Numerical Methods for Partial Differential Equations
A.Y. 2023/2024

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:

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

By partial integration, 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 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. Write the associated linear system, and express the integrals involved by means of numerical quadrature formulas.

Solution. Let us introduce a uniform partition (the *mesh*, see Figure 1) 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 $N_h = \dim(V_h) = rN + r - 1$, due to Dirichlet boundary conditions. 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 checking (4) 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 $j = 1, 1, \ldots, N_h$, 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

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

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

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

To compute the approximate solution to problem (1), we can construct and solve the linear system (6). We now proceed to write the definitions of A and $\mathbf{f}$ in a way that is convenient for the software implementation.

The integrals in (7) and (8) can be rewritten as the sum of integrals over individual elements:

$$A_{ij} = \sum_{c=1}^{N+1} \int_{K_c} \mu(x)\varphi_j'(x)\varphi_i'(x)\, dx \ , \tag{9}$$

$$\mathbf{f}_i = \sum_{c=1}^{N+1} \int_{K_c} f(x)\varphi_i(x)\, dx \ . \tag{10}$$

The integrals above are performed on several different subintervals K_c. It is more convenient to express them as integrals over the same fixed interval. To do so, we observe that each element K_c is the image of a *reference element* $\widehat{K} = (0, 1)$ (see Figure 1) through a map

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

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). Then, we perform a change

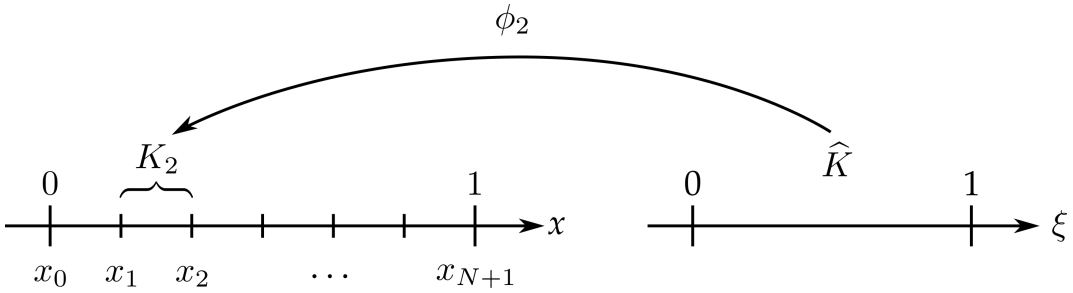

Figure 1: Left: partition of the domain Ω into a mesh with equally sized elements. Right: reference element.

of variables in the integrals in (9) and (10) using the maps ϕ_c:

$$A_{ij} = \sum_{c=1}^{N+1} \int_0^1 \mu(\phi_c(\xi))\, \varphi_j'(\phi_c(\xi))\, \varphi_i'(\phi_c(\xi))\, J_c(\xi)\, d\xi \,, \tag{11}$$

$$\mathbf{f}_i = \sum_{c=1}^{N+1} \int_0^1 f(\phi_c(\xi))\, \varphi_i(\phi_c(\xi))\, J_c(\xi)\, d\xi \,, \tag{12}$$

where $J_c(\widehat{x}) = \phi_c'(\widehat{x})$ is the Jacobian of the affine map from the reference element $\widehat{K}$ to the current one K_c (needed for the change of variable).

On each element K_c, only a small number $n_{\text{loc}} = 2r - 1$ basis functions are *locally supported*, i.e. they are different from zero when restricted to K_c. Therefore, in practice, each element contributes to only a few of the entries of the matrix A and vector $\mathbf{f}$. This information is collected by introducing a *local matrix* and a *local right-hand side vector*, that for each element contain only the local contributions to the system.

On the element K_c, the locally supported basis functions can be seen as the image of the reference basis functions $\widehat{\varphi}_{\widehat{i}}$, with $\widehat{i} = 0, 1, \ldots, n_{\text{loc}} - 1$, defined on $\widehat{K}$ (see Figure 2. 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 terms of the reference basis functions, the local matrix and vector can be defined as

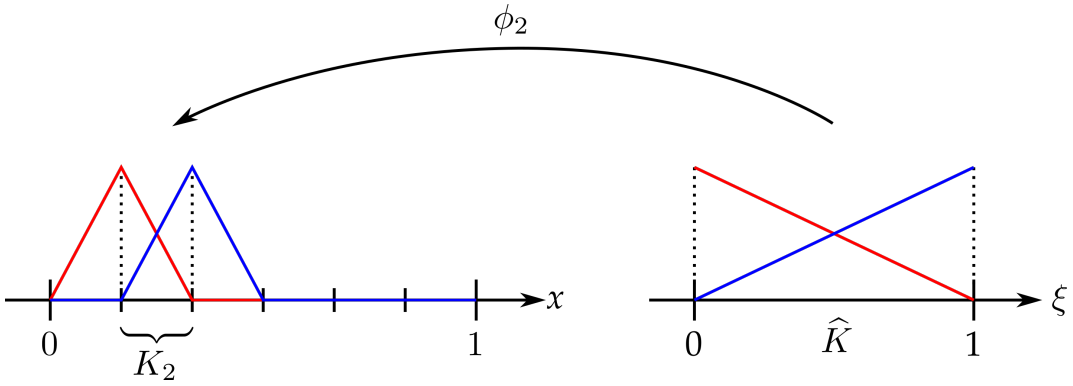

Figure 2: For the case $r = 1$: basis functions locally supported on the element K_2 (left) and corresponding reference basis functions on $\widehat{K}$ (right).

