Using *hp*-Adaptivity in Diffpack

Xing Cai Klas Samuelsson

January 27, 2004

Abstract

This short note presents the current implementation of the *hp*-adaptivity functionality in Diffpack. First, we explain the mathematical foundation– the discontinuous Galerkin method, followed by the presentation of an *hp*-adaptive grid refinement strategy in association with solving partial differential equations. Then, we give an overview of the new C++ classes that have been implemented to provide Diffpack with the *hp*-version of the finite element method. Finally, we show a coding example of solving an elliptic boundary value problem using *hp*-adaptivity.

1 Introduction

In [2], we have explained how the newly implemented *p*-FEM module extends the standard Diffpack [3] finite element functionality with high-order elements. The new high-order elements include triangular and tetrahedral elements with basis functions being polynomials of degree up to ten. Based on these high-order elements, the follow-up question becomes how to introduce adaptivity also with respect to the order of the elements. More precisely, we wish to allow different types of elements in different regions of a solution domain, so that difficult features of the underlying problem can be more efficiently resolved by high-order elements. In addition, we wish to combine this type of *p*-adaptivity with the existing *h*-adaptivity of Diffpack to achieve even better flexibility. Furthermore, we wish to use the multigrid strategy to solve the discretized equations that arise from a series of *hp*-adaptively refined grids.

2 The Discontinuous Galerkin Method

The key to *p*-adaptivity is to allow neighboring elements to have different types of basis functions, i.e., polynomials of different degrees. On the border between two neighboring elements, grid points from the two elements are allowed to not coincide, or have different values on a same point when approaching from the two sides. A finite element solution defined on such a grid is in general only piecewise continuous. That is, there may arise discontinuities across the borders between neighboring elements. Mathematically, the so-called *discontinuous*

Galerkin method (DGM) suits very well for such situations. So the DGM will be the topic of this section.

Let us denote by Ω the spatial domain on which we wish to solve a partial differential equation (PDE). The boundary of Ω is assumed to consist of two parts: $\partial\Omega = \Gamma_D \cup \Gamma_N$, $\Gamma_D \cap \Gamma_N = \emptyset$, where Dirichlet boundary conditions are valid on Γ_D, and Neumann conditions valid on Γ_N. Suppose we partition Ω into a set of elements, which may choose polynomials of different degrees as the basis functions. Let us denote by $\mathcal{T} = \{T\}$ the set of all the elements in the grid. For the DGM, it is important to also consider the *element edges*, which constitute the boundary of each element. (We remark that an element edge in 3D is in fact an element side.) Let us thus denote by $\mathcal{E}$ the set of all the element edges in a grid. Moreover, $\mathcal{E}$ is considered to be consisting of three disjoint subsets:

$$\mathcal{E} \quad = \quad \mathcal{E}_I \cup \mathcal{E}_D \cup \mathcal{E}_N,$$

where $\mathcal{E}_I$ contains all the interior element edges, i.e., edges between two neighboring elements. The subsets $\mathcal{E}_D$ and $\mathcal{E}_N$ contain the element edges that constitute Γ_D and Γ_N, respectively.

The DGM is based on a principle of preserving some kind of local conservativity for each element, its mathematical formulation heavily involves integrals along the element edges. There exist many versions of the DGM, however, we will only focus on a symmetric variant, see, e.g., [4]. For this purpose, let us consider the following elliptic PDE:

$$-\nabla \cdot (K\nabla u) \quad = \quad f \quad \text{in } \Omega, \tag{1}$$

$$(K\nabla u) \cdot \vec{n} \quad = \quad g_N \quad \text{on } \Gamma_N, \tag{2}$$

$$u \quad = \quad g_D \quad \text{on } \Gamma_D. \tag{3}$$

Suppose that $\mathcal{W}_d^p$ denotes a space of discontinuous piecewise polynomials of degree up to p defined on $\mathcal{T}$. The particular DGM of our interest has the following weak form: Finding $u \in \mathcal{W}_d^p$ such that

$$\sum_{T\in\mathcal{T}} \int_T K\nabla u \cdot \nabla v \, dx - \sum_{E\in\mathcal{E}_I\cup\mathcal{E}_D} \int_E \left(\left\langle \frac{\partial(Ku)}{\partial n} \right\rangle [v] + [u] \left\langle \frac{\partial(Kv)}{\partial n} \right\rangle - \frac{\beta}{h} [u]\,[v] \right) ds$$
$$= \sum_{T\in\mathcal{T}} \int_T f v \, dx + \int_{\Gamma_N} g_N v \, ds + \int_{\Gamma_D} \frac{\beta}{h} g_D v \, ds, \quad \forall v \in \mathcal{W}_d^p. \tag{4}$$

