Numerical Methods for Partial Differential Equations
A.Y. 2024/2025

Laboratory 01
Finite Element method for the Poisson equation in 1D

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). Then, 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, 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 use the fact that $u(0) = u(1) = 0$, 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 $u, v \in V = H_0^1(\Omega) = \{v \in L^2(\Omega) \colon v' \in L^2(\Omega),\ v(0) = v(1) = 0\}$.

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 $N+1$ subintervals (*elements*) K_i, $i = 1, 2, \ldots, N+1$. Let X_h^r be the space of piecewise polynomials over the mesh elements, that is

$$X_h^r(\Omega) = \{ v_h \in C^0(\bar{\Omega}) \ : \ v_h(x) \in \mathbb{P}_r \quad \forall x \in K_i, \ \forall i = 1, 2, \ldots, N+1 \} .$$

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 $N_h = \dim(V_h)$.

There holds $\dim(X_h^r) = rN + r + 1$, and $N_h = \dim(V_h) \leq \dim(X_h^r)$ (in this case, $N_h = rN + r - 1$, due to Dirichlet boundary conditions, so that, if $r = 1$, $N_h = N$).

Let $\varphi_i(x)$, for $i = 1, 2, \ldots, N_h$, 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 $i = 1, 2, \ldots, N_h$. Then, the discrete weak formulation (4) rewrites as: find U_j, for $j = 1, 2, \ldots, N_h$, such that

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

Equations (5) can be rewritten as a linear system

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

where

$$\mathbf{u} \in \mathbb{R}^n \qquad \mathbf{u} = \left(U_1, U_2, \ldots, U_{N_h}\right)^T \ ,$$

$$A \in \mathbb{R}^{N_h \times N_h} \qquad A_{ij} = a(\varphi_j, \varphi_i) = \int_0^1 \mu(x)\varphi_j'(x)\varphi_i'(x)\, dx \ , \tag{7}$$

$$\mathbf{f} \in \mathbb{R}^{N_h} \qquad \mathbf{f}_i = F(\varphi_i) = \int_0^1 f(x)\varphi_i(x)\, dx \ . \tag{8}$$

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 $N + 1 = 20$.

Solution. See the file `src/lab-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.

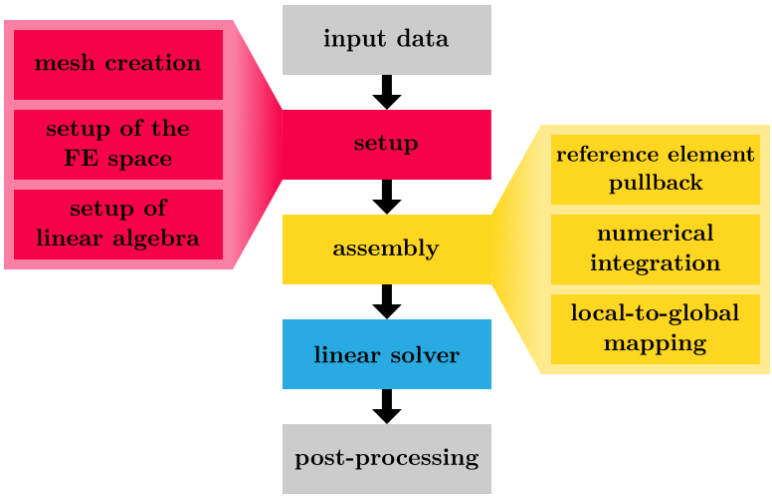

Figure 1: Schematic representation of the main steps in a (linear, time independent) finite element solver.

