Numerical Methods for Partial Differential Equations - Notes - v0.3.0

260236

September 2025

Preface

Every theory section in these notes has been taken from the sources:

- Course slides. [1]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on numerical methods for partial differential equations. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

1 Basic Concepts

In this course, we introduce numerical methods for the solution of **Partial Differential Equations** (PDEs), with focus on the **Finite Element** (FE) **method**[1] and the use of the computer for the construction of the PDEs numerical solution.

We will consider the numerical approximation of elliptic and parabolic PDEs by considering their variational formulation, Galërkin and FE approximations in 1D/2D/3D, the theoretical properties and practical use of the methods, algorithmic aspects, and interpretation of the numerical results.

Advanced topics include the approximation of saddle-point PDEs (Stokes equations), vectorial, nonlinear, and multiphysics differential problems, domain decomposition methods exploiting the properties of the PDEs, and the introduction to parallel computing for the FE method, i.e., in the *High Performance Computing* (HPC) framework.

Finally, the course will feature the use of the `deal.II` software library, a C++ open source FE library, and ParaView for the visualization of numerical solution and scientific computing data.

[1] The **Finite Element Method (FEM)** is a popular method for numerically solving differential equations arising in engineering and mathematical modeling. Typical problem areas of interest include the traditional fields of structural analysis, heat transfer, fluid flow, mass transport, and electromagnetic potential. Computers are usually used to perform the calculations required. With high-speed supercomputers, better solutions can be achieved, and are often required to solve the largest and most complex problems. (source)

1.1 Mathematical Models and Scientific Computing

Definition 1: Mathematical Model

A **Mathematical Model** is a **set of** (algebraic or differential) **equations that is able to represent the features of a complex system or process**.

? Why do they exist?

Models are **developed** to:

- Describe
- Forecast
- Control

The **behavior or evolution of such systems**.

We are interested in the physics models. **Physics-based models** are those **mathematical models that are derived from physical principles** (like conservation laws of mass, momentum, energy, etc.) **and that encode natural laws of leading to (differential) equations whose solutions are often represented in the form of functions**. However, the analytical solution of such models is rarely available in closed form, for which numerical approximation methods are instead employed.

Definition 2: Numerical Modelling

Numerical Modelling indicates **sets of numerical methods that determine an approximate solution of the original** (often infinite-dimensional) **mathematical model**, by turing it into a *discrete problem* (algebraic, finite-dimensional), whose dimension (size) is typically very large.

Definition 3: Scientific Computing

Scientific Computing is **a branch** of Mathematics **that numerically solves** (differential) **mathematical models by building approximate solutions though the use of a calculator**.

For numerical models of large size, parallel architectures for calculators and the HPC framework are typically used.

❓ Why did we introduce mathematical models and physical models?

Because they are connected and used together. Mathematical models are conventionally used altogether with theoretical (mathematical) models and experimental tests. Unfortunately, in several cases theoretical models are not available (like in Computational Medicine) or experimental tests are not meaningful or cannot be performed (for example, for nuclear testing). Physics-based models have witnessed an increasing role in the modern society in virtue of the massive developments of Scientific Computing and computational tools.

Since a large amount of data is becoming available from multiple sources nowadays, data-driven models are fundamentals. **Data-driven models** are those mathematical models built from meaningful data that do not rely on physical principles, because the latter are not available or are not reliable, and whose construction calls for statical learning methods.

Physics-based mathematical models (**mathematical problems**) are a fundamental pillar in the understanding and prediction of several physical phenomena and processes (**physical problems**). However, these mathematical models lead to problems that can rarely be solved analytically, or in an exact way (**exact solution**), especially for PDEs: with only a few exceptions, it is not possible to write their solution explicitly.

Numerical methods and numerical approximation techniques (**numerical problems**) serve the purpose to determine an **approximate solution** of a mathematical model. When the calculator is used to determine such approximate solution, the latter is called **numerical solution** (see the Figure 1).

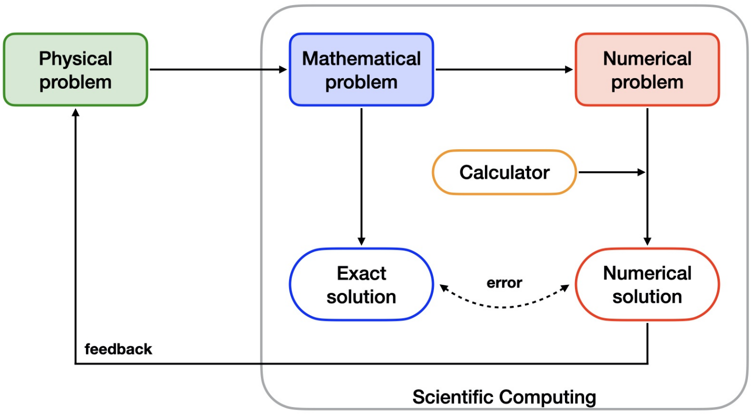

Figure 1: Scientific Computing.

1.2 Differential Models and PDEs

Definition 4: Partial Differential Equation (PDE)

A **differential equation** (model) is an equation that involves **one or more derivatives of an unknown function**. In an **Ordinary Differential Equation** (ODE), **every derivative of the unknown solution is with respect to a single independent variable**. If instead, derivatives are partial, then we have a **Partial Differential Equation (PDE)**.

In other words, it is a differential equation where its derivatives are partial.

There are different types of PDEs, and their nature depends on the conditions and their type. Mathematically, we can represent a **differential model** (equation) as follows:

$$\mathcal{P}\left(u; g\right) = 0 \qquad \text{differential equation (mathematical problem)} \tag{1}$$

Where:

- $\mathcal{P}$ indicates the ***model***;
- u is the ***exact solution***, a function of one or more independent variables (space and/or time variables);
- g indicates the ***data***.

1.2.1 ODEs

Ordinary Differential Equation (ODE) is also known as **initial value problem**.

I°ODE - Cauchy problem

A **first order** ODE, a **Cauchy problem**, is a differential problem, whose:

- ***Solution*** $u = u\left(t\right)$ is a function of a single independent variable t, often interpreted as time.
- A ***single condition*** is assigned on the solution, at a point (usually, the left end of the integration interval).

Its form is the following find $u : I \subset \mathbb{R} \to \mathbb{R}$ such that:

$$\begin{cases} \dfrac{\mathrm{d}u}{\mathrm{d}t}\left(t\right) = f\left(t, u\left(t\right)\right) \quad t \in I \\ u\left(t_0\right) = u_0 \end{cases} \tag{2}$$

Where:

- $I = \left(t_0, t_f\right] \subset \mathbb{R}$ is a ***time interval***;
- u_0 is the ***initial value*** assigned at $t = t_0$;
- $f : I \times \mathbb{R} \to \mathbb{R}$

Meaning. The equation describes the **evolution of a scalar quantity** u **over time** t, **without distribution in space**.

Vectorial problems. In vectorial problems, the **unknown is a vector-valued function** $\mathbf{u} = \mathbf{u}\left(t\right)$, where $\mathbf{u} = \left(u_1, \dots, u_m\right) \in \mathbb{R}^m$, with $m \geq 1$. The first order Cauchy problem reads: find $\mathbf{u} : I \subset \mathbb{R} \to \mathbb{R}^m$ such that:

$$\begin{cases} \dfrac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\left(t\right) = \mathbf{f}\left(t, \mathbf{u}\left(t\right)\right) \quad t \in I \\ \mathbf{u}\left(t_0\right) = \mathbf{u}_0 \end{cases}$$

Where $\mathbf{u}_0 \in \mathbb{R}^m$ is the initial datum and $\mathbf{f} : I \times \mathbb{R}^m \to \mathbb{R}^m$.

II°ODE - Cauchy problem

A **second order Cauchy problem** sees second order time derivatives and two initial conditions. It reads as: find $u : I \subset \mathbb{R} \to \mathbb{R}$ such that:

$$\begin{cases} \dfrac{\mathrm{d}^2u}{\mathrm{d}t^2}\left(t\right) = f\left(t, u\left(t\right), \dfrac{\mathrm{d}u}{\mathrm{d}t}\left(t\right)\right) \quad t \in I \\ \dfrac{\mathrm{d}u}{\mathrm{d}t}\left(t_0\right) = v_0 \\ u\left(t_0\right) = u_0 \end{cases} \tag{3}$$

Where the initial data are u_0 and v_0, while $f : I \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.

1.2.2 PDE, boundary value problem in 1D

The **Boundary value problem in 1D** is characterized by a **single independent variable** x, which represents the **space coordinate in an interval** $\Omega = (a, b) \in \mathbb{R}$ (1D).

The problem involves **second order derivatives of the unknown solution** $u = u(x)$ with respect to x. The value of u, or the **value of its first derivate**, is a **set at the two boundaries of the domain** (interval) Ω, that is at $x = a$ and $x = b$ (the domain boundary is $\partial\Omega = \{a, b\}$).

Let us consider the following **Poisson problem** with (homogeneous) Dirichlet boundary conditions: find $u : \Omega \subset \mathbb{R} \to \mathbb{R}$ such that:

$$\begin{cases} -\dfrac{\mathrm{d}^2 u}{\mathrm{d}x^2}(x) = f(x) & x \in \Omega = (a, b) \\ u(a) = u(b) = 0 \end{cases} \tag{4}$$

This equation models a **stationary phenomenon** (the time variable doesn't appear in fact) and represent a **diffusion model**.

Example 1

For example, the diffusion model models the diffusion of a pollutant along a 1D channel $\Omega = (a, b)$ or the vertical displacement of an *elastic thread* fixed at its ends. In the first case, $f = f(x)$ indicates the source of the pollutant along the flow, while in the second case, f is the traverse force acting on the elastic thread, in the hypothesis of negligible mass and small displacements of the thread.

⚖ Boundary value problem in 1D vs ODE

We remark that the **boundary value problem in 1D is a particular case of PDEs**, even if it involves only derivatives with respect to a single independent variable x. Indeed, even if apparently similar to a second order ODE, the boundary value problem is in reality substantially **different** from an ODE:

- In ODE, two conditions are set at $t = t_0$;
- In the boundary value problem in 1D, one condition is set at $x = a$ and the other one at $x = b$.

The conditions in the boundary value problem determine to the so-called global nature of the model.

1.2.3 PDE, initial and boundary value problem in 1D

Initial and boundary value problem in 1D is a type of problems that concern equations that **depend on space and time**:

- The **unknown solution** $u = u(x,t)$ both depends on the space coordinate $x \in \Omega \subset \mathbb{R}$ in 1D;
- The **time variable** $t \in I \subset I$.

In this case, the initial conditions at $t = 0$ must be prescribed, as well as the boundary conditions at the ends of the interval in 1D.