In (4), the symbols $\langle\rangle$ and $[]$ have the following definitions on an element edge E:

$$\langle v \rangle \quad = \quad \begin{cases} (v^+ + v^-)/2 & E \in \mathcal{E}_I, \\ v^+ & E \in \mathcal{E}_D, \end{cases} \tag{5}$$

$$[v] \quad = \quad \begin{cases} v^+ - v^- & E \in \mathcal{E}_I, \\ v^+ & E \in \mathcal{E}_D, \end{cases} \tag{6}$$

where v^+ and v^- represent the limit values of v on E, approaching from the two sides of E.

Remarks. The first term on the left-hand side of (4), together with the first and second terms on the right-hand side, are the same those in the standard continuous Galerkin method. The new terms on the left-hand side, which involve integrals along E, are due to a local conservativity principle and the assumption that discontinuity may arise across E. In particular, the fourth term on the left-hand side of (4) is a penalty term for ensuring stability, where β is a suitable positive parameter. The appearance of g_D in the third term on the right-hand side of (4) is related to only *weakly* enforcing the Dirichlet boundary condition (3), which is a natural consequence of the DG framework. Finally, the parameter h in (4) depends on both the element sizes and edge length, this parameter is needed for obtaining the correct dimensionality of the integral terms involving β. One possible choice of h is as follows (see [1]):

$$h|_E \quad = \quad \begin{cases} \frac{\text{area}(T^+)+\text{area}(T^-)}{3\times\text{length}(E)} & E \in \mathcal{E}_I,\ E = T^+ \cap T^-, \\ \frac{2\times\text{area}(T^+)}{3\times\text{length}(E)} & E \in \mathcal{E}_D. \end{cases} \tag{7}$$

3 An *hp*-Refinement Strategy

We propose the following strategy for carrying out *hp*-adaptive grid refinements associated with solving a PDE on Ω:

1. Create a triangular or tetrahedral starting grid $\mathcal{T}_0$ for discretizing Ω, where all the elements use polynomial of the first degree as the basis functions.

2. The finite element grid $\mathcal{T}_0$ then undergoes a number of *hp*-refinements. During each *hp*-refinement, $\mathcal{T}_i \to \mathcal{T}_{i+1}$, a certain amount of the elements in $\mathcal{T}_i$ is either p-refined or h-refined. That is, a chosen element for refinement either increases the degree of the polynomials as its basis functions (by introducing new points into the element and adopting the definition of high-order elements explained in [2]), or is split into several new elements in $\mathcal{T}_{i+1}$, where these new elements maintain the same order of the basis functions.

3. To determine which elements in $\mathcal{T}_i$ that need to receive either p- or h-refinement, a solution $u(\mathcal{T}_i)$ to the PDE is first found on the finite element grid $\mathcal{T}_i$. Then, we also find

   (a) $u(\mathcal{T}_i^{p+1})$, which corresponds to a solution associated with maintaining the h-resolution of $\mathcal{T}_i$ (i.e., the same number of elements) but increasing the degree of polynomials by one for the basis functions in every element, and

   (b) $u(\mathcal{T}_i^{h/2})$, which corresponds to a solution associated with maintaining the p-resolution throughout $\mathcal{T}_i$ but increasing the h-resolution by a factor of two everywhere.

