Numerical Methods for Partial Differential Equations
A.Y. 2025/2026

Laboratory 01

The finite element method for
the 1D Poisson equation

Exercise 1.

Let $\Omega = (0,1)$. Let us consider the Poisson problem

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

with $\mu(x) = 1$ for $x \in \Omega$, and

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

1.1. Write the weak formulation of problem (1).

Solution. Let $v(x) \in V$ be a test function in a suitable function space V (to be defined). We multiply v to (1a) and integrate over Ω:

$$\int_0^1 -\left(\mu(x)\; u'(x)\right)' v(x)\, dx = \int_0^1 f(x)v(x)\, dx\;.$$

Integrating by parts the first integral, we get

$$\int_0^1 \mu(x)u'(x)v'(x)\, dx - \left[\mu(x)u'(x)v(x)\right]_{x=0}^1 = \int_0^1 f(x)v(x)\, dx\;.$$

Finally, we assume that $v(0) = v(1) = 0$ (that is, we assume that the test function satisfies the same conditions as u), so that the second term on the left-hand side vanishes:

$$\int_0^1 \mu(x)u'(x)v'(x)\, dx = \int_0^1 f(x)v(x)\, dx\;. \tag{2}$$

For the weak formulation to be well defined, we need the integrals in (2) to be well defined. This is true if $u, v \in V = H^1_0(\Omega) = \left\{v \in L^2(\Omega)\colon v' \in L^2(\Omega),\; v(0) = v(1) = 0\right\}$.

We introduce the shorthand notation

$$a : V \times V \to \mathbb{R} \, , \qquad a(u,v) = \int_0^1 \mu(x) u'(x) v'(x) \, dx \, ,$$
$$F : V \to \mathbb{R} \, , \qquad F(v) = \int_0^1 f(x) v(x) \, dx \, .$$

Then, the weak formulation reads:

$$\text{find } u \in V \; : \; a(u,v) = F(v) \text{ for all } v \in V \, . \tag{3}$$

1.2. Write the Galerkin formulation of problem (1).

Solution. The Galerkin formulation is obtained by restricting (3) to V_h, a finite dimensional subspace of V. It reads:

$$\text{find } u_h \in V_h \; : \; a(u_h, v_h) = F(v_h) \text{ for all } v_h \in V_h \, . \tag{4}$$

1.3. Write the finite element formulation of problem (1), using piecewise polynomials of degree r, and write the associated linear system.

Solution. Let us introduce a uniform partition (the *mesh*, see Figure 2) of Ω into N_{el} subintervals (*elements*) K_c, $c = 1, 2, \dots, N_{\text{el}}$. Let X_h^r be the space of piecewise polynomials over the mesh elements, that is

$$X_h^r(\Omega) = \left\{ v_h \in C^0(\bar{\Omega}) \; : \; v_h(x) \in \mathbb{P}_r \quad \forall x \in K_c, \, \forall c = 1, 2, \dots, N_{\text{el}} \right\} \, .$$

Let $V_h = X_h^r(\Omega) \cap H_0^1(\Omega)$ be the finite element approximation of V, using piecewise polynomials of degree r, and let $N_h = \dim(V_h)$.

For a one-dimensional domain, there holds $\dim(X_h^r) = rN_{\text{el}}$, and $N_h = \dim(V_h) \leq \dim(X_h^r)$ (in this case, $N_h = rN_{\text{el}} - 1$, due to Dirichlet boundary conditions, so that, if $r = 1$, $N_h = N_{\text{el}} - 1$).

Let $\varphi_i(x)$, for $i = 1, 2, \dots, N_h$, be the Lagrangian basis functions of the space V_h. We look for $u_h \in V_h$, so that it can be expressed as:

$$u_h(x) = \sum_{j=1}^{N_h} U_j \varphi_j(x) \qquad x \in \overline{\Omega} \, ,$$

where $U_j \in \mathbb{R}$ are the (unknown) *control variables* or *degrees of freedom* (DoFs).

Instead of requiring that (4) holds for all functions $v_h \in V_h$, we can just do it for each basis function φ_i, for $i = 1, 2, \dots, N_h$. Then, the discrete weak formulation (4) rewrites as: find U_j, for $j = 1, 2, \dots, N_h$, such that

$$\sum_{j=1}^{N_h} U_j a(\varphi_j, \varphi_i) = F(\varphi_i) \qquad \text{for } i = 1, 2, \dots, N_h \, . \tag{5}$$

Equations (5) can be rewritten as a linear system

$$A\mathbf{u} = \mathbf{f}\ , \tag{6}$$

where

$$\begin{aligned}
\mathbf{u} &\in \mathbb{R}^n & \mathbf{u} &= \left(U_1, U_2, \dots, U_{N_h}\right)^T\ , \\
A &\in \mathbb{R}^{N_h \times N_h} & A_{ij} &= a(\varphi_j, \varphi_i) = \int_0^1 \mu(x)\varphi_j'(x)\varphi_i'(x)\,dx\ , \qquad (7) \\
\mathbf{f} &\in \mathbb{R}^{N_h} & \mathbf{f}_i &= F(\varphi_i) = \int_0^1 f(x)\varphi_i(x)\,dx\ . \qquad (8)
\end{aligned}$$

1.4. Implement in `deal.II` a finite element solver for (1), using piecewise polynomials of degree $r = 1$ and with a number of mesh elements $N_{\text{el}} = 20$.

Solution. See the file `src/exercise-01.cpp` for the implementation. A high-level explanation of the structure of the code is provided below, while detailed information is given as comments in the source file.

