Journal of Computational and Applied Mathematics 270 (2014) 152–165

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Hermes2D, a C++ library for rapid development of adaptive *hp*-FEM and *hp*-DG solvers

Pavel Solin[a,b], Lukas Korous[b,*], Pavel Kus[b,c]

[a] *University of Nevada, Reno, USA*
[b] *RICE, University of West Bohemia, Pilsen, Czech Republic*
[c] *KTE, University of West Bohemia, Pilsen, Czech Republic*

ARTICLE INFO

Article history:
Received 27 November 2013
Received in revised form 20 January 2014

Keywords:
Finite element method
Discontinuous Galerkin method
Automatic adaptivity
Multiphysics coupled problems
Time-dependent problems
Software

ABSTRACT

In this paper we describe Hermes2D, an open-source C++ library for the development and implementation of adaptive higher-order finite element and DG solvers for partial differential equations (PDE) and multiphysics coupled PDE problems. The library is suitable for applications ranging from simple linear PDE solvers to time-dependent solvers for nonlinear multiphysics coupled problems with dynamically changing meshes. We cover several typical application scenarios, and give a brief overview of methods and algorithms that the library provides. Numerical examples are provided.

1. Introduction

The number of adaptive finite element (FEM) codes is growing very fast. They differ in deployment operating systems and hardware platforms, ways of loading the physical model, error estimation mechanisms, algebraic solvers they use, mesh formats, boundary conditions handling, input/output formats, and other aspects. Some of them are designed for a narrow class of problems while others are supposed to cover various types of physical applications. This paper presents Hermes2D, an open source C++ library for the rapid development of adaptive *hp*-FEM and *hp*-DG solvers. The library has been used to solve problems in many different areas including civil, electrical, mechanical and nuclear engineering, and hydrology [1–8]. It has also been used as underlying library for development of Agros2D, a universal engineering application for solution of different physical fields, see [9] for details. Based on the large scope of the Hermes project, we do not attempt to provide a comprehensive description of all its features. Instead, the goal of this paper is to help the reader get started using the library. Note that as the library evolves, some minor changes of the methods and classes occur. The code in the paper comes from the currently newest version 3.0 of the library.

2. Defining weak forms

Consider a model problem of the form

$$-\operatorname{div}(\lambda \nabla u) = C, \tag{1}$$

[*] Corresponding author. Tel.: +420 732262595.
E-mail addresses: solin@unr.edu (P. Solin), korous@rice.zcu.cz (L. Korous), pkus@rice.zcu.cz (P. Kus).

http://dx.doi.org/10.1016/j.cam.2014.02.007
0377-0427/© 2014 Elsevier B.V. All rights reserved.

where $\lambda > 0$ and C are constants, in a bounded two-dimensional domain Ω. We begin with zero Dirichlet boundary conditions. Moreover, assume that the domain Ω is split into two subdomains Ω_{Al} and Ω_{Cu} with different values λ_{Al} and λ_{Cu}, respectively.

Problems in Hermes are treated as linear or nonlinear. For nonlinear problems we use the Newton method or the fixed point method with Anderson acceleration to solve it. In the following, we will use formulations for the Newton method, since it is more general. Simplification to the linear case is straightforward. Hence in the weak formulation, all terms are on the left-hand side:

$$\int_{\Omega_{Al}} \lambda_{Al} \nabla u \cdot \nabla v \, \mathrm{d}x\mathrm{d}y + \int_{\Omega_{Cu}} \lambda_{Cu} \nabla u \cdot \nabla v \, \mathrm{d}x\mathrm{d}y - \int_{\Omega} C v \, \mathrm{d}x\mathrm{d}y = 0. \tag{2}$$

In short, we can write

$$F(u) = 0 \tag{3}$$

where F is the residual function. The code for the weak formulation follows Eq. (2) closely. One thing to notice is that even though the jacobian and residual forms represent the same integral, we have to distinguish them for pragmatic reasons – they come to different sides of the equations and their arguments differ – whereas for the jacobian forms we pass a basis function and a test function, for the residual forms we pass the previous Newton iteration and a test function. The resulting code looks like this:

```
CustomWeakFormPoisson::CustomWeakFormPoisson(std::string material_marker_Al,
  Hermes1DFunction<double>* lambda_Al, std::string material_marker_Cu,
  Hermes1DFunction<double>* lambda_Cu, Hermes2DFunction<double>*
  src_term) : WeakForm<double>(1)
{
  // Jacobian forms.
  add_matrix_form(new DefaultJacobianDiffusion<double>(0, 0, material_marker_Al,
    lambda_Al));
  add_matrix_form(new DefaultJacobianDiffusion<double>(0, 0, material_marker_Cu,
    lambda_Cu));

  // Residual forms.
  add_vector_form(new DefaultResidualDiffusion<double>(0, material_marker_Al,
    lambda_Al));
  add_vector_form(new DefaultResidualDiffusion<double>(0, material_marker_Cu,
    lambda_Cu));
  add_vector_form(new DefaultVectorFormVol<double>(0, HERMES_ANY, src_term));
};
```

Here, the class `CustomWeakFormPoisson` is inherited from a base class `Weak Form<double>`. The argument `1` in the constructor `WeakForm<double>(1)` represents the number of equations and `<double>` stands for the specialization of the base class template for double-precision real numbers calculation (the base class template could as easily be used with complex numbers, single precision, and so on). Jacobian forms are bilinear forms that constitute the Jacobian matrix of the problem. The `DefaultJacobianDiffusion<double>` represents

$$\int_{\Omega} \lambda \nabla u \cdot \nabla v \, \mathrm{d}x\mathrm{d}y.$$

The meaning of the arguments `(0, 0, material_marker_Al, lambda_Al)` is as follows: `0, 0` is the block coordinate of the bilinear form in the Jacobian matrix. It is always `0, 0` for single-PDE problems. The text string `material_marker_Al` identifies finite elements in the mesh that belong to Ω_{Al}, and `lambda_Al` is the value of λ_{Al}. The same goes for the second contribution that corresponds to the material marker `material_marker_Cu`.

