Numerical simulations, topic: Compressible Flow

Lukas Korous, hp-FEM group

University of West Bohemia, Pilsen, University of Nevada, Reno

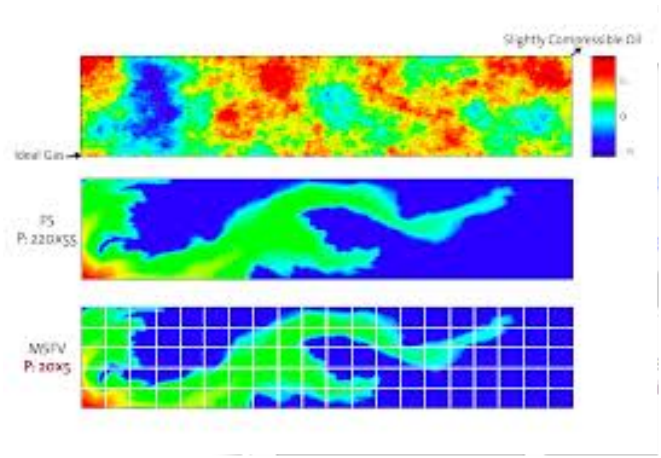
November 4, 2012

Introduction - Compressible flow in nature




Figure: Compressible flow in nature

Introduction - Compressible flow modelling



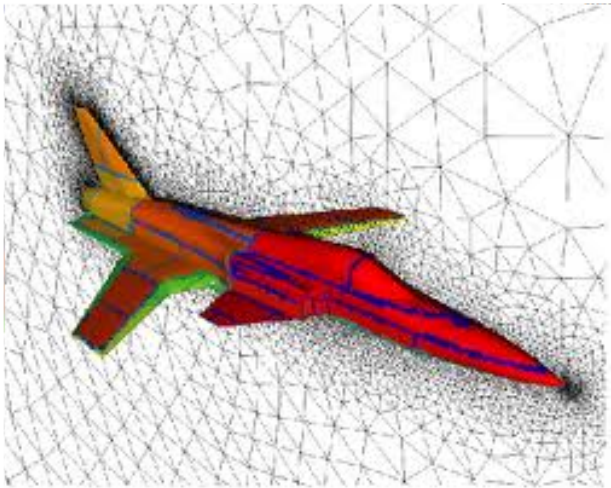


Figure: Compressible flow modelling

Introduction - Hermes: numerical software

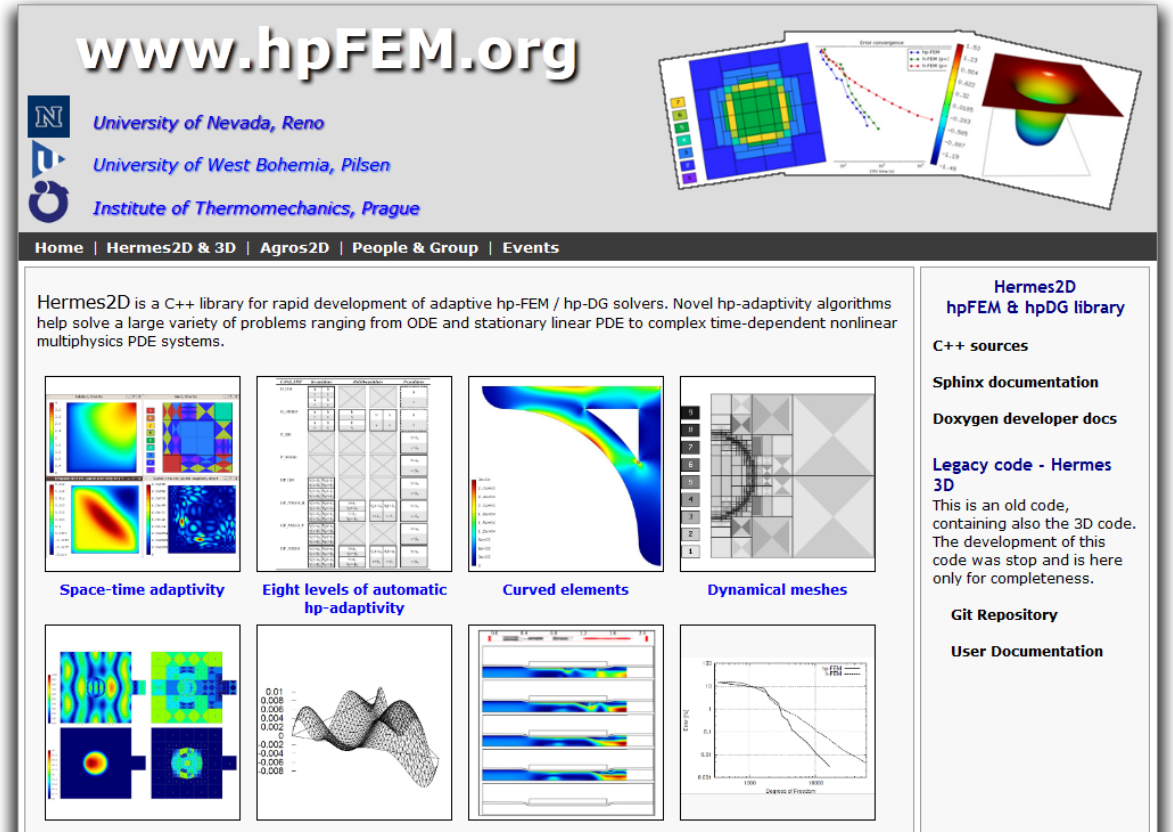


Figure: http://www.hpfem.org/hermes

Introduction - Hermes

- Unified library for handling both real and complex problems using C++ templating
- Library capable of solving problems using hp-FEM and hp-DG methods and their combination
- Division of the library's code into easy-to-manage namespace- and class- structure
- Shared memory parallel code (using OpenMP)
- XML save / load of the most important classes using provided XML schemas
- Calculations with physical quantities defined in different subdomains
- Exception safe API
- Supported platforms: Linux, Windows, (MAC)
- Well arranged doxygen documentation
- User documentation, with described examples

Hermes: physical fields

- Electrostatic field
- Electric current field
- Magnetic field
- Heat transfer and Flame Propagation
- Structural mechanics and Thermoelasticity
- Acoustic field
- Incompressible and **Compressible flow**
- Advection-diffusion-reaction
- Richards Equation, Wave Equation, Nernst-Planck Equation, Neutronics...
- RF field: in development

Compressible flow requirements

In modelling of compressible flow, we would like to solve problems that

- we possess no a-priori knowledge about
- have high Mach numbers and irregularly shaped domains
- may have sharp discontinuities as well as smooth areas
- may have rapidly changing solutions

Plus we would like to solve those in a reasonable time.

Compressible flow problems solutions

- our discretization should be able to capture the solution
- there should not be a need for thoroughly prepared mesh as an input
- we should use appropriate discretization that can handle discontinuities, but which will not waste degrees of freedom for smooth areas
- our computational mesh should evolve with the solution

All this is achieved with adaptive hp-DG method.

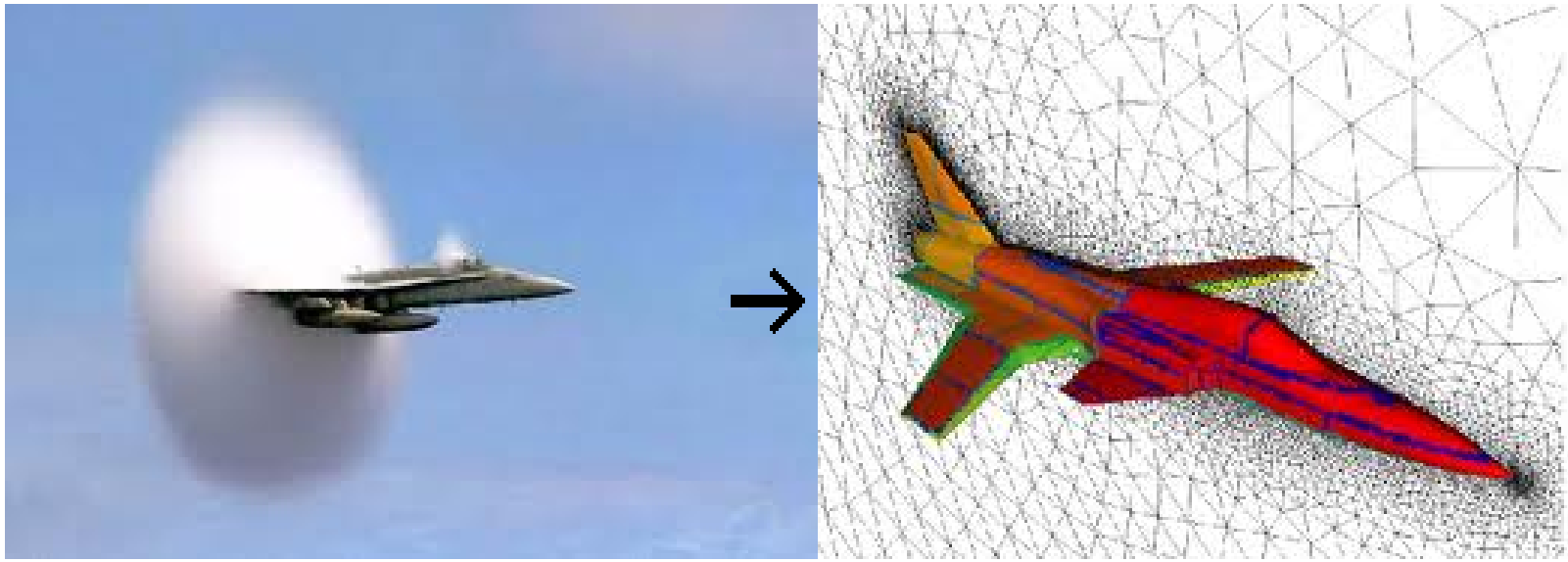


Figure: Discretization

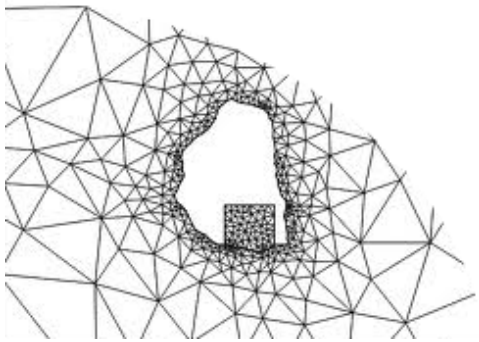


Figure: Discretization

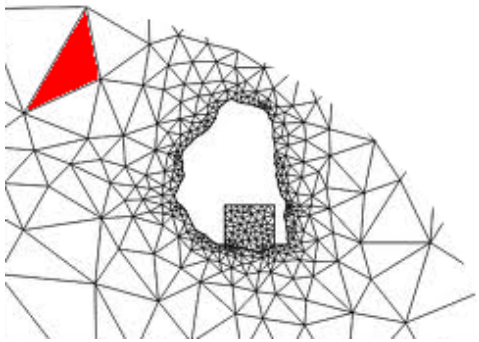


Figure: Discretization - element

Numerical software internals - 2. Equations of the physical field

$$
\begin{aligned}
\frac{\partial \rho}{\partial t} + \operatorname{div}(\rho v) &= 0, \\
\frac{\partial(\rho v)}{\partial t} + \operatorname{div}(\rho v \otimes v) &= \rho f - \nabla p + \mu \Delta v + (\mu + \lambda)\, \nabla \operatorname{div} v, \\
\frac{\partial E}{\partial t} + \operatorname{div}(E v) &= \rho f \cdot v - \operatorname{div}(p v) + \operatorname{div}(\lambda v \operatorname{div} v) + \\
&+ \operatorname{div}(2\mu \mathbb{D}(v)\, v) + \rho q - \operatorname{div} \phi_q, \\
p &= (\gamma - 1)\left(E - \rho |v|^2/2\right), \\
\theta &= \left(E/\rho - |v|^2/2\right)/c_v.
\end{aligned}
$$

