我的研究笔记

liyiqiang(lyq105 AT 163.com)

2012 年 10 月 8 日

目录

1 快速多极边界元的四叉树生成算法

快速多极边界元方法借助了自适应四叉树，给出了边界元的加速算法。在使用这四叉树的边界元算法中，使得内存的使用量，和迭代计算的代价得到了改善。

下面给出自适应四叉树的生成算法：

Step 0. 给定树节点中最大单元数 $emax$。

Step 1. 初始化根结点，并设定其所在的层为第 0 层, 并令 $depth = 0$。

Step 2. 设 $k = depth$, 遍历第 $k-1$ 层的非叶子树节点，[1]

- **Case 1.** 如果树节点中的单元数小于等于 $emax$, 标记该树节点为叶子节点。
- **Case 2.** 如果树节点中的单元数大于 $emax$, 标记该树节点为非叶子节点，并调用四分树节点算法，生成下一层的树节点。

Step 3. 若有新的树节点产生， $depth = depth + 1$, 并转 **Step 2**，否则转 **Step 4**。

Step 4. 结束算法。

四分树节点的算法：

1. 读取设定的树节点最大单元数 $emax$;
2. 计算该节点四个象限中的单元数;
3. 遍历四个象限，
   (a) 若该象限中的单元数不为 0，则创建树节点，并将该子节点加入到树节点列表, 若节点中的单元数大于 $emax$, 则将树节点标记为非叶子节点，否则将该树节点标记为叶子节点。
   (b) 若该象限中的单元数为零，转到计算下一个象限。
4. 算法结束。

遍历树算法

[1]在这一步之前并不知道 $k-1$ 层的节点是否是叶子节点。

2 计算边界积分的退化核

若核函数可以表示为 [1]

$$K(x, y) = \sum_{k=1}^{p} \varphi_k(x) \varsigma_k(y)$$

那么如下的矩

$$A_k = \sum_{i=1}^{N} \wedge_i \varphi_k(y_i). \tag{1}$$

可以先计算好，要计算源点处的函数值则只需要做 p 次乘法和 $p-1$ 次加法。

$$u(x) = \sum_{i=1}^{p} A_k \varsigma_k(x). \tag{2}$$

则要是计算在 N 个点处的位势值的话，只需要 $O(N)$ 的计算量。

特别地在边界元中，若核函数可以展开为远场和近场同时适用的级数形式，那么计算则十分的简便。

参考文献

[1] Beatson R, Greengard L. A short course on fast multipole methods [J]. Wavelets, multilevel methods and elliptic PDEs. 1997: 1–37.

3 二维三维 Voronoi 图生成

在多尺度计算中经常会用到一些 Voronoi 图来计算单晶的一些物理过程。下面给出一些 Voronoi 图的生成办法。

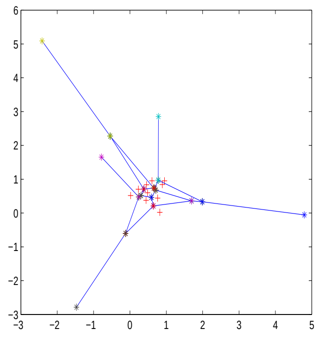
生成 Voronoi 图的软件是 qhull，它可以快速地生成一些点集的凸包，也可以生成一些点集的 Voronoi 图。简单地说 qhull 软件包的功能是输入一些点集，生成的是 Voronoi 图的顶点，以及 cell，也就是 Voronoi 格子。这些格子由一组围成这个格子的顶点的编号来表示。

但是在多尺度计算中经常需要的是一个立方体（3D）或正方性（2D）的单胞，这个单胞与生成的 Voronoi 格子要做一个交，也就说有一个方框将其包住。这个问题貌似很复杂，不好描述。

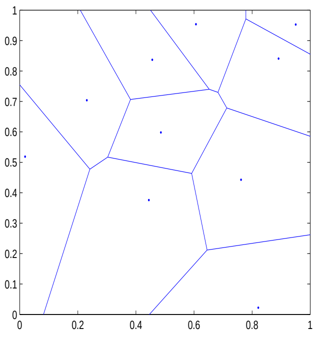
有一个简便的办法，第一步，输入点集，生成 Voronoi 图；第二步，将不包含无限点的 cell 做一次凸包计算，生成 cell 的面，将这些面导入 ansys；第三步，创建一个体，使得这个体的 x, y, z 坐标在 Voronoi 顶点的范围内。第四步，使用 ansys 的 substract 操作，用体减去面，就可以得到想要的 Voronoi 图。

注记 1. 第三步的要求是自然的，单胞结构需要这样构造。

下面给一个二维的图



(a)



(b)

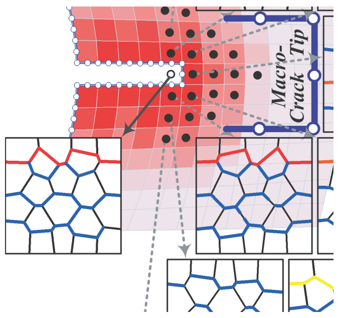


图 1: 计算中的 Voronoi 格子

其中的 b 图是 a 图的局部放大图。可以看出这个才是需要使用的真实计算模型。

3.1 使用凸包生成软件 qhull 生成 Voronoi 图

qhull 包含了一系列的工具，其中 qvoronoi 就是用来生成 Voronoi 图的。

软件的输入是一系列的点集例如

```
2
5
0 0.4
1 0
0 1
1 1
0.5 0.5
```

其中第一行指的是点集的维数，第二行指的是点的数目，以后依次为点的坐标。

输出一般指定为

```
qvoronoi p Fv
```

参数 p 表示的是输出点的坐标， Fv 表示的是输出 voronoi 边，三维的情形表示的是面。

例如上述点集的输出为

```
2
4
0.3666666666666666 -0.1333333333333334
     1    0.5
   0.2    0.7
   0.5      1
8
4 0 2 0 3
4 0 1 0 1
4 0 4 1 3
4 1 3 0 2
4 1 4 1 2
4 2 3 0 4
4 2 4 3 4
4 3 4 2 4
```

