)]dx = -\varphi_j(1) \mid_{j=n}, \quad \text{where } j = 0, ..., n$$

Therefore, we obtain the linear system. In order to solve it to obtain the coefficients $\{c_i\}_{i=0}^n$, we then need to find the basis functions $\varphi(x)$.

5.4 Basis function

By using the FEM with linear elements, we assume that the function on the element is linearly related to x, so we define the local linear function on the ith element as:

$$\tilde{u}(x) = \alpha_0 + \alpha_1 \cdot x$$

Each element is bounded by the left node x_l and the right node x_r, which have values:

$$\begin{cases} \tilde{u}(x_l) = \alpha_0 + \alpha_1 \cdot x_l \\ \tilde{u}(x_r) = \alpha_0 + \alpha_1 \cdot x_r \end{cases}$$

The values of α_0 and α_1 are then:

$$\alpha_0 = \frac{\tilde{u}(x_l)x_r - \tilde{u}(x_r)x_l}{x_r - x_l} \alpha_1 = \frac{\tilde{u}(x_r) - \tilde{u}(x_l)}{x_r - x_l}$$

Thus, the linear function becomes:

$$\begin{aligned} \tilde{u}(x) &= \alpha_0 + \alpha_1 \cdot x \\ &= \frac{\tilde{u}(x_l)x_r - \tilde{u}(x_r)x_l}{x_r - x_l} + \frac{\tilde{u}(x_r) - \tilde{u}(x_l)}{x_r - x_l} \cdot x \\ &= \frac{x - x_l}{x_r - x_l} \cdot \tilde{u}(x_r) + \frac{x_r - x}{x_r - x_l} \cdot \tilde{u}(x_l) \end{aligned}$$

As $x_r - x_l = h$, we then have:

$$\tilde{u}(x) = \frac{x - x_l}{h} \cdot \tilde{u}(x_r) + \frac{x_r - x}{h} \cdot \tilde{u}(x_l) = L_1 \tilde{u}(x_l) + L_2 \tilde{u}(x_r)$$

where L_l, L_r are linear operators $L_l = \frac{x - x_l}{h}, L_r = \frac{x_r - x}{h}$.

Note that the u_l and u_r here denote the left and right local nodes of an element, instead of the whole domain. To be clear, we write the approximated solution of each local element as:

$$\tilde{u}^{[1]}(x) = L_l^{[1]}\tilde{u}(x_l) + L_r^{[1]}\tilde{u}(x_r), \text{ where } \begin{cases} L_l^{[1]} = \frac{x-x_l}{h} = 1 - \frac{x}{h} \\ L_r^{[1]} = \frac{x_r - x}{h} = \frac{x}{h} \end{cases} \quad \text{within element [1]}$$

$$\tilde{u}^{[2]}(x) = L_l^{[2]}\tilde{u}(x_l) + L_r^{[2]}\tilde{u}(x_r), \text{ where } \begin{cases} L_l^{[2]} = \frac{x-x_l}{h} = 2 - \frac{x}{h} \\ L_r^{[2]} = \frac{x_r - x}{h} = \frac{x}{h} - 1 \end{cases} \quad \text{within element [2]}$$

.....

$$\tilde{u}^{[n]}(x) = L_l^{[n]}\tilde{u}(x_l) + L_r^{[n]}\tilde{u}(x_r), \text{ where } \begin{cases} L_l^{[n]} = \frac{x-x_l}{h} = -\frac{x-1}{h} \\ L_r^{[n]} = \frac{x_r - x}{h} = 1 + \frac{x-1}{h} \end{cases} \quad \text{within element [n]}$$

For each node x_i, $i = 0, 1, ..., n$, we define the basis functions $\varphi_i(x)$ to describe the nodal functions:

$$\varphi_0(x) = \begin{cases} L_1^{[1]}(x) \ , \text{ with element [1]} \\ 0 \end{cases} = \begin{cases} 1 - \frac{x}{h} \ , x_0 < x < x_1 \\ 0 \quad , \text{ elsewhere} \end{cases} \quad \text{at node } x_0$$

$$\varphi_1(x) = \begin{cases} L_2^{[1]}(x) \ , \text{ with element [2]} \\ L_1^{[2]}(x) \ , \text{ with element [3]} \end{cases} = \begin{cases} n + \frac{x-1}{h} \ , x_0 < x < x_1 \\ 2 - \frac{x}{h} \quad , x_1 < x < x_2 \end{cases} \quad \text{at node } x_1$$

.....

$$\varphi_{n-1}(x) = \begin{cases} L_2^{[n-1]}(x) \ , \text{ with element } [n-1] \\ L_1^{[n]}(x) \quad , \text{ with element } [n] \end{cases} = \begin{cases} 2 + \frac{x-1}{h} \ , x_{n-2} < x < x_{n-1} \\ n - \frac{x}{h} \quad , x_{n-1} < x < x_n \end{cases} \quad \text{at node } x_{n-1}$$

$$\varphi_n(x) = \begin{cases} L_2^{[n]}(x) \ , \text{ with element } [n] \\ 0 \end{cases} = \begin{cases} 1 + \frac{x-1}{h} \ , x_{n-1} < x < x_n \\ 0 \quad , \text{ elsewhere} \end{cases} \quad \text{at node } x_n$$

Then, we write the basis (nodal) functions and the elements in a table:

Element [e]	$\varphi_0(x)$	$\varphi_1(x)$	.....	$\varphi_{n-1}(x)$	$\varphi_n(x)$
[1]	$1 - \frac{x}{h}$	$n + \frac{x-1}{h}$	.....	0	0
[2]	0	$2 - \frac{x}{h}$	.....	0	0
.....	.....	.....	.....	.....	.....
$[n-1]$	0	0		$2 + \frac{x-1}{h}$	0
$[n]$	0	0	.....	$n - \frac{x}{h}$	$1 + \frac{x-1}{h}$

As $x_i = ih \Rightarrow i = \frac{x_i}{h}$ and $h = \frac{1}{n}$, the basis function can be rewritten as:

$$\varphi_i(x) = \begin{cases} (n - i + 1) + \frac{x-1}{h} = 1 + \frac{x-x_i}{h} = 1 - \frac{|x-x_i|}{h} \ , \text{ if } x_{i-1} < x < x_i \\ (i+1) - \frac{x}{h} = 1 + \frac{x_i - x}{h} = 1 - \frac{|x-x_i|}{h} \quad , \text{ if } x_i < x < x_{i+1} \end{cases}$$

Thus, we obtain the basis functions:

$$\varphi_i(x) = 1 - \frac{|x - x_i|}{h} \quad , i \in \{0, 1, ..., n\}$$

We will now write the basis functions in codes. With $x_{i-1} = x_l, x_{i+1} = x_r$, we write the cases $x_l < x < x_i$ and $x_i < x < x_r$ into the `step` function:

$$\begin{aligned} step(x) &= heaviside(x - x_l, 1) - heaviside(x - x_r, 0) \\ &= \begin{cases} 0 - 0 = 0 & , x < x_l \\ 1 - 0 = 1 & , x_l \leq x \leq x_r \\ 1 - 1 = 0 & , x > x_r \end{cases} \end{aligned}$$

This step function makes sure the local basis function $\varphi_i(x)$ only lying inside the region $x \in [x_l, x_r]$.

$$\Rightarrow \varphi_i(x) = step(x) \cdot (1 - \frac{|x - x_i|}{h})$$

Now we define the basis function $\varphi_i(x)$ as `phi(i)`:

```
def phi(self, i: int) -> Callable:
    xm = self.x[i]
    xl = self.x[i - 1] if i != 0 else xm
    xr = self.x[i + 1] if i != self.N else xm
    step = lambda x: np.heaviside(x - xl, 1) - np.heaviside(x - xr, 0)
    return lambda x: step(x) * (1 - np.abs((x - xm)) / self.h)
```

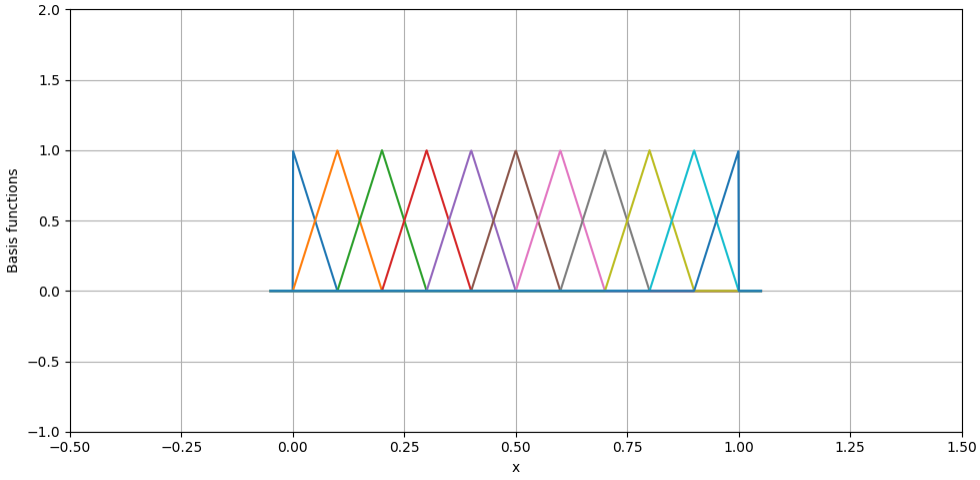
We also calculate the first-order derivative of the basis function as $\varphi_i'(x)$, written in codes:

$$\varphi_i'(x) = \begin{cases} step(x) \cdot \frac{-1}{h} & , i = 0 \\ step(x) \cdot \frac{+1}{h} & , i = n \\ step(x) \cdot \frac{-1}{h} \cdot sign(x - x_i) = \begin{cases} step(x) \cdot \frac{-1}{h} & \text{if } x \geq x_i \\ step(x) \cdot \frac{+1}{h} & \text{if } x < x_i \end{cases} & , i = 1, ..., n-1 \end{cases}$$

We define the $\varphi_i'(x)$ as `phi_x(i)`:

```
def phi_x(self, i: int) -> Callable:
    x_i = self.x[i]
    x_l = self.x[i - 1] if i != 0 else x_i
    x_r = self.x[i + 1] if i != self.N else x_i
    step = lambda x: np.heaviside(x - x_l, 1) - np.heaviside(x - x_r, 0)
    if i == 0:
        return lambda x: step(x) * (- 1 / self.h)
    elif i == self.N:
        return lambda x: step(x) * (+ 1 / self.h)
    else:
        return lambda x: step(x) * (- 1 / self.h) * np.sign(x - x_i)$\varphi_i'(x)$,
```

We are now able to find the basis functions. For example, for the case when the value of k is $k = 4$ and the number of partition is $n = 10$, the basis functions are plotted as:



5.5 Matrix System

With the calculated basis functions, we are now able to solve the linear system:

$$c_i \cdot \int_0^1 [-\frac{d\varphi_i(x)}{dx}\frac{d\varphi_j(x)}{dx} + k^2\varphi_i(x) \cdot \varphi_j(x)]dx = -\varphi_j(1) \mid_{j=n}, \quad \text{where } j = 0, ..., n$$

Transform this equation into the matrix system:

$$\boldsymbol{Ac} = \boldsymbol{d}$$

- $\boldsymbol{A} = \{a_{ij}\}_{i,j=0}^{n}$ is a (n+1) x (n+1) matrix, containing the left hand side of the linear system except for c_i:

$$a_{ij} = \int_0^1 (-\varphi_i'(x)\varphi_j'(x) + k^2\varphi_i(x)\varphi_j(x))dx, \quad i, j = 0, 1, ..., n$$

where $\int_0^1 k^2\varphi_i(x)\varphi_j(x)dx, \quad i, j \in \{0, ..., n\}$ is calculated as:

$$\int_0^1 \varphi_i(x)\varphi_j(x)dx = \begin{cases} \frac{k^2h}{3} & \text{, if } (i = 0, j = 0) \text{ or } (i = N, j = N) \\ \frac{2k^2h}{3} & \text{, if } i = j, i = 1, 2, ..., N-1 \\ \frac{k^2h}{6} & \text{, if } |i - j| = 1 \\ 0 & \text{, otherwise} \end{cases}$$

and $\int_0^1 \varphi_i'(x)\varphi_j'(x)dx, \quad i, j \in \{0, ..., n\}$ is calculated as:

$$\int_0^1 \varphi_i'(x)\varphi_j'(x)dx = \begin{cases} \frac{1}{h} & \text{, if } (i = 0, j = 0) \text{ or } (i = N, j = N) \\ \frac{2}{h} & \text{, if } i = j, i = 1, 2, ..., N-1 \\ \frac{-1}{h} & \text{, if } |i - j| = 1 \\ 0 & \text{, otherwise} \end{cases}$$

Thus, the a_{ij} becomes:

$$a_{ij} = \int_0^1 (-\varphi_i'(x)\varphi_j'(x) + k^2\varphi_i(x)\varphi_j(x))dx = \begin{cases} -\frac{1}{h} + \frac{k^2 h}{3} & \text{, if } (i=0, j=0) \text{ or } (i=N, j=N) \\ -\frac{2}{h} + \frac{2k^2 h}{3} & \text{, if } i=j, i=1,2,...,N-1 \\ \frac{1}{h} + \frac{k^2 h}{6} & \text{, if } |i-j|=1 \\ 0 & \text{, otherwise} \end{cases}$$

- $\boldsymbol{c} = \{c_j\}_{j=0}^n$ is a (n+1) x 1 vector, containing the unknown constant coefficients:

$$\boldsymbol{c} = \{c_j\}, \quad j = 0, 1, .., n$$

- $\boldsymbol{d} = \{d_j\}_{j=0}^n$ is a (n+1) x 1 vector, containing the right hand side of the linear system:

$$\boldsymbol{d} = -\varphi_j(1) \mid_{j=n}$$

Thus, the matrix system becomes:

$$\begin{bmatrix} -\frac{1}{h} + \frac{k^2 h}{3} & \frac{1}{h} + \frac{k^2 h}{6} & 0 & 0 & 0 \\ 1 & -\frac{2}{h} + \frac{2k^2 h}{3} & \frac{1}{h} + \frac{k^2 h}{6} & 0 & 0 \\ 0 & \frac{1}{h} + \frac{k^2 h}{6} & -\frac{2}{h} + \frac{2k^2 h}{3} & \frac{1}{h} + \frac{k^2 h}{6} & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & \frac{1}{h} + \frac{k^2 h}{6} & -\frac{2}{h} + \frac{2k^2 h}{3} & \frac{1}{h} + \frac{k^2 h}{6} \\ 0 & 0 & 0 & \frac{1}{h} + \frac{k^2 h}{6} & -\frac{1}{h} + \frac{k^2 h}{3} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ ... \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ ... \\ 0 \\ -\varphi_n(1) \end{bmatrix}$$

With the known basis functions $\varphi(x)$, we will obtain the (n+1) constant coefficients $\{c_i\}_{i=0}^n$ by solving this matrix system.