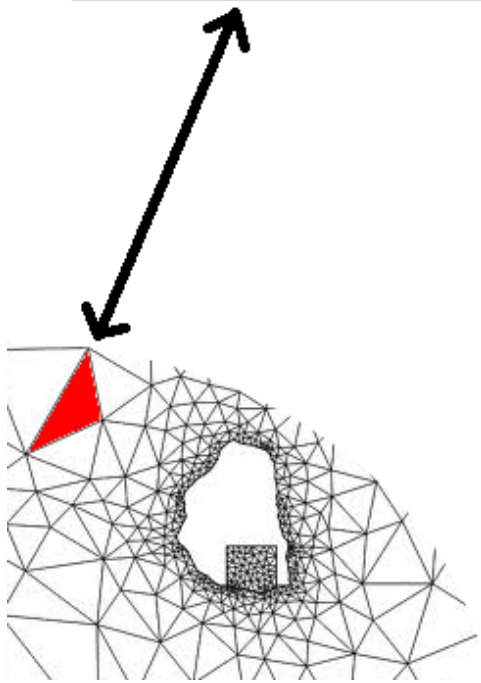


Figure: Equations of the physical field calculated on the element

Numerical software internals - 2. Boundary conditions, problem data

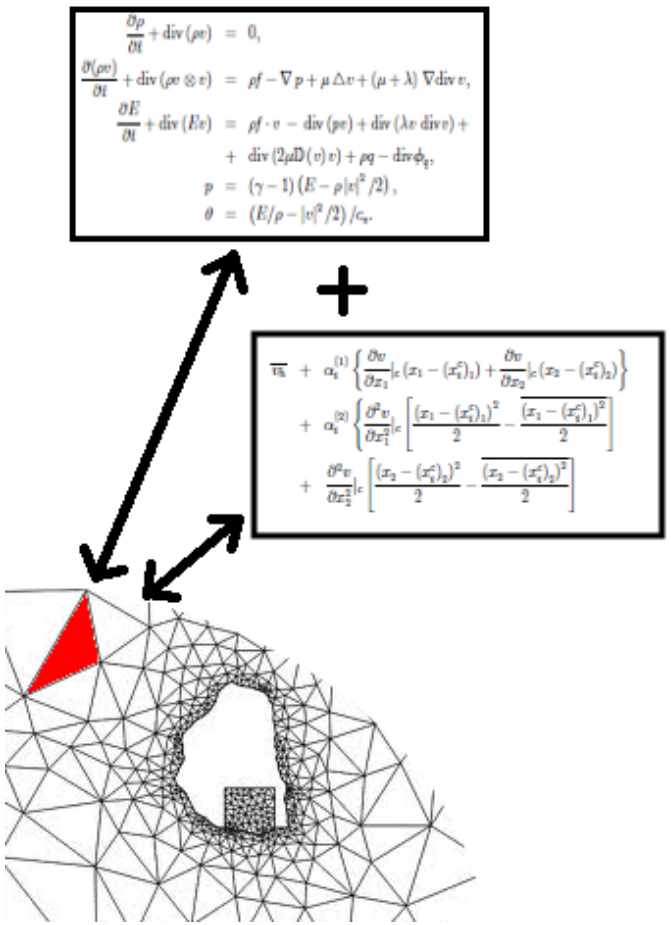


Figure: Boundary conditions, specific data - complete the calculation on the element