We remark that the DGM is used to carry out the discretization on $\mathcal{T}_i$, $\mathcal{T}_i^{p+1}$, and $\mathcal{T}_i^{h/2}$. When $u(\mathcal{T}_i)$, $u(\mathcal{T}_i^{p+1})$, and $u(\mathcal{T}_i^{h/2})$ are ready, each element T_j in $\mathcal{T}_i$ computes a p-refinement indicator $I_p(T_j)$ based on the difference between $u(\mathcal{T}_i)$ and $u(\mathcal{T}_i^{p+1})$ inside T_j, and an h-refinement indicator $I_h(T_j)$ based on the difference between $u(\mathcal{T}_i)$ and $u(\mathcal{T}_i^{h/2})$ inside T_j. Whether an element T_j should undergo either p- or h-refinement is based on whether the following condition is fulfilled:

$$I_p(T_j) + I_h(T_j) \ > \ \varrho \, \frac{\sum_j \left(I_p(T_j) + I_h(T_j)\right)}{\#\text{ elements in } \mathcal{T}_i}, \tag{8}$$

where ϱ is a prescribed constant. For an element T_j that satisfies (8), the ratio between the values of $I_p(T_j)$ and $I_h(T_j)$ is used to determine whether p- or h-refinement should be done. More specifically, if

$$I_p(T_j) \ > \ \gamma \, I_h(T_j), \tag{9}$$

then p-refinement is chosen, otherwise h-refinement is chosen. A possible choice of γ in (9) is 0.85 for two-dimensional cases, and $\gamma = 1$ for three-dimensional cases. That is, p-refinement is slightly favored in two-dimensional cases.

4 Overview of the New Classes

The following new classes, together with the new classes introduced in [2], constitute the *hp*-adaptivity module of Diffpack.

- Class `DgFEM` is implemented as a subclass of class `FEM`, such that the different versions of the virtual `FEM::makeSystem` functions are modified to incorporate the weak form (4) of the DGM. Therefore, a user who wishes to use the DGM for solving a PDE must derive his/her simulator class from `DgFEM`. The mandatory programming tasks include re-implementation of two new virtual functions of this new class:

```
virtual void integrands4InteriorEdge ( Mat(NUMT)& elmat,
                                       Vec(NUMT)& elvec,
                                       const FiniteElement& fe1,
                                       const FiniteElement& fe2,
                                       const real h);

virtual void integrands4DirichletEdge ( Mat(NUMT)& elmat,
                                        Vec(NUMT)& elvec,
                                        const FiniteElement& fe,
                                        const real h);
```

  We remark that the `integrands4InteriorEdge` function is used to compute the integral terms along E on the left-hand side of (4), whereas the `integrands4DirichletEdge` function is used to enforce the Dirichlet boundary condition weakly, i.e., computing all the integral terms along $E \in \mathcal{E}_D$

in (4). Section 5 will show a coding example of both functions. Moreover, class `DgFEM` provides two versions of a virtual function `getBeta`, which has a default implementation and can be used to calculate the β parameter needed in (4). The internal calculation of β inside `DgFEM::getBeta` is based on the type of the elements alongside an element edge E. More specifically, we have

```
virtual real getBeta(const FiniteElement& fe1,
                     const FiniteElement& fe2,
                     real beta_scaling = 1.0);

virtual real getBeta(const FiniteElement& fe,
                     real beta_scaling = 1.0);
```

where the first version of the `getBeta` function is to be used inside the `integrands4InteriorEdge` function, for an interior edge $E \in \mathcal{E}_I$ between two elements. The second version is for use inside `integrands4DirichletEdge`, for an edge $E \in \mathcal{E}_D$. We note that both versions of `getBeta` scale the computed default result of β by a factor `beta_scaling`, which can be provided by the user if desired.

- Class `DgUtils` contains a few static functions that are used by class `DgFEM` internally. An important example is the following function:

  ```
  static real getDGElementSize(GridFEAdT& grid,
                               const int e1,
                               const int e2,
                               const int side1);
  ```

  which computes h based on (7). A user seldom needs to use the functions of `DgUtils` directly.

- Class `DiscontinuousGalerkinInfo` contains an internal data structure describing all the interior edges $E \in \mathcal{E}_I$. The functionality of class is frequently used by class `DgFEM` but rarely by a user.

- Class `ProjInterpSparseHP` is implemented as a subclass of `ProjInterpSparse`, therefore belongs to the class hierarchy of `Proj`. The purpose of this new class is to build the projection operator between two *hp*-adaptively refined grids, when a multigrid algorithm is desired. The only thing a user needs to actively do is to choose `ProjInterpSparseHP` as the projection type in an input parameter file for an MG-enabled solver. That is,

  ```
  sub Proj_prm
   set projection type = ProjInterpSparseHP
  ok
  ```

- Class `QuadratureRulesSide` contains functions needed internally by class `DgFEM` and class `DgUtils` for computing the integrals numerically along an edge E, using high-order quadrature rules. A user seldom needs direct access to `QuadratureRulesSide`.