Now we will write the matrix system into codes. The matrix $\boldsymbol{A}$ is written as:

```
def Matrix_A(self, i: int, j: int): # for matrix A
    if i == j:
        if i == 0 or i == self.N:
            return - 1 / self.h + self.k ** 2 * self.h / 3
        else:
            return - 2 / self.h + self.k ** 2 * self.h * 2 / 3
    elif abs(i - j) == 1:
        return 1 / self.h + self.k ** 2 * self.h / 6
    else:
        return - 0
```

The vector $\boldsymbol{d}$ is written as:

```
def Vector_d(self, j): # for vector d
    phi_j = self.bases[j]
    return self.ga * phi_j(self.a) - self.gb * phi_j(self.b)
```

Then, we insert them into the matrix system and solve the constant vector $\boldsymbol{c}$. We use the built-in function `linalg.solve` from the `numpy` library to solve the matrix system $\boldsymbol{Ac} = \boldsymbol{d}$.

```
def solve(self):  # To solve the linear system: Ac = d in order to get u(x)

    for i in range(self.N + 1):
        # --- vector d ---
        self.d[i] = self.RHS(i)

        # --- matrix A ---
        self.A[i, i] = self.LHS(i, i)
        if i == 0:
            self.A[i, i + 1] = self.LHS(i, i + 1)
        elif i == self.N:
            self.A[i, i - 1] = self.LHS(i, i - 1)
        else:
            self.A[i, i - 1] = self.LHS(i, i - 1)
            self.A[i, i + 1] = self.LHS(i, i + 1)

    # --- vector c ---
    self.c = np.linalg.solve(self.A, self.d)
```

With the calculated basis functions $\varphi_i(x)$ and constant coefficients $\{\boldsymbol{c}\}$, we obtain the final trial solution $\tilde{u}(x)$:

$$\tilde{u}(x) = \sum_{i=0}^{n} c_i \varphi_i(x)$$

The final trial solution $\tilde{u}(x)$ is calculated as:

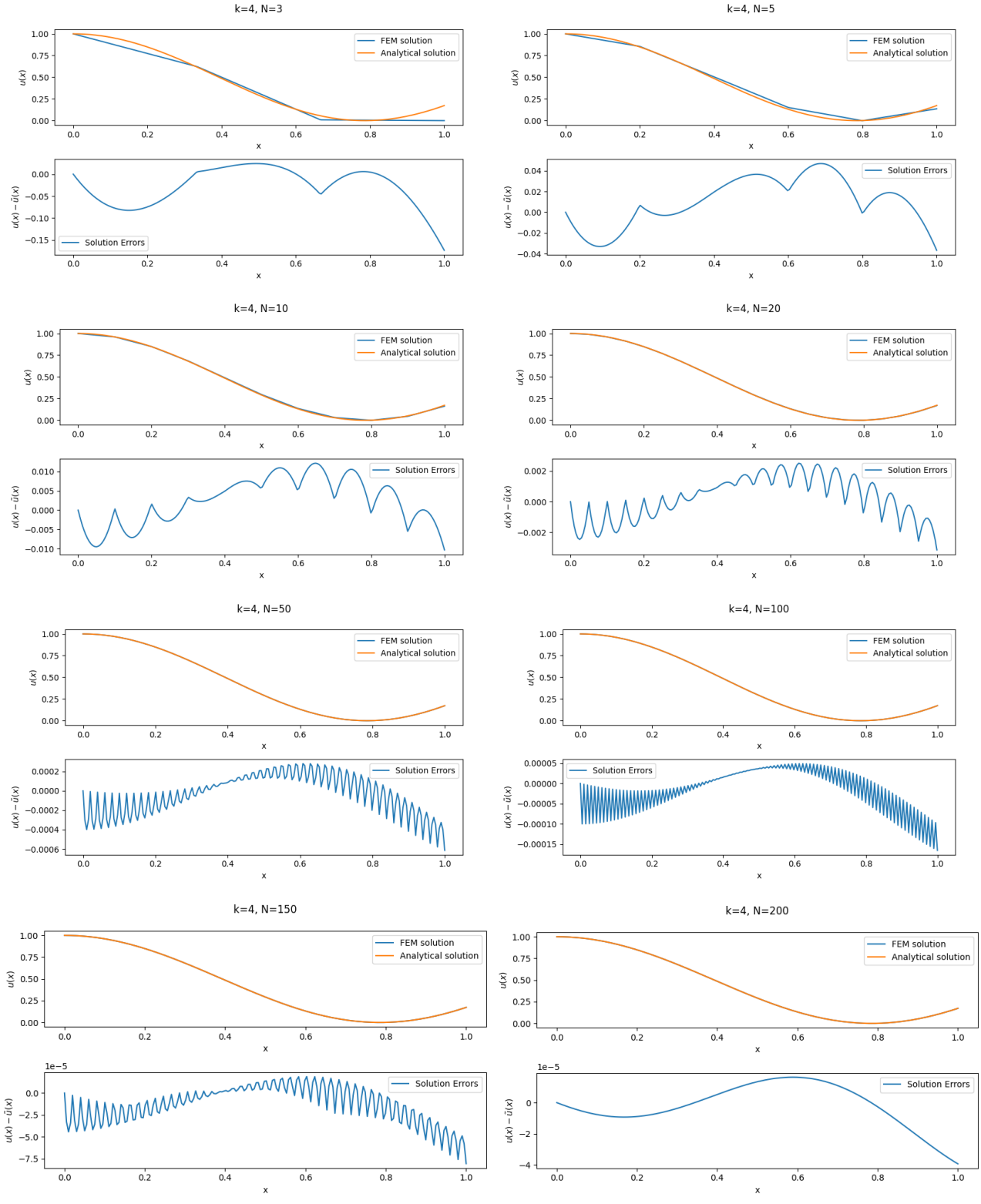
```
# --- solution u(x) ---
self.sol = lambda x: np.sum(np.array(
    [self.c[i] * self.phi(i)(x) for i in range(self.N + 1)]
), axis=0)
```

5.6 Outputs

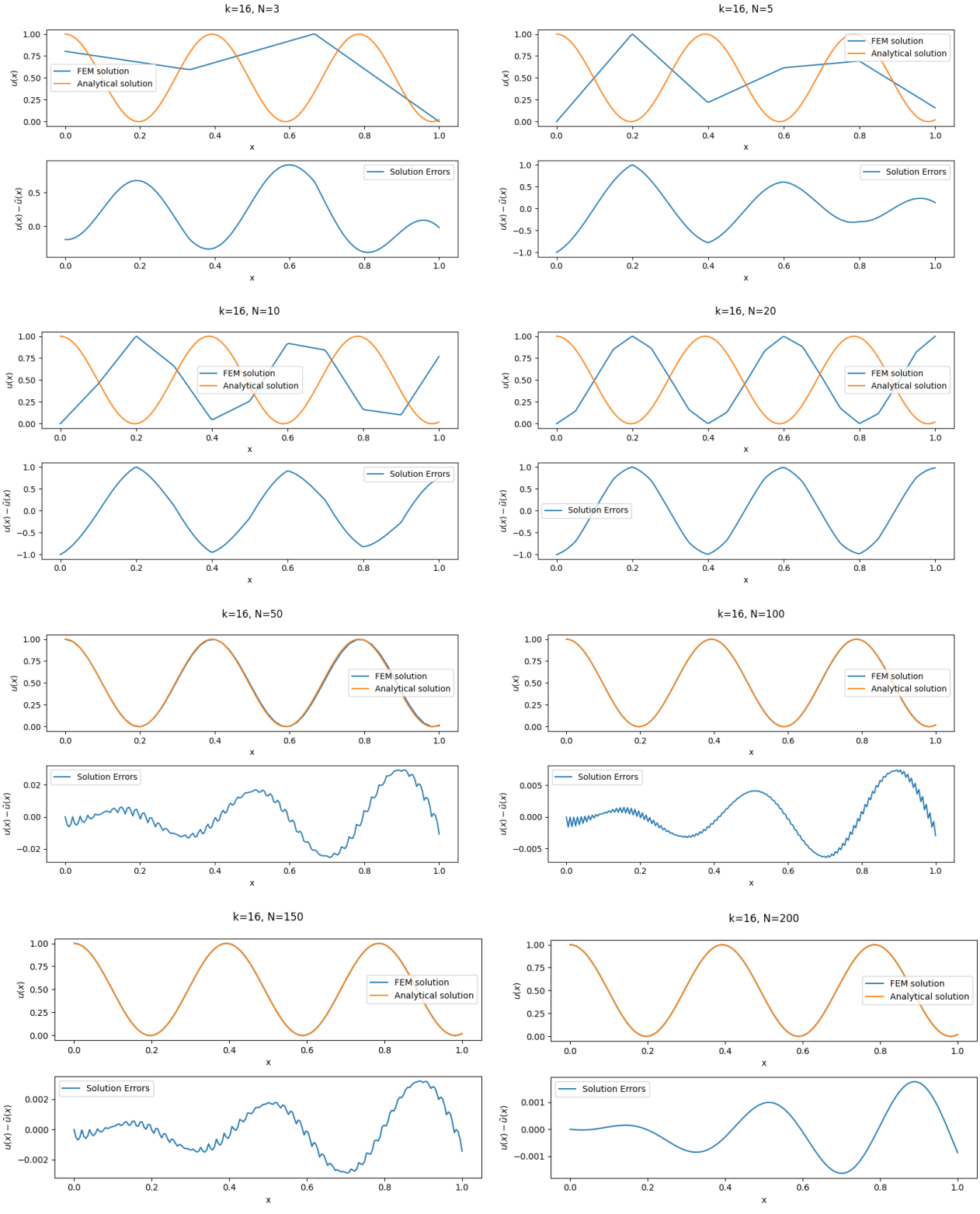
Finally, we will solve the model Helmholtz equation respectively for the cases when k = 4, k = 16, and k = 32. We choose the number of partitions as N = 3, 5, 10, 20, 50, 100, 150, 200.

In the following plots, we show both the FEM solution $\tilde{u}(x)$ in blue compared to the analytical solution $u(x)$ in orange, and also the solution errors calculated from the difference between them: $u(x) - \tilde{u}(x)$.

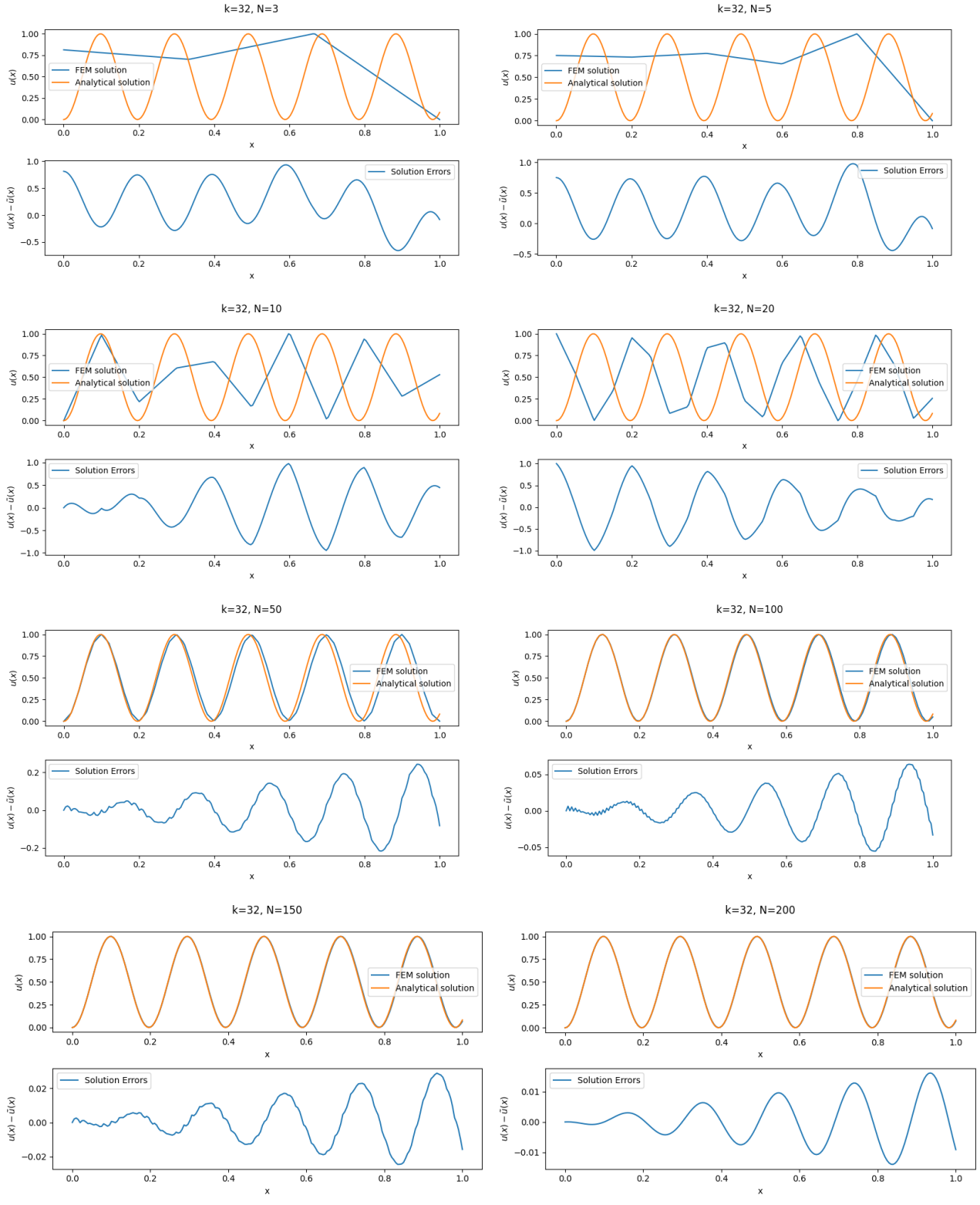
For the case when k = 4:



For the case when k = 16:



For the case when k = 32:



We can observe roughly that the larger the value of k, the more partitions n are needed to obtain an approximately accurate trial solution.