Residual forms are generally nonlinear. The `DefaultResidualDiffusion <double>` form represents

$$\int_{\Omega} \lambda \nabla u \cdot \nabla v \, \mathrm{d}x\mathrm{d}y$$

and `DefaultVectorFormVol<double>` stands for

$$\int_{\Omega} C v \, \mathrm{d}x\mathrm{d}y.$$

Again, the first argument is 0 for single-PDE problems, and the other arguments tie the constants to different subdomains. In the last form, `HERMES_ANY` stands for any material marker (meaning the entire domain Ω) and `src_term` is C.

3. Creating a simple linear PDE solver

Once the weak forms have been defined, the solver can be created in a few standard steps:

Step 1—Load the mesh

The main.cpp file begins with loading the mesh. Mesh can be accepted both in the native Hermes format:

```
MeshSharedPtr mesh(new Mesh);
MeshReaderH2D mloader;
mloader.load("domain.mesh", mesh);
```

or in the XML format:

```
MeshSharedPtr mesh(new Mesh);
MeshReaderH2DXML mloader;
mloader.load("domain.xml", mesh);
```

or in the binary BSON format:

```
MeshSharedPtr mesh(new Mesh);
MeshReaderH2DBSON mloader;
mloader.load("domain.bson", mesh);
```

Step 2—Perform initial mesh refinements

A number of initial refinement operations can be done as explained in example P01/01-mesh in Hermes tutorial (available at `http://hpfem.org/hermes`). In this case we just perform optional uniform mesh refinements:

```
for (int i = 0; i < INIT_REF_NUM; i++)
  mesh->refine_all_elements();
```

Step 3—Initialize weak formulation

Next, an instance of the corresponding weak form class is created:

For constant λ_{Al} and λ_{Cu}, the form is instantiated as follows (note the minus (−) sign in front of `src_term` that is there due to the Newton method formulation):

```
CustomWeakFormPoisson wf("Aluminum", new Hermes1DFunction<double>(lambda_Al),
 "Copper", new Hermes1DFunction<double>(lambda_Cu),
 new Hermes2DFunction<double>(-src_term));
```

Step 4—Set constant Dirichlet conditions

Constant Dirichlet boundary conditions are assigned to the boundary markers Bottom, Inner, Outer, and Left as follows:

```
DefaultEssentialBCConst<double> bc_essential(Hermes::vector<std::string>(
  "Bottom", "Inner", "Outer", "Left"), FIXED_BDY_TEMP);
EssentialBCs<double> bcs(&bc_essential);
```

The `Hermes::vector` is derived from `std::vector` enhanced with a few extra constructors. It is used to avoid using variable-length arrays.

Step 5—Initialize finite element space

As a next step, we initialize a finite element subspace of $H^1(\Omega)$:

```
SpaceSharedPtr<double> space(new H1Space<double>(mesh, &bcs, P_INIT));
```

Here, P_INIT stands for a uniform polynomial degree of elements.

Step 6—Initialize discrete problem

The weak formulation and finite element space(s) together constitute a discrete finite element problem. To define it, one needs to create an instance of the `DiscreteProblem` class:

```
DiscreteProblem<double> dp(&wf, space);
```

Step 7—Initialize Newton solver

The `NewtonSolver` class is initialized using a pointer to `DiscreteProblem`:

```
NewtonSolver<double> newton(&dp);
```

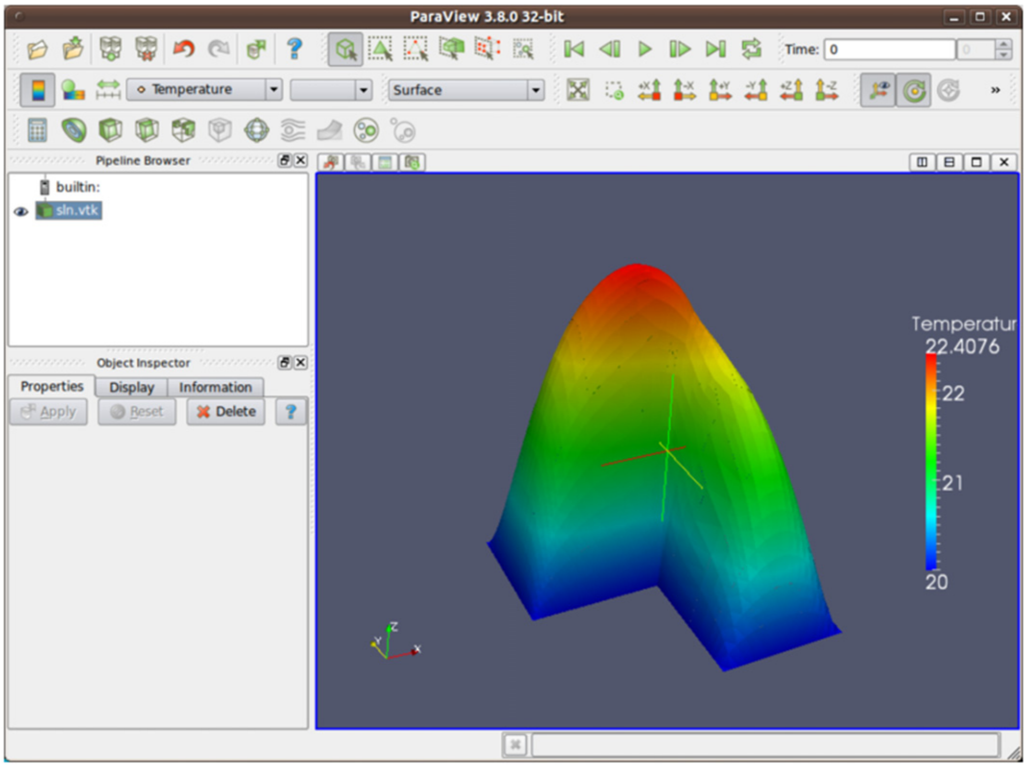

Fig. 1. Sample VTK visualization.