Implementing a finite element solver (for a linear, stationary problem) ultimately boils down to writing a program that constructs and solves the linear system (6) as efficiently as possible. A generic solver is usually organized along four basic steps (also depicted in Figure 1):

1. setup: all the data structures for the problem are intialized. This includes the creation of the mesh, the selection of appropriate finite element spaces, and the allocation and initialization of algebraic data structures for matrices and vectors;
2. assembly: the matrix and right-hand side of the linear system (6) are constructed, relying on the decomposition into local contributions represented by (9) and (10);
3. linear solver: the system (6) is solved by means of a suitable method (direct or iterative), possibly using an appropriate preconditioner;
4. post-processing phase: the solution is exported to file, and/or used to compute relevant quantities of interest.

In line with the object-oriented paradigm of `C++` and `deal.II`, we wrap all the above steps in a class, `Poisson1D` (files `src/Poisson1D.hpp` and `src/Poisson1D.cpp`). For each step, the class exposes a method: `setup()`, `assemble()`, `solve()`, `output()`. The class stores:

- the discretization settings (number of elements and polynomial degree);
- the mesh, through an object of type `dealii::Triangulation`;
- an object representing the finite element space used for the discretization, using the generic class `dealii::FiniteElement`;

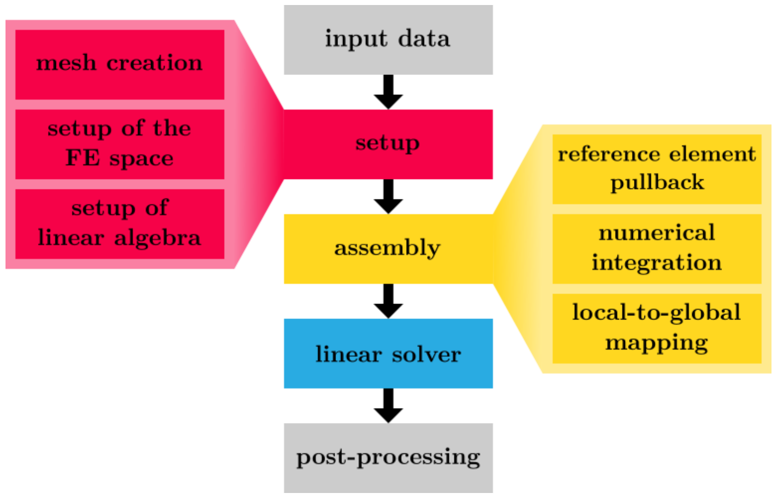


Figure 1: Schematic representation of the main steps in a (linear, time independent) finite element solver.

- an object that manages the numbering of degrees of freedom, using the class `dealii::DoFHandler`;
- an object that represents the quadrature rule used to approximate integrals (see below), of type `dealii::Quadrature`;
- several objects representing the different terms of the linear system: the system matrix `system_matrix`, its sparsity pattern `sparsity_pattern`, the system right-hand side `system_rhs` and its solution `solution`.

Notice that most of `deal.II`'s classes are templated on the physical dimension (usually denoted by `dim`). For this first tests, we consider `dim = 1`, and define a static member in the class `Poisson1D` to store this information. Placing this sort of shared dimension-related information in a single place (e.g. a template argument, or a constant as in this case) is a key ingredient in writing *dimension-independent code*, as will become clear when we move to 2D or 3D problems.

Setup. Here we construct the mesh (or load it from an external file). The mesh is managed by the `deal.II` class `Triangulation` (and its derived classes). The creation of the mesh can be done inside the program itself, by using the tools of the `GridGenerator` namespace (documentation: `https://dealii.org/9.5.0/doxygen/deal.II/namespaceGridGenerator.html`). Alternatively, the mesh can be created externally and loaded from a file using the `GridIn` class (we will see this later in the course).

Then, we initialize the finite element space. This is done using the `deal.II` classes `FiniteElement` (and its derived classes) and `DoFHandler`. The former provides a "local" representation of the finite element space (i.e. it represents concepts such as the polynomial degree and basis functions over each element, the continuity across element

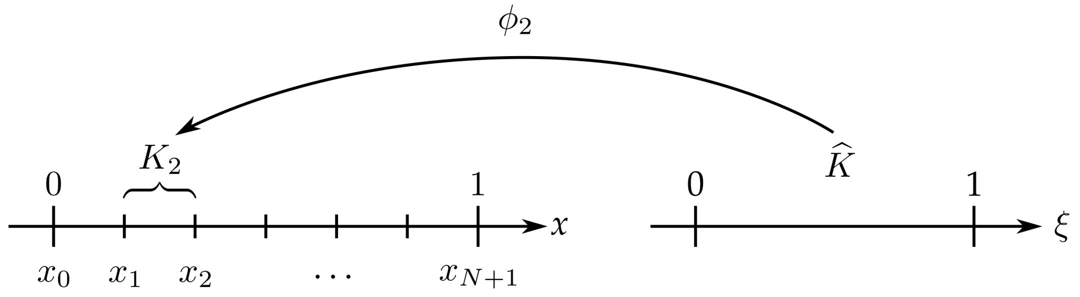


Figure 2: Left: partition of the domain Ω into a mesh with equally sized elements. Right: reference element.

boundaries). The latter takes care of the putting together the global information of the mesh with the local information of the mesh, enumerating the degrees of freedom over the whole domain.

Finally, we construct the algebraic structures for the linear system. The system matrix A is sparse (only some of the basis functions have intersecting support), therefore we define it using the `SparseMatrix<double>` class. To make its storage and assembly efficient, we first create its sparsity pattern (`SparsityPattern`), and then use it to initialize the matrix itself.

Assembly. The efficient assembly of the system arising from finite elements is not trivial, and the associated algorithm is worth discussing in detail.

