MATH0024 PDEs Homework 1

Julien Brandoit – s200710

Academic year 2024-2025

Excellent work!

check my minor remarks.

Lecture 1

The objective of this exercise is to classify PDEs according to different criteria: their order, and their linearity (linear, quasi-linear, semi-linear, or fully nonlinear).

To begin, we will recall the theoretical concepts necessary for this classification.

Theoretical Reminder

We assume a PDE of the form:

$$f\left(\boldsymbol{x}, \{\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}} u\}_{|\boldsymbol{\alpha}|\leq k}\right) = 0$$

which relates a function u of the variable $\boldsymbol{x} \in \mathbb{R}^m$ and its derivatives up to order k. *k is the order of this PDE* – which corresponds to the order of the highest derivative present in the equation.
Such a PDE is called *linear* if f is a linear function of $\{\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}}\}_{|\boldsymbol{\alpha}|\leq k}$ and can thus be written as:

$$\sum_{\boldsymbol{\alpha}\leq k} a_{\boldsymbol{\alpha}}(\boldsymbol{x})\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}} u = b(\boldsymbol{x})$$

If the equation is not linear but f is a linear function of $\{\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}}\}_{|\boldsymbol{\alpha}|=k}$—i.e., linear with respect to the highest-order term—and can thus be written as:

$$\sum_{\boldsymbol{\alpha}\leq k} a_{\boldsymbol{\alpha}}(\boldsymbol{x}, \{\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}} u\}_{|\boldsymbol{\alpha}|\leq k-1})\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}} u = b(\boldsymbol{x}, \{\partial_{\boldsymbol{x}}^{\boldsymbol{\alpha}} u\}_{|\boldsymbol{\alpha}|\leq k-1})$$

then the PDE can be classified as *quasi-linear*. Some quasi-linear equations can be more specifically referred to as *semi-linear* if the coefficients $a_{|\boldsymbol{\alpha}|=k}$ reduce to a function of $\boldsymbol{x}$ alone and are thus independent of the solution.
Finally, a PDE that does not fit into any of the previous categories (*linear, semi-linear, or quasi-linear*) is considered *fully nonlinear*.

It is then straightforward to classify the following PDEs:

1. Burger's equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{1}$$

   We assume that $\nu \neq 0$ – otherwise, the equation reduces to the inviscid Burger's equation (2), whose classification is done below.

   With a non-zero coefficient, the highest-order derivative term is the term $\nu\frac{\partial^2 u}{\partial x^2}$, which is of order $k = 2$. This PDE cannot be classified as linear because of the term $u\frac{\partial u}{\partial x}$, but it can be characterized as quasi-linear since the highest-order term is indeed linear with respect to $\frac{\partial^2 u}{\partial x^2}$. Furthermore, the coefficient associated with this term is ν, which is not a function of the solution u. This finally allows us to classify this PDE as a *second-order semi-linear PDE*.

2. Inviscid Burger's equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{2}$$

The two terms in this equation are of the same order, which is the highest order, $k = 1$. The term $u\frac{\partial u}{\partial x}$ is nonlinear, so the PDE cannot be classified as linear. The first-order term $\frac{\partial u}{\partial t}$ is fully linear, while the second first-order term $u\frac{\partial u}{\partial x}$ is linear with respect to the highest-order derivative, allowing us to classify this equation as quasi-linear. Since the coefficient u is a function of the solution itself, this PDE cannot be classified as semi-linear. Therefore, equation (2) is a *first-order quasi-linear PDE*.

3. Van der Pol equation:

$$\frac{d^2u}{dt^2} - \mu\left(1 - u^2\right)\frac{du}{dt} + u = 0 \tag{3}$$

The classification of the Van der Pol equation depends on the coefficient μ:

$\underline{\mu = 0}$: The equation (3) reduces to: $\frac{d^2u}{dt^2} + u = 0$, which is a *second-order linear ODE* because the highest derivative is $\frac{d^2u}{dt^2}$ and is of order $k = 2$. All terms are linear.

$\underline{\mu \neq 0}$: The highest-order derivative remains unchanged, so the order of the ODE is still $k = 2$. The term $-\mu(1 - u^2)\frac{du}{dt}$ is nonlinear, making the ODE nonlinear. However, this nonlinear term is of lower order than the highest derivative, meaning the highest-order term $\frac{d^2u}{dt^2}$ is still linear. Therefore, this ODE can be classified as a *second-order semi-linear ODE*.

Lecture 2

In order to show that

$$E(x) = -\frac{i\exp\left(ik|x|\right)}{2k} \tag{4}$$

is indeed a solution of

$$\frac{\partial^2 E}{\partial x^2} + k^2 E = \delta_0, \quad x \in \mathbb{R} \tag{5}$$

it is necessary to demonstrate that E and $\frac{\partial^2 E}{\partial x^2}$ satisfy the relationship described by (5) for all $x \in \mathbb{R}$.

Since $E(x)$ is not differentiable everywhere in the classical sense of differentiation—due to the term $|x|$ which does not have a defined derivative at $x = 0$—it is essential to show that $E(x)$ is a solution in the sense of distributions, and thus with $\frac{\partial^2 E}{\partial x^2}$ being the second derivative in the sense of distributions of $E(x)$.

First, we express $\frac{dE(x)}{dx}$, which is defined by its action on test functions:

$$\left\langle \frac{dE}{dx}, \phi \right\rangle, \quad \forall \phi \in D\left(\mathbb{R}\right)$$

where $D(\Omega)$ denotes the set of smooth functions having closed and bounded support in Ω. By the definition of the adjoint of the differentiation operator, we have:

$$\begin{aligned}
\left\langle \frac{dE}{dx}, \phi \right\rangle &= -\left\langle E, \frac{d\phi}{dx} \right\rangle && (6)\\
&= -\int_{\mathbb{R}} E(x)\frac{d\phi(x)}{dx}dx, \quad E(x) \text{ is locally integrable} && (7)\\
&= -\left(\int_{-\infty}^{0} E(x)\frac{d\phi(x)}{dx}dx + \int_{0}^{\infty} E(x)\frac{d\phi(x)}{dx}dx\right) && (8)\\
&= -\left(\cancel{\left[E(x)\phi(x)\right]_{-\infty}^{0}} - \int_{-\infty}^{0} \frac{dE(x)}{dx}\phi(x)dx \right.\\
&\quad \left. + \cancel{\left[E(x)\phi(x)\right]_{0}^{\infty}} - \int_{0}^{\infty} \frac{dE(x)}{dx}\phi(x)dx\right) && (9)\\
&= \int_{\mathbb{R}} \frac{dE(x)}{dx}\phi(x)dx && (10)
\end{aligned}$$

Thus, the first derivative in the sense of distributions of $E(x)$ coincides with the classical derivative. The expression (8) is obtained using the relation of Chasles applied to integrals, (9) by integration by parts on the two integrals, and (10) by the inverse application of the relation of Chasles. Since $E(x)$ is continuous at $x = 0$ and the test function $\phi(x)$ is smooth and compactly supported, the two integrals over $]-\infty, 0]$ and $[0, \infty[$ can be recombined into a single integral over $]-\infty, \infty[$ without introducing any discontinuities or singularities at $x = 0$. The canceled terms in (9) result from two observations:

1. The evaluation of $E(x)\phi(x)$ as $x \to \pm\infty$ gives 0 since $\phi(x)$ has compact support in $\mathbb{R}$ and the function $E(x)$ is bounded.