Numerical software internals - 3. Assemble the matrix equation

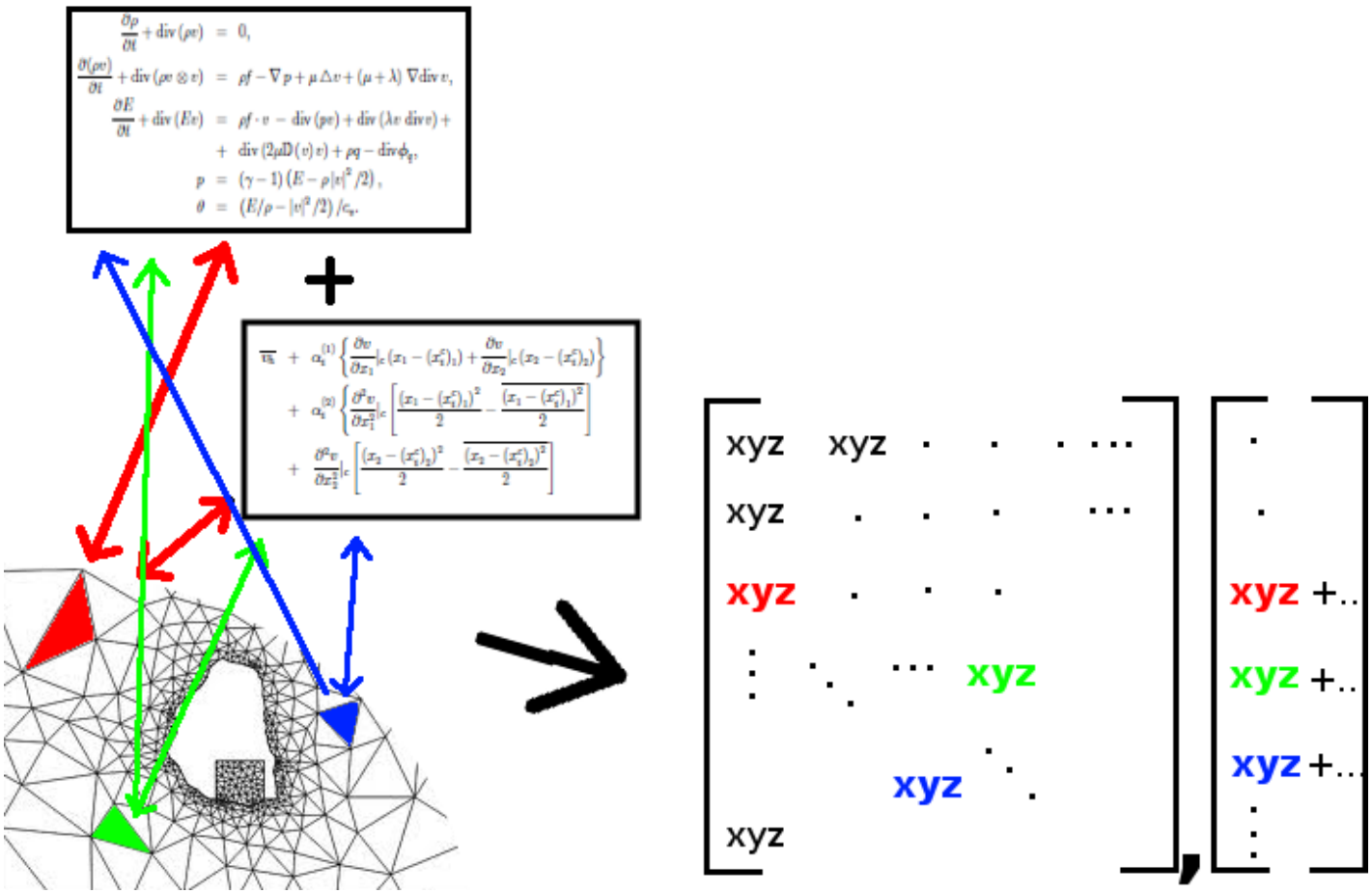


Figure: Assemble (calculate) matrix and right-hand side entries

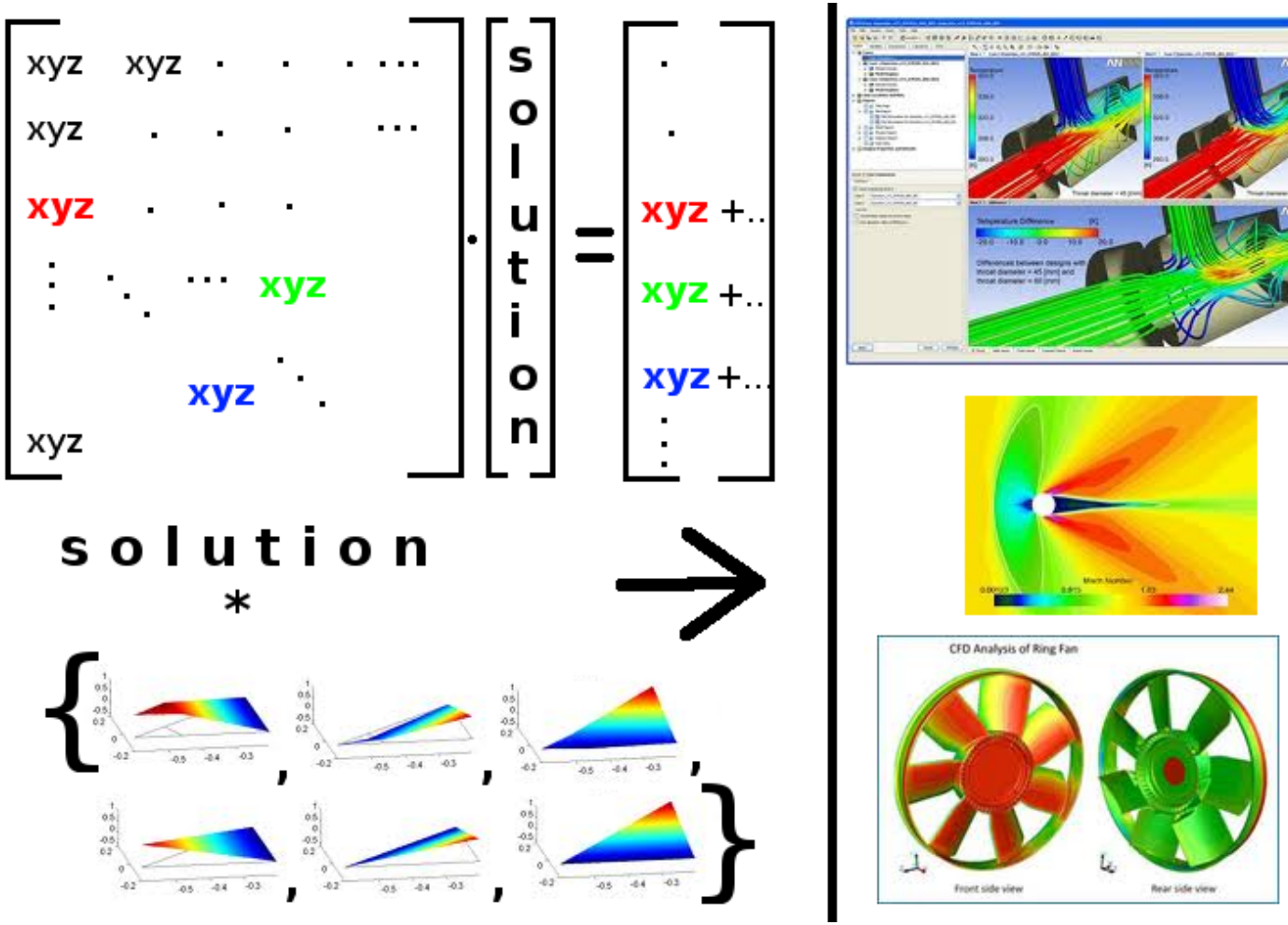


Figure: 1.Solve the matrix equation, obtain results

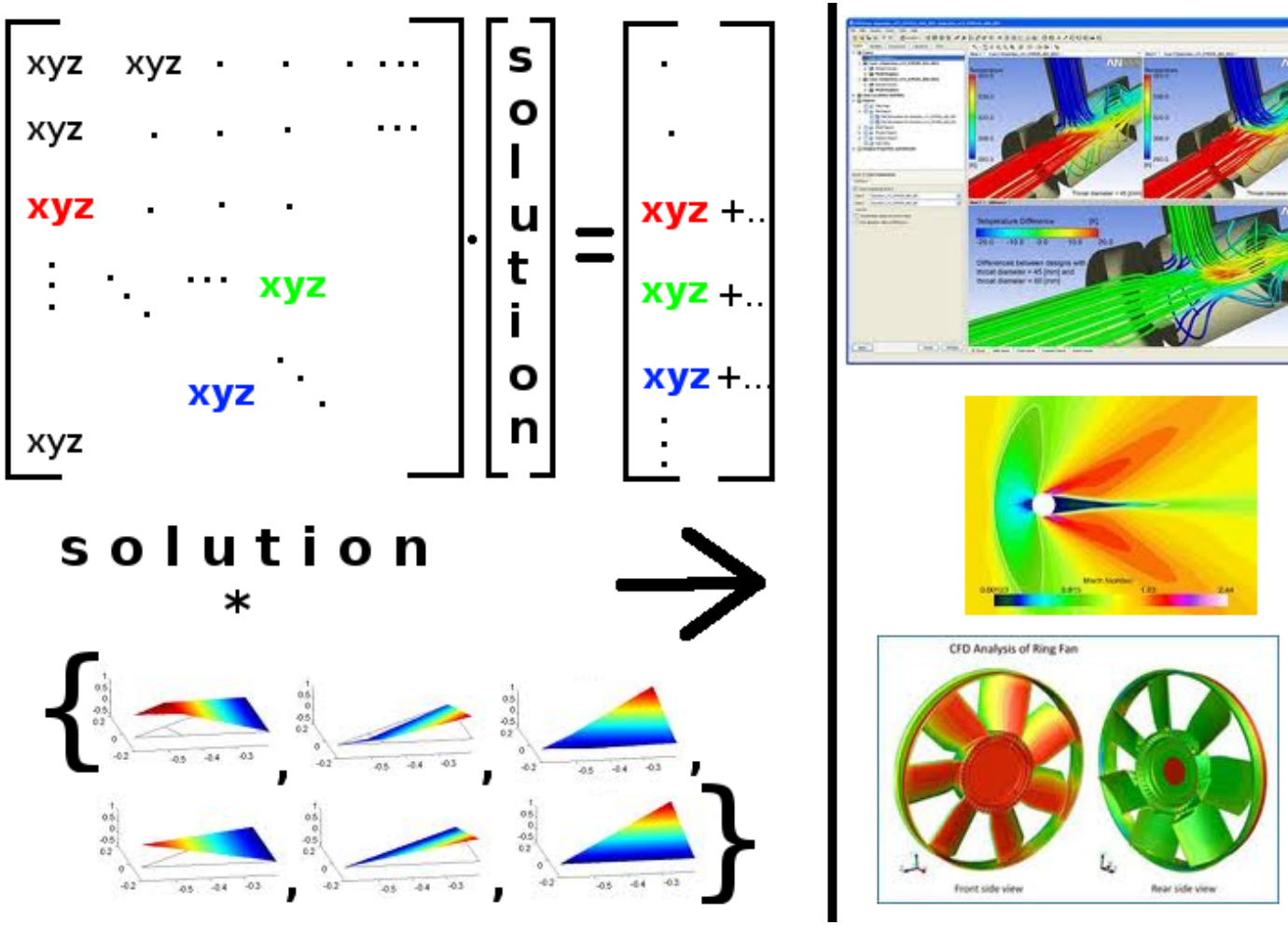


Figure: Qualitatively and quantitatively inspect results, improve the discretization and start over if not satisfied

Hermes parallel hp-FEM and hp-DG (multimesh) assembling of (nonlinear) problems


```
#pragma omp parallel shared(traverse_master, matrix, right_hand_side)
num_threads(Hermes2DApi.getParamValue(Hermes::Hermes2D::numThreads))
{
#pragma omp for schedule(dynamic, CHUNKSIZE)
  for (state_i = 0; state_i < num_states; state_i++)
  {
#pragma omp critical (get_next_state)
    current_state = traverse[thread_id].get_next_state
        (&traverse_master.top, &traverse_master.id);

    basis_functions = basis_functions[thread_id];
    test_functions = test_functions[thread_id];
    ref_mappings = refmaps[thread_id];
    assembly_lists = assembly_lists[thread_id];
    forms = &(forms[thread_id].front());

    // Assemble volumetric forms and surface forms corresponding to boundary conditions.
    assemble_one_state(basis_functions, test_functions, ref_mappings, assembly_lists,
      &current_state, forms);

    // DG
    for (edge = 0; edge < current_state.get_number_of_edges(); edge++)
    {
      // Find the neighbors of this element over this edge.
      neighbor_groups[edge]->find_neighbors(&current_state, edge, basis_functions,
        test_functions, ref_mappings, assembly_lists, forms);

      // Go through possibly many neighbors and assemble the forms.
      for (neighbor = 0; neighbor < neighbor_groups[edge].number_of_neighbors; neighbor++)
        neighbor_groups[edge]->assemble_one_DG_edge(neighbor);
    }
  }
}
```

Hermes parallel hp-FEM and hp-DG (multimesh) assembling of (nonlinear) problems


```
#pragma omp parallel shared(traverse_master, matrix, right_hand_side)
num_threads(Hermes2DApi.getParamValue(Hermes::Hermes2D::numThreads))
{
#pragma omp for schedule(dynamic, CHUNKSIZE)
  for (state_i = 0; state_i < num_states; state_i++)
  {
#pragma omp critical (get_next_state)
    current_state = traverse[thread_id].get_next_state
        (&traverse_master.top, &traverse_master.id);

    basis_functions = basis_functions[thread_id];
    test_functions = test_functions[thread_id];
    ref_mappings = refmaps[thread_id];
    assembly_lists = assembly_lists[thread_id];
    forms = &(forms[thread_id].front());

    // Assemble volumetric forms and surface forms corresponding to boundary conditions.
    assemble_one_state(basis_functions, test_functions, ref_mappings, assembly_lists,
      &current_state, forms);

    // DG
    for (edge = 0; edge < current_state.get_number_of_edges(); edge++)
    {
      // Find the neighbors of this element over this edge.
      neighbor_groups[edge]->find_neighbors(&current_state, edge, basis_functions,
        test_functions, ref_mappings, assembly_lists, forms);

      // Go through possibly many neighbors and assemble the forms.
      for (neighbor = 0; neighbor < neighbor_groups[edge].number_of_neighbors; neighbor++)
        neighbor_groups[edge]->assemble_one_DG_edge(neighbor);
    }
  }
}
```

Hermes parallel hp-FEM and hp-DG (multimesh) assembling of (nonlinear) problems


```
#pragma omp parallel shared(traverse_master, matrix, right_hand_side)
num_threads(Hermes2DApi.getParamValue(Hermes::Hermes2D::numThreads))
{
#pragma omp for schedule(dynamic, CHUNKSIZE)
  for (state_i = 0; state_i < num_states; state_i++)
  {
#pragma omp critical (get_next_state)
    current_state = traverse[thread_id].get_next_state
       (&traverse_master.top, &traverse_master.id);

    basis_functions = basis_functions[thread_id];
    test_functions = test_functions[thread_id];
    ref_mappings = refmaps[thread_id];
    assembly_lists = assembly_lists[thread_id];
    forms = &(forms[thread_id].front());

    // Assemble volumetric forms and surface forms corresponding to boundary conditions.
    assemble_one_state(basis_functions, test_functions, ref_mappings, assembly_lists,
      &current_state, forms);

    // DG
    for (edge = 0; edge < current_state.get_number_of_edges(); edge++)
    {
      // Find the neighbors of this element over this edge.
      neighbor_groups[edge]->find_neighbors(&current_state, edge, basis_functions,
        test_functions, ref_mappings, assembly_lists, forms);

      // Go through possibly many neighbors and assemble the forms.
      for (neighbor = 0; neighbor < neighbor_groups[edge].number_of_neighbors; neighbor++)
        neighbor_groups[edge]->assemble_one_DG_edge(neighbor);
    }
  }
}
```

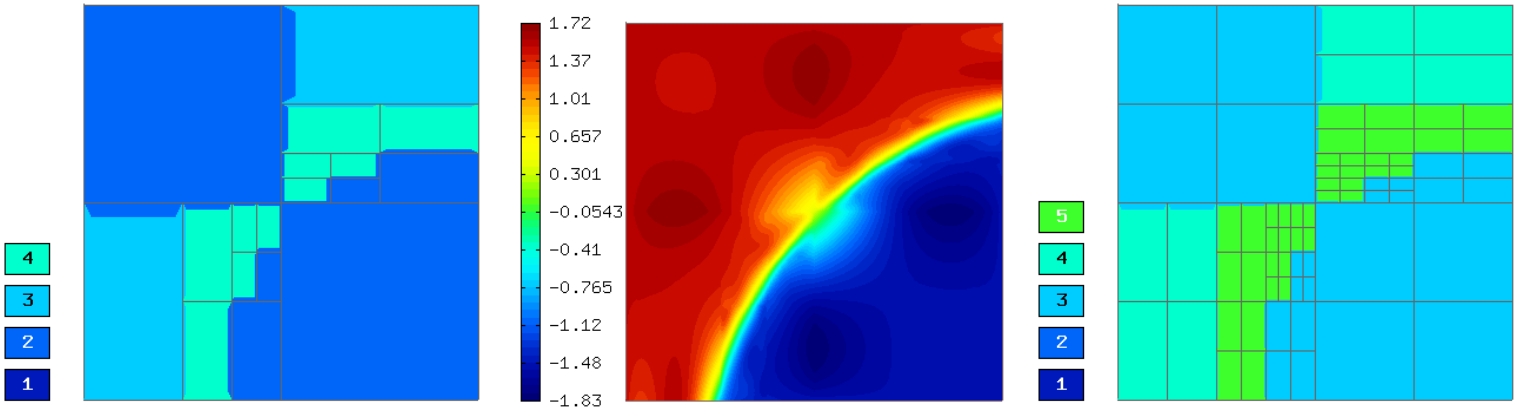
Hermes parallel hp-FEM and hp-DG (multimesh) assembling of (nonlinear) problems


```
#pragma omp parallel shared(traverse_master, matrix, right_hand_side)
num_threads(Hermes2DApi.getParamValue(Hermes::Hermes2D::numThreads))
{
#pragma omp for schedule(dynamic, CHUNKSIZE)
  for (state_i = 0; state_i < num_states; state_i++)
  {
#pragma omp critical (get_next_state)
    current_state = traverse[thread_id].get_next_state
        (&traverse_master.top, &traverse_master.id);

    basis_functions = basis_functions[thread_id];
    test_functions = test_functions[thread_id];
    ref_mappings = refmaps[thread_id];
    assembly_lists = assembly_lists[thread_id];
    forms = &(forms[thread_id].front());

    // Assemble volumetric forms and surface forms corresponding to boundary conditions.
    assemble_one_state(basis_functions, test_functions, ref_mappings, assembly_lists,
      &current_state, forms);

    // DG
    for (edge = 0; edge < current_state.get_number_of_edges(); edge++)
    {
      // Find the neighbors of this element over this edge.
      neighbor_groups[edge]->find_neighbors(&current_state, edge, basis_functions,
        test_functions, ref_mappings, assembly_lists, forms);

      // Go through possibly many neighbors and assemble the forms.
      for (neighbor = 0; neighbor < neighbor_groups[edge].number_of_neighbors; neighbor++)
        neighbor_groups[edge]->assemble_one_DG_edge(neighbor);
    }
  }
}
```