The **Heat equation**, also known as **Diffusion equation**, with Dirichlet boundary conditions assumes the following form: find $u : \Omega \times I \to \mathbb{R}$ such that:

$$\begin{cases} \dfrac{\partial u}{\partial t}(x,t) - \mu \dfrac{\partial^2 u}{\partial x^2}(x,t) = f(x,t) & x \in \Omega = (a,b), t \in I \\ u(a,t) = u(b,t) = 0 & t \in I \\ u(x,t_0) = u_0(x) & x \in \Omega = (a,b) \end{cases} \tag{5}$$

Example 2

For example, the unknown function $u(x,t)$ describes the temperature in a point $x \in \Omega = (a,b)$ and time $t \in I$ of a metallic bar covering the space interval Ω. The diffusion coefficient μ represents the thermal response of the material and it is related to its thermal conductivity. The Dirichlet boundary conditions express the fact that the ends of the bar are kept at a reference temperature (zero degrees in this case), while at time $t = t_0$ the temperature is assigned in each point $x \in \Omega$ through the initial function $u_0(x)$. Finally, the bar is subject to a heat source of linear density $f(x,t)$.

1.2.4 PDE, boundary value problem in multidimensional domains

The Poisson problem (equation 4, page 9) can be **extended in multidimensional domains** $\Omega \subset \mathbb{R}^d$, with $d = 2, 3$; the solution is $u = u(\mathbf{x})$, where $\mathbf{x} = (x_1, \dots, x_d)^T \in \mathbb{R}^d$. This leads to the following Poisson problem with (homogeneous) Dirichlet boundary conditions: find $u : \Omega \subset \mathbb{R}^d \to \mathbb{R}$ such that:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \ \text{(i.e. } \mathbf{x} \in \Omega) \\ u = 0 & \text{on } \partial\Omega \ \text{(i.e. } \mathbf{x} \in \partial\Omega) \end{cases} \tag{6}$$

Where:

- The **Laplace operator**:

$$\Delta u(\mathbf{x}) := \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}(\mathbf{x})$$

- The **domain** $\Omega \subset \mathbb{R}^d$ is endowed with boundary $\partial\Omega$;
- $f = f(x)$ is the ***external forcing term***.

This equation is used **for example** to **model the vertical displacement of an elastic membrane fixed at the boundaries**.

1.2.5 PDE, initial and boundary value problem in multidimensional domains

The **multidimensional** counterpart of the **heat equation** (5, page 10) reads: find $u : \Omega \times I \to \mathbb{R}$ such that:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \mu \Delta u = f & \mathbf{x} \in \Omega, t \in I \\ u(\mathbf{x}, t) = 0 & \mathbf{x} \in \partial\Omega, t \in I \\ u(\mathbf{x}, t_0) = u_0(\mathbf{x}) & \mathbf{x} \in \Omega \end{cases} \tag{7}$$

Where u_0 is the **initial datum**. The **solution** is $u = u(\mathbf{x}, t)$.

1.2.6 Classification of PDEs

A PDE is a relationship among:

- The partial derivatives of a function $u = u\left(\mathbf{u}, t\right)$, that is the PDE **solution**;
- **Spatial coordinates** $\mathbf{x} = \left(x_1, \dots, x_d\right)^T \in \mathbb{R}^d$ on which the solution depends (if the problem is defined in a spatial domain $\Omega \subset \mathbb{R}^d$).
- **Time variable** t.

Therefore, a PDE can be written as:

$$\mathcal{P}\left(u, \frac{\partial u}{\partial t}, \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_d}, \dots, \frac{\partial^{p_1 + \dots + p_d + p_t} u}{\partial x_1^{p_1} \dots \partial x_d^{p_d} \, \partial t^{p_t}}, \mathbf{x}, t; g\right) = 0 \tag{8}$$

Where $p_1, \dots, p_d, p_t \in \mathbb{N}$ and g are the data.

Definition 5: PDE order

The **PDE order** is the **maximum order of derivation** that appears in $\mathcal{P}$, that is:

$$q = p_1 + \dots + p_d + p_t \tag{9}$$

Definition 6: PDE is linear

The **PDE is linear** if $\mathcal{P}$ **linearly depends** on u and its **derivatives**.

Classification

Let us focus on linear PDEs of order $q = 2$ with constant coefficients, so that the general PDE formulation is:

$$\mathcal{L}u = g$$

Where $\mathcal{L}$ is a second order, **linear differential operator**. When only two independent variables (our case) x_1 and x_2 are considered, the operator $\mathcal{L}$ applied to the function u reads:

$$\mathcal{L}u = A \cdot \frac{\partial^2 u}{\partial x_1^2} + B \cdot \frac{\partial^2 u}{\partial x_1 \, \partial x_2} + C \cdot \frac{\partial^2 u}{\partial x_2^2} + D \cdot \frac{\partial u}{\partial x_1} + E \cdot \frac{\partial u}{\partial x_2} + F \cdot u$$

For some constant coefficients $A, B, C, D, E, F, G \in \mathbb{R}$. If $d = 2$ (our case), the **independent variables** can represent the *space coordinates*:

- $x_1 = x$
- $x_2 = y$

After introducing the **PDE discriminant** (a quantity that helps determine the type of PDE):

$$\Delta := B^2 - 4AC \tag{10}$$

The PDE can be classified as:

- **Elliptic PDE** if $\Delta < 0$
- **Parabolic PDE** if $\Delta = 0$
- **Hyperbolic PDE** if $\Delta > 0$

? What are the implications of PDE classification?

The different nature of the PDE impacts on:

- **Type** and **amount of data to prescribe as boundary**;
- **Initial conditions** to ensure the well-posedness of the problem (existence and uniqueness of the solution);
- The **phenomena that can be described** by the PDE;
- The **information that encapsulates**.

In general:

- **Elliptic PDE** typically describes **stationary phenomena**, without time evolution of quantities.
- **Parabolic PDE** describes **wave propagation phenomena** with infinite velocity of propagation.
- **Hyperbolic PDE** describes **wave propagation phenomena** but with finite velocity of propagation.

1.3 Numerical Methods

Since in most cases of practical interest we **cannot solve a PDE analytically**, we need to use **numerical methods** that allow us to construct an *approximation* u_h of the *exact solution* u, for which the corresponding *error* $(u - u_h)$ can be quantified and/or estimated.

$$\begin{array}{cc} \mathcal{P}\left(u; g\right) = 0 & \text{PDE (mathematical problem)} \\ \downarrow & \textit{numerical method} \\ \mathcal{P}_h\left(u_h; g_h\right) = 0 & \text{approximate PDE (numerical problem)} \end{array}$$

Where:

- g_h is an approximation of the data g;
- $\mathcal{P}_h$ is a characterization of the approximate problem.

The subscript h indicates a **discretization parameter** that characterizes the numerical approximation. Conventionally, the smaller is h, the better is the approximation of u made by u_h. Furthermore, the error $(u - u_h)$ tends to zero as h gets smaller and smaller. In this course, we will specifically introduce the FE method (page 4) to build the numerical approximation of PDEs.

Summary Notation

Notation	Description
$\mathcal{P}\left(u; g\right) = 0$	PDE (mathematical problem)
u	***exact solution*** of a PDE
u_h	***approximate solution*** of a PDE
$(u - u_h)$	***error*** (quantified and/or estimated; tends to zero if h is smaller)
h	***discretization parameter*** (↓ smaller h, better approximation; ↑ higher h, poor approximation)
$\mathcal{P}_h\left(u_h; g_h\right) = 0$	approximate PDE (numerical problem)
g_h	***approximation*** of the ***data*** g
$\mathcal{P}_h$	***characterization*** of the approximate problem.

Table 1: Notation used to approximate the PDE with numerical methods.

1.4 From Mathematical to Numerical Problem

1.4.1 The Mathematical Problem (MP)

Let us consider a **Physical Problem (PP)** endowed with a **physical solution**, let say u_{ph}, and **dependent on data** indicated with g.

The **Mathematical Problem (MP)** is represented by the **mathematical formulation of the PP** and has **mathematical solution** u. Therefore, we indicate the MP as:

$$\mathcal{P}(u; g) = 0 \tag{11}$$

Where:

- $u \in \mathcal{U}$
- $g \in \mathcal{G}$, and $\mathcal{G}$ is the set or space of **admissible data**.

Where $\mathcal{U}$ and $\mathcal{G}$ are suitable sets or spaces.

Definition 7: Model Error

The error between the physical and mathematical solutions is called **Model Error**:

$$e_m := u_{ph} - u \tag{12}$$

Where:

- u_{ph} is the physical solution;
- u is the mathematical solution.

The model error takes into account all those **characteristics of the PP that are not represented or captured by the MP**.

? When a Mathematical Problem is *well-posed*?

Definition 8: *well-posed* MP

The mathematical problem MP is *well-posed* (**stable**) if and only if there **exists a unique solution** $u \in \mathcal{U}$ **that continuously depend on the data** $g \in \mathcal{G}$.

From the previous definition, we remark that $\mathcal{G}$ is the set of admissible data, i.e., those for which the MP admits a unique solution. Furthermore, *continuously depend on the data* means that **small perturbations on data** $g \in \mathcal{G}$ **lead to small changes on the solution** $u \in \mathcal{U}$ of the MP. However, a measure of this sensitivity is given by the condition number of the MP.

1.4.2 The Numerical Problem (NP)

The **Numerical Problem (NP)** is an **approximation of the Mathematical Problem** (MP, equation 11, page 15). We indicate its **numerical solution** as u_h, where h stands as a suitable **discretization parameter**.

$$\mathcal{P}_h\left(u_h; g_h\right) = 0 \tag{13}$$

Where:

- $u_h \in \mathcal{U}_h$
- $g_h \in \mathcal{G}_h$, and g_h is the representation of the **data in the NP**.

Where $\mathcal{U}_h$ and $\mathcal{G}_h$ are suitable sets or spaces.

Definition 9: Truncation Error

The error between the mathematical and numerical solutions is called **Truncation Error**:

$$e_h := u - u_h \tag{14}$$

Where:

- u is the mathematical solution;
- u_h is the numerical solution.

The truncation error can be considered as the error resulting from the **discretization of the MP**.

Numerical solution calculated on the computer

When the numerical solution is computed by running the algorithm on a computer, we need more notations and concepts.

- $\widehat{u}_h$ is the **final solution**.
- The final solution is affected by a **Round-Off error** e_r:

  $$e_r := u_h - \widehat{u}_h \tag{15}$$

  Such round-off errors depend on the machine architecture, on the representation of the numbers at the calculator, and on operations made in floating-point arithmetic.

- The truncation error e_h (equation 14, page 16) and the Round-Off error e_r (equation 15) concur to determine the **Computational error** e_c:

  $$e_c := e_h + e_r = (u - \cancel{u_h}) + (\cancel{u_h} - \widehat{u}_h) = u - \widehat{u}_h \tag{16}$$

  For some NP, we can have a round-off error less than a truncation error $|e_r| \ll |e_h|$, for which $e_c \approx e_h$.