2. The terms $E(0)\phi(0)$ and $-E(0)\phi(0)$ cancel each other out.

Finally, the first derivative is defined as:

$$\frac{dE(x)}{dx} = \begin{cases} \frac{d}{dx}\left(-\frac{i\exp(-ikx)}{2k}\right), & x < 0 \\ \frac{d}{dx}\left(-\frac{i\exp(ikx)}{2k}\right), & x > 0 \\ \texttt{undefined}, & x = 0 \end{cases}$$

$$= \begin{cases} \frac{-1}{2}\exp(-ikx), & x < 0 \\ \frac{1}{2}\exp(ikx), & x > 0 \\ \texttt{undefined}, & x = 0 \end{cases} \tag{11}$$

Next, it is possible to express $\frac{d^2E(x)}{dx^2}$, once again by its action on test functions:

$$\left\langle \frac{d^2E}{dx^2}, \phi \right\rangle, \quad \forall \phi \in D(\mathbb{R})$$

By the definition of the adjoint of the differentiation operator, we have:

$$\left\langle \frac{d^2E}{dx^2}, \phi \right\rangle = -\left\langle \frac{dE}{dx}, \frac{d\phi}{dx} \right\rangle \tag{12}$$

$$= -\int_{\mathbb{R}} \frac{dE(x)}{dx}\frac{d\phi(x)}{dx}dx, \quad \left(\text{since } \frac{dE(x)}{dx} \text{ is locally integrable}\right) \tag{13}$$

$$= -\left(\int_{-\infty}^{0} \frac{-1}{2}\exp(-ikx)\frac{d\phi(x)}{dx}dx + \int_{0}^{\infty} \frac{1}{2}\exp(ikx)\frac{d\phi(x)}{dx}dx\right) \tag{14}$$

$$= -\left(\left[-\frac{1}{2}\exp(-ikx)\phi(x)\right]_{\cancel{-\infty}}^{0} - \int_{-\infty}^{0} \frac{ik}{2}\exp(-ikx)\phi(x)dx + \left[\frac{1}{2}\exp(ikx)\phi(x)\right]_{0}^{\cancel{+\infty}} - \int_{0}^{\infty} \frac{ik}{2}\exp(ikx)\phi(x)dx\right) \tag{15}$$

$$= \phi(0) - k^2\int_{\mathbb{R}} E(x)\phi(x)dx \tag{16}$$

$$= \langle \delta_0 - k^2E, \phi \rangle \tag{17}$$

The generalization of the derivative in the sense of distributions allows us to express the second derivative of E everywhere. The relation (14) is obtained by substituting the result (11) and using the property of Chasles. The result (15) is achieved through integration by parts. The evaluated terms at $\pm\infty$ are zero for the same reasons previously explained. The final result (17) is obtained by identifying the definition of the action of a distribution on test functions.

Thus, we can finally write $\frac{d^2E(x)}{dx^2} = \delta_0 - k^2E(x)$ in the sense of distributions.

By injecting these results into (5), we verify that $E(x)$ is indeed a solution in the sense of distributions of the equation:

$$\delta_0 - k^2E + k^2E = \delta_0$$

$$\Longrightarrow \delta_0 \stackrel{!}{=} \delta_0$$

Lecture 3

The generalized Fourier transform for tempered distributions is defined through its action on Schwartz functions. Denoting by $\mathcal{F}$ the Fourier transform operator, it is known that for any tempered distribution T, we have:

$$\langle \mathcal{F}\{T\}, \varphi \rangle = \langle T, \mathcal{F}\{\varphi\} \rangle, \quad \forall \varphi \in \mathcal{S} \tag{18}$$

In this notation, φ is a Schwartz function and $\mathcal{S}$ is the space of Schwartz functions.

Assuming that E_{3D} is indeed a tempered distribution, we can show that $\hat{E}_{3D}$ — where $\hat{f}$ is shorthand for $\mathcal{F}\{f\}$ — is indeed the Fourier transform of E_{3D} by demonstrating that the property (18) holds. By definition:

$$E_{3D}(\boldsymbol{x}) = \frac{-1}{4\pi} \frac{1}{\|\boldsymbol{x}\|} \tag{19}$$

$$\hat{E}_{3D}(\boldsymbol{\xi}) = \frac{-1}{\|\boldsymbol{\xi}\|^2} \tag{20}$$

Starting from the following known result:

$$\int_{\mathbb{R}^3} a^{-\frac{3}{2}} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a}\right) \varphi(\boldsymbol{\xi}) d\boldsymbol{\xi} = \int_{\mathbb{R}^3} \exp\left(-\pi a \|\boldsymbol{x}\|^2\right) \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} \tag{21}$$

This means that in the sense of distributions:

$$\mathcal{F}\left\{\exp\left(-\pi a \|\boldsymbol{x}\|^2\right)\right\}(\boldsymbol{\xi}) = a^{-\frac{3}{2}} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a}\right)$$

We can:

- Multiply by $a^{\frac{-1}{2}}$;
- Integrate with respect to a from 0 to $+\infty$;

The following equivalences hold, after applying Fubini's theorem to exchange the integration symbols and obtain the relation (24).

$$\int_{\mathbb{R}^3} a^{-2} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a}\right) \varphi(\boldsymbol{\xi}) d\boldsymbol{\xi} = \int_{\mathbb{R}^3} a^{-\frac{1}{2}} \exp\left(-\pi a \|\boldsymbol{x}\|^2\right) \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} \tag{22}$$

$$\int_0^\infty \int_{\mathbb{R}^3} a^{-2} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a}\right) \varphi(\boldsymbol{\xi}) d\boldsymbol{\xi} da = \int_0^\infty \int_{\mathbb{R}^3} a^{-\frac{1}{2}} \exp\left(-\pi a \|\boldsymbol{x}\|^2\right) \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} da \tag{23}$$

$$\int_{\mathbb{R}^3} \left(\int_0^\infty a^{-2} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a}\right) da\right) \varphi(\boldsymbol{\xi}) d\boldsymbol{\xi} = \int_{\mathbb{R}^3} \left(\int_0^\infty a^{-\frac{1}{2}} \exp\left(-\pi a \|\boldsymbol{x}\|^2\right) da\right) \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} \tag{24}$$

We can identify in the right-hand side of (24) that the expression $\int_0^\infty a^{-\frac{1}{2}} \exp\left(-\pi a \|\boldsymbol{x}\|^2\right) da$ is equivalent to the expression $\frac{\Gamma(z)}{b^z} = \int_0^\infty t^{z-1} \exp(-bt) dt$ where we identify $b = -\pi \|\boldsymbol{x}\|^2$, where a plays the role of the variable of integration, equivalently to t, and where $z = \frac{1}{2}$. Thus, by replacing and identifying that $\Gamma(\frac{1}{2}) = \sqrt{\pi}$, we finally find for the right-hand side:

$$= \int_{\mathbb{R}^3} \frac{\cancel{\Gamma(\frac{1}{2})}}{\cancel{\sqrt{\pi}}\|\boldsymbol{x}\|} \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} \tag{25}$$

$$= -4\pi \int_{\mathbb{R}^3} E_{3D}(\boldsymbol{x}) \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} \tag{26}$$

One can also focus on the left-hand side of the relation (24):

$$\int_{\mathbb{R}^3} \left(\int_0^\infty a^{-2} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a} \right) da \right) \varphi(\boldsymbol{\xi}) d\boldsymbol{\xi}$$

The following change of variable is introduced to transform the integral:

$$\int_0^\infty a^{-2} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a} \right) da \tag{27}$$

Let introduce λ in (27) such that:

$$\lambda = \frac{\|\boldsymbol{\xi}\|^2}{4\pi a}$$

$$d\lambda = -\frac{\|\boldsymbol{\xi}\|^2}{4\pi a^2} da \iff -4\pi \frac{1}{\|\boldsymbol{\xi}\|^2} d\lambda = a^{-2} da$$

$$a \to 0^+ \iff \lambda \to +\infty$$

$$a \to +\infty \iff \lambda \to 0^+$$

Which leads to:

$$\int_0^\infty a^{-2} \exp\left(-\frac{\|\boldsymbol{\xi}\|^2}{4\pi a} \right) da = \int_\infty^0 -\exp(-\lambda) 4\pi \frac{1}{\|\boldsymbol{\xi}\|^2} d\lambda \tag{28}$$

$$= 4\pi \frac{1}{\|\boldsymbol{\xi}\|^2} \int_0^\infty \exp(-\lambda) d\lambda \tag{29}$$

$$= 4\pi \frac{1}{\|\boldsymbol{\xi}\|^2} \tag{30}$$

$$= -4\pi \hat{E}_{3D}(\boldsymbol{\xi}) \tag{31}$$

The final expression (31), where we have identified $\hat{E}_{3D}(\boldsymbol{\xi})$, can be injected back into the left-hand side of (24). We finally have:

$$\cancel{-4\pi} \int_{\mathbb{R}^3} \hat{E}_{3D}(\boldsymbol{\xi}) \varphi(\boldsymbol{\xi}) d\boldsymbol{\xi} = \cancel{-4\pi} \int_{\mathbb{R}^3} E_{3D}(\boldsymbol{x}) \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} \tag{32}$$

$$\int_{\mathbb{R}^3} \hat{E}_{3D}(\boldsymbol{\xi}) \varphi(\boldsymbol{\xi}) d\boldsymbol{\xi} = \int_{\mathbb{R}^3} E_{3D}(\boldsymbol{x}) \hat{\varphi}(\boldsymbol{x}) d\boldsymbol{x} \tag{33}$$

$$\langle \mathcal{F}\{E_{3D}\}, \varphi \rangle = \langle E_{3D}, \mathcal{F}\{\varphi\} \rangle, \quad \forall \varphi \in \mathcal{S} \tag{34}$$

Thus, by establishing the equivalence of the integrals and satisfying the required properties of the Fourier transform, we conclude that the generalized Fourier transform of the tempered distribution E_{3D} is indeed given by $\hat{E}_{3D}(\boldsymbol{\xi}) = -\frac{1}{\|\boldsymbol{\xi}\|^2}$.