5 An Example of *hp*-Adaptivity

As an example of using *hp*-adaptivity in Diffpack, we consider solving the elliptic boundary value problem (1)-(3) with $g_N = 0$ in (2). The solution domain Ω is allowed to be in both two and three dimensions. Multigrid algorithms are used as the overall solver for a grid hierarchy such as $\mathcal{T}_0, \mathcal{T}_1, \ldots$, whereas the DGM is used for discretizing the PDE at each grid level.

Suppose we name such a simulator by `Poisson1HP`, an outline of the class definition can be as follows:

```
class Poisson1HP : public DgFEM
{
  // main data structure
  Handle(GridFE)     grid;   // finite element grid
  VecSimple(int)     p_orders;     // degree of polynomilas in each element
  Handle(GridFEAdT)  gridlinear;   // where all elements use linear basis func
  Handle(FieldFE)    u;      // finite element field, the primary unknown

  Handle(GridFE)     grid_h2;   // the (h/2, p) grid
  Handle(GridFEAdT)  gridlinear_h2;   // finite element grid using linear elems
  Handle(FieldFE)    u_h2;   // solution on the (h/2, p) grid

  Handle(GridFE)     grid_p1;   // the (h, p+1) grid
  Handle(FieldFE)    u_p1;   // solution on the (h, p+1) grid

  // other help data structure
  // ...

  virtual void integrands  // weak form of the continuous Galerkin method
    (ElmMatVec& elmat, const FiniteElement& fe);

  // integral along interior edges
  virtual void integrands4InteriorEdge (Mat(NUMT)& elmat,
                                        Vec(NUMT)& elvec,
                                        const FiniteElement& fe1,
                                        const FiniteElement& fe2,
                                        const real h);

  // integral along Dirichlet boundaries, weak enforcement of BC
  virtual void integrands4DirichletEdge (Mat(NUMT)& elmat,
                                         Vec(NUMT)& elvec,
                                         const FiniteElement& fe,
                                         const real h);

  virtual void solveProblem ();  // main driver routine

  // other member functions
  // ...
};
```