When a Numerical Problem is *well-posed*?

Definition 10: *well-posed* NP

The numerical problem NP is *well-posed* (**stable**) if and only if there **exists a unique solution** $u_h \in \mathcal{U}_h$ **that continuously depends on the data** $g_h \in \mathcal{G}_h$.

Consider the numerical solution calculated only on the computer

In practice, numerical solutions are computed on a computer. Therefore, it is reasonable to obtain a computational error that tends to zero as the numerical method improves, namely as the discretization parameter h goes to zero. This concept is encoded in the definition of convergence.

Definition 11: *convergence* NP

The NP is **convergent** when the **computational error tends to zero** for h tending to zero, that is:

$$\lim_{h \to 0} e_c = 0 \tag{17}$$

A crucial aspect is to qualify the convergence of the NP, that is determining the convergence order of the NP.

Definition 12: convergence order

If $|e_c| \leq Ch^p$, with C a positive constant independent of h and p, then the NP is **convergent with order** p.

How to estimate the convergence order?

The convergence order can be estimated for many reasons (error estimation, method comparison, accuracy verification, etc.). If there exists a constant $\tilde{C} \leq C$ independent of h and p such that $\tilde{C}h^p \leq |e_c| \leq Ch^p$, then we can write $|e_c| \approxeq Ch^p$ and we can **estimate the convergence order** p **of the NP by using the known solution** u **of the MP**. There are two approaches:

1. **Algebraic estimation** of p.

   (a) We compute the computational errors e_{c1} and e_{c2} for the NP corresponding to two different values of h that are "sufficietly" small, say h_1 and h_2.

   (b) Then:

   - Writing $|e_{c1}| \approxeq Ch_1^p$ and $|e_{c2}| \approxeq Ch_2^p$
   - Noticing that $\dfrac{|e_{c1}|}{|e_{c2}|} = \left(\dfrac{h_1}{h_2}\right)^p$

We estimate the order p as:

$$p = \frac{\log\left(\frac{|e_{c1}|}{|e_{c2}|}\right)}{\log\left(\frac{h_1}{h_2}\right)} \tag{18}$$

2. **Graphical estimation** of p. We represent the errors $|e_c|$ and h on a plot in log-log scale. As $\log|e_c| = \log(Ch^p) = log(C) + p\log(h)$, we have $p = \arctan(\theta)$, where θ is the slope of the curve (h, e_c), a straight line in log-log scale. Instead of computing θ, it is possible to verify that the curves (h, e_c) and (h, h^p) are parallel in log-log scale.

   In other words it involves plotting the error against the step size on a log-log scale and analyzing the resulting graph:

   (a) ***Compute Errors***: Perform the numerical method for several step sizes h, such as $h_1, h_2, h_3, \ldots$, and compute the corresponding errors $e_1, e_2, e_3, \ldots$.

   (b) ***Log-Log Plot***: Plot the errors e_i against the step sizes h_i on a log-log scale. This means we plot $\log(h_i)$ on the x-axis and $\log(e_i)$ on the y-axis.

   (c) ***Linear Relationship***: If the method has a convergence order p, the relationship between the error and the step size should follow $e \approx Ch^p$. Taking the logarithm of both sides gives:

   $$\log(e) \approx \log(C) + p\log(h)$$

   This indicates that the plot of $\log(e)$ versus $\log(h)$ should be a straight line with a slope equal to p.

   (d) ***Determine Slope***: The slope of the line in the log-log plot is the convergence order p. We can estimate this slope by fitting a linear regression line to the data points.

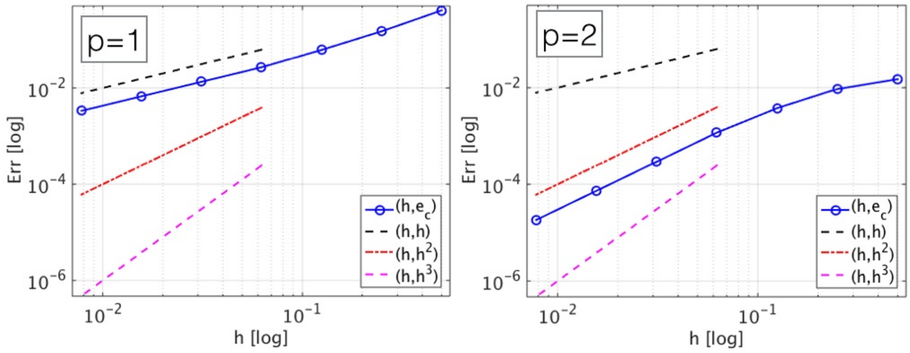

Figure 2: Graphical estimation of the convergence order p of a NP: computational errors $|e_c|$ vs h.

❓ When is convergence guaranteed in NP?

Unfortunately, a ***well-posed*** **NP is not necessarily convergent**. To ensure convergence of the NP, this is required to satisfy the consistency property (roughly speaking, the NP must be a "faithful copy" of the original MP).

Definition 13: NP consisten and strongly consistent

The Numerical Problem NP is **consistent** if and only if:

$$\lim_{h \to 0} \mathcal{P}_h\left(u; g\right) = \mathcal{P}\left(u; g\right) = 0 \qquad g \in \mathcal{G}_h$$

The Numerical Problem NP is ***strongly*** **consistent** if and only if:

$$\mathcal{P}_h\left(u; g\right) \equiv \mathcal{P}\left(u; g\right) = 0 \qquad \forall h > 0,\ g \in \mathcal{G}_h$$

Let highlights the main differences:

- Definition:
  - ***Consistent***. Consistency requires that as the discretization parameter h tends to zero $\lim\limits_{h \to 0}$, the process $\mathcal{P}_h(u; g)$ approaches the exact process $\mathcal{P}(u; g)$ and both become zero. This means that **over time and with finer discretization**, the **numerical approximation converges to the exact solution**.
  - ***Strongly Consistent***. Strong consistency means that for any positive value of h ($\forall h > 0$, no matter how small), the process $\mathcal{P}_h(u; g)$ is exactly equal to the exact process $\mathcal{P}(u; g)$ and both are zero. This implies that the **numerical approximation already matches the exact solution for any step size**.
- Condition of h:
  - ***Consistent***. The condition applies in the limit as h approaches zero. The **process gradually converges to the exact solution as the discretization parameter becomes infinitesimally small**.
  - ***Strongly Consistent***. The condition applies for all $h > 0$. This is a **stronger requirement** because it demands that the numerical method is **accurate for any discretization parameter**, not just in the limit.

In practice, the *Consistent* indicates that the numerical method improves and approaches the exact solution as the discretization parameter is refined. It guarantees eventual ***accuracy***, ***but not necessarily immediate or uniform accuracy for larger*** h. On the other hand, *Strongly Consistent* indicates that the numerical method is always accurate, regardless of the discretization parameter. This implies a ***higher level of reliability and precision for any*** h, making it a stronger and more robust form of consistency.

The **Lax-Richtmyer Equivalence Theorem** is a cornerstone of numerical analysis, linking the concepts of consistency, well-posedness (stability), and convergence. It provides a **rigorous framework for validating numerical methods and ensuring that they produce accurate and reliable solutions**. Furthermore, the following theorem guarantees that if a ***numerical problem is well-posed and consistent, then the NP is also convergent***.

Theorem 1 (Lax-Richtmyer, equivalence)**.** *If the Numerical Problem NP:*

$$\mathcal{P}_h\left(u_h; g_h\right) = 0 \qquad u_h \in \mathcal{U}_h,\ g_h \in \mathcal{G}_h$$

Is consistent:

$$\lim_{h \to 0} \mathcal{P}_h\left(u; g\right) = \mathcal{P}\left(u; g\right) = 0 \qquad g \in \mathcal{G}_h$$

Then, it is well-posed if and only if it is also convergent.

It is a fundamental theorem in numerical analysis because **it ensures that stability and consistency are sufficient to guarantee convergence**. Conversely, if we have a proof that the NP is consistent, we "only" need to show that the problem is well-posed to automatically prove convergence (and vice versa).

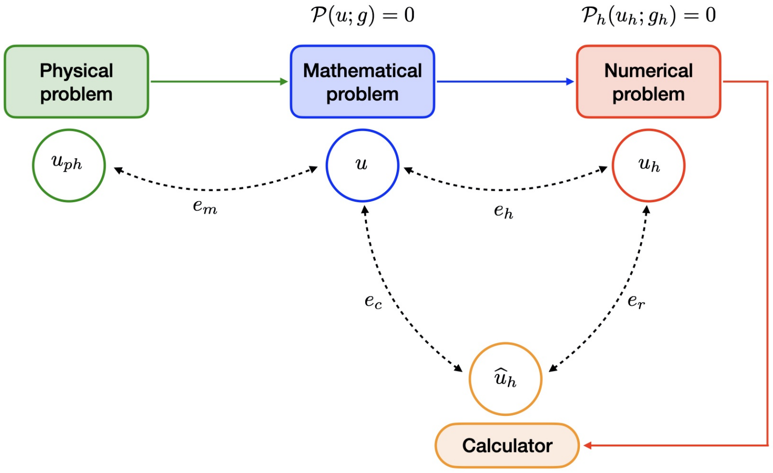

Figure 3: Physical (PP), Mathematical (MP), and Numerical (NP) problems. Corresponding solutions (u_{ph}, u, u_h, and $\widehat{u}_h$) and errors (model $e_m = u_{ph} - u$, truncation $e_h = u - u_h$, round-off $e_r = u_h - \widehat{u}_h$, and computational $e_c = e_h + e_r$ errors).

2 Laboratory

2.1 Introduction

The laboratory sessions complement the theoretical course by providing a **hands-on experience** in the numerical approximation of PDEs. The main goal is to bridge the gap between the mathematical formulation of PDEs, their variational and finite element discretizations, and their actual computer implementation.

Throughout the laboratories, we will progressively construct finite element solvers for a variety of model PDEs:

- Starting from the Poisson equation in 1D, moving towards multidimensional diffusion-reaction problems;
- Introducing verification and validation strategies for numerical codes;
- Extending to time-dependent problems such as the heat equation;
- Exploring nonlinear PDEs, elasticity, and saddle-point problems such as Stokes flows.

Each laboratory is designed to emphasize not only the **mathematical correctness** of the discretization, but also the **computational aspects**: efficiency, robustness, and scalability.

📗 Software

The laboratory relies on:

- `deal.II` is an open-source `C++` software library for solving partial differential equations (PDEs) using the finite element method (FEM).

  The name `deal.II` stands for "Differential Equations Analysis Library, version II". It is a finite element framework designed to make it easier to implement complex numerical methods for PDEs. It is widely used in both academia and industry for research, education, and simulation.

  ? Why is it used in laboratories? It provides full control over every step of the FEM pipeline: mesh generation, assembly, linear solvers, visualization. It forces us to understand the mathematics and the implementation, instead of just using a black-box solver. Finally, it is well documented and has extensive tutorial programs (step-1, step-2, . . .), which the laboratory exercises build upon.

