Numerical Methods for Partial Differential Equations - Notes - v0.4.0

260236

October 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.

For the laboratory, I created a simple repository containing all the code implementations and CMake files necessary to compile the code in a CLion environment:

NM4PDE GitHub Repository



Contents

1	Basic Concepts			4
	1.1		ematical Models and Scientific Computing	5
	1.2		ential Models and PDEs	7
		1.2.1	ODEs	9
		1.2.2	PDE, boundary value problem in 1D	10
		1.2.3	PDE, initial and boundary value problem in 1D	11
		1.2.4	PDE, boundary value problem in multidimensional domains	12
		1.2.5	PDE, initial and boundary value problem in multidimen-	
		1.2.0	sional domains	12
		1.2.6	Classification of PDEs	13
	1.3		rical Methods	15
	1.4	From	Mathematical to Numerical Problem	16
		1.4.1	The Mathematical Problem (MP)	16
		1.4.2	The Numerical Problem (NP)	17
2		orator	*	22
	2.1		luction	22
	2.2		for Poisson 1D	25
		2.2.1 $2.2.2$	What is the Poisson Equation?	25
			Problem definition	29
		2.2.3 $2.2.4$	Weak formulation	33 37
			Galerkin formulation	
		2.2.5	Finite Element formulation	42
			2.2.5.1 Constructing the finite-element space V_h	42
		0.0.0	2.2.5.2 From V_h to the Discrete Problem	51
		2.2.6	Implementation in deal.II	55
			2.2.6.1 Install & Setup	55
			2.2.6.2 Program Structure	58
			2.2.6.3 General Structure	59
			2.2.6.4 Header File	64
			2.2.6.5 Setup: turning math objects into deal.II objects 2.2.6.6 Assemble: compute A and f	69 79
			2.2.6.7 Solve phase	93
			2.2.6.8 Output phase	96
			2.2.6.9 Extensions	99
	2.3	Poisso		102
	2.5	2.3.1	- v	102
		2.3.1 $2.3.2$	· ·	102
		2.3.2 $2.3.3$		109
		2.3.4	9	119
		2.3.4 $2.3.5$		125
		2.3.6	V1 1 00 0	129
		2.0.0	DATOMONO. I difetioni di sei	123
In	\mathbf{dex}		1	.35

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.

¹The **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).

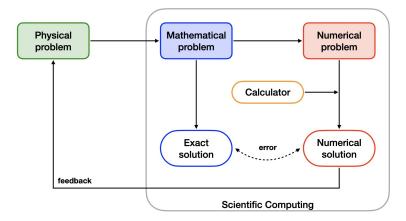


Figure 1: Scientific Computing.

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:

$$\mathcal{P}(u;g) = 0$$
 differential equation (mathematical problem) (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*.

Deepening: PDE

A Partial Differential Equation is the mathematical language used to describe how continuous physical quantities (temperature, velocity, pressure, stress, concentration, electromagnetic field, etc.) evolve in space and time. They express: conservation laws (mass, momentum, energy), constitutive relations (material response), and external influences (forces, sources, boundaries). Everything around us is governed by PDEs:

- Physics: Maxwell's equations (electromagnetism), Schrödinger equation (quantum mechanics), Navier-Stokes equations (fluid dynamics).
- Engineering: Heat equation (thermal analysis), wave equation (vibrations, acoustics), elasticity equations (structural mechanics).
- **Biology**: Reaction-diffusion equations (pattern formation, population dynamics), diffusion equations (spread of substances).
- Finance: Black-Scholes equation (option pricing).

They are the continuous models of reality.

? Why discretization of PDEs? Analytic solutions exist only for

simple geometries or constant coefficients. Most real-world problems, like airflow around a car, stress in a bridge, or temperature in a turbine, are too complex. So we **discretize** the PDE to convert it into a **finite-dimensional algebraic system** that we can solve numerically:

$$Au_h = b$$

That's the bridge from **physics** to **mathematics** to **computation**. Discretization (via FEM, FDM, FVM, etc.) transforms "infinitely many unknowns" into a **finite but huge system** that captures the essential physics at chosen resolution.

? Why solve PDEs numerically? (the *purpose*) Because solving PDEs means simulating the real-world without physically building it. Instead of running an expensive or dangerous experiment, we compute the solution on a computer. Some examples:

- **Aerospace**: Simulate airflow over a new aircraft design to optimize aerodynamics before building prototypes.
- Climate Science: Model global climate systems to predict future climate change scenarios.
- **Biomedical Engineering**: Simulate blood flow in arteries to design medical devices.
- Automotive: Crash simulations to improve vehicle safety features
- Energy: Model heat transfer in nuclear reactors for safety analysis.

Every time we discretize and solve a PDE, we create a digital experiment.

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(t) 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:

$$\begin{cases} \frac{\mathrm{d}u}{\mathrm{d}t}(t) = f(t, u(t)) & t \in I\\ u(t_0) = u_0 \end{cases}$$
(2)

Where:

- $I = (t_0, t_f] \subset \mathbb{R}$ 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** $\mathbf{u} = \mathbf{u}(t)$, where $\mathbf{u} = (u_1, \dots, u_m) \in \mathbb{R}^m$, with $m \geq 1$. The first order Cauchy problem reads: find $\mathbf{u} : I \subset \mathbb{R} \to \mathbb{R}^m$ such that:

$$\begin{cases} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}(t) = \mathbf{f}(t, \mathbf{u}(t)) & t \in I \\ \mathbf{u}(t_0) = \mathbf{u}_0 \end{cases}$$

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:

$$\begin{cases}
\frac{\mathrm{d}^{2} u}{\mathrm{d}t^{2}}(t) = f\left(t, u(t), \frac{\mathrm{d}u}{\mathrm{d}t}(t)\right) & t \in I \\
\frac{\mathrm{d}u}{\mathrm{d}t}(t_{0}) = v_{0} \\
u(t_{0}) = u_{0}
\end{cases}$$
(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:

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

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

Example 2

For example, the unknown function $u\left(x,t\right)$ 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 $u_0\left(x\right)$. Finally, the bar is subject to a heat source of linear density $f\left(x,t\right)$.

1.2.4 PDE, boundary value problem in multidimensional domains

The Poisson problem (equation 4, page 10) can be **extended in multidimensional domains** $\Omega \subset \mathbb{R}^d$, with d=2,3; the solution is $u=u(\mathbf{x})$, where $\mathbf{x}=(x_1,\ldots,x_d)^T\in\mathbb{R}^d$. This leads to the following Poisson problem with (homogeneous) Dirichlet boundary conditions: find $u:\Omega\subset\mathbb{R}^d\to\mathbb{R}$ such that:

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

Where:

• The Laplace operator:

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

- 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 11) reads: find $u: \Omega \times I \to \mathbb{R}$ such that:

$$\begin{cases} \frac{\partial u}{\partial t} - \mu \Delta u = f & \mathbf{x} \in \Omega, t \in I \\ u(\mathbf{x}, t) = 0 & \mathbf{x} \in \partial \Omega, t \in I \\ u(\mathbf{x}, t_0) = u_0(\mathbf{x}) & \mathbf{x} \in \Omega \end{cases}$$
(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 $u = u(\mathbf{u}, t)$, that is the PDE solution:
- Spatial coordinates $\mathbf{x} = (x_1, \dots, x_d)^T \in \mathbb{R}^d$ 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:

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

Where $p_1, \ldots, p_d, p_t \in \mathbb{N}$ 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.

√x Classification

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

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

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*:

- $\bullet \ \ x_1 = x$
- \bullet $x_2 = y$

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.
- \bullet Parabolic PDE describes wave propagation phenomena with $\underline{\rm infinite}$ 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.

$$\mathcal{P}\left(u;g\right)=0$$
 PDE (mathematical problem)
 \downarrow numerical method
 $\mathcal{P}_{h}\left(u_{h};g_{h}\right)=0$ approximate PDE (numerical problem)

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	$approximate\ solution\ { m of\ a\ PDE}$
$(u-u_h)$	\boldsymbol{error} (quantified and/or estimated; tends to zero if h is smaller)
h	$\begin{array}{l} \textbf{discretization parameter} \ (\downarrow \text{ smaller } h, \text{ better approximation}; \ \uparrow \text{ higher } h, \text{ poor approximation}) \end{array}$
$\mathcal{P}_h\left(u_h;g_h\right)=0$	approximate PDE (numerical problem)
g_h	$m{approximation}$ of the $m{data}$ g
\mathcal{P}_h	${\it characterization}$ 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:

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

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;
- \bullet *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 16). 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 17) and the Round-Off error e_r (equation 15) concur to determine the Computational error e_c :

$$e_c := e_h + e_r = (u - y_h) + (y_h - \hat{u}_h) = u - \hat{u}_h$$
 (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 $|e_c| \approx Ch^p$ 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 $|e_{c1}| \cong Ch_1^p$ and $|e_{c2}| \cong Ch_2^p$
 - Noticing that $\frac{|e_{c1}|}{|e_{c2}|} = \left(\frac{h_1}{h_2}\right)^p$

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 loglog 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.

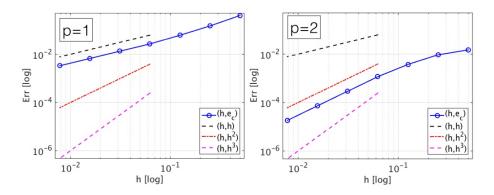


Figure 2: Graphical estimation of the convergence order p of a NP: computational errors $|e_c|$ vs h.

? 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:

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

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

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

Let highlights the main differences:

• Definition:

- Consistent. Consistency requires that as the discretization parameter h tends to zero $\lim_{h\to 0}$, 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:

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

Is consistent:

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

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).

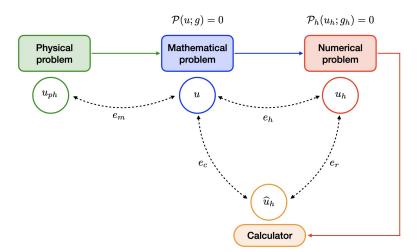


Figure 3: Physical (PP), Mathematical (MP), and Numerical (NP) problems. Corresponding solutions $(u_{ph}, u, u_h, \text{ and } \widehat{u}_h)$ and errors (model $e_m = u_{ph} - u$, truncation $e_h = u - u_h$, round-off $e_r = u_h - \widehat{u}_h$, and computational $e_c = e_h + e_r$ errors).

2 Laboratory

For the laboratory, I created a simple repository containing all the code implementations and CMake files necessary to compile the code in a CLion environment:

NM4PDE GitHub Repository



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.

2 Laboratory 2.1 Introduction

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 multiphysics 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.

2 Laboratory 2.1 Introduction

✗ 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:



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

1 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.

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \cdots$$
 (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:

$$-u''(x) = 0 \implies u''(x) = 0$$

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).

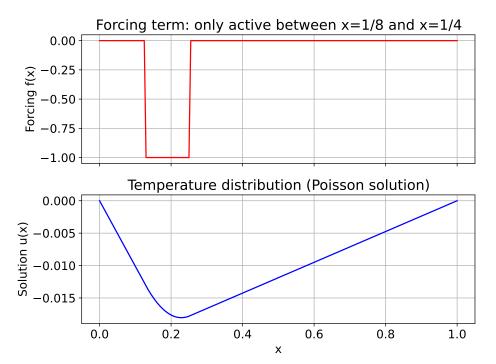


Figure 4: Visual explanation of the 1D Poisson problem. Top graph (red) is the forcing term f(x). It is **zero everywhere**, except between $x = \frac{1}{8}$ and $x = \frac{1}{4}$, where it is negative (-1). That means only in that region we have a "sink" of heat. The **bottom graph (blue)** is the solution 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 27. 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:

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

- 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** \rightarrow fix the **value** of the solution at the boundary.
- 2. Neumann \rightarrow fix the derivative/flux of the solution at the boundary.
- 3. Robin (mixed) \rightarrow fix a combination of value and derivative.

A Dirichlet condition prescribes directly the solution value:

$$u(x) = g(x)$$
 on $\partial\Omega$ (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:

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

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:

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

There is a **negative source term** (a "sink" of heat, or a downward force density) only in the interval $\left(\frac{1}{8}, \frac{1}{4}\right]$. 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:

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

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:

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

X 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.

⁵ 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):

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

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

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

The boundary term $\left[-u'(x)v(x)\right]_0^1$ 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.

mulation).

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.

Example 4: Integration by parts - Analogy

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

This relaxes the regularity (that's the whole point of the weak for-

- 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**:

Find
$$u \in V$$
 such that $a(u, v) = F(v) \quad \forall v \in V$ (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.

Ogalerkin 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 36):

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

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:

Find
$$u_h \in V_h$$
 such that $a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h$ (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.

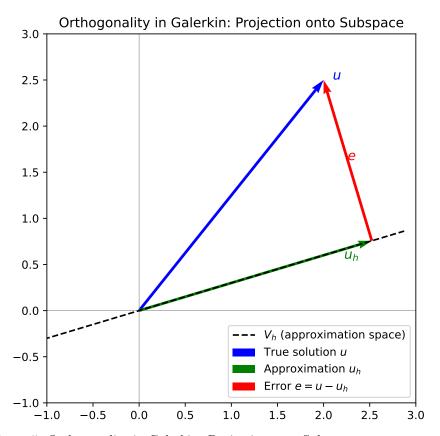


Figure 5: Orthogonality in Galerkin: Projection onto Subspace.

- 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.

2 Laboratory

Remark 1: Orthogonality

Crthogonality 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 $< 90^{\circ}$.
- If $u \cdot v < 0$, angle $> 90^{\circ}$.
- If $u \cdot v = 0$, then $\cos \theta = 0$, then $\theta = 90^{\circ}$.

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

X¹ 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 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).
- \bullet Mesh: cut into N intervals.
- (V_h) : functions that are continuous, zero at the ends, and linear on each small interval.
- Approximation space:

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

Deepening: Where does V_h come from?

From the previous sections, we had the following:

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

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:

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

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 33), find $u \in V$ such that:

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

Where:

$$- V = H_0^1(\Omega) - a(u, v) = \int_0^1 u'(x) v'(x) dx - F(v) = \int_0^1 f(x)v(x) dx$$

• Galerkin formulation (page 37), 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**.

1D Mesh Partition of (0,1) into 4 Elements

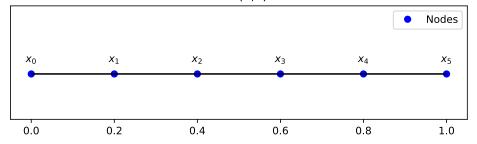


Figure 6: An example of a 1D mesh for (0,1), partitioned into 4 elements (so 5 internal nodes plus the two boundaries). Each segment is an **element**, and the blue dots are the **nodes** x_0, x_1, \ldots, x_5 .

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

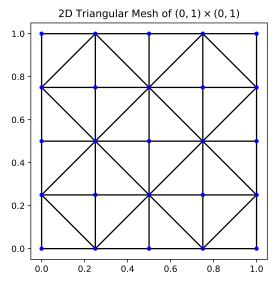


Figure 7: An example of a 2D triangular mesh of the unit square $(0,1) \times (0,1)$. The blue dots are the **nodes**. The black lines are the **edges of the triangular elements**. Each small triangle is one **finite element**.

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

3D Hexahedral Mesh of the Unit Cube $(0, 1)^3$

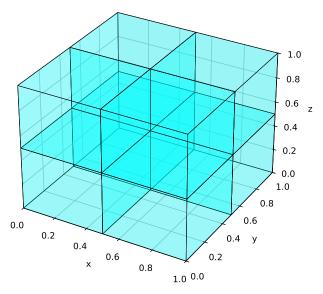


Figure 8: An example of a 3D mesh of the unit cube $(0,1)^3$, divided into smaller hexahedral elements (little cubes). Each transparent cyan block is one **element**. The black lines are the **edges of the mesh**.

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. $\bigcup_{K \in \mathcal{T}_h} K = \Omega$ (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:

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

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 $h \to \text{more elements} \to \text{better approximation of the true solution.}$
- Computational cost: Smaller $h \to \text{bigger system of equations} \to \text{more memory and CPU time.}$
- Convergence theory: Error estimates are usually written like:

$$||u - u_h|| \le Ch^p$$

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 $h = \frac{1}{(N+1)}$. 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:

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

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

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

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:

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

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).

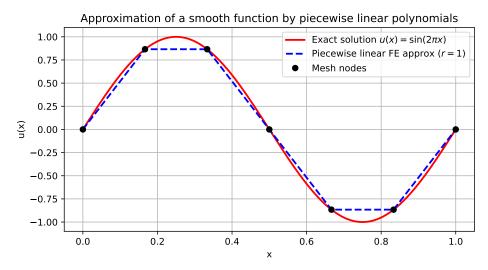


Figure 9: Graphical 1D example showing how a smooth curve can be approximated by piecewise polynomials of degree one (straight lines) on the mesh.

- 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.

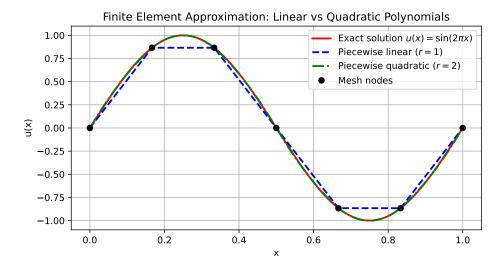


Figure 10: Graphical 1D example showing how a smooth curve can be approximated by piecewise polynomials of degree one (straight lines) on the mesh.

- 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:

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

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

= $\{v \in H^1(\Omega) : v(0) = v(1) = 0\}$

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 $\varphi_1, \ldots, \varphi_N$ such that:

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

⁵ 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 33). 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 42). 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 45). 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 47). Require that these piecewise polynomials join continuously across elements:

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

5. Impose boundary conditions (page 48). 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 49). 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).

2.2.5.2 From V_h to the Discrete Problem

We want:

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

With:

•
$$a(u,v) = \int_0^1 \mu(x) \cdot u'(x) \cdot v'(x) dx$$

•
$$F(v) = \int_0^1 f(x) \cdot v(x) dx$$

Pick a basis $\{\varphi_j\}_{j=1}^{N_h}$ of V_h . Write the unknown as:

$$u_h(x) = \sum_{j=1}^{N_h} U_j \,\varphi_j(x) \tag{37}$$

Where:

- U_j are the unknown coefficients (nodal values in the Lagrangian case).
- φ_j are the known basis functions (hat functions for r=1).

Now, we plug into the weak form. Take $v_h = \varphi_i$, one basis function. Then:

$$a(u_h, \varphi_i) = F(\varphi_i)$$

Now compute the left-hand side:

$$a(u_h, \varphi_i) = \int_0^1 \mu(x) \cdot \left(\sum_{j=1}^{N_h} U_j \cdot \varphi_j'(x) \right) \cdot \varphi_i'(x) dx$$

Since the integral is linear:

$$a(u_h, \varphi_i) = \sum_{j=1}^{N_h} U_j \cdot \int_0^1 \mu(x) \cdot \varphi_j'(x) \cdot \varphi_i'(x) dx$$

We now define:

$$A_{ij} = \int_0^1 \mu(x) \cdot \varphi_j'(x) \cdot \varphi_i'(x) \, \mathrm{d}x \tag{38}$$

$$f_i = \int_0^1 f(x) \cdot \varphi_i(x) \, \mathrm{d}x \tag{39}$$

So the weak form equation becomes:

$$\sum_{i=1}^{N_h} A_{ij} \cdot U_j = f_i, \quad i = 1, \dots, N_h$$
 (40)

■ Matrix form

That's exactly the linear system:

$$AU = f \tag{41}$$

Where:

• $A = (A_{ij})$ is the **Stiffness Matrix**.

What is the Stiffness Matrix? In the finite element method, when we discretize a PDE like the Poisson problem, the weak form introduces integrals of derivatives of the basis functions. These integrals become the entries of the stiffness matrix (see above):

$$A_{ij} = \int_{\Omega} \mu(x) \cdot \nabla \varphi_j(x) \cdot \nabla \varphi_i(x) \, \mathrm{d}x \tag{42}$$

Where A is called the Stiffness Matrix. It is symmetric positive definite (SPD) if the bilinear form is symmetric and coercive. Its size is $N_h \times N_h$, where N_h is the number of degrees of freedom (basis functions).

? Why is it called stiffness? The name comes from structural mechanics. Originally, FEM was used to model elastic bars, beams, membranes. The relation "force = stiffness \times displacement" in elasticity correspond to the matrix equation:

$$Ku = f$$

Where:

- u are displacement at nodes;
- -f are nodal forces;
- K is the stiffness matrix, encoding how "resistant" (stiff) the structure is to deformation.