In order to study the performance of the FEM in depth, we will specifically analyse its errors and time consumption.

5.7 Performance Analysis

5.7.1 Errors

We will calculate the mean squared error of the solution as:

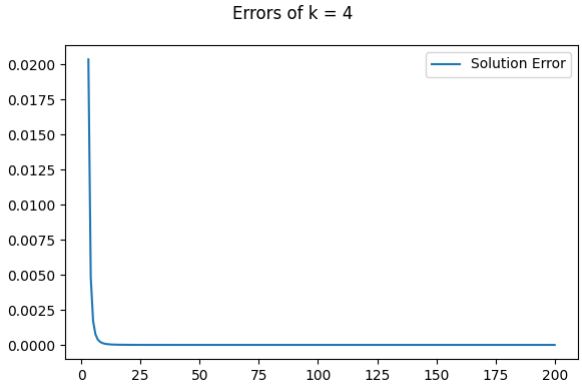
$$\text{Solution Error} = \frac{1}{n}\sum_{i=0}^{n} |\tilde{u}(x_i) - u(x_i)|^2$$

We write the error as a function `error`:

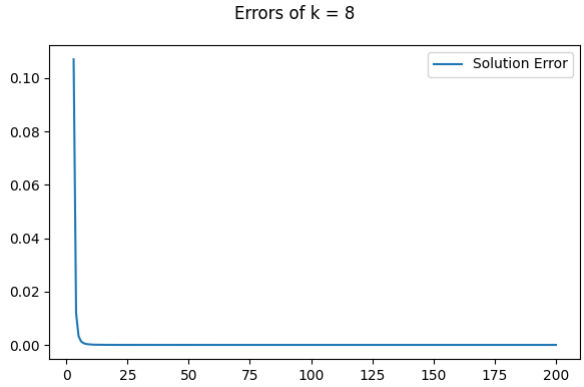
```
def error(self, u: Callable) -> Callable:
    u2  = lambda x: abs(u(x) - self.sol(x)) ** 2
    err_u = sum(u2(self.x[i]) for i in range(self.N + 1)) / self.N
    return err_u
```

Then we can calculate and plot the errors of the cases respectively when k = 4, k = 8, k = 16, and k = 32:

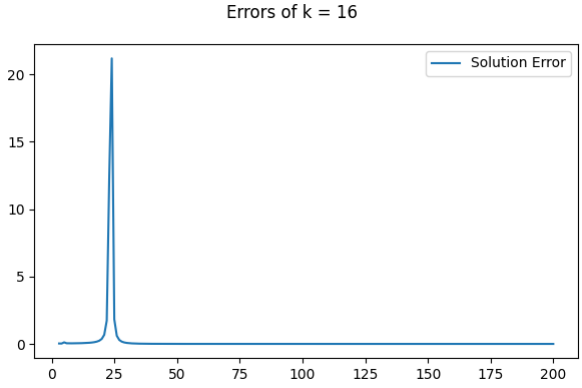
For the case when k = 4:



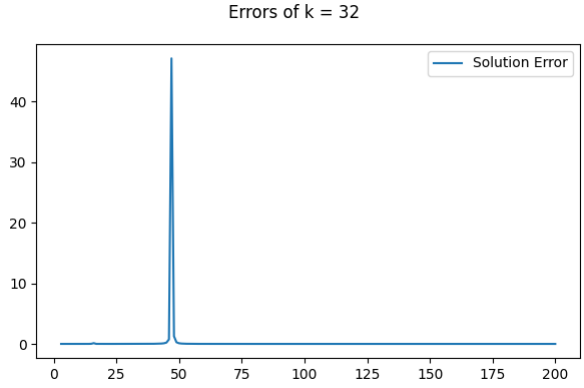
For the case when k = 8:



For the case when k = 16:



For the case when k = 32:



For k = 4, the first eligible number of segmentation of errors less than or equal to 10^{-3} is N = 7.

For k = 8, the first eligible number of segmentation of errors less than or equal to 10^{-3} is N = 7.

For k = 16, the first eligible number of segmentation of errors less than or equal to 10^{-3} is N = 70.

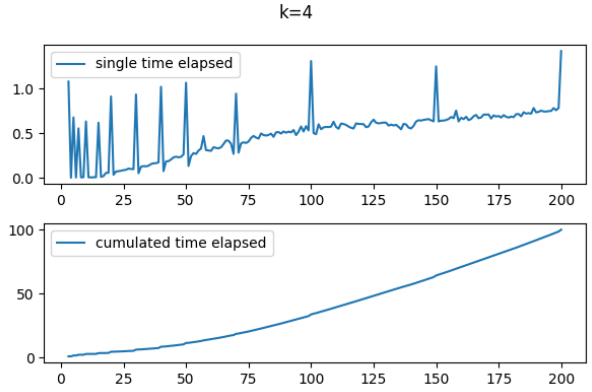
For k = 32, the first eligible number of segmentation of errors less than or equal to 10^{-3} is N = 100.

From the above plots of errors and the minimum number of partitions for increasing values of k, we verified that the higher the value of k, the more partitions are needed to achieve the accuracy.

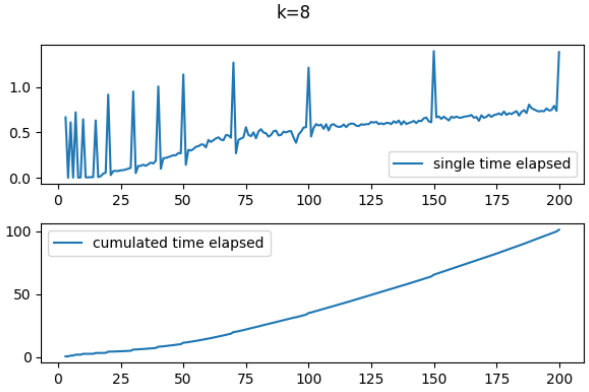
5.7.2 Time Consumption

We will now plot the time elapsed for the cases respectively when k = 4, k = 8, k = 16, and k = 32 as below:

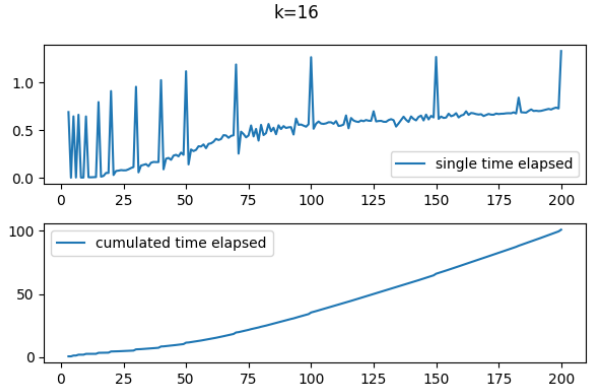
For the case when k = 4:



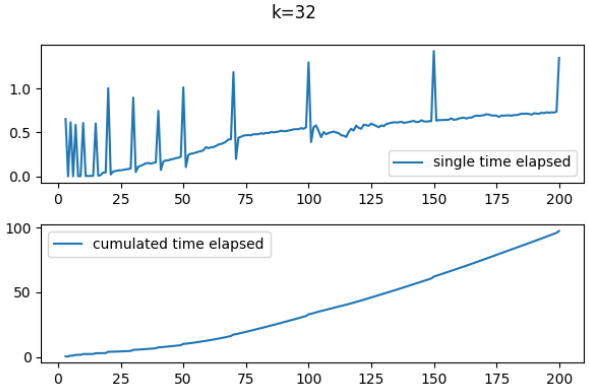
For the case when k = 8:



For the case when k = 16:



For the case when k = 32:



6 Deep Learning Method

6.1 Motivation

The deep learning method is a machine learning method based on artificial neural networks, which is always implemented in self-driving cars and graphic recognition. It is also increasingly being used in engineering and physical models.

Specifically, the deep learning method designed to solve differential equations is known as the Physics-informed neural networks (PINNs). Given the training data and the imposed governing equations, PINNs can be trained to calculate derivatives and solve partial differential equations through their automatic differentiation (AD) function. The problem of solving the equations is therefore transformed into an optimization problem.

We will now see how the PINN method can be implemented in the Helmholtz equation. The main idea of the deep neural network approach is to convert the PDE into an **optimization problem** and train the model by iteratively refining the parameters in order to obtain a solution with minimal losses. Therefore, we will now start by defining the loss function.

6.2 Loss Function

6.2.1 Linear System

As we assumed in the FEM part, the trial solution $\tilde{u}(x)$ is a linear combination of the basis functions $\varphi_i(x)$:

$$\tilde{u}(x) = \sum_{i=0}^{N} c_i \varphi_i(x)$$

We aim at finding the approximated solution $\tilde{u}(x)$ satisfying the linear system:

$$\sum_{i=0}^{n} \int_0^1 (-\varphi_i'(x)\varphi_j'(x) + k^2 \varphi_i(x)\varphi_j(x)) \cdot c_i dx = -\varphi_j(1)$$

There are n basis functions $\varphi_i(x)$ and K test functions $\varphi_j(x)$, where $i = 0, 1, ..., n$, $j = 1, 2, ..., K$.

Then we simplify the linear system as:

$$LHS \cdot \{c_i\}_{i=0}^{N} = RHS$$

Now we will write the LHS and RHS into codes:

LHS: $\int_a^b (-\varphi_i'(x)\varphi_j'(x) + k^2 \varphi_i(x)\varphi_j(x))$, written in codes as:

```
LHS = - self.intg(lambda x: phi_i_x(x) * phi_j_x(x), quadpoints) \
          + self.k.pow(2) * self.intg(lambda x: phi_i(x) * phi_j(x), quadpoints)
```

RHS: $-\varphi_j(1)$, written in codes as:

```
RHS = - phi_j(1)
```

6.2.2 Loss Function

To calculate the trial solution $\tilde{u}(x)$, we need to find the vector $\{c_i\}_{i=0}^{N}$ which minimizes the errors of the solution. We define the residual $\boldsymbol{r}(x) = \{r_k(x)\}_{k=1}^{K}$ as the measure of the errors of the trial solution:

$$\boldsymbol{r}(x) = LHS \cdot \{c_i\}_{i=0}^{N} - RHS$$

Thus, the loss function is defined as the mean squared residual:

$$\begin{aligned} L(\tilde{u}(x)) &= \frac{1}{K}\sum_{k=1}^{K} |r_k(x)|^2 \\ &= \frac{1}{K}\sum_{k=1}^{K} |LHS \cdot \{c_i\}_{i=0}^{n} - RHS|^2 \end{aligned}$$

The main goal of the VPINN method is to minimize the loss, which we calculate in the `loss` function:

```
def loss(self, index, tf_type, quadpoints):
    LHS = - self.intg(lambda x: phi_i_x(x) * phi_j_x(x), quadpoints) \
          + self.k.pow(2) * self.intg(lambda x: phi_i(x) * phi_j(x), quadpoints)
    RHS = - phi_j(1)
    return (LHS - RHS).pow(2)

loss = 0
for index in range(1, K + 1):
    loss += self.loss(index, tf_type, self.quadpoints)
loss /= K
```

6.3 Test functions $\{\varphi_j(x)\}_{j=1}^{K}$

In order to calculate the specific value of the loss, we need to find the test functions $\{\varphi_j(x)\}_{j=1}^{K}$ and the basis functions $\{\varphi_i(x)\}_{j=i}^{n}$.

For the test functions, they should satisfy:

$$u_x(0) = 0, \;\; u_x(1) = 1 \;\; \Rightarrow \;\; \varphi_j'(0) = 0, \;\; \varphi_j'(1) = 1$$

We have a variety of choices for test functions, usually such as Chebyshev polynomials and Jacobian polynomials.