  ? Why do people say that `deal.II` is complicated? It's a `C++` library, not a GUI tool. Unlike COMSOL or ANSYS, we write `C++` code that uses `deal.II`'s classes. That means we need to understand:

  - FEM theory (variational forms, weak formulations, basis functions);
  - The `C++` programming model (templates, object-oriented design);
  - The linear algebra backend (solvers, preconditioners).

The first labs are deliberately kept simple because even setting up a finite element mesh, assembling the stiffness matrix, and applying boundary conditions requires some work.

? **Why is `deal.II` respected?** Despite the initial complexity, `deal.II` is **very mature and widely used in scientific computing**. Aerospace, automotive, and energy companies use FEM frameworks like `deal.II`, FEniCS, or proprietary codes to simulate physical systems. Research groups in Europe and the US use `deal.II` on HPC clusters for multi-physics and optimization problems.

- `ParaView` is an **open-source data analysis and visualization application**. It's designed to handle very large scientific datasets (from MBs to TBs). Our finite element codes produce numerical solutions (vectors of values at mesh nodes, or fields defined in VTK/VTU file formats). These are not human-friendly to interpret. ParaView lets us **load the mesh and solution** files produced by `deal.II`. We can then **plot solutions in 2D/3D**, extract values along a line or surface, animate time-dependent results, compute integrals, etc..

- `gmsh` is an **open-source mesh generator** (also with a built-in post-processor). It allows us to create computational grids for finite element methods. For simple domains (intervals, unit squares, unit cubes), `deal.II` can generate meshes internally. But for **non-trivial geometries** (like irregular domains, or those with boundary partitions), we need an external mesh generator.

In addition, profiling and debugging tools (e.g. `TimerOutput`, `gperftools`) are used to analyze and optimize performance.

Computational Environment

While the course suggests using the MK module system, a more versatile approach is to work with either:

- a **native installation** of the required software stack (`deal.II`, `ParaView`, `gmsh`, etc.), or

- a **Docker container**, which ensures reproducibility and avoids configuration issues.

These alternatives are recommended for who prefer independence from the university's MK modules and allow seamless experimentation on personal or cloud-based machines.

Environment Setup

Download version `9.5.0` of `deal.II` (which we are using for the course) from their website, following their guide. If we are using WSL or Ubuntu, we can download it more easily using the command:

```
sudo apt-get install libdeal.ii-dev
```

Download the `ParaView` visualization software from its original website:

ParaView

If we're using Ubuntu, the easier command for the latest version is:

```
sudo apt install paraview
```

2.2 FEM for Poisson 1D

2.2.1 What is the Poisson Equation?

The **Poisson equation** is one of the most fundamental Partial Differential Equations (PDEs). In general form (in multiple dimensions):

$$-\Delta u(x) = f(x), \quad x \in \Omega \tag{19}$$

With some boundary conditions on $\partial\Omega$. Where:

- $u(x)$ is the **unknown function** (temperature, displacement, potential, etc.).
- Δ is the **Laplacian operator**, i.e. sum of second derivatives.

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \cdots \tag{20}$$

- $f(x)$ is the **source term** (where heat is produced, where force acts, etc.).

So the Poisson equation says **the curvature of u (second derivate) balances the source f.**

❓ What does "1D" mean?

Normally the Poisson equation is written in 2D or 3D (for surfaces and volumes).

- In **2D**, it's like heat distribution on a plate.
- In **3D**, it's like heat or potential in a cube.

In **1D**, the domain $\Omega = (0, 1)$ is just a line (an interval). So the Laplacian reduces to an ordinary second derivative:

$$\Delta u(x) = \frac{d^2 u}{dx^2}$$

Therefore, in 1D, the Poisson equation looks like:

$$-u''(x) = f(x) \tag{21}$$

If we allow a variable coefficient $\mu(x)$, it becomes:

$$-\left(\mu(x)u'(x)\right)' = f(x)$$

Example 1: Physical analogy

Imagine a **metal bar of length 1**. Fix both ends to zero temperature (they're in contact with ice). Apply a **heat source** (or sink, if negative) in some region of the bar. Then the **temperature distribution** inside the bar is described by the 1D Poisson equation.
Another analogy. Think of a **stretched elastic string** fixed at both

ends. Apply a **vertical load** (the forcing f) on some region. The resulting shape of the string $u(x)$ satisfies the Poisson equation.

The core idea

The Poisson equation links:

- The **curvature** of the unknown function $u(x)$ (its second derivate)
- To the **source term** $f(x)$.

Formally:

$$-u''(x) = f(x) \quad \text{(in 1D)}$$

That means wherever $f(x)$ is nonzero, it *forces* the function $u(x)$ to bend (curve). If $f(x) = 0$, the equation reduces to:

$$-u''(x) = 0 \quad \Longrightarrow \quad u''(x) = 0$$

Whose solutions are straight lines. So **no sources**, **flat solution**.

Physical interpretations

We can understand Poisson in multiple ways, depending on the field:

1. **Heat conduction**. The equation says: "The way temperature curves along the bar is dictated by how much heat we add/remove locally".
   - $u(x)$: temperature.
   - $f(x)$: heat sources (positive) or sinks (negative).
2. **Electrostatics**. The equation says: "Charges create curvature in the potential".
   - $u(x)$: electric potential.
   - $f(x)$: charge distribution (density).
3. **Elasticity**. The equation says: "Where we press on the string, it bends".
   - $u(x)$: displacement of a string or membrane.
   - $f(x)$: applied load (force per unit length).

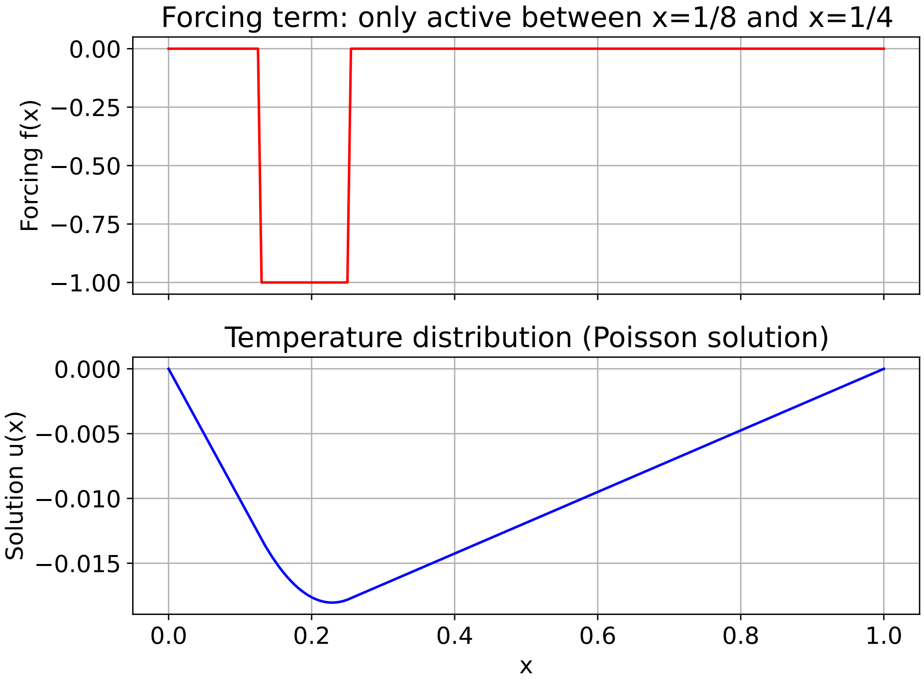

Figure 4: Visual explanation of the 1D Poisson problem. **Top graph (red)** is the forcing term $f(x)$. It is **zero everywhere**, except between $x = \frac{1}{8}$ and $x = \frac{1}{4}$, where it is negative (-1). That means only in that region we have a "sink" of heat. The **bottom graph (blue)** is the solution $u(x)$, i.e. the **temperature profile** along the bar. The ends are fixed at $u(0) = u(1) = 0$. In the middle, the solution bends downward because of the negative forcing. Outside the forcing region, the solution is almost straight (since $f = 0$, then the curvature is 0). So visually, the bar stays at 0 at the ends, but dips in the middle where we apply the negative forcing.

For example, this graph could represent heat conduction in a bar.

- $u(x)$: **temperature** along a thin bar of length 1.
- Boundaries: both ends clamped to 0 °C (in contact with ice).
- $f(x) = -1$ between $x = \frac{1}{8}$ and $x = \frac{1}{4}$: this is like a **cooling region** (a heat sink).
- Physical picture: the bar stays cold at the ends and gets even colder in the middle where cooling is applied, producing the "dip".

What is the purpose of the Poisson equation?

The Poisson equation gives us the **response of the system**, not just the input. Why? Because physical systems are not isolated points:

- Temperature at a point depends on how heat flows along the entire bar.
- Displacement of a string at one point depends on forces applied nearby and the string's stiffness.
- Potential at a point depends on all surrounding charges.

The Poisson equation encodes that **interaction through curvature**:

$$-u''(x) = f(x)$$

- The second derivate u'' measures how much the solution bends.
- The boundary conditions (ends fixed at 0) propagate constraints.
- The combination of f and boundaries gives the actual **shape of the solution**.

For example, consider figure 4 on page 26. In particular, consider our forcing (the red curve in the top graph). It is localized, but the solution (the blue curve in the bottom graph) is **spread out**:

- The dip is not only in the forcing region, it extends beyond, up to the ends.
- This spreading comes from the diffusion mechanism encoded by Poisson.

So if we only look at $f(x)$, we know *where the cause is.* But if we solve Poisson, we know *how the whole system reacts.*

Example 2: Real-world analogy

Imagine putting an ice cube (the sink) in one part of a metal bar. From the forcing alone, we know *where* the cooling happens. But we don't know how the **temperature profile along the whole bar** looks; is the bar uniformly cold? Does the dip propagate? Solving Poisson tells us exactly the **temperature distribution everywhere**, considering both the sink and the fixed-zero boundary conditions.

2.2.2 Problem definition

The Poisson equation itself is a general PDE:

$$-u''(x) = f(x) \quad \text{in some domain}$$

To make it a **mathematical problem** we say:

- *Where* we are solving it: the **domain** $\Omega = (0, 1)$.
- *What constraints* we impose: the **boundary conditions** $u(0) = u(1) = 0$.
- *What data* we have: here $\mu(x) = 1$ and a particular forcing $f(x)$.

Using these definitions, we can formulate the problem as a Boundary Value Problem (BVP). Without these specifications, "the Poisson equation" is too vague: infinitely many situations are possible, and no unique solution can be defined.

Deepening: Boundary Value Problem (BVP)

Before explaining what a BVP is, it is important to understand the difference between a classic ODE and a PDE.

