DUNE PDELab Tutorial 04
Finite Elements for the Wave Equation

DUNE/PDELab Team

August 31, 2021

Contents

1 Introduction

In this tutorial we solve the wave equation formulated as a first order in time system. This way the example serves as a model for the treatment of systems of partial differential equations in PDELab.

Depends On

This tutorial depends on tutorial 01 and 03.

2 PDE Problem

As an example for a system we consider the wave equation with reflective boundary conditions:

$$\partial_{tt} u - c^2 \Delta u = 0 \qquad \text{in } \Omega \times \Sigma, \tag{1a}$$
$$u = 0 \qquad \text{on } \partial\Omega, \tag{1b}$$
$$u = q \qquad \text{at } t = 0, \tag{1c}$$
$$\partial_t u = w \qquad \text{at } t = 0, \tag{1d}$$

where c is the speed of sound. Renaming $u_0 = u$ and introducing $u_1 = \partial_t u_0 = \partial_t u$ we can write the wave equation as a system of two equations:

$$\partial_t u_1 - c^2 \Delta u_0 = 0 \qquad \text{in } \Omega \times \Sigma, \tag{2a}$$
$$\partial_t u_0 - u_1 = 0 \qquad \text{in } \Omega \times \Sigma, \tag{2b}$$
$$u_0 = 0 \qquad \text{on } \partial\Omega, \tag{2c}$$
$$u_1 = 0 \qquad \text{on } \partial\Omega, \tag{2d}$$
$$u_0 = q \qquad \text{at } t = 0, \tag{2e}$$
$$u_1 = w \qquad \text{at } t = 0. \tag{2f}$$

Since $u_0 = u = 0$ on the boundary we also have $\partial_t u = u_1 = 0$ on the boundary. But one may also omit the boundary condition on u_1.

Note that there are several alternative ways how to write the scalar equation (1) as a system of PDEs:

- Eriksson et al. in [1] apply the Laplacian to equation (2b)

  $$\Delta \partial_t u_0 - \Delta u_1 = 0 \tag{3}$$

  which has advantages for energy conservation but requires additional smoothness properties.

- Alternatively, we may introduce the abbreviations $q = \partial_t u$ and $w = -\nabla u$, so $\partial_{tt} u - c^2 \Delta u = \partial_{tt} u - c^2 \nabla \cdot \nabla u = \partial_t q + c^2 \nabla \cdot w = 0$. Taking partial derivatives of the introduced variables we obtain $\partial_{x_i} q = \partial_{x_i} \partial_t u = \partial_t \partial_{x_i} u = -\partial_t w_i$. This results in a first-order hyperbolic system of PDEs for q and w

  $$\partial_t q + c^2 \nabla \cdot w = 0$$
  $$\partial_t w + \nabla q = 0$$

which are called equations of linear acoustics [2]. This formulation is physically more relevant. It can be modified to handle discontinuous material properties and upwind finite volume methods can be used for numerical treatment.

Here we will stay, however, with the simplest formulation (2) for simplicity.

Weak Formulation

Multiplying (2a) with the test function v_0 and (2b) with the test function v_1 and using integration by parts we arrive at the weak formulation: Find $(u_0(t), u_1(t)) \in U_0 \times U_1$ s.t.

$$\begin{aligned} d_t(u_1, v_0)_{0,\Omega} + c^2(\nabla u_0, \nabla v_0)_{0,\Omega} = 0 \quad \forall v_0 \in U_0 \\ d_t(u_0, v_1)_{0,\Omega} - (u_1, v_1)_{0,\Omega} = 0 \quad \forall v_1 \in U_1 \end{aligned} \tag{4}$$

where we used the notation of the L^2 inner product $(u, v)_{0,\Omega} = \int_\Omega uv\, dx$. An equivalent formulation to (4) that hides the system structure reads as follows:

$$\begin{aligned} d_t \left[(u_0, v_1)_{0,\Omega} + (u_1, v_0)_{0,\Omega}\right] \\ + \left[c^2(\nabla u_0, \nabla v_0)_{0,\Omega} - (u_1, v_1)_{0,\Omega}\right] = 0 \quad \forall (v_0, v_1) \in U_0 \times U_1 \end{aligned} \tag{5}$$

