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

# Laboratory 02
Finite Element method for the Poisson equation in 1D: code verification and convergence analysis

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 [3].

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 laboratory, by checking that the numerical solution satisfies known constraints (such as boundary conditions), and that it in agreement 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 1.

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

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

$\mu(x) = 1$ and $f(x) \in L^2(\Omega)$.

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

**1.2.** Starting from the solution of the first laboratory, implement a method `double Poisson1D::compute_error(const VectorTools::NormType &norm_type) const` 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:

$$e_{L^2} = \|u_h - u_{\text{ex}}\|_{L^2} = \sqrt{\int_0^1 |u_h - u_{\text{ex}}|^2 \, dx} \,,$$

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

**1.3.** With polynomial degree $r = 1$, solve the problem (1) with finite elements, setting $N + 1 = 10, 20, 40, 80, 160$. 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.

**1.4.** Repeat the previous point setting $r = 2$.

**1.5.** 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

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

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

## Possibilities for extension

**Patch test** Repeat Exercise 1 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 in the extensions to previous laboratory, 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 1 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] 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.