? Differential Equations: ODE vs PDE

- An **ODE (Ordinary Differential Equation)** involves derivatives with respect to *one variable* (usually time t).
- A **PDE (Partial Differential Equation)** involves derivatives with respect to *several variables* (like space x, y, z, and maybe time).

Both need some extra information to be **solvable**. That "extra information" comes in two main flavors:

- **Initial conditions** (tell us the state at $t = 0$, then we can evolve forward in time).
- **Boundary conditions** (tell us what happens at the edges of the spatial domain, then we can solve inside).

Boundary Value Problem (BVP)

A **Boundary Value Problem (BVP)** is a differential equation (ODE or PDE) with conditions prescribed **at the boundary of the domain**. Formally:

$$\begin{cases} Lu(x) = f(x), & x \in \Omega, \\ \text{Boundary conditions on } \partial\Omega. \end{cases}$$

- L: differential operator (e.g. $-u''$ in 1D Poisson).
- Ω: $=$ the domain (like the interval $(0, 1)$).
- $\partial\Omega$: the boundary (here just the two points 0 and 1).

❓ Why Dirichlet Boundary Condition (Dirichlet BC)?

In this laboratory, we impose $u = 0$ at both ends. That corresponds physically to the ends of the bar are held at zero temperature. Other choices are possible (Neumann, Robin, etc.), but **Dirichlet** is the simplest starting case.

❓ Why we call it "Problem Definition"?

Because in numerical methods (and PDE theory) the workflow is always:

1. **Continuous problem definition**: PDE + domain + boundary conditions.
2. **Weak formulation**: rewrite it in an integral form suitable for analysis.
3. **Galerkin formulation**: restrict to a finite-dimensional space.
4. **Finite element formulation**: translate into linear algebra.
5. **Implementation**: write the solver in `deal.II`.

So "problem definition" is step 1 of this workflow.

Deepening: Dirichlet Boundary Condition

When we solve a PDE, we don't just solve "inside" the domain Ω. We must also tell the solver **what happens at the edges** (the boundary $\partial\Omega$).
There are three classical types:

1. **Dirichlet** → fix the **value** of the solution at the boundary.
2. **Neumann** → fix the **derivative/flux** of the solution at the boundary.
3. **Robin** (mixed) → fix a **combination** of value and derivative.

A **Dirichlet condition** prescribes directly the solution value:

$$u(x) = g(x) \quad \text{on } \partial\Omega \tag{22}$$

- If $g(x) = 0$, it's called a **Homogeneous Dirichlet condition**.
- If $g(x) \neq 0$, it's **Non-Homogeneous Dirichlet**.

In our case, we impose: $u(0) = u(1) = 0$. At the both ends of the bar, the temperature (or displacement, or potential) is forced to **zero**. That's a **homogeneous Dirichlet boundary condition**.

Problem Definition: Poisson Equation in 1D

We are working on the interval (domain):

$$\Omega = (0, 1)$$

The **equation** is:

$$\begin{cases} -\left(\mu(x)u'(x)\right)' = f(x), & x \in \Omega, \\ u(0) = u(1) = 0 \end{cases}$$

The unknown function is $u(x)$, for example the **temperature along a 1D bar**. The coefficient $\mu(x) = 1$ is the **diffusion coefficient** or **conductivity** of the material. If it varied, it would mean the material has regions that conduct more or less. Finally, the forcing term $f(x)$ is a piecewise function:

$$f(x) = \begin{cases} 0, & x \leq \frac{1}{8} \text{ or } x > \dfrac{1}{4}, \\ -1, & \dfrac{1}{8} < x \leq \dfrac{1}{4}. \end{cases}$$

There is a **negative source term** (a "sink" of heat, or a downward force density) only in the interval $\left(\dfrac{1}{8}, \dfrac{1}{4}\right]$. Outside, nothing happens ($f = 0$).

We use Dirichlet boundary conditions:

$$u(0) = u(1) = 0$$

- Physically: the ends of the bar are fixed to zero temperature.
- Mathematically: they "anchor" the solution and ensure uniqueness.

The PDE we wrote above (differential equation + boundary conditions) is called the **strong formulation**. It's "*strong*" because it requires $u(x)$ to be smooth enough so that derivatives exist in the classical sense. Later, we'll relax this condition with the **weak formulation**.

Deepening: Strong Formulation

The **Strong Formulation** of a PDE is the problem written:

- as a **differential equation** (derivatives explicitly present),
- together with **boundary conditions** (Dirichlet, Neumann, ...),
- requiring the solution $u(x)$ to be smooth enough for the derivatives to make sense pointwise.

So, for this laboratory, the strong formulation is exactly:

$$\begin{cases} -\left(\mu(x)u'(x)\right)' = f(x), & x \in (0, 1), \\ u(0) = u(1) = 0, \end{cases}$$

With $\mu(x) = 1$, and our piecewise forcing $f(x)$.

❓ Why "strong"?

Because it requires "strong" regularity:

- The solution u must be differentiable enough so that $u'(x)$ and $(\mu u')'(x)$ exist as classical derivatives.
- We must be able to plug $u(x)$ **directly into the PDE** and check if it satisfies the equation *point by point*.

If f is discontinuous (as in our lab, where it jumps at $x = \frac{1}{8}$ and $x = \frac{1}{4}$), the strong formulation becomes tricky because classical derivatives may not exist everywhere. That's why we move to the **weak formulation**: it relaxes smoothness requirements but still captures the PDE.

☰ Workflow in PDE analysis

1. **Strong formulation**: the PDE as we would write it in physics. Clear, but often too strict mathematically.
2. **Weak formulation**: rewrite as an integral equation using test functions and integration by parts. Because it is more flexible (allows solutions with less regularity, e.g. only square-integrable derivatives). This is the starting point for numerical methods.
3. **Galerkin / Finite Element formulation**: approximate the weak formulation in a finite-dimensional space, leading to a linear algebra system.

2.2.3 Weak formulation

We start from the **strong problem**:

$$\begin{cases} -u''(x) = f(x), & x \in (0,1), \\ u(0) = u(1) = 0 \end{cases}$$

To obtain the weak formulation, we use test functions.

1. **Introduce test functions.** We don't try to force $u(x)$ to satisfy the equation pointwise. Instead, we "test" it against a set of functions $v(x)$ called **test functions**. They live in the same function space as u, with the same boundary conditions (so $v(0) = v(1) = 0$). This space is called:

$$V = H_0^1(0,1) = \left\{ v \in L^2(0,1) \;\mid\; v' \in L^2(0,1),\; v(0) = v(1) = 0 \right\} \tag{23}$$

Deepening: Test function

A **Test Function** is not the solution itself, but an *arbitrary function* we use to "probe" whether the PDE is satisfied. Formally:

- A test function is usually called $v(x)$.
- It belongs to a certain function space V (often the same as the solution space).
- It must satisfy the **same boundary conditions** as the solution if those are homogeneous (like Dirichlet $u = 0$ on the boundary).

In our case:

$$V = H_0^1(0,1) = \left\{ v \in L^2(0,1) \mid v' \in L^2(0,1),\; v(0) = v(1) = 0 \right\}$$

❓ Why do we need them?

Because instead of requiring the PDE to hold **pointwise** (strong), we require it to hold **on average against all test functions**:

$$a(u,v) = F(v) \quad \forall v \in V$$

This means:

- If the equality holds for all possible test functions v, then the solution u must encode the correct behavior of the PDE.
- Test functions are like "magnifying glasses": by choosing different v, we check the equation in different ways.

Example 3: Analogy

Imagine we don't know the exact shape of a curve, but we can test it with different weights.

- Multiplying by $v(x)$ and integrating is like asking: "*How does the error of my solution project onto this particular pattern* $v(x)$*?*"
- If the error is orthogonal to *all* test functions, then the solution must be correct.

In practice (Finite Elements)

Later, when we do the **Galerkin method**, we restrict test functions v to be combinations of **basis functions** (hat functions, polynomials, ...). So instead of "all possible test functions", we only require the PDE to hold for a finite set of them. That's how we get a linear system $AU = f$.

Summary

A **test function** v is an arbitrary function from a suitable space (here $H_0^1(0,1)$). We multiply the PDE by v and integrate, to weaken the formulation. The condition "for all test functions" ensures the weak solution is equivalent to the strong one (if enough regularity). In finite elements, test functions become the basis functions of the discrete space.

2. **Multiply by a test function and integrate.** Multiply the PDE by $v(x)$ and integrate over $(0,1)$:

$$\int_0^1 (-u''(x)) \cdot v(x)\,\mathrm{d}x = \int_0^1 f(x) \cdot v(x)\,\mathrm{d}x \tag{24}$$

This is already "weaker", because we're not asking the PDE to hold pointwise, only **in an averaged sense** against all test functions.

❓ **Why multiply by a test function?** If we just write the residual of the PDE:

$$R(x) = -u''(x) - f(x) \tag{25}$$

Then the strong form requires $R(x) = 0$ **at every point**. That's too strict. Instead, we say:

– "We don't care if $R(x)$ is exactly 0 everywhere,

– We only care that $R(x)$ produces no effect when measured against any admissible test function $v(x)$".

So, multiplying by $v(x)$ is like "projecting the error" onto a shape.

– If for every possible shape v, the projection vanishes, then the error must be zero.

- If the residual had any "component" left, some test function would catch it.

Great Analogy: Imagine we don't know if a sound is silent. We pass it through every possible frequency filter (test functions). If all filters output 0, then the sound is really zero.

❓ **Why integrate?** Because multiplying alone just gives a function $R(x) \cdot v(x)$. To reduce it to a single **number** (a condition we can actually impose), we integrate over the domain:

$$\int_0^1 R(x) \cdot v(x) \, \mathrm{d}x = 0$$

Integration turns the **pointwise condition** into a **global/average condition**. It's like asking: "*what's the net effect of the residual when weighted by $v(x)$ across the whole domain?*". If this is 0 for all v, it forces the residual itself to be 0 in the weak sense.

Analogy: If we want to check if water in a pipe is really at 0 pressure, we don't measure every molecule. We place sensors (test functions) that average pressure over regions. If all averages say 0, the whole pipe is at 0.

3. **Integration by parts.** We move one derivative away from u (so we don't require u'' to exist, only u'):

$$\int_0^1 -u''(x) \cdot v(x) \, \mathrm{d}x = \Big[-u'(x) \cdot v(x) \Big]_0^1 + \int_0^1 u'(x) \cdot v'(x) \, \mathrm{d}x$$

The boundary term $\Big[-u'(x)v(x) \Big]_0^1$ vanishes, because $v(0) = v(1) = 0$. We are left with:

$$\int_0^1 u'(x) \cdot v'(x) \, \mathrm{d}x = \int_0^1 f(x) \cdot v(x) \, \mathrm{d}x \tag{26}$$

This is the **core weak formulation equation**.