With the latter we readily identify the temporal and spatial residual forms:

$$m^{\text{WAVE}}((u_0, u_1), (v_0, v_1)) = (u_0, v_1)_{0,\Omega} + (u_1, v_0)_{0,\Omega}, \tag{6}$$

$$r^{\text{WAVE}}((u_0, u_1), (v_0, v_1)) = c^2(\nabla u_0, \nabla v_0)_{0,\Omega} - (u_1, v_1)_{0,\Omega}\ , \tag{7}$$

while with the former the system structure is more visible which might help to understand the implementation presented in section 4.3. The spaces U_0 and U_1 can differ as different types of boundary conditions can be incorporated into the ansatz spaces. But here both spaces are constrained by homogeneous Dirichlet boundary conditions.

Generalization

The abstract setting of PDELab with its weighted residual formulation carries over to the case of systems of partial differential equations when cartesian products of functions spaces are introduced, i.e. the abstract *stationary* problem then reads

$$\text{Find } u_h \in U_h = U_h^1 \times \ldots \times U_h^s \text{ s.t.:} \quad r_h(u_h, v) = 0 \quad \forall v \in V_h = V_h^1 \times \ldots \times V_h^s \tag{8}$$

with s the number of components in the system. Again the concepts are completely orthogonal meaning that r_h might be affine linear or nonlinear in its first argument and the instationary case works as well.

From an organizational point of view it makes sense to allow that a component space U_h^i in the cartesian product is itself a product space. This naturally leads to a *tree structure* in the function spaces.

Consider as an example the Stokes equation in d space dimensions. There one has pressure p and velocity v with components $v_1, \ldots, v_d$ as unknowns. An appropriate function space then would be

$$U = (P, (V^1, \ldots, V^d)).$$

3 Finite Element Method

The finite element method applied to (5) is straightforward. We may use the conforming space $V_h^{k,d}(\mathcal{T}_h)$ of degree k in dimension d for each of the components. Typically one would choose the same polynomial degree for both components.

4 Realization in PDELab

The structure of the code is very similar to that of tutorial 01 and 03. It consists of the following files:

1) The ini-file `tutorial04.ini` holds parameters read by various parts of the code which control the execution.

2) The main file `tutorial04.cc` includes the necessary C++, DUNE and PDELab header files and contains the `main` function where the execution starts. The purpose of the `main` function is to instantiate DUNE grid objects and call the `driver` function.

3) File `driver.hh` instantiates the necessary PDELab classes for solving a linear instationary problem and finally solves the problem.

4) File `wavefem.hh` contains the local operator classes `WaveFEM` and `WaveL2` realizing the spatial and temporal residual forms.

4.1 Ini-File

The ini-file contains the usual sections for structured and 1d grids. The `fem` section is the same as in tutorial 01 and allows to set the polynomial degree, temporal integration order and the time step size. The `problem` section has a new parameter for the speed of sound.

```
[grid]
dim=2
refinement=4

[grid.structured]
LX=2.5
LY=1.0
LZ=1.0
NX=5
NY=2
NZ=2

[grid.oned]
a=0.0
b=2.5
elements=5

[fem]
```

```
degree=2
torder=2
dt=0.025

[problem]
speedofsound=1.0
T=4.0

[output]
filename=wave2d
subsampling=2
```

4.2 Function `main`

The `main` function is very similar to the one in tutorial 03. In order to simplify things only the structured grids `OneDGrid` and `YaspGrid` are used.

4.3 Function `driver`

The `driver` function gets a grid view, a finite element map and a parameter tree and its purpose is to solve the problem on the given mesh.

```
template<typename GV, typename FEM>
void driver (const GV& gv, const FEM& fem,
             Dune::ParameterTree& ptree)
```

There are several changes now in the driver due to the system of PDEs. The first step is to set up the grid function space using the given finite element map:

```
  using CON = Dune::PDELab::ConformingDirichletConstraints;
  using VBE0 = Dune::PDELab::ISTL::VectorBackend<>;
  using GFS0 = Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE0>;
  GFS0 gfs0(gv,fem);
```