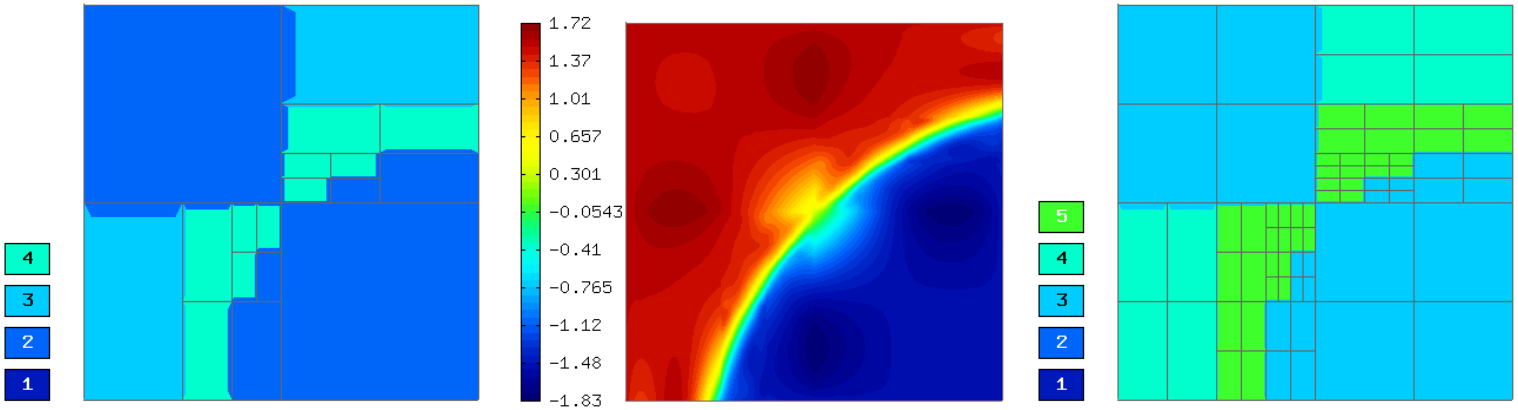
Hermes parallel hp-FEM and hp-DG (multimesh) assembling of (nonlinear) problems


```
#pragma omp parallel shared(traverse_master, matrix, right_hand_side)
num_threads(Hermes2DApi.getParamValue(Hermes::Hermes2D::numThreads))
{
#pragma omp for schedule(dynamic, CHUNKSIZE)
  for (state_i = 0; state_i < num_states; state_i++)
  {
#pragma omp critical (get_next_state)
    current_state = traverse[thread_id].get_next_state
        (&traverse_master.top, &traverse_master.id);

    basis_functions = basis_functions[thread_id];
    test_functions = test_functions[thread_id];
    ref_mappings = refmaps[thread_id];
    assembly_lists = assembly_lists[thread_id];
    forms = &(forms[thread_id].front());

    // Assemble volumetric forms and surface forms corresponding to boundary conditions.
    assemble_one_state(basis_functions, test_functions, ref_mappings, assembly_lists,
      &current_state, forms);

    // DG
    for (edge = 0; edge < current_state.get_number_of_edges(); edge++)
    {
      // Find the neighbors of this element over this edge.
      neighbor_groups[edge]->find_neighbors(&current_state, edge, basis_functions,
        test_functions, ref_mappings, assembly_lists, forms);

      // Go through possibly many neighbors and assemble the forms.
      for (neighbor = 0; neighbor < neighbor_groups[edge].number_of_neighbors; neighbor++)
        neighbor_groups[edge]->assemble_one_DG_edge(neighbor);
    }
  }
}
```

Hermes PDE-independent hp-adaptive algorithm for FEM



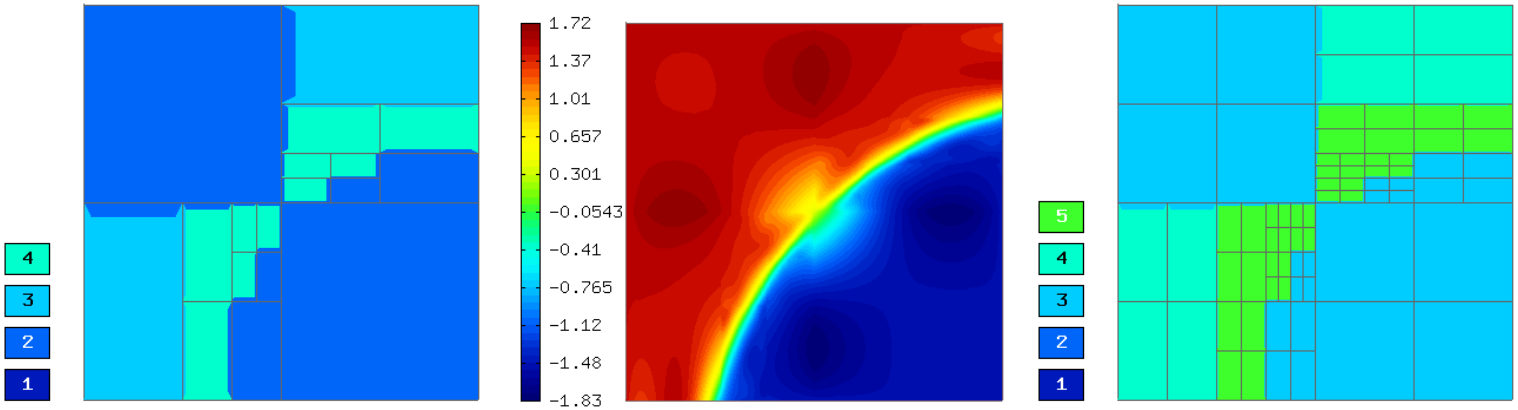
Hermes PDE-independent hp-adaptive algorithm for FEM



- By default (overridable), error estimator as a norm of the difference between the (reference) solution and the solution obtained by projection of the (reference) solution onto the *coarse Space*. The formula is $Error = Norm\left(ref - coarse\right) / Norm\left(ref\right)$.
- For every element, multiple refinement Candidates (h- Candidates, p- Candidates, hp- (aniso-) Candidates) are considered, and their element error is estimated using, again, local projections, but any other error estimator supplied by the user via C++ derived classes can be used.

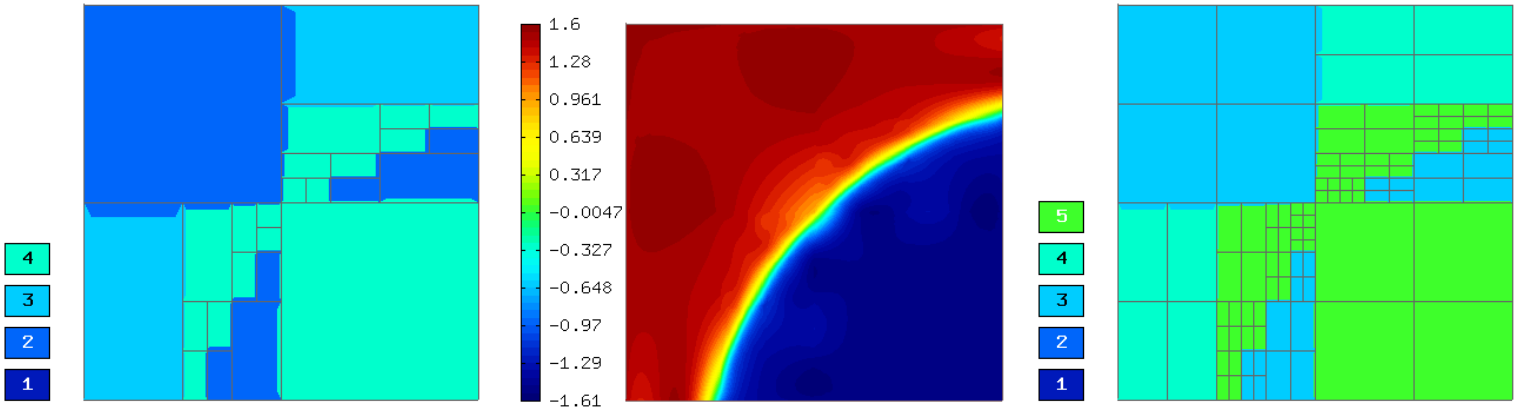
coarse Space: The (reference) Space is actually created from this coarse Space by globally increasing polynomial degree and refining all Elements.

Hermes PDE-independent hp-adaptive algorithm for DG



- Error estimator based on a norm of the difference between the (reference) solution and the solution obtained by projection of the (reference) solution onto the *coarse Space*. The formula is $Error = Norm\left(ref - coarse\right) / Norm\left(ref\right)$.
- Suitable Shock capturing / Flux limiting method needs to be applied to the (reference) solution BEFORE projecting onto the coarse mesh, as the calculated error estimate would give incorrect results otherwise.
- For every element, multiple refinement Candidates (h- Candidates, p- Candidates, hp- (aniso-) Candidates) are considered (customizable by the user), and their error estimate is calculated using local projections any other error estimator supplied by the user via C++ derived classes.

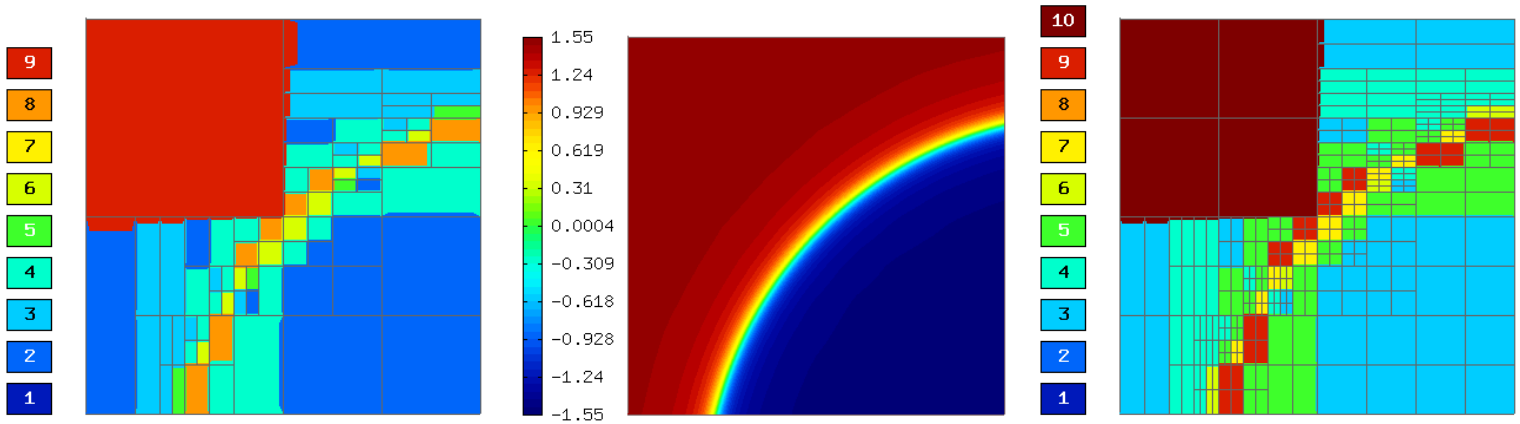
Hermes PDE-independent hp-adaptive algorithm



- All Elements are initially ordered by their error estimate and they are being refined in this order unless the user does not specify otherwise.
- The Hermes library offers various strategies how to limit the number of refinements (amount of error processed, count of Elements refined, relative error wrt. the Element with the largest error, etc.).
- If an Element is chosen for refinement, the Candidate with the lowest *score* is then selected and appropriate changes in Mesh and Space structures are made.

score: The number of degrees of freedom with respect to added Degrees of Freedom. The relation calculating the score is customizable using C++ derived classes from the RefinementSelector class.

Hermes PDE-independent hp-adaptive algorithm



- The refined Spaces are then used as the coarse Spaces in the next adaptivity step.
- The adaptivity process (at that time step) ends after a threshold for the relative difference is met, or the prescribed number of Degrees of Freedom is achieved, etc.
- This way (by piling up refinement layers) we would end up with unnecessarily large meshes (a lot of elements, high polynomial degrees), that is why *derefinement* should be done periodically (every n-th time step):
  - Decrease polynomial degree of an Element (for any set of Elements, e.g. all of them)
  - Unrefine an Element, i.e. take out 1 level of refinement (the same)
  - Reset the mesh completely to some base mesh.