6.3.1 1. Chebyshev Polynomials

The Chebyshev polynomial is denoted as $T_n(x)$, which can be calculated by the built-in function `chebyt` from library `scipy.special`. Denote the order of derivatives by the number d. Thus, the derivatives of the Chebyshev polynomials are shown as:

$$
\begin{cases}
T_n(x) = \frac{n}{2} \sum_{k=0}^{[\frac{n}{2}]} \frac{(n-k-1)!}{k!(n-2k)!} (2x)^{n-2k} & , d = 0 \\
\frac{d^d}{dx^d} T_n(x) = 2^d n \cdot \sum_{0 \le i \le n-d, \quad i \equiv n-d \mod 2} \binom{\frac{n+d-i}{2} - 1}{\frac{n-d-i}{2}} \cdot \frac{(\frac{n+d+i}{2} - 1)!}{(\frac{n-d+i}{2})!} \cdot T_i(x) & \\
\qquad\qquad = 2^d n \cdot \sum_{0 \le i \le n-d, \quad i \equiv n-d \mod 2} \binom{\frac{n+d-i}{2} - 1}{\frac{n-d-i}{2}} \cdot \frac{\Gamma(\frac{n+d+i}{2})}{\Gamma(\frac{n-d+i}{2} + 1)} \cdot T_i(x) & , 1 \le d \le n \\
0 & , d > n
\end{cases}
$$

We then write them in the function `_chebyshev_poly`:

```
def _chebyshev_poly(x: Any, n: int, *, d: int) -> float:
    n = n.item() if isinstance(n, np.ndarray) else n
    if d > n:
        return torch.zeros_like(x)
    elif d == 0:
        res = chebyt(n)(x)
        return res
    else:
        res = 0
        for i in range(n - d + 1):
            if (n - d) % 2 != i % 2:
                continue
            A = binom((n + d - i) / 2 - 1, (n - d - i) / 2)
            B = gamma((n + d + i) / 2)
            C = gamma((n - d + i) / 2 + 1)
            D = chebyshev(x, i, d=0)
            v = A * B / C * D
            if i == 0:
                v *= 1.0 / 2
            res += v
        res *= 2 ** d * n
        return torch.tensor(res)
```

6.3.2 2. Jacobi Polynomials

For the jacobi polynomials, we use the built-in function `jacobi(n, a, b)(x)` from library `scipy.special` to calculate $P_n^{(\alpha,\beta)}(x)$. d stands for the order of derivatives. The jacobi polynomial is written as:

$$
\text{jacobi} = \begin{cases}
P_n^{(\alpha,\beta)}(x) & , d = 0 \\
\frac{d^d}{dx^d} P_n^{(\alpha,\beta)}(x) = \frac{\Gamma(\alpha+\beta+n+1+d)}{2^d \cdot \Gamma(\alpha+\beta+n+a)} P_{n-d}^{(\alpha+d,\beta+d)}(x) & , 1 \le d \le n \\
0 & , d > n
\end{cases}
$$

We then write them in the function `jacobi`:

```
def jacobi(x: Any, n: int, *, d: int, a: int, b: int) -> float:
    if d > n:
        return torch.zeros_like(x)
    elif d == 0:
```

```
        return jacobi(n, a, b)(x)
    else:
        res = (gamma(a + b + n + 1 + d)
               / (2 ** d * gamma(a + b + n + 1))
               * jacobi(n - d, a + d, b + d)(x))
        return torch.tensor(res)
```

Then we calculate the test functions $\varphi_j(x)$ by the `test_functions` function.

```
def test_functions(x, index, d, typetest):
    x = torch.tensor(x)
    if typetest == "chebyshev":
        return chebyshev(x, index+1, d=d) - chebyshev(x, index-1, d=d)
    elif typetest == "legendre":
        return jacobi(x, index+1, a=0, b=0, d=d) - jacobi(x, index-1, a=0, b=0, d=d)
```

Therefore, the test functions $\varphi_j(x)$ and its first-order derivatives are calculated as:

$\varphi_j(x)$: `phi_j = lambda x: test_functions(x, index, 0, tf_type)`

$\varphi'_j(x)$: `phi_j_x = lambda x: test_functions(x, index, 1, tf_type)`

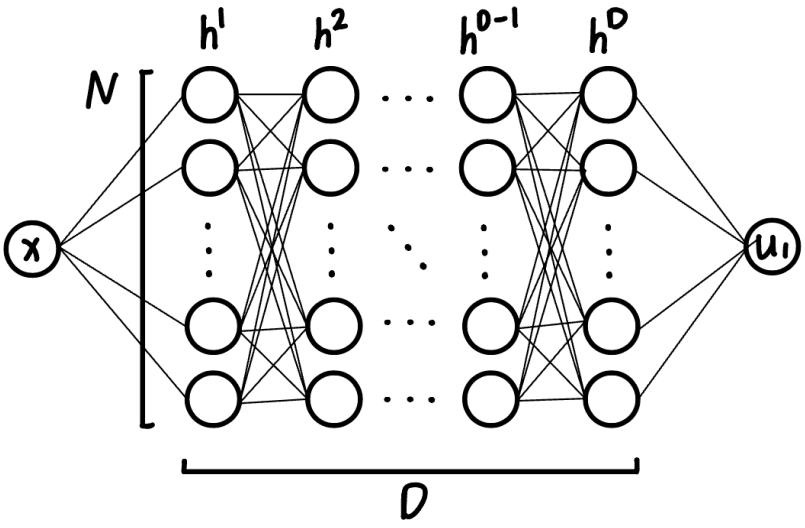
Now that we have the test function $\varphi_j(x)$, we will next find the basis function $\varphi_i(x)$ that arises in the VPINN model.

6.4 Deep Learning Model

6.4.1 Model Architecture

We will build a fully-connected multi-Layer perceptron for Deep Neural Network architecture consisting of:

- an **input** layer with one node x
- D **hidden** layers $h_1, h_2, \ldots, h_D$ with K nodes each layer
- an **output** layer with one node $\tilde{u}$
- N is the length of the layer (i.e. the number of nodes on a single layer), D is the number of layers.



6.4.2 Forward Propagation

1. Input layer

   Insert $x = a$ to the input layer, and perform a linear transformation with weights $\{w_i^1\}_{i=1}^N$ and bias $\{b_i^1\}_{i=1}^N$ to obtain the output $\{z_i\}_{i=1}^N$:

   $$z_i^1 = (weight * input) + bias = (w_i^1 \cdot a) + b_i^1, \quad i = 1, 2, ..., N$$

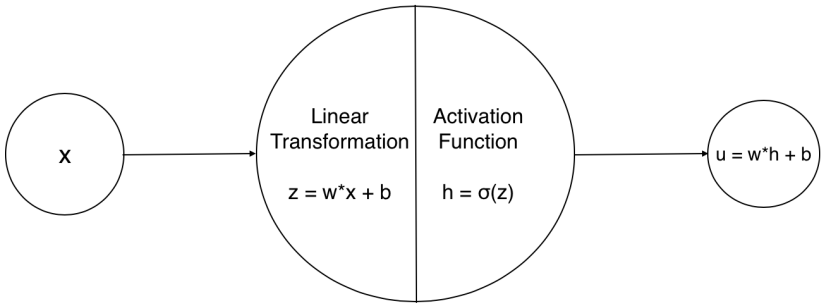
2. First hidden layer

   Input the output of the input layer $\{z_i^1\}_{i=1}^N$ to the first hidden layer h_1. Then we impose a nonlinear function called the activation function $\sigma(\cdot)$ to $\{z_i^1\}_{i=1}^N$. We will discuss the necessary and benefits of the activation function after building the structure.

   $$h_i^1 = \sigma(z_i^1) = \sigma(w_i^1 \cdot a + b_i^1), \quad i = 1, 2, ..., N$$

   h_i^d stands for the output of the i-th node on the d-th hidden layer, where $i = 1, ..., N, d = 2, ..., D$.

   The procedure of the first hidden layer h^1 can be summarized as 1-to-N relationship, performing 2 steps: linear transformation and activation function.



3. Second hidden layer

   The outputs of the first hidden layer h^1 now act as the inputs of the second hidden layer h^2. Thus, what h^2 is different from h^1 is that the number of inputs is currently N instead of 1, other procedures stay the same.

   Step 1. Perform the **linear transformation** on the new inputs $\{h_i^1\}_{i=1}^N$ to obtain $\{z_i^2\}_{i=1}^N$:

   $$z_i^2 = (weight * input) + bias = (w_i^1 \cdot h_i^1) + b_i^1, \quad i = 1, 2, ..., N$$

   Step 2. Impose the **activation function** $\sigma(\cdot)$ to $\{z_i^2\}_{i=1}^N$:

   $$h_i^2 = \sigma(z_i^2) = \sigma(w_i^1 \cdot h_i^1 + b_i^1), \quad i = 1, 2, \ldots, N$$

   The procedure of the second hidden layer h^2 can be summarized as N-to-N^2 relationship, as every node of h_1 is linked to every node of h_2.

4. Other hidden layers

   For any other hidden layers, we repeat the same procedure. The output of hidden layers can then be summarized as:

$$\begin{cases} h_i^1 = \sigma(w_i^1 \cdot x + b_i^1) \\ \\ h_i^d = \sigma(\sum_{i=1}^N w_i^d \cdot h_i^{d-1} + b_i^d) = \sigma(\sum_{i=1}^N w_i^d \cdot \sigma(h^{d-1} \circ h^{d-2} \circ ... \circ h^1) + b_i^d) \quad , \quad i = 1, 2, ..., N, d = 2, ..., D \end{cases}$$

5. Output layer

   At the output layer, we have one node $\tilde{u}$. We will only perform the linear transformation without using the activation function, the output is expressed as:

$$\tilde{u}(x) = \sum_{i=1}^{N} w_i h_i^D + b_i, \quad i = 1, 2, ..., N$$

   The relationship between the last hidden layer h^D and the output layer is N-to-1. In the FEM method, the solution $\tilde{u}(x) = \sum_{i=0}^{N} c_i \varphi_i(x)$ is the weighted sum of the basis functions on N points of the domain: $x_n, n = 1, 2, \ldots, N$.

   Assume that the output is an accurate enough result, so the bias of the output layer is set as zero. Then we obtain the final output of the total architecture:

$$\tilde{u}(x) = \sum_{i=1}^{N} w_i h_i^D = \sum_{i=1}^{N} w_i \cdot \sigma(h^D \circ h^{D-1} \circ ... \circ h^1(x))$$

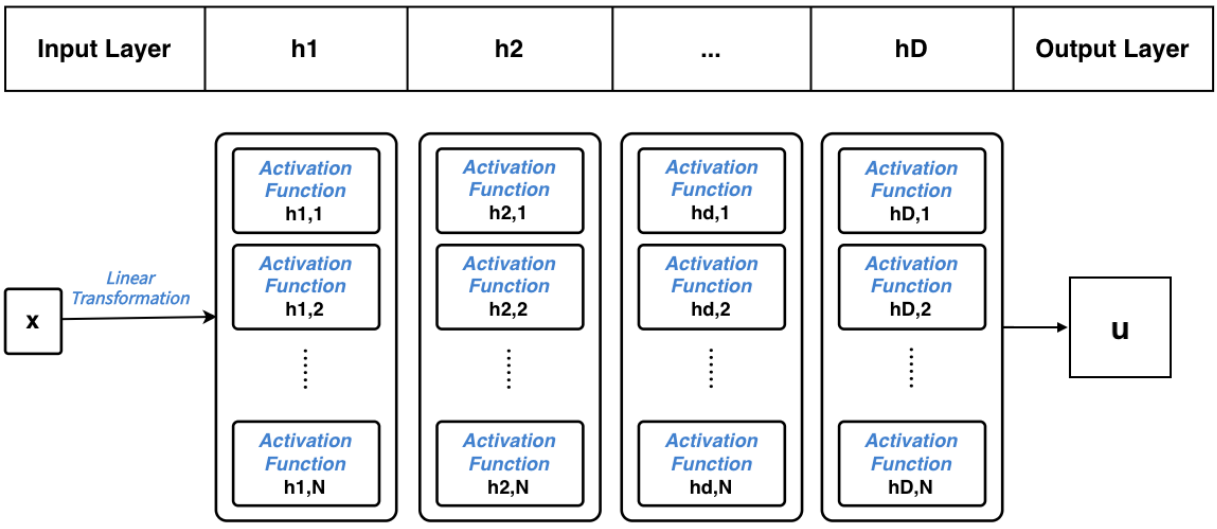
   Equivalently, we also initialize the biases c_0 as zero to obtain: $\tilde{u}(x) = \sum_{i=1}^{N} c_i \cdot \varphi_i(x)$.

   Therefore, we equal the two solutions to obtain the relationship:

$$\tilde{u}(x) = \sum_{i=1}^{N} w_i \cdot \sigma(h^D \circ h^{D-1} \circ ... \circ h^1(x)) = \sum_{i=1}^{N} c_i \cdot \varphi_i(x)$$

$$w_i \sim c_i \quad , \quad \sigma(h^D \circ h^{D-1} \circ ... \circ h^1(x)) \sim \varphi_i(x) \quad , \text{where } i = 1, ..., N$$

In summary, we sketch the process into a flowchart:



6.4.3 Activation function

As the final neural network solution shows, it contains multiple multiplications between the nodes of layers:

$$\tilde{u}(x) = \sum_{i=1}^{N} w_i \cdot \sigma(h^D \circ h^{D-1} \circ ... \circ h^1(x))$$

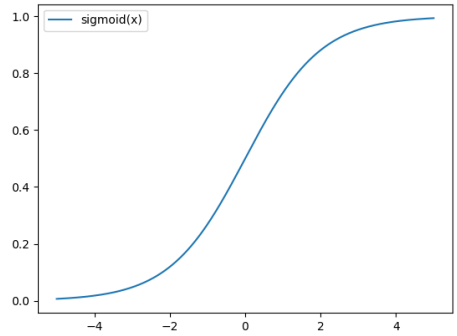
Thus, it would be highly complex to compute directly. However, the noninear activation function helps to simplify the computations. The commonly used activation function are for example, sigmoid and tanh.

1. **sigmoid**

   The sigmoid function is defined as:

   $$y = \frac{1}{1 + e^{-x}}$$

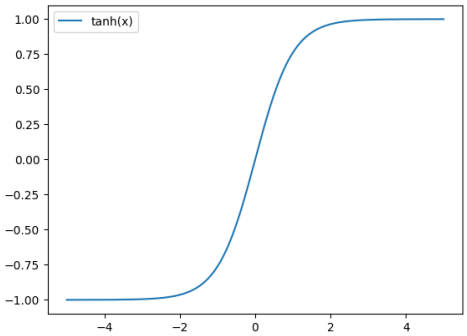
   It restricts the output value in a particular range $y \in (0, 1)$. Thus, the large change in the input x will only cause a small change in the output y, which increases efficiency and accuracy of the model.



   We can calculate sigmoid function by using the built-in function `torch.sigmoid`.

2. **tanh**

   tanh(x) function has a larger range (-1, 1) of the output values than *sigmoid*.



   We can calculate tanh function by using the built-in function `torch.tanh`.

If no non-linear activation function is implemented, we will only perform linear transformations in the model. The final output would then just be a linear transformation of the input x, which restricts the model to a linear equation. Linear associations are not always applicable to all models, thus we introduce a non-linear activation function to improve the accuracy.

Therefore, the nonlinear activation function introduces the non-linearity to improve modelling accuracy and convert high values to small values to improve the efficiency of the calculation.

6.4.4 Forward Propagation Codes

Now we will write the structure of the flowchart into codes:

1. **Layers List**

   Firstly, we show the layers as a list `layers`:

   ```
   layers = [1] + [N for _ in range(D)] + [1]
   ```

   N stands for the number of nodes each hidden layer, and D stands for the number of hidden layers.

   The `layers` list describes that the model consists of 1 input layer with 1 node, D hidden layers with N nodes per layer, and 1 output layer with 1 node.

   For example, if we set that $N = 6, D = 3$, we will gain the `layers` list: `[1, 6, 6, 6, 1]`.