We remark that class `Poisson1HP` must be derived as a subclass of `DgFEM` such that Diffpack's new functionality of the DGM becomes accessible. The mandatory programming effort includes the implementation of three virtual functions: `integrands`, `integrands4InteriorEdge`, and `integrands4DirichletEdge`. For the `integrands` function, its implementation should be exactly the same as that for the standard continuous Galerkin method, i.e., without considering the integrals along element edges in (4). The implementation of the `integrands4InteriorEdge` function for our elliptic boundary value problem (1)-(3) is as follows:

```
void Poisson1HP:: integrands4InteriorEdge (Mat(NUMT)& elmat,
                                          Vec(NUMT)& elvec,
                                          const FiniteElement& fe1,
                                          const FiniteElement& fe2,
                                          const real h)
{
  int i1,i2,d;
  real k_value_1 = k(fe1);               // find the coefficient K on one side
  real k_value_2 = k(fe2);               // K on the other side
  const int nbf1 = fe1.getNoBasisFunc(); // no of nodes (or basis functions)
  const int nbf2 = fe2.getNoBasisFunc(); // no of nodes (or basis functions)
  const real detSideJxW = fe1.detSideJxW();
  const int nsd = fe1.getNoSpaceDim();
  Ptv(real) normal1(nsd);
  Ptv(real) normal2(nsd);
  fe1.getNormalVectorOnSide(normal1);
  fe2.getNormalVectorOnSide(normal2);

  const real b_h = getBeta(fe1,fe2,dg_beta_scaling)/h;  // beta/h

  for( i1=1; i1<=nbf1; i1++ ) {  // contribution from the side of elm 1
    real n_Du_1 = 0.0;
    for(d=1;d<=nsd;d++) n_Du_1 += normal1(d)*fe1.dN(i1,d);
    n_Du_1 *= k_value_1;
    real u_1 = fe1.N(i1);

    for( i2=1; i2<=nbf1; i2++ ) {
      // ++  basis and test functions both on side 1
      real n_Dv_1 = 0.0;
      for(d=1;d<=nsd;d++) n_Dv_1 += normal1(d)*fe1.dN(i2,d);
      n_Dv_1 *= k_value_1;
      real v_1 = fe1.N(i2);

      elmat(i1,i2)
        += (0.5*( - n_Du_1*v_1 - u_1*n_Dv_1 ) + b_h* u_1*v_1 )*detSideJxW;
    }

    for( i2=1; i2<=nbf2; i2++ ) {
      // +-  basis fcn on side 1 test functions on side 2
      real n_Dv_2 = 0.0;
      for(d=1;d<=nsd;d++) n_Dv_2 += normal2(d)*fe2.dN(i2,d);
      n_Dv_2 *= k_value_2;
      real v_2 = fe2.N(i2);

      elmat(i1,nbf1+i2)
        += (0.5*( + n_Du_1*v_2 + u_1*n_Dv_2 ) - b_h* u_1*v_2 )*detSideJxW;
    }
  }

  for( i1=1; i1<=nbf2; i1++ ) {  // contribution from the side of elm 2
    real n_Du_2 = 0.0;
    for(d=1;d<=nsd;d++) n_Du_2 += normal2(d)*fe2.dN(i1,d);
    n_Du_2 *= k_value_2;
    real u_2 = fe2.N(i1);

    for( i2=1; i2<=nbf1; i2++ ) {
      // -+  basis fcn on side 2 test functions on side 1
      real n_Dv_1 = 0.0;
      for(d=1;d<=nsd;d++) n_Dv_1 += normal1(d)*fe1.dN(i2,d);
      n_Dv_1 *= k_value_1;
      real v_1 = fe1.N(i2);

      elmat(nbf1+i1,i2)
        += (0.5*( + n_Du_2*v_1 + u_2*n_Dv_1 ) - b_h* u_2*v_1 )*detSideJxW;
    }

    for( i2=1;i2<=nbf2;i2++ ) {
      // --  basis fcn on side 2 test functions on side 2
      real n_Dv_2 = 0.0;
      for(d=1;d<=nsd;d++) n_Dv_2 += normal2(d)*fe2.dN(i2,d);
      n_Dv_2 *= k_value_2;
      real v_2 = fe2.N(i2);

      elmat(nbf1+i1,nbf1+i2)
        += (0.5*( - n_Du_2*v_2 - u_2*n_Dv_2 ) + b_h* u_2*v_2 )*detSideJxW;
    }
  }
}
```

We remark that the above **`integrands4InteriorEdge`** function corresponds to the integral terms along $E \in \mathcal{E}_I$ on the left-hand side of (4). Moreover, the size of the matrix **`elmat`** is the summed number of basis functions in the two elements that border along the common edge E. The contributions from the edge integrals are placed in **`elmat`** according to the local numbering of the basis functions in the two elements **`fe1`** and **`fe2`**. The vector **`elvec`** is not used for this case. We also note that the parameter h is already calculated internally inside **`DgFEM`** before invoking this function.

In following, we also show the content of the **`integrands4DirichletEdge`** function that corresponds to the integral terms along $E \in \mathcal{E}_D$ in (4).

```
void Poisson1HP:: integrands4DirichletEdge (Mat(NUMT)& elmat,
                                            Vec(NUMT)& elvec,
                                            const FiniteElement& fe,
                                            const real h)
{
  int i1,i2,d;
  const int nbf = fe.getNoBasisFunc(); // no of nodes (or basis functions)
  const real detSideJxW = fe.detSideJxW();// det J times numerical itg.-weight
  const int nsd = fe.getNoSpaceDim();
  const real b_h = getBeta(fe,dg_beta_scaling)/h;
  real k_value = k (fe);
  Ptv(real) x(nsd);
  fe.getGlobalEvalPt (x);
  real g_value = g (x);
  Ptv(real) normal(nsd);
  fe.getNormalVectorOnSide(normal);

  for( i1=1; i1<=nbf; i1++ ) {
    real n_Du = 0.0;
    for(d=1;d<=nsd;d++) n_Du += normal(d)*fe.dN(i1,d);
    n_Du *= k_value;
    real u = fe.N(i1);

    for( i2=1; i2<=nbf; i2++ ) {
      // ++  basis and test functions both on same side
      real n_Dv = 0.0;
      for(d=1;d<=nsd;d++) n_Dv += normal(d)*fe.dN(i2,d);
      n_Dv *= k_value;
      real v = fe.N(i2);
      elmat(i1,i2)+= ( (- n_Du*v - u*n_Dv) + b_h*u*v )*detSideJxW;
    }
    // weak form of dirichlet condition
    elvec(i1) += g_value*( - n_Du +  b_h*u )*detSideJxW;
  }

}
```