Let us first consider the system matrix A. The naive (and wrong!) way to assemble it would be to iterate through its entries and compute them according to their definition (7). In pseudo-code:

```
for $i = 1, 2, \ldots, N_h$ do
    for $j = 1, 2, \ldots, N_h$ do
        $A_{ij} \leftarrow a(\varphi_j, \varphi_i) = \int_0^1 \mu(x)\varphi_j'(x)\varphi_i'(x)\,dx$
    end for
end for
```

The integrals in the innermost loop are not trivial to compute, and need to be addressed with appropriate numerical methods. We begin by splitting the integral over Ω into the sum of *local integrals* over individual elements K_c:

```
$A \leftarrow 0$
for $i = 1, 2, \ldots, N_h$ do
    for $j = 1, 2, \ldots, N_h$ do
        for $c = 1, 2, \ldots, N_{\text{el}}$ do
            $A_{ij} \leftarrow A_{ij} + \int_{K_c} \mu(x)\varphi_j'(x)\varphi_i'(x)\,dx$
        end for
        end for
    end for
```

Then, we compute each local integral through a *quadrature formula*. For a generic function $g(x)$, a quadrature formula approximates its integral over a certain domain $D \subset \mathbb{R}^d$ as

$$\int_D g(x)\,dx \approx \sum_{q=1}^{n_q} g(x_q) w_q \ ,$$

where $x_q \in D$ are the n_q *quadrature points* and $w_q \in \mathbb{R}$ the associated *quadrature weights*. There exist many different quadrature rules, depending on the desired accuracy of the approximation, see e.g. [3, 4]. A quadrature formula is said to have *degree of exactness* p if it computes exactly the integral of polynomials of degree not greater than p. We will use Gauss-Legendre quadrature, which has degree of exactness equal to $2n_q - 1$. Typically, one wants to integrate exactly the product of basis functions (as it occurs for reaction terms in advection-diffusion-reaction problems), i.e. $g(x) = \varphi_j(x)\varphi_i(x)$, which is a polynomial of degree $2r$. This is obtained by setting $n_q = r + 1$ (which satisfies $2n_q - 1 \geq 2r$).

Inserting numerical integration into the previous matrix assembly algorithm, we get the following pseudo-code:

```
$A \leftarrow 0$
for $i = 1, 2, \dots, N_h$ do
    for $j = 1, 2, \dots, N_h$ do
        for $c = 1, 2, \dots, N_{\text{el}}$ do
            for $q = 1, 2, \dots, n_q$ do
                $A_{ij} \leftarrow A_{ij} + \mu(x_q^c)\varphi_j'(x_q^c)\varphi_i'(x_q^c)\, w_q^c$
            end for
        end for
    end for
end for
```

Notice that, since elements K_c may have different size (and shape, in 2D and 3D), the quadrature points and weights depend on the element (hence the superscript c).

Albeit straightforward, this algorithm is extremely inefficient. Indeed, the innermost computation is repeated $N_h^2 N_{\text{el}} n_q$ times, which becomes an extremely large number for even moderately refined meshes and/or two- or three-dimensional problems.

We can do much better by exploiting our knowledge of the support of the basis functions φ_i. On every element K_c, only a small number $n_{\text{loc}} \ll N_h$ of basis functions are different from zero (in 1D, $n_{\text{loc}} = r + 1$; see Figure 3). Therefore, the term $A_{ij}^{cq} = \mu(x_q^c)\varphi_j'(x_q^c)\varphi_i'(x_q^c)\, w_q^c$ is zero at most iterations (i.e., most iterations of the previous loop actually do nothing). To incorporate this information in the assembly algorithm, we start by reversing the order of the loops, so that we first iterate over the elements

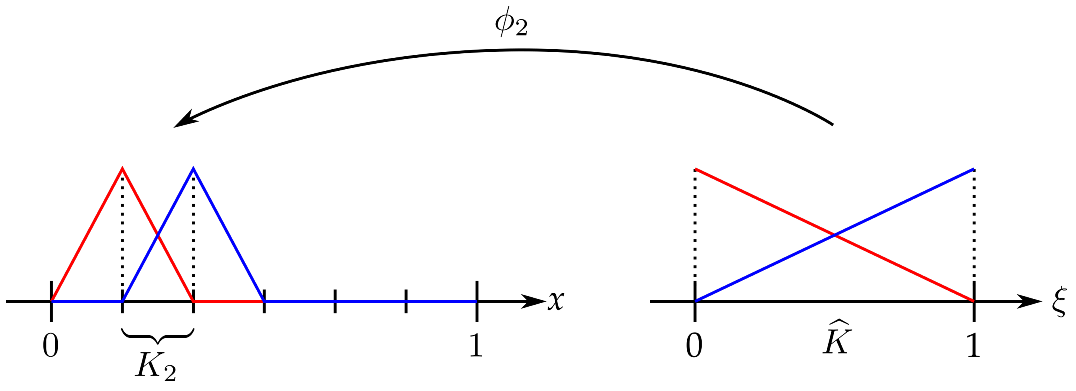


Figure 3: For the case $r = 1$: basis functions locally supported on the element K_2 (left) and corresponding reference basis functions on $\widehat{K}$ (right).

K_c. Then, we can optimize the loops over i and j by only looping over the set $\mathcal{I}_c$ of basis function indices that are not zero over the element K_c (the *local indices* or *local DoF indices* in `deal.II` jargon). We get:

```
$A \leftarrow 0$
**for** $c = 1, 2, \dots, N_{\text{el}}$ **do**
    **for** $i \in \mathcal{I}_c$ **do**
        **for** $j \in \mathcal{I}_c$ **do**
            **for** $q = 1, 2, \dots, n_q$ **do**
                $A_{ij} \leftarrow A_{ij} + \mu(x_q^c)\varphi_j'(x_q^c)\varphi_i'(x_q^c)\, w_q^c$
            **end for**
        **end for**
    **end for**
**end for**
```

This is orders of magnitude better than before, since the innermost iterations now run $n_{\text{loc}}^2 N_{\text{el}} n_q \ll N_h^2 N_{\text{el}} n_q$ times.

