Exercises for Tutorial05

Adaptivity

Exercise 1 Playing with the Parameters

The adaptive finite element algorithm is controlled by several parameters which can be set in the ini-file. Besides the polynomial degree these are:

- `steps` : how many times the solve– adapt cycle is executed.
- `uniformlevel` : how many times uniform mesh refinement is used before adaptive refinement starts.
- `tol` : tolerance where the adaptive algorithm is stopped.
- `fraction` : Remember that the global error is estimated from element-wise contributions:
$$\gamma^2(u_h) = \sum_{T\in\mathcal{T}_h} \gamma_T^2(u_h).$$
The fraction parameter controls which elements are marked for refinement. More precisely, the largest threshold γ^* is determined such that
$$\sum_{\{T\in\mathcal{T}_h\,:\,\gamma_T(u_h)\geq\gamma^*\}} \gamma_T^2(u_h) \geq \texttt{fraction}\cdot\gamma^2(u_h).$$
If `fraction` is set to zero no elements need to be refined, i.e. $\gamma^* = \infty$. When `fraction` is set to one then all elements need to be refined, i.e. $\gamma^* = 0$. A smaller `fraction` parameter results in a more optimized mesh but needs more steps and thus might be more expensive. If `fraction` is chosen too large then almost all elements are refined which might also be expensive, so there exists an (unknown) optimal value for `fraction`.

Carry out the following experiments:

1. Choose polynomial degree 1 and fix a tolerance, e.g. `tol`=0.01. Set `steps` to a large number so that the algorithm is not stopped before the tolerance is reached. Set `uniformlevel` to zero. Now vary the `fraction` parameter and measure the execution time with
```
$ time ./exercise05
```
2. Repeat the previous experiment with polynomial degrees two and three. Compare the optimal meshes generated for the different polynomial degrees.

3. The idea of the heuristic refinement algorithm refining only the elements with the largest contribution to the error is to *equilibrate* the error. To check how well this is achieved use the calculator filter in ParaView (use cell data) to plot $\log(\gamma_T(u_h))$. Note that the cell data exported by the (unchanged) code is $\gamma_T^2(u_h)$! Now check the minimum and maximum error contributions. What happens directly at the singularity, i.e. the point $(0, 0)$?

Exercise 2 Compute the True L_2-error

Disclaimer: In this exercise we compute and evalute the L_2-norm of the error. The residual based error estimator implemented in the code, however, estimates the H^1-seminorm (which is equivalent to the H^1-norm here) and thus also optimizes the mesh with respect to that norm. This is done to make the exercise as simple as possible. It would be more appropriate to carry out the experiments either with the true H^1-norm or to change the estimator to the L_2-norm.

To do this exercise we need some more background on PDELab grid functions. The following code known from several tutorials by now constructs a PDELab grid function object `g`:

```
Problem<RF> problem(eta);
auto glambda =
[&](const auto& e, const auto& x){return problem.g(e,x);};
auto g = Dune::PDELab::makeGridFunctionFromCallable(gv,glambda);
```

Also the following code segment used in many tutorials constructs a PDELab grid function object `zdgf` from a grid function space and a coefficient vector:

```
typedef Dune::PDELab::DiscreteGridFunction<GFS,Z> ZDGF;
ZDGF zdgf(gfs,z);
```

PDELab grid functions can be evaluated in local coordinates given by an element and a local coordinate within the corresponding reference element. The result is stored in an additional argument passed by reference. Here is a code segment doing the job:

```
Dune::FieldVector<RF,1> truesolution(0.0);
g.evaluate(e,x,truesolution);
```

Here we assume that `e` is an element and `x` is a local coordinate. The result is stored in a `Dune::FieldVector` with one component of type `RF`. This allows also to return vector-valued results.

Now here are your tasks:

1. Extend the code in the file `driver.hh` in `tutorial05/exercise/task` to provide a grid function that provides the error $u - u_h$. Note that the method `g` in the parameter class already returns the true solution (for $\eta = 0$). You can solve this task by writing a lambda function subtracting the values returned by `g.evaluate` and `zdgf.evaluate`.