Lecture 4

In order to construct a weak formulation of the following boundary-value problem:

$$\begin{cases} -\dfrac{d^2u}{dx^2}(x) = \alpha(L-x)^2, & x \in]0, L[, \\ u(0) = u_0, & \text{at}\{x = 0\}, \\ \dfrac{du}{dx}(L) = g_L, & \text{at}\{x = L\}, \\ \alpha, L, u_0, g_L \in \mathbb{R} \end{cases} \tag{35}$$

One can introduce test functions $v \in V$ and integrate over the domain $\Omega =]0, L[$. This space function V will be detailed after the mathematical derivation and will be imposed by the existence and the coherence of each step. We can write the following equivalences from the PDE:

$$\int_\Omega -\frac{d^2u}{dx^2}(x)v(x)dx = \int_\Omega \alpha(L-x)^2 v(x)dx \tag{36}$$

$$\int_\Omega \frac{du}{dx}(x)\frac{dv}{dx}(x)dx + \left[-v(x)\frac{du}{dx}(x)\right]_0^L = \int_\Omega \alpha(L-x)^2 v(x)dx \tag{37}$$

$$\int_\Omega \frac{du}{dx}(x)\frac{dv}{dx}(x)dx - v(L)\frac{du}{dx}(L) + \cancel{v(0)\frac{du}{dx}(0)} = \int_\Omega \alpha(L-x)^2 v(x)dx \tag{38}$$

$$\begin{aligned} \int_\Omega \frac{du}{dx}(x)\frac{dv}{dx}(x)dx &= \int_\Omega \alpha(L-x)^2 v(x)dx \\ &\quad + v(L)\frac{du}{dx}(L) \end{aligned} \tag{39}$$

The right-hand side of equation (36) is well-defined provided that v is square-integrable on Ω – thanks to the fact that $\alpha(L-x)^2$ is square-integrable over Ω.

`Condition 1:` $\quad v \in L^2(\Omega)$

Equation (37) is obtained after integration by parts on the left-hand side of (36), and the remaining integral is well-defined provided that $\dfrac{du}{dx}(x)$ and $\dfrac{dv}{dx}(x)$ are both square-integrable.

`Condition 2:` $\quad \dfrac{dv}{dx}, \dfrac{du}{dx} \in L^2(\Omega)$

Since the boundary conditions of the initial problem (35) are respectively a *Dirichlet* condition for $x = 0$ and a *Neumann* condition for $x = L$, one can impose $v(0) = 0$ in order to cancel the $\frac{du}{dx}(0)v(0)$ term in (38). This canceling operation is desired because the problem does not specify the value of the derivative of the solution but only the value of the solution itself. For the second term, however, one can directly apply the Neumann boundary condition that explicitly gives the value of $\frac{du}{dx}(L) = g_L$. These equivalence relations must be satisfied for all functions v that satisfy both Condition 1 and Condition 2 – these two conditions are what define our function space V.

The fact of imposing $v(0) = 0$ is an artifice that allows us to avoid having to handle the Dirichlet condition directly in the expression (39), but it in no way imposes that the

condition be respected. It is therefore necessary to impose it a priori by introducing a third condition.

$$\texttt{Condition 3:} \quad u(0) = u_0$$

This condition is not required for the existence of the various expressions, but is necessary to ensure that the solution constructed from the weak formulation is indeed a solution of the initial problem.

One can finally write a weak formulation of (35) as the following:

Given $\alpha, L, g_L, u_0 \in \mathbb{R}$, find $u \in H^1(]0, L[)$ with $u(0) = u_0$ such that

$$\int_0^L \frac{du}{dx}(x)\frac{dv}{dx}(x)dx = \int_0^L \alpha(L-x)^2 v(x)dx + v(L)g_L$$

$\forall v \in H^1(]0, L[)$ with $v(0) = 0$, where $H^1(I)$ denotes the first-order Sobolev space defined by:

$$H^1(I) = \left\{ f \in L^2(I) : \frac{df}{dx} \in L^2(I) \right\}$$

and where derivatives are 'in the sense of distributions'.

Lecture 5

Analytical solution

One can solve the following boundary value problem (BVP) and find an exact analytical solution:

$$\begin{cases} \dfrac{d^2u}{dx^2}(x) + k^2u(x) = 0, & x \in]0,1[, \\ u(0) = 1, & \text{at } x = 0, \\ \dfrac{du}{dx}(1) - iku(1) = 0, & \text{at } x = 1, \end{cases} \tag{40}$$

The ODE can first solved using the *characteristic polynomial* method. By solving for the roots of $P(z)$, one can find the general form of $u(x)$. We have:

$$P(z) = z^2 + k^2$$

$$\implies u(x) = A\cos(kx) + B\sin(kx), \quad A, B \in \mathbb{C} \tag{41}$$

The constants A and B can be determined by applying the boundary conditions. From $u(0) = 1$, it follows that $A = 1$, and from the Robin boundary condition, it follows that $B = i$. The exact solution of the BVP is therefore given by:

$$u(x) = \cos(kx) + i\sin(kx), \quad x \in [0,1] \tag{42}$$

✓

The solution (42) can be visualized for different wavenumbers in Fig. 1. The wavenumber is directly related to the oscillation frequency of the solution: the larger k, the higher the frequency. We also observe that the real and imaginary parts have the same frequency and are phase-shifted by $\frac{\pi}{2}$ [radian].

The Robin boundary condition is sometimes called an "absorbing" boundary condition because it allows certain waves approaching the boundary to pass without reflection. To see this more clearly, let us consider the solution with a time dependency reintroduced as follows:

$$\begin{aligned} e(x,t) &= u(x)\exp(-i\omega t) \\ &= (\cos(kx) + B\sin(kx))(\cos(\omega t) - i\sin(\omega t)). \end{aligned} \tag{43}$$

When we set $B = i$, the solution simplifies significantly:

$$e_{B=i}(x,t) = \exp(i(kx - \omega t)). \tag{44}$$

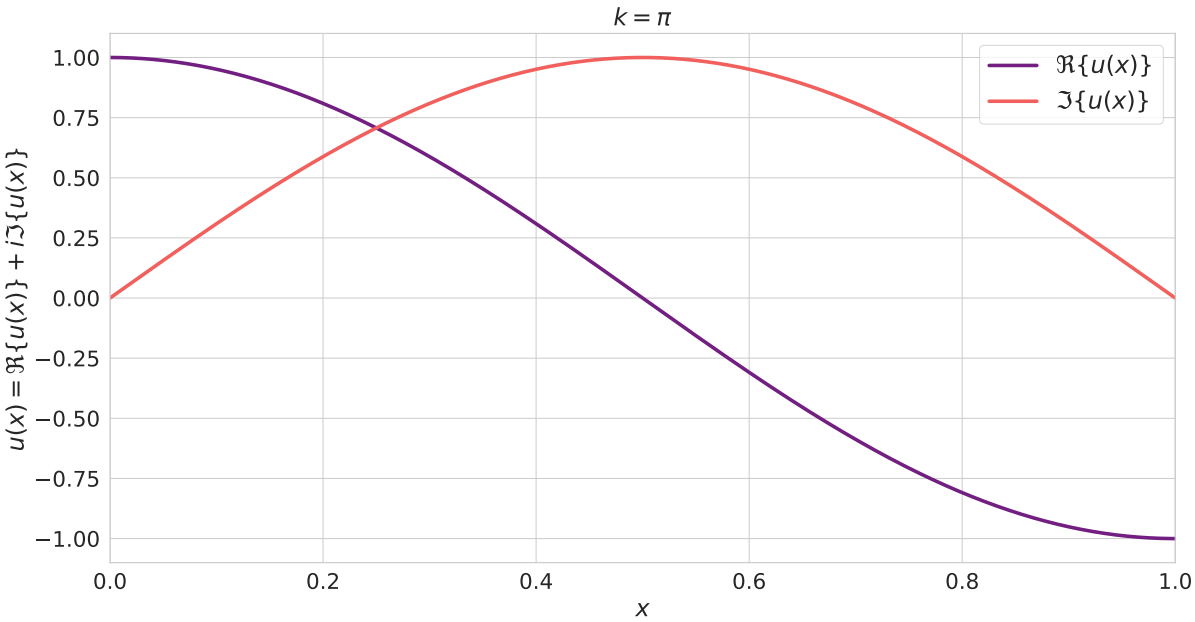
This specific choice of $B = i$ corresponds to a wave traveling in the positive x-direction without reflections at the boundary.