Finally, let us show the main content of the **`solveProblem`** function in the following:

```
void Poisson1HP:: solveProblem ()
{
  const int nsd = gridlinear->getNoSpaceDim();
  int i,j,e;

  // we assume that grid, gridlinear, p_orders, and discont_options are ready
  for (i=1; i<=no_refinements; i++) {
    grid.rebind(new GridFE);
    // make an hp-grid based on a linear grid and 'p_orders'
    GridUtil::makeHigherOrderGrid( grid.getPtr(),
                                   *gridlinear,
                                   p_orders,
                                   discont_options,
                                   force_weak_bc);
    // use multigrid algorithms to solve the PDE
    solveMG( grid, u );

    // create a linear grid which is uniformly refined
    int ref_method = (nsd==2)?3:6;// maximal refinement of each elm
    gridlinear_h2.rebind(gridlinear->refineUniformly(ref_method,true));

    // copy the p-orders and discontinuity options
    const int nel = gridlinear->getNoElms();
    const int nel_h2 = gridlinear_h2->getNoElms();
    VecSimple(int) p_orders_h2(nel_h2);
    VecSimple(int) discont_options_h2(nel_h2);
    for ( e=1; e<=nel; e++ )
      for(j=1; j<=gridlinear->getNoChildren(e); j++) {
        int elm =  gridlinear->getChild(e,j);
        p_orders_h2(elm) = p_orders(e);
        discont_options_h2(elm) = discont_options(e);
     }

    // create the corresponding (h/2, p) grid
    grid_h2.rebind(new GridFE);
    GridUtil::makeHigherOrderGrid( grid_h2.getPtr(),
                                   *gridlinear_h2,
                                   p_orders_h2,
                                   discont_options_h2,
                                   force_weak_bc);
    // use multigrid algorithms to solve the PDE on the (h/2, p) grid
    solveMG( grid_h2, u_h2 );
    // compute the h-indicators from the difference between u and u_h2
    VecSimple(real) indicators_h2;
    DgHpMgTest::computeErrorIndicators(*u,*u_h2,
                                       *gridlinear,*gridlinear_h2,
                                       indicators_h2,this);

    // create the corresponding (h, p+1) grid
    VecSimple(int) p_orders_p1(nel);
    for (e=1;e<=nel;e++)
      p_orders_p1(e) = min(p_orders(e)+1, max_p_order);

    grid_p1.rebind(new GridFE);
    GridUtil::makeHigherOrderGrid( grid_p1.getPtr(),
                                   *gridlinear,
                                   p_orders_p1,
                                   discont_options,
                                   force_weak_bc);
    // use multigrid algorithms to solve the PDE on the (h, p+1) grid
    solveMG( grid_p1, u_p1 );
    // compute the p-indicators from the difference between u and u_p1
    VecSimple(real) indicators_p1;
    DgHpMgTest::computeErrorIndicators(*u,*u_p1,
                                       *gridlinear,
                                       indicators_p1,this);
    // find those elements for h-refinement and store info into 'marked_elms'
    // also adjust the values in 'p_orders' to introduce p-refinement
    VecSimple(int) marked_elms(nel);
    DgHpMgTest::findHPRefinementStrategy(indicators_h2,
                                         indicators_p1,
                                         rho_constant,
                                         gamma_constant,
                                         (nsd==2)?3:6,
                                         max_p_order,
                                         marked_elms,
                                         p_orders,
                                         p_orders_p1);

    // update the linear grid 'gridlinear'
    gridlinear->removeChildGrid ();
    gridlinear.rebind(gridlinear->refine(marked_elms)); // h-refinement
  }
}
```