❓ **Why do integration by parts?** Because, before this step, the integral contains the second derivative, $u''(x)$. That means the solution u must be **twice differentiable** (second derivate must exist in the classical sense). But in practice, with discontinuous sources or with finite element approximations, u'' might not exist everywhere.

Integration by parts transfers one derivate from u onto the test function v. That way, we only require u' to exist (so u just needs to be once differentiable) and v is smooth enough so that v' exists too. This relaxes the regularity (that's the whole point of the weak formulation).

Example 4: Integration by parts - Analogy

Imagine we want to test if someone's handwriting is smooth:

- Checking **second derivatives** is like asking for very fine, strict smoothness (hard!).

- By integrating by parts, we only ask them to be "once smooth", not "twice smooth".
- That's much easier to satisfy, and still captures the essence.

So in other words, integration by parts is necessary because it removes the second derivative on u, replacing it with first derivatives on both u and the test function. This reduces the regularity requirement, making the weak problem solvable in larger spaces (like H^1).

4. **Define bilinear and linear forms.** Writing integrals every time is messy. Mathematicians like to give names to these two operations:

   - The **left-hand side** looks like a special product between u and v. So we call it a **bilinear form** (because it's linear in u and in v separately):

     $$a(u, v) := \int_0^1 u'(x) \cdot v'(x) \, \mathrm{d}x \tag{27}$$

     It is like the **energy inner product** between u and v.

   - The **right-hand side** is an integral that only depends on v. So we call it a **linear functional** (linear in v):

     $$F(v) := \int_0^1 f(x) \cdot v(x) \, \mathrm{d}x \tag{28}$$

     It is like the **effect of the forcing** f measured against v.

   ❓ **Why do this?** Because once we define these two objects, the weak problem looks **super compact**:

   $$\text{Find } u \in V \text{ such that } a(u, v) = F(v) \quad \forall v \in V \tag{29}$$

   That's just a clean way of saying:

   $$\int_0^1 u'(x) \cdot v'(x) \, \mathrm{d}x = \int_0^1 f(x) \cdot v(x) \, \mathrm{d}x \quad \forall v \in V$$

   In simple terms, the weak problem says: "Find u such that, when tested against all possible v, the internal energy balance $a(u, v)$ equals the external forcing $F(v)$".

2.2.4 Galerkin formulation

Now we move from the **weak formulation** (still infinite-dimensional) to the **Galerkin formulation**, which is the bridge to something a computer can handle.

Galerkin idea

We have a PDE that is too difficult to solve point by point. Therefore, we obtain a weaker form that is more flexible but still **infinite-dimensional** (function space). We use the **Galerkin method**, which is a general approach for **approximating weak problems in finite-dimensional subspaces**.

Take the weak formulation (29, page 35):

$$a(u, v) = F(v) \quad \forall v \in V$$

Restrict to a **finite-dimensional subspace** $V_h \subset V$, or $V_h \subset H_0^1$:

$$a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h \tag{30}$$

So, instead of "all possible functions in V", we only allow functions in V_h. And instead of infinitely many test functions, we only check the condition for test functions in V_h. So the **Galerkin formulation** is:

$$\text{Find } u_h \in V_h \text{ such that } a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h \tag{31}$$

In theory (mathematical meaning)

This is a **projection**:

- The true solution u might not be in V_h.
- But we find the **closest approximation** u_h in that space, such that the **residual is orthogonal to V_h**.
- That's why Galerkin works: the error $u - u_h$ is "perpendicular" to all test functions in V_h.

> When we require the error $e = u - u_h$ to be orthogonal to the approximation space V_h:
>
> $$a(e, v_h) = 0 \quad \forall v_h \in V_h$$
>
> We are saying: "*the error has no component along any direction inside V_h*". That's the analogue of the vector case, the shortest path from a point to a line is along the perpendicular. So Galerkin says: *take the approximation u_h such that the error is perpendicular to the chosen subspace.*

So in theory, Galerkin is just **orthogonal projection** of the weak problem onto a finite subspace.

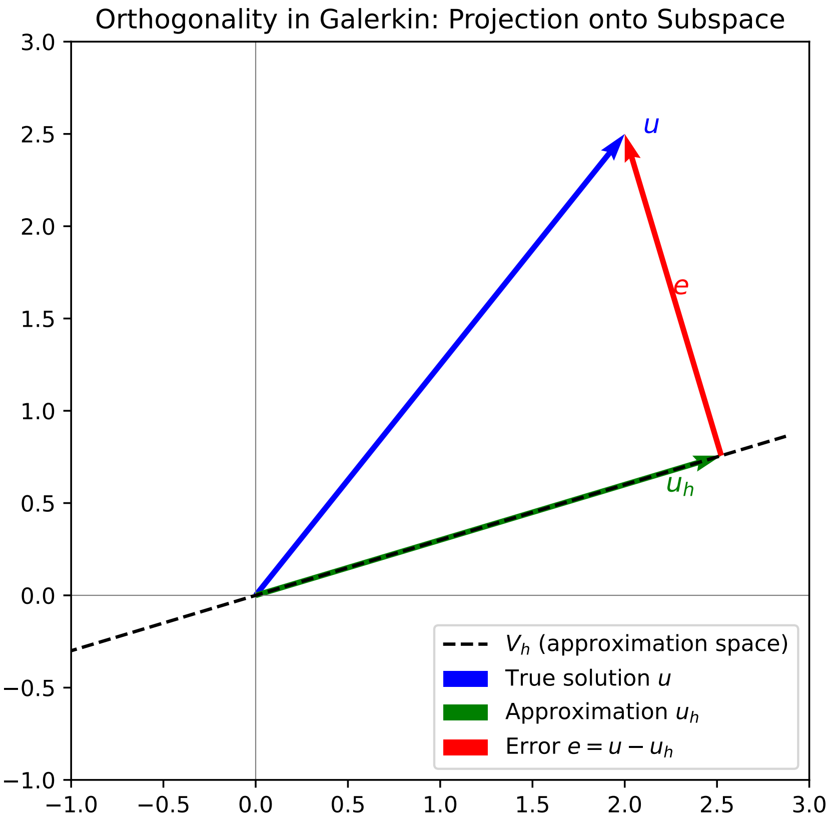

Figure 5: Orthogonality in Galerkin: Projection onto Subspace.

- The dashed line is our **approximation space** V_h (all the functions we can represent).
- The **blue vector** is the true solution u (not in V_h).
- The **green vector** is the Galerkin approximation u_h, which lies in V_h.
- The **red vector** is the error $e = u - u_h$.

Notice: the red error is **perpendicular** to the line V_h. That's exactly what "orthogonality of the residual" means; Galerkin forces the error to be perpendicular to our chosen approximation space, ensuring the *closest possible* approximation.

Remark 1: Orthogonality

Orthogonality in high school math

In Euclidean space $(\mathbb{R}^2, \mathbb{R}^3)$, two vectors are **Orthogonal** if their dot product is zero:

$$x \cdot y = 0 \tag{32}$$

That means they are "perpendicular". Here, orthogonality means *the shortest distance from a point to a line is the perpendicular.*

Perpendicular? Take two vectors in the plane:

$$u = (u_1, u_2), \quad v = (v_1, v_2)$$

Their dot product is:

$$u \cdot v = u_1 v_1 + u_2 v_2$$

Now, recall the formula with the angle θ between them:

$$u \cdot v = \|u\| \, \|v\| \cos(\theta)$$

So:

- If $u \cdot v > 0$, angle $< 90°$.
- If $u \cdot v < 0$, angle $> 90°$.
- If $u \cdot v = 0$, then $\cos\theta = 0$, then $\theta = 90°$.

That's exactly why "orthogonal" means "perpendicular": the inner product vanishes when vectors meet at a right angle.

Orthogonality in function spaces

Now, when we move from vectors to **functions**, the dot product is replaced by an **inner product**. For example, in $L^2(0,1)$ (square-integrable functions), the inner product is:

$$(u, v) = \int_0^1 u(x) \cdot v(x) \, \mathrm{d}x$$

So two functions u, v are **Orthogonal** if:

$$\int_0^1 u(x) \cdot v(x) \, \mathrm{d}x = 0$$

This is the function-space version of "perpendicular". Here, orthogonality means *the Galerkin solution u_h is the closest function in V_h to the true solution u, with distance measured in the PDE's energy norm.*

That's the **Galerkin formulation**.

- The exact solution u lives in V (infinite world).
- The approximate solution u_h lives in V_h (finite world).
- We require the weak form to hold for all test functions in V_h.

? Choosing V_h

The **true solution** lives in an *infinite-dimensional world* (all possible admissible functions). We cannot compute in infinity. So we **pick a smaller world**, a *finite-dimensional subspace* V_h. That smaller world is where Galerkin will search for the approximate solution u_h.

? What is V_h really? Think of V_h as our **toolbox of functions**. It's the set of shapes that our approximation is based on. For example:

- If we choose **straight lines** between mesh points, V_h are piecewise linear functions.
- If we choose **parabolas** on each mesh cell, V_h are piecewise quadratic functions.
- If we choose **sines and cosines**, V_h are trigonometric polynomials (spectral method).

In our laboratory, V_h is always:

$$V_h = \{\text{functions that are continuous and piecewise polynomials on a mesh, with } u = 0 \text{ at the boundary}\}$$

? Why do we care about which V_h? Because:

- A **bad choice** of V_h: we cannot approximate the solution well.
- A **good choice** of V_h: as we refine (smaller mesh, higher polynomial degree), the approximation converges to the true solution.

So the whole art of finite element is: *how do we design V_h so that it's expressive enough but still computable?*

- Domain: $(0, 1)$.
- Mesh: cut into N intervals.
- (V_h): functions that are continuous, zero at the ends, and linear on each small interval.
- Approximation space:

$$V_h = \left\{ v \in C^0([0,1]) : v|_K \in \mathbb{P}_1, \forall K \in \mathcal{T}_h, v(0) = v(1) = 0 \right\}$$

Deepening: Where does V_h come from?

From the previous sections, we had the following:

$$\text{Find } u \in V = H_0^1(0,1) \quad \text{such that } a(u,v) = F(v) \quad \forall v \in V$$

So the exact solution lives in:

$$V = H_0^1(0,1) = \left\{ v \in L^2(0,1) : v' \in L^2(0,1), v(0) = v(1) = 0 \right\}$$

This space is **infinite-dimensional** (all functions with square-integrable derivative, vanishing at endpoints).

We want a **finite-dimensional** subspace $V_h \subset V$. So we must impose two things: (1) **boundary condition** (keep $v(0) = v(1) = 0$, so that $V_h \subset H_0^1$), (2) **finite dimension** (instead of "all functions", choose a restricted family of functions easy to handle).

1. **Choose a *mesh*.** Split $[0,1]$ into small subintervals:

   $$\mathcal{T}_h = \{K_1, K_2, \dots, K_N\}, \quad K_i = [x_{i-1}, x_i]$$

   Here $h = \max_i |K_i|$ is the **mesh size**.