Hermes PDE-independent hp-adaptive algorithm for DG


```
1  while(!done)
2  {
3    ref_spaces = construct_ref_spaces(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
4
5    // Uses the assembling algorithm & linear algebraic routines from TPLs.
     // "solver" is either LinearSolver<double>, or NewtonSolver<double>, or PicardSolver<double>.
6    solver.solve(ref_spaces_const);
7
8    // Here, the flux_limiter stands for an implementation of flux limiter,
     // in the example that follows, limiter according to
     // D. Kuzmin.:  A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods,
     // J. Comput. Appl. Math., 233(12), 3077-3085, 2010. was used.
9    flux_limiter.limit_according_to_detector(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
10   flux_limiter.get_limited_solutions(&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e);
11
12   OGProjection<double>::project_global([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e],
       [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e], [&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e]);
13
14   // Calculate the errors and create list of Elements accordingly.
15   if(adaptivity->calc_error_estimate([&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e],
           [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e]) < threshold) { done = true; break; }
16
17   // Adapt.
18   #pragma omp parallel shared(...) ...
19   {
20   #pragma omp for schedule(dynamic, CHUNKSIZE)
21     for(element = 0; element < elements.size(); element++)
22     {
23       if(adaptive_strategy.done) break;
24       refinement_selectors = global_refinement_selectors[thread_id];
25       rslns = rslns[thread_id];
26
27       refinement_selectors->create_candidates(element, rslns, "user settings");
28       refinement_selectors->evaluate_candidates(element, rslns, &avg_error, &dev_error);
29       refinement_selectors->select_best_candidate(element, avg_error, dev_error);
30     }
31   }
32   adaptivity->apply_refinements([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e]);
33 }
```

Hermes PDE-independent hp-adaptive algorithm for DG


```
1  while(!done)
2  {
3    ref_spaces = construct_ref_spaces(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
4
5    // Uses the assembling algorithm & linear algebraic routines from TPLs.
     // "solver" is either LinearSolver<double>, or NewtonSolver<double>, or PicardSolver<double>.
6    solver.solve(ref_spaces_const);
7
8    // Here, the flux_limiter stands for an implementation of flux limiter,
     // in the example that follows, limiter according to
     // D. Kuzmin.:  A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods,
     // J. Comput. Appl. Math., 233(12), 3077-3085, 2010. was used.
9    flux_limiter.limit_according_to_detector(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
10   flux_limiter.get_limited_solutions(&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e);
11
12   OGProjection<double>::project_global([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e],
       [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e], [&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e]);
13
14   // Calculate the errors and create list of Elements accordingly.
15   if(adaptivity->calc_error_estimate([&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e],
           [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e]) < threshold) { done = true; break; }
16
17   // Adapt.
18   #pragma omp parallel shared(...) ...
19   {
20   #pragma omp for schedule(dynamic, CHUNKSIZE)
21     for(element = 0; element < element.size(); element++)
22     {
23       if(adaptive_strategy.done) break;
24       refinement_selectors = global_refinement_selectors[thread_id];
25       rslns = rslns[thread_id];
26
27       refinement_selectors->create_candidates(element, rslns, "user settings");
28       refinement_selectors->evaluate_candidates(element, rslns, &avg_error, &dev_error);
29       refinement_selectors->select_best_candidate(element, avg_error, dev_error);
30     }
31   }
32   adaptivity->apply_refinements([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e]);
33 }
```

Hermes PDE-independent hp-adaptive algorithm for DG


```
while(!done)
{
  ref_spaces = construct_ref_spaces(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);

  // Uses the assembling algorithm & linear algebraic routines from TPLs.
  // "solver" is either LinearSolver<double>, or NewtonSolver<double>, or PicardSolver<double>
  solver.solve(ref_spaces_const);

  // Here, the flux_limiter stands for an implementation of flux limiter,
  // in the example that follows, limiter according to
  // D. Kuzmin.:  A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods,
  // J. Comput. Appl. Math., 233(12), 3077-3085, 2010. was used.
  flux_limiter.limit_according_to_detector(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
  flux_limiter.get_limited_solutions(&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e);

  OGProjection<double>::project_global([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e],
    [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e], [&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e]);

  // Calculate the errors and create list of Elements accordingly.
  if(adaptivity->calc_error_estimate([&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e],
        [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e]) < threshold) { done = true; break; }

  // Adapt.
  #pragma omp parallel shared(...) ...
  {
  #pragma omp for schedule(dynamic, CHUNKSIZE)
    for(element = 0; element < element.size(); element++)
    {
      if(adaptive_strategy.done) break;
      refinement_selectors = global_refinement_selectors[thread_id];
      rslns = rslns[thread_id];

      refinement_selectors->create_candidates(element, rslns, "user settings");
      refinement_selectors->evaluate_candidates(element, rslns, &avg_error, &dev_error);
      refinement_selectors->select_best_candidate(element, avg_error, dev_error);
    }
  }
  adaptivity->apply_refinements([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e]);
}
```

Hermes PDE-independent hp-adaptive algorithm for DG


```
1  while(!done)
2  {
3    ref_spaces = construct_ref_spaces(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
4
5    // Uses the assembling algorithm & linear algebraic routines from TPLs.
     // "solver" is either LinearSolver<double>, or NewtonSolver<double>, or PicardSolver<double>.
6    solver.solve(ref_spaces_const);
7
8    // Here, the flux_limiter stands for an implementation of flux limiter,
     // in the example that follows, limiter according to
     // D. Kuzmin.:  A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods,
     // J. Comput. Appl. Math., 233(12), 3077-3085, 2010. was used.
9    flux_limiter.limit_according_to_detector(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
10   flux_limiter.get_limited_solutions(&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e);
11
12   OGProjection<double>::project_global([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e],
       [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e], [&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e]);
13
14   // Calculate the errors and create list of Elements accordingly.
15   if(adaptivity->calc_error_estimate([&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e],
           [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e]) < threshold) { done = true; break; }
16
17   // Adapt.
18   #pragma omp parallel shared(...) ...
19   {
20   #pragma omp for schedule(dynamic, CHUNKSIZE)
21     for(element = 0; element < element.size(); element++)
22     {
23       if(adaptive_strategy.done) break;
24       refinement_selectors = global_refinement_selectors[thread_id];
25       rslns = rslns[thread_id];
26
27       refinement_selectors->create_candidates(element, rslns, "user settings");
28       refinement_selectors->evaluate_candidates(element, rslns, &avg_error, &dev_error);
29       refinement_selectors->select_best_candidate(element, avg_error, dev_error);
30     }
31   }
32   adaptivity->apply_refinements([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e]);
33 }
```

Hermes PDE-independent hp-adaptive algorithm for DG


```
1  while(!done)
2  {
3    ref_spaces = construct_ref_spaces(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
4
5    // Uses the assembling algorithm & linear algebraic routines from TPLs.
     // "solver" is either LinearSolver<double>, or NewtonSolver<double>, or PicardSolver<double>.
6    solver.solve(ref_spaces_const);
7
8    // Here, the flux_limiter stands for an implementation of flux limiter,
     // in the example that follows, limiter according to
     // D. Kuzmin.:  A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods,
     // J. Comput. Appl. Math., 233(12), 3077-3085, 2010. was used.
9    flux_limiter.limit_according_to_detector(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
10   flux_limiter.get_limited_solutions(&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e);
11
12   OGProjection<double>::project_global([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e],
       [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e], [&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e]);
13
14   // Calculate the errors and create list of Elements accordingly.
15   if(adaptivity->calc_error_estimate([&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e],
           [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e]) < threshold) { done = true; break; }
16
17   // Adapt.
18   #pragma omp parallel shared(...) ...
19   {
20   #pragma omp for schedule(dynamic, CHUNKSIZE)
21     for(element = 0; element < element.size(); element++)
22     {
23       if(adaptive_strategy.done) break;
24       refinement_selectors = global_refinement_selectors[thread_id];
25       rslns = rslns[thread_id];
26
27       refinement_selectors->create_candidates(element, rslns, "user settings");
28       refinement_selectors->evaluate_candidates(element, rslns, &avg_error, &dev_error);
29       refinement_selectors->select_best_candidate(element, avg_error, dev_error);
30     }
31   }
32   adaptivity->apply_refinements([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e]);
33 }
```

Hermes PDE-independent hp-adaptive algorithm for DG


```
1  while(!done)
2  {
3    ref_spaces = construct_ref_spaces(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
4
5    // Uses the assembling algorithm & linear algebraic routines from TPLs.
     // "solver" is either LinearSolver<double>, or NewtonSolver<double>, or PicardSolver<double>.
6    solver.solve(ref_spaces_const);
7
8    // Here, the flux_limiter stands for an implementation of flux limiter,
     // in the example that follows, limiter according to
     // D. Kuzmin.:  A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods,
     // J. Comput. Appl. Math., 233(12), 3077-3085, 2010. was used.
9    flux_limiter.limit_according_to_detector(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
10   flux_limiter.get_limited_solutions(&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e);
11
12   OGProjection<double>::project_global([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e],
       [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e], [&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e]);
13
14   // Calculate the errors and create list of Elements accordingly.
15   if(adaptivity->calc_error_estimate([&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e],
         [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e]) < threshold) { done =  true; break; }
16
17   // Adapt.
18   #pragma omp parallel shared(...) ...
19   {
20   #pragma omp for schedule(dynamic, CHUNKSIZE)
21     for(element = 0; element < elements.size(); element++)
22     {
23       if(adaptive_strategy.done) break;
24       refinement_selectors = global_refinement_selectors[thread_id];
25       rslns = rslns[thread_id];
26
27       refinement_selectors->create_candidates(element, rslns, "user settings");
28       refinement_selectors->evaluate_candidates(element, rslns, &avg_error, &dev_error);
29       refinement_selectors->select_best_candidate(element, avg_error, dev_error);
30     }
31   }
32   adaptivity->apply_refinements([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e]);
33 }
```

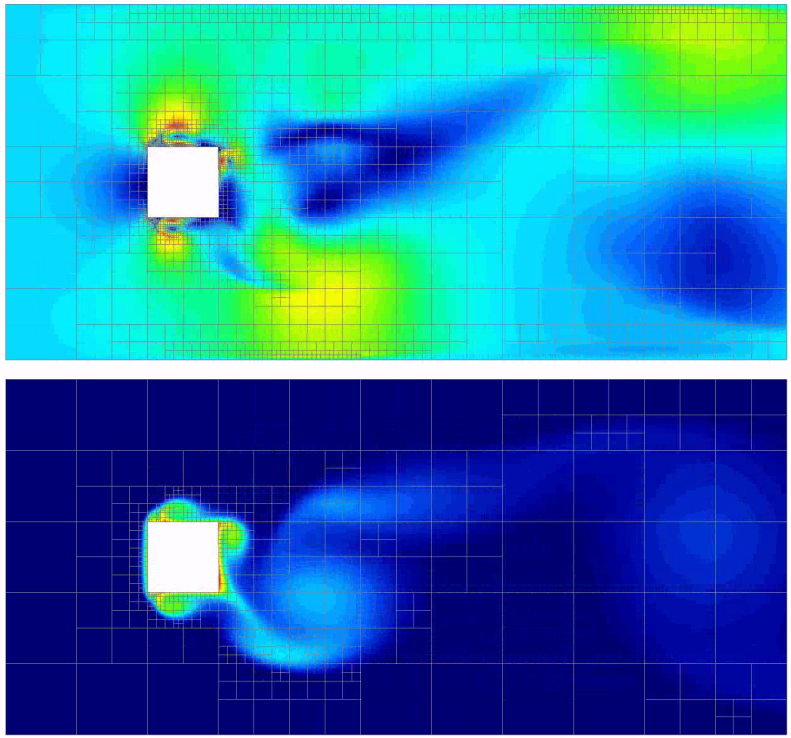
Hermes PDE-independent hp-adaptive algorithm for DG


```
while(!done)
{
  ref_spaces = construct_ref_spaces(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);

  // Uses the assembling algorithm & linear algebraic routines from TPLs.
  // "solver" is either LinearSolver<double>, or NewtonSolver<double>, or PicardSolver<double>.
  solver.solve(ref_spaces_const);

  // Here, the flux_limiter stands for an implementation of flux limiter,
  // in the example that follows, limiter according to
  // D. Kuzmin.:  A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods,
  // J. Comput. Appl. Math., 233(12), 3077-3085, 2010. was used.
  flux_limiter.limit_according_to_detector(&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e);
  flux_limiter.get_limited_solutions(&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e);

  OGProjection<double>::project_global([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e],
    [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e], [&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e]);

  // Calculate the errors and create list of Elements accordingly.
  if(adaptivity->calc_error_estimate([&sln_rho, &sln_rho_v_x, &sln_rho_v_y, &sln_e],
       [&rsln_rho, &rsln_rho_v_x, &rsln_rho_v_y, &rsln_e]) < threshold) { done = true; break; }

  // Adapt.
  #pragma omp parallel shared(...) ...
  {
  #pragma omp for schedule(dynamic, CHUNKSIZE)
    for(element = 0; element < element.size(); element++)
    {
      if(adaptive_strategy.done) break;
      refinement_selectors = global_refinement_selectors[thread_id];
      rslns = rslns[thread_id];

      refinement_selectors->create_candidates(element, rslns, "user settings");
      refinement_selectors->evaluate_candidates(element, rslns, &avg_error, &dev_error);
      refinement_selectors->select_best_candidate(element, avg_error, dev_error);
    }
  }
  adaptivity->apply_refinements([&space_rho, &space_rho_v_x, &space_rho_v_y, &space_e]);
}
```