Remarks

1. The way of creating an *hp*-adaptive finite element grid in Diffpack is to invoke the static function `makeHigherOrderGrid` belonging to the new "repository" class `GridUtil` (see [2]). As a starting point, this functions needs a purely *h*-adaptively refined grid `gridlinear`, where all the elements use polynomials of the first degree as the basis functions. The information about the degree of polynomials in each element in the resulting *hp*-grid is stored in an integer vector named `p_order`, whose length is the same as the number of elements in `gridlinear`. We note that the resulting *hp*-grid has the same number of elements as that in `gridlinear`, but the *hp*-grid will have more grid points than `gridlinear`. Another integer vector named `discont_options` is also needed by the `makeHigherOrderGrid` function. This vector is of the same length as `p_order` and contains for each element an integer, which is used to indicate how "discontinuous" a solution can be when moving across an edge to a neighboring element. Here, we will use integer zero to indicate a total independence between the two values associated with the same point on an element edge when approaching from its two sides, i.e., the standard situation in the DGM. A larger integer as an entry in `discont_options` indicates an improved continuity in the resulting solution. For example, integer one means that two neighboring elements should have the same solution at their shared vertices. Finally, the boolean variable `force_weak_bc` allows the user to choose whether Dirichlet boundary conditions should be enforced weakly.

2. Multigrid algorithms are used to solve the resulting system of linear equations arising from a discretization of the weak form (4) of the DGM. That is, the finest grid is the *hp*-grid resulting from `makeHigherOrderGrid`, whereas the second finest grid is the top-level of the *h*-adaptively refined grid `gridlinear`, and so on. The work of the multigrid algorithms is contained in the member function `solveMG` whose content is roughly as follows:

   ```
   void Poisson1HP:: solveMG( Handle(GridFE)& hpgrid,
   ```

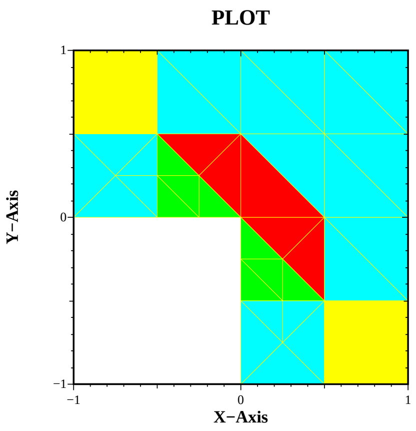
```
                             Handle(FieldFE)& u_)
{
  // extract all the grid levels into a vector
  VecSimplest(Handle(GridFE)) allgrids;
  GridUtil:: createHPGridsForMG( *hpgrid,
                                 allgrids );

  // build a GridCollector object
  const int no_of_spaces = allgrids.size();
  Handle(GridCollector) gridcoll = new GridCollector();
  gridcoll->setNoOfSpaces ( no_of_spaces );
  for (int i=1; i<=no_of_spaces; i++)
    gridcoll->attach(*allgrids(i),i);

  // build a MGtools object
  Handle(MGtools) mgtools = new MGtools(*this));
  mgtools->gridcoll.rebind(*gridcoll);

  // some additional space allocations and redimensioning of objects
  // ...

  // discretization using the DGM at all grid levels
  mgtools->makeSystemMl();

  // multigrid algorithms as the solver
  linsol.fill (0.0);
  dof->insertEssBC (linsol);
  lineq->solve();  // we assume 'MLIter' is chosen as the solver method
  dof->vec2field (linsol, *u_);
}
```

3. The procedure of creating an $(h/2, p)$-grid on the basis of an (h, p)-grid consists of two steps. The first step constructs an $(h/2, 1)$-grid (where all the basis functions are polynomials of the first degree) based on an (h, p)-grid. Then, the second step creates the desired $(h/2, p)$-grid by introducing p-adaptivity into the intermediate $(h/2, 1)$-grid, i.e., by use of the `GridUtil::makeHigherOrderGrid` function.

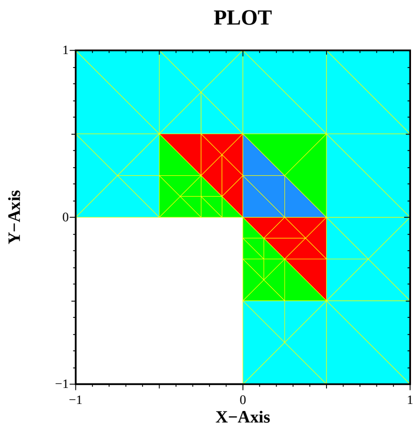
4. An additional help class `DgHpMgTest` has been programmed to simplifying the coding effort inside `Poisson1HP`. Class `DgHpMgTest` contains a set of static functions, where the most important ones are two functions named `computeErrorIndicators` and the `findHPRefinementStrategy` function. These functions can be used to compute the refinement indicators $I_h(T_j)$ and $I_p(T_j)$ needed in (8), and determine whether p-refinement or h-refinement should be used for a chosen element following (9), respectively.