Step 8—Perform the nonlinear iteration

Next, the Newton method is employed in an exception-safe way. For a linear problem, it usually only takes one step, but sometimes it may take more if the matrix is ill-conditioned or if for any other reason the residual after the first step is not under the prescribed tolerance. If all arguments are skipped in `newton.solve()`, this means that the Newton method will start from a zero initial vector, with a default tolerance `1e-8`, and with a default maximum allowed number of `100` iterations:

```
try
{
  newton.solve();
}
catch(Hermes::Exceptions::Exception& e)
{
  e.printMsg();
}
```

The Newton solver is able to include a pre-defined damping factor, or choose the damping factor adaptively. Heuristics deciding when to reuse the Jacobian are also available. Their detailed description, as well as additional useful methods of the `NewtonSolver` class, are described in Hermes Doxygen documentation.

Step 9—Translate the coefficient vector into a Solution

The coefficient vector can be converted into a piecewise-polynomial solution (represented by the class `Solution`) via the function
`Solution<Scalar>::vector_to_solution()`:

```
MeshFunctionSharedPtr<double> sln(new Solution<double>);
Solution<double>::vector_to_solution(newton.get_sln_vector(), space, sln);
```

Step 10—Visualize the solution

The solution can either be displayed using a built-in OpenGL visualization of Hermes, or saved in VTK format and visualized, e.g., using ParaView as shown in Fig. 1:

More details on this solver can be found in example `A-linear/03-Poisson` in the Hermes tutorial repository.

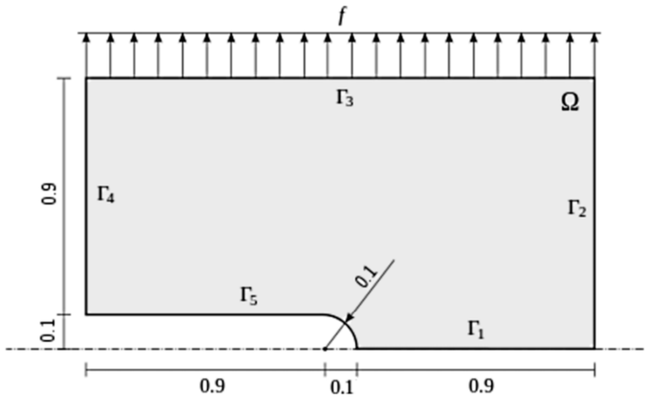

Fig. 2. Computational domain.

4. Extension to the nonlinear case

Once a linear version of a problem works, it is very easy to extend it to a nonlinear case. For example, to replace the constants λ_{Al} and λ_{Cu} above with cubic splines that depend on the solution u, one just needs to do:

```
CubicSpline lambda_Al(...);
CubicSpline lambda_Cu(...);
CustomWeakFormPoisson wf("Aluminum", &lambda_Al, "Copper", &lambda_Cu,
                         new Hermes2DFunction<double>(-src_term));
```

This is possible since `CubicSpline` is a descendant of `Hermes1DFunction`. Analogously, the constant `src_term` can be replaced with an arbitrary function of x and y by subclassing `Hermes2DFunction`:

```
class CustomNonConstSrc<Scalar> : public Hermes2DFunction<Scalar>
```

If cubic splines are not enough, then one can subclass `Hermes1DFunction` to define arbitrary nonlinearities:

```
class CustomLambdaAl<Scalar> : public Hermes1DFunction<Scalar>
class CustomLambdaCu<Scalar> : public Hermes1DFunction<Scalar>
```

5. Solving PDE systems

In this presentation we skip the details of defining non-constant Dirichlet, Neumann, and Newton boundary conditions, and we move on to solving PDE systems (example `08-system` in Hermes tutorial).

First let us understand how Hermes handles systems of n linear PDE whose weak formulation is written as

$$\begin{aligned}
&a_{11}(u_1, v_1) + a_{12}(u_2, v_1) + \cdots + a_{1n}(u_n, v_1) - l_1(v_1) = 0,\\
&a_{21}(u_1, v_2) + a_{22}(u_2, v_2) + \cdots + a_{2n}(u_n, v_2) - l_2(v_2) = 0,\\
&\vdots\\
&a_{n1}(u_1, v_n) + a_{n2}(u_2, v_n) + \cdots + a_{nn}(u_n, v_n) - l_n(v_n) = 0.
\end{aligned}$$

The vector-valued solution $u = (u_1, u_2, \ldots, u_n)$ and test function $v = (v_1, v_2, \ldots, v_n)$ belong to a product space $V = V_1 \times V_2 \times \cdots \times V_n$, where each V_i is one of the available function spaces H^1, $H(\mathrm{curl})$, $H(\mathrm{div})$ or L^2. These spaces can be arbitrarily combined and moreover, they can be discretized on different meshes. The resulting discrete matrix problem has a $n \times n$ block structure.

Let us solve a sample linear elasticity problem. The domain Ω is shown in Fig. 2.

Both displacement components are zero at the bottom edge Γ_1, and a vertical force f acting in the upward direction is applied to the top edge Γ_3. In 2D, this problem is described by a system of two coupled Lamé equations of linear elasticity.