The next step is to set up the product space containing two components. This is done by the following code section:

```
  using VBE =
    Dune::PDELab::ISTL::VectorBackend<
      Dune::PDELab::ISTL::Blocking::fixed
    >;
  using OrderingTag = Dune::PDELab::EntityBlockedOrderingTag;
  using GFS =
    Dune::PDELab::PowerGridFunctionSpace<GFS0,2,VBE,OrderingTag>;
  GFS gfs(gfs0);
```

PDELab offers two different class templates to build product spaces. The one used here is `PowerGridFunctionSpace` which creates a product of a compile-time given number (2 here) of *identical* function spaces (`GFS0` here) which may only differ in the constraints. With the class template `CompositeGridFunctionSpace` you can create a product space where all components might be different spaces.

We also have to set up names for the child spaces to facilitate VTK output later on:

```
using namespace Dune::Indices;
gfs.child(_0).name("u0");
gfs.child(_1).name("u1");
```

An important aspect of product spaces is the ordering of the corresponding degrees of freedom. Often the solvers need to exploit an underlying block structure of the matrices.

This works in two stages: An ordering has first to be specified when creating product spaces which is then subsequently exploited in the backend. Here we use the `EntityBlockedOrderingTag` to specify that all degrees of freedom related to a geometric entity should be numbered consecutively in the coefficient vector. Other options are the `LexicographicOrderingTag` ordering first all degrees of freedom of the first component space, then all of the second component space and so on. With the Iterative Solver Template Library ISTL it is now possible to exploit the block structure at compile-time. Here we use the tag `fixed` in the ISTL vector backend to indicate that at this level we want to create blocks of fixed size (in this case the block size will be two – corresponding to the degrees of freedom per entity). Another option would be the tag `none` which is the default. Then the degrees of freedom are still ordered in the specified way but no block structure is introduced on the ISTL level. *Important notice:* Using fixed block structure in ISTL requires that there is the same number of degrees of freedom per entity. This is true for polynomial degrees one and two but *not* for higher polynomial degree!

In order to define a function that specifies the initial value we can use the same techniques as in the scalar case. We first define a lambda closure

```
// define the initial condition
auto ulambda = [dim](const auto& x){
  Dune::FieldVector<RF,2> rv(0.0);
  for (int i=0; i<dim; i++) rv[0]+=(x[i]-0.375)*(x[i]-0.375);
  rv[0] = std::max(0.0,1.0-8.0*sqrt(rv[0]));
  return rv;
};
```

now returning two components in a `FieldVector`. The first component is the initial value for u and the second component is the initial value for $\partial_t u$. Then a PDELab grid function can be constructed from the lambda closure

```
auto u = Dune::PDELab::makeGridFunctionFromCallable(gv,ulambda);
```

Using the grid function a coefficient vector can now be initialized:

```
using Z = Dune::PDELab::Backend::Vector<GFS,RF>;
Z z(gfs);
Dune::PDELab::interpolate(u,gfs,z);
```

The next step is to assemble the constraints container for the composite function space. Unfortunately there is currently no way to define the constraints for both components in one go. We need to set up a separate lambda closure for each component:

```
auto b0lambda = [](const auto& x){return true;};
auto b0 = Dune::PDELab::
  makeBoundaryConditionFromCallable(gv,b0lambda);
auto b1lambda = [](const auto& x){return true;};
auto b1 = Dune::PDELab::
  makeBoundaryConditionFromCallable(gv,b1lambda);
```

and then combine it using:

```
using B = Dune::PDELab::CompositeConstraintsParameters<
  decltype(b0),decltype(b1)
  >;
B b(b0,b1);
```

Note that you could define different constraints for each component space although it is the same underlying function space.

Now the constraints container can be assembled as before:

```
using CC = typename GFS::template ConstraintsContainer<RF>::Type;
CC cc;
Dune::PDELab::constraints(b,gfs,cc);
```

As we do not want to manually extract the subspaces for u_0 and u_1 from the overall space to add to them to the VTK writer, we call a PDELab helper function that handles this automatically:

```
Dune::PDELab::addSolutionToVTKWriter(vtkSequenceWriter,gfs,z);
```

Note that in order to use this function, we have to set the names of the subspaces, as we did earlier in the tutorial.

The rest of the driver is the same as for tutorial 03 except that a linear solver is used instead of Newton's method.