Some Examples of *hp*-Adaptive Grid Refinement

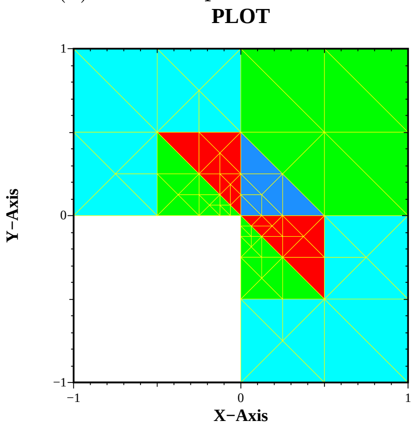
Figure 1 shows a series of hp-adaptively refined grids that are associated with solving a Poisson equation on a two-dimensional L-shaped domain. We note that the solution at the inner corner has a singularity, which explains why the hp-adaptive grid refinements are mostly concentrated around the location.



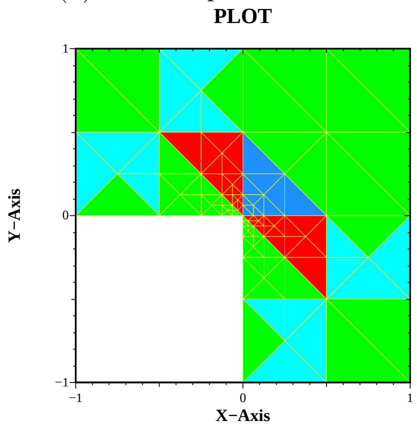
(a) After 2 *hp*-refinements



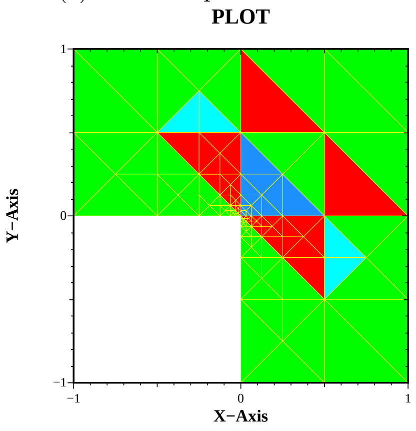
(b) After 3 *hp*-refinements



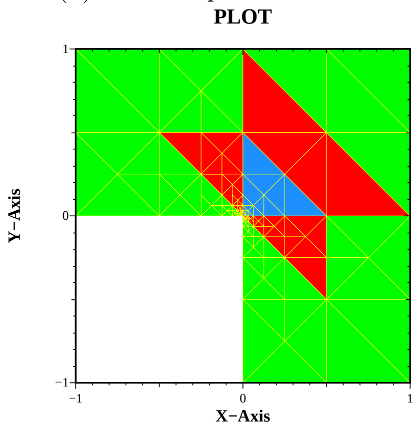
(c) After 4 *hp*-refinements



(d) After 5 *hp*-refinements



(e) After 6 *hp*-refinements



(f) After 7 *hp*-refinements

Figure 1: A series of *hp*-adaptively refined finite element grids. Different colors indicate different orders of the elements.

References

[1] R. Becker, P. Hansbo, and M. G. Larson. Energy norm a posteriori error estimation for discontinuous Galerkin methods. Technical report, Chalmers Finite Element Center, 2001. Preprint 2001-11.

[2] X. Cai and K. Samuelsson. High-order finite elements in Diffpack. Technical report, Simula Research Laboratory, 2003.

[3] H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming.* Textbooks in Computational Science and Engineering. Springer, 2nd edition, 2003.

[4] M. G. Larson and A. J. Niklasson. Conservation properties for the continuous and discontinuous Galerkin methods. Technical report, Chalmers Finite Element Center, 2001. Preprint 2000-08.