其中第一行表示的是点集的维数，第二行表示的是有限点的个数，以后四行表示的是有限点的坐标，接下来的一行表示的是 voronoi 边的个数，接着依次是 voronoi 边它里面的数据的表示的是 2+Voronoi 点数，接下来的两个是输入点编号，并且这两个输入点的中面就是 Voronoi 边（面），其余的数字表示的是 Voronoi 边（面）上的 Voronoi 顶点编号。注意，其中有限点的编号从 1 开始，无限点的编号为 0，也就是说包含编号 0 的 Voronoi 边（面）是开放的。

通常为了处理这样的点，将输出加上 Fi 选项，即

```
qvoronoi p Fv Fi
```

输出的结果为

```
2
4
0.3666666666666666 -0.1333333333333334
     1    0.5
   0.2    0.7
   0.5      1
8
4 0 2 0 3
4 0 1 0 1
4 0 4 1 3
4 1 3 0 2
4 1 4 1 2
4 2 3 0 4
4 2 4 3 4
4 3 4 2 4
4
5 0 4 0.9805806756909201 0.196116135138184 -0.3333974297349128
5 1 4 -0.7071067811865476 0.7071067811865476 0.3535533905932738
5 2 4 0.7071067811865476 -0.7071067811865475 0.3535533905932737
5 3 4 -0.7071067811865476 -0.7071067811865476 1.060660171779821
```

这里的增加了 5 行为了输出无限点，他们分别表示的是，第一行表示的是包含无限点的面数，其余的为面的信息，其中第一个数据表示的是该行的数据个数，接下来的两个数表示的是输入点，其余的三个数分别表示的是两个输入点所夹的面的方程系数。

$$Ax + By + C = 0$$

或者是

$$Ax + By + Cz + D = 0.$$

这样就可以用上面生成的数据就可以完整描述一个 Voronoi 图。

直接读取文件可以使用的 qhull 的命令行为

```
qvoronoi TI test TO file2 p Fv Fi
```

其中 test 是输入的文件名，file2 为输出的文件名，其余为输出参数。

注记 2. 注意到如果要加一个边框的话，只需要计算包含无限点的面和边框的交点即可，二维的情形下是就是包含无限点的直线与边框的交点，三维的情形下，就是两个边框面与无限面之间的交点。

4 使用 libmesh 求解悬臂梁的弯曲问题

悬臂梁问题的控制方程是如下的弹性力学方程组

$$-\partial_j(c_{ijkl}\partial_k u_l) = f_i, \qquad i = 1\ldots d,$$

其中 d 表示维数，为 3， c_{ijkl} 表示刚度系数张量，描述的是材料系数的，在大多数情形下，材料都是各向同性的材料，张量 c_{ijkl} 可以用两个系数 λ 和 μ 来表示， $c_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$ 从而弹性力学方程就可以表示为如下形式

上式的右端相应的双线性形式

$$a(\mathbf{u},\mathbf{v}) = (\lambda\nabla\cdot\mathbf{u},\nabla\cdot\mathbf{v})_\Omega + \sum_{k,l}(\mu\partial_k u_l,\partial_k v_l)_\Omega + \sum_{k,l}(\mu\partial_k u_l,\partial_l v_k)_\Omega,$$

或者写为

$$a(\mathbf{u},\mathbf{v}) = \sum_{k,l}(\lambda\partial_l u_l,\partial_k v_k)_\Omega + \sum_{k,l}(\mu\partial_k u_l,\partial_k v_l)_\Omega + \sum_{k,l}(\mu\partial_k u_l,\partial_l v_k)_\Omega.$$

下面讨论如何定义向量值形函数，对位移的每一个分量进行插值，则

$$\mathbf{u}_h(\mathbf{x}) = \sum_i \Phi_i(\mathbf{x})\, U_i$$

在每一个单元上有， $u_i(x) = \sum\limits_j \phi_j(x)u_i^j, i = 1, 2, \cdots, d$ 其中 u_j^i 表示第 i 个位移的第 j 个插值系数。将双线性变分原理分解为 $a(u,v) = \sum\limits_k a_k(u,v)$ 则有

$$\begin{aligned}
a(\mathbf{u},\mathbf{v}) &= \sum_e\left(\sum_{k,l}(\lambda\partial_l u_l,\partial_k v_k)_{\Omega_e} + \sum_{k,l}(\mu\partial_k u_l,\partial_k v_l)_{\Omega_e} + \sum_{k,l}(\mu\partial_k u_l,\partial_l v_k)_{\Omega_e}\right)\\
&= \sum_e\left(\sum_{k,l}\left(\lambda\partial_l\sum_j\phi_j(x)u_l^j,\partial_k v_k\right)_{\Omega_e} + \sum_{k,l}\left(\mu\partial_k\sum_j\phi_j(x)u_l^j,\partial_k v_l\right)_{\Omega_e} + \sum_{k,l}\left(\mu\partial_k\sum_j\phi_j(x)u_l^j,\partial_l v_k\right)_{\Omega_e}\right)\\
&= \sum_e\sum_{k,l}\sum_j u_l^j(\lambda\partial_l\phi_j(x),\partial_k v_k)_{\Omega_e} + \sum_{k,l}\sum_j u_l^j(\mu\partial_k\phi_j(x),\partial_k v_l)_{\Omega_e} + \sum_{k,l}\sum_j u_l^j(\mu\partial_k\phi_j(x),\partial_l v_k)_{\Omega_e}
\end{aligned}$$

若再取测试函数 v 为 ϕ 定义：

$$\begin{aligned}
&\sum_{i,j}U_iV_j\sum_{k,l}\left\{\left(\lambda\partial_l(\Phi_i)_l,\partial_k(\Phi_j)_k\right)_\Omega + \left(\mu\partial_l(\Phi_i)_k,\partial_l(\Phi_j)_k\right)_\Omega + \left(\mu\partial_l(\Phi_i)_k,\partial_k(\Phi_j)_l\right)_\Omega\right\}\\
&= \sum_j V_j\sum_l\left(f_l,(\Phi_j)_l\right)_\Omega.
\end{aligned}$$

