我的研究笔记

liyiqiang(lyq105 AT 163.com)

2013 年 1 月 4 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。

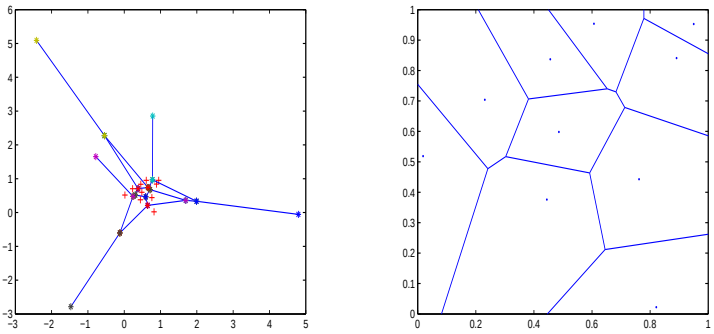
生成 Voronoi 图的软件是 qhull，它可以快速地生成一些点集的凸包，也可以生成一些点集的 Voronoi 图。简单地说 qhull 软件包的功能是输入一些点集，生成的是 Voronoi 图的顶点，以及 cell，也就是 Voronoi 格子。这些格子由一组围成这个格子的顶点的编号来表示。

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)

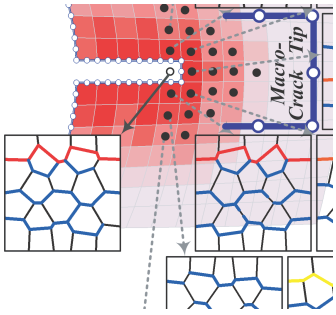


图 1: 计算中的 Voronoi 格子

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

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

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， $u_i(x) = \sum_j \phi_j(x) u_i^j, i = 1, 2, \cdots, d$ 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 $a(u,v) = \sum_k a_k(u,v)$ 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：

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

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"

```

```
59 // Bring in everything from the libMesh namespace
60 using namespace libMesh;
61
62 // Matrix and right-hand side assemble
63 void assemble_elasticity(EquationSystems& es,
64                          const std::string& system_name);
65
66 // Define the elasticity tensor, which is a fourth-order tensor
67 // i.e. it has four indices i,j,k,l
68 Real eval_elasticity_tensor(unsigned int i,
69                             unsigned int j,
70                             unsigned int k,
71                             unsigned int l);
72
73 // Begin the main program.
74 int main (int argc, char** argv)
75 {
76   // Initialize libMesh and any dependent libaries
77   LibMeshInit init (argc, argv);
78
79   // Initialize the cantilever mesh
80   const unsigned int dim = 2;
81
82   // Skip this 2D example if libMesh was compiled as 1D-only.
83   libmesh_example_assert(dim <= LIBMESH_DIM, "2D support");
84
85   Mesh mesh(dim);
86   MeshTools::Generation::build_square (mesh,
87                                        50, 10,
88                                        0., 1.,
89                                        0., 0.2,
90                                        QUAD9);
91
92
93   // Print information about the mesh to the screen.
94   mesh.print_info();
95
96
97   // Create an equation systems object.
98   EquationSystems equation_systems (mesh);
99
100  // Declare the system and its variables.
101  // Create a system named "Elasticity"
102  LinearImplicitSystem& system =
103    equation_systems.add_system<LinearImplicitSystem> ("Elasticity");
104
105
106  // Add two displacement variables, u and v, to the system
107  unsigned int u_var = system.add_variable("u", SECOND, LAGRANGE);
108  unsigned int v_var = system.add_variable("v", SECOND, LAGRANGE);
109
110
111  system.attach_assemble_function (assemble_elasticity);
112
113  // Construct a Dirichlet boundary condition object
114  // We impose a "clamped" boundary condition on the
115  // "left" boundary, i.e. bc_id = 3
116  std::set<boundary_id_type> boundary_ids;
```

