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

Laboratory 07

Finite Element method for non linear equations and vectorial problems

Exercise 1.

Let $\Omega = (0,1)^3$ be the unit cube and let us consider the following non linear problem:

$$\begin{cases} -\boldsymbol{\nabla} \cdot \left(\left(\mu_0 + \mu_1 u^2 \right) \boldsymbol{\nabla} u \right) = f & \text{in } \Omega, \quad (1a) \\ u = 0 & \text{on } \partial\Omega, \quad (1b) \end{cases}$$

where $\mathbf{x} = (x, y, z)^T$, $\mu_0 = 1$, $\mu_1 = 10$ and $f(\mathbf{x}) = 1$.

1.1. Write the weak formulation of problem (1), expressing it in the residual form $R(u, v) = 0$.

Solution. Let $V = H_0^1(\Omega)$ and $v \in V$. Following the usual procedure (multiply v to (1a), then integrate by parts), we obtain

$$\underbrace{\int_\Omega (\mu_0 + \mu_1 u^2) \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v \mathrm{d}\mathbf{x}}_{b(u,v)} = \underbrace{\int_\Omega f v \mathrm{d}\mathbf{x}}_{F(v)} .$$

By defining the *residual*

$$R(u, v) = b(u, v) - F(v) ,$$

we can write the weak formulation as

$$\text{find } u \in V \text{ such that } R(u, v) = 0 \text{ for all } v \in V .$$

Notice that $b(u, v)$, and thus $R(u, v)$, are non linear with respect to u.

1.2. Compute the Fréchet derivative $a(u)(\delta, v)$ of the residual $R(u)(v)$, then write Newton's method for the solution of problem (1).

Solution. The Fréchet derivative is a generalization of the concept of derivative that also works for functionals and bilinear forms. For a given functional $G : V \to \mathbb{R}$, the Frćhet derivative at a point $u \in V$ is a linear operator $A(u) = \dfrac{\mathrm{d}G}{\mathrm{d}u} : V \to \mathbb{R}$ such that

$$\lim_{\|\delta\| \to 0} \frac{|G(u+\delta) - G(u) - A(u)(\delta)|}{\|\delta\|} = 0 \, .$$

In other words, A is the linear operator that best approximates the functional G near the point u.

Considering the residual $R(u, v)$, seen as a function of only u, we have

$$\begin{aligned} R(u+\delta, v) &= b(u+\delta, v) - F(v) \\ &= \int_\Omega (\mu_0 + \mu_1 (u+\delta)^2) \boldsymbol{\nabla}(u+\delta) \cdot \boldsymbol{\nabla} v \mathrm{d}\mathbf{x} - \int_\Omega f v \mathrm{d}\mathbf{x} \\ &= \int_\Omega \Big[(\mu_0 + \mu_1 u^2) \boldsymbol{\nabla} u + (\mu_1 \delta^2) \boldsymbol{\nabla} u + (2\mu_1 u \delta) \boldsymbol{\nabla} u + \\ &\qquad (\mu_0 + \mu_1 u^2) \boldsymbol{\nabla} \delta + (\mu_1 \delta^2) \boldsymbol{\nabla} \delta + (2\mu_1 u \delta) \boldsymbol{\nabla} \delta \Big] \cdot \boldsymbol{\nabla} v \mathrm{d}\mathbf{x} - \int_\Omega f v \mathrm{d}\mathbf{x} \, . \end{aligned}$$

This leads to

$$R(u+\delta, v) - R(u, v) = \underbrace{\int_\Omega (2\mu_1 u \delta) \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v \mathrm{d}\mathbf{x} + \int_\Omega (\mu_0 + \mu_1 u^2) \boldsymbol{\nabla} \delta \cdot \boldsymbol{\nabla} v \mathrm{d}\mathbf{x}}_{\frac{\mathrm{d}R}{\mathrm{d}u}(\delta, v) = a(u)(\delta, v)} + o(||\delta||^2) \, .$$

Notice that the bilinear form a is linear with respect to both δ and v. The Newton's method for this problem reads: given an initial guess $u^{(0)}$, iterate for $k = 0, 1, 2, \ldots$ and until convergence:

1. compute $\delta^{(k)}$ by solving the linear problem: $a(u^{(k)})(\delta^{(k)}, v) = -R(u^{(k)}, v)$ for all $v \in V$;
2. set $u^{(k+1)} = u^{(k)} + \delta^{(k)}$.