在单元上就要求解如下的矩阵。

$$\begin{aligned}
A_{ij}^K &= \sum_{k,l}\left\{\left(\lambda\partial_l(\Phi_i)_l,\partial_k(\Phi_j)_k\right)_\Omega + \left(\mu\partial_l(\Phi_i)_k,\partial_l(\Phi_j)_k\right)_\Omega + \left(\mu\partial_l(\Phi_i)_k,\partial_k(\Phi_j)_l\right)_\Omega\right\}\\
f_j^K &= \sum_l\left(f_l,(\Phi_j)_l\right)_K = \sum_l\left(f_l,\phi_j\delta_{l,\mathrm{comp}(j)}\right)_K = \left(f_{\mathrm{comp}(j)},\phi_j\right)_K.
\end{aligned}$$

```
/* The Next Great Finite Element Library. */
/* Copyright (C) 2003  Benjamin S. Kirk */

/* This library is free software; you can redistribute it and/or */
/* modify it under the terms of the GNU Lesser General Public */
/* License as published by the Free Software Foundation; either */
/* version 2.1 of the License, or (at your option) any later version. */

/* This library is distributed in the hope that it will be useful, */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU */
/* Lesser General Public License for more details. */

/* You should have received a copy of the GNU Lesser General Public */
/* License along with this library; if not, write to the Free Software */
/* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA */



 // <h1> Systems of Equations 4 - Linear elastic cantilever </h1>
 //      By David Knezevic
 //
 // In this example we model a homogeneous isotropic cantilever
 // using the equations of linear elasticity. We set the Poisson ratio to
 // \nu = 0.3 and clamp the left boundary and apply a vertical load at the
 // right boundary.


// C++ include files that we need
#include <iostream>
#include <algorithm>
#include <math.h>

// libMesh includes
#include "libmesh.h"
#include "mesh.h"
#include "mesh_generation.h"
#include "exodusII_io.h"
#include "gnuplot_io.h"
#include "linear_implicit_system.h"
#include "equation_systems.h"
#include "fe.h"
#include "quadrature_gauss.h"
#include "dof_map.h"
#include "sparse_matrix.h"
#include "numeric_vector.h"
#include "dense_matrix.h"
#include "dense_submatrix.h"
#include "dense_vector.h"
#include "dense_subvector.h"
#include "perf_log.h"
#include "elem.h"
#include "boundary_info.h"
#include "zero_function.h"
#include "dirichlet_boundaries.h"
#include "string_to_enum.h"
#include "getpot.h"

```

```
// Bring in everything from the libMesh namespace
using namespace libMesh;

// Matrix and right-hand side assemble
void assemble_elasticity(EquationSystems& es,
                         const std::string& system_name);

// Define the elasticity tensor, which is a fourth-order tensor
// i.e. it has four indices i,j,k,l
Real eval_elasticity_tensor(unsigned int i,
                            unsigned int j,
                            unsigned int k,
                            unsigned int l);

// Begin the main program.
int main (int argc, char** argv)
{
  // Initialize libMesh and any dependent libaries
  LibMeshInit init (argc, argv);

  // Initialize the cantilever mesh
  const unsigned int dim = 2;

  // Skip this 2D example if libMesh was compiled as 1D-only.
  libmesh_example_assert(dim <= LIBMESH_DIM, ”2D support”);

  Mesh mesh(dim);
  MeshTools::Generation::build_square (mesh,
                                       50, 10,
                                       0., 1.,
                                       0., 0.2,
                                       QUAD9);


  // Print information about the mesh to the screen.
  mesh.print_info();


  // Create an equation systems object.
  EquationSystems equation_systems (mesh);

  // Declare the system and its variables.
  // Create a system named ”Elasticity”
  LinearImplicitSystem& system =
    equation_systems.add_system<LinearImplicitSystem> (”Elasticity”);


  // Add two displacement variables, u and v, to the system
  unsigned int u_var = system.add_variable(”u”, SECOND, LAGRANGE);
  unsigned int v_var = system.add_variable(”v”, SECOND, LAGRANGE);


  system.attach_assemble_function (assemble_elasticity);

  // Construct a Dirichlet boundary condition object
  // We impose a ”clamped” boundary condition on the
  // ”left” boundary, i.e. bc_id = 3
  std::set<boundary_id_type> boundary_ids;
  boundary_ids.insert(3);

  // Create a vector storing the variable numbers which the BC applies to
  std::vector<unsigned int> variables(2);
  variables[0] = u_var; variables[1] = v_var;

  // Create a ZeroFunction to initialize dirichlet_bc
  ZeroFunction<> zf;

  DirichletBoundary dirichlet_bc(boundary_ids,
                                 variables,
                                 &zf);

  // We must add the Dirichlet boundary condition _before_
  // we call equation_systems.init()
  system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);

  // Initialize the data structures for the equation system.
  equation_systems.init();

  // Print information about the system to the screen.
  equation_systems.print_info();

  // Solve the system
  system.solve();

  // Plot the solution
#ifdef LIBMESH_HAVE_EXODUS_API
  ExodusII_IO (mesh).write_equation_systems("displacement.e",equation_systems);
#endif // #ifdef LIBMESH_HAVE_EXODUS_API

  // All done.
  return 0;
}


void assemble_elasticity(EquationSystems& es,
                         const std::string& system_name)
{
  libmesh_assert (system_name == "Elasticity");

  const MeshBase& mesh = es.get_mesh();

  const unsigned int dim = mesh.mesh_dimension();

  LinearImplicitSystem& system = es.get_system<LinearImplicitSystem>("Elasticity");

  const unsigned int u_var = system.variable_number ("u");
  const unsigned int v_var = system.variable_number ("v");

  const DofMap& dof_map = system.get_dof_map();
  FEType fe_type = dof_map.variable_type(0);
  AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));
  QGauss qrule (dim, fe_type.default_quadrature_order());
  fe->attach_quadrature_rule (&qrule);

  AutoPtr<FEBase> fe_face (FEBase::build(dim, fe_type));
  QGauss qface(dim-1, fe_type.default_quadrature_order());
```

```
fe_face->attach_quadrature_rule (&qface);

const std::vector<Real>& JxW = fe->get_JxW();
const std::vector<std::vector<RealGradient> >& dphi = fe->get_dphi();

DenseMatrix<Number> Ke;
DenseVector<Number> Fe;

DenseSubMatrix<Number>
  Kuu(Ke), Kuv(Ke),
  Kvu(Ke), Kvv(Ke);

DenseSubVector<Number>
  Fu(Fe),
  Fv(Fe);

std::vector<unsigned int> dof_indices;
std::vector<unsigned int> dof_indices_u;
std::vector<unsigned int> dof_indices_v;

MeshBase::const_element_iterator       el     = mesh.active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();

for ( ; el != end_el; ++el)
  {
    const Elem* elem = *el;

    dof_map.dof_indices (elem, dof_indices);
    dof_map.dof_indices (elem, dof_indices_u, u_var);
    dof_map.dof_indices (elem, dof_indices_v, v_var);

    const unsigned int n_dofs   = dof_indices.size();
    const unsigned int n_u_dofs = dof_indices_u.size();
    const unsigned int n_v_dofs = dof_indices_v.size();

    fe->reinit (elem);

    Ke.resize (n_dofs, n_dofs);
    Fe.resize (n_dofs);

    Kuu.reposition (u_var*n_u_dofs, u_var*n_u_dofs, n_u_dofs, n_u_dofs);
    Kuv.reposition (u_var*n_u_dofs, v_var*n_u_dofs, n_u_dofs, n_v_dofs);

    Kvu.reposition (v_var*n_v_dofs, u_var*n_v_dofs, n_v_dofs, n_u_dofs);
    Kvv.reposition (v_var*n_v_dofs, v_var*n_v_dofs, n_v_dofs, n_v_dofs);

    Fu.reposition (u_var*n_u_dofs, n_u_dofs);
    Fv.reposition (v_var*n_u_dofs, n_v_dofs);

    for (unsigned int qp=0; qp<qrule.n_points(); qp++)
    {
        for (unsigned int i=0; i<n_u_dofs; i++)
          for (unsigned int j=0; j<n_u_dofs; j++)
          {
            // Tensor indices
            unsigned int C_i, C_j, C_k, C_l;
            C_i=0, C_k=0;