Even though we now use FEM for general PDEs, the name stuck.

X Composition of the Stiffness Matrix

- **Diagonal entries** A_{ii} : measure how strongly a degree of freedom (a node) resists deformation, i.e. the "self-stiffness".
- Off-diagonal entries A_{ij} : measure the coupling between neighboring basis functions (nodes). If two nodes share an element, the integral is nonzero; otherwise, it's zero.

Thus, the stiffness matrix is:

- **Sparse**: only nearby nodes interact.
- Structured: for 1D linear elements, it is tridiagonal.
- Conditioning: its conditions number grows like h^{-2} (with mesh size h).

■ Key properties (why it's nice computationally)

- Symmetric: $A_{ij} = A_{ji}$
- Positive definite (for $\mu > 0$): $U^T A U > 0$ for all nonzero vectors U.
- **Sparse/Local**: φ_i overlaps only with a few neighbors, so only a few nonzeros per row.
- Conditioning scales with mesh size (roughly $\kappa(A) \sim h^{-2}$ in 1D), motivating preconditioners.

In summary, the stiffness matrix shows how the domain resists changes.

• $U = (U_1, ..., U_{N_h})^T$ are the unknown nodal values.

? How U is defined. We write the approximate solution in the finite element space V_h :

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

Where:

- $\{\varphi_j\}_{j=1}^{N_h}$ are the basis functions of V_h .
- U_i are scalars: degrees of freedom (DoFs).

Thus, $U = (U_1, U_2, \dots, U_{N_h})^T$.

 \blacksquare Nodal values. Because we chose the Lagrangian (nodal) basis, each φ_j has the property:

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

That means:

$$U_i = u_h(x_i)$$

i.e. the unknown coefficients are literally the approximate solution at the mesh nodes (internal nodes only, since boundary ones are fixed to zero).

* Physical interpretation

- In structural mechanics: U are nodal displacements.
- In heat problems: *U* are nodal temperatures.
- In Poisson problems (our lab): U are just nodal values of the solution.

In other words, solving AU=f gives us the "best" nodal approximation of the exact PDE solution.

• $f = (f_1, \dots, f_{N_h})^T$ is the **load vector**.

 \blacksquare Definition of the load vector. For each basis function φ_i , the load vector entry is:

$$f_i = F(\varphi_i) = \int_{\Omega} f(x) \cdot \varphi_i(x) dx$$

So the load vector f collects the effect of the forcing term f(x) applied to the PDE, projected onto the finite element basis.

? Why it appears. The weak formulation was:

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

With:

$$F(v_h) = \int_{\Omega} f(x) \, v_h(x) \, dx$$

When we take $v_h = \varphi_i$, we get exactly f_i . Thus, the right-hand side of the system AU = f is the vector with entries f_i .

> Physical meaning

- In mechanics: f represents external forces applied at the nodes.
- In heat transfer: f represents **heat sources** distributed in the domain.
- In general PDEs: it's how the forcing term f(x) excites the system.

So just like A encodes "resistance", f encodes "applied load".

- **Local-to-global composition.** Just like the stiffness matrix, the load vector is built **element by element**:
 - On each element K, compute

$$f_{\alpha}^{(K)} = \int_{K} f(x) \cdot N_{\alpha}(x) \, \mathrm{d}x$$

Where N_{α} are local shape functions.

- Then assemble into the global vector f.

In summary, the load vector f is the FEM representation of the source term f(x). Each entry f_i measures how much the source excites the basis function φ_i . Physically, it is the "external input" applied to the system.

Now we have a **discrete**, **well-posed algebraic system**. This is exactly what FEM libraries (like deal.II) are built to assemble and solve. The next step is coding.

2.2.6 Implementation in deal.II

2.2.6.1 Install & Setup

At the Politecnico di Milano, professors suggest installing mk, a project that provides environment modules for scientific computing libraries and packages with portable x86-64 Linux libraries. Developed at the MOX Center at the Politecnico di Milano, it is based on Lmod, a tool for managing user environments dynamically. To install it, we can simply follow the instructions in the README on GitHub.

There are other common ways to install deal.II. Follow this guide to install deal.II (either natively or via Docker, etc.):

Getting deal.II







Now we move to the project skeleton. We assume a structure of this type:

```
1 lab01/
2   CMakeLists.txt
3   src/
4   lab-01.cpp
5   Poisson1D.cpp
6   Poisson1D.hpp
```

The CMakeLists.txt file:

```
# Minimum CMake version required.
cmake_minimum_required(VERSION 3.12)
3 # Project name and language.
4 project (01_poisson_1d LANGUAGES CXX)
# Set C++ standard to C++11.
7 set (CMAKE_CXX_STANDARD 11)
8 # Require C++11 standard.
9 set (CMAKE_CXX_STANDARD_REQUIRED "ON")
# Set default build type to Release.
12 if (NOT CMAKE_BUILD_TYPE OR "${CMAKE_BUILD_TYPE}" STREQUAL "")
      set(CMAKE_BUILD_TYPE "Release" CACHE STRING "" FORCE)
13
14 endif()
15 message(STATUS)
message(STATUS "Build type: ${CMAKE_BUILD_TYPE}")
17 message(STATUS)
if("${CMAKE_BUILD_TYPE}" STREQUAL "Debug")
      add_definitions(-DBUILD_TYPE_DEBUG)
19
20 endif()
21
# Locate MPI compiler.
23 find_package(MPI REQUIRED)
24 set(CMAKE_CXX_COMPILER "${MPI_CXX_COMPILER}")
25
# Locate Boost.
find_package(Boost 1.72.0 REQUIRED
          COMPONENTS filesystem iostreams serialization
28
          HINTS ${BOOST_DIR} $ENV{BOOST_DIR} $ENV{mkBoostPrefix})
```

Where:

• Build type (Release or Debug):

```
# Set default build type to Release.
if(NOT CMAKE_BUILD_TYPE OR "${CMAKE_BUILD_TYPE}" STREQUAL "")
set(CMAKE_BUILD_TYPE "Release" CACHE STRING "" FORCE)
endif()
message(STATUS)
message(STATUS "Build type: ${CMAKE_BUILD_TYPE}")
message(STATUS)
if("${CMAKE_BUILD_TYPE}" STREQUAL "Debug")
add_definitions(-DBUILD_TYPE_DEBUG)
endif()
```

Sets **default build type** to release (faster and optimized).

• MPI:

```
# Locate MPI compiler.

prind_package(MPI REQUIRED)

set(CMAKE_CXX_COMPILER "${MPI_CXX_COMPILER}")
```

MPI (Message Passing Interface) is the standard for distributed parallelism (multiple nodes, HPC clusters). deal.II uses MPI for parallel FEM solvers. Even if the first lab is serial, MPI will still be useful in the future.

• Boost:

```
# Locate Boost.
find_package(Boost 1.72.0 REQUIRED

COMPONENTS filesystem iostreams serialization
HINTS ${BOOST_DIR} $ENV{BOOST_DIR} $ENV{mkBoostPrefix}

message(STATUS "Using the Boost-${Boost_VERSION} configuration found at ${Boost_DIR}")
message(STATUS)
include_directories(${Boost_INCLUDE_DIRS})
```

Boost is a large C++ library, like a "standard library extension". The version ≥ 1.72 is required with the components:

- filesystem: utilities to traverse folders, read/write files.
- iostreams: extensions for input/output streams.
- serialization: save/load objects in binary/text formats.

It is needed because deal. II itself depends on Boost for many utilities.

- find_package(deal.II ...): Locates our deal.II installation and imports its CMake configuration.
- deal_ii_initialize_cached_variables(): Initializes compiler flags, include paths, etc.
- Compiler flags:

```
# Add useful compiler flags.

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wfloat-conversion -
Wmissing-braces -Wnon-virtual-dtor")
```

Adds extra warnings to help catch common FEM bugs:

- --Wfloat-conversion: warns when ints are silently converted to floats (dangerous in mesh indexing).
- - Wmissing-braces: warns about missing braces in initializers.
- Wnon-virtual-dtor: warns if a base class has virtual methods but no virtual destructor (classic memory leak risk).
- add_executable(...): Defines the actual program we want to compile (Lab 01 solver).
- deal_ii_setup_target(...): Connects our target to deal.II's libraries, so we can use all its functionality.

Remark 2: CMakeLists

A CMakeLists.txt file is essentially the *recipe* that CMake uses to configure and build our project. We can think of it as a "build script" written in a declarative mini-language.

In the context of deal.II labs, the CMakeLists.txt file plays a central role because:

- It tells **CMake** the **project name** and which **languages** we are using (C++ in our case).
- It specifies the **minimum CMake version** required.
- It imports the deal.II library and makes its headers and compiled code available to our project.
- It defines which source files should be compiled into an executable.
- It links our executable against deal.II (and possibly other libraries like MPI or LAPACK if enabled).

2.2.6.2 Program Structure

In the first Poisson 1D laboratory, we will create three source files:

- lab-01.cpp
- Poisson1D.cpp
- Poisson1D.hpp

This split follows a classic C++ organization:

- Header (.hpp): contains the **class declaration**. It's the "blueprint": all member variables and method signatures are written here. No implementation details.
- Implementation (.cpp): contains the **class implementation**: actual code for each method declared in the header. Here is where we'll find the implementation.
- Driver (lab-01.cpp): contains the main() function. This is the entry point: it creates an instance of Poisson1D, calls run(), and handles exceptions.

This program structure mirrors the workflow of the Finite Element Method (FEM):

- Declare the **problem** (class declaration = PDE + parameters).
- Implement the **steps** (methods = weak form \rightarrow discrete system).
- Run the **pipeline** (main = assemble + solve).

So the program is not "just code files". It's intentionally structured to mirror the mathematical procedure we derived in the written part of the lab.

2.2.6.3 General Structure

We stopped in theory with:

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

Restricted to the finite element space V_h , this became:

$$\sum_{j=1}^{N_h} U_j a(\varphi_j, \varphi_i) = F(\varphi_i), \quad i = 1, \dots, N_h$$

Which is the linear system:

$$Au = f$$

With:

- $A_{ij} = a(\varphi_i, \varphi_i)$ (stiffness matrix)
- $f_i = F(\varphi_i)$ (load vector)
- $u = (U_1, \ldots, U_{N_b})^T$ (vector of unknown coefficients).

So mathematically, we "only" need to assemble A, assemble f, and solve Au=f.

Mapping math objects to deal. II objects

We wrap everything in a class Poisson1D to keep the code modular (easy to reuse in future labs, 2D or 3D). Inside the class we store:

- Discretization parameters: number of elements N, polynomial degree r.
- Mesh: the triangulation of $\Omega = (0,1) \to \text{Triangulation} < \dim >$
- Basis functions & polynomial degree: Finite Element space definition
 → FE_Q<dim>(r)
- Integration of bilinear or linear forms: quadrature rules

 $\rightarrow \mathtt{QGauss} \small{<} \mathtt{dim} \small{>} \ \mathrm{and} \ \mathtt{FEValues}$

What is a quadrature rule? A Quadrature Rule is a numerical method used to approximate the definite integral of a function. It involves selecting specific points (called nodes) within the integration interval and assigning weights to these points. The integral is then approximated by a weighted sum of the function values at these nodes.

Common quadrature rules include the trapezoidal rule, Simpson's rule, and Gaussian quadrature (which is the most widely used). In the context of finite element methods (FEMs), quadrature rules are essential for accurately integrating functions over elements, especially when dealing with polynomial basis functions.

Why Gaussian quadrature and not trapezoidal or Simpson? Gaussian quadrature is preferred in FEM because it provides the highest degree of accuracy for a given number of integration points. It is particularly effective for integrating polynomial functions, which are commonly used as basis functions in FEM. In contrast, the trapezoidal and Simpson's rules may require more points to achieve the same level of accuracy, making them less efficient for this purpose.

With "highest degree of accuracy" we mean that, for n integration points, Gaussian quadrature can exactly integrate polynomials of degree up to 2n-1, while the trapezoidal rule is exact for polynomials of degree 1 and Simpson's rule is exact for polynomials of degree 3. This efficiency is crucial in FEM, where computational resources are often limited.

- System matrix (sparse): $A \rightarrow \text{SparseMatrix} < \text{double} >$
- Sparsity structure: pattern of nonzeros → SparsityPattern
- Load vector & Solution vector: $f, u \rightarrow Vector < double >$

We use the static member dim to make the code dimension-independent (if we later change dim = 2, we can reuse the same structure for 2D). However, every mathematical entity from the variational formulation has a "natural home" in deal.II.

Note: It's normal that this mapping feels not immediate the first time. It is like learning a new programming framework:

- At the math level, we already know what we need (mesh, basis, test functions, bilinear/linear forms, linear system).
- At the code level, deal.II already has these structures, but the names and abstractions are new.

However, once learned, the same structure applies to all future PDE solvers in deal.II.

■ Why split into setup, assemble, solve, output?

After mapping mathematical objects to deal.II objects, we can create a powerful plan. Typically, the best solution is a 4-step workflow:

- 1. **Setup.** Before any computation, we need to **initialize** all the structures: mesh, FE space, DoFs (Degree of Freedoms), sparsity pattern, vectors. This is a one-time preparation step.
- 2. **Assemble**. Local integrals (element stiffness matrices, local RHS) must be computed and inserted into global A and f. This mirrors the theory: local basis function \rightarrow global system.
- 3. Solve. The heart of the problem: solve the linear system Au = f. In theory, "find u in V_h ". In code, apply a solver (solvers that have already been seen in the Numerical Linear Algebra course, such as Conjugate Gradient, GMRES, etc.).
- 4. Output. Once we have u, we need to use it: visualize (ParaView), compute errors, post-process quantities. This is not math anymore, but engineering practice: without output, the solution is useless.

The next sections will cover each step in detail.

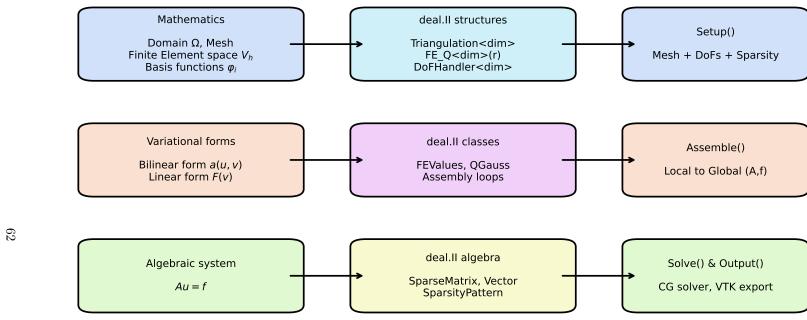


Figure 11: Map of translation from math \rightarrow code \rightarrow implementation flow.

Regarding the figure on the previous page:

- Mathematics (left column). This is what we derived in the theory part (weak → Galerkin → FEM). It is pure math, independent of any programming language.
- deal.II Objects (middle column). Here we see the software representation of each math object.

1. Geometry & discretization

- Triangulation<dim> o the mesh Ω_h
- FE_Q<dim>(r) \rightarrow basis functions of degree r
- DoFHandler<dim> → numbering of the unknowns (degrees of freedom, DoFs)

2. Assembly tools

- FEValues, QGauss \rightarrow integration over elements (numerical quadrature)
- Assembly loops \rightarrow compute local A^K , f^K and scatter them into the global system

3. Algebraic structures

- SparseMatrix<double> \rightarrow the stiffness matrix A
- Vector<double> \rightarrow the load vector f and solution u
- SparsityPattern → structure of nonzeros (saves memory and computation)

Each mathematical ingredient has a deal. II class implementing it.

• Solver Pipeline (right column). Finally, the workflow of the finite element solver. This is the *four-step structure*: setup, assemble, solve and output. This is the **concrete implementation**. It correspond 1-to-1 with the previous two columns.

2.2.6.4 Header File

The following header defines a **Poisson 1D solver** with the classic FEM workflow:

```
setup() \rightarrow assemble() \rightarrow solve() \rightarrow output()
```

Each member maps 1-to-1 to a mathematical object.

```
#ifndef POISSON1D_HPP
2 #define POISSON1D_HPP
#include <deal.II/base/function.h>
5 #include <deal.II/base/quadrature_lib.h>
7 #include <deal.II/dofs/dof_handler.h>
8 #include <deal.II/fe/fe_q.h>
#include <deal.II/grid/tria.h>
#include <deal.II/lac/sparsity_pattern.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/vector.h>
16 #include <memory>
17
18 using namespace dealii;
19
20 /**
   * Minimal Poisson 1D solver skeleton.
21
* Only the core fields from our math 
ightarrow deal.II mapping.
23 */
24 class Poisson1D
25 {
26 public:
    // Physical dimension (1D, 2D, 3D)
27
    static constexpr unsigned int dim = 1;
28
29
30
    // \mu(x) - diffusion coefficient (Lab 01: \mu \equiv 1).
    class DiffusionCoefficient : public Function < dim >
31
32
33
      // Constructor.
34
      DiffusionCoefficient() = default;
36
      // Evaluation.
37
      double value(const Point < dim > &, const unsigned int = 0) const
38
      override {
39
        return 1.0;
      }
40
    };
41
    // f(x) - forcing term (Lab 01: -1 on (1/8, 1/4], 0 elsewhere).
43
44
    class ForcingTerm : public Function < dim >
45
    public:
46
47
      // Constructor.
      ForcingTerm() = default;
48
49
50
      // Evaluation.
      double value(const Point < dim > &p, const unsigned int = 0) const
51
       override {
        const double x = p[0];
      return (x > 1.0/8.0 \&\& x \le 1.0/4.0) ? -1.0 : 0.0;
53
```

```
54
    };
55
56
    // Constructor: N = (N+1) elements on [0,1], r = FE degree.
57
    Poisson1D(const unsigned int &N_, const unsigned int &r_)
58
     : N(N_), r(r_) {}
59
60
    // FEM pipeline (defined later).
61
    62
    void assemble(); // local integrals \rightarrow global A,f and apply
63
     Dirichlet
                    // linear solver (CG)
64
    void solve();
65
    void output() const; // VTK write
66
67 protected:
   // Discretization parameters
68
    const unsigned int N; // N+1 elements
69
    const unsigned int r; // polynomial degree
70
71
    // Problem data
72
    DiffusionCoefficient diffusion_coefficient;
73
    ForcingTerm
74
                         forcing_term;
75
    // Geometry & FE space
76
                                        mesh:
    Triangulation < dim >
77
    std::unique_ptr<FiniteElement<dim>> fe;
                                                      // e.g., FE_Q<
     dim>(r)
79
    std::unique_ptr<Quadrature<dim>>
                                        quadrature;
                                                      // e.g., QGauss
      <dim>(r+1)
    DoFHandler < dim >
                                        dof_handler;
80
81
    // Algebraic objects: A u = f
82
                     sparsity_pattern;
    SparsityPattern
83
    SparseMatrix < double > system_matrix;
    Vector < double >
                        system_rhs;
85
86
    Vector < double >
                         solution;
87 };
89 #endif //POISSON1D_HPP
```





Public section

- static constexpr unsigned int dim = 1;. We hard-code the physical dimension to 1. It simplifies the lab (no templates), while preserving deal.II's dimension-aware types (Triangulation<dim>, DoFHandler<dim>, ...). If we later want 2D/3D, we'd typically turn the class into template<int dim> and reuse the same structure.
- Problem data as Function<dim>

```
class DiffusionCoefficient : public Function < dim > { ... };
class ForcingTerm : public Function < dim > { ... };
```

deal. II algorithms expect coefficients and source terms as "Function objects", a polymorphic interface for values/derivatives at points. We override value() function to return $\mu(x)$ and f(x). However, for this lab:

$$-\mu(x) \equiv 1$$

 $-f(x) = -1$ on $\left(\frac{1}{8}, \frac{1}{4}\right]$, and 0 elsewhere.

Keeping them as members makes it easy to pass them to assembly and boundary utilities (e.g., VectorTools, MatrixTools).

? Function objects? In deal.II, coefficients, source terms, boundary data, exact solutions, etc. are modeled by classes derived from dealii:: Function<dim>. Function<dim> class is abstract and provides some default implementations. Since it is abstract, we cannot create objects directly using this class. Instead, we need to create a derivative, such as the ForcingTerm class. Function<dim> class exposes virtual methods like:

- value(const Point<dim>&, unsigned int component=0)
- gradient(const Point<dim>&, unsigned int component=0)
- vector_value(...), vector_gradient(...), etc.