Internally, the evaluation of the quadrature weights, nodes and of the basis functions (or their derivatives) on quadrature nodes is made more efficient by only storing them for a *reference element* $\widehat{K}$ (see Figures 2 and 3). Each mesh element K_c is mapped onto the reference element by a map ϕ_c. This is automatically taken care by `deal.II` classes, so we will not cover the details here, but you can refer to `deal.II`'s documentation for the classes `Quadrature`, `FiniteElement` and `Mapping`.

We now observe that every element K_c contributes only to a few entries of the matrix A (those corresponding to the locally supported basis functions). In practice, it is often convenient to separate the computation of those contributions from their actual addition into the global matrix. To this end, we introduce a *local matrix* $A^c \in \mathbb{R}^{n_{\text{loc}} \times n_{\text{loc}}}$ to store the local contributions and add them to the global matrix only at the end of the assembly process. Notice that the local matrix is indexed with respect to the locally supported basis functions (so that we can defer the evaluation of the local-to-global map). The local matrix has entries

$$A^c_{kl} = \sum_{q=1}^{n_q} \mu(x^c_q)\varphi'_k(x^c_q)\varphi'_l(x^c_q)\, w_q \;. \tag{9}$$

Local matrices can overlap in the global matrix (see Figure 4), in correspondence of basis functions that are shared by multiple elements. In those cases, the contributions from different elements are added together.

Finally, the assembly of the system right-hand-side vector $\mathbf{f}$ follows the same strategy (except that it is a one-dimensional object, so we need one less loop). This gives rise to a local right-hand-side vector $\mathbf{f}^c \in \mathbb{R}^{n_{\text{loc}}}$, with the following elements:

$$\mathbf{f}^c_k = \sum_{q=1}^{n_q} f(x^c_q)\varphi_k(x^c_q)\, w_q \;. \tag{10}$$

Putting everything together, we get the following pseudo-code for the system assembly:

```
$A \leftarrow 0$                                                        ▷ initialize global matrix
$\mathbf{f} \leftarrow 0$                                                        ▷ initialize global vector

for $c = 1, 2, \ldots, N_{\text{el}}$ do
    $A^c \leftarrow 0$                                                   ▷ initialize local matrix
    $\mathbf{f}^c \leftarrow \mathbf{0}$                                                   ▷ initialize local vector

    for $q = 1, 2, \ldots, n_q$ do
        for $k = 1, 2, \ldots, n_{\text{loc}}$ do
            for $l = 1, 2, \ldots, n_{\text{loc}}$ do
                $A^c_{kl} \leftarrow A^c_{kl} + \mu(x^c_q)\,\varphi'_k(x^c_q)\,\varphi'_l(x^c_q)\,w_q$
            end for
            $\mathbf{f}^c_k \leftarrow \mathbf{f}^c_k + f(x^c_q)\varphi_k(x^c_q)\,w_q$
        end for
    end for

    add entries of local matrix $A_c$ to global matrix $A$
    add entries of local vector $\mathbf{f}_c$ to global vector $\mathbf{f}$
end for
```

To wrap up: in order to obtain the approximate solution to problem (1), we have to assemble the matrix A and the vector $\mathbf{f}$. We do so by computing the local matrix and vector for each element using numerical quadrature, as defined in (9) and (10). All the local contributions are added together, and then the resulting linear system (6) is solved to compute the vector of control variables $\mathbf{u}$.

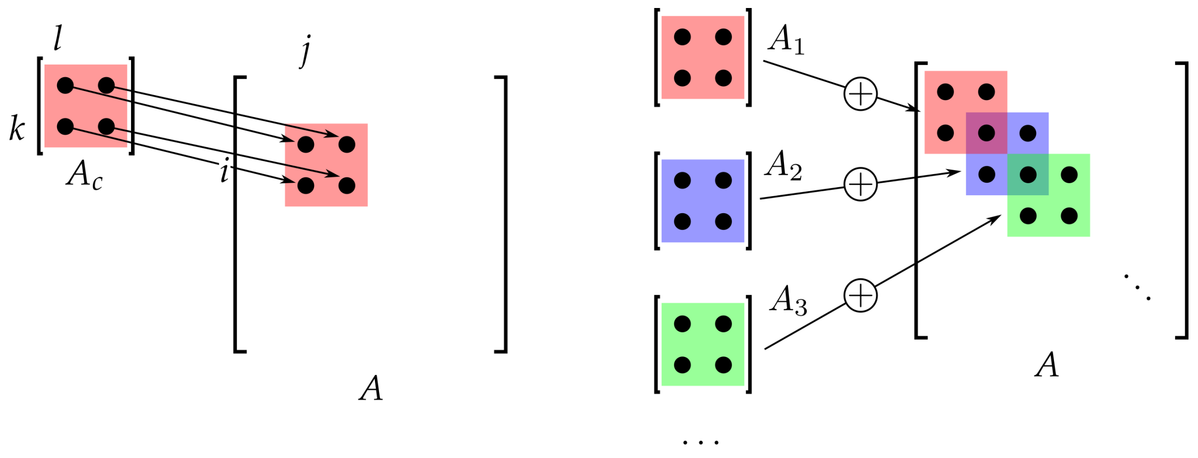


Figure 4: For the case $r = 1$: correspondence between the local matrix A_c for one element and the global matrix A (left); matrices of multiple elements are added together to form the global matrix (right). Non-zero entries are represented by black bullets.

All this translates directly to the code of the `Poisson1D::assemble_system` method. For each element, we construct the local matrix and vector using the definitions (9) and (10). The sum over quadrature points translates to a loop over quadrature points in the code, nested within the loop over elements. For each element K_c and quadrature node x_q^c, we need to compute the terms of (9) and (10). `deal.II` offers a class that has exactly this purpose, called `FEValues` (documentation: https://dealii.org/9.5.0/doxygen/deal.II/classFEValues.html). The class is used to "walk" over the mesh elements, and for each of them computes the needed quantities for each quadrature point.