2. **Choose a *polynomial degree*.** On each element K, we allow only polynomials of degree $\leq r$.

   - If $r = 1$: linear functions on each element.
   - If $r = 2$: quadratics.
   - And so on.

   This is written as $v|_K \in \mathbb{P}_r$.

3. **Impose continuity.** Finite element functions are not just piecewise polynomials: they must be **globally continuous** (otherwise they wouldn't belong to H_0^1). So we require $v \in C^0([0,1])$.

4. **Impose boundary conditions.** Finally, we enforce $v(0) = v(1) = 0$ to respect the homogeneous Dirichlet boundary conditions.

So the space is:

$$V_h = \left\{ v \in C^0([0,1]) : v|_K \in \mathbb{P}_r, \ \forall K \in \mathcal{T}_h, \ v(0) = v(1) = 0 \right\}$$

If $r = 1$, that's piecewise linear FE functions. If $r = 2$, piecewise quadratic, and so on. In summary, we obtain the formula for V_h by restricting the infinite weak space $V = H_0^1$ in 4 steps: first, we *mesh* the domain. Then, on each mesh cell, we allow only low-degree polynomials. Next, we glue them together with continuity. Finally, we impose boundary conditions. That's exactly the standard definition of a finite element space.

2.2.5 Finite Element formulation

We already derived:

- **Weak formulation** (page 32), find $u \in V$ such that:

$$a(u, v) = F(v) \quad \forall v \in V$$

  Where:

  - $V = H_0^1(\Omega)$
  - $a(u, v) = \displaystyle\int_0^1 u'(x)\, v'(x)\, \mathrm{d}x$
  - $F(v) = \displaystyle\int_0^1 f(x) v(x)\, \mathrm{d}x$

- **Galerkin formulation** (page 36), restrict to a finite-dimensional space $V_h \subset V$:

$$a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h$$

So the question is: **how to choose V_h?**

2.2.5.1 Constructing the finite-element space V_h

We want to approximate the infinite-dimensional space. The weak formulation lives in $V = H_0^1(\Omega)$, which is infinite-dimensional. To make it computable, Galerkin requires a **finite-dimensional subspace** $V_h \subset V$. But how do we describe V_h? We need a concrete way.

Mesh gives structure. A **Mesh** is a way to divide our domain Ω (here $(0, 1)$) into **smaller, simple pieces** called **elements**.

- In 1D: elements are **intervals**.

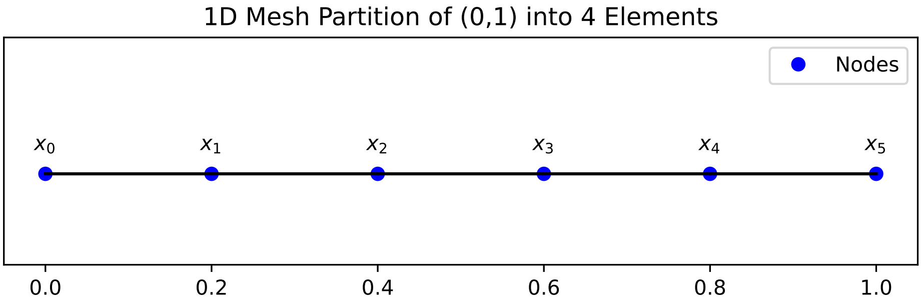

Figure 6: An example of a 1D mesh for $(0, 1)$, partitioned into 4 elements (so 5 internal nodes plus the two boundaries). Each segment is an **element**, and the blue dots are the **nodes** $x_0, x_1, \ldots, x_5$.

- In 2D: elements are usually **triangles** or **quadrilaterals**.

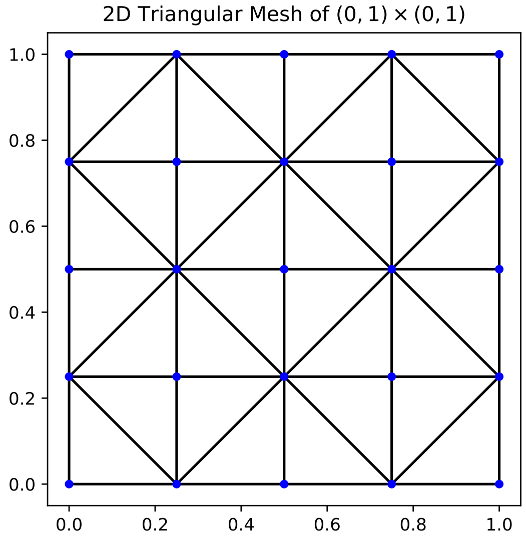

Figure 7: An example of a 2D triangular mesh of the unit square $(0,1) \times (0,1)$. The blue dots are the **nodes**. The black lines are the **edges of the triangular elements**. Each small triangle is one **finite element**.

- In 3D: elements are **tetrahedra** or **hexahedra (cubes)**.

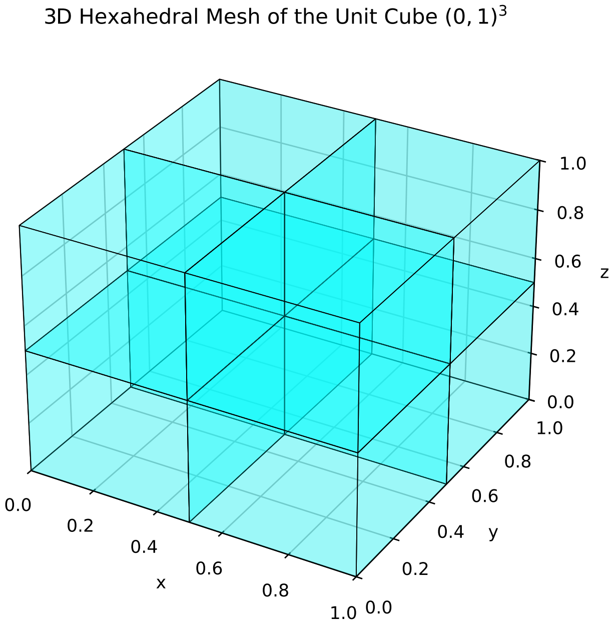

Figure 8: An example of a 3D mesh of the unit cube $(0,1)^3$, divided into smaller hexahedral elements (little cubes). Each transparent cyan block is one **element**. The black lines are the **edges of the mesh**.

These elements are “building blocks” on which we define our basis functions.

Formally, a **Mesh** (or triangulation) $\mathcal{T}_h$ of a domain Ω is a collection of elements K such that:

1. $\displaystyle\bigcup_{K \in \mathcal{T}_h} K = \Omega$ (the elements cover the whole domain).
2. Two elements only touch on their boundary; they don’t overlap inside.
3. Each element has a “small” size related to the **mesh parameter** h, typically the maximum diameter of all elements:

   $$h = \max_{K \in \mathcal{T}_h} \operatorname{diam}(K)$$

   In 1D, where $K = [x_{i-1}, x_1]$, its **diameter** is just its length:

   $$\operatorname{diam}(K) = |x_i - x_{i-1}|$$

   In 2D, if K is a triangle, its diameter is the length of the **longest edge**. In 3D, if K is a tetrahedron, its diameter is the longest distance between two of its vertices. So, diam(K) gives a size measure of that element.

It is called a “mesh” because, when viewed in two or three dimensions, its elements form a grid or net, much like the mesh of a fishing net or the pixels of an image.

In short, a **mesh is the discretization of the geometry of our domain into small, simple elements**. It is the foundation of finite elements. Without a mesh, we wouldn’t know *where* to place our basis functions.

❓ So, what exactly is a mesh parameter?

We define the **Mesh Parameter** h (or **mesh size**) as:

$$h = \max_{K \in \mathcal{T}_h} \operatorname{diam}(K) \tag{33}$$

This is a **global measure**: it takes the largest element in the mesh. If the mesh is uniform (all elements equal size), then h is just the common element size. If the mesh is non-uniform (some small, some large elements), then h tells us the size of the **worst (largest) element**.

❓ Why does the Mesh Parameter matter?

- **Accuracy**: Smaller h → more elements → better approximation of the true solution.
- **Computational cost**: Smaller h → bigger system of equations → more memory and CPU time.
- **Convergence theory**: Error estimates are usually written like:

  $$\|u - u_h\| \le C h^p$$

  Where p depends on the polynomial degree r. So the quality of the mesh directly controls how fast we converge to the true solution.

For example, suppose $\Omega = (0, 1)$, partitioned into $N+1$ intervals. Each interval has length $h = \dfrac{1}{(N+1)}$. If $N = 9$, then $h = 0.1$. If $N = 99$, then $h = 0.01$, so the mesh is 10 times finer.

1D Poisson Problem

Our laboratory has the domain $\Omega = (0, 1)$. We take a number of mesh elements of $N + 1 = 20$. Then we have nodes:

$$x_0 = 0,\ x_1 = h,\ x_2 = 2h, \ldots, x_{20} = 1$$

With $h = \dfrac{1}{N+1} = \dfrac{1}{20}$. Each element is $K_i = [x_{i-1}, x_i]$. So the mesh is simply the collection:

$$\mathcal{T}_h = \{[0, h]\,,\ [h, 2h]\,,\ [2h, 3h]\,, \ldots, [1 - h, 1]\}$$

This breaks the continuous problem into "small, simple pieces". So the domain is now "atomic pieces" K_i. On each element we can define **local polynomials**.

Why do we approximate with polynomials on each piece?

There are four reasons:

- Because polynomials are **simple and computable**. Polynomials have **explicit formulas** for derivatives and integrals. In FEM we need to compute integrals like:

  $$\int_{K_i} u_h'(x) \cdot v_h'(x)\, \mathrm{d}x$$

  And with polynomials these are straightforward.

- Because polynomials are **good local approximators**. By Taylor's theorem, any smooth function can be approximated locally by a polynomial. On a small element K_i, the solution $u(x)$ doesn't change much, so a low-degree polynomial (linear, quadratic) already gives a good fit. The smaller the element (smaller h), the better a polynomial of fixed degree approximates the true solution.

- Because they "glue together" nicely. If we define one polynomial per element, we can impose **continuity** at shared nodes. This gives us **global continuous functions** built from local building blocks. With polynomials, enforcing continuity at nodes is natural (hat functions are 1 at one node, 0 at others).

- Because they give sparse algebraic systems. Each polynomial (hat function) has **local support**: it is nonzero only on 2 neighboring elements (in 1D, for $r = 1$). This locality produces a **sparse stiffness matrix** (tridiagonal in 1D, banded in higher dimensions). Sparse systems are efficient to store and solve, essential for HPC.

Example 5: Physical analogy

Imagine we cut a bent stick (the real solution) into small pieces:

- On each piece, we approximate it with a simple ruler (straight line is a linear polynomial).
- The smaller the pieces, the more the rulers together resemble the original curve.
- If we want higher accuracy per piece, we can replace the ruler with a curved template (quadratic, cubic polynomial).