Because they're virtual, algorithms (assembly, interpolation, error post-processing) can accept a **reference to the base type** Function<& and work with *any* concrete subclass we provide; that's **runtime polymorphism**. Here we have override only the value function, but we could override also other methods if we need it.

Constructor

```
Poisson1D(const unsigned int &N_, const unsigned int &r_)
: N(N_), r(r_) {}
```

- \mathbb{N} controls the **mesh resolution** (here: the implementation uses N+1 elements on [0,1]).
- r is the FE polynomial degree (so V_h is \mathbb{P}_r continuous).

Both are stored as const so they're fixed for the life of the object.

- Pipeline methods. This mirrors the four canonical FEM phases.
 - setup(). Build everything static: mesh, FE, DoFs, sparsity, allocate
 A, f, u.
 - assemble(). Compute local element matrices/vectors and scatter into global A, f; apply Dirichlet.
 - solve(). Run a linear solver (CG here, as A is SPD² for Poisson).
 - output(). Write results (VTK) for ParaView, etc.

²A real square matrix A is Symmetric Positive Definite (SPD) if $A^T = A$ and $x^T A x > 0$ for all nonzero vectors x. Consequences: all eigenvalues are real and strictly positive; A admits a Cholesky factorization $A = R^T R$; it defines an inner product $\langle x, y \rangle_A = x^T A y$; Krylov solvers like Conjugate Gradient (CG) are applicable and guaranteed to converge. In FEM: stiffness matrices from coercive elliptic problems with homogeneous Dirichlet BCs are SPD.

**Protected section (the state of our solver)

• Discretization parameters

```
const unsigned int N; // N+1 elements on [0,1]
const unsigned int r; // FE degree (P_r)
```

These determine mesh size and polynomial degree; together they define V_h .

• Problem data (instances)

```
DiffusionCoefficient diffusion_coefficient;
ForcingTerm forcing_term;
```

Kept as objects (not pointers). They're cheap and provide an easy API during assembly and BCs.

• Geometry & FE

```
Triangulation < dim > mesh;

td::unique_ptr < FiniteElement < dim > fe;

td::unique_ptr < Quadrature < dim > quadrature;

DoFHandler < dim > dof_handler;
```

- Triangulation<dim>: the mesh \mathcal{T}_h .
- FiniteElement<dim> is an abstract base; FE_Q<dim> derives from it. Storing a unique_ptr<FiniteElement<dim> lets we choose the concrete FE at runtime (here FE_Q<1>(r), but later we could switch type/degree without changing the class definition).

fe is our finite element type, it tells deal.II

- * What the **local basis functions** are on each cell K;
- * Where the **degrees of freedom (DoFs)** live (endpoints, midpoints, ...);
- * How to get shape values $\phi_i(x_q)$ and gradients $\phi_i'(x_q)$ at quadrature points;
- * How many **DoFs per cells** there are, etc.
- Same idea for Quadrature<dim>: we'll usually pick QGauss<dim> (r+1) in setup().
- DoFHandler<dim> binds FE to the mesh and assigns global DoF indices (the algebraic unknowns).

Lifetime ordering: class members are destroyed in the **reverse** order of their declaration in C++. Here, dof_handler is declared after mesh, so it is destroyed **before** mesh \rightarrow this is the safe order required by deal.II (the handler must not outlive the mesh).

• Algebraic structures

```
SparsityPattern sparsity_pattern;
SparseMatrix < double > system_matrix;
Vector < double > system_rhs;
Vector < double > solution;
```

- SparsityPattern captures where nonzeros can appear (from FE connectivity).
- SparseMatrix<double> holds the global stiffness matrix A.
- Vector<double>: RHS f and solution u.

A Strict style nit: many teams avoid using namespace dealii; in headers to prevent symbol pollution; for a lab it's okay, but in production we'd prefer dealii:: prefixes or put using in the .cpp.

2.2.6.5 Setup: turning math objects into deal. II objects

The setup process has four parts:

1. Mesh + boundary IDs

```
// Create the mesh.
2 {
      printf("Initializing the mesh\n");
      GridGenerator::subdivided_hyper_cube(
          mesh, N + 1, 0.0, 1.0, true
      printf(
             Number of elements = %d\n",
          mesh.n_active_cells()
10
      ):
11
      // Write the mesh to file.
12
      const std::string mesh_file_name = "mesh-" +
13
          std::to_string(N + 1) + ".vtk";
14
15
      const GridOut grid_out;
      std::ofstream grid_out_file(mesh_file_name);
      grid_out.write_vtk(mesh, grid_out_file);
17
18
      printf(" Mesh saved to %s\n", mesh_file_name.c_str());
```

The mesh is the scaffold that turns the continuous PDE on Ω into a finite-dimensional problem we can actually compute.

- Math picture (what we want)
 - Domain: the closed interval [0, 1].
 - Partition (mesh): split [0,1] into small sub-intervals (cells/elements). If we have N cells, the vertices are N+1 points: $x_0 = 0 < x_1 < \cdots < x_M = 1$.
 - Boundary: the two endpoint x = 0 (left) and x = 1 (right).
- deal. II objects (how it represents that picture). Triangulation

 dim> is the mesh container (cells + faces + links). The name

 "triangulation" comes from numerical PDE jargon: any partition of

 a domain into simple elements (segments in 1D, triangles/quads in

 2D, tets/hexes in 3D) is often called a "triangulation", even when the

 cells aren't literal triangles.
 - What does a Triangulation store? Geometry and topology of the mesh. It stores vertices (coordinates) and cells (their vertex indices), plus who's adjacent to whom (links). This is the scaffold on which we place basis functions and do integrals cell-by-cell.

In 1D:

- A **cell** is an interval $[x_i, x_{i+1}]$;
- A face is a 0-D endpoint (vertex).
- **Boundary faces** are endpoints located on the boundary of the domain (here, just x = 0 and x = 1).

In summary, it creates a 1D mesh on [0,1] with N+1 elements (so N+2 nodes for r=1). The last boolean parameter in the subdivided_hyper_cube invocation means that the colorize parameter is set to

true (colorize=true), and assigns boundary id 0 to x = 0 and id 1 to x = 1. We'll use these IDs to impose Dirichlet in assemble().

? Why VTK export now (write_vtk)? So we can see the mesh and quickly catch wrong counts/IDs.

X API Meaning

- (a) subdivided_hyper_cube (mesh, M, a, b, colorize). In 1D, this creates a line segment [a,b] split into M equal cells. Here, M=N+1, a=0.0 and b=1.0. So we get N+1 intervals.
- (b) What does "active" mean? Triangulations can be adaptively refined, creating a tree of parent/child cells. "Active cells" are the leaves (the current mesh we compute on). Immediately after creation, every cell is active.
- (c) colorize = true (crucial for boundary IDs)
 - If false, all boundary faces get the same boundary_id (typically 0). That's OK for "Dirichlet everywhere = 0", but useless for different BCs on different sides.
 - If true, deal.II assigns distinct IDs per "side" of the hypercube:
 - In 1D:
 * ID 0: left endpoint x = a (the "minus" face along x).
 * ID 1: right endpoint x = b (the "plus" face along x).
 In 2D:
 * ID 0: -x (left)
 - * ID 1: +x (right)
 - * ID 2: -y (bottom)
 - * ID 3: +y (top)
 - In 3D:
 - * ID 0: -x
 - * ID 1: +x
 - * ID 2: -y
 - * ID 3: +y
 - * ID 4: -z
 - * ID 5: +z

Because the IDs are distinct, we could assign different functions to 0 and 1 if we wanted mixed boundary conditions.

Why is it called colorize? In the context of meshing, "colorize" refers to labeling parts with distinct tags. deal.II uses integer tags called boundary_id. Setting colorize=true tells the generator to color each side with a different boundary_id so we can target them individually for boundary conditions.

2. Finite element + quadrature

```
1 // Initialize the finite element space.
       printf("Initializing the finite element space\n");
3
       // Finite elements in one dimension are obtained with
       // the FE_Q class (which for higher dimensions represents
       // hexahedral finite elements).
      // In higher dimensions, we must use FE_Q for
       // hexahedral elements or FE_SimplexP for tetrahedral
      \ensuremath{//} elements. They are both derived from
       \ensuremath{//} FiniteElement, so that the code is generic.
       fe = std::make_unique<FE_Q<dim>>(r);
      printf(
14
15
              Degree
                                            = %d\n",
           fe->degree
16
      );
17
      printf(
18
           " DoFs per cell
                                            = %d n'',
19
           fe->dofs_per_cell
20
      ):
21
22
       // Construct the quadrature formula of the
23
       // appropriate degree of exactness.
24
25
       // This formula integrates exactly the mass
       // matrix terms (i.e. products of basis functions).
26
       quadrature = std::make_unique < QGauss < dim >> (r + 1);
27
28
       printf(
29
30
              Quadrature points per cell = %d\n",
           quadrature ->size()
31
      ):
32
```

? Finite element (FE_Q): What does the math need? We decide to approximate the solution in a space:

$$V_h = \{ v \in C^0([0,1]) \mid v|_K \in \mathbb{P}_r \text{ for every cell } K \in \mathcal{T}_h \}$$

So on each cell K we use polynomials of degree r, and we glue cells together with continuity at the nodes.

- ? Finite element (FE_Q): How deal.II represents that
 - FE_Q<dim>(r): Lagrange \mathbb{P}_r elements on a (hyper)cube cell (Q = hypercube elements, segments in 1D, quads in 2D, hexes in 3D). In 1D, the reference cell is [0,1]. The local basis are the Lagrange shape functions $\{\phi_i\}_{i=0}^r$ that are 1 at the one node and 0 at the others (nodal basis).

In other words, $FE_Q(r)$ says "on each cell, use r+1 nodal polynomials and glue cells continuously".

- DoFs per cell in 1D: fe -> dofs_per_cell = r + 1. For example, with r = 1, 2 DoFs per cell (endpoints); r = 2 3 DoFs per cell (endpoints + midpoint).
- Global DoFs with M cells and degree r in 1D:

$$N_{\text{dofs}} = M \cdot r + 1$$

Because adjacent cells share nodes. In our code, M=N+1, so for r=1: $N_{\text{dofs}}=(N+1)\cdot 1+1=N+2$.

? Quadrature (QGauss): What does the math need? For Poisson, on each cell *K* we must compute:

• Stiffness:
$$A_{ij}^K = \int_K \mu \cdot \phi_j' \cdot \phi_i' \, \mathrm{d}x$$

• Load:
$$f_i^K = \int_K f \, \phi_i \, \mathrm{d}x$$

These are integrals of polynomials (or polynomials \times coefficient) over K. We approximate them with Gauss-Legendre quadrature.

? Quadrature (QGauss): How deal.II represents that.

In deal.II, QGauss is a numerical integration rule: it tells deal.II which points x_q to sample on a cell and which weights w_q to multiply by, so that an integral is approximated by a weighted sum

$$\int_{K} g(x) dx \approx \sum_{q=1}^{n_q} g(x_q) \cdot w_q \quad \text{(on cell } K)$$

Specifically, Gauss-Legendre with n_q points is exact for all polynomials up to degree $2n_q - 1$ on an interval. That's why we like it for FEM on (hyper)cube cells.

QGauss controls how we integrate on each cell during assembly (and post-processing): i.e., how many sample points inside the cell we use, and with which weights, to approximate integrals.

- Triangulation / mesh: how many cells, their size/shape, boundary tags. This sets where the domain is cut and how fine the grid is.
- Finite element FE_Q(r): which basis we use on each cell (degree r, continuity). This sets how "complex" the polynomial we can represent per cell is.
- Quadrature QGauss(n_q): how many sample points per cell we use to compute the integrals that build A and f. This sets how accurately we evaluate those integrals, not the mesh, not the basis.

? What does QGauss actually give us? For an interval (our 1D cell), QGauss<1>(n_q) returns n_q reference points and weights that are exact for all polynomials up to degree $2n_q - 1$.

⚠ Undershooting n_q . If we pick too few points, we'll get consistency/accuracy issues (quietly!). The standard rule is generally to use r + 1.

3. DoFHandler: global numbering of unknowns

```
1 // Initialize the DoF handler.
      printf("Initializing the DoF handler\n");
3
      // Initialize the DoF handler with the mesh
      // we constructed.
      dof_handler.reinit(mesh);
      // "Distribute" the degrees of freedom.
      // For a given finite element space,
      // initializes info on the control variables
11
12
         (how many they are, where
      // they are collocated, their "global indices", ...).
13
      dof_handler.distribute_dofs(*fe);
14
15
      printf(" Number of DoFs = %d\n", dof_handler.n_dofs());
16
17 }
```

- What problem it solves (conceptually). After we pick the FE space V_h with FE_Q<1>(r) and we have a mesh \mathcal{T}_h , we need a **single global index** for every unknown (degree of freedom, DoF) in the problem. That global numbering is what lets us build the global vector u, f and the sparse matrix A.
 - Locally (per cell) we have r+1 shape functions $\{\phi_i\}_{i=0}^r$.
 - Globally many of those DoFs are shared at interfaces (e.g., the right vertex of one cell is the left vertex of the next).
 - The **DoFHandler** walks the mesh and **assign unique global indices** to those shared DoFs exactly once, so the global system reflects continuity correctly.
- **?** What the two calls do

```
// attach to triangulation
dof_handler.reinit(mesh);
// assign global DoF indices using 'fe'
dof_handler.distribute_dofs(*fe);
```

- reinit(mesh): tells the handler which mesh to use.
- distribute_dofs(*fe): asks fe where DoFs live on each cell and then produces a global numbering that respects sharing across cell interfaces.

In general, **triangulation first**, **then DoFHandler**, **then distribute DoFs**, and only *after that* we can build the sparsity pattern and allocate matrices/vectors.

? How many DoFs do we get in 1D? Let M be the number of cells (in our code M = N + 1). For FE_Q(r) in 1D:

$$\#DoFs = M \cdot r + 1$$

The first cell contributes r+1 DoFs, and each subsequent cell contributes r new ones (the interface node is shared), so:

$$(r+1) + (M-1) \cdot r = Mr + 1$$

Example 6: A tiny concrete picture

If r = 1 and N = 3:

- Mesh: M = N + 1 = 4 cells, nodes at $x_0 = 0, x_1, x_2, x_3, x_4 = 1$
- Global DoFs (before BCs): indices 0,1,2,3,4 (so n_dofs = 5 = N+2).
- Cells and their local to global maps:
 - Cell 0 $[x_0, x_1]$: local (0, 1) \to global (0, 1)
 - Cell 1 $[x_1, x_2]$: local (0, 1) \rightarrow global (1, 2)
 - Cell 2 $[x_2, x_3]$: local (0, 1) \to global (2, 3)
 - Cell 3 $[x_3, x_4]$: local (0, 1) \to global (3, 4)
- During assembly on Cell 1, for example, when we add cell_matrix(0, 1) we write into global A(1, 2); on Cell 2 the same local entry writes into A(2, 3), etc. Shared nodes cause sums into the same rows/cols (that's how continuity/gluing shows up algebraically).

4. Sparse structure + algebra shells

```
1 // Initialize the linear system.
2 {
      printf("Initializing the linear system\n");
      // First, we need to initialize the sparsity pattern.
      // This object describes the structure of the system
      // matrix (i.e. which entries are nonzero).
      // We use a DynamicSparsityPattern object to generate
      // the sparsity pattern, and then copy it into a
      // SparsityPattern object,
10
      // which is more efficient for computations.
      printf(" Initializing the sparsity pattern\n");
      DynamicSparsityPattern dsp(dof_handler.n_dofs());
13
      DoFTools::make_sparsity_pattern(dof_handler, dsp);
14
      sparsity_pattern.copy_from(dsp);
15
16
      // Then, we use the sparsity pattern to initialize
17
18
      // the system matrix
      printf(" Initializing the system matrix\n");
19
      system_matrix.reinit(sparsity_pattern);
20
21
      // Finally, we initialize the right-hand side and
22
      // solution vectors.
23
      printf(" Initializing the system right-hand side\n");
24
      system_rhs.reinit(dof_handler.n_dofs());
25
      printf(" Initializing the solution vector\n");
26
      solution.reinit(dof_handler.n_dofs());
```

The goal of this step is to encode which pairs of DoFs can couple (i.e., where A_{ij} might be nonzero), and allocate the sparse matrix A and vectors f, u with the right sizes and layout.

From FE connectivity \rightarrow nonzero pattern. In FEM, DoFs are the unknowns (one per basis function). Two DoFs couple if their basis functions overlap on at least one cell. In the matrix A, this means the entry A_{ij} can be nonzero. So the sparsity pattern is just the list of all index pairs (i,j) where A_{ij} might be nonzero. We precompute this once (fast) before assembly so the sparse matrix knows exactly which slots to allocate.

Example 7: 1D

Let r = 1, 4 cells and then 5 DoFs. Cells (with their **global** DoFs):

- Cell 0: $[x_0, x_1] \to \text{DoFs } (0,1)$
- Cell 1: $[x_1, x_2] \to \text{DoFs} (1,2)$
- Cell 2: $[x_2, x_3] \to \text{DoFs} (2,3)$
- Cell 3: $[x_3, x_4] \to \text{DoFs} (3,4)$

Each cell has **2 local DoFs**: its left and right vertices. With 4 cells, we might have global DoFs 0, 1, 2, 3, 4. The sparsity pattern it built by inserting **all pairs** from each cell's set $I \times I$:

- From cell 0: (0,0), (0,1), (1,0), (1,1)
- From cell 1: (1,1), (1,2), (2,1), (2,2)
- Etc.

Collecting unique pairs gives this **tridiagonal** pattern:

Those x are the positions we **pre-allocate**. Later, assembly just **adds numbers** into these slots.

? Why "precompute" the pattern? Making the sparse matrix grow on-the-fly is expensive and unsafe. Instead, we do:

```
DynamicSparsityPattern dsp(n_dofs);
for (each cell K) {
    I = global dof indices on K;
    for (i in I) for (j in I) dsp.add(i,j); // mark possible nonzeros
}
sparsity_pattern.copy_from(dsp);
system_matrix.reinit(sparsity_pattern); // allocate exact slots
```

Now assembly can do constant-time adds into known positions.

? Why this structure?

- Build the graph dynamically (cheap inserts): DynamicSparsityPattern dsp(dof_handler.n_dofs()); The sparsity pattern tells us which entries of the global matrix may be non zero. This data structure is an insertion-friendly container. While we scan each cell and its local DoFs, we "declare" couplings (i, j) by inserting them into dsp. Inserts are frequenti and irregular, so we need a structure that doesn't reallocate like crazy.
 - **?** Why are we talking about graphs? deal.II never draws a graph, but under the hood uses an abstract adjacency graph of the sparse matrix:
 - Vertices are DoFs (global unknown indices $0 \dots N-1$).
 - Edges (i, j) are potential nonzero A_{ij} (i.e., DoF i couples with DoF j on at least one cell).

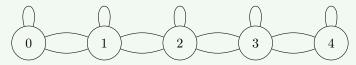
DoFTools::make_sparsity_pattern(dof_handler,dsp) walks the mesh cell by cell, looks at the local DoFs on that cell, and for every pair (i,j) in that local set, it marks an entry in dsp. That is literally building the **adjacency** of the global matrix, just not visually.

Example 8: Graph 1D

For example, with r=1 and a uniform mesh, the degrees of freedom (DoFs) are 0,1,2,3 and 4 along the line. Cells: [0, 1], [1, 2], [2, 3], [3, 4]. Each cell contributes couplings among its two DoFs:

- Cell [0, 1] \rightarrow Edges (0,0)
- Cell [0, 1] \rightarrow Edges (0,0), (0,1), (1,0), (1,1)
- Cell [1, 2] \rightarrow Edges (1,1), (1,2), (2,1), (2,2)
- Etc.

Collecting unique edges gives this **graph** (self-edges are the diagonal):



We never see a drawing, but dsp stores exactly this connectivity. Then:

- sparisty_pattern.copy_from(dsp) freezes it in a compressed format;
- And system_matrix.reinit(sparisty_pattern) allocates numeric storage only for those (i, j).

• Freeze the graph into a compressed pattern (CSR-like):

```
sparsity_pattern.copy_from(dsp);
```

Converts that flexible graph into a compact, cache-friendly layout (row-compressed). This step is all about **memory efficiency and fast access** downstream.

• Create global Stiffness Matrix A:

```
system_matrix.reinit(sparsity_pattern);
```

Where sparsity_pattern tells which entries (i, j) are allowed to exist. After this, the matrix has the right shape and memory allocated, but its entries are all zero.