The weak formulation is created using predefined Jacobian and residual forms for linear elasticity. We find it useful to show them here since they illustrate how the block structure of the Jacobian matrix and residual vector are implemented:

```
CustomWeakFormLinearElasticity::CustomWeakFormLinearElasticity(double E,
  double nu, double rho_g, std::string surface_force_bdy, double f0,
  double f1) : WeakForm<double>(2)
{
  double lambda = (E * nu) / ((1 + nu) * (1 - 2*nu));
  double mu = E / (2*(1 + nu));

  // Jacobian.
  add_matrix_form(new DefaultJacobianElasticity_0_0<double>(0, 0,
    lambda, mu));
  add_matrix_form(new DefaultJacobianElasticity_0_1<double>(0, 1,
    lambda, mu));
  add_matrix_form(new DefaultJacobianElasticity_1_1<double>(1, 1,
    lambda, mu));

  // Residual - first equation.
  add_vector_form(new DefaultResidualElasticity_0_0<double>(0,
    HERMES_ANY, lambda, mu));
  add_vector_form(new DefaultResidualElasticity_0_1<double>(0,
    HERMES_ANY, lambda, mu));
  // Surface force (first component).
  add_vector_form_surf(new DefaultVectorFormSurf<double>(0,
    surface_force_bdy, new Hermes2DFunction<double>(-f0)));

  // Residual - second equation.
  add_vector_form(new DefaultResidualElasticity_1_0<double>(1,
    HERMES_ANY, lambda, mu));
  add_vector_form(new DefaultResidualElasticity_1_1<double>(1,
    HERMES_ANY, lambda, mu));
  // Gravity loading in the second vector component.
  add_vector_form(new DefaultVectorFormVol<double>(1, HERMES_ANY,
    new Hermes2DFunction<double>(-rho_g)));
  // Surface force (second component).
  add_vector_form_surf(new DefaultVectorFormSurf<double>(1,
    surface_force_bdy, new Hermes2DFunction<double>(-f1)));
}
```

Here, the CustomWeakFormLinearElasticity is derived from the base class WeakForm<double>. The parameter 2 in the constructor means that we deal with a two-equation system. This also means that the Jacobian matrix has a 2×2 block structure and the residual vector has two components that correspond to the two displacement components u_1 and u_2.

The bilinear weak forms DefaultJacobianElasticity_i_j and the linear vector forms DefaultResidualElasticity_i_j are predefined and they include weak integrals from the Lamé equations. The volumetric vector form DefaultVectorFormVol corresponds to the integral

$$-\int_{\Omega} \varrho g v \,\mathrm{d}x\mathrm{d}y$$

and the surface vector form DefaultVectorFormSurf corresponds to

$$-\int_{\Gamma_3} f v \,\mathrm{d}S$$

where f is the acting force (recall that only its second component is nonzero).

Boundary conditions are implemented as follows:

```
DefaultEssentialBCConst<double> zero_disp("Bottom", 0.0);
EssentialBCs<double> bcs(&zero_disp);
```

Next we define function spaces for the displacement components u_1 and u_2, and (optionally) we calculate the number of degrees of freedom:

```
SpaceSharedPtr<double> u1_space(new H1Space<double>(mesh, &bcs, P_INIT));
SpaceSharedPtr<double> u2_space(new H1Space<double>(mesh, &bcs, P_INIT));
Hermes::vector<SpaceSharedPtr<double> > spaces(u1_space, u2_space);
int ndof = Space<double>::get_num_dofs(spaces);
info("ndof = %d", ndof);
```

The weak formulation is initialized as follows:

```
CustomWeakFormLinearElasticity wf(E, nu, rho*g1, "Top", f0, f1);
```

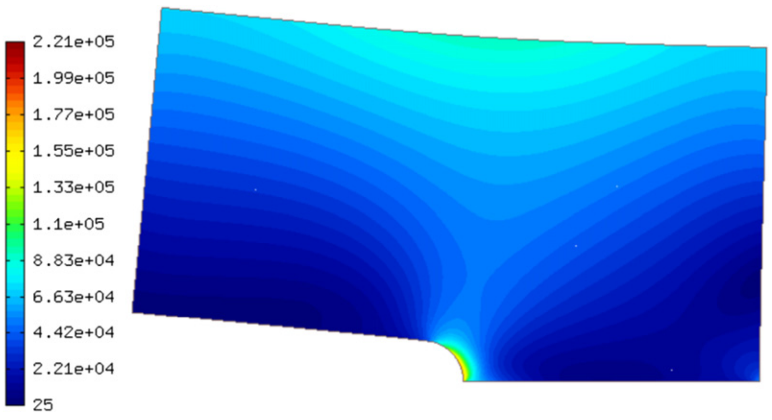

Fig. 3. Domain deformation and von Mises stress distribution. The deformation is exaggerated for the visualization.

Next we create the discrete finite element problem:

```
DiscreteProblem<double> dp(&wf, spaces);
```

The Newton solver is initialized as follows:

```
NewtonSolver<double> newton(&dp);
```

Then the Newton iteration is performed:

```
try
{
  newton.solve();
}
catch(std::exception& e)
{
  e.printMsg();
}
```

Finally, the solution coefficient vector is translated into piecewise-polynomial functions in the two finite element spaces for displacement components:

```
MeshFunctionSharedPtr<double> u1_sln(new Solution<double>),
                              u2_sln(new Solution<double>);
Hermes::vector<MeshFunctionSharedPtr<double> > slns(u1_sln, u2_sln);
Solution<double>::vector_to_solutions(newton.get_sln_vector(), spaces, slns);
```

The result of the computation is shown in Fig. 3.

The complete source code can be found in example `A-linear/08-system` in the Hermes tutorial repository.

6. Axisymmetric 3D problems

Let us solve stationary heat transfer in a hollow cylindrical object shown in Fig. 4:

In the absence of internal heat sources, the equation describing the process has the form

$$-\mathrm{div}(\lambda \nabla T) = 0.$$

The object stands on a hot plate

$$T = T_{bottom} \quad \text{on } \Gamma_{bottom}$$

and on the rest of the boundary we prescribe a natural convection boundary condition

$$-\lambda \frac{\partial T}{\partial n} = \alpha (T - T_{ext}).$$

Here λ is the thermal conductivity of the material, α the heat transfer coefficient between the object and the air, and T_{ext} the exterior air temperature.