```

```

              C_j=0, C_l=0;
              Kuu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=0;
              Kuu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=0, C_l=1;
              Kuu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=1;
              Kuu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));
            }

          for (unsigned int i=0; i<n_u_dofs; i++)
            for (unsigned int j=0; j<n_v_dofs; j++)
            {
              // Tensor indices
              unsigned int C_i, C_j, C_k, C_l;
              C_i=0, C_k=1;


              C_j=0, C_l=0;
              Kuv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=0;
              Kuv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=0, C_l=1;
              Kuv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=1;
              Kuv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));
            }

          for (unsigned int i=0; i<n_v_dofs; i++)
            for (unsigned int j=0; j<n_u_dofs; j++)
            {
              // Tensor indices
              unsigned int C_i, C_j, C_k, C_l;
              C_i=1, C_k=0;


              C_j=0, C_l=0;
              Kvu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=0;
              Kvu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=0, C_l=1;
              Kvu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=1;
              Kvu(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));
            }

          for (unsigned int i=0; i<n_v_dofs; i++)
            for (unsigned int j=0; j<n_v_dofs; j++)
            {
              // Tensor indices
              unsigned int C_i, C_j, C_k, C_l;
              C_i=1, C_k=1;


              C_j=0, C_l=0;
              Kvv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=0;
              Kvv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=0, C_l=1;
              Kvv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));

              C_j=1, C_l=1;
              Kvv(i,j) += JxW[qp]*(eval_elasticity_tensor(C_i,C_j,C_k,C_l) * dphi[i][qp](C_j)*dphi[j][qp](C_l));
            }
      }

      {
        for (unsigned int side=0; side<elem->n_sides(); side++)
          if (elem->neighbor(side) == NULL)
            {
              boundary_id_type bc_id = mesh.boundary_info->boundary_id (elem,side);
              if (bc_id==BoundaryInfo::invalid_id)
                  libmesh_error();

              const std::vector<std::vector<Real> >&  phi_face = fe_face->get_phi();
              const std::vector<Real>& JxW_face = fe_face->get_JxW();

              fe_face->reinit(elem, side);

              for (unsigned int qp=0; qp<qface.n_points(); qp++)
              {
                if( bc_id == 1 ) // Apply a traction on the right side
                {
                  for (unsigned int i=0; i<n_v_dofs; i++)
                  {
                    Fv(i) += JxW_face[qp]* (-1.) * phi_face[i][qp];
                  }
                }
              }
            }
      }

      dof_map.constrain_element_matrix_and_vector (Ke, Fe, dof_indices);

      system.matrix->add_matrix (Ke, dof_indices);
      system.rhs->add_vector    (Fe, dof_indices);
    }
}

Real eval_elasticity_tensor(unsigned int i,
                            unsigned int j,
                            unsigned int k,
                            unsigned int l)
{
```

```cpp
  // Define the Poisson ratio
  const Real nu = 0.3;

  // Define the Lame constants (lambda_1 and lambda_2) based on Poisson ratio
  const Real lambda_1 = nu / ( (1. + nu) * (1. - 2.*nu) );
  const Real lambda_2 = 0.5 / (1 + nu);

  // Define the Kronecker delta functions that we need here
  Real delta_ij = (i == j) ? 1. : 0.;
  Real delta_il = (i == l) ? 1. : 0.;
  Real delta_ik = (i == k) ? 1. : 0.;
  Real delta_jl = (j == l) ? 1. : 0.;
  Real delta_jk = (j == k) ? 1. : 0.;
  Real delta_kl = (k == l) ? 1. : 0.;

  return lambda_1 * delta_ij * delta_kl + lambda_2 * (delta_ik * delta_jl + delta_il * delta_jk);
}
```