Discretization of time dependent problems

Rothe's method and dynamical meshes

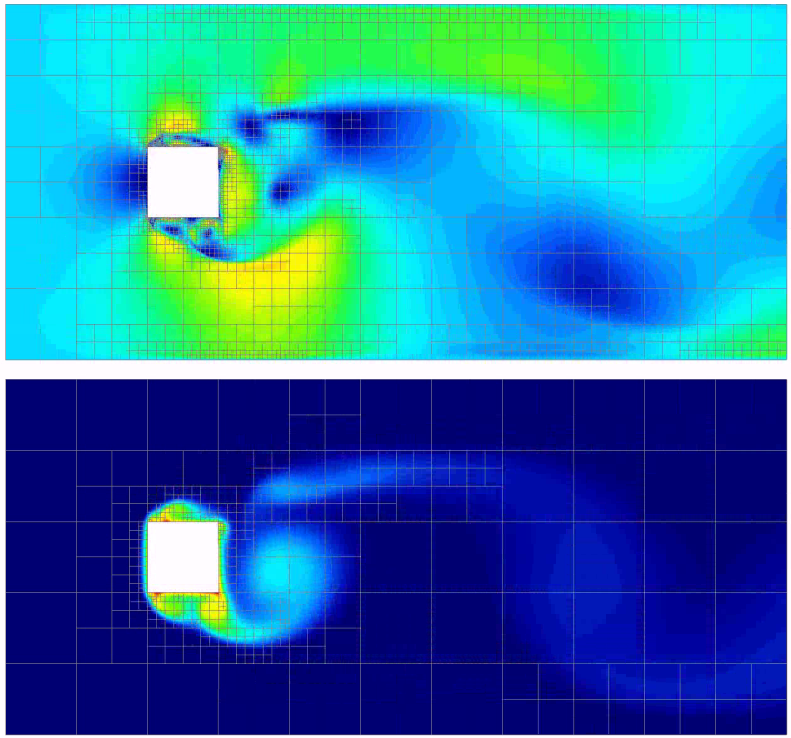
- discretization in time first
- discretized problem at each time level solved by adaptive hp-DG
- meshes evolve in time independently of each other (common original mesh)



Discretization of time dependent problems

Rothe's method and dynamical meshes

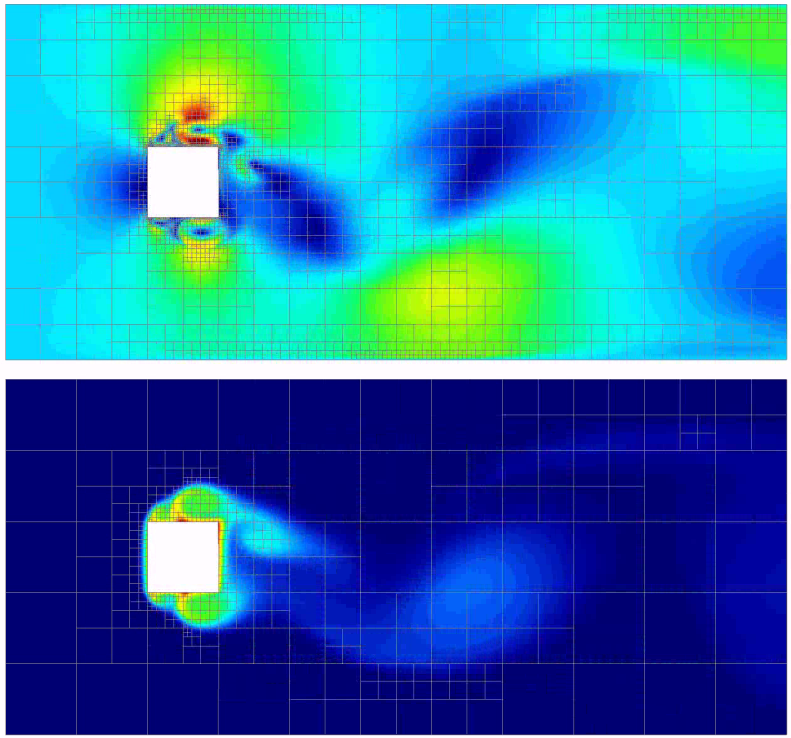
- discretization in time first
- discretized problem at each time level solved by adaptive hp-DG
- meshes evolve in time independently of each other (common original mesh)



Discretization of time dependent problems

Rothe's method and dynamical meshes

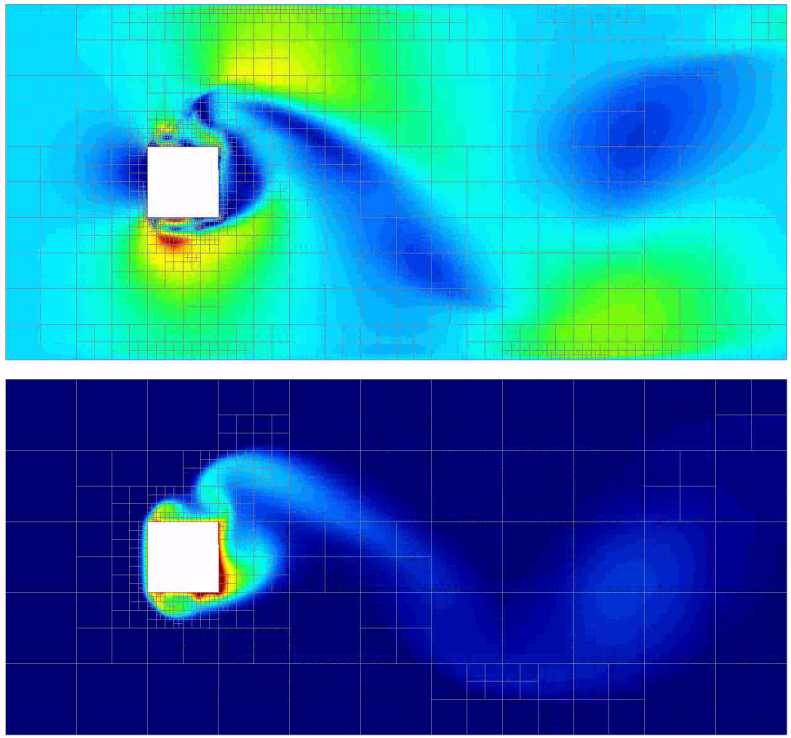
- discretization in time first
- discretized problem at each time level solved by adaptive hp-DG
- meshes evolve in time independently of each other (common original mesh)



Discretization of time dependent problems

Rothe's method and dynamical meshes

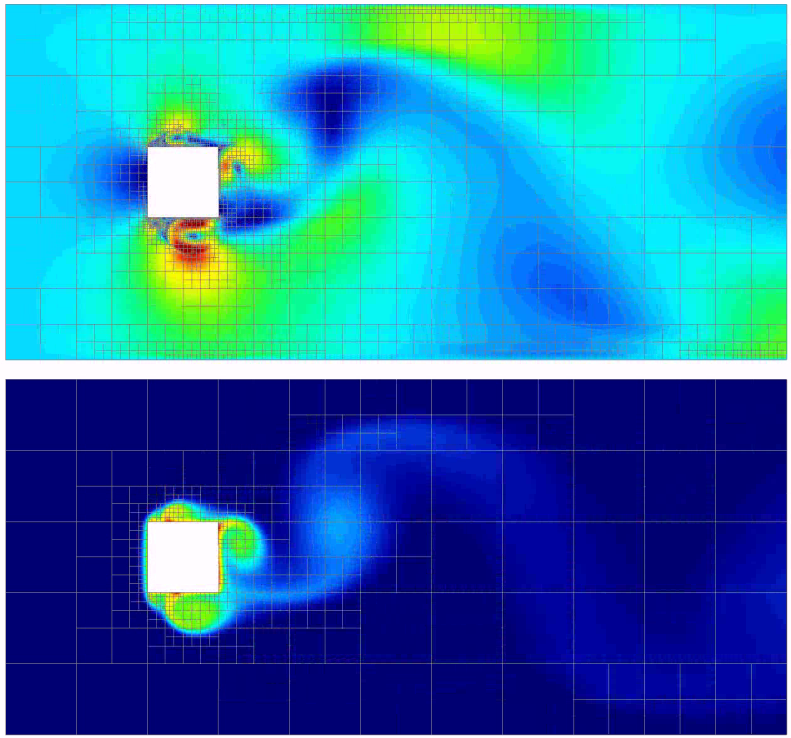
- discretization in time first
- discretized problem at each time level solved by adaptive hp-DG
- meshes evolve in time independently of each other (common original mesh)