The problem at step 1 is a linear differential problem in weak form, and thus we can solve it using finite elements.

Upon finite element discretization, the bilinear form $a(u)(\delta, v)$ gives rise to the following matrix:

$$(A(u))_{ij} = a(u)(\varphi_j, \varphi_i) = \int_\Omega (2\mu_1 u \varphi_j) \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} \varphi_i \mathrm{d}\mathbf{x} \int_\Omega (\mu_0 + \mu_1 u^2) \boldsymbol{\nabla} \varphi_j \cdot \boldsymbol{\nabla} \varphi_i \mathrm{d}\mathbf{x} \, ,$$

whereas the residual yields the vector

$$(\mathbf{r}(u))_i = R(u, \varphi_i) = \int_\Omega (\mu_0 + \mu_1 u^2) \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} \varphi_i \mathrm{d}\mathbf{x} - \int_\Omega f \varphi_i \mathrm{d}\mathbf{x} \ .$$

We can stop the iterations when either $\|\delta^{(k)}\|$ or $\|R(u^{(k)})\|$ become smaller than a given tolerance (stopping based on the increment between iterations or on the residual, respectively).

1.3. Using Newton's method, implement a solver for problem (1). Then, solve the problem on the mesh `mesh/mesh-cube-20.msh`, with polynomial degree $r = 1$, and using a tolerance of 10^{-6} on the norm of the residual for the Newton's method.

Solution. See file `src/lab-07-exercise1.cpp` for the implementation. The solution is reported in Figure 1a.

When assembling the system matrix and the residual for step 1 of Newton's method, we need to evaluate the solution $u^{(k)}$ on quadrature nodes. This can be done by exploiting the FE representation, i.e. for a given quadrature node $\mathbf{x}_q$ on element K,

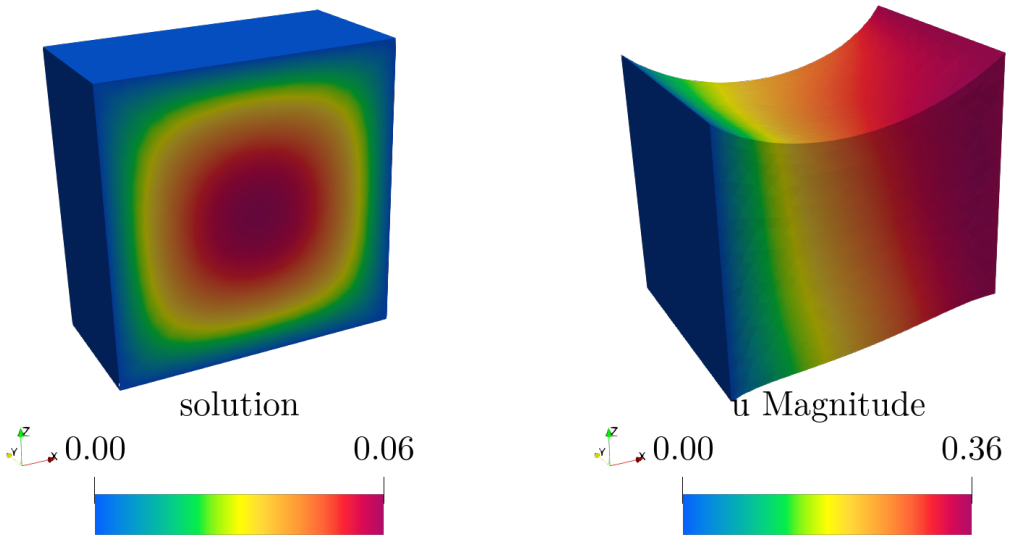
$$u^{(k)}(\mathbf{x}_q) = \sum_{i \in I_K} U_i^{(k)} \varphi_i(\mathbf{x}_q) \ ,$$

where I_K is the set of basis function indices that have support over the element K. We could do the above computation manually, but since it is so common `deal.II` offers an easy way to do that through `FEValues::get_function_values`. Similarly, we need to evaluate $\boldsymbol{\nabla} u^{(k)}$, and we can do that through `FEValues::get_function_gradients`.