Hermes makes the definition of axisymmetric weak forms very easy. The forms are almost identical to the planar 2D case, only there is an extra flag `HERMES_AXISYM_X` or `HERMES_AXISYM_Y` to indicate that this is an axisymmetric problem with respect to the x or y axis, respectively. The weak formulation is defined as follows:

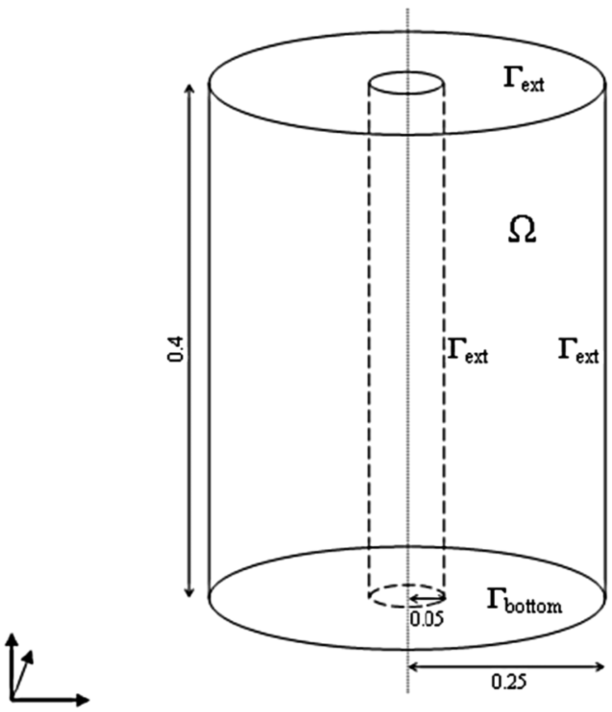

Fig. 4. Axisymmetric computational domain.

```
CustomWeakFormPoissonNewton::CustomWeakFormPoissonNewton(double lambda,
  double alpha, double Text, std::string bdy_heat_flux):WeakForm<double>(1)
{
  // Jacobian form - volumetric.
  add_matrix_form(new WeakFormsH1::DefaultJacobianDiffusion<double>(0, 0,
    HERMES_ANY, new Hermes1DFunction<double>(lambda), HERMES_SYM,
    HERMES_AXISYM_Y));

  // Jacobian form - surface.
  add_matrix_form_surf(new WeakFormsH1::DefaultMatrixFormSurf<double>(0, 0,
    bdy_heat_flux, new Hermes2DFunction<double>(alpha), HERMES_AXISYM_Y));

  // Residual forms - volumetric.
  add_vector_form(new WeakFormsH1::DefaultResidualDiffusion<double>(0,
    HERMES_ANY, new Hermes1DFunction<double>(lambda), HERMES_AXISYM_Y));

  // Residual form - surface.
  add_vector_form_surf(new WeakFormsH1::DefaultResidualSurf<double>(0,
    bdy_heat_flux, new Hermes2DFunction<double>(alpha), HERMES_AXISYM_Y));
  add_vector_form_surf(new WeakFormsH1::DefaultVectorFormSurf<double>(0,
    bdy_heat_flux, new Hermes2DFunction<double>(-alpha * Text),
    HERMES_AXISYM_Y));
};
```

Otherwise the solver is created using the same sequence of steps as a planar 2D problem. Results for the values $T_{bottom} = 100$, $T_{ext} = 0$, $\lambda = 386$ and $\alpha = 20$ are shown in Fig. 5.

Complete source code of this example can be found in example `A-linear/ 09-axisym` in the Hermes tutorial repository.

7. Creating an *hp*-adaptive solver

Let us demonstrate the use of adaptive *hp*-FEM on a linear elliptic problem describing an electrostatic micromotor. This is a MEMS device free of any coils, and thus (as opposed to classical electromotors) resistive to strong electromagnetic waves. Fig. 6 shows a symmetric half of the domain (dimensions need to be scaled with 10^{-5} and they are in meters).

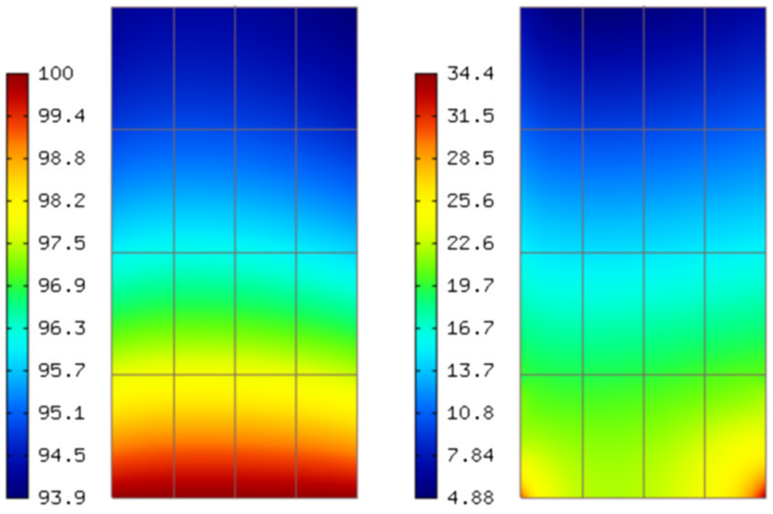

Fig. 5. Temperature (left) and temperature gradient (right). The hot plate is the bottom part of the boundary.

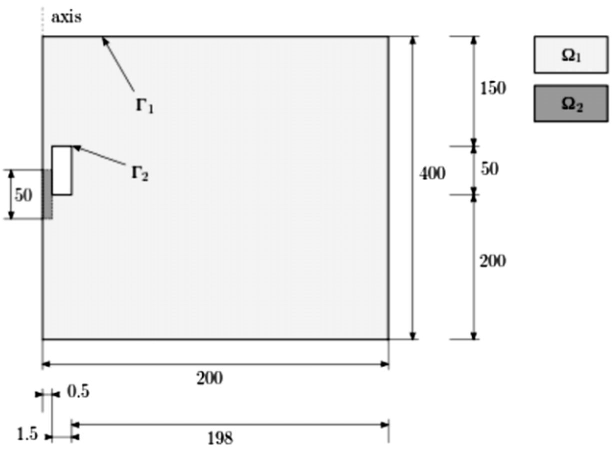

Fig. 6. Computational domain.