4.4 Spatial Local Operator

The spatial residual form (7) is implemented by the local operator `WaveFEM` in file `wavefem.hh`. Cache construction and flags settings are the same as in tutorial 01 and 03. Only volume terms are used here. Note also that no parameter object is necessary as the only parameter is the speed of sound c.

`alpha_volume` method

The method `alpha_volume` has the *same* interface as in the scalar case:

```
template<typename EG, typename LFSU, typename X,
         typename LFSV, typename R>
void alpha_volume (const EG& eg, const LFSU& lfsu, const X& x,
                   const LFSV& lfsv, R& r) const
```

However the trial and test function spaces `LFSU` and `LFSV` now reflect the component structure of the global function space, i.e. they consist of two components.

Important notice: Here we assume that trial and test space are identical (up to constraints) and that also both components are identical!

The two components can be extracted with the following code

```
using namespace Dune::Indices;
auto lfsu0 = lfsu.child(_0);
auto lfsu1 = lfsu.child(_1);
```

The function spaces `lfsu0` and `lfsu1` are now scalar spaces (which we assume to be identical).

After extracting the dimension

```
const int dim = EG::Entity::dimension;
```

we select a quadrature rule

```
auto geo = eg.geometry();
int order = 2*lfsu0.finiteElement().localBasis().order();
auto rule = Dune::PDELab::quadratureRule(geo,order);
```

and may now loop over the quadrature points.

For each quadrature point, evaluate the basis function of the first component:

```
      auto& phihat = cache.evaluateFunction(ip.position(),
                          lfsu0.finiteElement().localBasis());
```

As the components are identical we need only evaluate the basis once and can compute the value of u_1 at the quadrature point

```
      RF u1=0.0;
      for (size_t i=0; i<lfsu1.size(); i++)
        u1 += x(lfsu1,i)*phihat[i];
```

Then we evaluate the gradients of the basis functions

```
      auto& gradphihat = cache.evaluateJacobian(ip.position(),
                          lfsu0.finiteElement().localBasis());
```

transform them from the reference element to the real element

```
      const auto S =
        geo.jacobianInverseTransposed(ip.position());
      auto gradphi = makeJacobianContainer(lfsu0);
      for (std::size_t i=0; i<lfsu0.size(); i++)
        S.mv(gradphihat[i][0],gradphi[i][0]);
```

and compute the gradient of u_0:

```
      Dune::FieldVector<RF,dim> gradu0(0.0);
      for (std::size_t i=0; i<lfsu0.size(); i++)
        gradu0.axpy(x(lfsu0,i),gradphi[i][0]);
```

With the integration factor

```
      RF factor = ip.weight()*
        geo.integrationElement(ip.position());
```

the residuals can now be accumulated:

```
        for (std::size_t i=0; i<lfsu0.size(); i++) {
          r.accumulate(lfsu0,i,c*c*(gradu0*gradphi[i][0])*factor);
          r.accumulate(lfsu1,i,-u1*phihat[i]*factor);
        }
```

`jacobian_volume` method

As the problem is linear it is advisable to also implement the `jacobian_volume` method for efficiency and accuracy.

The interface is the same as in the scalar case:

```
  template<typename EG, typename LFSU, typename X,
           typename LFSV, typename M>
  void jacobian_volume (const EG& eg, const LFSU& lfsu, const X& x,
                        const LFSV& lfsv, M& mat) const
```

Component selection, quadrature rule selection and basis evaluation are the same as in `alpha_volume`. We only consider the accumulation of the Jacobian entries here:

```
        RF factor = ip.weight()*
          geo.integrationElement(ip.position());
        for (std::size_t j=0; j<lfsu0.size(); j++)
          for (std::size_t i=0; i<lfsu0.size(); i++) {
            mat.accumulate(lfsu0,i,lfsu0,j,
                     c*c*(gradphi[j][0]*gradphi[i][0])*factor);
            mat.accumulate(lfsu1,i,lfsu1,j,
                    -phihat[j]*phihat[i]*factor);
          }
```

Note how the diagonal sub-blocks of the Jacobian with respect to the first and second component are accessed.

Finally, `WaveFEM` also implements the matrix-free versions for Jacobian application.