Boundary conditions must be treated with care when solving a non-linear problem through Newton's method. Neumann boundary conditions are included in the weak formulation of the problem, and thus are inherited by the linearized problem in step 1 of Newton's method. Dirichlet boundary conditions must be imposed essentially. We want that, upon convergence, $u = g$ on some boundary Γ_D. One way to achieve that is imposing that $u^{(k)} = g$ on Γ_D for all iterations k. This is equivalent to setting $u^{(0)} = g$ on Γ_D and $\delta^{(k)} = 0$ on Γ_D for all iterations k. In practical terms, this means that:

1. we impose the (possibly non-homogeneous) Dirichlet boundary conditions of the original problem on the initial guess $u^{(0)}$ of Newton's method;
2. we impose homogeneous Dirichlet boundary conditions on all the increments $\delta^{(k)}$.

Newton's method is not guaranteed to converge. Therefore, if we just check that the residual (or the increment between iterations) falls below a given tolerance, we might end up with an infinite loop, which is obviously not good. To avoid this situation, we choose a maximum number of iterations (say 1000), and if that number is reached we assume that Newton is not converging and stop the execution. This is similar to what is done in most iterative algorithms (e.g. iterative linear solvers).



(a) (b)

Figure 1: (a) Solution to exercise 1. The domain was clipped along the plane $y = 0.5$. (b) Solution to exercise 2. The domain was warped by the solution $\mathbf{u}$ (using the filter "Warp by vector").

Exercise 2.

Let $\Omega = (0,1)^3$ be the unit cube and let us consider the following linear elasticity problem: find a displacement field $\mathbf{u} : \Omega \to \mathbb{R}^3$ such that

$$\begin{cases} -\boldsymbol{\nabla} \cdot \sigma(\mathbf{u}) = \mathbf{f} & \text{in } \Omega \,, \\ \mathbf{u} = \mathbf{g} & \text{on } \Gamma_0 \cup \Gamma_1 \,, \\ \sigma(\mathbf{u})\mathbf{n} = \mathbf{0} & \text{on } \Gamma_2 \cup \Gamma_3 \cup \Gamma_4 \cup \Gamma_5 \,, \end{cases} \tag{2}$$

where

$$\begin{aligned} \sigma(\mathbf{u}) &= \mu \boldsymbol{\nabla} \mathbf{u} + \lambda (\boldsymbol{\nabla} \cdot \mathbf{u}) I \,, \\ \Gamma_0 &= \{x = 0, y \in (0,1), z \in (0,1)\} \,, \\ \Gamma_1 &= \{x = 1, y \in (0,1), z \in (0,1)\} \,, \\ \Gamma_2 &= \{x \in (0,1), y = 0, z \in (0,1)\} \,, \\ \Gamma_3 &= \{x \in (0,1), y = 1, z \in (0,1)\} \,, \\ \Gamma_4 &= \{x \in (0,1), y \in (0,1), z = 0\} \,, \\ \Gamma_5 &= \{x \in (0,1), y \in (0,1), z = 1\} \,, \end{aligned}$$

$\mu = 1$, $\lambda = 10$, $\mathbf{g}(\mathbf{x}) = (0.25x, 0.25x, 0)^T$ and $\mathbf{f}(\mathbf{x}) = (0, 0, -1)^T$.

2.1. Write the weak formulation of problem (2).

Solution. Since the unknown of the problem is vector-valued, we work in spaces of vector-valued functions (denoted in bold). Other than that, the procedure for deriving a weak formulation is essentially unchanged, provided we use the appropriate notions of product (e.g. scalar product between vectors or tensors) and spatial derivatives.