```
117   boundary_ids.insert(3);
118
119   // Create a vector storing the variable numbers which the BC applies to
120   std::vector<unsigned int> variables(2);
121   variables[0] = u_var; variables[1] = v_var;
122
123   // Create a ZeroFunction to initialize dirichlet_bc
124   ZeroFunction<> zf;
125
126   DirichletBoundary dirichlet_bc(boundary_ids,
127                                  variables,
128                                  &zf);
129
130   // We must add the Dirichlet boundary condition _before_
131   // we call equation_systems.init()
132   system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);
133
134   // Initialize the data structures for the equation system.
135   equation_systems.init();
136
137   // Print information about the system to the screen.
138   equation_systems.print_info();
139
140   // Solve the system
141   system.solve();
142
143   // Plot the solution
144 #ifdef LIBMESH_HAVE_EXODUS_API
145   ExodusII_IO (mesh).write_equation_systems("displacement.e",equation_systems);
146 #endif // #ifdef LIBMESH_HAVE_EXODUS_API
147
148   // All done.
149   return 0;
150 }
151
152
153 void assemble_elasticity(EquationSystems& es,
154                          const std::string& system_name)
155 {
156   libmesh_assert (system_name == "Elasticity");
157
158   const MeshBase& mesh = es.get_mesh();
159
160   const unsigned int dim = mesh.mesh_dimension();
161
162   LinearImplicitSystem& system = es.get_system<LinearImplicitSystem>("Elasticity");
163
164   const unsigned int u_var = system.variable_number ("u");
165   const unsigned int v_var = system.variable_number ("v");
166
167   const DofMap& dof_map = system.get_dof_map();
168   FEType fe_type = dof_map.variable_type(0);
169   AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));
170   QGauss qrule (dim, fe_type.default_quadrature_order());
171   fe->attach_quadrature_rule (&qrule);
172
173   AutoPtr<FEBase> fe_face (FEBase::build(dim, fe_type));
174   QGauss qface(dim-1, fe_type.default_quadrature_order());
```

```
175   fe_face->attach_quadrature_rule (&qface);
176
177   const std::vector<Real>& JxW = fe->get_JxW();
178   const std::vector<std::vector<RealGradient> >& dphi = fe->get_dphi();
179
180   DenseMatrix<Number> Ke;
181   DenseVector<Number> Fe;
182
183   DenseSubMatrix<Number>
184     Kuu(Ke), Kuv(Ke),
185     Kvu(Ke), Kvv(Ke);
186
187   DenseSubVector<Number>
188     Fu(Fe),
189     Fv(Fe);
190
191   std::vector<unsigned int> dof_indices;
192   std::vector<unsigned int> dof_indices_u;
193   std::vector<unsigned int> dof_indices_v;
194
195   MeshBase::const_element_iterator       el     = mesh.active_local_elements_begin();
196   const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();
197
198   for ( ; el != end_el; ++el)
199     {
200       const Elem* elem = *el;
201
202       dof_map.dof_indices (elem, dof_indices);
203       dof_map.dof_indices (elem, dof_indices_u, u_var);
204       dof_map.dof_indices (elem, dof_indices_v, v_var);
205
206       const unsigned int n_dofs   = dof_indices.size();
207       const unsigned int n_u_dofs = dof_indices_u.size();
208       const unsigned int n_v_dofs = dof_indices_v.size();
209
210       fe->reinit (elem);
211
212       Ke.resize (n_dofs, n_dofs);
213       Fe.resize (n_dofs);
214
215       Kuu.reposition (u_var*n_u_dofs, u_var*n_u_dofs, n_u_dofs, n_u_dofs);
216       Kuv.reposition (u_var*n_u_dofs, v_var*n_u_dofs, n_u_dofs, n_v_dofs);
217
218       Kvu.reposition (v_var*n_v_dofs, u_var*n_v_dofs, n_v_dofs, n_u_dofs);
219       Kvv.reposition (v_var*n_v_dofs, v_var*n_v_dofs, n_v_dofs, n_v_dofs);
220
221       Fu.reposition (u_var*n_u_dofs, n_u_dofs);
222       Fv.reposition (v_var*n_u_dofs, n_v_dofs);
223
224       for (unsigned int qp=0; qp<qrule.n_points(); qp++)
225       {
226           for (unsigned int i=0; i<n_u_dofs; i++)
227             for (unsigned int j=0; j<n_u_dofs; j++)
228             {
229               // Tensor indices
230               unsigned int C_i, C_j, C_k, C_l;
231               C_i=0, C_k=0;
232
```