• Right-hand side:

```
system_rhs.reinit(dof_handler.n_dofs());
```

Initializes the **right-hand side vector** f. Its size equals the number of global unknowns (DoFs). This vector will later be filled with the load term integrals $\int f\varphi_i$. In other words, it is the input forcing vector for the linear system.

• Create Solution vector:

```
solution.reinit(dof_handler.n_dofs());
```

Initializes the **solution vector** u. Same size as **system_rhs**. At the beginning, it is just a vector of zeros. After solving, it will contain the approximate solution at each DoF. In other words, it is the unknown vector that will store the result after the solver runs.

After setup, we are ready to assemble the system: Au = f.

- system_matrix: A
- ullet solution: u
- \bullet system_rhs: f





Output:

```
Setup started
Initializing the mesh
Number of elements = 40
Mesh saved to mesh - 40. vtk
Initializing the finite element space
Degree = 1
DoFs per cell = 2
Quadrature points per cell = 2
```

```
Initializing the DoF handler

Number of DoFs = 41

Initializing the linear system

Initializing the sparsity pattern

Initializing the system matrix

Initializing the system right-hand side

Initializing the solution vector
```

2.2.6.6 Assemble: compute A and f

This is the part at the top of assemble():

```
1 // Number of local DoFs for each element.
  const unsigned int dofs_per_cell = fe->dofs_per_cell;
4 // Number of quadrature points for each element.
5 const unsigned int n_q = quadrature->size();
7 // FEValues instance.
_{8} // This object allows to compute basis functions, their
_{9} // derivatives, the reference-to-current element mapping
10 // and its derivatives on all quadrature points
11 // of all elements.
FEValues < dim > fe_values (
       *fe,
       *quadrature,
14
15
      \ensuremath{//} Here we specify what quantities we need FEValues
      // to compute on quadrature points.
16
      // For our test, we need:
17
      // - the values of shape functions
19
      // - the derivative of shape functions
      // - the position of quadrature points
20
      // - the product J_c(x_q)*w_q
21
      update_values |
22
23
      update_gradients |
      update_quadrature_points |
      update_JxW_values
25
26);
28 // Local matrix and right-hand side vector.
_{29} // We will overwrite them for each element within the loop.
30 FullMatrix < double > cell_matrix(dofs_per_cell, dofs_per_cell);
                      cell_rhs(dofs_per_cell);
31 Vector < double >
33 // We will use this vector to store the global indices
34 // of the DoFs of the current element within the loop.
35 std::vector<types::global_dof_index> dof_indices(dofs_per_cell);
_{\rm 37} // Reset the global matrix and vector, just in case.
system_matrix = 0.0;
system_rhs = 0.0;
```

- dofs_per_cell. Each element (cell) of the mesh has its own set of *local* basis functions (shape functions). This number depends on:
 - The finite element type (FE_Q<dim>(r) \rightarrow polynomial degree);
 - The dimension (dim).

For example, 1D linear elements (r = 1), then 2 local DoFs per cell.

- n_q = quadrature->size(). Quadrature rule defines how many integration points we use *per cell*. Needed because we approximate integrals by weighted sums at quadrature points.
- FEValues object. This is the bridge between reference element and physical cell. It knows how to evaluate:
 - Values of shape functions (update_values)

- Gradients of shape functions (update_gradients)
- Coordinates of quadrature points (update_quadrature_points)
- Jacobian × weight factors (update_JxW_values)

Essentially, this object does all the geometric bookkeeping for us.

- Why is FEValues needed?
 - **?** What we want to compute. From laboratory 1, we know the weak form turns into local contributions like (page 52):

$$A_{ij}^{(K)} = \int_{K} \mu(x) \cdot \nabla \varphi_{i}(x) \cdot \nabla \varphi_{j}(x) \, \mathrm{d}x$$

And (page 51):

$$f_i^{(K)} = \int_K f(x) \cdot \varphi_i(x) \, \mathrm{d}x$$

For each element K. So we need:

- * Basis functions $\varphi_i(x)$;
- * Their gradients $\nabla \varphi_i(x)$;
- * Quadrature weights and Jacobians for mapping integrals from reference element to physical cell.
- The "manual" alternative. In principle we could:
 - 1. Hard-code the formulas for each basis function;
 - 2. Transform coordinates from the reference element (e.g. [-1,1]) to the physical element;
 - 3. Compute the Jacobian determinant J and its inverse transpose for gradients;
 - 4. Loop over quadrature nodes, evaluating shape functions and gradients by hand.

This is **tedious**, **error-prone**, **and messy** (especially in 2D/3D or for higher order elements).

- ✓ What FEValues does. FEValues is the automation tool:
 - * Given:
 - · The finite element space (FE_Q),
 - · The quadrature rule (QGauss),
 - · The flags (update_values | update_gradients | ...),
 - * It precomputes, for all quadrature points of a given cell:
 - $\cdot \varphi_i(x_q) \to \text{basis values};$
 - $\nabla \varphi_i(x_q) \to \text{basis gradients, mapped to physical cell;}$
 - · $x_q \rightarrow$ actual coordinates of quadrature points;
 - · $J(x_q)w_q \to$ the correct measure for integration.

So instead of writing integrals by hand, we just write:

```
fe_values.shape_value(i, q)
fe_values.shape_grad(i, q)
fe_values.quadrature_point(q)
fe_values.JxW(q)
```

And it gives the correct quantity.

- Local containers (cell_matrix, cell_rhs). Each cell contributes a local stiffness matrix and a local load vector. These are small (size = dofs_per_cell × dofs_per_cell). We recompute them for each element, then add them into the big global system.
- dof_indices vector. Temporary storage: maps local DoFs of the cell to global DoF indices. For example, on cell 3, local DoFs might correspond to global DoFs [5, 6].
- Reset global system. Before assembly, set matrix and RHS to zero, so we don't accumulate leftovers from a previous call.

*Element-wise assembly of local matrices and vectors into the global system

```
for (const auto &cell : dof_handler.active_cell_iterators())
2
       // Reinitialize the FEValues object on current element.
3
       // This precomputes all the quantities we requested when
      // constructing FEValues (see the update_* flags above)
5
       // for all quadrature nodes of the current cell.
      fe_values.reinit(cell);
       // We reset the cell matrix and vector
      // (discarding any leftovers from previous element).
      cell_matrix = 0.0;
cell_rhs = 0.0;
      cell_rhs
14
      for (unsigned int q = 0; q < n_q; ++q)
15
           // Here we assemble the local contribution for
16
17
           // current cell and current quadrature point,
           // filling the local matrix and vector.
18
19
           // Here we iterate over *local* DoF indices.
20
           for (unsigned int i = 0; i < dofs_per_cell; ++i)</pre>
21
           {
22
               for (unsigned int j = 0; j < dofs_per_cell; ++j)</pre>
               {
24
25
                    // FEValues::shape_grad(i, q) returns
                   // the gradient of the i-th basis function
26
                   // at the q-th quadrature node, already mapped
28
                   // on the physical element:
                   // we don't have to deal with the
29
                   // mapping, it's all hidden inside FEValues.
30
                   cell_matrix(i, j) += diffusion_coefficient.value(
31
                       fe_values.quadrature_point(q)) // mu(x)
32
33
                        * fe_values.shape_grad(i, q)
                                                         // (I)
                       * fe_values.shape_grad(j, q)
34
                                                         // (II)
                                                          // (III)
                        * fe_values.JxW(q);
35
               }
36
37
               cell_rhs(i) += forcing_term.value(
38
39
                   fe_values.quadrature_point(q)
                 * fe_values.shape_value(i, q) * fe_values.JxW(q);
40
           }
41
      }
42
43
```

```
// At this point the local matrix and
      // vector are constructed: we need to sum them into the
45
      // global matrix and vector. To this end, we need to
46
47
      // retrieve the global indices of the DoFs of current cell.
      cell->get_dof_indices(dof_indices);
48
49
       // Then, we add the local matrix and vector into
      // the corresponding positions of the global matrix
      // and vector.
      system_matrix.add(dof_indices, cell_matrix);
53
      system_rhs.add(dof_indices, cell_rhs);
54
55 }
```

- for (const auto &cell: dof_handler.active_cell_iterators()) dof_handler is the object that knows the mesh (triangulation) and the distribution of DoFs on it. active_cell_iterators() returns iterators to all active cells in the mesh.
 - What is an active cell? deal. II supports adaptive meshes: some cells can be refined into smaller children. The active cells are those that are leaves of the refinement tree, i.e. the cells we actually want to assemble on. If we're using a uniform mesh (like in our laboratory), then all cells are active. So this loop literally means: "for each element of the mesh on which the solution is represented, do the following steps".
 - **?** Why loop over cells? Finite element assembly is **element-by- element**:
 - 1. Compute the *local* contributions (cell matrix and RHS).
 - 2. Scatter them into the global system.

That's why we need to visit each cell in turn.

- fe_values.reinit(cell); fe_values was created outside the loop with a finite element (fe), a quadrature rule (quadrature), and some update flags (update_values, update_gradients, ...). However, at that particular moment, it was not linked to any specific cell. So, calling reinit(cell) means: "take this cell, map the reference element to its physical shape, and precompute all the quantities we asked for at quadrature points (shape values, gradients, JxW, coordinates, ...)".
 - Why it's necessary. Every cell can have different geometry (different size, position, refinement). The mapping (and therefore Jacobian, gradients, quadrature point positions) changes per cell. reinit(cell) makes sure that fe_values.shape_grad(i,q) or fe_values.JxW(q) (see below) correspond to this specific cell.
- cell_matrix = 0.0; cell_rhs = 0.0; These two variables are the local contributions for the current cell. They are small (size = dofs_per_cell × dofs_per_cell for the matrix, and dofs_per_cell for the RHS). Since we are looping over all cells, we need to clear them out before computing the contributions for this cell. Otherwise, leftovers from the previous cell would pollute the new computations.

- for (unsigned int q = 0; q < n_q; ++q) {...}. Iterate all quadrature points on the current cell to accumulate the cell's local matrix and RHS via numerical integration.
 - **?** Why it exists. Our integrals:

$$A_{ij}^{(K)} = \int_K \mu \cdot \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}x, \qquad f_i^{(K)} = \int_K f \cdot \phi_i \, \mathrm{d}x$$

Are approximated by sums over quadratures nodes x_q :

$$A_{ij}^{(K)} \approx \sum_{\substack{q=1\\n_q}}^{n_q} \mu(x_q) \cdot \nabla \phi_i(x_q) \cdot \nabla \phi_j(x_q) \cdot JxW(q)$$
$$f_i^{(K)} \approx \sum_{\substack{q=1\\n_q}}^{n_q} f(x_q) \cdot \phi_i(x_q) \cdot JxW(q)$$

Here JxW(q) is the mapped weight $|\det J_K| \cdot w_q$ from the reference element to the physical cell.

- **?** What are x_q ? They are the quadrature points, i.e. the positions inside each cell where the integral is sampled. When we replace integrals with quadrature, we need both:
 - the weights w_q (how much each sample contributes);
 - the **points** x_q (where to evaluate the integrand).

So x_q are the specific coordinates where we evaluate the shape functions, their gradients, and the data f(x), $\mu(x)$, etc.

? Why do we need them? Look at our weak form:

$$a(u, v) = \int_{\Omega} \mu(x) \cdot \nabla u(x) \cdot \nabla v(x) \, dx, \qquad F(v) = \int_{\Omega} f(x) \cdot v(x) \, dx$$

The quadrature approximation is:

$$a(u, v) \approx \sum_{q=1}^{n_q} \mu(x_q) \cdot \nabla u(x_q) \cdot \nabla v(x_q) \cdot JxW(q)$$

$$F(v) \approx \sum_{q=1}^{n_q} f(x_q) \cdot v(x_q) \cdot JxW(q)$$

Here we see explicitly: we need to know where to evaluate $\mu(x)$, f(x), $\phi_i(x)$, $\nabla \phi_i(x)$. That "where" is exactly the quadrature point x_q .

? Where does JxW(q) come from? In FEM, we don't integrate directly on each physical cell K. We map everything back to a **reference** cell \hat{K} (like [-1,1] in 1D, the unit square in 2D, ...). The map $F_K: \hat{K} \to K$ has a **Jacobian matrix**:

$$J_K\left(\hat{\xi}\right) = \frac{\partial x}{\partial \hat{\xi}} \tag{43}$$

When we change variables in an integral, the measure transforms as:

$$\int_{K} g(x) dx = \int_{\hat{K}} g\left(F_{K}\left(\hat{\xi}\right)\right) \cdot \left|\det J_{K}\left(\hat{\xi}\right)\right| d\hat{\xi}$$

On the reference cell, we approximate the integral with quadrature:

$$\int_{\hat{K}} g\left(F_K\left(\hat{\xi}\right)\right) \cdot \left|\det J_K\left(\hat{\xi}\right)\right| \, \mathrm{d}\hat{\xi} \approx \sum_{q=1}^{n_q} g\left(F_K\left(\hat{\xi}_q\right)\right) \cdot \left|\det J_K\left(\hat{\xi}_q\right)\right| \cdot w_q$$

Where w_q are the quadrature weights on \hat{K} .

deal.II packages the product:

$$\left| \det J_K \left(\hat{\xi}_q \right) \right| \cdot w_q$$

Into one quantity:

```
fe_values.JxW(q)
```

- **?** Why we need it. If we didn't multiply by JxW(q):
 - Our integral would be computed as if the cell had unit size and shape.
 - For example, in 1D, integrating over [0, h] would give the same result as over [0, 1].
 - That's completely wrong: the measure (length, area, volume) of each cell must appear in the integral.

JxW(q) is the correct measure of the little piece of the cell around quadrature point x_q .

- for (unsigned int i = 0; i < dofs_per_cell; ++i) {...}. Each cell has dofs_per_cell local shape functions $\{\varphi_i\}$. This loop means: "for each local basis function φ_i on this element, compute its contribution".
 - **?** Why is it needed?
 - 1. For the matrix $A^{(K)}$ we need contributions for every pair (i, j). That's why inside this loop there's another nested loop over j. So we fill an entries of the local stiffness matrix.
 - 2. For the RHS $f^{(K)}$ each equation corresponds to a test function φ_i . So for each i, we compute:

$$f_{i}^{(K)} \approx \sum_{q} f\left(x_{q}\right) \cdot \varphi_{i}\left(x_{q}\right) \cdot JxW(q)$$

• Fill Stiffness Matrix

Inside the surrounding loops over the cell and the quadrature point q, that line accumulates the quadrature contribution to:

$$A_{ij}^{(K)} \approx \sum_{q=1}^{n_q} \mu(x_q) \cdot \nabla \phi_i(x_q) \cdot \nabla \phi_j(x_q) \cdot JxW(q)$$

Which is the discrete version of:

$$A_{ij}^{(K)} = \int_K \mu(x) \cdot \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, \mathrm{d}x \text{ (local form)}$$

Summing over all cells later gives the global entries A_{ij} used in Au = f. A few precise points about what happens here:

- Role of j: for a fixed i, we pair the test function ϕ_i with every trial function ϕ_j on the cell. That's why this loop runs over all local DoFs. The result is the full dense local matrix $\left(A_{ij}^{(K)}\right)_{i,j=0}^{\text{dofs_per_cell-1}}$.
- What each factor supplies:

```
* \mu(x_q) is:

diffusion_coefficient.value(

fe_values.quadrature_point(q)

)
```

- * fe_values.shape_grad(i,q) is $\nabla \phi_i(x_q)$ in physical coordinates.
- * fe_values.shape_grad(j,q) is $\nabla \phi_j(x_q)$.
- * fe_values.JxW(q) is the mapped weight $\left|\det J_K\left(\hat{\xi}_q\right)\right|\cdot w_q.$
- **Symmetry**: since $\nabla \phi_i \cdot \nabla \phi_j = \nabla \phi_j \cdot \nabla \phi_i$ and $\mu \geq 0$, the local (and global) matrix is symmetric: $A_{ij} = A_{ji}$. Many codes exploit this to halve work, but it's fine (and clearer) to assemble all (i, j).
- Cost structure: the nested i,j loops make the local work $O(n_q \cdot dofs_per_cell^2)$ per cell (big O notation). For r=1 in 1D, dofs_per_cell = 2, so this is tiny; in higher order/dimension, careful placement of the $mu(x_q)$ load (hoist it outside the j loop), and reuse of $shape_grad(i,q)$ values help with performance.
- Connection to the notes: the exact same bilinear form shown in our laboratory 1 solution:

$$A_{ij} = a(\phi_j, \phi_i) = \int_{\Omega} \mu \cdot \phi'_j \cdot \phi'_i \, \mathrm{d}x$$

Is what we're discretizing here via quadrature on each cell.

After this j loop finishes (for the current i and current q), we've added the q-th quadrature contribution to the entire $\mathbf{row}\ i$ of the local matrix. The outer loops over q (quadrature points) and then over i complete the whole local matrix for the cell; afterwards we scatter it to the global matrix with system_matrix.add(dof_indices, cell_matrix).

• Assemble load vector (right-hand side)

We start from the weak formulation of the Poisson problem:

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

With:

$$a(u,v) = \int_{\Omega} \mu(x) \cdot u'(x) \cdot v'(x) \, dx, \quad F(v) = \int_{\Omega} f(x) \cdot v(x) \, dx$$

When we discretize, the **load vector entry** for basis/test function ϕ_i is:

$$f_i = \int_{\Omega} f(x) \cdot \phi_i(x) \, \mathrm{d}x$$

In the code each part corresponds to the quadrature approximation of this integral:

- forcing_term.value(fe_values.quadrature_point(q)) is $f(x_q)$, the source term evaluated at the quadrature point x_q .
- fe_values.shape_value(i, q) is $\phi_i(x_q)$, the value of the *i*-th basis (test) function at the quadrature point.
- fe_values.JxW(q) is the Jacobian times weight, it converts the integral over a possibly deformed cell into a sum of weighted contributions at quadrature points. Formally, it's:

$$JxW(q) = \det(J_T(x_q)) \cdot w_q$$

Where $det(J_T)$ is the Jacobian determinant of the mapping from the reference element, and w_q is the quadrature weight.

- $cell_rhs(i)$ += ... we sum these contributions over all quadrature points q. This approximates:

$$f_i \approx \sum_q f(x_q) \cdot \phi_i(x_q) \cdot JxW(q)$$

This line is building the discrete version of the **load vector**. For each degree of freedom (basis function ϕ_i), it accumulates the weighted contributions of the source term f over the cell, using numerical quadrature. We can think of it as: "take the forcing term at each integration point, multiply by the shape function, scale by the correct measure of volume (JxW), and sum them up to get the correct right-hand-side entry".

cell->get_dof_indices(dof_indices);

The context. Up to this point, we've assembled the local stiffness matrix (cell_matrix) and the local load vector (cell_rhs) for one element (cell). These are written in terms of the local basis functions

on that cell. But the **global linear system** is assembled in terms of **global degrees of freedom (DoFs)**, one big vector **u** and one big matrix A for the whole mesh. So, we need a map: which global indices do our local basis functions correspond to?

? What does get_dof_indices do? Each cell in deal.II has a small number of local DoFs (e.g. 2 in 1D linear case, 3 in quadratic case, etc.). The method:

```
cell->get_dof_indices(dof_indices);
```

Fills the vector dof_indices with the global numbering assigned to those local DoFs by the DoFHandler. For example, in 1D with N=3 cells, linear elements (r=1):

- Nodes: x_0, x_1, x_2, x_3 .
- Global DoFs: numbered [0,1,2,3].
- Cell 0 (interval $[x_0, x_1]$): local DoFs \rightarrow global indices [0,1].
- Cell 1 ($[x_1, x_2]$): local DoFs \to [1,2].
- Cell 2 ($[x_2, x_3]$): local DoFs \to [2,3].

When we call get_dof_indices on cell 1, we'll get 1,2.