Dirichlet boundary conditions. To impose Dirichlet conditions on the boundary, we assemble the linear system as if no boundary conditions were imposed, i.e. we include the basis functions corresponding to the boundary nodes (the first and the last). This way, we obtain an "extended" matrix of $\widehat{A}$ of size $(N_h+2)\times(N_h+2)$, and an "extended" vector of unknowns $\widehat{\mathbf{u}} = \left(U_0, U_1, U_2, \ldots, U_{N_h}, U_{N_h+1}\right)^T \in \mathbb{R}^{N_h+2}$. The control variables U_0 and U_{N_h+1} correspond to the first and last vertex of the mesh. In this setting, the Dirichlet conditions can be written as

$$U_0 = 0 \qquad U_{N_h+1} = 0\,. \tag{11}$$

Therefore, we can impose them by discarding the first and last equation from $\widehat{A}$ and replacing them with (11). `deal.II` offers several ways to achieve this. In the simplest approach, we store in an `std::map` the relations (11) (in this case, the map would contain the two pairs (0, 0) and ($N_h + 1$, 0)). Finally, the `deal.II` function `MatrixTools::apply_boundary_values` modifies the linear system to account for conditions (11).

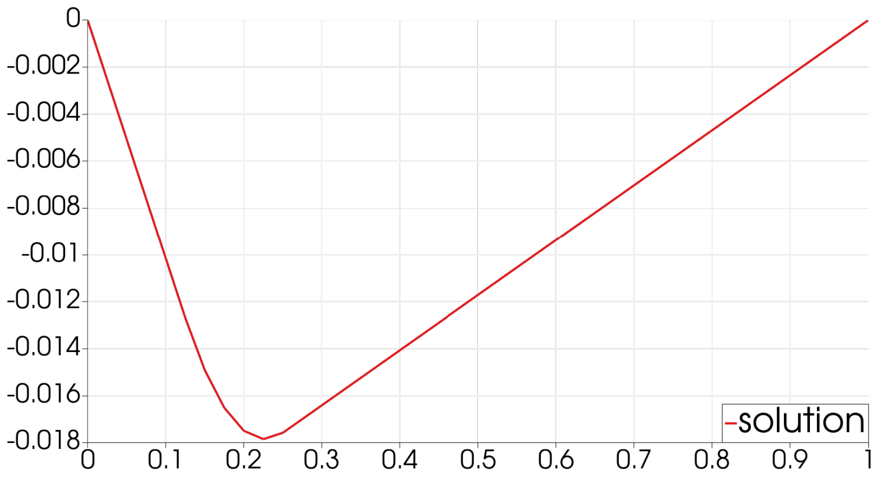


Figure 5: Approximate solution to problem (1), obtained with finite elements of degree 1 and using 40 elements. This plot is obtained by opening the solution in Paraview and applying the "Plot over line" filter.

Linear solver. The linear system can be solved with a direct or an iterative method. Generally speaking, systems arising from the finite element discretization of differential problems are large and ill-conditioned. The choice of the linear solver is therefore critical.

In our example, the matrix A is symmetric and positive definite. Therefore, it is convenient to solve the associated system with the *conjugate gradient* (CG) method. Be careful when choosing the stopping criterion for the iterative method: iterative solvers yield an approximation of the solution of the linear system, and if the tolerance used is too high this might introduce a significant additional error in the computation.

Notice that for sufficiently small problems (such as the one we consider) a direct solver can also be used (for instance, based on LU factorization). In this case, we use the CG method for convenience of implementation, since `deal.II` v9.5.1 offers direct solvers only through wrappers to the Trilinos library, which require a slightly more advanced management of linear algebra structures.

In our test case, the linear system is relatively small, so that no preconditioning is needed. We will see how to use preconditioners later in the course.

Post-processing. Here we just write the results of our computation to a file. The file can then be visualized with any scientific visualization software, such as Paraview (official website: `https://www.paraview.org/`). To be fair, Paraview is not very effective for 1D data (such as in this example), and more appropriate software (e.g. Python's plotting packages) should be used for quality plots. However, Paraview will become very useful when considering 2D or 3D problems.

Figure 5 displays the numerical solution to problem (1). To obtain the plot, after opening the solution file `output-40.vtk` in Paraview, the filter "Plot over line" was applied.

Verification. An important aspect of post-processing is verifying the correctness of the solution, which means verifying the correctness and reliability of the implemented software. Indeed, software frequently has bugs, and may produce incorrect results. Therefore, the results of numerical simulations should **never** be trusted blindly, and you should always ask yourselvers whether the results are reasonable or not.

Unfortunately, most problems do not admit any analytical expression for the solution, so that it is not possible to compare the numerical and exact solutions (after all, the purpose of numerics is to solve problems that we cannot solve otherwise). This issue can be overcome with the method of *manufactured solutions* and *convergence analysis*, which we will address in the next exercise.

Nonetheless, we can compare the numerical solution against our expectations based on our physical understanding of the problem at hand. For example, we can check that boundary conditions are verified (as is the case here the numerical solution is equal to zero on both ends of the domain). Moreover, if we interpret the problem as that of the displacement of an elastic cord subject to the vertical force f, we can expect to observe a behavior similar to the one depicted in Figure 5.

Verification and validation

When writing a finite element program, it is extremely common to make mistakes, even for very experienced programmers and computational scientists.

To address this, it is important on the one hand to adopt programming practices that reduce the amount of errors. These include, for example, automatic testing, meticulous documentation, and in general the adherence to best practices for scientific computing software [5].

On the other hand, it is also important to learn how to assess the validity of the solver, that is to verify *a posteriori* that it produces the correct solution. In this respect, application codes should aim at two objectives: *software verification* and *model validation* [1].

Model validation

Model validation is the very general task of assessing whether the mathematical model, discretized with appropriate numerical methods, and implemented in software, yields results that are consistent with the real-world system that it represents.