2. **Module List**

   As the flow chart shows, our model propagates forwards step by step. Now we write the input layer and the output layer layers of each step into a total list `self.lins`. Each element of the list contains the input layer and the output layer of each single step in the total forwarding propagation.

   ```
   self.lins = nn.ModuleList()  # Linear blocks
   ```

   `nn.ModuleList()` is a list specifically used in the neural networks. After building the layers, we can now perform the procedure: linear transformation and activation function.

3. **Linear Transformation**

   We define the linear transformation by using the build-in function `nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)`from the torch library. Now we execute it on each element of `self.lins`.

   For the input layer and the hidden layer, the linear transformation is calculated as:

   ```
   # Input layer and Hidden layers
   for input, output in zip(layers[0:-2], layers[1:-1]):
       self.lins.append(nn.Linear(input, output, bias=True))
   ```

   For the output layer, the linear transformation is calculated as:

   ```
   # Output Layer
   self.lins.append(nn.Linear(layers[-2], layers[-1], bias=True))
   ```

4. **Activation Function**

   After imposing the linear transformation, we will execute the activation function. Here we denote the linear transformation as `f(x)` and the activation function as `self.activation`.

In summary, we write the forward propagation as the following combination of the linear transformation `f(x)` and the activation function `self.activation`. Note that we only perform the linear transformation `f(x)` on the output layer.

```
def forward(self, x): # forward propagation
  for i, f in zip(range(self.length), self.lins):
    if i == len(self.lins) - 1:
      # Output layer
      x = f(x)
    else:
      # Hidden layers
      x = self.activation(f(x))
  return x
```

6.5 Basis function $\{\varphi_i(x)\}_{i=1}^N$

The equivalence of the neural network solution and the original solution gives us the basis functions $\varphi_i(x)$:

$$\varphi_i(x) = \sigma(h^D \circ h^{D-1} \circ ... \circ h^1(x)) \quad \text{, where } i = 1, ..., N$$

where $\{h^d\}_{d=1}^D$ is defined as:

$$\begin{cases} h_i^1 = \sigma(w_i^1 \cdot x + b_i^1) \\ h_i^d = \sigma(\sum_{i=1}^N w_i^d \cdot h_i^{d-1} + b_i^d) = \sigma(\sum_{i=1}^N w_i^d \cdot \sigma(h^{d-1} \circ h^{d-2} \circ ... \circ h^1) + b_i^d) \quad , \quad i = 1, 2, ..., N, d = 2, ..., D \end{cases}$$

Thus, the basis functions depends on the input x, the weights w and the bias b. Then we write the basis function and its first order derivative in codes as:

```
self.deriv(0, x)
self.deriv(1, x)
```

6.6 Weights and Biases

6.6.1 Minimization of the Loss Function

With the deduced basis function $\varphi_i(x)$, we insert it into the residual function:

$$\begin{aligned} r(x) &= \sum_{i=0}^{N}(w_i \cdot LHS - RHS) \\ &= \sum_{i=0}^{N} w_i \cdot \int_0^1 (-\varphi_i'(x)\varphi_j'(x) + k^2\varphi_i(x)\varphi_j(x))dx + \varphi_j(1) \\ &= \sum_{i=0}^{N} w_i \cdot \int_0^1 (-\sigma'(h^D \circ h^{D-1} \circ ... \circ h^1(x)) \cdot \varphi_j'(x) + k^2\sigma(h^D \circ h^{D-1} \circ ... \circ h^1(x)) \cdot \varphi_j(x))dx + \varphi_j(1) \end{aligned}$$

The residual depends only on the input x, weights w and bias b, so we write it as $r(x, w, b)$. The loss function, i.e. the mean squared residual, then becomes:

$$\begin{aligned} L(x, w, b) &= \frac{1}{K}\sum_{k=1}^{K} |r_k(x, w, b)|^2 \\ &= \frac{1}{K}\sum_{k=1}^{K} |\sum_{i=0}^{N}(w_i \cdot LHS - RHS)|^2 \\ &= \frac{1}{K}\sum_{k=1}^{K} |\boldsymbol{w^T}\{LHS\} - \{RHS\}|^2 \end{aligned}$$

The main objective of the PINN method is to find the trial solution with minimal loss. Except for the input x, the accuracy of the trial solution depends only on the accuracy of the weights and biases. Therefore, our current goal is to find the weights and biases denoted as w^* and b^* satisfying:

$$w^*, b^* = \arg\min_{w,b}\{\mathrm{L(x,\ w,\ b)}\}$$

$$\text{, where } \ w^* = \{w_i^d\}, b^* = \{b_i^d\}, \ \ i = 0, 1, ..., N, d = 0, 1, ..., D, b_i^D = 0$$

Therefore, in practice, to obtain w^* and b^*, we will first guess the initial values of the weights and biases (initialization) and then improve their accuracy by minimizing the error of the trial solution.

6.6.2 Initialization of weights and biases

We can randomly guess the values of the weights and biases because we always need to improve their accuracy afterwards. However, a good initial guess will greatly help us to save more time in getting closer to the true solution.

The Xavier initialization method, proposed by Glorot, X. & Bengio, Y. (2010), provides initial weights such that the variance is the same for all layers. Specifically, this method makes weights and biases are assumed to be zero-centred and to be independently and identically distributed.

We will now use this method provided by the function `nn.init.xavier_normal_` to initialize the weights, and the function `nn.init.uniform_` to initialize the biases of the layers. Note that the bias of the output layer is assumed to be zero.

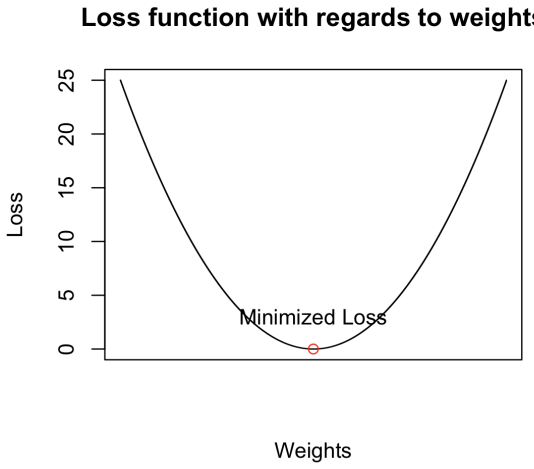
```
# Hidden Layers
for lin in self.lins[:-1]:
    # Weights at hidden layers
    nn.init.xavier_normal_(lin.weight, gain=nn.init.calculate_gain('tanh'))
    # Bias at hidden layers
    nn.init.uniform_(lin.bias, a=int(self.a), b=int(self.b))
# Output Layer
  # Weights at output layer
nn.init.xavier_normal_(self.lins[-1].weight, gain=nn.init.calculate_gain('tanh'))
  # Bias at output layer: assumed as zero
nn.init.zeros_(self.lins[-1].bias)
```

6.6.3 Improvement of weights and biases

With the initial guesses of weights and bias values, we now need to refine them to achieve the higher accuracy. As shown before, the loss function is a convex function with respect to w or b:

$$L(x, w, b) = \frac{1}{K}\sum_{k=1}^{K} |r_k(x, w, b)|^2 = \frac{1}{K}\sum_{k=1}^{K} |w \cdot LHS - RHS|^2$$

We first plot the approximated relationship between the loss $L(x, w, b)$ and the weight w as below:



The red point in the above diagram denotes the minimal loss $L_{\min}$, so the weight at the lowest point is w^*. Thus, the tangent of the curve at the lowest point $(w^*, L_{\min})$ is parallel to the x-axis, and the gradient of the curve becomes zero:

$$\frac{\partial L(x, w, b)}{\partial w} \mid_{w=w^*} = 0$$

Thus, we will find the ideal weights and biases by approaching the gradients $\frac{\partial L(x,w,b)}{\partial w}$ and $\frac{\partial L(x,w,b)}{\partial b}$ to zero.

The gradient of the loss curve to weights is calculated as:

$$\frac{\partial L(x,w,b)}{\partial w} = \frac{2}{K} \cdot LHS \cdot \sum_{k=1}^{K} |w \cdot LHS - RHS|$$

Similar to the bias, the gradient of the loss curve to bias is calculated as:

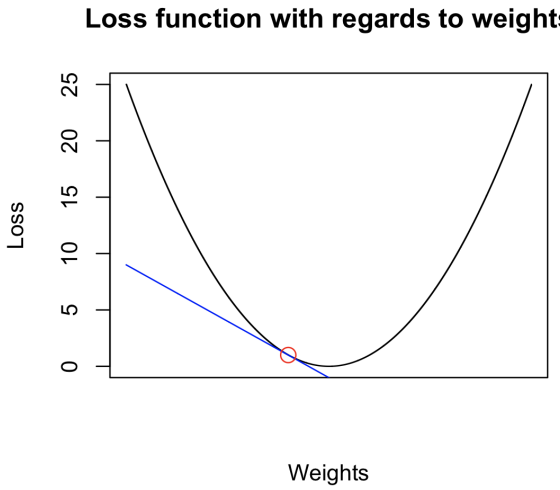
$$\frac{\partial L(x,w,b)}{\partial b} = \frac{2}{K} \cdot \sum_{k=1}^{K} |w \cdot LHS - RHS|$$

6.6.4 Gradient Descent

In order to approximate the initial weights w to the ideal weights w^*, we will first compare their gradients of the loss. The loss gradient can determine whether the current weight w is less than, equal to, or larger than the ideal weight w^*.

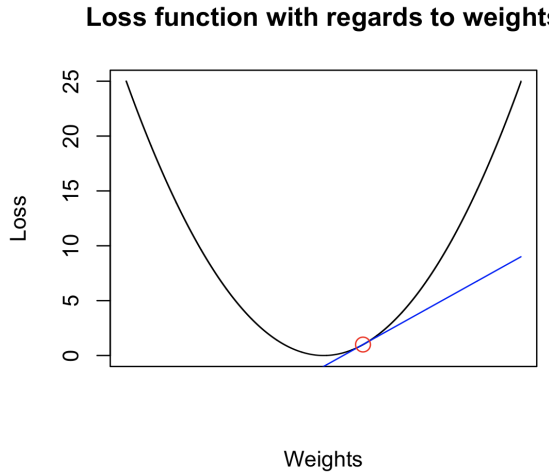
- If $w < w^*$, we have a point (w, l) at the left of the lowest point. The current weight is lower than the ideal weight, and the gradient of the loss curve at the current weight is negative:

$$\frac{\partial L(x,w,b)}{\partial w} |_{w<w^*} < 0$$



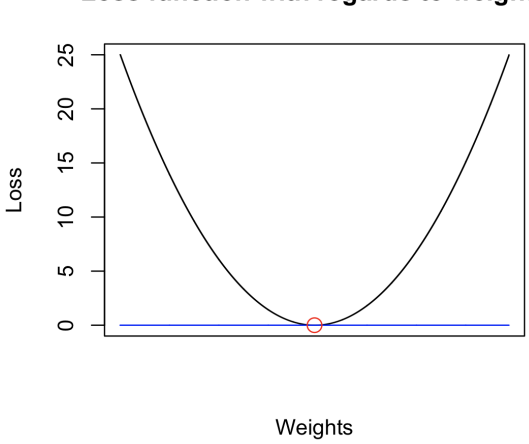
- If $w > w^*$, we have a point (w, l) at the right of the lowest point. The line of the initial w to the ideal w^* is parallel to the x-axis, so the gradient is positive:

$$\frac{\partial L(x,w,b)}{\partial w} |_{w>w^*} > 0$$



- If $w = w^*$, the weight is the ideal weight, and the loss is minimized to $L_{\min}$. The tangent of the loss curve is parallel to the x-axis, so the gradient becomes zero. In this case, the current weight is our ideal weight.

$$\frac{\partial L(x, w, b)}{\partial w} \mid_{w=w^*} = 0$$



Although we do not know the value of ideal weights, the gradients of the loss at the current weights can tell us whether we should increases, decrease, or maintain the current weights. From the previous discussions, we know:

1. The negative gradient $\frac{\partial L(x,w,b)}{\partial w} < 0$ implies $w < w^*$, then we need to increase the current weights.
2. The positive gradient $\frac{\partial L(x,w,b)}{\partial w} > 0$ implies $w > w^*$, then we need to decrease the current weights.
3. The zero gradient $\frac{\partial L(x,w,b)}{\partial w} = 0$ implies $w = w^*$, then we obtain the ideal weights.

Thus, we can write an updating function to approach the current weights w to the ideal weights w^*:

$$w_{updated} = w_{current} - \alpha \cdot \frac{\partial L(x, w, b)}{\partial w} \quad , \alpha > 0$$

The amount of changes of weights is $\alpha \cdot |\frac{\partial L(x,w,b)}{\partial w}|$, the product of the value of the gradient and a changing

size. The changing size α is a small positive constant, called 'learning rate' or 'step size'. It can be any random small values, normally such as 0.10, 0.05, and 0.005.

Therefore, we will train the model by continuously computing the required gradients, until the weights and biases are optimized and the loss is minimized. The number of training is called 'epoch'.

At the [e] epoch, we update the weights and bias to:

$$w^{[e]} = w^{[e-1]} - \alpha \cdot \frac{\partial L(x,w,b)}{\partial w} \mid_{w^{[e-1]}} = \begin{cases} w^{[e]} + \alpha \cdot \left| \frac{\partial L(x,w,b)}{\partial w} \right| & , \text{ if } \frac{\partial L(x,w,b)}{\partial w} \mid_{w^{[e-1]}} < 0 \quad \Rightarrow w^{[e]} < w^{[e-1]} \\ w^{[e]} - \alpha \cdot \left| \frac{\partial L(x,w,b)}{\partial w} \right| & , \text{ if } \frac{\partial L(x,w,b)}{\partial w} \mid_{w^{[e-1]}} > 0 \quad \Rightarrow w^{[e]} > w^{[e-1]} \end{cases}$$

$$b^{[e]} = b^{[e-1]} - \alpha \cdot \frac{\partial L(x,w,b)}{\partial b} \mid_{b=b^{[e-1]}} = \begin{cases} b^{[e]} + \alpha \cdot \left| \frac{\partial L(x,w,b)}{\partial b} \right| & , \text{ if } \frac{\partial L(x,w,b)}{\partial b} \mid_{b^{[e-1]}} < 0 \quad \Rightarrow b^{[e]} < b^{[e-1]} \\ b^{[e]} - \alpha \cdot \left| \frac{\partial L(x,w,b)}{\partial b} \right| & , \text{ if } \frac{\partial L(x,w,b)}{\partial b} \mid_{b^{[e-1]}} > 0 \quad \Rightarrow b^{[e]} > b^{[e-1]} \end{cases}$$

where $w^{[e]}$ and $b^{[e]}$ denote the weight and bias at the [e] epoch.