Example 9: Physical analogy

Think of each **cell** like a small team in a company:

- Locally, the team has roles "Alice = i=0, Bob = i=1".
- But globally, Alice is employee #123, Bob is employee #124 in the company's HR system.
- get_dof_indices is the lookup that tells us "local slot → global employee number", so we can update the global payroll system (the linear system) correctly.
- Scatter-Add from the element (cell) to the global system

```
system_matrix.add(dof_indices, cell_matrix);
system_rhs.add(dof_indices, cell_rhs);
```

They are just compact shorthands for the "triple loop" we'd otherwise write:

- Matrix

```
for (unsigned int i = 0; i < dofs_per_cell; ++i)

for (unsigned int j = 0; j < dofs_per_cell; ++j)

system_matrix.add(

dof_indices[i],

dof_indices[j],

cell_matrix(i,j)

);</pre>
```

This adds each local entry $\left(A^K\right)_{ij}$ into the **global** entry $A_{\mathrm{glob}(i),\mathrm{glob}(j)}$

- RHS vector

```
for (unsigned int i = 0; i < dofs_per_cell; ++i)
system_rhs(dof_indices[i]) += cell_rhs(i);</pre>
```

This adds the local load entry f_i^K into the global $f_{\text{glob}(i)}$.

Here dof_indices[i] is exactly the global DoF index for the *i*-th local basis function (provided by cell->get_dof_indices(...)), so each cell contributes to the same global rows/columns where nodes are shared. Over all cells, these adds build the global linear system Au = f (the discrete version of a(u, v) = F(v)).

A Boundary Conditions

```
1 // Boundary conditions.
2
       // We construct a map that stores,
3
       \ensuremath{//} for each DoF corresponding to a Dirichlet condition,
       // the corresponding value.
       // E.g., if the Dirichlet condition is
       // u_i = b, the map will contain the pair (i, b).
       std::map<types::global_dof_index, double> boundary_values;
       // This object represents our boundary data
      // as a real-valued function
       // (that always evaluates to zero).
12
       Functions::ZeroFunction<dim> bc_function;
13
14
       // Then, we build a map that,
      // for each boundary tag, stores the
16
17
      // corresponding boundary function.
       std::map <
18
           types::boundary_id,
19
20
           const Function < dim > *
21
      > boundary_functions;
       boundary_functions[0] = &bc_function;
22
       boundary_functions[1] = &bc_function;
23
24
      // interpolate_boundary_values fills
25
       // the boundary_values map.
      VectorTools::interpolate_boundary_values(
27
28
           dof_handler,
           boundary_functions,
29
           boundary_values
30
      );
31
32
      \ensuremath{//} Finally, we modify the linear system
33
       // to apply the boundary conditions.
      // This replaces the equations
35
36
      \ensuremath{//} for the boundary DoFs with the corresponding
       // u_i = 0 equations.
37
       MatrixTools::apply_boundary_values(
38
39
           boundary_values,
40
           system_matrix,
41
           solution,
           system_rhs
           true
43
      );
44
45 }
```

- std::map<types::global_dof_index,double> boundary_values is essentially preparing a container to store Dirichlet boundary conditions.
 - types::global_dof_index is just deal.II's typedef for an unsigned integer that labels a degree of freedom (DoF) in the global numbering created by the DoFHandler. Every unknown of the system (i.e. each node where we want to compute the solution) gets such an index.
 - double is the prescribed value of the solution at that DoF, coming from the boundary condition.

So this map will be filled later with pairs of the form:

```
(i,b) meaning: DoF i must take the value b.
```

For our Poisson 1D (with homogeneous Dirichlet BCs u(0) = u(1) = 0), this means we will eventually store something like:

```
- (DoF index at x=0, 0.0)
- (DoF index at x=1, 0.0)
```

And possibly more if the mesh has multiple boundary faces. In other words, this line declares a data structure that will **hold all the boundary constraints** needed to later enforce them in the linear system.

- Functions::ZeroFunction<dim> bc_function; creates an object that represents the mathematical function u(x) = 0 in the whole domain of dimension dim.
 - Functions::ZeroFunction<dim> is a class provided by deal.II that inherits from Function<dim>.
 - It is a simple function object: whenever we evaluate it at some point $x \in \mathbb{R}^{\dim}$, it always returns 0.0.
 - The template parameter <dim> just means it works in 1D, 2D, or 3D, depending on our problem setup.
- Why do we need it? Because boundary conditions in deal.II are expressed as functions defined on the boundary. Even if the condition is simply homogeneous Dirichlet (u = 0), we still need to pass a function object. So here, bc_function is the prescribed boundary function that tells the solver: "on the boundary, the solution must equal zero".

Later on, this object is associated with the boundary ids (e.g. left and right ends of the interval in 1D), and then interpolated into the discrete finite element space to actually generate the values for each boundary DoF.

• Set the Dirichlet functions

```
std::map<
types::boundary_id,
const Function<dim> *
boundary_functions;
boundary_functions[0] = &bc_function;
boundary_functions[1] = &bc_function;
```

First of all, we prepare a map between boundary identifiers and the corresponding boundary conditions.

- types::boundary_id is just a deal.II typedef (an integer type) used to tag different parts of the boundary of the mesh.
- The value is a pointer to a const Function<dim> object, because different boundaries may have different prescribed functions.

So this map will tell deal.II "on boundary with id X, apply function Y as boundary condition".

About assignments:

- boundary_functions[0] = &bc_function; Boundary with id = 0 (e.g. in 1D, this corresponds to x = 0) is assigned the function bc_function. Recall bc_function is ZeroFunction
 dim>, so: "on boundary 0, impose u = 0".
- boundary_functions[1] = &bc_function; Boundary with id = 1 (e.g. in 1D, the point x = 1) is also assigned the same zero function. So "on boundary 1, impose u = 0".
- **?** Why two lines? In our lab problem (Poisson in 1D), the domain is $\Omega = (0,1)$ and the BCs are homogeneous Dirichlet:

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

deal.II distinguishes boundaries using IDs. By default: left endpoint of the 1D interval \rightarrow id 0; right endpoint \rightarrow id 1. So we explicitly say: both boundary ids (0 and 1) are constrained with the zero Dirichlet function.

• Computes boundary condition values

```
1 VectorTools::interpolate_boundary_values(
2     dof_handler,
3     boundary_functions,
4     boundary_values
5 );
```

This function from deal. II computes the actual values of the boundary conditions on the degrees of freedom (DoFs) that live on the boundary, and fills the boundary_values map we created earlier.

1. Inputs

- dof_handler: knows the finite element space and which DoFs are attached to which cells/faces. It can tell us which DoFs sit on the boundary.
- boundary_functions: the map that says: "on boundary id =
 X, apply function Y".
- boundary_values: the map that will be filled.

2. Interpolation process

 For each boundary id in boundary_functions, deal.II finds all the DoFs lying on that boundary part.

- It then evaluates the corresponding function (here bc_function, i.e. always 0.0) at the DoF locations.
- It records the pair (global_dof_index, value) into boundary_values.
- 3. Result: after this call, boundary_values contains entries like:

$$\{0 \to 0.0, 19 \to 0.0\}$$

Assuming DoF and DoF 19 happen to be the endpoints of our 1D mesh with 20 subintervals.

? Why is this interpolation needed? Because the function is continuous (mathematical), but the finite element solution space is discrete: it only knows about DoFs. We must translate "u(x) = 0 on boundary" into "DoF #i = 0.0" for every DoF lying on the boundary. So this line is the **bridge** between the abstract boundary condition (function on the boundary) and the concrete algebraic system (specific entries in vectors/matrices).

• Impose Dirichlet BCs on the linear system

```
MatrixTools::apply_boundary_values(
boundary_values,
system_matrix,
solution,
system_rhs,
true

7 );
```

Inputs:

- boundary_values: the map $\{i \mapsto b_i\}$ saying "DoF i must take value b_i ".
- system_matrix A, system_rhs f, and the current solution vector u (used/updated).
- The last flag true means also eliminate columns (keeps the system symmetric and clean).

 \blacksquare What apply_boundary_values does (Dirichlet enforcement). For each constrained DoF i in boundary_values:

- 1. Zero the entire row i of A.
- 2. Put a 1 on the diagonal: $A_{ii} \leftarrow 1$.
- 3. Set the RHS: $f_i \leftarrow b_i$.
- 4. If eliminate_columns == true (our case), also zero column i and correct the RHS of the unconstrained equations to account for the known value $u_i = b_i$:

$$f_j \leftarrow f_j - A_{ji}^{(\text{old})} \cdot b_i \quad \text{for all unconstrained } j$$

Then set $A_{ji} \leftarrow 0$ for all $j \neq i$.

5. Set the solution entry: $u_i \leftarrow b_i$ (so the vector is consistent for output/initial guesses).

Net effect: the *i*-th equation becomes exactly $u_i = b_i$ and the rest of the system is modified so those fixed values no longer appear implicitly on the left-hand side.

? Why the last true matters

- true (eliminate columns): preserves symmetry/positive definiteness (important for CG), avoids spurious nonzeros, and makes the algebra cleaner.
- false: leaves columns intact; sometimes used to keep certain preconditioner patterns, but less common for symmetric SPD problems.

So this single call rewrites our system so that **boundary DoFs are hard- set to the prescribed values** and the remaining unknowns solve a system that already accounts for those fixed notes.

In conclusion, the assembly routine builds the algebraic system corresponding to our PDE. First, it computes the local stiffness matrices and load vectors on each element using quadrature, then it inserts them into the global matrix and right-hand side. Finally, boundary conditions are enforced by directly modifying the system so that boundary DoFs take the prescribed values. After this step, the discrete problem is fully defined as a linear system Au = f, ready to be solved with the chosen linear solver.





2.2.6.7 Solve phase

Once the assembly stage is completed and boundary conditions have been enforced, we are left with a well-posed linear system of equations:

$$Au = f$$

Where A is the global stiffness matrix, f is the right-hand side vector, and u is the vector of unknown DoFs. The next step is to actually **solve this system** numerically.

In finite element problems (FEMs), the system matrix is typically **symmetric** and **positive definite** (SPD) for elliptic PDEs like the Poisson equation. This property makes the **Conjugate Gradient (CG) method** an excellent choice, since it is designed specifically for large, sparse SPD systems. The code sets up a solver with a tolerance proportional to the residual norm of the right-hand side, specifies a maximum number of iterations, and then calls the CG routine to compute the solution vector. For now, the solver uses no preconditioner, meaning convergence may be slower on more challenging problems, but the overall workflow is clear:

- 1. Define a stopping criterion (SolverControl).
- 2. Create the CG solver object (SolverCG).
- 3. Apply it to the matrix, right-hand side, and unknown vector (solve).

The result is that the unknown vector solution now contains the discrete approximation of the PDE solution at the mesh nodes.

```
void Poisson1D::solve()
2 {
      3
      // Here we specify the maximum number of iterations of the
5
      // iterative solver, and its tolerance.
      SolverControl solver_control(1000, 1e-6 * system_rhs.12_norm())
      // Since the system matrix is symmetric and positive definite (
      SPD),
      // we solve the system using the conjugate gradient method.
      SolverCG < Vector < double >> solver(solver_control);
12
      printf(" Solving the linear system\n");
13
      // We don't use any preconditioner for now,
14
15
      // so we pass the identity matrix as preconditioner.
      solver.solve(system_matrix, solution, system_rhs,
16
      PreconditionIdentity()):
      std::cout << " " << solver_control.last_step()</pre>
               << " CG iterations'
18
               << std::endl;
19
20 }
```

• SolverControl

```
SolverControl solver_control(
1000, 1e-6 * system_rhs.l2_norm()
3);
```

SolverControl is a deal.II utility class that defines the stopping criteria for iterative solvers. Every iterative solver (like CG, GMRES, etc.) needs to know *when to stop*.That can be either:

- When the residual becomes smaller than a given tolerance, or
- When a maximum number of iterations is reached.

The arguments:

- 1000 is the maximum number of iterations. If the solver hasn't converged within 1000 iterations, it stops anyway. This prevents infinite loops or wasted computation.
- 1e-6 * system_rhs.12_norm() is the tolerance on the residual norm. Where system_rhs.12_norm() computes the Euclidean norm of the right-hand side vector f. Multiplying by 10^{-6} sets the tolerance proportional to the size of the problem data. This means: we ask the solver to reduce the residual until:

$$||r|| \le 10^{-6} \cdot ||f||$$

Where r = Au - f is he residual vector.

② Why this choice? Using a relative tolerance $\frac{\|r\|}{\|f\|} < 10^{-6}$ is standard, because it makes the convergence criterion independent of the scale of the problem. For example, if f is very large, then asking for an absolute residual smaller than 10^{-6} would be meaningless. By scaling with $\|f\|$, the solver stops when the residual is six orders of magnitude smaller than the forcing term.

In other words, this line says "run the iterative solver for at most 1000 steps, but stop earlier if the residual has dropped below 10^{-6} times the size of the right-hand side".

• SolverCG

SolverCG < Vector < double >> solver(solver_control);

- SolverCG is deal.II's implementation of the Conjugate Gradient (CG) method. The CG method is an iterative algorithm designed for solving symmetric positive definite (SPD) linear systems Au = f. In our Poisson problem, the stiffness matrix A is SPD (because the bilinear form $a(u, v) = \int \mu \cdot \nabla u \cdot \nabla v$ is coercive), so CG is the ideal solver.
- The solver is templated on the vector type it operates with. Here we use <Vector<double» (deal.II's own vector class) to represent the solution, RHS, and residual vectors. This ensures compatibility with the algebra objects created during assembly.
- The solver object is constructed and linked with the SolverControl defined earlier. This means the CG solver will follow those stopping rules:
 - * Maximum 1000 iterations, or

* Residual norm below $10^{-6} ||f||$.

• solve

```
solver.solve(
system_matrix,
solution,
system_rhs,
PreconditionIdentity()
);
```

- system_matrix is the global stiffness matrix A assembled in the previous phase. It encodes how basis functions interact (the bilinear form).
- solution is the vector u of unknowns. On entry, it usually contains zero (or a guess). On exit, it will contain the computed approximation of the PDE solution at the mesh DoFs.
- $system_rhs$ is the right-hand side vector f. It represents the load functional, i.e. the forcing term integrated against basis functions.
- PreconditionIdentity() specifies the preconditioner to use.
 Here it's just the *identity preconditioner* (i.e. no preconditioning).
 In practice, the solver just uses the raw system matrix. For small or simple problems, this is fine; for larger systems, one would replace it with something like PreconditionJacobi, PreconditionSSOR, or even more advanced preconditioners.

When this function returns, solution contains the discrete FE approximation u_h . solver_control.last_step() tells us how many CG iterations were needed.





Output:

```
Solving the linear system
Linear system solved in 37 CG iterations
```

2.2.6.8 Output phase

After assembling and solving the linear system, the final step is to **visualize** the computed finite element solution. The output phase is responsible for exporting the discrete solution (stored in the vector solution) together with the mesh information, into a file format that can be inspected with external visualization tools such as ParaView. In deal.II, this task is managed by the DataOut class, which collects numerical results defined on the mesh and writes them in common scientific formats (e.g. VTK).

```
void Poisson1D::output() const
2
      3
      // The DataOut class manages writing the results to a file.
      DataOut < dim > data_out;
6
      // It can write multiple variables (defined on the same mesh)
        to a single file. Each of them can be added by calling
      // add_data_vector, passing the
      // associated DoFHandler and a name.
      data_out.add_data_vector(dof_handler, solution, "solution");
13
      // Once all vectors have been inserted, call build_patches
14
        to finalize the DataOut object,
      // preparing it for writing to file.
      data_out.build_patches();
18
      // Then, use one of the many write_* methods to write
19
20
      // the file in an appropriate format.
      const std::string output_file_name =
21
       "output-" + std::to_string(N + 1) + ".vtk";
      std::ofstream output_file(output_file_name);
23
      data_out.write_vtk(output_file);
24
      printf("Output written to %s\n", output_file_name.c_str());
26
      printf("======\n");
27
28 }
```

- DataOut<dim> data_out; DataOut is a deal.II utility class that handles the export of finite element data for visualization. It knows how to:
 - Collect solution vectors (and possibly multiple variables).
 - Reconstruct them on each cell.
 - $-\,$ Write them into output formats like VTK, VTU, DX, etc.

Like most deal.II classes, it is templated on the dimension of the problem (1D, 2D, 3D). This is needed so that it knows how to traverse the mesh and build patches (pieces of the mesh decorated with solution values).

Why do we need it? By itself, the vector solution is just a list of numbers, one per degree of freedom. DataOut takes those numbers, matches them with the geometry of the mesh (dof_handler), and generates the geometric + data description that visualization software understands. In other words, this line creates an empty DataOut object, which will be filled with the solution data and then asked to write an output file.

- data_out.add_data_vector(dof_handler, solution, "solution");
 This call registers a vector of values (in our case, the finite element solution) with the DataOut objects. It links:
 - The mesh and DoF structure (dof_handler);
 - The numerical data (solution vector);
 - And a name ("solution") that will appear in the output file.

Input parameters:

- dof_handler: Tells DataOut how the DoFs are placed on the mesh, so it knows where each entry of the solution vector belongs.
- solution: The vector of computed values at all DoFs. This is the result of the solver, now ready to be visualized.
- "solution": A label for the data field. If we open the .vtk file in ParaView, we'll see this name in the list of available scalar fields.
- Why do we need it? We can add multiple data vectors with different names (e.g. "temperature", "flux", "error_indicator"). Here, we just add the single primary unknown, the scalar solution of the Poisson equation. In other words, this line tells DataOut: "here is the solution vector, defined over the mesh in dof_handler. Save it as a field called solution".
- data_out.build_patches(); This command tells the DataOut object: "take the mesh + DoF information, interpolate the solution, and build the geometric representation that will be written to file". Concretely, it constructs patches: small pieces of the domain (usually one per cell) where the solution values are stored in a format compatible with visualization software.
 - Why "patches"? In finite elements, the solution is represented by shape functions on each cell. For output, visualization tools expect a piecewise polynomial field that can be plotted as colors, contours, etc. build_pathces() evaluates the solution on each cell and prepares this piecewise description. By default, one patch corresponds to one cell of the mesh. If the FE degree is > 1, deal.II subdivides the cell appropriately so that curved shape functions are represented smoothly in the output.

In other words, this step **finalizes the DataOut object**, making it ready for export. Before this call, data_out only know "there's a solution vector"; after it, it has all the geometric and field data structured for writing.

Write vtk

```
const std::string output_file_name =
     "output-" + std::to_string(N + 1) + ".vtk";
std::ofstream output_file(output_file_name);
data_out.write_vtk(output_file);
```

1. **File name construction**. Builds the name of the output file as a string. It will look like output-20.vtk if N+1=20 (i.e. the number of mesh nodes). This way, the filename encodes the resolution of the mesh, useful when running with different discretizations.

- 2. **Open file stream**. Creates an output stream and opens the file with the chosen name. If the file already exists, it will be overwritten.
- 3. Write data in VTK format. Writes the content of data_out (mesh + solution values) into the file, in VTK format. VTK is a standard format for scientific visualization, compatible with tools like ParaView. Once written, we can load the .vtk file in one of these programs to see plots, color maps, or isosurfaces of our computed solution.

In other words, this block **exports the numerical solution to disk** in a standard visualization format.

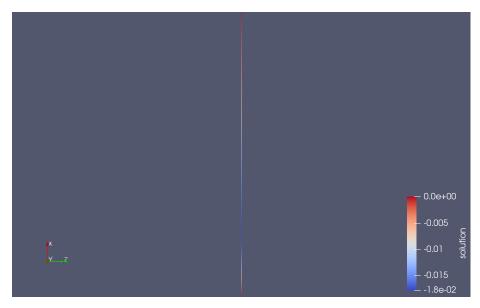


Figure 12: Solution in ParaView.

- The **domain is 1D** (the Poisson problem in (0,1)), but ParaView always tries to render in 3D space. That's why we see a thin vertical line at the center.
- The **color map** represents our finite element solution along that 1D mesh.
- The scale bar on the right shows that the solution is close to zero, with small negative values (down to about -1.8×10^{-2}).
- The **axes triad** in the bottom left confirms: the mesh is along the x axis, with y=z=0.



2.2.6.9 Extensions

In the basic implementation of our Poisson solver, all parameters are hard-coded inside the C++ source:

- The number of mesh elements N + 1;
- The polynomial degree r;
- The diffusion coefficient $\mu(x)$;
- The right-hand side f(x), etc.

This means that if we want to change, for example, the mesh size or ghe degree of the finite elements, we need to **edit the code and recompile**.

The class ParameterHandler in deal.II solves this inconvenience: it lets us define, document, and read parameters from a text file at runtime:

- We register each parameter in the handler with a name, default value, and description.
- We then parse an input .prm file.
- The solver reads the values directly from that file, so we can change parameters without touching or recompiling the C++ code.