This task is fundamental to assess the reliability of simulation software for practical applications, and thus all commercial finite element solvers go through this during their development. However, validation is also quite difficult. Proper validation requires access to experimental results (which might require the software developers to run the experiments themselves). Moreover, one usually has to account for uncertainty arising from many sources (including measurement errors, choice of the model, choice of the discretization method and its parameters). Generally speaking, validation is usually a very interdisciplinary effort.

We carried out a very preliminary form of model validation at the end of the previous exercise, by checking that the numerical solution satisfies known constraints (such as boundary conditions), and that it agreed with the physical intuition we had on the meaning of the PDE model.

Software verification

Software verification (or code verification) is the process of verifying that the numerical solution is consistent with the mathematical model and numerical methods used to solve the target problem. It has a smaller scope than model validation, which checks the correctness of the entire pipeline, from the continuous model to the approximate solution, and it is therefore a somewhat easier task, although equally important.

Software verification is typically done by solving a problem for which the exact solution is known. The numerical solution will be different from the exact one, due to the approximate nature of the finite element solver and of numerical methods in general. However, we can verify that the error reduces as the mesh resolution is reduced, and that it does so with a convergence rate that is consistent with the theoretical properties of the method we use.

However, for most problems, it is not possible to compute an expression of the exact solution. To work around this issue, we can rely on the *method of manufactured solutions*: we arbitrarily choose an expression of the exact solution, we plug that expression into the PDE and compute the forcing term (and possibly the problem coefficients and/or its boundary conditions) to match the exact solution.

This is a fairly simple task, but it needs to be done with some care. To begin with, the exact solution should satisfy the boundary conditions.

Additionally, if the exact solution is an element of the function space where we look for the solution (for instance, if it is a polynomial of degree less than or equal to the discretization degree), the numerical solution will be equal to the exact solution (up to errors due to linear solvers and finite arithmetic), thanks to Céa's lemma [2]. This sort of situation is sometimes referred to as *patch test*: the error is always very close to zero, regardless of the mesh size.

While useful in itself, a patch test does not verify that the solver exhibits the convergence properties expected from the theory. For this reason, one usually runs a *convergence test*, wherein we choose a solution that cannot be represented by the finite element space. The error will no longer be zero, and will (hopefully) behave as predicted by the theory as we vary the mesh resolution. To this end, we can rely on transcendental functions such as trigonometric functions or exponentials (both having the added benefit of being easy to differentiate).

The agreement of the numerical solution with the exact one can be verified both quantitatively (e.g. by computing an appropriate norm of the error) and qualitatively, by visualizing and comparing the two solutions. Similarly, convergence properties can be assessed quantitatively (estimating the convergence order) and visually (plotting the convergence order against the theoretically expected trend). It is very important to always do both, to have an all-round picture of our simulation results.

This is especially true if something goes wrong. For example, if you don't get the expected convergence order, it might be because of an error in boundary conditions, which will become evident when plotting the solution.

The next exercise aims at implementing a convergence test through the method of manufactured solutions.

Exercise 2.

Consider again the general Poisson problem (1), with $\mu = 1$ and $f(x) \in L^2(\Omega)$ a generic function.

2.1. Assuming that the exact solution to (1) is $u_{\text{ex}}(x) = \sin(2\pi x)$, determine the expression of $f(x)$.

Solution. There holds $u_{\text{ex}}(0) = u_{\text{ex}}(1) = 0$, so that (1b) is satisfied. Moreover,

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

so that

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

2.2. Starting from the solution of Exercise 1, implement a new method that computes the $L^2(\Omega)$ or $H^1(\Omega)$ norm (depending on the input argument) of the error between the computed solution and the exact solution:

$$e_{L^2} = \|u_h - u_{\text{ex}}\|_{L^2} = \sqrt{\int_0^1 |u_h - u_{\text{ex}}|^2 \, dx} \; ,$$
$$e_{H^1} = \|u_h - u_{\text{ex}}\|_{H^1} = \sqrt{\int_0^1 |u_h - u_{\text{ex}}|^2 \, dx + \int_0^1 |\boldsymbol{\nabla} u_h - \boldsymbol{\nabla} u_{\text{ex}}|^2 \, dx} \; .$$

The new method should have the signature

```
double
Poisson1D::compute_error(
  const VectorTools::NormType &norm_type,
  const Function<dim>         &exact_solution) const
```

Solution. See the file `src/Poisson1D.cpp`.

2.3. With polynomial degree $r = 1$, solve the problem (1) with finite elements, setting $N_{\text{el}} = 10, 20, 40, 80, 160$. Compute the error in both the $L^2(\Omega)$ and $H^1(\Omega)$ norms as a function of the mesh size h, and compare the results with the theory.

Solution. See the file `src/exercise-02.cpp` for the implementation. The solution for this test is plotted in Figure 6.

From the convergence theory, we expect that, if $u_{\text{ex}} \in H^{r+1}(\Omega)$ (in this case, $u_{\text{ex}} \in H^p$ for all $p = 1, 2, 3, \dots$),

$$e_{L^2} \leq C_1 h^{r+1} |u_{\text{ex}}|_{H^{r+1}} \; ,$$
$$e_{H^1} \leq C_2 h^{r} |u_{\text{ex}}|_{H^{r+1}} \; ,$$

where $h = 1/N_{\text{el}}$ is the size of mesh elements. Therefore, we expect the error in the $L^2(\Omega)$ norm to converge to zero with order $r + 1 = 2$, and the error in the $H^1(\Omega)$ norm to converge to zero with order $r = 1$.