5 libmesh 备忘

使用 triangle 产生一个具有孔洞的三角网格。

```
        Mesh mesh(2);
        mesh.add_point(Point(-1,-1));
        mesh.add_point(Point(1,-1));
        mesh.add_point(Point(1,1));
        mesh.add_point(Point(-1,1));

        TriangleInterface t(mesh);

        // Customize the variables for the triangulation
        t.desired_area()       = .0001;
        t.triangulation_type() = TriangleInterface::PSLG;
        t.smooth_after_generating() = true;
        PolygonHole hole_1(Point(0.,  0.), // center
        0.51,              // radius
        100);               // n. points
        std::vector<Hole*> holes;
        holes.push_back(&hole_1);
        t.attach_hole_list(&holes);
        t.triangulate();
        mesh.prepare_for_use(false);
```

注意最后一行的 prepare_for_use(), 必须要调用才能使用网格。他的原型是

```
void libMesh::MeshBase::prepare_for_use (       const bool      skip_renumber_nodes_and_elements = true )
```

它包含三个步骤

- 1.) call find_neighbors()
- 2.) call partition()
- 3.) call renumber_nodes_and_elements()

```
        FEType fe_type(FIRST, LAGRANGE); // 指定逼近的单元族
  equation_systems.get_system("Poisson").add_variable("u", fe_type); //将逼近与变量结合起来
```

solve 包含两个基本的步骤，一个是调用组装函数，另一个是求解线性方程组。

```
 equation_systems.get_system("Poisson").solve();
```

这样可以直接输出计算结果到 Tecplot

```
        TecplotIO(mesh).write_equation_systems ("squre_tri_res.plt",equation_systems);
```

定义第一类边界条件，下面的代码表示将 3 号边界上的 u 和 v 均设置为 0.

```
 std::set<boundary_id_type> boundary_ids;
          boundary_ids.insert(3);
std::vector<unsigned int> variables(2);
          variables[0] = u_var; variables[1] = v_var;
ZeroFunction<> zf;
DirichletBoundary dirichlet_bc(boundary_ids,
variables,
&zf);
```

```
/*
 *
 *
 *
 *
 *
 *
 * */
/*
#include ”mesh.h”
#include ”mesh_triangle_interface.h”
#include ”mesh_generation.h”
#include ”elem.h”
#include ”mesh_tetgen_interface.h”
#include ”node.h”
#include ”face_tri3.h”
#include ”tecplot_io.h”
#include ”mesh_triangle_holes.h”
#include <math.h>

using namespace std;

typedef TriangleInterface::Hole Hole;
typedef TriangleInterface::PolygonHole PolygonHole;
typedef TriangleInterface::ArbitraryHole ArbitraryHole;

void trianglelate()
{
        Mesh mesh(2);
        mesh.add_point(Point(-1,-2));
        mesh.add_point(Point(2,-2));
        mesh.add_point(Point(2,2));
        mesh.add_point(Point(-2,2));

 TriangleInterface t(mesh);

  // Customize the variables for the triangulation
  t.desired_area()       = .01;
  t.triangulation_type() = TriangleInterface::PSLG;
  t.smooth_after_generating() = true;
        PolygonHole hole_1(Point(0.,  0.), // center
                        1,               // radius
                        100);                // n. points
        std::vector<Hole*> holes;
        holes.push_back(&hole_1);
        t.attach_hole_list(&holes);
        t.triangulate();
        TecplotIO(mesh).write(”squre_hole.plt”);
}

int triangle_circle (int argc, char** argv)

{
        LibMeshInit init (argc, argv);

  libmesh_example_assert(2 <= LIBMESH_DIM, ”2D support”);

        Mesh mesh(2);
```

```
                int n_outer_circle_points = 100;
        double outer_circle_ridus = 2;

        for(int i = 0; i< n_outer_circle_points; ++i)
        {
                double x = outer_circle_ridus * cos( i* 2 * libMesh::pi / n_outer_circle_points );
                double y = outer_circle_ridus * sin( i* 2 * libMesh::pi / n_outer_circle_points );
                mesh.add_point(Point (x , y));
        }

 TriangleInterface t(mesh);

  // Customize the variables for the triangulation
  t.desired_area()       = .0001;

  // A Planar Straight Line Graph (PSLG) is essentially a list
  // of segments which have to exist in the final triangulation.
  // For an L-shaped domain, Triangle will compute the convex
  // hull of boundary points if we do not specify the PSLG.
  // The PSLG algorithm is also required for triangulating domains
  // containing holes
  t.triangulation_type() = TriangleInterface::PSLG;

  // Turn on/off Laplacian mesh smoothing after generation.
  // By default this is on.
  t.smooth_after_generating() = true;

        PolygonHole hole_1(Point(0.,  0.), // center
                        1,              // radius
                        100);               // n. points
        std::vector<Hole*> holes;
        holes.push_back(&hole_1);
        t.attach_hole_list(&holes);

        t.triangulate();
        TecplotIO(mesh).write(”anulus.plt”);
        trianglelate(mesh2d);
        return 0;
}
*/

/* The Next Great Finite Element Library. */
/* Copyright (C) 2003  Benjamin S. Kirk */

/* This library is free software; you can redistribute it and/or */
/* modify it under the terms of the GNU Lesser General Public */
/* License as published by the Free Software Foundation; either */
/* version 2.1 of the License, or (at your option) any later version. */

/* This library is distributed in the hope that it will be useful, */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU */
/* Lesser General Public License for more details. */

/* You should have received a copy of the GNU Lesser General Public */
/* License along with this library; if not, write to the Free Software */
/* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA */

```

```

 // <h1>Example 3 - Solving a Poisson Problem</h1>
 //
 // This is the third example program.  It builds on
 // the second example program by showing how to solve a simple
 // Poisson system.  This example also introduces the notion
 // of customized matrix assembly functions, working with an
 // exact solution, and using element iterators.
 // We will not comment on things that
 // were already explained in the second example.

// C++ include files that we need
#include <iostream>
#include <algorithm>
#include <math.h>

// Basic include files needed for the mesh functionality.
#include ”libmesh.h”
#include ”mesh.h”
#include ”mesh_generation.h”
#include ”vtk_io.h”
#include ”linear_implicit_system.h”
#include ”equation_systems.h”
#include ”mesh_tetgen_interface.h”
#include ”tecplot_io.h”
#include ”mesh_triangle_interface.h”
#include ”mesh_triangle_holes.h”

// Define the Finite Element object.
#include ”fe.h”

// Define Gauss quadrature rules.
#include ”quadrature_gauss.h”

// Define useful datatypes for finite element
// matrix and vector components.
#include ”sparse_matrix.h”
#include ”numeric_vector.h”
#include ”dense_matrix.h”
#include ”dense_vector.h”
#include ”elem.h”

// Define the DofMap, which handles degree of freedom
// indexing.
#include ”dof_map.h”

typedef TriangleInterface::Hole Hole;
typedef TriangleInterface::PolygonHole PolygonHole;
typedef TriangleInterface::ArbitraryHole ArbitraryHole;
// Bring in everything from the libMesh namespace
using namespace libMesh;

// Function prototype.  This is the function that will assemble
// the linear system for our Poisson problem.  Note that the
// function will take the  EquationSystems object and the
// name of the system we are assembling as input.  From the
//  EquationSystems object we have access to the  Mesh and
// other objects we might need.
```

```
void assemble_poisson(EquationSystems& es,
                      const std::string& system_name);

// Function prototype for the exact solution.
Real exact_solution (const Real x,
                     const Real y,
                     const Real z = 0.);

int main (int argc, char** argv)
{
  // Initialize libraries, like in example 2.
  LibMeshInit init (argc, argv);

  // Brief message to the user regarding the program name
  // and command line arguments.
  std::cout << "Running " << argv[0];

  for (int i=1; i<argc; i++)
    std::cout << " " << argv[i];

  std::cout << std::endl << std::endl;

  // Skip this 2D example if libMesh was compiled as 1D-only.
  libmesh_example_assert(2 <= LIBMESH_DIM, "2D support");

        Mesh mesh(2);
        mesh.add_point(Point(-1,-1));
        mesh.add_point(Point(1,-1));
        mesh.add_point(Point(1,1));
        mesh.add_point(Point(-1,1));

 TriangleInterface t(mesh);

  // Customize the variables for the triangulation
  t.desired_area()       = .0001;
  t.triangulation_type() = TriangleInterface::PSLG;
  t.smooth_after_generating() = true;
        PolygonHole hole_1(Point(0.,  0.), // center
                        0.51,              // radius
                        100);               // n. points
        std::vector<Hole*> holes;
        holes.push_back(&hole_1);
        t.attach_hole_list(&holes);
        t.triangulate();
  mesh.find_neighbors();
        mesh.prepare_for_use();
        std::cout << "write mesh file done!!\n" << std::endl;
        /*mesh.clear();*/
        TecplotIO(mesh).write("squre_tri.plt");
  /*// Use the MeshTools::Generation mesh generator to create a uniform*/
  /*// 2D grid on the square [-1,1]^2.  We instruct the mesh generator*/
  // to build a mesh of 15x15 QUAD9 elements.  Building QUAD9
  // elements instead of the default QUAD4's we used in example 2
  // allow us to use higher-order approximation.
        //
        // MeshTools::Generation::build_square (mesh,
        //
15, 15,
        //
-1., 1.,
        //
-1., 1.,
        //
TRI3);

        /*TecplotIO(mesh).write("squre_tri_2.plt");*/
  /*// Print information about the mesh to the screen.*/
  /*// Note that 5x5 QUAD9 elements actually has 11x11 nodes,*/
  /*// so this mesh is significantly larger than the one in example 2.*/
        mesh.print_info();
        /*return 0;*/

  // Create an equation systems object.
  EquationSystems equation_systems (mesh);

  // Declare the Poisson system and its variables.
  // The Poisson system is another example of a steady system.
  equation_systems.add_system<LinearImplicitSystem> ("Poisson");

  // Adds the variable "u" to "Poisson".  "u"
  // will be approximated using second-order approximation.
        FEType fe_type(FIRST, LAGRANGE);

  equation_systems.get_system("Poisson").add_variable("u", fe_type);

  // Give the system a pointer to the matrix assembly
  // function.  This will be called when needed by the
  // library.

  equation_systems.get_system("Poisson").attach_assemble_function (assemble_poisson);

  // Initialize the data structures for the equation system.
  equation_systems.init();
        /*LinearImplicitSystem& system = equation_systems.get_system<LinearImplicitSystem>("Poisson");*/

  // A reference to the  DofMap object for this system.  The  DofMap
  // object handles the index translation from node and element numbers
  // to degree of freedom numbers.  We will talk more about the  DofMap
  // in future examples.
  /*const DofMap& dof_map = system.get_dof_map();*/

        /*dof_map.print_info();*/
  // Prints information about the system to the screen.
  equation_systems.print_info();

  // Solve the system "Poisson".  Note that calling this
  // member will assemble the linear system and invoke
  // the default numerical solver.  With PETSc the solver can be
  // controlled from the command line.  For example,
  // you can invoke conjugate gradient with:
  //
  // ./ex3 -ksp_type cg
  //
  // You can also get a nice X-window that monitors the solver
  // convergence with:
  //
  // ./ex3 -ksp_xmonitor
  //
  // if you linked against the appropriate X libraries when you
  // built PETSc.
  equation_systems.get_system("Poisson").solve();

#if defined(LIBMESH_HAVE_VTK) && !defined(LIBMESH_ENABLE_PARMESH)

  // After solving the system write the solution
  // to a VTK-formatted plot file.
  VTKIO (mesh).write_equation_systems ("out.pvtu", equation_systems);

#endif // #ifdef LIBMESH_HAVE_VTK

        TecplotIO(mesh).write_equation_systems ("squre_tri_res.plt",equation_systems);

  // All done.
  return 0;
}



// We now define the matrix assembly function for the
// Poisson system.  We need to first compute element