ParameterHandler: Example workflow

- 1. In our solver class, we declare a ParameterHandler prm; object.
- 2. In a dedicated method (say declare_parameters()), we register entries, e.g.:

```
prm.declare_entry(
        "N", "39", Patterns::Integer(),
        "Number of elements in the mesh"

4 );
prm.declare_entry(
        "r", "1", Patterns::Integer(),
        "Polynomial degree of FE space"

8 );
```

3. In another method (e.g. parse_parameters(const std::string &file-name)), we load the file:

```
prm.parse_input(filename);
N = prm.get_integer("N");
r = prm.get_integer("r");
```

4. Then we can run our program with a parameter file parameters.prm:

```
subsection Poisson problem
set N = 100
set r = 2
end
```

✓ Why ParameterHandler is useful

- Flexibility: we can run different problem configurations just by editing the text file.
- Reproducibility: parameter files can be saved and shared.
- Scalability: in larger PDE projects, the number of parameters grows quickly (mesh files, time steps, tolerances, preconditioners, ...). Hard-coding them becomes impractical.

Profiling: TimerOutput

The goal is to turn our solver from "it runs" into "it runs and we **know** why it takes that long". So we can reason about scalability, pick better solvers/preconditioners, and justify design changes like mesh resolution or FE degree.

After we write and test the code, we want to measure performance to find bottlenecks and possibly introduce optimizations. TimerOutput gives us an ata-glance table with wall times/percentages per section; if we later need flame graphs or per-function hotsposts, switch to an external profiler. What it measures:

- Setup (initialization): mesh creation/refinement, DoF enumeration, sparsity pattern allocation. Helps check overhead of changing N, r, or reading meshes from file.
- Assembly: local integrals to global matrix/vector. Typically dominates at moderate sizes; reveals impact of quadrature order, FE degree, cache behavior, and boundary handling.
- Solve: linear solver and preconditioner. On large problems this is often the **bottleneck**; timing it lets us compare CG vs GMRES, or Jacobi vs (S)SOR vs SSOR, etc.
- Output: I/O and data formatting (e.g., VTK). Useful to know if file writing is skewing short runs.

* How it works (conceptually)

We create a TimerOutput object once (.hpp file):

```
#include <deal.II/base/timer.h>
  class Poisson1D
3
4
  {
      public:
5
          // Constructor: N = (N+1) elements on [0,1], r = FE degree
          Poisson1D(const unsigned int &N_, const unsigned int &r_)
               : N(N_), r(r_),
               // Print one line per timer section, wall times only.
              timer(
                   std::cout, TimerOutput::summary,
12
                   TimerOutput::wall_times
13
```

```
14
15
           {}
16
           // FEM pipeline (defined later).
17
           void setup();
18
           void assemble();
19
20
           void solve();
           void output() const;
21
           void print_timing() const; // new
22
     protected:
23
          // Discretization parameters
24
25
26
          // Problem data
27
29
           // Geometry & FE space
30
32
           // Algebraic objects: A u = f
33
34
35
           // Profiling
36
           mutable TimerOutput timer;
37
38 }
```

We wrap each **phase** in a timed scope (.cpp file):

```
void Poisson1D::setup()
    TimerOutput::Scope t(timer, "1_setup");
3
4
5 }
7 void Poisson1D::assemble()
8 {
    TimerOutput::Scope t(timer, "2_assembly");
9
11 }
12
void Poisson1D::solve()
14 {
    TimerOutput::Scope t(timer, "3_solve");
15
  // ...
16
17 }
18
void Poisson1D::output() const
20 {
    TimerOutput::Scope t(timer, "4_output");
22
23 }
24
void Poisson1D::print_timing() const
26 {
27
   timer.print_summary();
28 }
```

On exit, it prints a **summary table** with times per label and percentages of total wall-time.

2.3 Poisson 1D: Convergence & Error Analysis

2.3.1 Why Convergence & Error Analysis?

In the previous laboratory (page 25) we built a **finite element solver** for the 1D Poisson problem. At that stage, the solver could generate a mesh, assemble the stiffness matrix, impose boundary conditions, solve the linear system, and output results. However, having code that "runs" is not enough: we must ask ourselves whether it is **mathematically correct**. This is where the second lab (this section) comes into play.

Verification vs. Validation

- Verification asks: "are we solving the equations correctly?". It focuses on the implementation of the numerical method. We want to ensure that our FEM discretization of Poisson's equation converges at the rates predicted by theory.
- Validation asks: "are we solving the correct equations?". This concerns the mathematical model. For example, is Poisson's equation with Dirichlet boundary conditions the right physical model for heat diffusion in a bar?

This lab is entirely about **verification**, not validation. Our task is to prove that the FEM code correctly implements the Galerkin formulation.

■ Method of Manufactured Solutions (MMS)

In practice, we rarely know the exact solution to a PDE. To verify convergence, however, we need one. The **Method of Manufactured Solutions (MMS)** provides a systematic way:

- 1. Choose a smooth exact solution $u_{\text{ex}}(x)$, even arbitrarily (e.g. $u_{\text{ex}}(x) = \sin(2\pi x)$).
- 2. Derive the forcing term f(x) by plugging u_{ex} into the PDE:

$$-u_{\rm ex}''(x) = f(x)$$

For the sine example, this gives $f(x) = 4\pi^2 \sin(2\pi x)$.

3. Impose boundary conditions consistent with $u_{ex}(0)$ and $u_{ex}(1)$.

By construction, u_{ex} is the exact solution of the PDE we feed to our solver. We can then compute the numerical error $u_h - u_{\text{ex}}$ and verify the theoretical convergence rates.

? Why not use trivial "patch tests"?

A Patch test is a very simple verification check used in structural mechanics and early FEM development. The idea: choose a problem whose exact solution belongs exactly to the finite element space V_h . For example, if we use linear finite element (r=1), take $u_{\rm ex}(x)=ax+b$. Since the finite element basis can represent this function exactly, the solver should reproduce $u_{\rm ex}$ with zero error (up to machine precision). While useful, patch tests do not tell us whether our solver converges at the right rates for general functions:

- With $u_{\text{ex}} \in V_h$, we always get zero error, independent of mesh size.
- Therefore, we learn nothing about the approximation properties of our FEM code.

That is why here we go beyond patch tests, we choose **Manufactured solutions that are not in the FE space**. These produce a nonzero error that decrease as $h \to 0$. Measuring that decrease is what confirms convergence.

Deepening: Method of Manufactured Solutions (MMS)

? The problem we want to solve. For example, take the Poisson problem:

$$-u''(x) = f(x), \quad x \in (0,1), \qquad u(0) = u(1) = 0$$

Normally, we **don't know** u(x), and we use FEM to approximate it numerically. But if we want to *check our FEM code*, we need to know what the **exact** u(x) is, so we can measure the error $(u - u_h)$. That's the purpose of MMS.

- **?** How does MMS work? Instead of starting from f(x) and trying to find u(x), we reverse the logic.
 - 1. Choose any smooth, convenient function $u_{ex}(x)$. For example:

$$u_{\rm ex}(x) = \sin(2\pi x)$$

We can compute all its derivatives easily.

2. Plug it into the PDE to manufacture the corresponding f(x). Take the PDE operatore, here $-\frac{\partial^2}{\partial x^2}$, and apply it to u_{ex} :

$$f(x) = -u''_{ex}(x) = 4\pi^2 \sin(2\pi x)$$

Now we have constructed an artificial right-hand side f(x).

3. Build the boundary conditions consistently. For Dirichlet BCs, set:

$$u(0) = u_{\text{ex}}(0) = 0, \quad u(1) = u_{\text{ex}}(1) = 0$$

And this works for us since $\sin(0) = \sin(2\pi) = 0$.

4. Solve the *manufactured* PDE with our FEM code. Our FEM solver does not know the "trick". It just solves:

$$-u_h''(x) = f(x), \quad x \in (0,1), \quad u_h(0) = u_h(1) = 0$$

Since f was built exactly from u_{ex} , the exact solution of this PDE is indeed $u_{\text{ex}}(x)$ itself.

? Why isn't it magic? We are not giving a function and asking the code to "return" it. Instead, we are *creating* a problem whose solution we already know:

- 1. Constructing a full PDE whose analytic solution we know.
- 2. Feeding the PDE (through its forcing term f and BCs) to our FEM solver.
- 3. Comparing the numerical result u_h with our known u_{ex}

So the "black box" isn't guessing $u_{\rm ex}$; it's solving a PDE that was designed so that $u_{\rm ex}$ is its exact solution.

? Why did we choose $u_{\rm ex}$ outside the FEM space? Because if $u_{\rm ex}$ were representable exactly (e.g. a linear function for linear FE), the FEM would reproduce it perfectly for any mesh, giving zero error and no convergence information. By picking $u_{\rm ex}$ that is **not** in the finite element space, like a sine, the solver can only approximate it. When we refine the mesh, $u_h \to u_{\rm ex}$, and the error decreases at a predictable rate:

$$|u - u_h|_{L^2} = O(h^{r+1})$$
 $|u - u_h|_{H^1} = O(h^r)$

We can then measure these rates and verify that the implementation behaves as theory predicts.

■ Analogy. In a similar way, when testing a sorting algorithm, we might generate random lists of numbers (the "manufactured input") and know the expected sorted output. We then run the algorithm and compare its output to our known correct result. The algorithm isn't "magically" producing the sorted list; it's being tested against a case where we already know the answer. Think of MMS like a **unit test** for our numerical solver.

In programming	In PDE verification
We create a function f() and check if its output matches what we expect.	We create a PDE (with known u_{ex}) and check if the solver approximates it with the right accuracy.
The test is artificial, not a real-world case.	MMS PDE is artificial, not physical, but mathematically consistent.

In summary, MMS is a systematic way to create test problems with known solutions, allowing us to verify that our numerical methods are implemented correctly and converge as expected.

2.3.2 Problem Setup for MMS

? How to choose a smooth exact solution

The key idea is to pick something we like, as long as it is smooth enough. Typical choices are **polynomials**, **trigonometric functions** (e.g. sines and cosines), or **exponentials**, because their derivatives are easy to compute.

However, the solution must satisfy three primary requirements:

1. The exact solution u_{ex} must satisfy the **boundary conditions** of the problem (e.g., homogeneous Dirichlet u(0) = u(1) = 0).

Example 10: Choose a smooth exact solution

If we pick $u_{\rm ex}(x)=\sin(2\pi x)$, it automatically vanishes at both ends:

$$u_{\rm ex}(0) = \sin(0) = 0, \quad u_{\rm ex}(1) = \sin(2\pi) = 0$$

2. The exact solution u_{ex} must be **sufficiently smooth** (at least C^2) so that plugging into -u'' = f makes sens and the theory applies.

Example 10 (continue): Choose a smooth exact solution

Taking $u_{\rm ex}(x)=\sin(2\pi x)$ is a good choice since it is infinitely differentiable:

$$u'_{\text{ex}}(x) = 2\pi \cos(2\pi x), \quad u''_{\text{ex}}(x) = -4\pi^2 \sin(2\pi x) \implies u_{\text{ex}} \in C^{\infty}$$

3. It should **not belong to our FE space** (so that error is not identically zero). For example, avoid linear functions when using r = 1 elements.

Example 10 (continue): Choose a smooth exact solution

The choice $u_{\text{ex}}(x) = \sin(2\pi x)$ is suitable since it is not a polynomial and cannot be exactly represented by piecewise linear functions.

■ Deriving the forcing term

Once we have chosen a suitable exact solution $u_{\text{ex}}(x)$, we can derive the corresponding forcing term f(x) by substituting u_{ex} into the Poisson equation:

$$-u_{\rm ex}''(x) = f(x)$$

That's our manufactured forcing term³.

³Note: the manufactured forcing term is not necessarily physically meaningful, but it serves our purpose for verifying the numerical method.

2 Laboratory

Example 11: Deriving the forcing term

For our example $u_{\rm ex}(x) = \sin(2\pi x)$, we compute:

$$u_{\text{ex}}''(x) = -4\pi^2 \sin(2\pi x) \implies f(x) = 4\pi^2 \sin(2\pi x)$$

Thus, the manufactured solution $u_{\rm ex}(x)=\sin(2\pi x)$ satisfies the Poisson equation with the forcing term $f(x)=4\pi^2\sin(2\pi x)$ and homogeneous Dirichlet boundary conditions.

Defining the PDE problem

We can now summarize the complete PDE problem for the Method of Manufactured Solutions (manufactured PDE):

$$\begin{cases} -u''(x) = 4\pi^2 \sin(2\pi x), & x \in (0,1) \\ u(0) = 0, \\ u(1) = 0 \end{cases}$$

with the exact solution:

$$u_{\rm ex}(x) = \sin(2\pi x)$$

This setup allows us to implement the finite element method, compute the numerical solution u_h , and then compare it to the exact solution u_{ex} to analyze convergence and error.

? What do we gain?

By using the Method of Manufactured Solutions:

- We know the **exact solution** u_{ex} , allowing us to compute the error $e_h = u_{\text{ex}} u_h$ directly.
- We can **compute the error in different norms** (e.g., L^2 , H^1) to assess the accuracy of our numerical method.
- We can check if the solver exhibits the expected convergence rates:
 - $-e_{L^2} = O(h^{r+1})$, means the error in the L^2 norm decreases proportionally to h^{r+1} as the mesh is refined.

? Why? Because the L^2 error is related to the smoothness of the exact solution and the order of the finite element space. As we refine the mesh (reduce h), the approximation becomes more accurate, leading to a faster decrease in the error.

It's obvious that more elements mean more mesh points and more computational work, but also a better approximation of the exact solution. $-e_{H^1} = O(h^r)$, means the error in the H^1 norm decreases proportionally to h^r as the mesh is refined.

? Why? The reason behind the convergence rates is rooted in the mathematical properties of the finite element method. The L^2 norm measures the error in the function values, while the H^1 norm measures the error in both the function values and their first derivatives. As the mesh is refined, the finite element space can better approximate the exact solution, leading to a faster decrease in the L^2 error. However, since the H^1 norm also considers the derivative, which is generally less smooth than the function itself, the convergence rate is typically one order lower.

Deepening: $e_{L^2} = O(h^{r+1})$ and $e_{H^1} = O(h^r)$

These two relations are fundamental results in finite element theory, describing how the error between the exact solution and the finite element approximation behaves as the mesh is refined.

$$e_{L^2} = O(h^{r+1}), \qquad e_{H^1} = O(h^r)$$

- u(x): exact (analytic) solution of the PDE.
- $u_h(x)$: finite-element approximation of degree r (e.g., r=1 for linear elements, r=2 for quadratic, etc.).
- h: characteristic **mesh size**, i.e. the largest element length. When we refine the mesh, $h \to 0$.
- $e = u u_h$: the **numerical error**, the difference between the exact and approximate solutions.
- $O(\cdot)$: **Big-O notation**, meaning "asymptotically bounded by". A good tutorial here:



So the goal is to understand how fast this error goes to zero as the mesh is refined $(h \to 0)$.

To quantify the error we use norms, which are ways to measure the size of functions. The two most common norms in FEM are:

• L^2 norm: measures the averaged (integrated) error over the domain. It is defined as:

$$\|u - u_h\|_{L^2}^2 = \left(\int_0^1 (u(x) - u_h(x))^2 dx\right)^{\frac{1}{2}}$$

It captures how close u_h is to u on average across the entire domain.

• H^1 norm: measures both the averaged error and the derivative error. It is defined as:

$$\|u - u_h\|_{H^1}^2 = \|u - u_h\|_{L^2}^2 + \|u' - u_h'\|_{L^2}^2$$

This norm is more "energetic": it considers not only how close the functions are, but also how close their slopes (derivatives) are. This is important in many physical problems where the gradient of the solution matters (e.g., stress in mechanics).

2.3.3 What changes from the first Poisson 1D solver

The PDE and solver skeleton are **the same** as in the first Poisson 1D solver lab. The second laboratory adds a **verification layer** on top of the existing solver, to check that the numerical solution is correct and converges to the exact solution as the mesh is refined. This verification layer is based on the **Method** of **Manufactured Solutions (MMS)**, (see Section 2.3.2, page 105).

✓ New capability in Poisson1D: computing errors

We extend the header file Poisson1D.hpp with the declaration of a new method:

```
double compute_error(const VectorTools::NormType &norm_type) const;
```

This method performs a-posteriori error evaluation against a known exact solution (MMS). The implementation is in Poisson1D.cpp:

```
double Poisson1D::compute_error(
      const VectorTools::NormType &norm_type
  )
3
    const
4
  {
       // The error is an integral,
      // and we approximate that integral using a quadrature formula.
6
       // To make sure we are accurate enough,
      // we use a quadrature formula with one node more than
9
      // what we used in assembly.
      const QGauss < dim > quadrature_error = QGauss < dim > (r + 2);
      // First we compute the norm on each element,
       // and store it in a vector.
13
      Vector < double > error_per_cell(mesh.n_active_cells());
14
      VectorTools::integrate_difference(
           dof_handler,
16
17
           solution,
           ExactSolution(),
           error_per_cell,
19
20
           quadrature_error,
21
           norm_type
      ):
22
23
      // Then, we add out all the cells.
24
      const double error = VectorTools::compute_global_error(
25
26
           mesh, error_per_cell, norm_type
27
28
29
      return error;
30 }
```

• It integrates the element-wise error using deal.II's

```
VectorTools::integrate_difference(...)
```

then collapses to a global norm with

```
VectorTools::compute_global_error(...)
```

This yields either the L^2 or H^1 norm of the error, depending on the input argument norm_type. Later we will see how to use this method in the convergence analysis.

- For accuracy, it uses a **richer quadrature** than assembly (we choose QGauss(r+2)), so the **integration of the error** does not become the bottleneck.
 - Why richer quadrature? The error is the difference between the numerical solution and the exact solution. The numerical solution is a polynomial of degree r (the FE degree), while the exact solution is generally not a polynomial. Therefore, the error is not a polynomial of degree r, but a more complex function. To accurately integrate this function, we need a quadrature that can handle higher-degree polynomials. Using QGauss(r+2) ensures that we can accurately capture the behavior of the error function, leading to more reliable error estimates. Indeed, with QGauss(r+2), n = r + 2, we can exactly integrate polynomials of degree up to 2n 1 = 2(r + 2) 1 = 2r + 3, which is sufficient for our purposes.
 - Why isn't the integration of errors the bottleneck? In numerical methods, the accuracy of error estimation is crucial for assessing the quality of the solution. If the quadrature used for integrating the error is not sufficiently accurate, it can lead to significant discrepancies in the error estimates. By using a richer quadrature like QGauss(r+2), we ensure that the integration process is precise enough to capture the nuances of the error function. This prevents the integration step from becoming a limiting factor in the overall accuracy of the error estimation, allowing us to trust the results of our convergence analysis.
 - Why does this method exist? In numerical simulations, especially when solving PDEs, it is essential to assess the accuracy of the computed solution. The compute_error method provides a systematic way to quantify the difference between the numerical solution and a known exact solution (from MMS). It lets us to quantify whether the method achieves the theoretical convergence rates:

```
- ||u - u_h||_{L^2} = O(h^{r+1}) for the L^2 norm;

- ||u - u_h||_{H^1} = O(h^r) for the H^1 norm.
```

Implementation note: ExactSolution::value(...) and ExactSolution::gradient(...) must be consistent, because

VectorTools::integrate_difference(...)

Uses both to compute the L^2 and H^1 norms of the error.

✗ Modifications to Forcing Term and Exact Solution

To implement the Method of Manufactured Solutions (MMS), we need to modify the forcing term class to correspond to a known exact solution. Also, we need to implement the exact solution class. These changes are necessary to verify the correctness of our numerical solution.

? Why change the ForcingTerm? In the first Poisson 1D solver lab, the forcing term was a simple constant function defined as:

```
class ForcingTerm : public Function < dim >
2 {
3
  public:
      // Constructor.
      ForcingTerm() : Function < dim > () {}
       // Evaluation.
       virtual double value(
           const Point < dim > &p,
9
           const unsigned int /*component*/ = 0
10
11
      ) const override
12
           if (p[0] <= 1.0 / 8 || p[0] > 1.0 / 4.0)
13
               return 0.0;
14
           else
15
16
               return -1.0;
      }
17
18 };
```

Listing 1: Original Forcing Term

Where -1.0 in the interval $(\frac{1}{8}, \frac{1}{4}]$ and 0.0 elsewhere. This choice was arbitrary and did not correspond to any specific exact solution.