follows:

$$A_c \in \mathbb{R}^{n_{\text{loc}} \times n_{\text{loc}}} \qquad (A_c)_{kl} = \int_0^1 \mu(\phi_c(\xi)) \left(\widehat{\varphi}_k'(\xi)\,(J_c(\xi))^{-1}\right) \left(\widehat{\varphi}_l'(\xi)\,(J_c(\xi))^{-1}\right) J_c(\xi)\,d\xi\,, \tag{13}$$

$$\mathbf{f}_c \in \mathbb{R}^{n_{\text{loc}}} \qquad (\mathbf{f}_c)_k = \int_0^1 f(\phi_c(\xi))\,\widehat{\varphi}_k(\xi)\,J_c(\xi)\,d\xi\,. \tag{14}$$

The local matrix and vectors are indexed with respect to the locally supported basis functions, i.e. their rows (and columns) range from 0 to $n_{\text{loc}} - 1$. Then, they are added together to construct the global matrix A and right-hand side vector $\mathbf{f}$: the (k, l) element of the local matrix is added into the (i, j) element of the global matrix, where i and j are the global indices associated to the local indices k and l, respectively (see Figure 3). The same process is used to obtain the vector $\mathbf{f}$ from the local contributions $\mathbf{f}_c$.

It is not convenient to compute the integrals of (13) and (14) by hand (especially in two- or three-dimensional cases). Therefore, we approximate them by means of a *quadrature formula*. For a general function $g(\xi)$, its integral over $\widehat{K}$ can be approximated as

$$\int_0^1 g(\xi)\,d\xi \approx \sum_{q=0}^{n_q} g(\xi_q) w_q\,, \tag{15}$$

where $\xi_q \in \widehat{K}$ are the $n_q + 1$ *quadrature points* and $w_q \in \mathbb{R}$ the associated *quadrature weights*. Many different quadrature rules exist. We consider the Gauss-Legendre quadrature rules, which have the property that they integrate exactly (without approximation) polynomials of degree $2n_q + 1$ or lower (*degree of exactness*). Typically, one wants to integrate exactly the product of basis functions (as it occurs for reaction terms in advection-diffusion-reaction problems), i.e. $g(\xi) = \widehat{\varphi}_l(\xi)\widehat{\varphi}_k(\xi)$, which is a polynomial of degree $2p$. This is obtained by setting $n_q + 1 = p + 1$ (which satisfies $2n_q + 1 \geq 2p$).

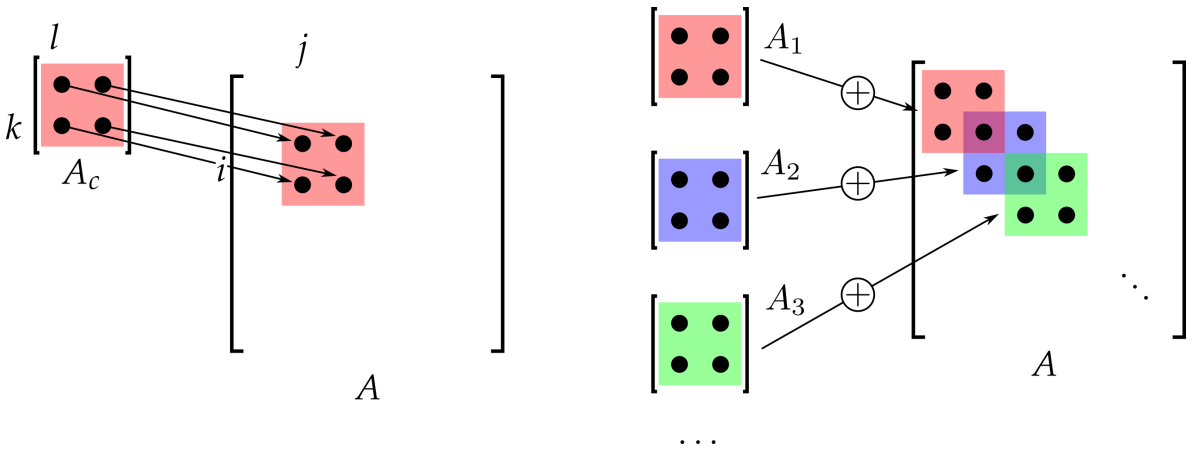

Figure 3: For the case $r = 1$: correspondence between the local matrix A_c for one element and the global matrix A (left); matrices of multiple elements are added together to form the global matrix (right). Non-zero entries are represented by black bullets.