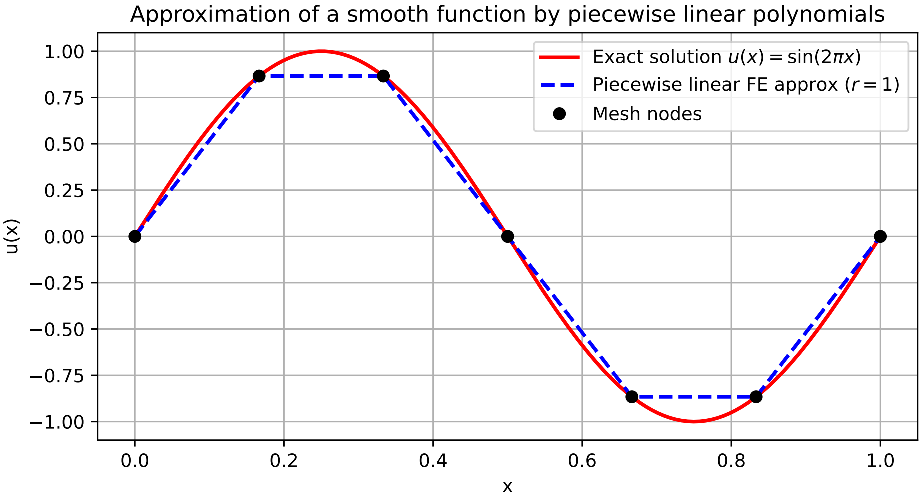

Figure 9: Graphical 1D example showing how a smooth curve can be approximated by piecewise polynomials of degree one (straight lines) on the mesh.

- The **red curve** is the exact function $u(x) = \sin(2\pi x)$.
- The **blue dashed line** is the finite element approximation with $r = 1$: piecewise linear segments between mesh nodes.
- The **black dots** are the mesh nodes, where the FE solution matches the exact one.

As we refine the mesh (N larger, h smaller), the blue curve hugs the red one more closely.

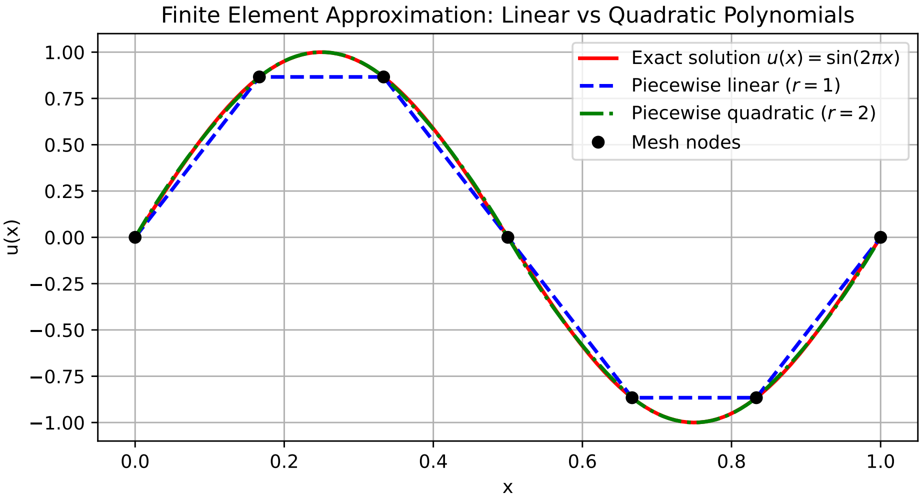

Figure 10: Graphical 1D example showing how a smooth curve can be approximated by piecewise polynomials of degree one (straight lines) on the mesh.

- The **red curve** is the exact function $u(x) = \sin(2\pi x)$.
- The **blue dashed curve** is the piecewise linear approximation ($r = 1$): straight line segments.
- The **green dash-dot curve** is the piecewise quadratic approximation ($r = 2$): parabolas on each element.

We can see that the quadratic elements hug the sine curve much better, especially between the nodes. However, despite the increased precision, the system size, cost, and DoFs (Degrees of Freedom, or the number of equations to be solved) all increase.

Define piecewise polynomials

For a given polynomial degree r:

$$X_h^r(\Omega) = \left\{ v_h \in C^0([0,1]) \ : \ v_h|_{K_i} \in \mathbb{P}_r, \ \forall i \right\} \tag{34}$$

Meaning:

- $v_h \in C^0([0,1])$: every function in $X_h^r(\Omega)$ must be **continuous** across the whole domain. No "jumps" are allowed between one element and the next. So when we glue local polynomials together, they must match at the common endpoints (nodes).

  Reason: the weak formulation requires $u_h \in H^1(\Omega)$, and H^1 functions must be continuous.

- $v_h|_{K_i} \in \mathbb{P}_r$: where $|_{K_i}$ means "restricted to the element K_i". On each mesh element $K_i = [x_{i-1}, x_i]$, the function is a polynomial of degree $\leq r$. For example:

  - If $r = 1$, on each element K_i, $v_h(x) = a + bx$ (a straight line).

  - If $r = 2$, then $v_h(x) = a + bx + cx^2$ (a parabola).

  So globally, v_h is "piecewise polynomial": it can change slope or curvature from one element to the next, but it remains continuous.

- $\forall i$: this condition applies **on every element of the mesh**. We cannot have a polynomial on some elements and something else on others, the rule is uniform across the mesh.

- Restricted to each element K_i, it is a polynomial of degree at most r.

- For $r = 1$, that means **straight lines on each interval**.

Impose boundary conditions

From the laboratory problem:

$$u(0) = 0, \quad u(1) = 0$$

The solution must vanish at the endpoints of the domain. However, the continuous weak space already encodes this. From the weak formulation:

$$\begin{aligned} u \in V &= H_0^1(\Omega) \\ &= \big\{ v \in H^1(\Omega) : v(0) = v(1) = 0 \big\} \end{aligned}$$

So in the continuous problem, we don't enforce the boundary conditions by extra conditions; they are **baked into the function space** itself. So far, we constructed:

$$X_h^r(\Omega) = \{ v_h \in C^0([0,1]) : v_h|_{K_i} \in \mathbb{P}_r \}$$

But these functions don't necessarily vanish at the boundary. To fix this, we take:

$$V_h = X_h^r(\Omega) \cap H_0^1(\Omega) \tag{35}$$

Meaning:

- Functions must belong to $X_h^r(\Omega)$ (continuous, piecewise polynomials).

- **And** they must vanish at the boundary, just like in $H_0^1(\Omega)$.

In other words, restrict the finite element space so that all functions automatically vanish at the boundary. In practice, this removes the boundary basis functions and leaves only the internal degrees of freedom.

📘 Choose a basis (Lagrangian "hat" functions)

Now we need a basis of V_h. A **Basis** is like a set of "Lego bricks" from which we can build any object in a space. The basis gives us a **small finite set of functions** from which all others in the space can be built. From the finite element space V_h, we can find N **basis functions** $\varphi_1, \dots, \varphi_N$ such that:

$$u_h(x) = \sum_{j=1}^{N} U_j \cdot \varphi_j(x) \quad \forall u_h \in V_h \tag{36}$$

Where the coefficients U_j are just numbers and the φ_j are the "bricks" (basis functions). In 1D, for $r = 1$, these φ_j are the hat functions. Without a basis, the space is just an abstract definition.

Once we expand the unknown u_h in this basis, the PDE problem reduces to solving for the coefficients U_j. This is how we go from an infinite-dimensional PDE to a finite-dimensional **linear system** $AU = f$ (**from functions to algebra**).

❓ **Why choose a Lagrangian basis?** There are many possible bases, but the **Lagrangian nodal basis** is the most natural for FEM. Its key properties:

1. **Interpolation property**. Each basis function φ_j satisfies:

   $$\varphi_j(x_i) = \delta_{ij}$$

   It equals **1 at its own node** and **0 at all others**. This makes coefficients U_j directly equal to the **nodal values** of the solution:

   $$u_h(x_i) = U_i$$

   So the unknowns are literally "the solution at the mesh nodes".

2. **Local support**. Each φ_j is nonzero only on a small neighborhood of nodes (two elements in 1D). This leads to a **sparse matrix**, which is essential for computational efficiency.

3. **Intuitive geometry**. The basis functions look like little "hats" (for $r = 1$) or "arches" (for $r = 2$), easy to visualize and implement. In higher dimensions, they become pyramids (2D triangles) or tents (3D tetrahedra).

4. **Implementation in FEM libraries**. Packages like `deal.II`, `FEniCS`, `gmsh`, etc. all rely on nodal (Lagrangian) bases as the standard choice. They are the simplest to code, especially for assembling element matrices and evaluating values at quadrature points.

❓ **Could we use another basis?** **Yes**, hierarchical bases, modal bases (e.g., Legendre polynomials), spectral methods (global polynomials). **But** those are more complex, less intuitive, and not the standard starting point. So for our laboratory, the goal is clarity and efficiency, that's why we stick to **Lagrangian hat functions**.

Summary

We successfully built the finite element space V_h. Here is a list of the steps we took:

1. **Start from the weak space** (page 32). The PDE requires solutions in:

   $$V = H_0^1(\Omega)$$

   i.e. continuous functions with square-integrable derivatives, vanishing on the boundary. This space is infinite-dimensional.

2. **Partition the domain (mesh)** (page 41). Divide $\Omega = (0, 1)$ into $N + 1$ small intervals:

   $$\mathcal{T}_h = \{K_i = [x_{i-1}, x_i] : i = 1, \dots, N+1\}, \quad h = \frac{1}{N+1}$$

3. **Define local polynomial shape** (page 44). On each element K_i, we decide that admissible functions are **polynomials of degree** r:

   $$v_h|_{K_i} \in \mathbb{P}_r$$

   Where $r = 1$ are straight lines, $r = 2$ are parabolas, etc.

4. **Enforce continuity** (page 46). Require that these piecewise polynomials join continuously across elements:

   $$X_h^r(\Omega) = \left\{v_h \in C^0([0, 1]) : v_h|_{K_i} \in \mathbb{P}_r \ \forall K_i\right\}$$

5. **Impose boundary conditions** (page 47). Since the PDE requires $u(0) = u(1) = 0$, we restrict to functions that vanish at the endpoints:

   $$V_h = X_h^r(\Omega) \cap H_0^1(\Omega)$$

6. **Choose a basis** (page 48). Pick a convenient set of basis functions for V_h.

   - Standard choice: **Lagrangian nodal basis** (hat functions).
   - They are 1 at one node, 0 at all others, and supported only on neighboring elements.

   Thus, any approximate solution is written as:

   $$u_h(x) = \sum_{j=1}^{N_h} U_j \cdot \varphi_j(x)$$

   With unknown coefficients U_j (the degrees of freedom).

References

[1] Quarteroni Alfio Maria. Numerical methods for partial differential equations. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

Index