// matrices and right-hand sides, and then take into
// account the boundary conditions, which will be handled
// via a penalty method.
void assemble_poisson(EquationSystems& es,
                      const std::string& system_name)
{

  // It is a good idea to make sure we are assembling
  // the proper system.
  libmesh_assert (system_name == "Poisson");


  // Get a constant reference to the mesh object.
  const MeshBase& mesh = es.get_mesh();

  // The dimension that we are running
  const unsigned int dim = mesh.mesh_dimension();

  // Get a reference to the LinearImplicitSystem we are solving
  LinearImplicitSystem& system = es.get_system<LinearImplicitSystem> ("Poisson");

  // A reference to the  DofMap object for this system.  The  DofMap
  // object handles the index translation from node and element numbers
  // to degree of freedom numbers.  We will talk more about the  DofMap
  // in future examples.
  const DofMap& dof_map = system.get_dof_map();

  // Get a constant reference to the Finite Element type
  // for the first (and only) variable in the system.
  FEType fe_type = dof_map.variable_type(0);

  // Build a Finite Element object of the specified type.  Since the
  // FEBase::build() member dynamically creates memory we will
  // store the object as an AutoPtr<FEBase>.  This can be thought
```

```
    // of as a pointer that will clean up after itself.  Example 4
    // describes some advantages of  AutoPtr's in the context of
    // quadrature rules.
    AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));

    // A 5th order Gauss quadrature rule for numerical integration.
    QGauss qrule (dim, FIFTH);

    // Tell the finite element object to use our quadrature rule.
    fe->attach_quadrature_rule (&qrule);

    // Declare a special finite element object for
    // boundary integration.
    AutoPtr<FEBase> fe_face (FEBase::build(dim, fe_type));

    // Boundary integration requires one quadraure rule,
    // with dimensionality one less than the dimensionality
    // of the element.
    QGauss qface(dim-1, FIFTH);

    // Tell the finite element object to use our
    // quadrature rule.
    fe_face->attach_quadrature_rule (&qface);

    // Here we define some references to cell-specific data that
    // will be used to assemble the linear system.
    //
    // The element Jacobian * quadrature weight at each integration point.
    const std::vector<Real>& JxW = fe->get_JxW();

    // The physical XY locations of the quadrature points on the element.
    // These might be useful for evaluating spatially varying material
    // properties at the quadrature points.
    const std::vector<Point>& q_point = fe->get_xyz();

    // The element shape functions evaluated at the quadrature points.
    const std::vector<std::vector<Real> >& phi = fe->get_phi();

    // The element shape function gradients evaluated at the quadrature
    // points.
    const std::vector<std::vector<RealGradient> >& dphi = fe->get_dphi();

    // Define data structures to contain the element matrix
    // and right-hand-side vector contribution.  Following
    // basic finite element terminology we will denote these
    // "Ke" and "Fe".  These datatypes are templated on
    //  Number, which allows the same code to work for real
    // or complex numbers.
    DenseMatrix<Number> Ke;
    DenseVector<Number> Fe;


    // This vector will hold the degree of freedom indices for
    // the element.  These define where in the global system
    // the element degrees of freedom get mapped.
    std::vector<unsigned int> dof_indices;

    // Now we will loop over all the elements in the mesh.
```

```
// We will compute the element matrix and right-hand-side
// contribution.
//
// Element iterators are a nice way to iterate through all the
// elements, or all the elements that have some property.  The
// iterator el will iterate from the first to the last element on
// the local processor.  The iterator end_el tells us when to stop.
// It is smart to make this one const so that we don’t accidentally
// mess it up!  In case users later modify this program to include
// refinement, we will be safe and will only consider the active
// elements; hence we use a variant of the \p active_elem_iterator.
MeshBase::const_element_iterator       el     = mesh.active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();

// Loop over the elements.  Note that  ++el is preferred to
// el++ since the latter requires an unnecessary temporary
// object.
for ( ; el != end_el ; ++el)
  {
    // Store a pointer to the element we are currently
    // working on.  This allows for nicer syntax later.
    const Elem* elem = *el;

    // Get the degree of freedom indices for the
    // current element.  These define where in the global
    // matrix and right-hand-side this element will
    // contribute to.
    dof_map.dof_indices (elem, dof_indices);

    // Compute the element-specific data for the current
    // element.  This involves computing the location of the
    // quadrature points (q_point) and the shape functions
    // (phi, dphi) for the current element.
    fe->reinit (elem);


    // Zero the element matrix and right-hand side before
    // summing them.  We use the resize member here because
    // the number of degrees of freedom might have changed from
    // the last element.  Note that this will be the case if the
    // element type is different (i.e. the last element was a
    // triangle, now we are on a quadrilateral).

    // The  DenseMatrix::resize() and the  DenseVector::resize()
    // members will automatically zero out the matrix  and vector.
    Ke.resize (dof_indices.size(),
               dof_indices.size());

    Fe.resize (dof_indices.size());

    // Now loop over the quadrature points.  This handles
    // the numeric integration.
    for (unsigned int qp=0; qp<qrule.n_points(); qp++)
      {

        // Now we will build the element matrix.  This involves
        // a double loop to integrate the test funcions (i) against
        // the trial functions (j).
```

```
          for (unsigned int i=0; i<phi.size(); i++)
            for (unsigned int j=0; j<phi.size(); j++)
              {
                Ke(i,j) += JxW[qp]*(dphi[i][qp]*dphi[j][qp]);
              }

          // This is the end of the matrix summation loop
          // Now we build the element right-hand-side contribution.
          // This involves a single loop in which we integrate the
          // "forcing function" in the PDE against the test functions.
          {