For MMS, we need a forcing term that corresponds to our chosen exact solution. We select a manufactured solution (e.g., $u(x) = \sin(\pi x)$) and derive the corresponding forcing term using the Poisson equation -u''(x) = f(x). This ensures that our numerical solution can be verified against a known exact solution (see Section 2.3.2, page 105). The modified ForcingTerm class is:

```
class ForcingTerm : public Function < dim >
2
  }
3
  public:
      // Constructor.
      ForcingTerm() : Function < dim > () {}
      // Evaluation.
      virtual double value(
           const Point < dim > &p
10
           const unsigned int /*component*/ = 0
      ) const override
           return 4.0 * M_PI * M_PI * std::sin(2.0 * M_PI * p[0])
13
      }
15 };
```

Listing 2: Modified Forcing Term for MMS

This corresponds to the forcing term derived from the manufactured solution $u(x) = \sin(2\pi x)$:

$$f(x) = -u''(x) = 4\pi^2 \sin(2\pi x)$$

Why add the ExactSolution class? The ExactSolution class is introduced to provide a reference solution against which we can compare our numerical solution. This class implements the exact solution and its gradient, which are essential for computing the error norms. The implementation is as follows:

```
1 // Exact solution.
  class ExactSolution : public Function < dim >
  {
3
4
       // Constructor.
      ExactSolution() {}
       // Evaluation.
9
       virtual double
       value(
           const Point < dim > &p,
           const unsigned int /*component*/ = 0) const override
12
13
         // Points 3 and 4.
14
15
         return std::sin(2.0 * M_PI * p[0]);
16
17
       // Gradient evaluation.
18
       // deal.II requires this method to return a Tensor
19
       // (not a double), i.e. a dim-dimensional vector.
20
       // In our case, dim = 1, so that the Tensor will in
21
       // practice contain a single number.
22
       // Nonetheless, we need to return an object
23
24
       // of type Tensor.
       virtual Tensor<1, dim>
25
26
       gradient (
           const Point < dim > &p,
27
           const unsigned int /*component*/ = 0) const override
28
29
         Tensor<1, dim> result;
30
31
32
         // Points 3 and 4.
         result[0] = 2.0 * M_PI * std::cos(2.0 * M_PI * p[0]);
33
34
35
         return result;
36
37
       static constexpr double A = -4.0 / 15.0 * std::pow(0.5,
38
39 };
```

Listing 3: Exact Solution Class for MMS

This class provides the exact solution $u(x) = \sin(2\pi x)$ (in value(...)) and its gradient $u'(x) = 2\pi \cos(2\pi x)$ (in gradient(...)). The gradient value is returned as a Tensor<1, dim> because deal.II requires this format for gradient evaluations, even in 1D where it effectively behaves like a vector with a single component.

? What does Tensor mean? A tensor is a generalization of scalars (rank 0), vectors (rank 1), and matrices (rank 2) to arbitrary rank. It's the natural object for representing quantities in continuum mechanics, PDEs, and FEM:

- Rank 0 tensor: scalar field value u(x);
- Rank 1 tensor: vector quantity, e.g. gradient 2D $\nabla u = (\partial_x u, \partial_y u)$;
- Rank 2 tensor: matrix quantity, e.g. stress tensor, Jacobian of a mapping.
- **Higher-rank tensors**: more complex relationships in advanced applications, e.g. elasticity, electromagnetism, etc.

In deal.II, the Tensor<rank, dim> is a template class representing a tensor of given rank in dim dimensions. For example:

- Tensor<0, dim> is just a scalar (like double);
- Tensor<1, dim> is a vector of length dim (like std::array<double, dim>);
- Tensor<2, dim> is a dim x dim matrix.

So when we compute the **gradient of a scalar field** in dim dimensions, we get a rank-1 tensor Tensor<1, dim>.

- In 1D, this is essentially a vector with one entry (the derivative $\partial_x u$);
- In 2D, it's a vector with two entries (the partial derivatives $\partial_x u$ and $\partial_u u$);
- In 3D, it's a vector with three entries (the partial derivatives $\partial_x u$, $\partial_y u$, and $\partial_z u$).

Remark 3: Gradient

The **Gradient** of a scalar field u(x) in 1D is simply its derivative:

$$\nabla u(x) = \frac{\partial u}{\partial x}$$

In higher dimensions, the gradient is a vector of partial derivatives. For example, in 2D, the gradient is:

$$\nabla u(x,y) = \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix} = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right)$$

And in 3D, it's:

$$\nabla u(x,y,z) = \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \\ \frac{\partial u}{\partial z} \end{bmatrix} = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z} \right)$$

The gradient points in the direction of the steepest increase of the function and its magnitude indicates how steep that increase is.

New capability in main: convergence driver

The main.cpp (or lab-02.cpp) file is modified to turn the solver into a convergence driver. A convergence driver runs the solver multiple times on a sequence of refined meshes, computes the errors, and estimates the convergence rates.

```
#include <deal.II/base/convergence_table.h>
3 #include <fstream>
4 #include <iostream>
5 #include <vector>
7 #include "Poisson1D.hpp"
9 int main(int /*argc*/, char * /*argv*/[])
10 {
       ConvergenceTable table;
11
12
       const std::vector<unsigned int> N_values = {
13
          9, 19, 39, 79, 159, 319
14
15
      const unsigned int
                                         degree = 2;
16
17
18
       std::ofstream convergence_file("convergence.csv");
      convergence_file << "h,eL2,eH1" << std::endl;</pre>
19
20
21
       for (const unsigned int &N : N_values)
22
23
           Poisson1D problem(N, degree);
24
           problem.setup();
25
26
           problem.assemble();
           problem.solve();
27
           problem.output();
28
                                  = 1.0 / (N + 1.0);
           const double h
30
           const double error_L2 = problem.compute_error(
31
               {\tt VectorTools}:: {\tt L2\_norm}
32
           );
33
34
           const double error_H1 = problem.compute_error(
               VectorTools::H1_norm
35
36
37
           table.add_value("h", h);
38
           table.add_value("L2", error_L2);
39
40
           table.add_value("H1", error_H1);
           convergence_file << h << "," << error_L2</pre>
41
                             << "," << error_H1 << std::endl;
42
43
44
      table.evaluate_all_convergence_rates(ConvergenceTable::
45
      reduction_rate_log2);
46
       table.set_scientific("L2", true);
47
      table.set_scientific("H1", true);
48
49
       table.write_text(std::cout);
50
51
52
       return 0;
53 }
```

Listing 4: main.cpp for convergence analysis

The key changes are:

- ConvergenceTable table; A helper class to store and analyze convergence data. ConvergenceTable is part of deal.II's base module that simplifies the collection and presentation of convergence results. For example, it can automatically compute convergence rates based on the errors and mesh sizes provided.
- Mesh ladder N_values = $\{9, 19, 39, 79, 159, 319\}$ A vector of mesh sizes (number of elements) to run the solver on. This creates a sequence of increasingly refined meshes for convergence analysis. They are taken from lab instructions ("with polynomial degree r = 1, solve the Poisson 1D problem, with finite elements, setting N + 1 = 10, 20, 40, 80, 160, 320"). Since our 1D mesh has N + 1 elements, the mesh size is:

$$h = \frac{1}{N+1} \in \left\{ \frac{1}{10}, \frac{1}{20}, \frac{1}{40}, \frac{1}{80}, \frac{1}{160}, \frac{1}{320} \right\}$$

Each step halves h is essential for estimating convergence rates.

- Fix the polynomial degree degree = 2 The polynomial degree of the finite element basis functions is fixed to r = 2 (quadratic elements). This choice affects the expected convergence rates:
 - L^2 norm: $O(h^{r+1}) = O(h^3)$; - H^1 norm: $O(h^r) = O(h^2)$.

We can later rerun the convergence analysis with different polynomial degrees to observe how the rates change.

• CSV for plotting

```
std::ofstream convergence_file("convergence.csv");
convergence_file << "h,eL2,eH1" << std::endl;</pre>
```

We create a CSV file to store the mesh size and errors for external plotting. The header row defines the columns: h (mesh size), eL2 (L^2 error), and eH1 (H^1 error). Each iteration appends a new row with the current mesh size and computed errors:

```
convergence_file << h << "," << error_L2 << "," << error_H1 <<
    std::endl;</pre>
```

This file can be used later for plotting convergence graphs using tools like Python's Matplotlib or Excel.

Loop over meshes, solve, compute the errors and store in table

```
for (const unsigned int &N : N_values)
{
    Poisson1D problem(N, degree);
    problem.setup();
    problem.assemble();
    problem.solve();
    problem.output();

const double h = 1.0 / (N + 1.0);
```

```
const double error_L2 = problem.compute_error(
           VectorTools::L2_norm
11
      ):
12
       const double error_H1 = problem.compute_error(
13
14
           VectorTools::H1_norm
      ):
15
16
       table.add_value("h", h);
17
       table.add_value("L2", error_L2);
18
       table.add_value("H1", error_H1);
19
       convergence_file << h << "," << error_L2 << "," <<
20
       error_H1 << std::endl;
21
```

For each mesh size in N_values, we:

- Instantiate a new Poisson1D object with the current number of elements and fixed polynomial degree.
- Call the full solver pipeline: setup(), assemble(), solve(), and output().
 - Why call output() here? Calling output() within the loop allows us to generate and save the solution for each mesh size. This is useful for visual inspection of how the solution improves with mesh refinement. It also provides a record of the numerical solutions corresponding to each level of mesh refinement, which can be valuable for debugging and analysis.
- Compute the mesh size h and the errors in L^2 and H^1 norms using the new compute_error(...) method.
- Store the mesh size and errors in the ConvergenceTable for later analysis.
- Append the results to the CSV file for external plotting.

The constant h = 1.0 / (N + 1.0) computes the mesh size based on the number of elements. This is crucial for convergence analysis, as we need to know how the error behaves as h decreases. About the errors, we call:

- problem.compute_error(VectorTools::L2_norm) to compute the L^2 norm of the error;
- problem.compute_error(VectorTools::H1_norm) to compute the H^1 norm of the error.

• Compute \log_2 convergence rates

```
table.evaluate_all_convergence_rates(ConvergenceTable::
    reduction_rate_log2);
table.set_scientific("L2", true);
table.set_scientific("H1", true);
table.write_text(std::cout);
```

After collecting all the data, we call evaluate_all_convergence_rates (...) to compute the convergence rates based on the errors and mesh sizes. We use ConvergenceTable::reduction_rate_log2 to get the rates in base 2, which is standard in numerical analysis. The set_scientific

(...) calls format the error values in scientific notation for better readability. Finally, write_text(...) outputs the convergence table to the console, showing the mesh sizes, errors, and estimated convergence rates.

? What is a convergence driver?

A convergence driver is a program or script that systematically tests a numerical method by solving a problem on a series of increasingly refined meshes. The goal is to observe how the error between the numerical solution and the exact solution decreases as the mesh is refined, thereby estimating the convergence rates of the method. Key features of a convergence driver include:

- Mesh Refinement: It runs the solver on a sequence of meshes with decreasing mesh sizes (e.g., halving the mesh size at each step).
- Error Computation: It computes the error norms (e.g., L^2 , H^1) between the numerical solution and a known exact solution (from MMS).
- Convergence Rate Estimation: It analyzes how the errors decrease with mesh refinement to estimate the convergence rates, which can be compared against theoretical predictions.
- Data Collection and Presentation: It collects the results in a structured format (e.g., tables, CSV files) for easy analysis and visualization.

By automating this process, a convergence driver helps verify the correctness and efficiency of numerical methods, ensuring they perform as expected in practice.

? What we should expect from the convergence analysis?

When we run the convergence driver, we should expect to see the following results in the output convergence table:

- **Decreasing Errors**: As the mesh is refined (i.e., as h decreases), the errors in both the L^2 and H^1 norms should decrease. This indicates that the numerical solution is converging to the exact solution.
- Convergence Rates: The estimated convergence rates should align with the theoretical predictions based on the polynomial degree r of the finite element basis functions:
 - For the L^2 norm, we expect a convergence rate of approximately r+1. For example, with r=2, we expect a rate of about 3.
 - For the H^1 norm, we expect a convergence rate of approximately r. For example, with r=2, we expect a rate of about 2.
- (Tip) **Scientific Notation**: The errors should be presented in scientific notation for better readability, especially as they become very small with mesh refinement.
- Consistency: The results should be consistent across different runs of the convergence driver, indicating that the numerical method is stable and reliable.

If the observed convergence rates match the theoretical expectations, it validates the correctness of the implementation and the effectiveness of the numerical method. If there are discrepancies, it may indicate issues in the implementation or the need for further investigation into the numerical method's properties.

The output of the convergence driver should look like this:

output-smooth.txt



The source code is available at:

Source code



2.3.4 Non-smooth manufactured case

Until now, our exact solution:

$$u_{\rm ex}(x) = \sin(2\pi x)$$

Is **infinitely smooth**, meaning that it has derivatives of all orders, and they are all continuous. This is a very nice property, but it is not always the case in real-world problems. That's the ideal case, it satisfies the theoretical assumptions under which FEM gives optimal convergence rates:

$$e_{L^2} = \mathcal{O}(h^{p+1}), \quad e_{H^1} = \mathcal{O}(h^p)$$

Where p is the polynomial degree of the basis functions. But in the real world (and in advanced numerical analysis), **many PDEs have solutions that are not perfectly smooth**, for example: corners in the domain, material interfaces, singular loads, cracks, etc. In those cases, the solution is continuous but not differentiable everywhere. This is called a **non-smooth solution**. In such cases, the question is: what happens to our convergence rates then? That's what the "non-smooth manufactured solution" experiment tests.

X Constructing the non-smooth manufactured solution

The non-smooth case is constructed by defining a block of code in the Poisson1D.hpp file, in the ForcingTerm and ExactSolution classes. The code is:

```
_{1} // Set to 1 for smooth case (points 3 and 4),
_2 // 0 for non-smooth case (point 5).
  #define SMOOTH_CASE 0
5
  class ForcingTerm : public Function < dim >
9
      public:
           // Constructor.
10
           ForcingTerm() = default;
           // Evaluation.
13
14
           virtual double value(const Point < dim > &p,
               const unsigned int /*component*/ = 0) const override
15
           {
16
  #if SMOOTH_CASE
17
               // Points 3 and 4; smooth case.
18
               return 4.0 * M_PI * M_PI * std::sin(2.0 * M_PI * p[0]);
19
20
               // Point 5; non-smooth case.
21
22
               if (p[0] < 0.5)
                    return 0.0;
23
               return -std::sqrt(p[0] - 0.5);
24
25
  #endif
           }
26
27
       };
  class ExactSolution : public Function < dim >
29
      {
30
31
          // Constructor.
32
```

```
ExactSolution() {}
33
34
           // Evaluation.
35
36
           virtual double value(
               const Point < dim > &p,
37
               const unsigned int /*component*/ = 0
38
39
           ) const override
           {
40
  #if SMOOTH_CASE
41
               // Points 3 and 4; smooth case.
42
               return std::sin(2.0 * M_PI * p[0]);
43
44
45
                // Point 5; non-smooth case.
               if (p[0] < 0.5)
46
47
                   return A * p[0];
               return A * p[0] + 4.0 / 15.0 * std::pow(p[0] - 0.5,
48
      2.5);
  #endif
50
51
           virtual Tensor<1, dim> gradient(
52
               const Point < dim > &p,
53
                const unsigned int /*component*/ = 0
           ) const override
56
57
               Tensor<1, dim> result;
58
59
               // Points 3 and 4; smooth case.
  #if SMOOTH_CASE
60
               result[0] = 2.0 * M_PI * std::cos(2.0 * M_PI * p[0]);
61
                // Point 5; non-smooth case.
63
               if (p[0] < 0.5)
64
                    result[0] = A;
66
                    result[0] = A + 2.0 / 3.0 * std::pow(p[0] - 0.5,
67
      1.5);
  #endif
68
69
               return result;
70
71
72
           static constexpr double A = -4.0 / 15.0 * std::pow(0.5,
73
       2.5);
```

This defines the **piecewise right-hand side** f(x) corresponding to the **non-smooth exact solution** $u_{\text{ex}}(x)$:

$$f(x) = \begin{cases} 0, & x < 0.5 \\ -\sqrt{x - 0.5}, & x \ge 0.5 \end{cases} \qquad u_{\text{ex}}(x) = \begin{cases} Ax, & x < 0.5 \\ Ax + \frac{4}{15}(x - 0.5)^{2.5}, & x \ge 0.5 \end{cases}$$

Where $A = -\frac{4}{15}(0.5)^{2.5}$ is a constant ensuring continuity at x = 0.5. This solution is continuous everywhere, but its first derivative is not continuous at x = 0.5 (it has a **corner** there). This is a typical example of a non-smooth solution.

We have wrap the block of code in a **preprocessor directive**:

```
1 #define SMOOTH_CASE 0
```

```
2 // ...
3 #if SMOOTH_CASE
4    // Smooth case code
5 #else
6    // Non-smooth case code
7 #endif
```

Setting SMOOTH_CASE to 1 activates the smooth case (points 3 and 4), while setting it to 0 activates the non-smooth case (point 5). This allows us to easily switch between the two cases by changing a single line of code.

Running the non-smooth case

Now, instead of the smooth $u_{\rm ex}(x)=\sin(2\pi x)$, we're solving the piecewise 2.5-power function defined above. Using the same main function as before, we can run the simulation for various mesh sizes and polynomial degrees. After running the simulations, we obtain the following error tables for both the smooth and non-smooth cases:

• Smooth case:

```
1 h L2 H1
2 0.1000 - 1.0028e-03 - 6.5007e-02 -
3 0.0500 1.00 1.2590e-04 2.99 1.6319e-02 1.99
4 0.0250 1.00 1.5754e-05 3.00 4.0840e-03 2.00
5 0.0125 1.00 1.9698e-06 3.00 1.0213e-03 2.00
6 0.0063 1.00 2.4624e-07 3.00 2.5533e-04 2.00
7 0.0031 1.00 3.0781e-08 3.00 6.3835e-05 2.00
```

• Non-smooth case:

```
1 h L2 H1
2 0.1000 - 1.3069e-05 - 3.7084e-04 -
3 0.0500 1.00 4.1481e-06 1.66 1.0101e-04 1.88
4 0.0250 1.00 1.4392e-06 1.53 2.7295e-05 1.89
5 0.0125 1.00 5.0736e-07 1.50 7.3559e-06 1.89
6 0.0063 1.00 1.7930e-07 1.50 1.9895e-06 1.89
7 0.0031 1.00 6.3388e-08 1.50 5.4447e-07 1.87
```

Each **row** corresponds to one **mesh refinement**. For a given N, the program computes $h = \frac{1}{N+1}$, then measures the L^2 and H^1 errors between the numerical solution and the exact solution.:

- Column h: mesh size (distance between nodes).
- Column L2: absolute L^2 error, $||u u_h||_{L^2}$.
- Column H1: absolute H^1 error, $||u u_h||_{H^1}$.
- Numbers between errors: **empirical convergence rate** (computed with ConvergenceTable::reduction_rate_log2).

Each line compares current and previous refinement:

$$rate = \frac{\log (e_{prev} \div e_{curr})}{\log (h_{prev} \div h_{curr})}$$

Because our h halves each time, this is essentially a base-2 logarithm of the error reduction:

$$\mathrm{rate} = \log_2 \left(\frac{e_{\mathrm{prev}}}{e_{\mathrm{curr}}} \right)$$

For example, in the smooth case, the second line is:

0.0500 1.00 1.2590e-04 2.99 1.6319e-02 1.99

Where:

- h = 0.0500: the current mesh size.
- First 1.00 is the ratio between current and previous h values (always 2 refinement, printed as reference).
- 1.2590e-04: the current L^2 error.
- 2.99: the empirical convergence rate for the L^2 error, computed as:

$$\operatorname{rate}_{L^2} = \log_2 \left(\frac{1.0028 \times 10^{-3}}{1.2590 \times 10^{-4}} \right) \approx 2.99$$

- 1.6319e-02: the current H^1 error.
- 1.99: the empirical convergence rate for the H^1 error, computed as:

$$\text{rate}_{H^1} = \log_2 \left(\frac{6.5007 \times 10^{-2}}{1.6319 \times 10^{-2}} \right) \approx 1.99$$

The H^1 and L^2 errors respect the theoretical convergence rates for quadratic elements (r=2):

$$e_{H^1} = O(h^r) = O(h^2), \quad e_{L^2} = O(h^{r+1}) = O(h^3)$$

The steady values of approximately 2 and 3 for the H^1 and L^2 errors, respectively, confirm that our numerical solution converges to the exact solution at the expected rates as the mesh is refined.