Discretization of time dependent problems

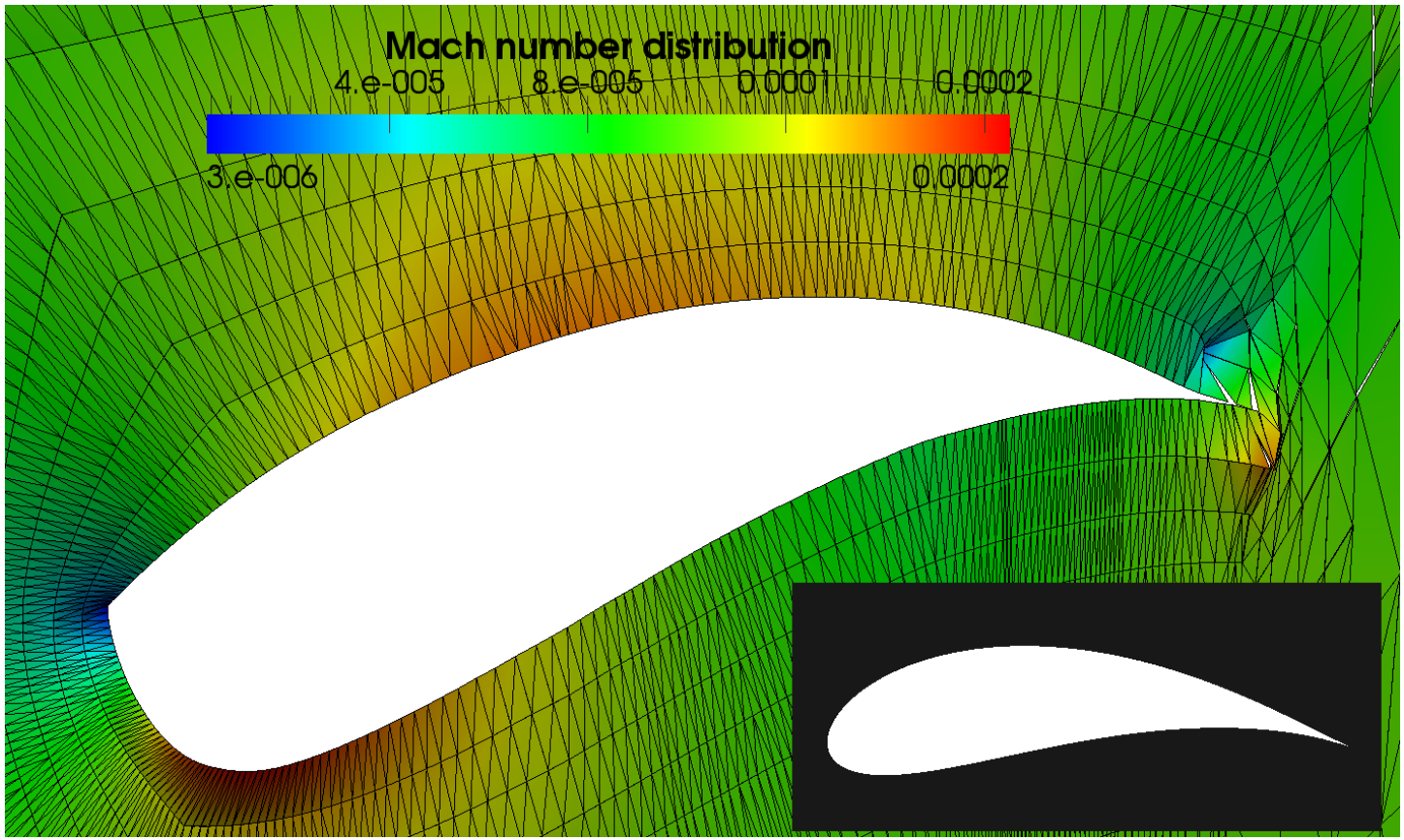
Rothe's method and dynamical meshes

- discretization in time first
- discretized problem at each time level solved by adaptive hp-DG
- meshes evolve in time independently of each other (common original mesh)



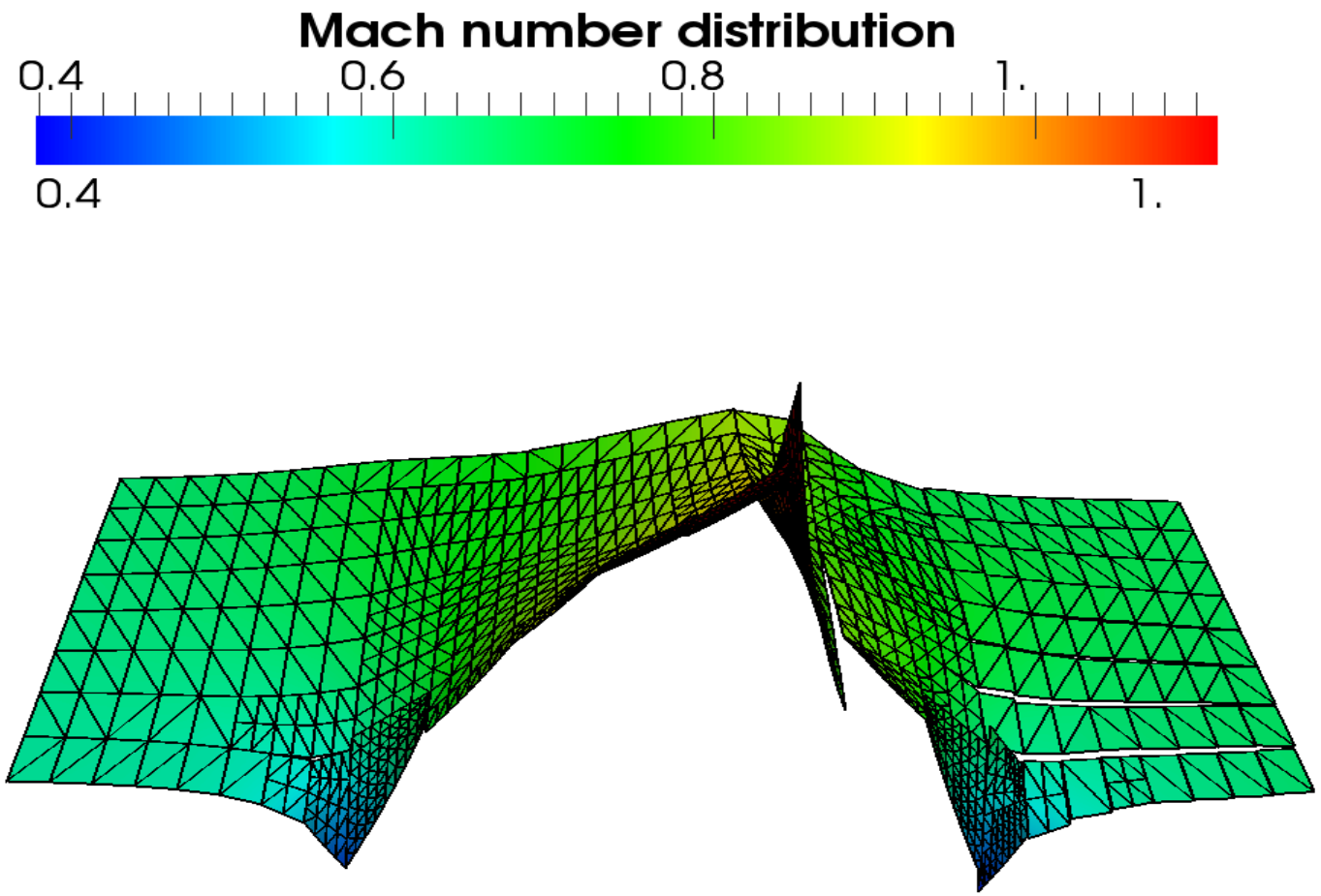
Numerical example - flow around a Joukowski profile

Benchmark: Subsonic flow around airfoil.



Numerical example - channel with a bump

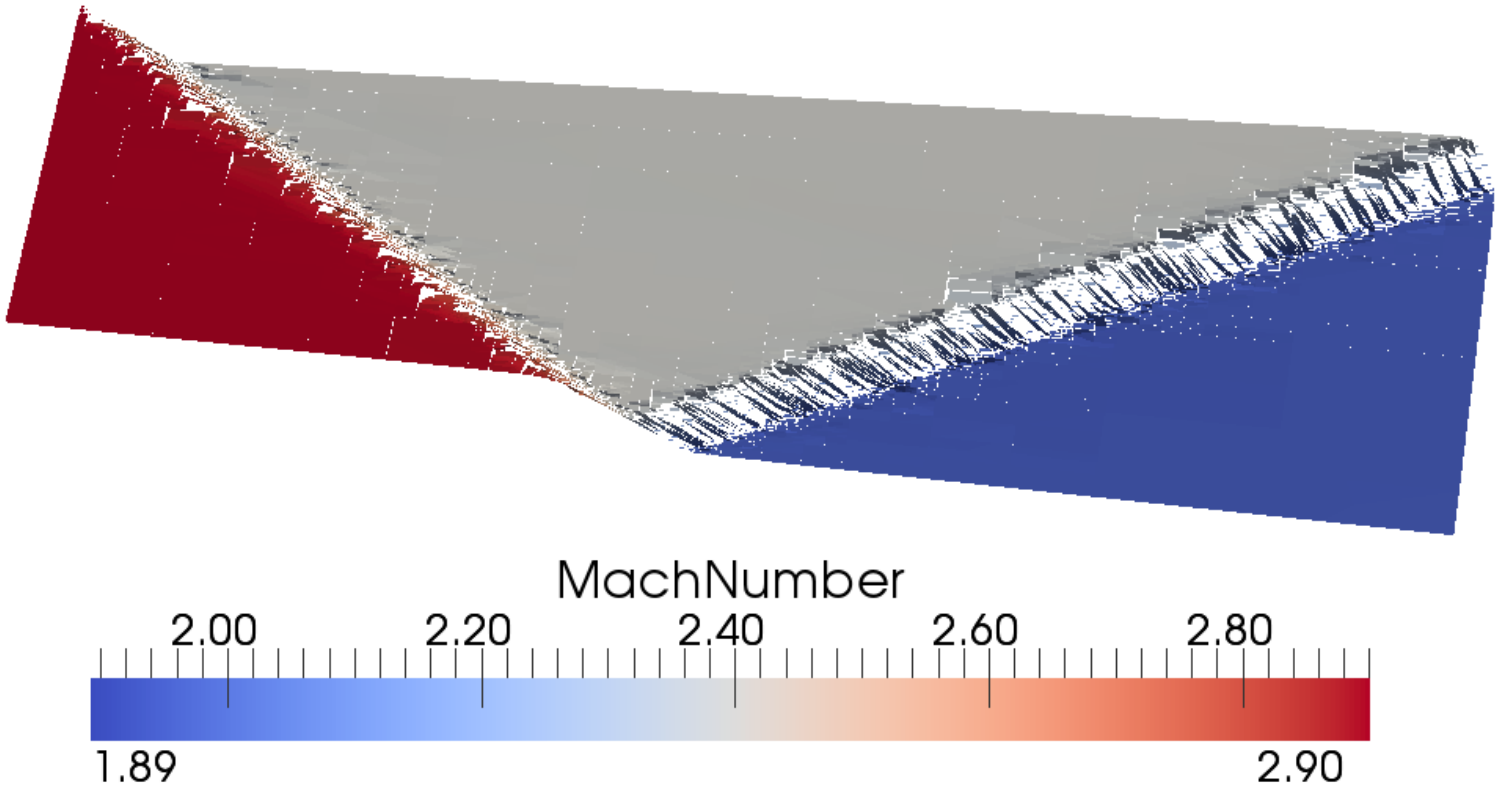
A supersonic flows hits a bump in a channel and produces a shockwave.



Numerical example - reflected shock

Benchmark: Two supersonic flows meeting and reflecting off a wall.

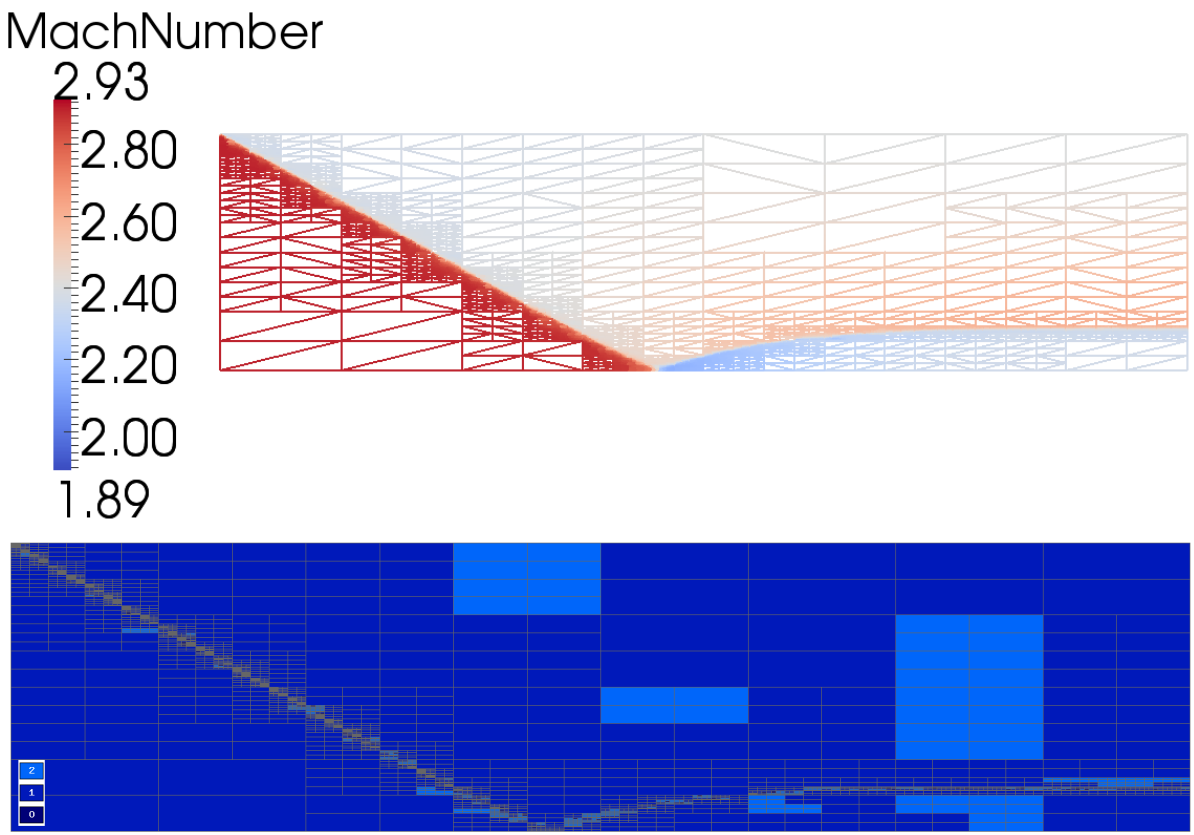
Known exact solution.