The subdomain Ω_2 represents the moving part of the domain and the area bounded by Γ_2 represents the electrodes that are fixed. The distribution of the electrostatic potential φ is governed by the equation

$$-\text{div}(\epsilon_r \epsilon_0 \nabla \varphi) = 0$$

which is equipped with Dirichlet boundary conditions $\varphi = 0$ on Γ_1 and $\varphi = 50$ on Γ_2. The relative permittivity ϵ_r is piecewise-constant, $\epsilon_r = 1$ in Ω_1, $\epsilon_r = 10$ in Ω_2 and ϵ_0 is permittivity of vacuum. As we would like to focus on the adaptivity aspect, for the weak forms we refer to example `D-adaptivity/01-intro` in the Hermes tutorial repository.

At the beginning of the code, we need to define a few parameters, starting with the uniform initial polynomial degree in elements:

```
const int P_INIT = 1;
```

Hermes provides eight modes of automatic adaptivity, ranging from isotropic *p*-refinements to fully anisotropic refinements in both *h* and *p*:

```
// Predefined list of element refinement candidates. Possible values are
// H2D_P_ISO, H2D_P_ANISO, H2D_H_ISO, H2D_H_ANISO, H2D_HP_ISO,
// H2D_HP_ANISO_H, H2D_HP_ANISO_P, H2D_HP_ANISO.
const CandList CAND_LIST = H2D_HP_ANISO_H;
```

Here `H2D_HP_ANISO_H` means *hp*-refinements which can be anisotropic in *h* but isotropic in *p*.

Next we select the way errors for adaptivity will be calculated, here we calculate the error in the H^1 norm and use the error relative to the global H^1 norm:

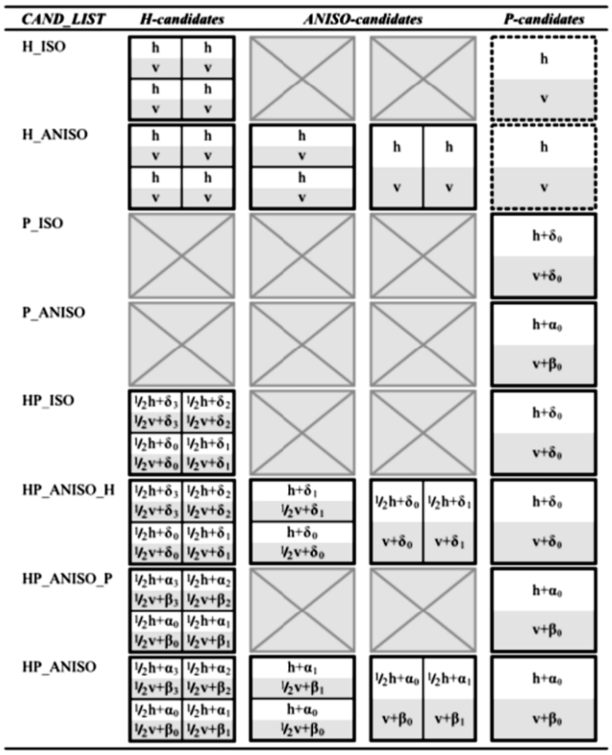

Fig. 7. Overview of refinement candidates.

```
DefaultErrorCalculator<double, HERMES_H1_NORM>
errorCalculator(RelativeErrorToGlobalNorm, 1);
```

We also set a stopping criterion for refinements done at a single level, in this case only the element yielding larger error than half the maximum element error will be refined:

```
const double THRESHOLD = 0.5;
AdaptStoppingCriterionSingleElement<double> stoppingCriterion(THRESHOLD);
```

Next we initialize the main adaptivity class and refinement selector:

```
Adapt<double> adaptivity(space, &errorCalculator, &stoppingCriterion);
H1ProjBasedSelector<double> selector(CAND_LIST);
// Relative error tolerance, here, 1%.
const double ERR_STOP = 0.01;
```

The selector is used by the class `Adapt` to determine how an element should be refined. For that purpose, the selector does the following:

- It generates candidates (proposed element refinements).
- It estimates their local errors by projecting the reference solution onto their FE spaces.
- It calculates the number of degrees of freedom (DOF) contributed by each candidate.
- It calculates a score for each candidate, and sorts them according to their scores.
- It selects a candidate with the highest score. If the next candidate has almost the same score and symmetric mesh is preferred, it skips both of them.

By default, the score is calculated as follows:

$$score = \frac{\log_{10} \epsilon_0 - \log_{10} \epsilon}{(d_0 - d)^{\xi}}$$

where ϵ and d are an estimated error and an estimated number of DOF of a candidate, respectively, ϵ_0 and d_0 are an estimated error and an estimated number of DOF of the examined element, respectively, and ξ is a convergence exponent.

The parameter `CAND_LIST` specifies which candidates are generated. In a case of quadrilaterals, all possible values and considered candidates are summarized in Fig. 7.