We implement a class, `Poisson1D` (files `src/Poisson1D.hpp` and `src/Poisson1D.cpp`), to wrap all the above steps. For each of the steps, the class exposes a method: `setup()`, `assemble()`, `solve()`, `output()`. The class stores:

- the discretization settings (number of elements and polynomial degree);
- the mesh, through a `dealii::Triangulation`;
- a member representing the finite element space used for the discretization, using the generic class `dealii::FiniteElement`;
- a member that manages the numbering of degrees of freedom, using the class `dealii::DoFHandler`;
- a member that represents the quadrature rule, of type `dealii::Quadrature`;
- several members that represent 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` in the context of `deal.II`). For this first tests, we consider `dim = 1`: a static member is introduced 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

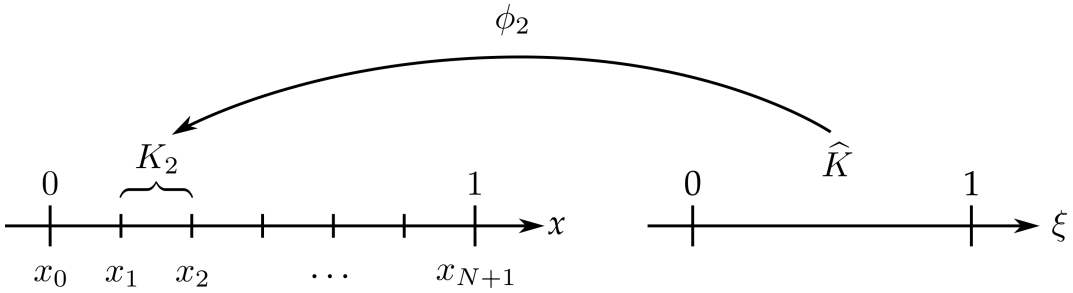

Figure 2: Left: partition of the domain Ω into a mesh with equally sized elements. Right: reference element.

the mesh can be done inside the program itself, by using the tools of the `GridGenerator` namespace (documentation: https://dealii.org/current/doxygen/deal.II/namespaceGridGenerator.html). Alternatively, the mesh can be created externally and loaded 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 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 $c = 1, 2, \ldots, N$ **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 numerical *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. [1, 2]. 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 $i = 1, 2, \ldots, N_h$ **do**
  for $j = 1, 2, \ldots, N_h$ **do**
    for $c = 1, 2, \ldots, N$ **do**
      for $q = 1, 2, \ldots, n_q$ **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

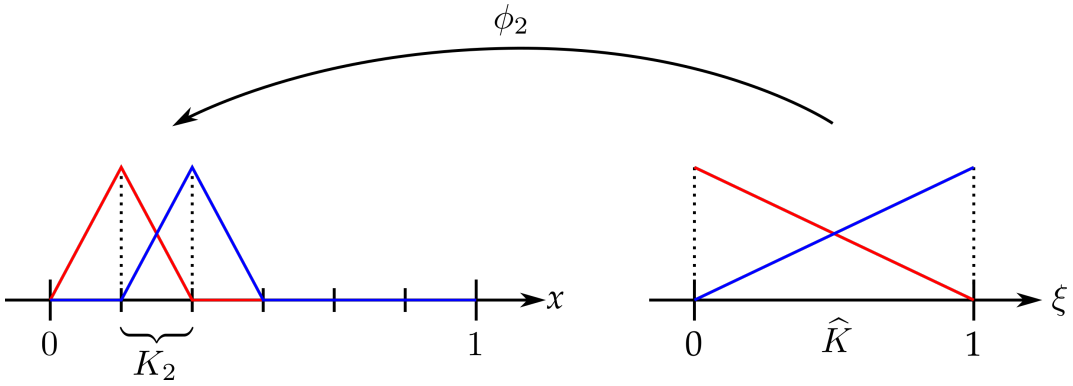

Figure 3: For the case $r = 1$: basis functions locally supported on the element K_2 (left) and corresponding reference basis functions on $\widehat{K}$ (right).

computation is repeated $N_h^2 N n_q$ 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. Indeed, by construction of the basis, 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 K_c. Then, we can optimize the loops over i and j by only iterating through 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 $c = 1, 2, \ldots, N$ **do**
  for $i \in \mathcal{I}_c$ **do**
    for $j \in \mathcal{I}_c$ **do**
      for $q = 1, 2, \ldots, n_q$ **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 $n_{\text{loc}}^2 N n_q \ll N_h^2 N n_q$ times.