Numerical example - reflected shock

Benchmark: Two supersonic flows meeting and reflecting off a wall.

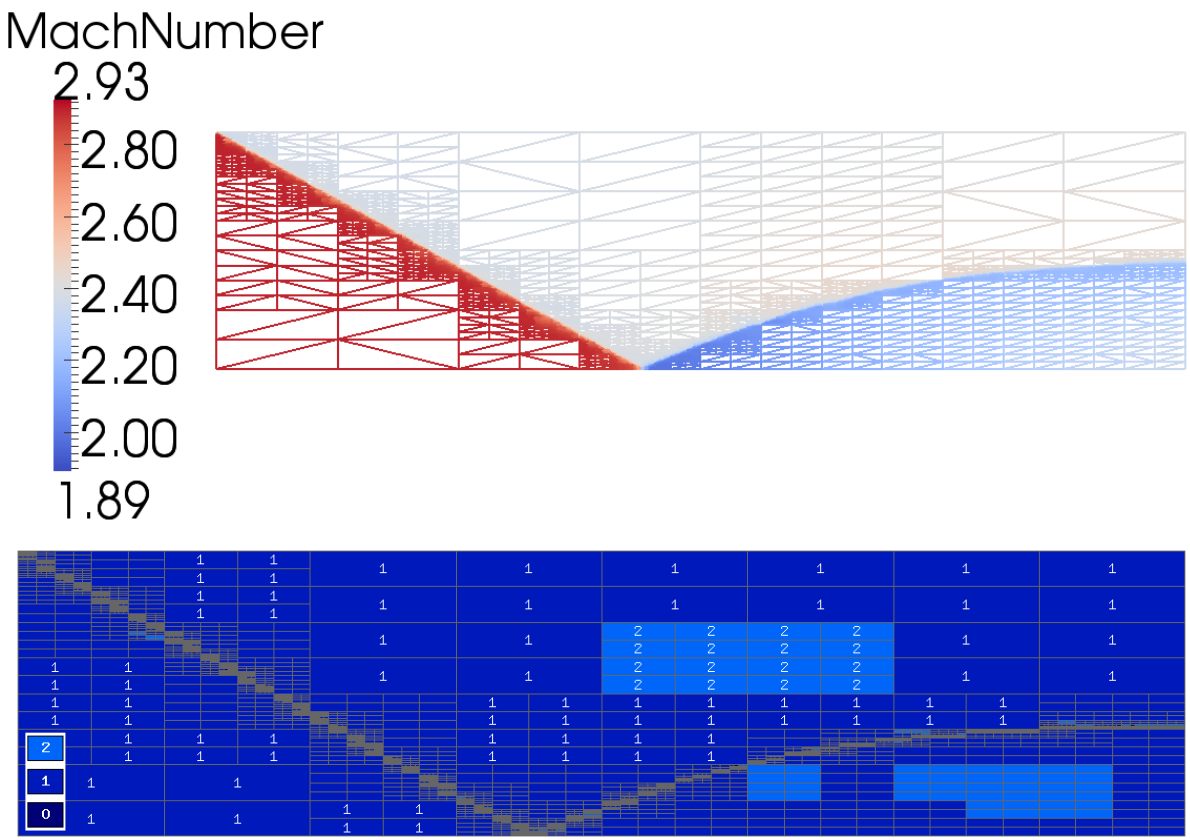
Known exact solution.



Numerical example - reflected shock

Benchmark: Two supersonic flows meeting and reflecting off a wall.

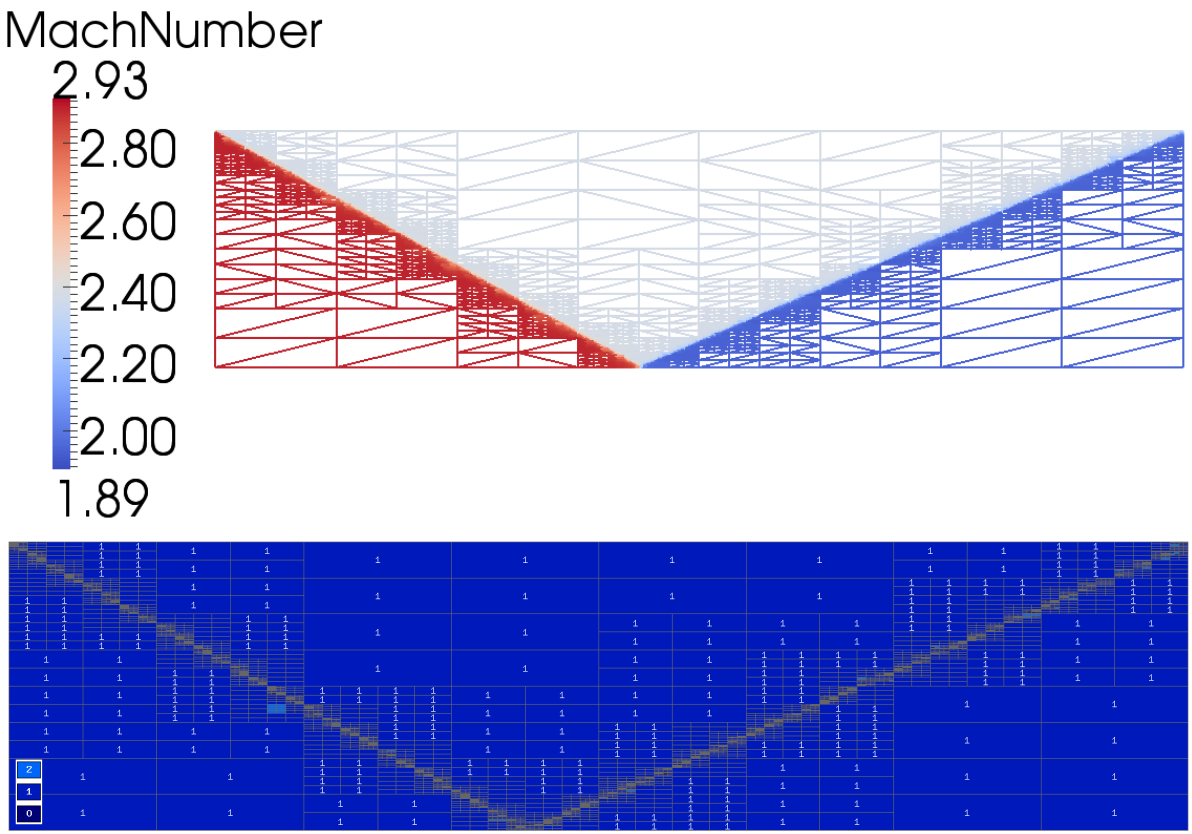
Known exact solution.



Numerical example - reflected shock

Benchmark: Two supersonic flows meeting and reflecting off a wall.

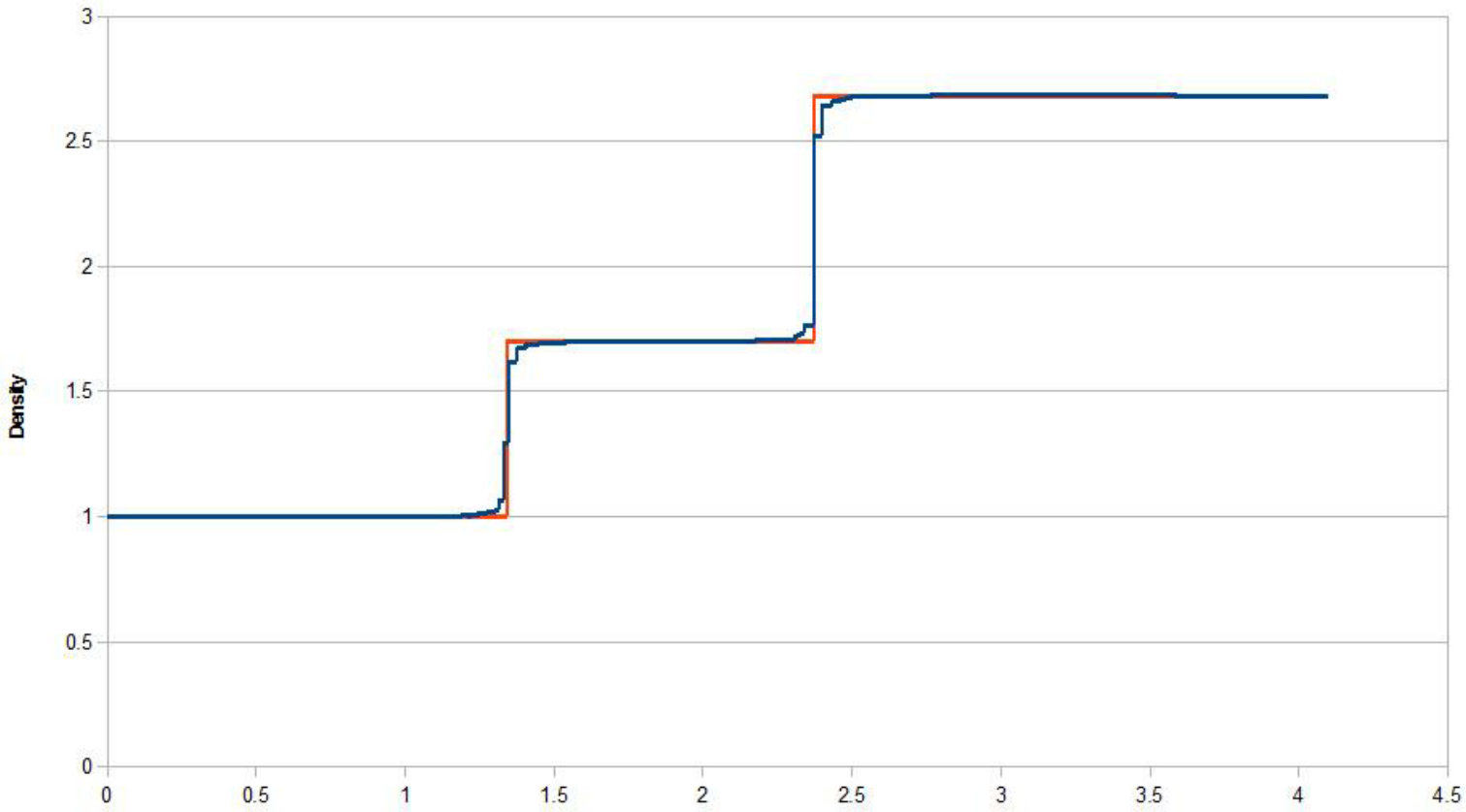
Known exact solution.



Numerical example - reflected shock

Two supersonic flows meeting and reflecting off a wall.

Known exact solution.



Outlook

Outlook:

- coupling with other physical fields (work in progress)
- similar capabilities in 3D
- More complicated geometries
- Domain decomposition, MPI, multi- (many-) node parallelization

The End

Thank you for your kind attention.