The adaptivity algorithm in Hermes calculates an approximation on fine mesh and uses orthogonal projection to a coarse submesh (so only the problem on the fine mesh is actually solved) to extract low-order part of the solution. This gives two approximations with different orders of accuracy whose difference is used as an a-posteriori error estimate (error function). The error function is used to decide which elements need to be refined as well as to select optimal *hp*-refinement for each element. Hence the adaptivity loop begins with refining the mesh globally:

```
Mesh::ReferenceMeshCreator ref_mesh_creator(mesh);
MeshSharedPtr ref_mesh = ref_mesh_creator.create_ref_mesh();
Space<double>::ReferenceSpaceCreator ref_space_creator(space, ref_mesh);
SpaceSharedPtr<double> ref_space = ref_space_creator.create_ref_space();
```

The new spaces have to be set to the Newton (or linear) solver that we already created (outside of the adaptivity loop):

```
newton.set_space(ref_space);
```

The Newton method is used to solve the fine mesh problem:

```
try
{
  newton.solve(coeff_vec);
}
catch(Hermes::Exceptions::Exception& e)
{
  e.printMsg();
}
```

The coefficient vector is translated into a `Solution`:

```
Solution<double>::vector_to_solution(newton.get_sln_vector(),
ref_space, ref_sln);
```

The `Solution` is projected on the coarse submesh to extract low-order part for error calculation:

```
OGProjection<double>::project_global(space, ref_sln, sln);
```

The function `project_global()` is very general; it can accept multiple spaces, multiple functions, and various projection norms as parameters. For more details, see Hermes Doxygen documentation.

The coarse and reference mesh approximations are inserted into the class `Adapt` instance created earlier `adaptivity` and a global error estimate as well as element error estimates are calculated using the `ErrorCalculator` instance `errorCalculator` (also created earlier):

```
errorCalculator.calculate_errors(sln, ref_sln);
double err_est_rel = errorCalculator.get_total_error_squared();
```

When working with another space than H^1, the `HERMES_H1_NORM` in the initialization of the `ErrorCalculator` can be replaced with `HERMES_HCURL_NORM`, `HERMES_HDIV_NORM`, or `HERMES_L2_NORM`.

If `err_est_rel` is above the prescribed tolerance `ERR_STOP`, we perform mesh adaptation:

```
// If err_est too large, adapt the mesh.
if (err_est_rel < ERR_STOP)
  done = true;
else
{
  // Adapt class may return true if no other elements could have been refined,
// therefore we would be done.
  done = adaptivity.adapt(&selector);

  // Increase the counter of performed adaptivity steps.
  if (done == false)
    as++;
}
```

Sample numerical results are shown in Figs. 8–10.

Convergence curves for adaptive *h*-FEM with linear and quadratic elements, as well as for the *hp*-FEM, both in terms of the number of degrees of freedom and CPU time, are shown in Figs. 11 and 12.

In harmony with theoretical prediction, the convergence curves of adaptive *h*-FEM with linear and quadratic elements have slopes $-1/2$ and -1 on the log–log scale, respectively. Convergence of the adaptive *hp*-FEM is exponential.

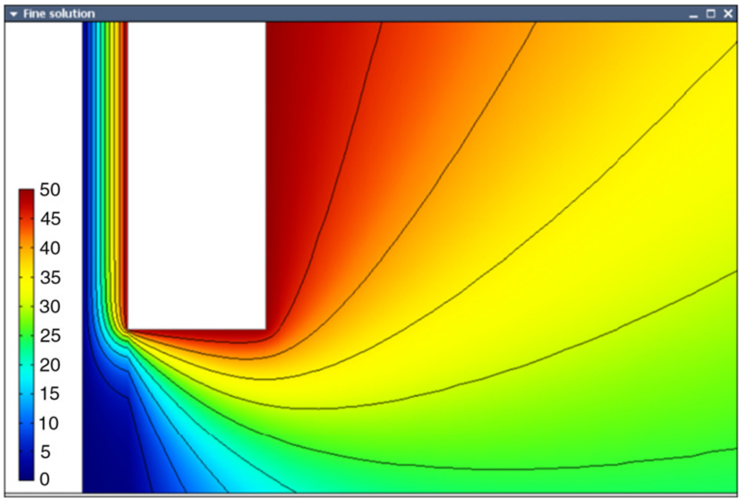

Fig. 8. Detail of the singular electric potential φ at the electrode.

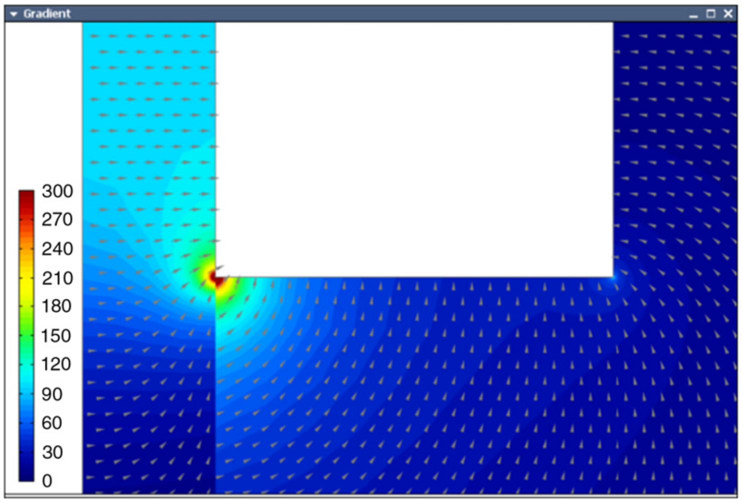

Fig. 9. Detail of the gradient of φ.

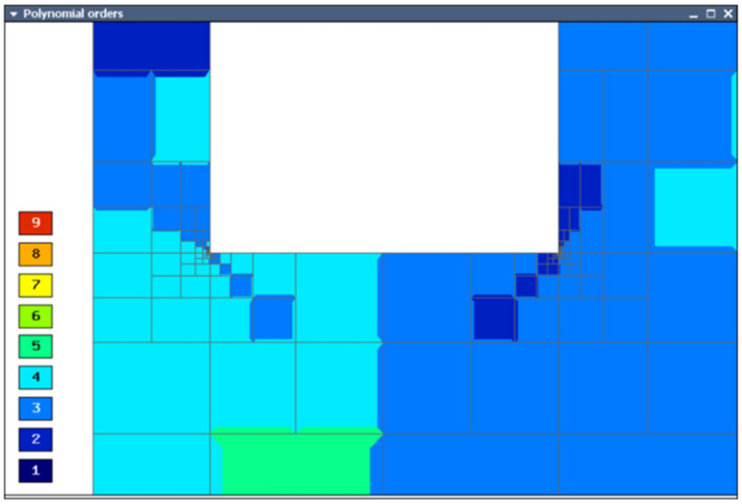

Fig. 10. Resulting *hp*-finite element mesh. Scale indicates polynomial degrees of elements.