We are left only with a minor inconvenience: with this implementation, we would need to keep track of the basis functions, quadrature points and weights on every element. Instead, we notice that all elements can be obtained as the image of a *reference element*

$\widehat{K} = (0,1)$ (see Figure 2) through a linear map

$$\phi_c : \widehat{K} \to K_c \qquad\qquad \widehat{K} \ni \xi \mapsto \phi_c(\xi) \in K_c \ ,$$

where we used ξ to denote points in the reference element $\widehat{K}$ (from here on, the hat $\widehat{\ }$ will always denote quantities in reference configuration). We will denote the derivative of this map as

$$J_c(\xi) = \phi_c'(\xi) \ .$$

The few basis functions that have support over K_c can be expressed as the image, through ϕ_c, of some reference basis functions $\widehat{\varphi}_{\widehat{i}}$, with $\widehat{i} = 1, 2, \ldots, n_{\text{loc}}$, defined on the reference element $\widehat{K}$ (see Figure 3). If φ_i is a locally supported basis function on K_c,

$$\begin{aligned} \varphi_i(\phi_c(\xi)) &= \widehat{\varphi}_{\widehat{i}_c}(\xi) \ , \\ \varphi_i'(\phi_c(\xi)) &= \widehat{\varphi}_{\widehat{i}_c}'(\xi)\,(J_c(\xi))^{-1} \ . \end{aligned}$$

In the above, $\widehat{i}_c$ is the *local index* of the i-th basis function for the element K_c. The association of $\widehat{i}_c$ to the corresponding global index i is known as *local-to-global index map* and stored in an *ID array*.

In a similar way, the quadrature points and weights on K_c can be expressed as the image of reference quadrature points ξ_q and weights $\widehat{w}_q$:

$$\begin{aligned} x_q^c &= \phi_c(\xi_q) \ , \\ w_q^c &= \widehat{w}_q J_c(\xi_q) \ , \\ \int_{K_c} g(x)\,dx &\approx \sum_{q=1}^{n_q} g(\phi_c(\xi_q))\,\widehat{w}_q J_c(\xi_q) \ . \end{aligned}$$

Therefore, it is enough to store for every element K_c the *reference-to-current map* ϕ_c (and its derivative), and define basis functions and quadrature formula weights and nodes only on the reference element.

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

$$A^c_{kl} = \sum_{q=1}^{n_q} \mu(x_q^c) \underbrace{\left(\widehat{\varphi_k}'(\xi_q) J_c(\xi_q)^{-1}\right)}_{(I)} \underbrace{\left(\widehat{\varphi_l}'(\xi_q) J_c(\xi_q)^{-1}\right)}_{(II)} \underbrace{\widehat{w}_q\, J_c(\xi_q)}_{(III)} \ . \tag{9}$$

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:

$$\mathbf{f}_k^c = \sum_{q=1}^{n_q} f(x_q^c)\widehat{\varphi_k}(\xi_q)\,\widehat{w}_q\,J_c(\xi_q)\;. \tag{10}$$

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 $c = 1, 2, \ldots, N$ **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**
        $A_{kl}^c \leftarrow A_{kl}^c + \mu(x_q^c)\left(\widehat{\varphi_k}'(\xi_q)J_c(\xi_q)^{-1}\right)\left(\widehat{\varphi_l}'(\xi_q)J_c(\xi_q)^{-1}\right)\,\widehat{w}_q\,J_c(\xi_q)$
      end for
      $\mathbf{f}_k^c \leftarrow \mathbf{f}_k^c + f(x_q^c)\widehat{\varphi_k}(\xi_q)\,\widehat{w}_q\,J_c(\xi_q)$
    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 $\mathbf{u}$.