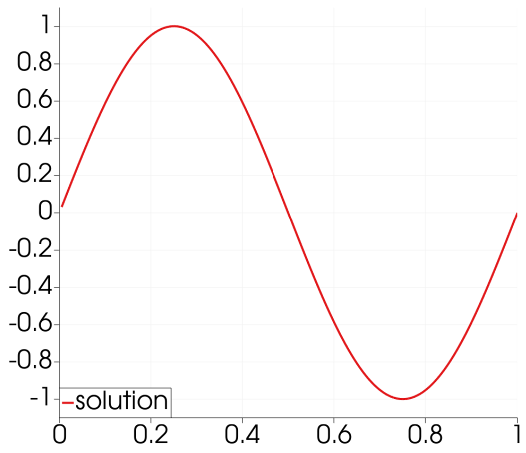
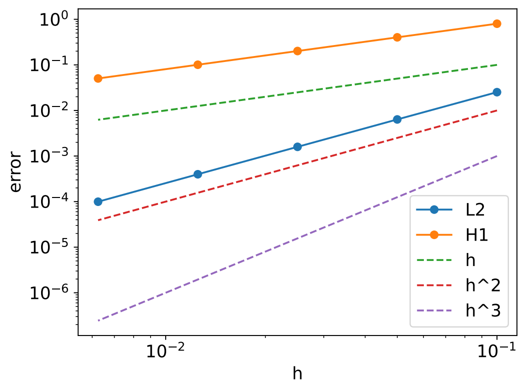
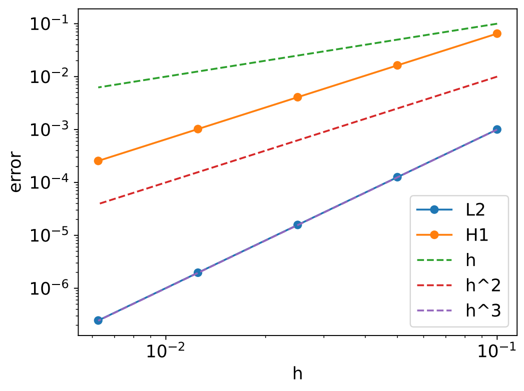


Figure 6: Solution u_h of Question 3, computed using $r = 1$ and $N_{\text{el}} = 160$.



(a)



(b)

Figure 7: Questions 3 and 4. Error in the L^2 and H^1 norms against h for $r = 1$ (a) and $r = 2$ (b).

To verify this, we use the `deal.II` class `ConvergenceTable` to compute the estimated convergence order. Notice that the class works as expected as long as the mesh size between subsequent solutions is halved. We obtain the following results:

h	e_{L^2}	convergence order e_{L^2}	e_{H^1}	convergence order e_{H^1}
0.1	2.5199e-02	-	8.0096e-01	-
0.05	6.3529e-03	1.99	4.0231e-01	0.99
0.025	1.5916e-03	2.00	2.0139e-01	1.00
0.0125	3.9811e-04	2.00	1.0072e-01	1.00
0.00625	9.9539e-05	2.00	5.0364e-02	1.00

The estimated convergence orders for the two errors are in agreement with the theory.

We can draw the same conclusions by plotting the error against h in the log-log plane. Indeed, we want to assess whether

$$e_{L^2} \approx C_1 |u_{\text{ex}}|_{H^{r+1}} h^{r+1}$$

holds. By taking the logarithm of both sides, we get

$$\log e \approx \log \left(C_1 |u_{\text{ex}}|_{H^{r+1}} \right) + (r+1) \log h \ ,$$

so that the logarithm of the L^2 error is a linear function of the logarithm of h, with slope given by the convergence rate $r + 1 = 2$. Therefore, in the log-log plane, the plot of the L^2 error should be a line parallel to the one obtained by plotting the function h^{r+1}. Similar considerations hold for the H^1 error, which should be represented by a line of slope $r = 1$.

The errors against h are written to a file `convergence.csv`, which can be opened in any 2D graphing software (MATLAB, Python, ...). For convenience, you can find a Python script at `scripts/plot-convergence.py`, that can be called as `./plot-convergence.py convergence.csv` and produces a PDF file `convergence.pdf` containing the plot. The script requires that the package `matplotlib` is available: you can download and install it by running `pip install matplotlib`.

The result is displayed in Figure 7a. We can see how the error in the L^2 norm is represented by a straight line parallel to the one representing h^2, as expected, confirming that it tends to zero with order $r + 1 = 2$. Similarly, the H^1 error tends to zero with order $r = 1$ as expected.

2.4. Repeat the previous point setting $r = 2$.

Solution. By changing the polynomial degree to $r = 2$, we obtain the following convergence table:

h	e_{L^2}	convergence order e_{L^2}	e_{H^1}	convergence order e_{H^1}
0.1	1.0028e-03	-	6.5007e-02	-
0.05	1.2590e-04	2.99	1.6319e-02	1.99
0.025	1.5754e-05	3.00	4.0840e-03	2.00
0.0125	1.9698e-06	3.00	1.0213e-03	2.00
0.00625	2.4624e-07	3.00	2.5533e-04	2.00

The corresponding error plots are displayed in Figure 7b. From, both, we can observe the expected convergence rates: the L^2 error tends to zero with rate $r+1=3$, and the H^1 error tends to zero with rate $r=2$.

2.5. Let us now redefine the forcing term as

$$f(x) = \begin{cases} 0 & \text{if } x \leq \dfrac{1}{2}\,, \\ -\sqrt{x-\dfrac{1}{2}} & \text{if } x > \dfrac{1}{2}\,. \end{cases}$$

The exact solution in this case is

$$u_{\text{ex}}(x) = \begin{cases} Ax & \text{if } x \leq \dfrac{1}{2}\,, \\ Ax + \dfrac{4}{15}\left(x-\dfrac{1}{2}\right)^{\frac{5}{2}} & \text{if } x > \dfrac{1}{2}\,, \end{cases}$$
$$A = -\frac{4}{15}\left(\frac{1}{2}\right)^{\frac{5}{2}}\,.$$

Check the convergence order of the finite element method in this case, with polynomial degrees $r=1$ and $r=2$. What can you observe?