We repeat this procedure until the weights and biases satisfy the equation:

$$w^{[e]} = w^{[e]} - \alpha \cdot \frac{\partial L(x,w,b)}{\partial w} \mid_{w=w^{[e]}}$$

$$b^{[e]} = b^{[e]} - \alpha \cdot \frac{\partial L(x,w,b)}{\partial b} \mid_{b=b^{[e]}}$$

We will use a gradient descent optimizer called the '**Adam**' optimizer to execute the algorithm. It can determine whether the gradient is positive, negative, or zero, and behaves differently regarding to the result.

Write the weights and biases we need to optimize in a list `optimizer_params`:

```
# optimizing the weights and biases
optimizer_params = []
for lin in model.lins[:-1]:
    optimizer_params.append({'params': lin.weight, 'lr': lr})
    optimizer_params.append({'params': lin.bias,   'lr': lr})
optimizer_params.append({'params': model.lins[-1].weight, 'lr': lr})
optimizer_params.append({'params': model.lins[-1].bias,   'lr': lr})
```

Then, we use the Adam optimizer to execute the gradient descent algorithm:

```
optimizer = torch.optim.Adam(optimizer_params,)
```

We also need to define the size of the learning rate α. As the accuracy of the trial solution is increasing with the times of training increases, we need less changes for the weights and biases when we have trained the model many times. Thus, we can define several stages of training with different sizes of learning rates.

For example:

Stage 1. If $e \in [1, 30)$, then $\alpha = 0.05$;

Stage 2. If $e \in [30, 80)$, then $\alpha = 0.005$;

Stage 3. If $e \in [80, 20000)$, then $\alpha = 0.0005$.

Write the learning rates of each stages in a list `gamma = [0.05, 0.005, 0.0005]`, and the corresponding number of epochs in a list `milestones = [30, 80, 20000]`.

Now we write the optimizer (Adam optimizer executing the gradient descent algorithm), the milestones (number of epochs of different stages), and the gamma (size of learning rates of different stages) into a scheduler:

```
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones, gamma, last_epoch=-1)
```

6.6.5 Backpropagation

With the update functions for the weights and biases, we know how to find the ideal values of all weights and biases for the neural network. Now all we need to do is to calculate the loss gradient of all weights and biases in the neural network.

The forwards propagation shows the flow of calculation from the input x to the output $u(x)$ is:

$$\text{Input } x \quad \Rightarrow \quad z_1 = w_0 \cdot x + b_0 \quad \Rightarrow \quad h_1 = \sigma(z_1) \quad \Rightarrow \quad z_2 = w_1 \cdot h_1 + b_1 \quad \Rightarrow \quad h_2 = \sigma(z_2) \quad \Rightarrow$$

$$\ldots \quad \Rightarrow \quad z_D = w_{D-1} \cdot h_{D-1} + b_{D-1} \quad \Rightarrow \quad h_D = \sigma(z^D) \quad \Rightarrow \quad u = w_D \cdot h_D \quad \Rightarrow \quad r(u) \quad \Rightarrow \quad L(r)$$

In order to calculate the derivative of loss function with regards to all weights and biases in the network, we will reserve the order of the forwarding flow, calculate the derivative from the bottom of the flow to the top.

- Calculate $\frac{\partial L(\tilde{u})}{\partial w_D}$

  $\frac{\partial L}{\partial w_D} = \frac{\partial L}{\partial r(\tilde{u})} \cdot \frac{\partial r(\tilde{u})}{\partial \tilde{u}} \cdot \frac{\partial \tilde{u}}{\partial w_D}$. With $L(\tilde{u}) = \frac{1}{K} \sum k = 1^K |r_k|^2$ and $r = \tilde{u}_{xx} + k^2 \tilde{u}$, we can calculate $\frac{\partial L}{\partial r(\tilde{u})} \cdot \frac{\partial r(\tilde{u})}{\partial \tilde{u}} = \frac{\partial L}{\partial \tilde{u}}$. Also with $\frac{\partial \tilde{u}}{\partial w_D} = h_D$, we have:

$$\Rightarrow \frac{\partial L}{\partial w_D} = \frac{\partial L}{\partial \tilde{u}} \cdot h_D$$

- Calculate $\frac{\partial L}{\partial w_{D-1}}$

  $\frac{\partial L}{\partial w_{D-1}} = \frac{\partial L}{\partial z_D} \cdot \frac{\partial z_D}{\partial w_{D-1}}$. With $\frac{\partial L}{\partial h_D} = \frac{\partial L}{\partial \tilde{u}} \cdot \frac{\partial \tilde{u}}{\partial h_D}$, we have $\frac{\partial L}{\partial z_D} = \frac{\partial L}{\partial h_D} \cdot \frac{\partial h_D}{\partial z_D} = \frac{\partial L}{\partial \tilde{u}} \cdot \frac{\partial \tilde{u}}{\partial h_D} \cdot \frac{\partial h_D}{\partial z_D}$. Also with $\frac{\partial z_D}{\partial w_{D-1}} = h_{D-1}$, we get:

$$\Rightarrow \frac{\partial L}{\partial w_{D-1}} = \frac{\partial L}{\partial \tilde{u}} \cdot (\frac{\partial \tilde{u}}{\partial h_D} \cdot \frac{\partial h_D}{\partial z_D}) \cdot h_{D-1}$$

- ......
- Calculate $\frac{\partial L}{\partial w_1}$

  With $\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \tilde{u}} \cdot (\frac{\partial \tilde{u}}{\partial h_D} \cdot \frac{\partial h_D}{\partial z_D}) \cdot (\frac{\partial z_D}{\partial h_{D-1}} \cdot \frac{\partial h_{D-1}}{\partial z_{D-2}}) \cdot \ldots \cdot (\frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2})$, $\frac{\partial z_2}{\partial w_1} = h_1$, we have:

$$\Rightarrow \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \tilde{u}} \cdot (\frac{\partial \tilde{u}}{\partial h_D} \cdot \frac{\partial h_D}{\partial z_D}) \cdot (\frac{\partial z_D}{\partial h_{D-1}} \cdot \frac{\partial h_{D-1}}{\partial z_{D-2}}) \cdot \ldots \cdot (\frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2}) \cdot h_1$$

- Calculate $\frac{\partial L}{\partial w_0}$

  With $\frac{\partial z_1}{\partial w_0} = x$, we have:

$$\Rightarrow \frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial \tilde{u}} \cdot (\frac{\partial \tilde{u}}{\partial h_D} \cdot \frac{\partial h_D}{\partial z_D}) \cdot (\frac{\partial z_D}{\partial h_{D-1}} \cdot \frac{\partial h_{D-1}}{\partial z_{D-2}}) \cdot ... \cdot (\frac{\partial z_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1}) \cdot x$$

In this way, we are able to calculate the gradient of loss with respect to all weights and the bias from the end of the forwarding flow to the beginning. This algorithm is called "**backpropagation**" and is the reversed order of the forwarding calculations.

Backpropagation transforms the complex calculation of derivatives into a combination of simple calculations. Especially for the deep learning method, this feature greatly improves the efficiency of the computation.

With the backpropagation algorithm of improving the weights and biases, we will now use a built-in technique called '**automatic differentiation**' to improve the efficiency of calculating the gradients. It can automatically compute the gradients for each node of the computational graph, i.e. our deep learning model. The automatic differentiation feature is activated when we require the calcualation of gradients.

For example, if we have a deep learning model in the size $D \times N = 6 \times 4$.

For the variable we don't need to calculate the gradients, such as our input x and the output u, we write them as:

```
x = torch.ones(6)
u = torch.zeros(4)
```

The property `requires_grad = True` requires the calculation of gradients on these parameters. Thus, for the variable we need to calculate the gradients, such as the weights w and the bias b, we write them as:

```
w = torch.randn(6 + 1, 4 + 1, requires_grad = True)
b = torch.rand(4, requires_grad = True)
```

Then we write in the forwarding calculation $h = \sigma(wx + b)$ and the loss function:

```
h = torch.activation(torch.matmul(x, w)+b)
loss = torch.nn.functional.binary_cross_entropy_with_logits(h, y)
```

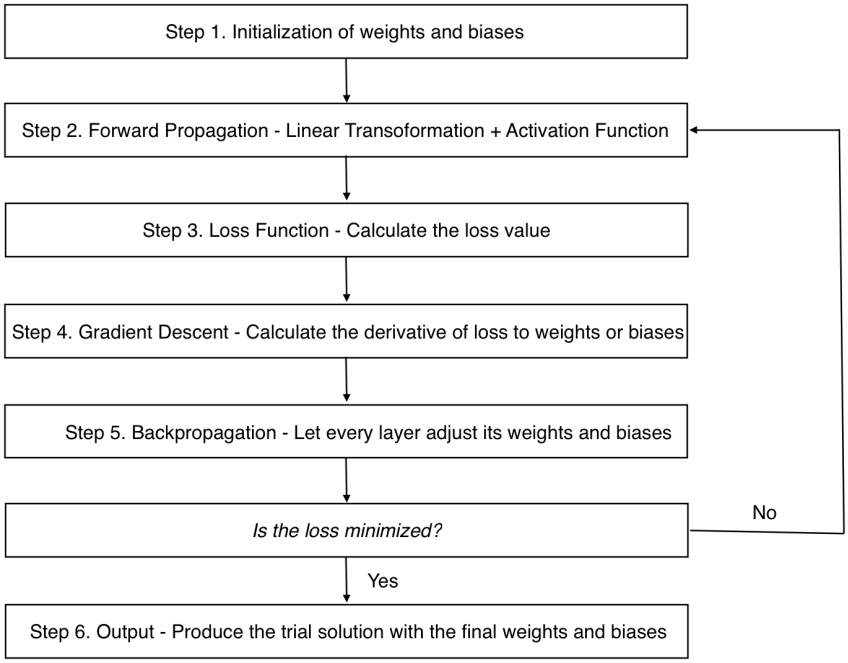
Finally, we can now compare the loss with zero and backpropagate to optimize the weights and biases:

```
loss.backward()
w_result = w.grad
b_result = b.grad
```

Therefore, this is a simple example to show how backpropagation works in the deep learning model.

6.7 Train the model

In summary, we have shown the algorithms and require techniques of finding the trial solution by minimizing the loss. We will train the model many epochs, to continuously optimize the weights and biases to the ideal values. In general, these steps are handled in this way.



We have written the initialization of weights and biases by the `Xavier` initialization, forward propagation by the `forward` function, and the loss function by the `loss` function, gradient descent by the `Adam` optimizer, backpropagation by the `loss.backwards`.

Finally, we define a `train` function to iterate the procedures:

```
def train_(self, tf_type, K, epochs: int, optimizer, scheduler, exact):
    self.train()
    for e in range(self.epoch, epochs + 1):
        time_start = time.process_time()

        # calculate the loss
        loss = 0
        for index in range(1, K + 1):
            loss += self.loss(index, tf_type, self.quadpoints) / K

        # backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # next epoch
        self.epoch += 1
```

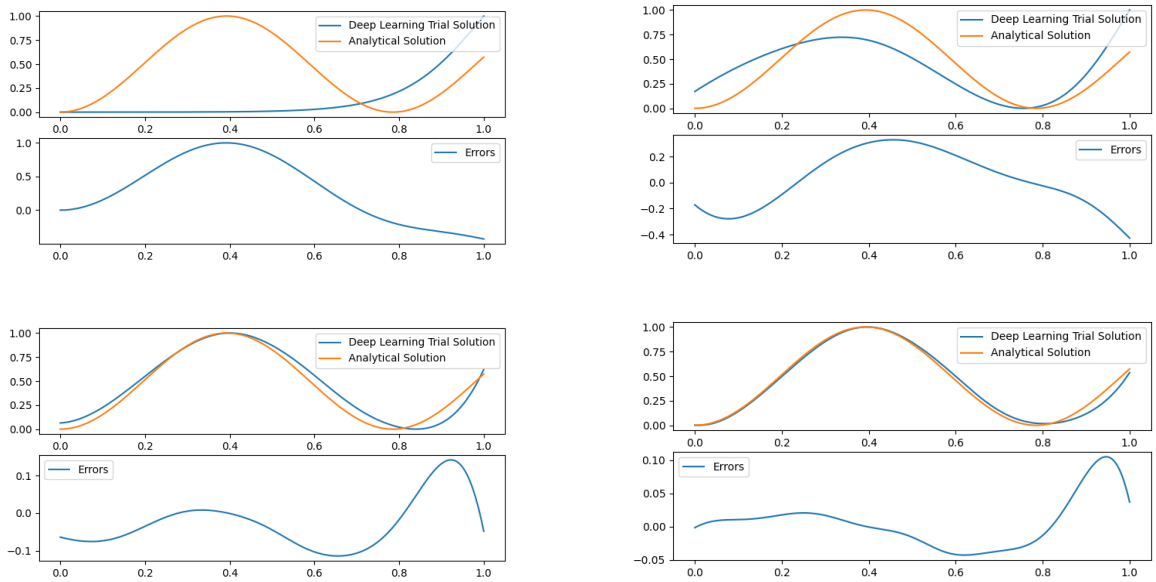
6.8 Outputs

Finally, we can train our model and produce the trial solution with optimized weights and biases. We will then show two examples with different settings at epoch e = 0, 100, 500, 1000, 2000, 5000, 10000, 20000.