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 ξ_q, we need to compute the terms (I), (II) and (III) of (9). `deal.II` offers a

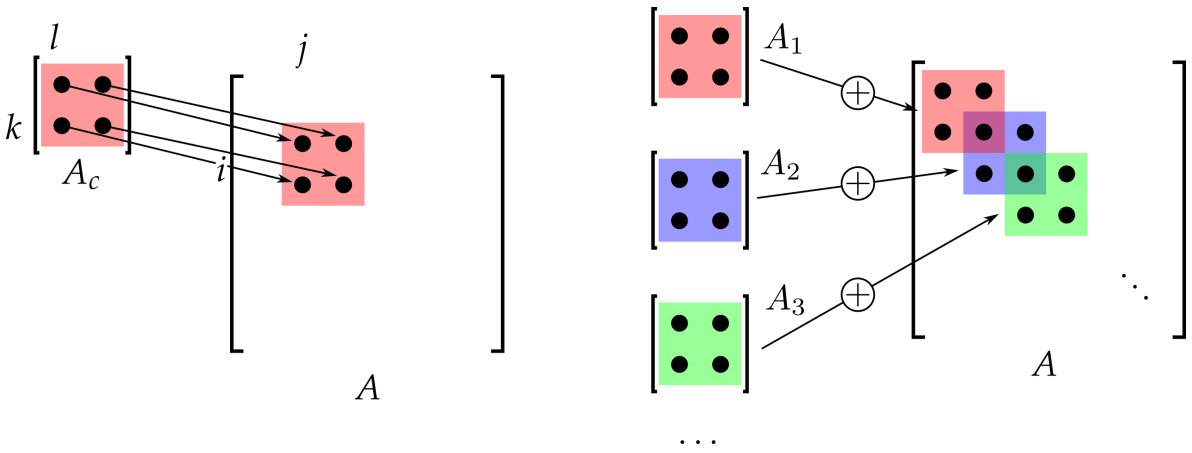

Figure 4: For the case $r = 1$: correspondence between the local matrix A_c for one element and the global matrix A (left); matrices of multiple elements are added together to form the global matrix (right). Non-zero entries are represented by black bullets.

class that has exactly this purpose, called `FEValues` (documentation: https://dealii.org/current/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 $\widehat{\mathbf{u}} = \big(U_0, U_1, U_2, \ldots, U_{N_h}, U_{N_h+1}\big)^T \in \mathbb{R}^{N_h+2}$. 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).

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.

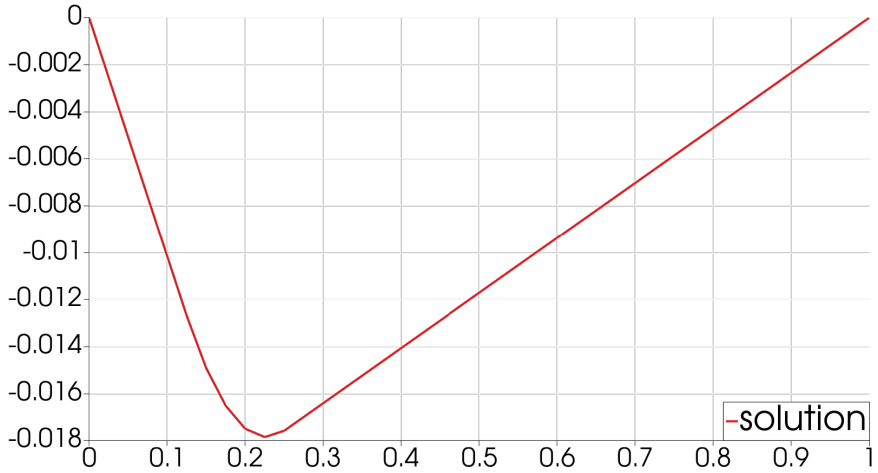

Figure 5: Approximate solution to problem (1), obtained with finite elements of degree 1 and using 40 elements. This plot is obtained by opening the solution in Paraview and applying the "Plot over line" filter.

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` 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 later laboratory lectures.

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.

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, ...) 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).

References

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

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