Solution. The solution for this test is plotted in Figure 8. For $r=1$, the convergence table reads

h	e_{L^2}	convergence order e_{L^2}	e_{H^1}	convergence order e_{H^1}
0.1	3.1998e-04	-	1.0198e-02	-
0.05	7.9874e-05	2.00	5.1018e-03	1.00
0.025	1.9899e-05	2.01	2.5514e-03	1.00
0.0125	4.9486e-06	2.01	1.2757e-03	1.00
0.00625	1.2281e-06	2.01	6.3788e-04	1.00

and the associated convergence plot is displayed in Figure 9a. We observe once again the expected convergence rates.

If we set $r=2$, however, we obtain the following convergence table:

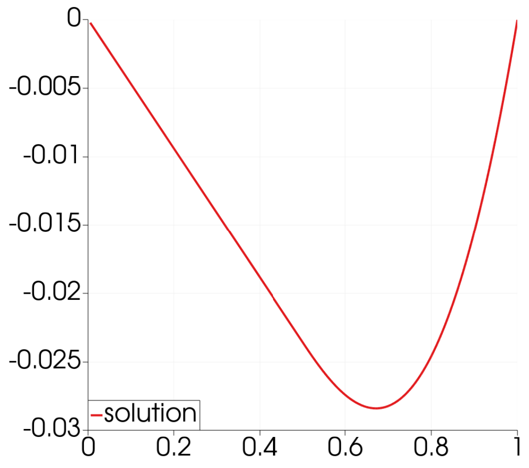
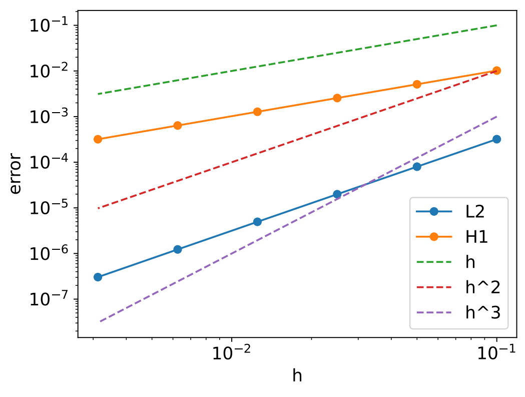


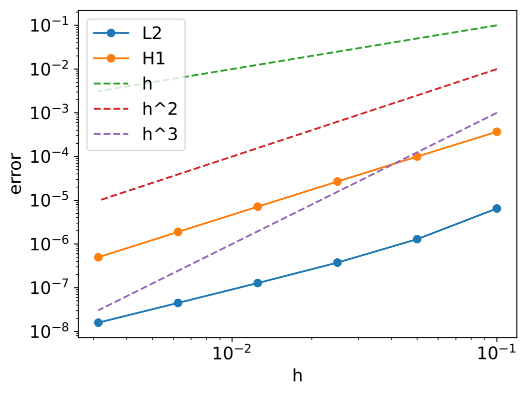
Figure 8: Solution u_h of Exercise 1.5, computed using $r = 1$ and $N_{\text{el}} = 160$.

h	e_{L^2}	convergence order e_{L^2}	e_{H^1}	convergence order e_{H^1}
0.1	6.4841e-06	-	3.6856e-04	-
0.05	1.2899e-06	2.33	9.9987e-05	1.88
0.025	3.7520e-07	1.78	2.6826e-05	1.90
0.0125	1.2792e-07	1.55	7.1384e-06	1.91
0.00625	4.4981e-08	1.51	1.8882e-06	1.92

and the convergence plot of Figure 9b. The convergence rates are not the optimal ones in this case. Indeed, the optimal convergence rates ($r + 1$ in the L^2 norm and r in the H^1 norm) can be observed only if the solution belongs to the space $H^{r+1}(\Omega)$. In this case, there holds $u \in H^2(\Omega)$ but $u \notin H^3(\Omega)$, so that the observed convergence rates, for the case $r = 2$, are slower than the optimal ones.



(a)



(b)

Figure 9: Exercise 1.5. Error in the L^2 and H^1 norms against h, for $r = 1$ (a) and $r = 2$ (b).

Possibilities for extension

Reading parameters from file. In the simple implementation proposed here, all the problem parameters (the coefficient μ, the number of mesh elements, $\dots$) are part of the source code. Therefore, to change them, a user would need to modify and recompile the program. To avoid this, `deal.II` offers a convenient interface for reading parameters from an external text file, without the need to recompile, through the class `ParameterHandler`. With the help of `deal.II`'s documentation and tutorials, add support for reading parameters from file to the program of Exercise 1.

Profiling. After writing and testing a program, we usually want to measure its performance to find bottlenecks and possibly introduce optimizations. `deal.II` offers the class `TimerOutput` for a simple way of profiling the code. With the help of `deal.II`'s documentation, add profiling to the program of Exercise 1, to measure the execution time of initialization, system assembly and system solution. (For more sophisticated profiling, you can check out `gperftools` at `https://github.com/gperftools/gperftools`).

Patch test. Repeat Exercise 2 choosing an exact solution that can be represented by the finite element space, and observe the behavior of the error as the mesh is refined.

Parsing functions from file. As seen above, large simulation programs usually allow problem parameters to be defined in an external file (as opposed to hard-coded into the source files). This can also be done for functional data. `deal.II` implements a class named `FunctionParser` for this precise purpose. With the help of `deal.II`'s documentation, modify the code from Exercise 2 so that it reads all relevant parameters from file, including the definitions of functional data.

References

[1] W. L. Oberkampf and C. J. Roy. *Verification and validation in scientific computing.* Cambridge university press, 2010.

[2] A. Quarteroni. *Numerical models for differential problems*, volume 16. Springer, 2017.

[3] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

[4] Wikipedia: Gaussian quadrature. `https://en.wikipedia.org/wiki/Gaussian_quadrature`.

[5] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, 2014.