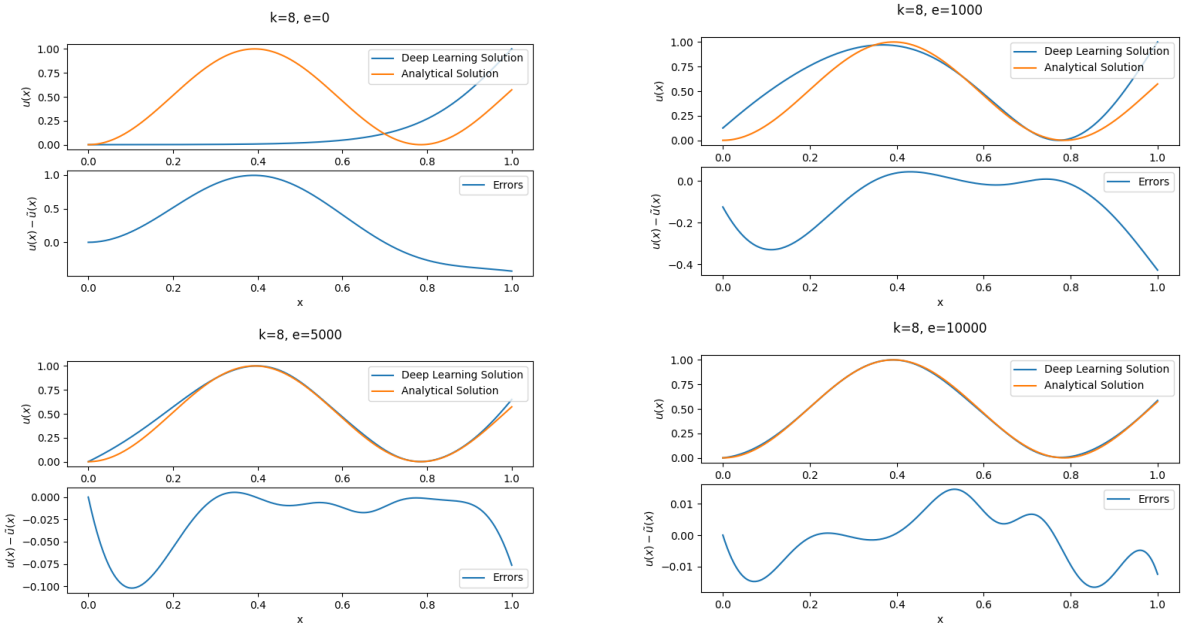
Compare the test functions: Chebyshev polynomial v.s. Legendre polynomial

Other basic settings: activation function: tanh function; size of the neural network model: $D \times N = 1 \times 10$; k = 8.

Case 1: chebyshev polynomial



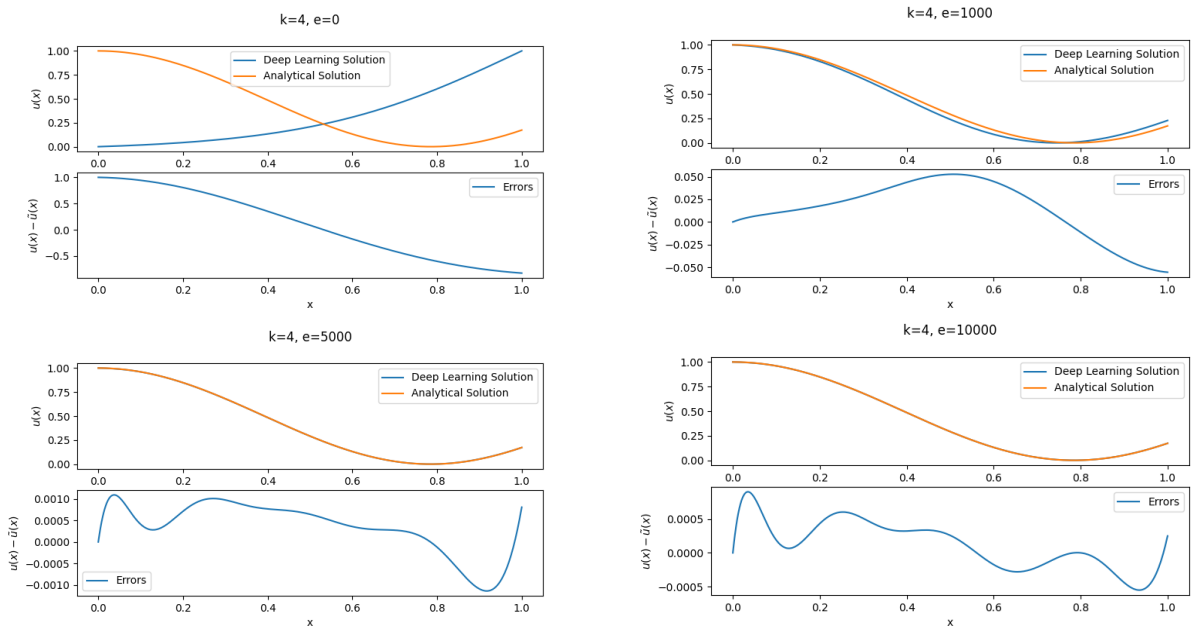
Case 2: legendre polynomial



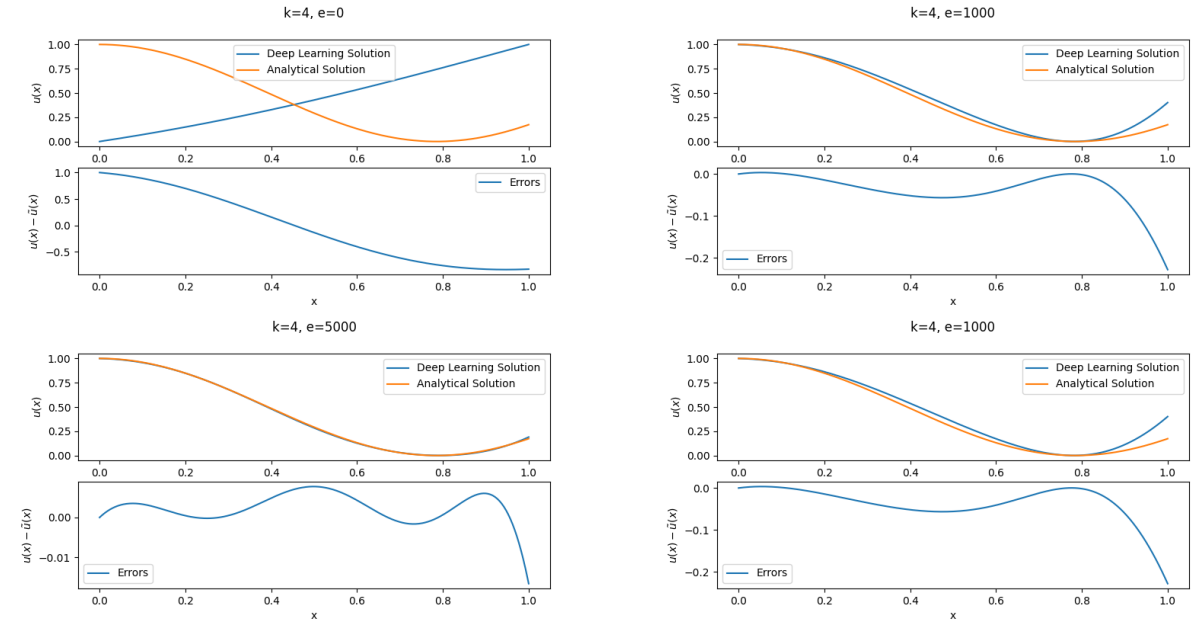
Compare the activation function: tanh vs sigmoid

Other basic settings: test function: chebyshev polynomial; size of the neural network model: $D \times N = 1 \times 10$; k = 4.

Case 1: tanh function



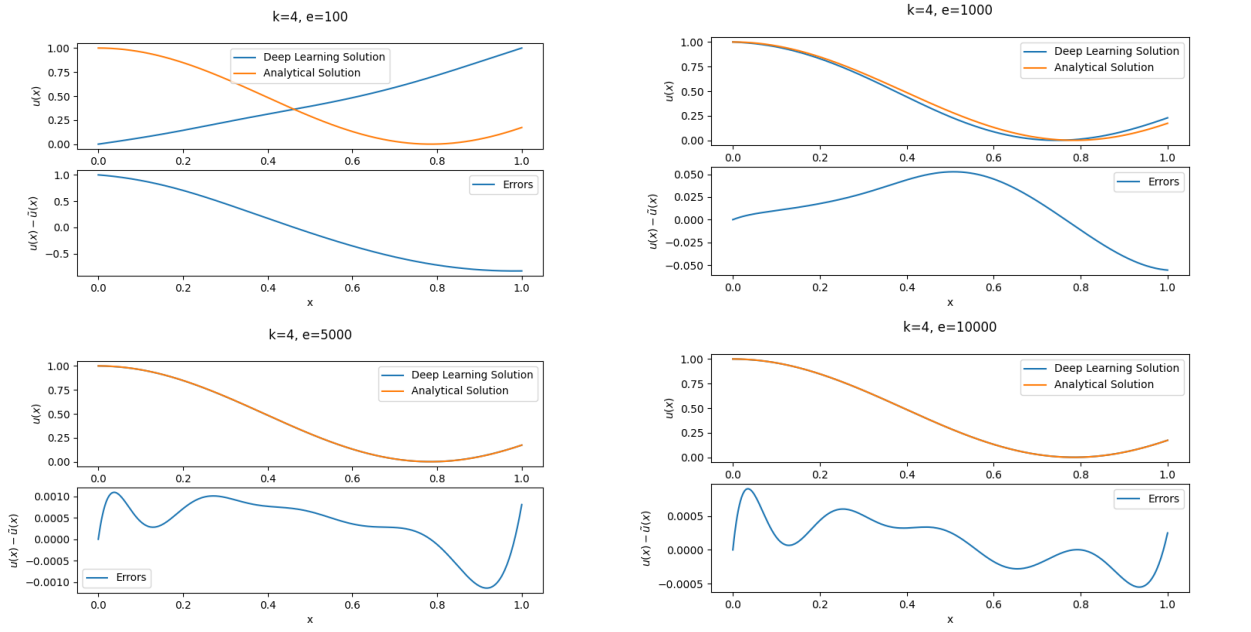
Case 2: sigmoid



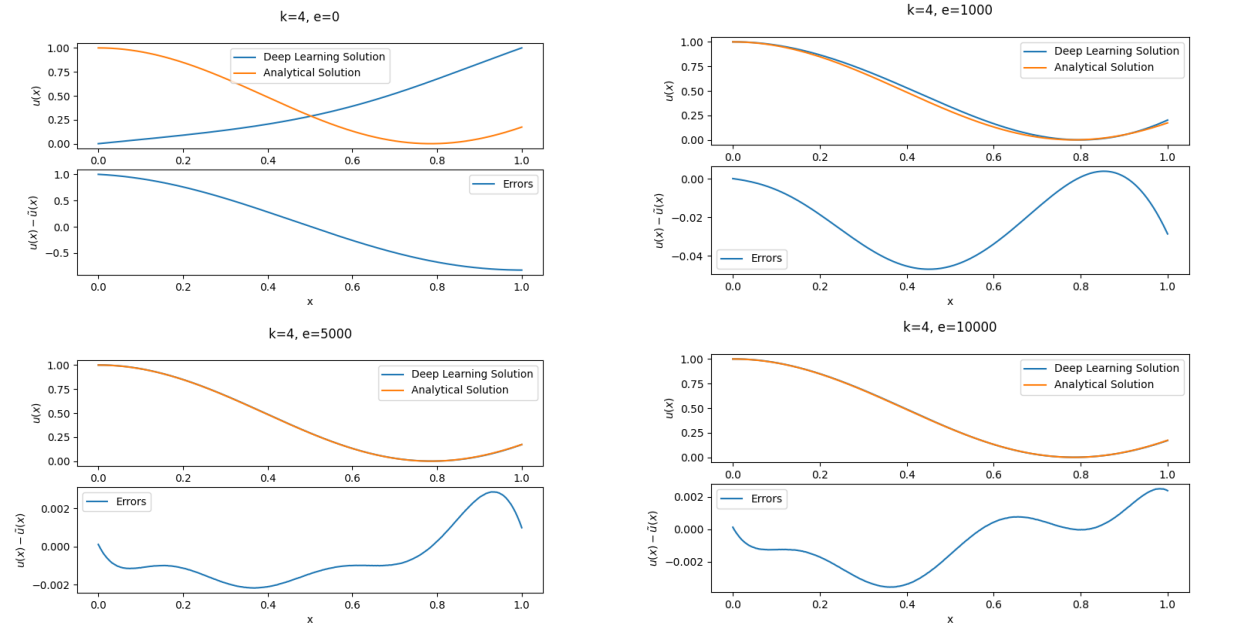
Compare the size of the neural network model

Other basic settings: test function: chebyshev polynomial; activation function: tanh function; k = 4.