Specializing (15) for the integrals in (13) and (14), we obtain

$$(A_c)_{kl} = \sum_{q=0}^{n_q} \mu(\phi_c(\xi_q)) \underbrace{\left(\widehat{\varphi}'_k(\xi_q)\,(J_c(\xi_q))^{-1}\right)}_{(I)} \underbrace{\left(\widehat{\varphi}'_l(\xi_q)\,(J_c(\xi_q))^{-1}\right)}_{(II)} \underbrace{J_c(\xi_q)\,w_q}_{(III)}\,, \tag{16}$$

$$(\mathbf{f}_c)_k = \sum_{q=0}^{n_q} f(\phi_c(\xi_q))\,\widehat{\varphi}_k(\xi_q)\,J_c(\xi_q)\,w_q\,. \tag{17}$$

To conclude, 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 (16) and (17). All the local contributions are added together, and then the resulting linear system (6) is solved to compute the control variables $\mathbf{U}$.

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.

A generic finite element solver (for a linear, stationary problem) is organized along four basic steps (also depicted in Figure 4):

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;

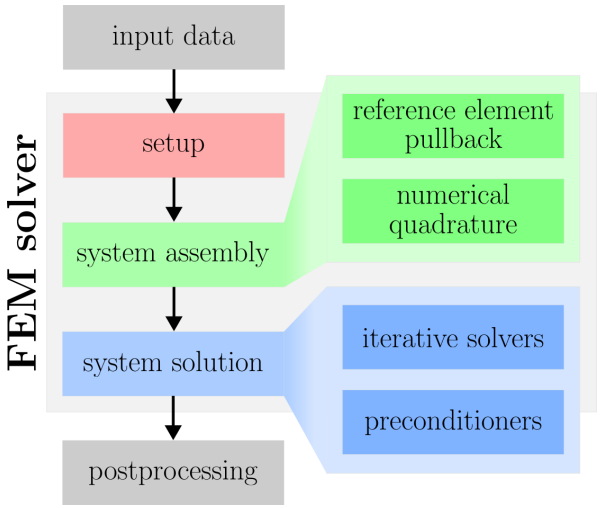

Figure 4: Schematic representation of the main steps in a (time independent) finite element solver.

2. Assembly: the matrix and right-hand side of the linear system (6) are constructed, relying on the decomposition into local contributions represented by (16) and (17);
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.

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 deal with the algebraic 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.

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/current/doxygen/deal.II/namespaceGridGenera
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` class `DoFHandler`, which takes care of handling the global and local indexing of degrees of freedom.

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. Following the construction of the matrix A and vector $\mathbf{f}$, we proceed to assemble them one element at a time: this translates to a loop over all elements in the mesh. For each element, we construct the local matrix and vector using the definitions (16) and (17). 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 (13). `deal.II` offers a 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 $\widehat{\mathbf{u}} = \left(U_0, U_1, U_2, \ldots, U_{N_h}, U_{N_h+1}\right)^T \in \mathbb{R}^{N_h+2}$. 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

$$U_0 = 0 \qquad U_{N_h+1} = 0\,. \tag{18}$$

Therefore, we can impose them by discarding the first and last equation from $\widehat{A}$ and replacing them with (18). `deal.II` offers several ways to achieve this. In the simplest approach, we store in an `std::map` the relations (18) (in this case, the map would contain the two pairs (0, 0) and $(N_h + 1,\ 0)$). Finally, the `deal.II` function

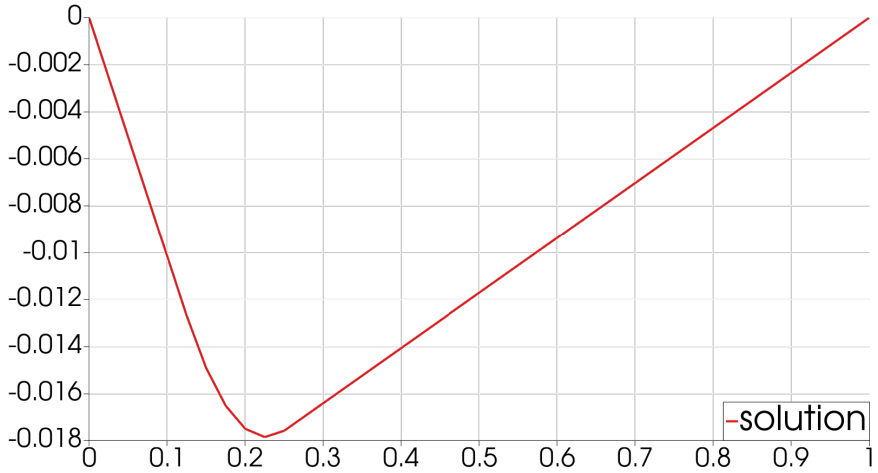

Figure 5: Approximate solution to problem (1), obtained with finite elements of degree 1 and using 40 elements. This plot is obtained by opening the solution in Paraview and applying the "Plot over line" filter.

`MatrixTools::apply_boundary_values` modifies the linear system to account for conditions (18).

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` 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/). While Paraview is not very effective for 1D data (such as in this example), it 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.