In the non-smooth case:

- H^1 rate stabilizes around 1.88 (slightly below 2).
- L^2 rate stabilizes around **1.5-1.6** (well below 3).

This indicates that our solver is **still converging**, but slower. The observed rates reflect **reduced regularity** of the exact solution due to the corner at x = 0.5.

? What changes in convergence behavior?

Now the solution is less regular because it has a **kink** (corner) in the curvature (second derivative discontinuity) at x = 0.5. Theory tells us that:

ullet For H^1 norm, the convergence rate is limited by the solution's regularity:

$$||u - u_h||_{H^1} \le C h^{\min(r,s-1)}$$
 (44)

Where s is the regularity of the exact solution and r is the polynomial degree of the finite element space. Here, $s \approx 2.5$ (since the solution is piecewise $C^{2.5}$), so:

$$\min(r, s - 1) = \min(2, 1.5) = 1.5$$

Thus, we expect the H^1 error to converge at a rate of about 1.5, which aligns with our observed rate of approximately 1.88 (slightly better due to mesh alignment with the kink).

• For L^2 norm, the convergence rate is typically one order higher than that of the H^1 norm:

$$||u - u_h||_{L^2} \le C h^{\min(r+1,s)}$$
 (45)

Here, $\min(r+1, s) = \min(3, 2.5) = 2.5$. Therefore, we expect the L^2 error to converge at a rate of about 2.5, which is consistent with our observed rate of approximately 1.5-1.6 (slightly lower due to the kink).

In summary, the presence of a corner in the exact solution reduces its regularity, which in turn lowers the convergence rates of our finite element solver. While we still achieve convergence, the rates are slower than in the smooth case, demonstrating the impact of solution regularity on numerical performance.

? Why this matters

Understanding how solution regularity affects convergence rates is crucial. This experiment teaches a **core finite element principle**: increasing the polynomial degree r improve accuracy **only if** the solution is smooth enough. If the solution has low regularity (e.g., corners, singularities), the **error is dominated** by those singular regions, and no matter how high r is, we won't reach the ideal convergence rate. ? And what can we do about it? Several strategies exist:

- Mesh refinement: Use finer meshes near singularities to better capture solution behavior.
- Adaptive methods: Dynamically refine the mesh based on error estimates.
- **Special basis functions**: Employ enriched or singular basis functions that can better represent non-smooth features.

These techniques help mitigate the impact of low regularity and improve overall solution accuracy.

• Key takeaways

- Solution regularity matters: The smoothness of the exact solution directly influences the convergence rates of finite element methods.
- Non-smooth solutions converge slower: Solutions with corners or singularities lead to reduced convergence rates, as observed in the non-smooth manufactured case.
- Empirical rates confirm theory: The observed convergence rates align with theoretical predictions based on solution regularity.
- **Practical implications**: In real-world problems, where solutions are often non-smooth, understanding and addressing these effects is crucial for effective numerical simulations.

The output of the convergence driver should look like this:

output-non-smooth.txt



The source code is available at:

Source code



2.3.5 Typical pitfalls & debugging checklist

Let's make a quick recap of what we have done so far. When we test a finite element code, like our 1D Poisson solver, we usually **don't know the true** analytic solution of a real PDE. So to **verify** that the implementation is mathematically correct and that the convergence rates match theory, we *manufacture* a problem where we **force** the analytical solution to be known. This is called the *Method of Manufactured Solutions (MMS)*, which we discussed in detail on page 102.

The MMS consists of start from a function $u_{\rm ex}(x)$ of our choice that is sufficiently smooth, simple and easy to differentiate, and we *pretend* it's the exact solution of the PDE. Then, we **compute the right-hand side** f(x) and **the boundary conditions** that make this $u_{\rm ex}$ solve the PDE exactly. For the 1D Poisson problem with constant diffusion coefficient, this means:

$$-(\mu u')' = f$$
 in $(0,1)$, $u(0) = u(1) = 0$,

And $\mu=1$ for simplicity. We pick a manufactured (exact) solution $u_{\rm ex}(x)$, for example:

$$u_{\rm ex}(x) = \sin(2\pi x)$$

Compute its second derivative:

$$u'_{\text{ex}}(x) = 2\pi \cos(2\pi x), \qquad u''_{\text{ex}}(x) = -4\pi^2 \sin(2\pi x)$$

Then, we plugo into the PDE:

$$-(\mu u')' = -\left(\mu u'\right)' = -u''_{\text{ex}}(x) = 4\pi^2 \sin(2\pi x)$$

So the **source term** f(x) is:

$$f(x) = 4\pi^2 \sin(2\pi x)$$

And this guarantees that our chosen $u_{\rm ex}(x)$ solves the PDE. Also, since $u_{\rm ex}(0) = \sin(0) = 0$ and $u_{\rm ex}(1) = \sin(2\pi) = 0$, we can use **homogeneous Dirichlet boundary conditions**.

₹ Bug Catchers: Diagnostic Method (Computes Error in Different Norms)

The compute_error function is the diagnostic part of our solver. It doesn't change the solution, it *measures* how good the solution is. So, when our numerical results don't match the expected convergence rates, this function is the first place where bugs reveal themselves.

- Rates don't match theory
 - **A** Symptom: with r = 1 we expect $e_{L^2} = O(h^2)$ and $e_{H^1} = O(h)$, but we get something else.
 - ✓ Likely causes & tests

- * Wrong BCs applied \rightarrow Error flat near boundaries dominates (the error is not decreasing as $h \rightarrow 0$). Mistakes in boundary conditions result in poor convergence rates.
 - **?** Test: Plot u_h and u_{ex} near boundaries; if they don't match, fix apply_boundary_values or boundary ids.
- * Non-geometric refinement (h not halving) → rates meaningless. If the mesh is not refined uniformly, the computed h may not reflect the actual element sizes, leading to incorrect rate calculations.
 - Fix: Use N+1=10, 20, 40, 80, 160 nodes for uniform meshes; check mesh size h halves each time. If not, fix mesh generation.
- * Under-integration in compute_error → polluted norms. If the quadrature rule used to compute the error is not sufficiently accurate, it can lead to incorrect error estimates.
 - \bigcirc Fix: Increase quadrature order in compute_error and see if results change. If they do, fix quadrature rule using r+2 points for r-th order elements.
- * H1 gradient mismatch (exact gradient not derivative of exact solution) \rightarrow wrong H^1 error. If the gradient used in the H^1 error computation does not match the derivative of the exact solution, the error will be incorrect.
 - **?** Test: Re-derive $u'_{ex}(x)$ and check it matches the code. If not, fix gradient computation.
- * Solver tolerance too lose → algebraic error contaminates asymptotics. If the linear solver does not converge sufficiently, the solution will have an additional error component that can affect convergence rates.
 - Fix: Tighten relative solver tolerance so linear solver error discretization error. Then rates reflect FE theory.

However, in general, the theory reference values are:

$$|u - u_h|_{H^1} = O(h^r), \quad ||u - u_h||_{L^2} = O(h^{r+1})$$

where r is the polynomial degree of the finite element space. This is true for smooth solutions. For non-smooth solutions, the rates may be lower, depending on the regularity of the solution (other rules apply, see 123).

- H^1 rate looks right, L^2 rate is off
 - **A** Symptom: H^1 rate is $O(h^r)$ as expected, but L^2 rate is not $O(h^{r+1})$.
 - ✓ Likely causes & tests
 - * We computed only the H1 seminorm $\to L^2$ norm not computed. If the L^2 norm is not computed correctly, it can lead to incorrect convergence rates.
 - ? Test: Confirm we call compute_error(L2_norm) and use QGauss(r+2). If not, fix it.
 - * Exact value ok, exact gradient wrong (or viceversa) \rightarrow one norm correct, other wrong. If the exact solution and its

gradient are not consistent, it can lead to one norm being correct while the other is wrong.

- **?** Test: If gradient is wrong, H^1 collapses first; if value part mis-specified, L^2 collapses first. Check both. If one is wrong, fix it.
- * Non-smooth exact solution test (reduced regularity) \rightarrow lower L^2 rate. If the exact solution is not smooth enough, it can lead to reduced convergence rates. It is not a bug, but a feature of the problem and the deterioration is **expected**.
- L^2 rate looks right, H^1 rate is off
 - **\hat{\mathbf{x}} Symptom**: L^2 rate is $O(h^{r+1})$ as expected, but H^1 rate is not $O(h^r)$.
 - ✓ Likely causes & tests
 - * ExactSolution::gradient() wrong $\to H^1$ error wrong. If the gradient function does not return the correct gradient, it can lead to incorrect H^1 error estimates. It is a very common bug.
 - Test: Print sample values of value() and gradient() for a few points and compare with hand formulas. If gradient is wrong, fix it.
 - * H1 "norm" vs "seminorm" confusion \rightarrow wrong rate. If the H^1 error is computed as a seminorm (only gradient part) instead of a full norm (value and gradient), it can lead to incorrect convergence rates.
 - **?** Test: Check if compute_error computes seminorm or full norm. If seminorm, fix it to compute full norm.
- Rates oscillate step-to-step
 - **Symptom: rates jump around, not smooth convergence.
 - ✓ Likely causes & tests
 - * Not actually halving h or mixing meshes that don't scale by $2 \to \text{rates}$ meaningless. If the mesh is not refined uniformly, the computed h may not reflect the actual element sizes, leading to incorrect rate calculations.
 - **♥** Fix: Enforce uniform refinement; use $N \to 2N$ nodes each step. Check mesh size h halves each time. If not, fix mesh generation.
 - * Quadrature order changed between runs or adaptive output interfering → inconsistent errors. If the quadrature rule changes between runs, it can lead to inconsistent error estimates.
 - \checkmark Fix: Fix quadrature order in compute_error to r+2 points for r-th order elements.
 - * Algebraic solver not converged on certain meshes (tolerance hit/iteration limit) \rightarrow inconsistent errors. If the linear solver does not converge sufficiently on certain meshes, it can lead to inconsistent error estimates.
 - **♥ Fix:** Check solver residual per mesh; if needed, precondition or tighten tolerance.

- Errors stop decreasing (plateau)
 - **A** Symptom: errors flatten out, no more decrease with refinement.
 - ✓ Likely causes & tests
 - * Quadrature order too low → integration error dominates. If the quadrature rule used to compute the error is not sufficiently accurate, it can lead to a plateau in the error.
 - \bigcirc Fix: Fix quadrature order in compute_error to r+2 points for r-th order elements.
 - * Round-off / Solver tolerance dominates at small $h \to \text{algebraic}$ error dominates. If the linear solver does not converge sufficiently, the solution will have an additional error component that can affect convergence rates.
 - **♥** Fix: Tighten relative solver tolerance so linear solver error discretization error.
 - * Manufactured data not implemented consistently (e.g., f(x) not matching $u_{\text{ex}}) \to \text{error}$ floor. If the source term or boundary conditions do not match the manufactured solution, it can lead to a plateau in the error.
 - **?** Test: Numerically evaluate the strong residual $-u_{\text{ex}}'' f$ at a few points; it should be close to zero for MMs choices (like $\sin(2\pi x)$). If not, fix f(x) or u_{ex} .

2.3.6 Extensions: FunctionParser

In the first labs, the source term f(x), diffusion coefficient $\mu(x)$, boundary data g(x), the FE degree r, and the mesh size N were all hard-coded in the source code of the solver. That makes every change require recompilation, which is fine for learning, but terrible for real applications.

To fix that, we use two deal. II tools:

- ParameterHandler that reads numerical or textual parameters from a .prm file;
- 2. FunctionParser that interprets analytic expressions at runtime, so f(x), $\mu(x)$, etc. can be written as formulas.

Together, they separate code (solver logic) from data (problem definition), the same philosophy as industrial-grade FEM frameworks.

X Implementation outline

1. Declare expected parameters. This part tells to deal.II which parameters to expect, their type (integer, double, string, etc.), and default values. This is done in a dedicated function, e.g., declare_parameters(). It is a sort of schema for the input file. In our case, we declare:

```
void declare_parameters() {
      prm.declare_entry(
           "mu", "1.0", Patterns::Anything(),
           "Diffusion coefficient mu(x)"
      ):
      prm.declare_entry(
           "f", "4*pi*pi*sin(2*pi*x)", Patterns::Anything(),
           "Right-hand side f(x)"
10
      prm.declare_entry(
           "exact_solution", "sin(2*pi*x)", Patterns::Anything(),
11
           "Exact solution (for error computation)
12
      prm.declare_entry(
14
           "degree", "1", Patterns::Integer(1, 5),
15
           "Polynomial degree of FE_Q<1>
16
17
      prm.declare_entry(
18
19
           "N", "10", Patterns::Integer(2),
           "Number of subintervals
20
21
      ):
22
  }
```

Each declare_entry gives: the key name (as in the .prm file), a default value, a pattern to validate the input (e.g., integer between 1 and 5), and a short description.

2. Read parameters from file. Once the schema is defined, we can read the actual values from a .prm file, e.g., input.prm. The best place to do this is in the constructor of the Poisson1D class:

```
explicit Poisson1DExt(const std::string &filename) {
    declare_parameters();
    prm.parse_input(filename);
    mu_expr = prm.get("mu");
    f_expr = prm.get("f");
    N = prm.get_integer("N");
    r = prm.get_integer("degree");
}
```

Here, parse_input reads the file and checks it against the schema (automatically!). Then, we retrieve the values with get or get_integer. To save these values, we need to declare some member variables:

```
// Discretization parameters
unsigned int N; // N+1 elements
unsigned int r; // polynomial degree

ParameterHandler prm;

// Parsed expressions; initialized in setup()
std::string mu_expr, f_expr, exact_solution_expr;

// \( \mu(x) \) - diffusion coefficient
FunctionParser<dim> mu_function;
// f(x) - forcing term
FunctionParser<dim> rhs_function;
// u(x) - exact solution (for error computation)
FunctionParser<dim> exact_solution_function;
```

Note the ParameterHandler prm object to manage parameters, the strings to hold the parsed expressions (what we read from the file), and the FunctionParser objects to evaluate them. These FunctionParser objects are the same concept as Function objects, but they can be initialized at runtime with a string expression. So they correspond exactly to: DiffusionCoefficient diffusion_coefficient and ForcingTerm forcing_term.

3. Build the runtime functions. The FunctionParser objects need to be initialized with the expressions read from the file. This is done in the setup() function, after reading the parameters:

```
void Poisson1DExt::setup()
2
     3
     nSetup started\n");
      // Parse the functions
5
6
         const std::string vars = "x"; // variables used in
     expressions
         const std::map<std::string,double> constants = {
9
             {"pi", numbers::PI},
10
             // (optional synonyms)
             {"PI", numbers::PI}, {"Pi", numbers::PI}
12
         };
13
14
         mu_function.initialize(vars, mu_expr, constants);
15
         rhs_function.initialize(vars, f_expr, constants);
16
         exact_solution_function.initialize(vars,
17
     exact_solution_expr, constants);
```

Here, we define the variable names used in the expressions (just "x" for 1D), and a map of constants (e.g., pi). Precisely, the vars string lists the names of the independent variables that can be used in the expressions, while the constants map defines named constants that can be used in the expressions.

For example, if we use "x" as the variable and define "pi" as a constant, we can write expressions like "sin(pi*x)" in the input file; it will be interpreted as $sin(\pi x)$.

Finally, the initialize() method sets up each FunctionParser with the variable names, the expression string, and the constants. At this point, we can use mu_function, rhs_function, and exact_solution_function just like any other Function object (like DiffusionCoefficient, ForcingTerm, and ExactSolution before).

4. Use the runtime functions. The rest of the code remains mostly unchanged. We just need to replace the old hard-coded functions with the new FunctionParser objects. For example, in assemble():

```
1
  void Poisson1DExt::assemble()
  {
2
      printf("============
      nAssembly the linear system\n");
6
      for (const auto &cell : dof_handler.active_cell_iterators
          // ...
9
          for (unsigned int q = 0; q < n_q; ++q)
              for (unsigned int i = 0; i < dofs_per_cell; ++i)</pre>
13
                   for (unsigned int j = 0; j < dofs_per_cell; ++</pre>
14
      j)
                   {
                       // use mu_function
16
                       cell_matrix(i, j) += mu_function.value(
17
18
                           fe_values.quadrature_point(q))
                           * fe_values.shape_grad(i, q)
19
                           * fe_values.shape_grad(j, q)
20
21
                           * fe_values.JxW(q);
                   }
22
23
24
                   // use rhs_function not forcing_term!
                   cell_rhs(i) += rhs_function.value(
25
26
                           fe_values.quadrature_point(q)
27
                      fe_values.shape_value(i, q) *
28
29
                      fe_values.JxW(q);
30
31
```

And in compute_error:

```
Poisson1DExt::compute_error(const VectorTools::NormType &
      norm_type) const
3
       const QGauss < dim > quadrature_error = QGauss < dim > (r + 2);
4
5
       Vector < double > error_per_cell(mesh.n_active_cells());
6
       VectorTools::integrate_difference(dof_handler,
                                           solution,
                                           // use parsed exact sol
9
                                           exact_solution_function,
10
11
                                           error_per_cell,
                                           quadrature_error,
12
13
                                           norm_type);
14
       const double error = VectorTools::compute_global_error(
15
16
           mesh, error_per_cell, norm_type
17
18
19
       return error;
20 }
```

5. Create the input file. Finally, we need to create a .prm file with the actual parameters and expressions. For example, poisson1d-9.prm:

```
1 set mu = 1.0
2 set f = 4*pi*pi*sin(2*pi*x)
3 set exact_solution = sin(2*pi*x)
4 set degree = 2
5 set N = 9
```

It corresponds to the same problem as before, but now we can change f(x), $\mu(x)$, N, and r without touching the source code. Here, an example of main file with all the prm files; the goal is to simulate the smooth manufactured solution $u(x) = \sin(2\pi x)$ with different meshes and polynomial degrees obtaining the same results of the hard-coded version:





? And what about non-smooth functions?

The powerful FunctionParser can handle piecewise definitions using the if construct, because it is based on the muParser library (a fast math expression parser). For example, for the non-smooth manufactured solution (page 123):

$$f(x) = \begin{cases} 0, & x < 0.5 \\ -\sqrt{x - 0.5}, & x \ge 0.5 \end{cases} \qquad u_{\text{ex}}(x) = \begin{cases} Ax, & x < 0.5 \\ Ax + \frac{4}{15}(x - 0.5)^{2.5}, & x \ge 0.5 \end{cases}$$

we can write in the .prm file:

Note the use of if(condition, value_if_true, value_if_false) and pow (base, exponent) functions. We could also define a constant for A:

$$A = -\frac{4}{15}(0.5)^{2.5}$$

In the constants map in setup(), to make the expression cleaner, but this is left as an exercise to the reader.

References References

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.

\mathbf{Index}

В	
Basis	49
Boundary Value Problem (BVP)	29
Boundary value problem in 1D	10
	10
C	
Computational error	17
convergence driver	117
Convergence order	18
D	
Diffusion equation	11
Dirichlet	30
E	
	1.4
Elliptic PDE	14
\mathbf{F}	
Finite Element Method (FEM)	4
G	
Galerkin formulation	37, 40
Galerkin method	37
Gaussian quadrature	60
Gradient	113
	110
H	
Heat equation	11
Homogeneous Dirichlet condition	30
Hyperbolic PDE	14
I	
Initial and boundary value problem in 1D	11
Initial value problem	9
т	
L Laplace operator	12
Lax-Richtmyer Equivalence Theorem	21
Lax-rachtmyer Equivalence Theorem	21
\mathbf{M}	
Mathematical Model	5
Mathematical Problem (MP)	16
Mesh	42, 44
Mesh Parameter h	44
Method of Manufactured Solutions (MMS)	102
Model Error	16
Multidimensional Heat equation	12
N	
Neumann	30

Index	Index
Non-Homogeneous Dirichlet	30
Numerical Modelling	5
Numerical Problem (NP)	17
0	
ODE (Ordinary Differential Equation)	29
Ordinary Differential Equation (ODE)	9
Orthogonal	39
P	
Parabolic PDE	14
Partial Differential Equation (PDE)	7
Patch test	102
PDE (Partial Differential Equation)	29
PDE discriminant	13
PDE is linear	13
PDE order	13
Physical Problem (PP)	16
Poisson equation	25
Poisson problem	10
Q	
Quadrature Rule	59
R	
Robin	30
Round-Off error	17
\mathbf{S}	
Scientific Computing	5
Stiffness Matrix	52
Strong Formulation	31
T	
Test Function	33
Truncation Error	17