            const Real x = q_point[qp](0);
            const Real y = q_point[qp](1);
            const Real eps = 1.e-3;


            // "fxy" is the forcing function for the Poisson equation.
            // In this case we set fxy to be a finite difference
            // Laplacian approximation to the (known) exact solution.
            //
            // We will use the second-order accurate FD Laplacian
            // approximation, which in 2D is
            //
            // u_xx + u_yy = (u(i,j-1) + u(i,j+1) +
            //                u(i-1,j) + u(i+1,j) +
            //                -4*u(i,j))/h^2
            //
            // Since the value of the forcing function depends only
            // on the location of the quadrature point (q_point[qp])
            // we will compute it here, outside of the i-loop
            const Real fxy = -(exact_solution(x,y-eps) +
                               exact_solution(x,y+eps) +
                               exact_solution(x-eps,y) +
                               exact_solution(x+eps,y) -
                               4.*exact_solution(x,y))/eps/eps;

                                                /*std::cout << fxy << std::endl;        */
            for (unsigned int i=0; i<phi.size(); i++)
              Fe(i) += JxW[qp]*fxy*phi[i][qp];
          }
        }



      // We have now reached the end of the RHS summation,
      // and the end of quadrature point loop, so
      // the interior element integration has
      // been completed.  However, we have not yet addressed
      // boundary conditions.  For this example we will only
      // consider simple Dirichlet boundary conditions.
      //
      // There are several ways Dirichlet boundary conditions
      // can be imposed.  A simple approach, which works for
      // interpolary bases like the standard Lagrange polynomials,
      // is to assign function values to the
      // degrees of freedom living on the domain boundary. This
      // works well for interpolary bases, but is more difficult
      // when non-interpolary (e.g Legendre or Hierarchic) bases
      // are used.
      //
      // Dirichlet boundary conditions can also be imposed with a
      // ”penalty” method.  In this case essentially the L2 projection
      // of the boundary values are added to the matrix. The
      // projection is multiplied by some large factor so that, in
      // floating point arithmetic, the existing (smaller) entries
      // in the matrix and right-hand-side are effectively ignored.
      //
      // This amounts to adding a term of the form (in latex notation)
      //
      // \frac{1}{\epsilon} \int_{\delta \Omega} \phi_i \phi_j = \frac{1}{\epsilon} \int_{\delta \Omega} u \phi_i
      //
      // where
      //
      // \frac{1}{\epsilon} is the penalty parameter, defined such that \epsilon << 1
      {

        // The following loop is over the sides of the element.
        // If the element has no neighbor on a side then that
        // side MUST live on a boundary of the domain.
        for (unsigned int side=0; side<elem->n_sides(); side++)
          if (elem->neighbor(side) == NULL)
            {
              // The value of the shape functions at the quadrature
              // points.
              const std::vector<std::vector<Real> >&  phi_face = fe_face->get_phi();

              // The Jacobian * Quadrature Weight at the quadrature
              // points on the face.
              const std::vector<Real>& JxW_face = fe_face->get_JxW();

              // The XYZ locations (in physical space) of the
              // quadrature points on the face.  This is where
              // we will interpolate the boundary value function.
              const std::vector<Point >& qface_point = fe_face->get_xyz();

              // Compute the shape function values on the element
              // face.
              fe_face->reinit(elem, side);

              // Loop over the face quadrature points for integration.
              for (unsigned int qp=0; qp<qface.n_points(); qp++)
                {

                  // The location on the boundary of the current
                  // face quadrature point.
                  const Real xf = qface_point[qp](0);
                  const Real yf = qface_point[qp](1);

                  // The penalty value.  \frac{1}{\epsilon}
                  // in the discussion above.
                  const Real penalty = 1.e10;

                  // The boundary value.
                  const Real value = exact_solution(xf, yf);

                  // Matrix contribution of the L2 projection.
                  for (unsigned int i=0; i<phi_face.size(); i++)
                    for (unsigned int j=0; j<phi_face.size(); j++)
                      Ke(i,j) += JxW_face[qp]*penalty*phi_face[i][qp]*phi_face[j][qp];

                  // Right-hand-side contribution of the L2
                  // projection.
                  for (unsigned int i=0; i<phi_face.size(); i++)
                    Fe(i) += JxW_face[qp]*penalty*value*phi_face[i][qp];
                }
            }
      }

      // We have now finished the quadrature point loop,
      // and have therefore applied all the boundary conditions.

      // If this assembly program were to be used on an adaptive mesh,
      // we would have to apply any hanging node constraint equations
      dof_map.constrain_element_matrix_and_vector (Ke, Fe, dof_indices);

      // The element matrix and right-hand-side are now built
      // for this element.  Add them to the global matrix and
      // right-hand-side vector.  The  SparseMatrix::add_matrix()
      // and  NumericVector::add_vector() members do this for us.
      system.matrix->add_matrix (Ke, dof_indices);
      system.rhs->add_vector    (Fe, dof_indices);
    }

  // All done!
}


/*int main(int argc, char **argv)*/
/*{*/
        /*trianglelate();*/
        /*triangle_circle(argc,argv);*/
        /*return 0;*/
/*}*/
```

6 自适应插值逼近

引理 1. *Let and $\hat{\Omega}$ be affine equivalent, i.e. there exists a bijective affine mapping*

$$F : \hat{\Omega} \to \Omega, F\hat{x} = B\hat{x} + b$$

with a nonsigular matrix B. If $v \in H^m(\Omega)$, then $\hat{v} = v \circ F \in H^m(\hat{\Omega})$ and there exist a constant $C = C(m, d)$ such that

$$|\hat{v}|_{H^m(\hat{\Omega})} \le C\|B\|^m |\det B|^{-1/2} |v|_{H^m(\Omega)}, \tag{3}$$

$$|v|_{H^m(\hat{\Omega})} \le C\|B^{-1}\|^m |\det B|^{-1/2} |v|_{H^m(\hat{\Omega})}. \tag{4}$$

Here $\|\cdot\|$ denotes the matrix noem associated with the Euclidean norm in $\mathcal{R}^d$.