In contrast, when $B = 1$, the solution becomes:

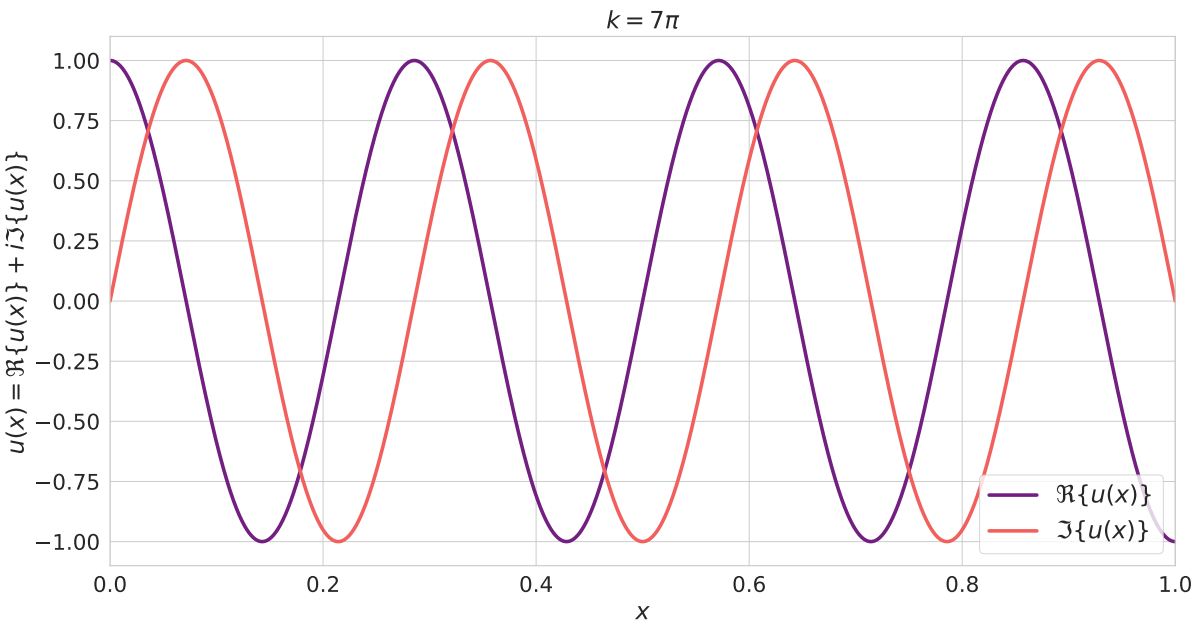
$$e_{B=1}(x,t) = (\cos(kx) + \sin(kx))\exp(-i\omega t). \tag{45}$$

In this case, the boundary does not absorb the wave entirely, and reflections occur. This leads to a more complex spatial pattern due to interference between the incoming and reflected waves.

The time evolution of both cases—(44) and (45)—can be visualized in Fig. 2. It becomes evident that under $B = i$, the wave appears undisturbed by reflections at the $x = 1$ boundary, illustrating why the Robin boundary condition with this specific parameter choice is termed "absorbing".

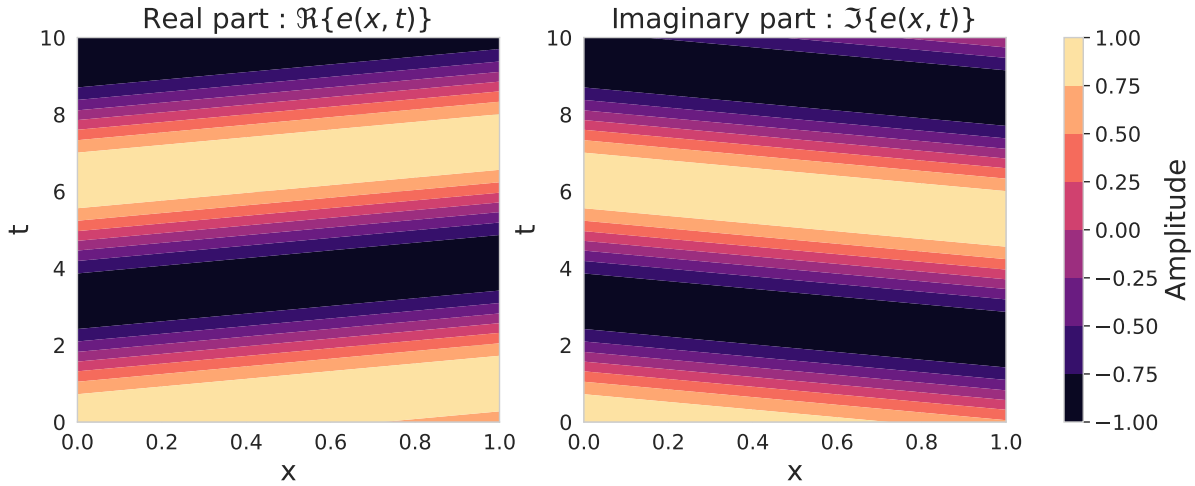


(a)

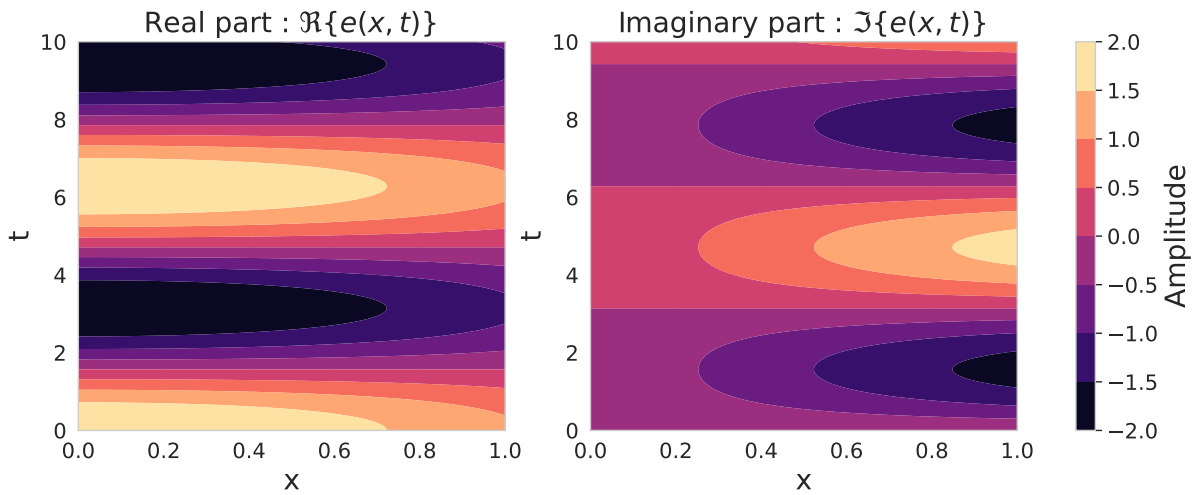


(b)

Figure 1: Visualization of the exact solution $u(x)$ for different wavenumbers k.



(a) $B = i$ - *This corresponds to the Robin boundary condition from problem (40)*



(b) $B = 1$ - *An arbitrary value corresponding to a non-absorbing boundary condition*

Figure 2: Visualization of wave propagation $e(x,t)$ in space-time. *For visualization purposes, arbitrary values of $\omega = 2$ and $k = 7$ are used.*

Finite Element Approximation

To construct an approximation of the solution via FEM, we begin by formulating a weak formulation of the problem. We can write:

$$\begin{aligned}
\int_0^1 \frac{d^2u}{dx^2}(x)v(x)dx + \int_0^1 k^2u(x)v(x)dx &= 0 \\
\left[\frac{du}{dx}(x)v(x)\right]_0^1 - \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x)dx + \int_0^1 k^2u(x)v(x)dx &= 0 \\
\frac{du}{dx}(1)v(1) - \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x)dx + \int_0^1 k^2u(x)v(x)dx &= 0, \quad v(0) = 0 \\
iku(1)v(1) - \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x)dx + \int_0^1 k^2u(x)v(x)dx &= 0
\end{aligned}$$

Thus, the weak formulation of the problem is:

Given $k \in \mathbb{R}$, find $u \in H^1(]0,1[)$ with $u(0) = 1$ such that

$$iku(1)v(1) - \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x)dx + \int_0^1 k^2 u(x)v(x)dx = 0$$

$\forall v \in H^1(]0,1[)$ with $v(0) = 0$.