2. Compute the L_2-norm of the error, i.e. $\|u - u_h\|_0 = \sqrt{\int_\Omega (u(x) - u_h(x))^2 \, dx}$. This can be done by creating a grid function with the squared error using `Dune::PDELab::SqrGridFunctionAdapter` from `<dune/pdelab/function/sqr.hh>` and the function `Dune::PDELab::integrateGridFunction` contained in the header file `<dune/pdelab/common/functionutilities.hh>`. Output the result to the console by writing in a single line a tag like `L2ERROR`, the number of degrees of freedom using `gfs.globalSize()` and the L_2-norm. This lets you grep the results later for post-processing.

3. Investigate the L_2-error for different polynomial degrees. Run the code using different polynomial degrees and also uniform and adaptive (use your optimal `fraction`) refinement. Use `gnuplot` to produce graphs such as those in Figure 1. An example gnuplot file `plot.gp` is given.

   The graph shows that for uniform refinement polynomial degree 2 is better than polynomial degree 1 only by a constant factor. For adaptive refinement a better convergence rate can be achieved. The plot also shows that for adaptive refinement the optimal convergence rate can be recovered. For P_1 finite elements and fully regular solution in $H^2(\Omega)$ the a-priori estimate for uniform refinement yields $\|u - u_h\|_0 \leq Ch^2$. Expressing h in terms of the number of degrees of freedom N yields $h \sim N^{-1/d}$, so $\|u - u_h\|_0 \leq Ch^2 \leq C'N^{-1}$ for $d = 2$. The curve N^{-1} is shown for comparison in the plot. Clearly, the result for P_1 is parallel to that line.

4. Optional: Also write the true error to the VTK output file.

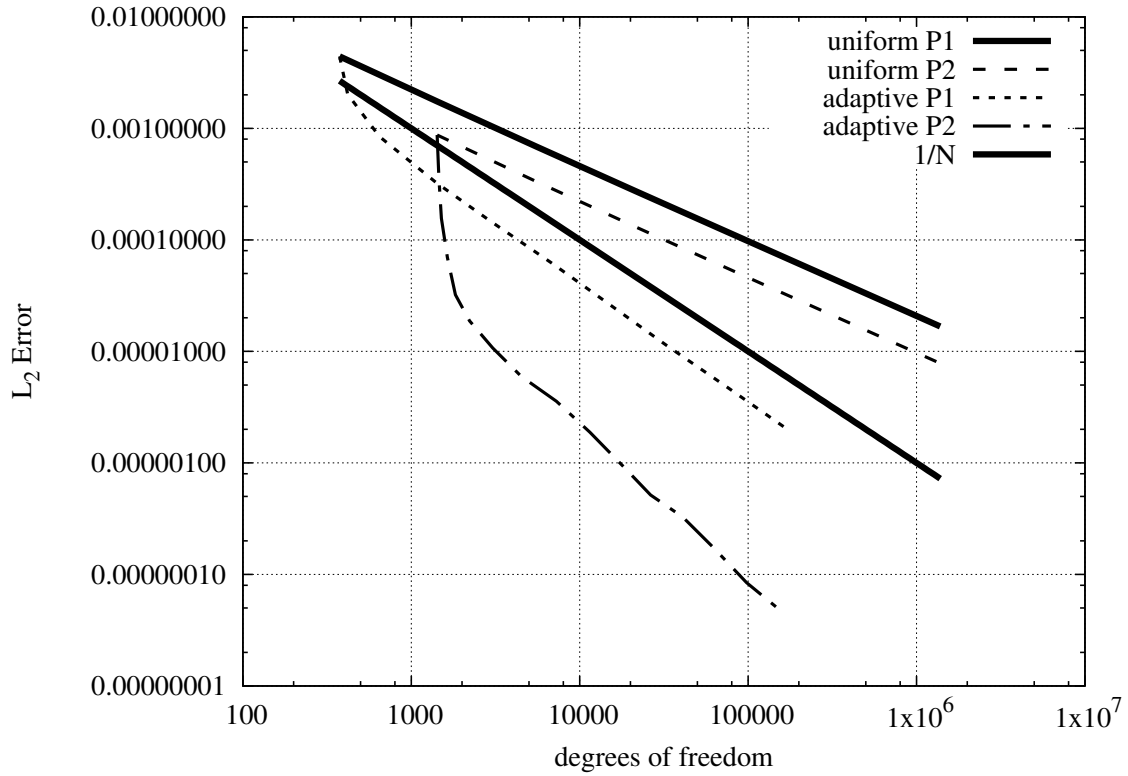

Figure 1: L_2-error versus number of degrees of freedom for different polynomial degrees.