```
            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)
{
```

```
349   // Define the Poisson ratio
350   const Real nu = 0.3;
351
352   // Define the Lame constants (lambda_1 and lambda_2) based on Poisson ratio
353   const Real lambda_1 = nu / ( (1. + nu) * (1. - 2.*nu) );
354   const Real lambda_2 = 0.5 / (1 + nu);
355
356   // Define the Kronecker delta functions that we need here
357   Real delta_ij = (i == j) ? 1. : 0.;
358   Real delta_il = (i == l) ? 1. : 0.;
359   Real delta_ik = (i == k) ? 1. : 0.;
360   Real delta_jl = (j == l) ? 1. : 0.;
361   Real delta_jk = (j == k) ? 1. : 0.;
362   Real delta_kl = (k == l) ? 1. : 0.;
363
364   return lambda_1 * delta_ij * delta_kl + lambda_2 * (delta_ik * delta_jl + delta_il * delta_jk);
365 }
```

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);
```

```
59                 int n_outer_circle_points = 100;
60         double outer_circle_ridus = 2;
61
62         for(int i = 0; i< n_outer_circle_points; ++i)
63         {
64                 double x = outer_circle_ridus * cos( i* 2 * libMesh::pi / n_outer_circle_points );
65                 double y = outer_circle_ridus * sin( i* 2 * libMesh::pi / n_outer_circle_points );
66                 mesh.add_point(Point (x , y));
67         }
68
69  TriangleInterface t(mesh);
70
71   // Customize the variables for the triangulation
72   t.desired_area()       = .0001;
73
74   // A Planar Straight Line Graph (PSLG) is essentially a list
75   // of segments which have to exist in the final triangulation.
76   // For an L-shaped domain, Triangle will compute the convex
77   // hull of boundary points if we do not specify the PSLG.
78   // The PSLG algorithm is also required for triangulating domains
79   // containing holes
80   t.triangulation_type() = TriangleInterface::PSLG;
81
82   // Turn on/off Laplacian mesh smoothing after generation.
83   // By default this is on.
84   t.smooth_after_generating() = true;
85
86         PolygonHole hole_1(Point(0.,  0.), // center
87                         1,              // radius
88                         100);               // n. points
89         std::vector<Hole*> holes;
90         holes.push_back(&hole_1);
91         t.attach_hole_list(&holes);
92
93         t.triangulate();
94         TecplotIO(mesh).write("anulus.plt");
95         trianglelate(mesh2d);
96         return 0;
97 }
98 */
99
100 /* The Next Great Finite Element Library. */
101 /* Copyright (C) 2003  Benjamin S. Kirk */
102
103 /* This library is free software; you can redistribute it and/or */
104 /* modify it under the terms of the GNU Lesser General Public */
105 /* License as published by the Free Software Foundation; either */
106 /* version 2.1 of the License, or (at your option) any later version. */
107
108 /* This library is distributed in the hope that it will be useful, */
109 /* but WITHOUT ANY WARRANTY; without even the implied warranty of */
110 /* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU */
111 /* Lesser General Public License for more details. */
112
113 /* You should have received a copy of the GNU Lesser General Public */
114 /* License along with this library; if not, write to the Free Software */
115 /* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA */
116
```

```
117
118  // <h1>Example 3 - Solving a Poisson Problem</h1>
119  //
120  // This is the third example program.  It builds on
121  // the second example program by showing how to solve a simple
122  // Poisson system.  This example also introduces the notion
123  // of customized matrix assembly functions, working with an
124  // exact solution, and using element iterators.
125  // We will not comment on things that
126  // were already explained in the second example.
127
128 // C++ include files that we need
129 #include <iostream>
130 #include <algorithm>
131 #include <math.h>
132
133 // Basic include files needed for the mesh functionality.