A Galerkin formulation can be obtained by introducing the basis of functions $\{\varphi_j\}_{j=1}^{\mu_h}$. By imposing the number μ_h of elements with the same size h, we have $h = \frac{1}{\mu_h}$ and restricting to piecewise linear polynomial shape functions:

$$\varphi_j(x) = \begin{cases} \dfrac{x - x_j}{x_j - x_{j-1}}, & x_{j-1} \leq x \leq x_j \\ \dfrac{x_{j+1} - x}{x_{j+1} - x_j}, & x_j \leq x \leq x_{j+1} \\ 0, & \text{otherwise} \end{cases} \implies \frac{d\varphi_j}{dx}(x) = \begin{cases} \dfrac{1}{x_j - x_{j-1}}, & x_{j-1} \leq x \leq x_j \\ \dfrac{-1}{x_{j+1} - x_j}, & x_j \leq x \leq x_{j+1} \\ 0, & \text{otherwise} \end{cases}$$

Under such a basis, the functions $u(x)$ and $v(x)$ are respectively approximated by $u^h(x)$ and $v^h(x)$ with

$$u^h(x) = \sum_{j=1}^{\mu_h} u_j \varphi_j(x), \quad u_1, \ldots, u_{\mu_h} \in \mathbb{C}$$
$$v^h(x) = \sum_{i=1}^{\mu_h} v_i \varphi_i(x), \quad v_1, \ldots, v_{\mu_h} \in \mathbb{C}$$

and the coefficients u_j are the unknowns of the problem. By introducing this Galerkin approximation into our weak formulation, we obtain a Galerkin formulation of our problem where the explicit dependence on the test functions can be simplified since the equivalence relation must hold for all v.

We have:

$$iku(1)v(1) - \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x)dx + \int_0^1 k^2 u(x)v(x)dx = 0 \tag{46}$$

$$\begin{aligned} ik \sum_{i=1}^{\mu_h} \left(\sum_{j=1}^{\mu_h} u_j [\varphi_j \varphi_i]\big|_{x=1} \right) v_i \\ - \sum_{i=1}^{\mu_h} \left(\sum_{j=1}^{\mu_h} \int_0^1 u_j \frac{d\varphi_j}{dx}(x) \frac{d\varphi_i}{dx}(x) dx \right) v_i \\ + k^2 \sum_{i=1}^{\mu_h} \left(\sum_{j=1}^{\mu_h} \int_0^1 u_j \varphi_j(x) \varphi_i(x) dx \right) v_i = 0 \end{aligned} \tag{47}$$

Or equivalently, for all $i \in \{1, \ldots, \mu_h\}$:

$$
\begin{aligned}
& ik \sum_{j=1}^{\mu_h} u_j [\varphi_j \varphi_i]|_{x=1} \\
& - \sum_{j=1}^{\mu_h} \int_0^1 u_j \frac{d\varphi_j}{dx}(x) \frac{d\varphi_i}{dx}(x) dx \\
& + k^2 \sum_{j=1}^{\mu_h} \int_0^1 u_j \varphi_j(x) \varphi_i(x) dx = 0 \qquad (48)
\end{aligned}
$$

$$
\implies [K]\boldsymbol{u} = \mathbf{0} \qquad (49)
$$

The matrix $[K]$ is defined element by element as:

$$
K_{ij} = ik\ [\varphi_j \varphi_i]|_{x=1} - \int_0^1 \frac{d\varphi_j}{dx}(x) \frac{d\varphi_i}{dx}(x) dx + k^2 \int_0^1 \varphi_j(x) \varphi_i(x) dx
$$

The Robin condition is thus translated into a term in the matrix $[K]$. It can also be observed that this term is zero for the majority of nodes during the FEM implementation since the support of the shape functions is restricted to the element and only one element is associated with the node at position $x = 1$.

Finally, to solve the problem using the finite element method, one can:

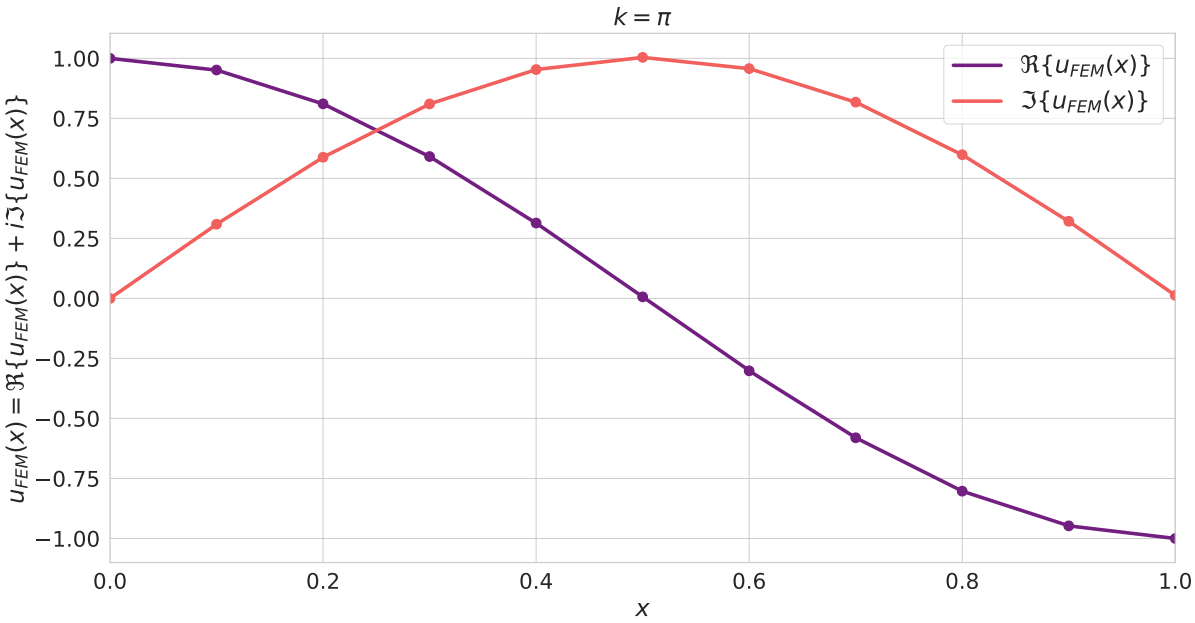
1. Construct the matrix $[K]$ by assembling,
2. Impose the Dirichlet condition $u_0 = 1$,
3. Solve the 'reduced' system $[K_{\neg 0}]\boldsymbol{u} = -\boldsymbol{K}_0 u_0$ - where $[K_{\neg 0}]$ denotes the submatrix obtained from $[K]$ after removing the first row and the first column (associated with u_0) and where $\boldsymbol{K}_0$ correspond to the column that we have remove from $[K]$ excluding the first element (corresponding to u_0). We have:

$$
[K] = \begin{bmatrix} * & * \\ \boldsymbol{K}_0 & [K_{\neg 0}] \end{bmatrix} \qquad (50)
$$

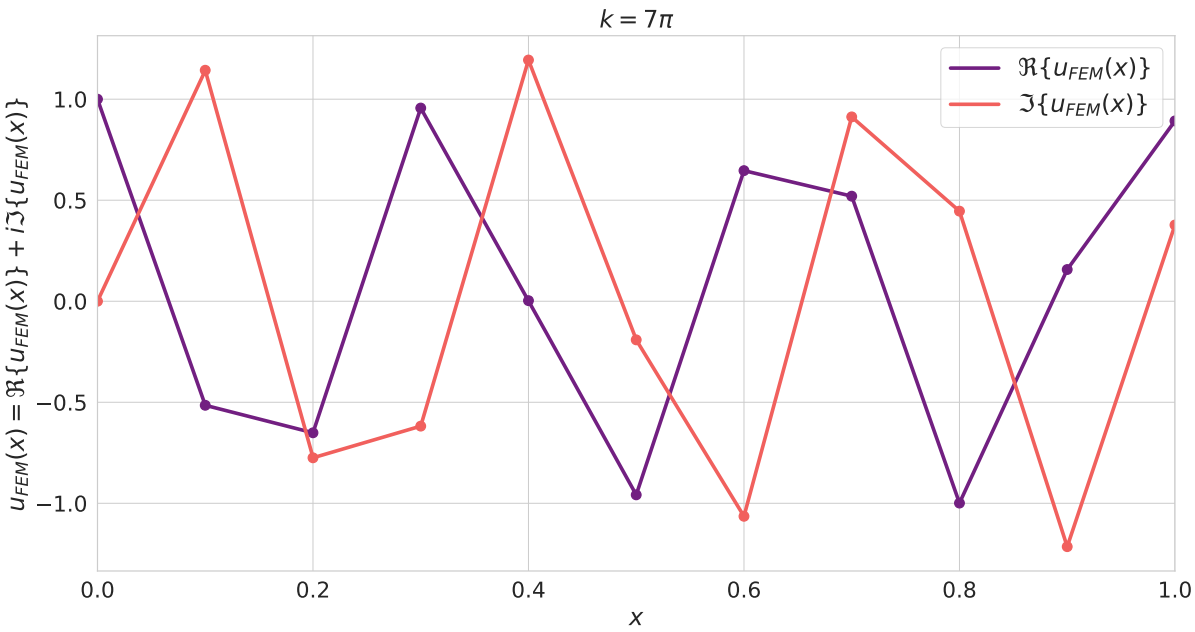
A `Python` implementation is carried out using various numerical tools to *construct*, *store*, *assemble*, and *solve* the problem. We will detail specifically:

- The integration terms are constructed element by element, and Gaussian quadrature integration is used. An order $n = 2$ is chosen for the method, ensuring that the integral is evaluated exactly since the product of the shape functions is of order 2. Indeed, [1] assures exact integration of polynomials up to order $2n-1$ with n points taken over the integration domain.
- The matrix $[K]$ of the system to be solved is constructed using sparse matrix tools from the `Scipy` library [2]. The matrix is assembled and the system is solved with a '`csr`' matrix format using the sparse system solving methods also provided by `Scipy`. This formats is used on the advice of the library.
- The integral term of the first derivatives of the shape functions is pre-calculated by taking advantage of the fact that the derivatives are constants.

The implementation is object-oriented and allows for adapting the homogeneous element size h. Fig. 3 shows the result of the obtained approximations. It can be seen that the number of elements required to achieve a desired precision (relative or absolute) between the true solution and the FEM solution depends on the value of k (one can compare Fig. 1 and Fig. 3). For a larger wavenumber - and thus a higher frequency - a greater spatial resolution is necessary - that is, a finer mesh.



(a)



(b)

Figure 3: Visualization of the approximated FEM solution for different wavenumbers k. *Circles represent the node values,* $h = 0.1$

One can explore the relation between the number of elements used to approximate the solution and the error that is commited by the FEM approximation. Using a L2-norm

measure of the error:

$$e_{L2} = \sqrt{\int_0^1 |u_{\text{exact}}(x) - u_{\text{FEM}}(x)|^2 \, dx} \tag{51}$$

Fig. 4 shows that the error decreases with the number of elements, and that this decrease is similar in rate for the two k values tested. We can see, however, that the initial error is quite different, which explains why we would have needed a much larger number of elements for $k = 7\pi$ if we wanted to obtain a solution of as high quality—in terms of error—as that obtained for $k = \pi$ in Fig. 3. The transient behavior in the error of the $k = 7\pi$ solution can be explained by the higher frequency of the exact solution, leading to the first part of the error curves being influenced by how the nodes are placed over the domain. Once the number of nodes is sufficient to capture the frequency of the solution, we get the constant convergence rate that is similar for both cases.

We also observe that as the number of elements continues to increase on the right side of the graph, the error begins to rise again. This is due to the precision limit of the computer. As we reach very high numbers of elements, the numerical representation of values becomes increasingly sensitive to floating-point precision errors, leading to accumulated numerical imprecision in the solution. Although one could use higher-precision data types (e.g., double or extended precision), this limit will still eventually be encountered. Thus, in this region, the accumulated imprecision from finite precision arithmetic causes the error to increase, preventing further reduction of the error with additional elements.

The error decreases with the number of elements but does not reach 0. Indeed, even without any numerical imprecision, the error would not reach zero due to the inherent limitation of approximating continuous trigonometric functions with piecewise linear elements. In practice, numerical errors further limit the precision achievable as the number of elements increases.

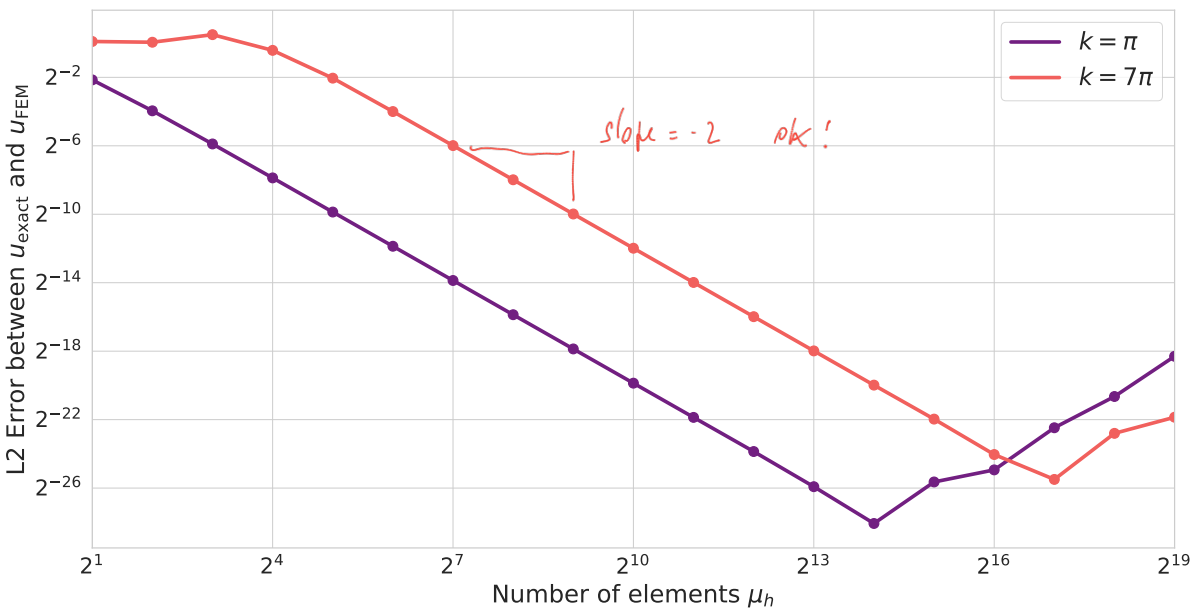


Figure 4: Evolution of the L2 error between the exact solution and the FEM solution as the number of elements increases

References

[1] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*. 3rd. Springer, 2002. ISBN: 978-0-387-95452-3.