4.5 Temporal Local Operator

The temporal residual form (6) is implemented by the local operator `WaveL2` in file `wavefem.hh`. Cache construction and flags settings are the same as in tutorial 01 and 03. Only volume terms are used here.

`alpha_volume` method

The `alpha_volume` method is pretty similar to the one in the spatial operator, except that the value of u_0 is needed instead of the gradient.

Here we just show the residual accumulation:

```
        RF factor=ip.weight()*geo.integrationElement(ip.position());
        for (std::size_t i=0; i<lfsu0.size(); i++) {
          r.accumulate(lfsu0,i,u1*phihat[i]*factor);
```

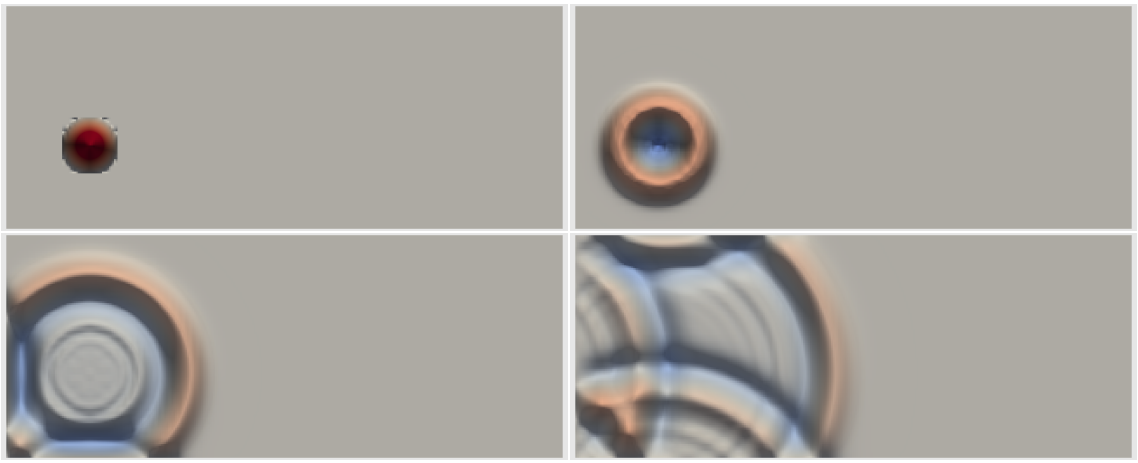

Figure 1: Solution of the wave equation at four different times. Time is proceeding left to right, top to bottom.

```
        r.accumulate(lfsu1,i,u0*phihat[i]*factor);
      }
```

Note that u_1 is integrated with respect to test function v_0 and vice versa.

`jacobian_volume` method

The corresponding Jacobian entries are accumulated in the `jacobian_volume` method:

```
      // accumulate matrix entries
      RF factor=ip.weight()*geo.integrationElement(ip.position());
      for (std::size_t j=0; j<lfsu0.size(); j++)
        for (std::size_t i=0; i<lfsu0.size(); i++) {
          mat.accumulate(lfsu0,i,lfsu1,j,
                         phihat[j]*phihat[i]*factor);
          mat.accumulate(lfsu1,i,lfsu0,j,
                         phihat[j]*phihat[i]*factor);
```

That's it! 293 lines of code to implement the finite element method for the wave equation.

4.6 Running the Example

The example solves the wave equation in the domain $\Omega = (0, 2.5) \times (0, 1)^{d-1}$ with a bump-like initial condition. The results for the two-dimensional case are illustrated in Figure 1. The initial bump has a height of 1 while the color code corresponds to blue for -0.5 and red for 0.5.

5 Outlook

The following ideas could be explored from this tutorial:

- Explore polynomial degree greater than 2 by changing the blocking to `none`.
- Compute the total energy $E(t) = \|\partial_t u\|_{0,\Omega}^2 + \|\nabla u\|_{0,\Omega}^2$ and check its conservation in the numerical scheme.
- Try various time integrators, in particular the Crank-Nicolson method.
- Implement the elliptic projection method of [1] from equation (2).

References

[1] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996. `http://www.csc.kth.se/~jjan/transfer/cde.pdf`.

[2] R. J. Leveque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.