Let us introduce the function spaces

$$V_0 = \left\{ \mathbf{v} \in \left[H^1(\Omega)\right]^3 \ : \ \mathbf{v} = \mathbf{0} \text{ on } \Gamma_0 \cup \Gamma_1 \right\} ,$$
$$V = \left\{ \mathbf{v} \in \left[H^1(\Omega)\right]^3 \ : \ \mathbf{v} = \mathbf{g} \text{ on } \Gamma_0 \cup \Gamma_1 \right\} .$$

Let $\mathbf{v} \in V_0$, we get:

$$\int_\Omega (\mu \boldsymbol{\nabla} \mathbf{u} \colon \boldsymbol{\nabla} \mathbf{v} + \lambda (\boldsymbol{\nabla} \cdot \mathbf{u}) I \colon \boldsymbol{\nabla} \mathbf{v}) \, \mathrm{d}\mathbf{x} = \int_\Omega \mathbf{f} \cdot \mathbf{v} \mathrm{d}\mathbf{x} .$$

Introducing

$$a(\mathbf{u}, \mathbf{v}) = \int_\Omega (\mu \boldsymbol{\nabla} \mathbf{u} \colon \boldsymbol{\nabla} \mathbf{v} + \lambda (\boldsymbol{\nabla} \cdot \mathbf{u})(\boldsymbol{\nabla} \cdot \mathbf{v})) \, \mathrm{d}\mathbf{x} ,$$
$$F(\mathbf{v}) = \int_\Omega \mathbf{f} \cdot \mathbf{v} \mathrm{d}\mathbf{x} ,$$

the weak formulation reads:

$$\text{find } \mathbf{u} \in V \text{ such that } a(\mathbf{u}, \mathbf{v}) = F(\mathbf{v}) \text{ for all } \mathbf{v} \in V_0 .$$

We can equivalently formulate the problem in a setting where the Lax-Milgram lemma is applicable by introducing a lifting function $\mathbf{R}(g) \in V$ such that $\mathbf{R}(\mathbf{g}) = \mathbf{g}$ on $\Gamma_0 \cup \Gamma_1$. Then, setting $\mathbf{u} = \mathbf{u}_0 + \mathbf{R}(\mathbf{g})$, the problem reads:

$$\text{find } \mathbf{u}_0 \in V_0 \text{ such that } a(\mathbf{u}_0, \mathbf{v}) = F(\mathbf{v}) - a(\mathbf{R}(\mathbf{g}), v) \text{ for all } \mathbf{v} \in V_0 .$$

2.2. Implement in `deal.II` a finite element solver for problem (2).

Solution. See the file `src/lab-07-exercise2.cpp`. The solution is displayed in Figure 1b.

When introducing the discrete weak formulation to (2), it is convenient to work with vector-valued basis functions. In other words, we assume the discrete solution can be decomposed as

$$\mathbf{u}_h(\mathbf{x}) = \sum_{i=1}^{N_h} U_i \boldsymbol{\varphi}_i(\mathbf{x}) ,$$

where $\boldsymbol{\varphi}_i$ are the vector-valued basis functions. There are different ways in which they can be constructed. One possible way, starting from the scalar basis functions φ_j (with $j = 1, 2, \ldots, N_h/3$), is the following:

$$\boldsymbol{\varphi}_1 = \begin{bmatrix} \varphi_1 \\ 0 \\ 0 \end{bmatrix} \quad , \quad \boldsymbol{\varphi}_2 = \begin{bmatrix} 0 \\ \varphi_1 \\ 0 \end{bmatrix} \quad , \quad \boldsymbol{\varphi}_3 = \begin{bmatrix} 0 \\ 0 \\ \varphi_1 \end{bmatrix} \quad , \quad \boldsymbol{\varphi}_4 = \begin{bmatrix} \varphi_2 \\ 0 \\ 0 \end{bmatrix} \quad , \quad \ldots$$

This kind of choice (i.e. each vector basis function has only one non-zero component, and its value for that component is that of a scalar basis function) is referred to as *primitive* in `deal.II`. Of course, the ordering is arbitrary (for instance, we could first have all basis functions associated to the first component, then all those associated to the second, then all those associated to the third).

`deal.II` allows to access vector-valued basis functions in two ways:

1. for a given index i of a vector-valued basis function, we can access the index c of its non-zero component through `FiniteElement::system_to_component_index` and the valueof the associated scalar basis function through `FEValues::shape_value` (and similar for e.g. the gradient or other derivatives);
2. combining `FEValues` and `FEValuesExtractors` to obtain `FEValuesView`, we can access the vector-valued shape function directly.

Generally, the second approach is more convenient to program, because it leads to code that more closely matches the mathematical notation. For this reason, this is the approach followed in the solution. The first approach may be marginally more efficient in some circumstances (especially as it is less prone to causing the construction of temporary `Tensors`), but this should be verified through a profiler.