[2] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

```python
"""
Author: BRANDOIT Julien

This code implements a finite element method (FEM) solution for a 1D Helmholtz
equation with complex solutions.

The code includes functions for generating exact solutions, assembling
stiffness matrices, and visualizing results.

Function documentation has been provided with the assistance of ChatGPT
to ensure clarity and consistency.
"""

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import fixed_quad
from scipy.sparse.linalg import spsolve
from scipy.sparse import csr_matrix, lil_matrix
from concurrent.futures import ProcessPoolExecutor

# == plotting functions ==

# Set the style of the plots
sns.set(style="whitegrid")
sns.set_context("paper")
sns_color = "magma"
resolution_plot = 2000


def plot_solution(x, u,
                  title=None, save_path=None, x_label=None,
                  y_label=None, marker='o', figsize=(16, 8), show=True,
                  legend=None, x_lim=None, y_lim=None, nodes=None,
                  use_loglog=False):
    """
    Plots the solution(s) over the given domain x.

    Parameters:
    - x : array-like, domain over which the solution is plotted.
    - u : array-like or list of arrays, solutions to be plotted.
    - title : str, optional, title of the plot.
    - save_path : str, optional, file path to save the plot as a PDF.
    - x_label, y_label : str, optional, axis labels.
    - marker : str, marker style for nodes.
    - figsize : tuple, figure size.
    - show : bool, whether to display the plot.
    - legend : list of str, optional, labels for each plotted solution.
    - x_lim, y_lim : tuple, optional, limits for x and y axes.
    - nodes : tuple, optional, nodes to highlight with markers.
    """

    _, ax = plt.subplots(figsize=figsize)

    if type(u) is not list:
        u = [u]

    colors = sns.color_palette(sns_color, len(u))

    for i in range(len(u)):
        if use_loglog:
            ax.plot(x, np.abs(u[i]), color=colors[i],
                    linewidth=3, label=legend[i] if legend else None)
            ax.set_xscale("log", base=2)
            ax.set_yscale("log", base=2)
        else:
            ax.plot(x, u[i], color=colors[i],
                    linewidth=3, label=legend[i] if legend else None)

        # Plot node markers if nodes are provided
```

```python
            if nodes is not None:
                if use_loglog:
                    ax.plot(nodes[0][i], nodes[1][i], marker,
                            color=colors[i], markersize=8, label="_nolegend_")
                    ax.set_xscale("log", base=2)
                    ax.set_yscale("log", base=2)
                else:
                    ax.plot(nodes[0], np.real(nodes[1]) if i == 0
                            else np.imag(nodes[1]), marker,
                            color=colors[i], markersize=8, label="_nolegend_")

        if x_lim is not None:
            ax.set_xlim(x_lim)
        if y_lim is not None:
            ax.set_ylim(y_lim)

        if title is not None:
            ax.set_title(title, fontsize=20)
        if x_label is not None:
            ax.set_xlabel(x_label, fontsize=20)
        if y_label is not None:
            ax.set_ylabel(y_label, fontsize=20)
        if legend is not None:
            ax.legend(legend, loc='best', fontsize=20)

        plt.xticks(fontsize=18)
        plt.yticks(fontsize=18)

        if save_path is not None:
            plt.savefig(save_path, format='pdf', bbox_inches='tight')
        if show:
            plt.show()


def plot_solution_wave_space_time(x, t,
                                  k=1, omega=1, A=1, B=0,
                                  save_path=None):
    """
    Plots the solution of the wave equation in space and time.

    Parameters:
    – x : array–like, spatial domain.
    – t : array–like, temporal domain.
    – k : float, wavenumber.
    – omega : float, angular frequency.
    – A, B : float, coefficients for the solution.
    – save_path : str, optional, file path to save the plot as a PDF.
    """

    X, T = np.meshgrid(x, t)
    Z_real = np.real(solution_wave_space_time(X, T, k, omega, A, B))
    Z_imag = np.imag(solution_wave_space_time(X, T, k, omega, A, B))

    fig, ax = plt.subplots(1, 2, figsize=(15, 5))
    contour_real = ax[0].contourf(X, T, Z_real, cmap=sns_color)
    ax[0].set_title(r"Real part : $\Re\left\{e(x,t)\right\}$",
                    fontsize=20)
    ax[0].set_xlabel("x", fontsize=20)
    ax[0].set_ylabel("t", fontsize=20)
    ax[0].set_xlim(0, 1)
    ax[0].set_ylim(0, 10)
    ax[0].xaxis.set_tick_params(labelsize=15)
    ax[0].yaxis.set_tick_params(labelsize=15)

    contour_imag = ax[1].contourf(X, T, Z_imag, cmap=sns_color)
    ax[1].set_title(r"Imaginary part : $\Im\left\{e(x,t)\right\}$",
                    fontsize=20)
    ax[1].set_xlabel("x", fontsize=20)
    ax[1].set_ylabel("t", fontsize=20)
    ax[1].set_xlim(0, 1)
```

```
        ax[1].set_ylim(0, 10)
        ax[1].xaxis.set_tick_params(labelsize=15)
        ax[1].yaxis.set_tick_params(labelsize=15)

        cbar = fig.colorbar(contour_real, ax=ax, orientation='vertical')
        cbar.set_label("Amplitude", fontsize=20)
        cbar.ax.tick_params(labelsize=15)

        if save_path is not None:
            plt.savefig(save_path, format='pdf', bbox_inches='tight')

        plt.show()

    # == exact_solution ==


    def solution_wave_space_time(x, t, k=1, omega=1, A=1, B=0):
        """
        Computes the solution of the wave equation in space and time.

        Parameters:
        - x : array-like, spatial domain.
        - t : array-like, temporal domain.
        - k : float, wavenumber.
        - omega : float, angular frequency.
        - A, B : float, coefficients for the solution.

        Returns:
        - np.ndarray of complex values representing the solution over x and t.
        """

        return A * np.cos(-omega * t) * np.cos(k * x) +\
            1j * B * np.sin(-omega * t) * np.sin(k * x) + \
            1j * A * np.sin(-omega * t) * np.sin(k * x) + \
            B * np.cos(-omega * t) * np.cos(k * x)


    def exact_solution(x, k):
        """
        Computes the exact solution for a 1D Helmholtz equation.

        Parameters:
        - x : array-like, domain over which to compute the solution.
        - k : float, wavenumber for the Helmholtz equation.

        Returns:
        - np.ndarray of complex values representing the exact solution over x.
        """

        return np.cos(k * x) + 1j * np.sin(k * x)


    def derivative_exact_solution(x, k):
        """
        Computes the derivative of the exact solution for a 1D Helmholtz equation.

        Parameters:
        - x : array-like, domain over which to compute the solution.
        - k : float, wavenumber for the Helmholtz equation.

        Returns:
        - np.ndarray of complex values representing the derivative of the exact
        solution over x.
        """

        return -k * np.sin(k * x) + 1j * k * np.cos(k * x)

    # == FEM implementation ==


```

```python
class Node:
    """
    Represents a 1D node in the FEM mesh with spatial and index attributes.

    Attributes:
    – x : float, position of the node.
    – global_idx : int, global index of the node in the mesh.
    """

    def __init__(self, x, global_idx):
        self.x = x
        self.global_idx = global_idx


class Element1D:
    """
    Represents a 1D finite element for FEM computations.

    Attributes:
    – nodes : list, the two nodes defining the element.
    – h : float, the element length.
    – k : float, wavenumber of the Helmholtz equation.
    """

    def __init__(self, nodes, h, k):
        # the local idx of the nodes is the idx of the node in the list
        self.nodes = nodes
        self.h = h
        self.k = k

    def phi(self, x):
        """
        Piece–wise linear polynomial shape functions for the element
        evaluated at x.

        Parameters:
        – x : float or array–like, points where shape functions are evaluated.

        Returns:
        – np.ndarray, shape function values.
        """

        x = np.atleast_1d(x)

        # Create a mask to exclude points outside the element
        mask = (x >= self.nodes[0].x) & (x <= self.nodes[1].x)

        v = np.zeros((2, *x.shape), dtype=complex)

        v[0, mask] = 1 / self.h * (self.nodes[1].x - x[mask])
        v[1, mask] = 1 / self.h * (x[mask] - self.nodes[0].x)

        return v if x.size > 1 else v[:, 0]

    def int_phi_phi(self, i, j):
        """
        Computes the integral of product of two shape functions using
        Gaussian quadrature.

        Parameters:
        – i, j : int, indices of the shape functions.

        Returns:
        – float, integral result.
        """

        def integrand(x):
            phi = self.phi(x)
            return phi[i] * phi[j]

```

```
        return fixed_quad(integrand, self.nodes[0].x, self.nodes[1].x, n=2)[0]

    def int_dphi_dphi(self, i, j):
        """
        Computes the integral of the derivatives of two shape functions.
        Since we have linear shape functions, the derivative is a constant.
        If i == j, the derivative is 1/h, otherwise it is -1/h.

        Parameters:
        - i, j : int, indices of the shape functions.

        Returns:
        - float, integral result.
        """

        if i == j:
            return 1 / self.h
        else:
            return -1 / self.h

    def stiffness_matrix(self):
        """
        Constructs the local stiffness matrix for the element.
        # $$K_{ij} = ik\\left.[\varphi_j\varphi_i]\right|_{x=1}
        # + \\int_0^1\\dfrac{d \varphi_j}{dx}(x)\\dfrac{d \varphi_i}{dx}(x)dx
        # + k^2 \\int_0^1 \varphi_j(x) \varphi_i(x) dx$$

        Returns:
        - np.ndarray, 2x2 complex stiffness matrix.
        """

        K = np.zeros((2, 2), dtype=complex)

        phi_1 = self.phi(1.)
        K[0, 0] = 1j * self.k * phi_1[0] * phi_1[0]
        K[0, 1] = 1j * self.k * phi_1[0] * phi_1[1]
        K[1, 0] = 1j * self.k * phi_1[1] * phi_1[0]
        K[1, 1] = 1j * self.k * phi_1[1] * phi_1[1]

        K[0, 0] += -self.int_dphi_dphi(0, 0)
        K[0, 1] += -self.int_dphi_dphi(0, 1)
        K[1, 0] += -self.int_dphi_dphi(1, 0)
        K[1, 1] += -self.int_dphi_dphi(1, 1)

        K[0, 0] += self.k**2 * self.int_phi_phi(0, 0)
        K[0, 1] += self.k**2 * self.int_phi_phi(0, 1)
        K[1, 0] += self.k**2 * self.int_phi_phi(1, 0)
        K[1, 1] += self.k**2 * self.int_phi_phi(1, 1)

        return K


class FEM1D:
    def __init__(self, n_elements, k, u_0=1., L=1.):
        """
        Solves the 1D Helmholtz equation using the FEM.
        The domain is assumed to be between 0 and L : 0 <= x <= L.

        Attributes:
        - n_elements : positive int, the number of elements in the mesh.
        - k : float, wavenumber for the Helmholtz equation.
        - u_0 : complex, Dirichlet boundary condition at the first node.
        - L : float, length of the domain.
        """

        if n_elements <= 0:
            raise ValueError("n_elements (the number of elements) \
            should be greater than 0")

        n_nodes = n_elements + 1
```