Case 1: size of the neural network model: $D \times N = 1 \times 10$



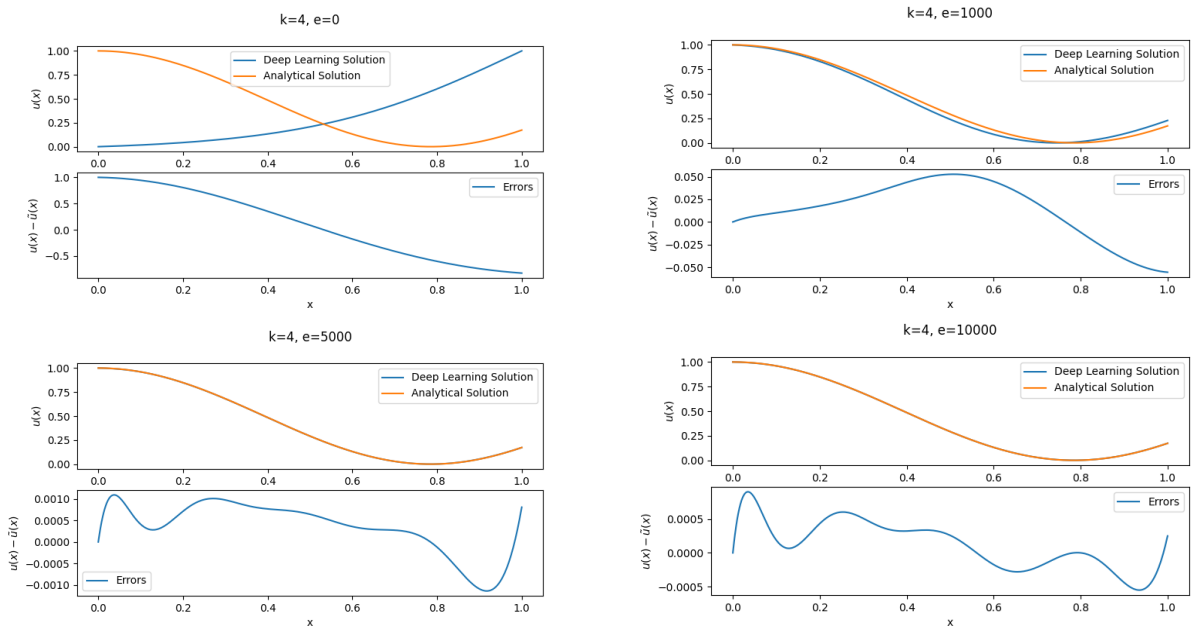
Case 2: size of the neural network model: $D \times N = 6 \times 10$



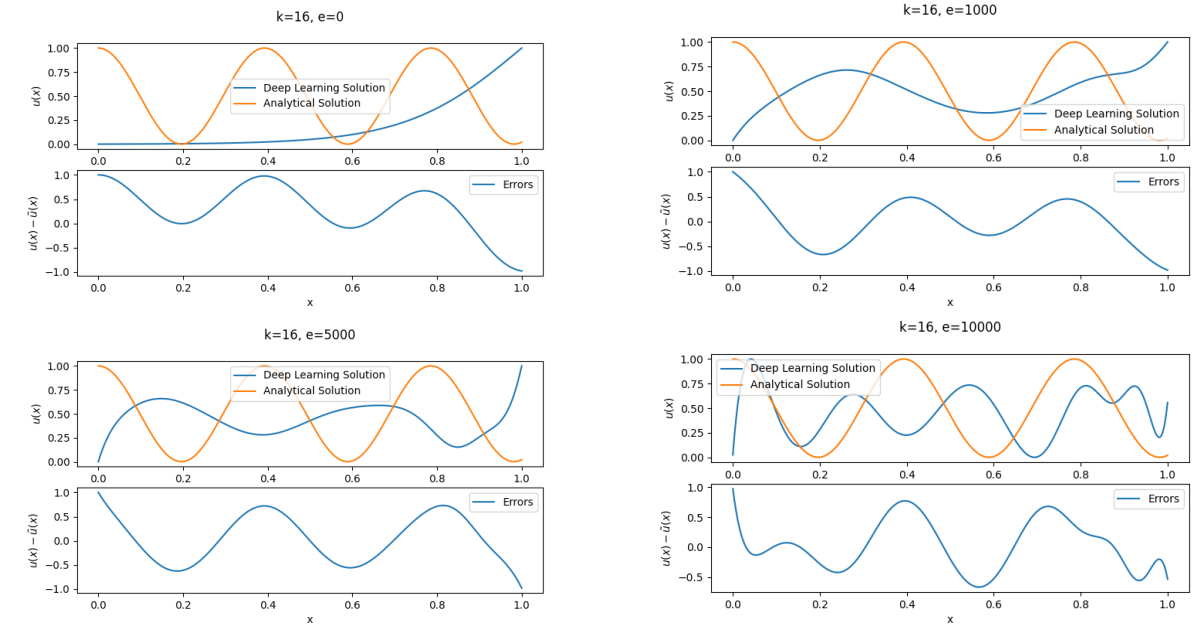
Compare the values of k: k = 8 vs k = 16

Other basic settings: test function: chebyshev polynomial; activation function: tanh function; size of the neural network model: $D \times N = 1 \times 10$.

Case 1: k = 4



Case 2: k = 16

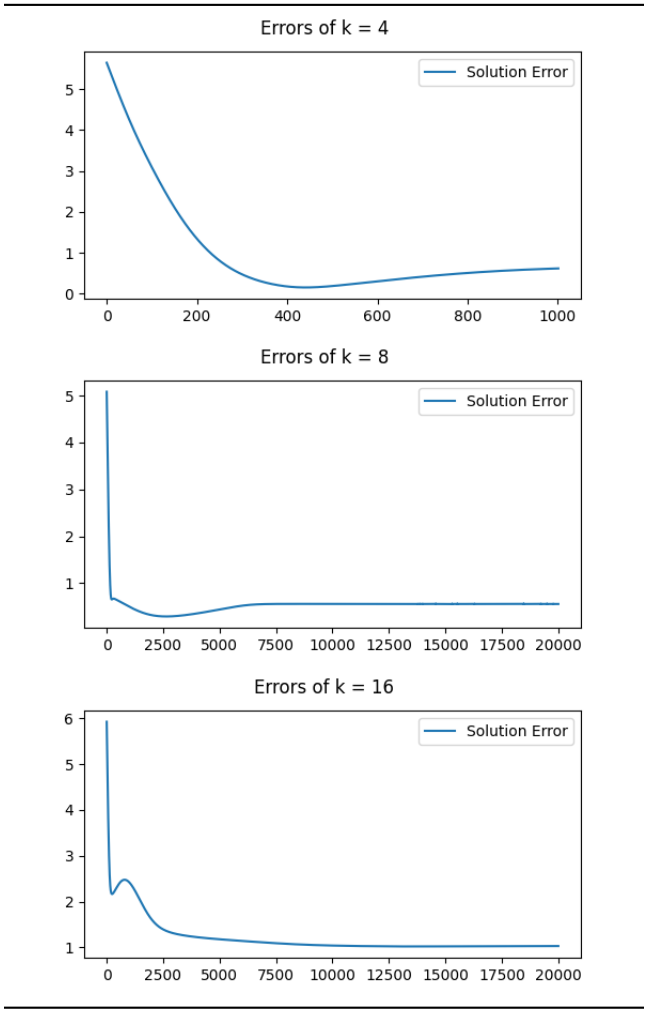


6.9 Performance Analysis

6.9.1 Errors

We will then analyze the errors of the deep learning solution. As in the FEM errors analysis, we use the mean squared residual to calculate the solution errors:

```
def Error(self, phi_i: Callable) -> float:
    def sol(x):
        sol = phi_i(x.detach().view(-1).cpu().numpy())
        return torch.Tensor([sol]).T.to(self.device)
    phi_i2   = lambda x: (sol(x) - self(x)).pow(2).sum(axis=1)
    self.eval()
    err_phi_i   = torch.sqrt(self.intg(phi_i2))
    return err_phi_i
```



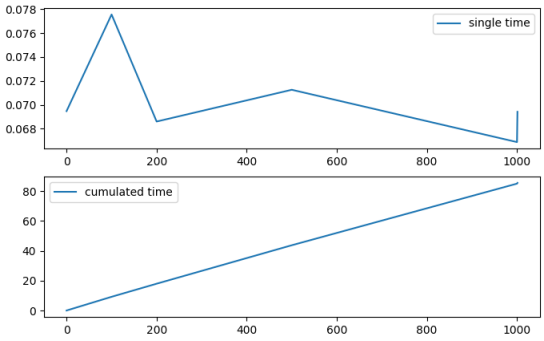
For k = 4, the first eligible number of epoches of errors less than or equal to 10^{-3} is epoches = 87.

For k = 8, the first eligible number of epoches of errors less than or equal to 10^{-3} is epoches = 5195.

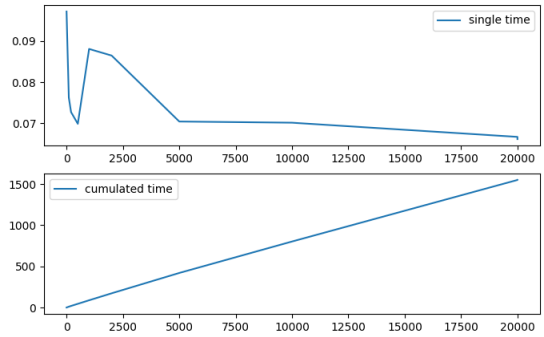
For k = 16, the first eligible number of epoches of errors less than or equal to 10^{-3} is epoches = 7793.

6.9.2 Time Consumption

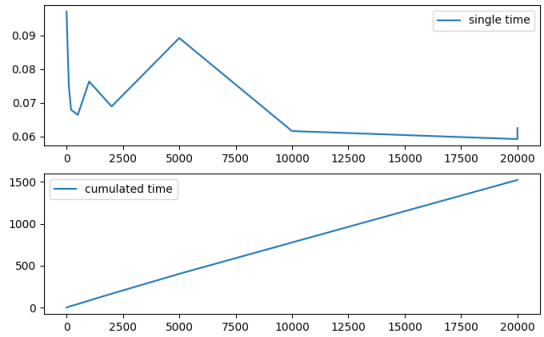
For the case when k = 4



For the case when k = 8



For the case when k = 16



7 Result

Finally, we will compare FDM, FEM and the deep learning method by two criteria: error and time consumption. We will then weigh the accuracy by comparing the error and the practicality by comparing the time consumption to decide on the most appropriate method for our Helmholtz equation model.

7.1 Errors Comparison

- FEM vs FDM

  Firstly, we compare the FDM with FEM by the minimum number of segmentation for different values of k to achieve the error less than 10^{-3}. We list the numbers in the below table:

	The minimum number of segmentation for error $< 10^{-3}$ of FDM	The minimum number of segmentation for error 10^{-3} of FEM
k = 4	30	7
k = 8	50	7
k = 16	100	70
k = 32	>200	100

  We can clearly observe that the FDM requires more parts of the domain to achieve the same accuracy as the FEM. Therefore, based on the accuracy of the solution, FEM is preferable to FDM.

- FEM vs Deep learning method

  When the value of k increases, the error of the FEM is higher. The reason for this is that the elements of the domain are uniformly distributed, the larger k-value implies more oscillations. We need to ensure that there are enough elements for each oscillation to guarantee the accuracy of the solution. Therefore, for the Helmholtz equation with high values of k, we need extremely fine meshes to achieve the desired accuracy, which is costly.

  The error of the deep learning method depends on many factors, such as the test equation, activation equation, learning rate, etc. Here we do not discuss in depth the best settings for the deep learning method with minimal error settings, but only briefly introduced them previously. However, deep learning methods have a large space for optimization and much of the performance is still in the process of optimization.

7.2 Time Consumption Comparison

From the time consumption plots of the FDM, FEM, and the deep learning method, we can compare them in terms of cumulative time consumption or single time consumption.

- The cumulative time consumption

  The cumulative elapsed time increases as the number of partitions (n) / epochs (e) increase. From the time consumption graph, we find that the cumulative elapsed time increases slightly less rapidly for the FDM method than for the FEM method. Therefore, in our model problem scenario, the FDM method takes less time than the FEM method. As for the deep learning method, it costs much more time than FEM in the early stage.

- The single-time consumption

  As the number of partitions (n) / epochs (e) increases, the single consumption time increases for both FDM and FEM, however, the single consumption time for the deep learning method is decreasing. Thus, this validates the benefit of the deep learning approach in that when we train the model to a good degree early on, the model will run faster later on.

8 Conclusion

In this paper, we use the finite element-based method (FEM) to solve the 1D Helmholtz equation. We compare the performance of FEM with other numerical methods: the finite difference method (FDM) and the deep learning method.

For solving the Helmholtz equation with relatively low k-values, FEM achieves a high level of accuracy and efficiency. Compared to FEM, FDM is relatively fast but causes higher errors, while the deep learning method has similar accuracy but runs relatively slowly.

For solving the Helmholtz equation with relatively high k-values, the error of FDM is greater. The deep learning method would be efficient and accurate if the prerequisite model training was accumulated. However, FEM achieves higher accuracy and efficiency if no previous model training has been performed.

In this paper, we focus only on the 1D Helmholtz equation, but we keep the same idea when solving the 2D equation.

9 References

Admin (2022) Helmholtz equation derivation, solution, applications - byju's, BYJUS. BYJU'S. Available at: https://byjus.com/physics/helmholtz-equation/#:~:text=What%20is%20the%20application%20of,the%20temperature%20is%20not%20298K. (Accessed: November 18, 2022).

Admin (2022) Helmholtz equation derivation, solution, applications - byju's, BYJUS. BYJU'S. Available at: https://byjus.com/physics/helmholtz-equation/#:~:text=What%20is%20the%20application%20of,the%20temperature%20is%20not%20298K. (Accessed: November 20, 2022).

Berrone, S., Canuto, C. and Pintore, M. (2022) Solving pdes by variational physics-informed Neural Networks: An A posteriori error analysis, arXiv.org. Available at: https://arxiv.org/abs/2205.00786 (Accessed: November 20, 2022).

Effect of triangular element orientation on finite element solutions of the Helmholtz equation - NASA technical reports server (NTRS) (no date) NASA. NASA. Available at: https://ntrs.nasa.gov/citations/19860022775 (Accessed: November 20, 2022).

Ihlenburg, F. and Babuška, I. (2000) Finite element solution of the Helmholtz equation with High Wave Number Part I: The H-version of the FEM, Computers & Mathematics with Applications. Pergamon. Available at: https://www.sciencedirect.com/science/article/pii/089812219500144N (Accessed: November 20, 2022).

Lewis, P.E. and Ward, J.P. (1991) The finite element method principles and applications. Wokingham: Addison-Wesley.

A posteriori error estimation techniques in practical finite element ... (no date). Available at: http://web.mit.edu/kjb/www/Principal_Publications/A_posteriori_Error_Estimation_Techniques_in_Practical_Finite_Element_Analysis.pdf (Accessed: November 20, 2022).

Stanziola, A. et al. (2021) A Helmholtz equation solver using unsupervised learning: Application to transcranial ultrasound, arXiv.org. Available at: https://arxiv.org/abs/2010.15761 (Accessed: November 20, 2022).

W;, L.L.L.M.M.C.S. (no date) A deep learning approach to estimate stress distribution: A fast and accurate surrogate of finite-element analysis, Journal of the Royal Society, Interface. U.S. National Library of Medicine. Available at: https://pubmed.ncbi.nlm.nih.gov/29367242/ (Accessed: November 20, 2022).

Wang, Z., Cui, T. and Xiang, X. (2020) A neural network with plane wave activation for Helmholtz equation, arXiv.org. Available at: https://arxiv.org/abs/2012.13870 (Accessed: November 20, 2022).