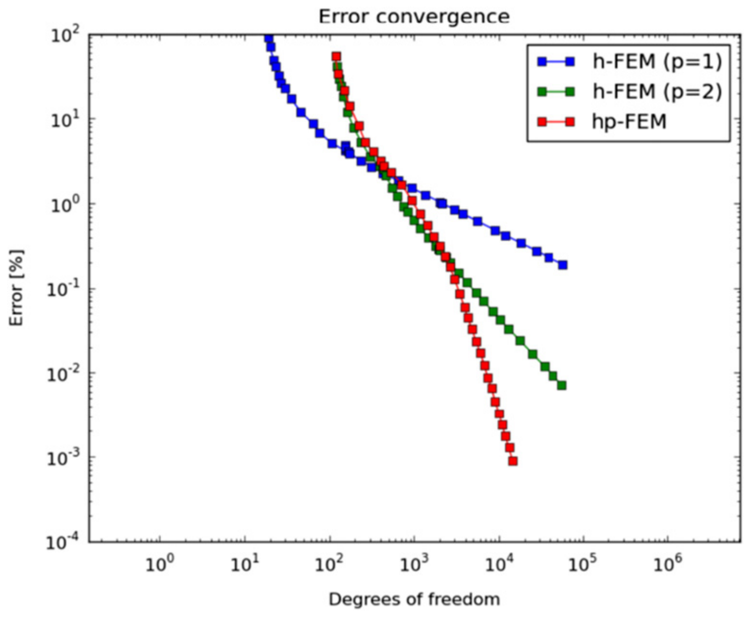

Fig. 11. Error convergence in terms of DOF.

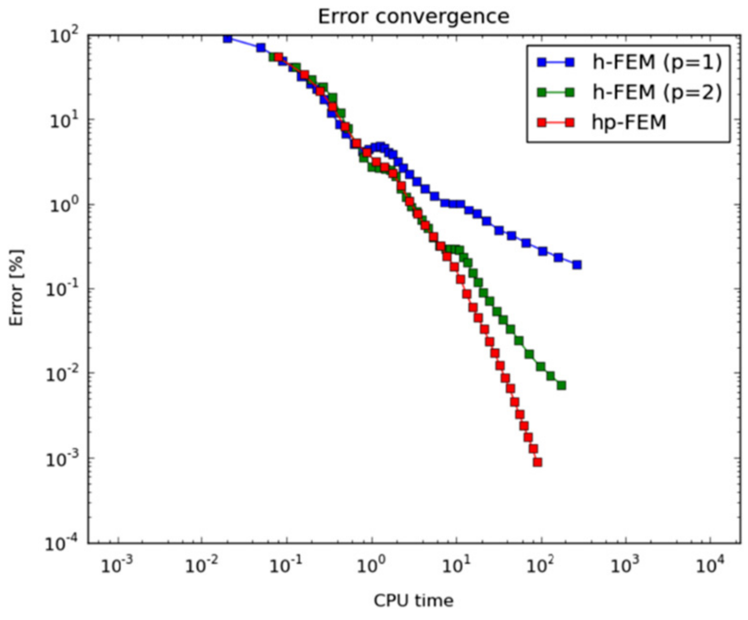

Fig. 12. Error convergence in terms of CPU time.

Acknowledgments

This work was supported by the European Regional Development Fund and Ministry of Education, Youth and Sports of the Czech Republic (Project No. CZ.1.05/2.1.00/03.0094: Regional Innovation Centre for Electrical Engineering—RICE) and Grant project GACR P102/11/0498.

References

[1] P. Solin, L. Korous, Adaptive higher-order finite element methods for transient PDE problems based on embedded higher-order implicit Runge–Kutta methods, J. Comput. Phys. 231 (4) (2012) 1635–1649.
[2] L. Dubcova, P. Solin, G. Hansen, H. Park, Comparison of multimesh *hp*-FEM to interpolation and projection methods for spatial coupling of reactor thermal and neutron diffusion calculations, J. Comput. Phys. 230 (2011) 1182–1197.

[3] P. Solin, J. Cerveny, L. Dubcova, D. Andrs, Monolithic discretization of linear thermoelasticity problems via adaptive multimesh *hp*-FEM, J. Comput. Appl. Math 234 (2010) 2350–2357.
[4] P. Solin, L. Korous, Adaptive *hp*-FEM with dynamical meshes for problems with traveling sharp fronts, Computing (2012) http://dx.doi.org/10.1007/s00607-012-0243-7. Published online.
[5] P. Solin, M. Kuraz, Solving the nonstationary Richards equation with adaptive *hp*-FEM, Adv. Water Resour. 34 (9) (2011) 1062–1081.
[6] P. Solin, J. Cerveny, I. Dolezel, Arbitrary-level hanging nodes and automatic adaptivity in the *hp*-FEM, Math. Comput. Simul. 77 (2008) 117–132.
[7] P. Solin, L. Demkowicz, Goal-oriented *hp*-adaptivity for elliptic problems, Comput. Methods Appl. Mech. Engrg. 193 (2004) 449–468.
[8] P. Solin, K. Segeth, I. Dolezel, Higher-Order Finite Element Methods, Chapman & Hall/CRC Press, Boca Raton, 2003.
[9] P. Karban, F. Mach, P. Kus, D. Panek, I. Dolezel, Numerical solution of coupled problems using code Agros2D, Computing (2013) http://dx.doi.org/10.1007/s00607-013-0294-4. Published online.