(same remark as for draft #3

BC should not be handled at the element level)

```
        self.nodes = [Node(x, i)
                      for i, x in enumerate(np.linspace(0, L, n_nodes))]
        self.elements = [Element1D([self.nodes[i],
                                    self.nodes[i + 1]],
                                   L / n_elements,
                                   k) for i in range(n_elements)]
        self.n_elements = n_elements
        self.n_nodes = n_nodes
        self.u_0 = u_0

    def assemble(self):
        """
        Assembles the global stiffness matrix by
        summing local element matrices.

        Returns:
        – csr_matrix, assembled stiffness matrix.
        """

        K_i = []
        K_j = []
        K_v = []

        for element in self.elements:
            local_stiffness = element.stiffness_matrix()
            node_ids = [element.nodes[0].global_idx,
                        element.nodes[1].global_idx]

            K_i.extend([node_ids[0], node_ids[0],
                        node_ids[1], node_ids[1]])
            K_j.extend([node_ids[0], node_ids[1],
                        node_ids[0], node_ids[1]])
            K_v.extend([local_stiffness[0, 0], local_stiffness[0, 1],
                        local_stiffness[1, 0], local_stiffness[1, 1]])

        return csr_matrix((K_v, (K_i, K_j)),
                          shape=(self.n_nodes, self.n_nodes))

    def solve(self):
        """
        Solves the FEM system, applying the Dirichlet boundary condition.

        Returns:
        – np.ndarray, solution vector at nodes.
        """

        K = self.assemble()

        u = np.zeros(self.n_nodes, dtype=complex)
        # Apply the Dirichlet boundary condition at the first node
        u[0] = self.u_0

        # Create the right-hand side vector (RHS)
        # RHS is -K times the boundary condition,
        # this is equivalent to [K_not_0] @ u = 0
        b = -K[:, 0] * self.u_0

        # Reduced stiffness matrix (excluding the first node)
        K_reduced = K[1:, 1:]

        # Reduced RHS (excluding the contribution from the first node)
        b_reduced = b[1:]

        u[1:] = spsolve(K_reduced, b_reduced)

        return u

    def sol(self, x):
        """
        Evaluates the FEM solution at arbitrary points using
```

```
        the Galerkin approximation : $u(x) = \\sum_{i=1}^{N} \\phi_i(x) u_i$.

        Parameters:
        – x : array–like, points at which to evaluate the solution.

        Returns:
        – tuple (solution values, (node coordinates, solution at nodes))
        """

            x = np.atleast_1d(x)
            u = self.solve()  # nodes values

            u_vals = np.zeros(x.shape, dtype=complex)

            for element in self.elements:
                phi = element.phi(x)

                for local_idx, node in enumerate(element.nodes):
                    u_vals += phi[local_idx] * u[node.global_idx]

            if x.size > 1:
                return u_vals, ([n.x for n in self.nodes], u)
            else:
                return u_vals[0], ([n.x for n in self.nodes], u)

# == Part c) Convergence Analysis ==


def errorL2(args):
    """
    Computes the L2 error between the FEM solution and the exact solution.

    Parameters:
    – args : tuple, arguments for the error computation (mu, k, x0, x1, res).

    Returns:
    – float, L2 error between the FEM and exact solutions
    """

        mu, k, x0, x1, res = args
        fem = FEM1D(mu, k)
        u_fem, _ = fem.sol(np.linspace(x0, x1, res))
        u_exact = exact_solution(np.linspace(x0, x1, res), k)
        return np.sqrt(np.trapz(np.abs(u_fem - u_exact)**2, dx=(x1 - x0) / res))


def convergence_analysis(k, mu_h, x0=0, x1=1, res=2000):
    """
    Conducts a convergence analysis for the FEM solution.

    Parameters:
    – k : float, wavenumber for the Helmholtz equation.
    – mu_h : array–like, number of elements to test for convergence.
    – x0, x1 : float, domain limits.
    – res : int, resolution for the solution.

    Returns:
    – np.ndarray, errors for each number of elements in mu_h.
    """

        errors = []
        # I do multi processing to speed up the computation
        with ProcessPoolExecutor() as executor:
            for error_val in executor.map(errorL2, [(mu, k, x0, x1, res)
                                                    for mu in mu_h]):
                errors.append(error_val)

        return np.array(errors)
```

```python
if __name__ == "__main__":
    x = np.linspace(0, 1, resolution_plot)

    # == Part a) Exact Solution ==
    u_exact_k_pi = exact_solution(x, np.pi)
    plot_solution(x, [np.real(u_exact_k_pi), np.imag(u_exact_k_pi)],
                  title=r"$k=\pi$", x_label=r"$x$",
                  y_label=r"$u(x) = \Re\{u(x)\} + i \Im\{u(x)\}$",
                  show=True, legend=[r"$\Re\{u(x)\}$", r"$\Im\{u(x)\}$"],
                  x_lim=[0, 1], save_path="fig1_a.pdf")

    u_exact_k_7pi = exact_solution(x, 7 * np.pi)
    plot_solution(x, [np.real(u_exact_k_7pi), np.imag(u_exact_k_7pi)],
                  title=r"$k=7\pi$", x_label=r"$x$",
                  y_label=r"$u(x) = \Re\{u(x)\} + i \Im\{u(x)\}$",
                  show=True, legend=[r"$\Re\{u(x)\}$", r"$\Im\{u(x)\}$"],
                  x_lim=[0, 1], save_path="fig1_b.pdf")

    # == Part a) Wave space-time plot ==
    x = np.linspace(0, 1, resolution_plot)
    t = np.linspace(0, 10, resolution_plot)

    plot_solution_wave_space_time(x, t, k=1, omega=1, A=1, B=1j,
                                  save_path="wave_space_time_B_i.pdf")
    plot_solution_wave_space_time(x, t, k=1, omega=1, A=1, B=1,
                                  save_path="wave_space_time_B_1.pdf")

    # == Part b) FEM Approximation ==
    num_elements = 10

    k = np.pi
    fem = FEM1D(num_elements, k)
    u_fem_k_pi, u_nodes_k_pi = fem.sol(x)
    plot_solution(x, [np.real(u_fem_k_pi), np.imag(u_fem_k_pi)],
                  title=r"$k=\pi$", x_label=r"$x$",
                  y_label=r"$u_{FEM}(x) = \Re\{u_{FEM}(x)\} \
            + i \Im\{u_{FEM}(x)\}$",
                  show=True,
                  legend=[r"$\Re\{u_{FEM}(x)\}$", r"$\Im\{u_{FEM}(x)\}$"],
                  x_lim=[0, 1], save_path="fig2_a.pdf", nodes=u_nodes_k_pi)

    k = 7 * np.pi
    fem = FEM1D(num_elements, k)
    u_fem_k_7pi, u_nodes_k_7pi = fem.sol(x)
    plot_solution(x, [np.real(u_fem_k_7pi), np.imag(u_fem_k_7pi)],
                  title=r"$k=7\pi$", x_label=r"$x$",
                  y_label=r"$u_{FEM}(x) = \Re\{u_{FEM}(x)\} \
        + i \Im\{u_{FEM}(x)\}$",
                  show=True,
                  legend=[r"$\Re\{u_{FEM}(x)\}$", r"$\Im\{u_{FEM}(x)\}$"],
                  x_lim=[0, 1], save_path="fig2_b.pdf", nodes=u_nodes_k_7pi)

    # == Part c) Convergence Analysis ==

    mu_h = np.array([2**i for i in range(1, 20)])

    k = np.pi
    error_k_pi = convergence_analysis(k, mu_h)

    k = 7 * np.pi
    error_k_7pi = convergence_analysis(k, mu_h)

    plot_solution(mu_h, [error_k_pi, error_k_7pi],
                  x_label=r"Number of elements $\mu_h$",
                  y_label=r"L2 Error between $u_{\text{exact}}$ \
        and $u_{\text{FEM}}$",
                  show=True, legend=[r"$k=\pi$", r"$k=7\pi$"],
                  save_path="fig3.pdf",
                  use_loglog=True, marker='o',
                  nodes=[[mu_h, mu_h], [error_k_pi, error_k_7pi]],
```

```
                    x_lim=[mu_h[0], mu_h[-1]])
```