134 #include "libmesh.h"
135 #include "mesh.h"
136 #include "mesh_generation.h"
137 #include "vtk_io.h"
138 #include "linear_implicit_system.h"
139 #include "equation_systems.h"
140 #include "mesh_tetgen_interface.h"
141 #include "tecplot_io.h"
142 #include "mesh_triangle_interface.h"
143 #include "mesh_triangle_holes.h"
144
145 // Define the Finite Element object.
146 #include "fe.h"
147
148 // Define Gauss quadrature rules.
149 #include "quadrature_gauss.h"
150
151 // Define useful datatypes for finite element
152 // matrix and vector components.
153 #include "sparse_matrix.h"
154 #include "numeric_vector.h"
155 #include "dense_matrix.h"
156 #include "dense_vector.h"
157 #include "elem.h"
158
159 // Define the DofMap, which handles degree of freedom
160 // indexing.
161 #include "dof_map.h"
162
163 typedef TriangleInterface::Hole Hole;
164 typedef TriangleInterface::PolygonHole PolygonHole;
165 typedef TriangleInterface::ArbitraryHole ArbitraryHole;
166 // Bring in everything from the libMesh namespace
167 using namespace libMesh;
168
169 // Function prototype.  This is the function that will assemble
170 // the linear system for our Poisson problem.  Note that the
171 // function will take the  EquationSystems object and the
172 // name of the system we are assembling as input.  From the
173 //  EquationSystems object we have access to the  Mesh and
174 // other objects we might need.
```

```
175 void assemble_poisson(EquationSystems& es,
176                        const std::string& system_name);
177
178 // Function prototype for the exact solution.
179 Real exact_solution (const Real x,
180                      const Real y,
181                      const Real z = 0.);
182
183 int main (int argc, char** argv)
184 {
185   // Initialize libraries, like in example 2.
186   LibMeshInit init (argc, argv);
187
188   // Brief message to the user regarding the program name
189   // and command line arguments.
190   std::cout << "Running " << argv[0];
191
192   for (int i=1; i<argc; i++)
193     std::cout << " " << argv[i];
194
195   std::cout << std::endl << std::endl;
196
197   // Skip this 2D example if libMesh was compiled as 1D-only.
198   libmesh_example_assert(2 <= LIBMESH_DIM, "2D support");
199
200         Mesh mesh(2);
201         mesh.add_point(Point(-1,-1));
202         mesh.add_point(Point(1,-1));
203         mesh.add_point(Point(1,1));
204         mesh.add_point(Point(-1,1));
205
206  TriangleInterface t(mesh);
207
208   // Customize the variables for the triangulation
209   t.desired_area()       = .0001;
210   t.triangulation_type() = TriangleInterface::PSLG;
211   t.smooth_after_generating() = true;
212         PolygonHole hole_1(Point(0.,  0.), // center
213                         0.51,              // radius
214                         100);               // n. points
215         std::vector<Hole*> holes;
216         holes.push_back(&hole_1);
217         t.attach_hole_list(&holes);
218         t.triangulate();
219   mesh.find_neighbors();
220         mesh.prepare_for_use();
221         std::cout << "write mesh file done!!\n" << std::endl;
222         /*mesh.clear();*/
223         TecplotIO(mesh).write("squre_tri.plt");
224   /*// Use the MeshTools::Generation mesh generator to create a uniform*/
225   /*// 2D grid on the square [-1,1]^2.  We instruct the mesh generator*/
226   // to build a mesh of 15x15 QUAD9 elements.  Building QUAD9
227   // elements instead of the default QUAD4's we used in example 2
228   // allow us to use higher-order approximation.
229         //
230         // MeshTools::Generation::build_square (mesh,
231         //
15, 15,
```

```
232         //
    -1., 1.,
233         //
    -1., 1.,
234         //
    TRI3);
235
236         /*TecplotIO(mesh).write("squre_tri_2.plt");*/
237   /*// Print information about the mesh to the screen.*/
238   /*// Note that 5x5 QUAD9 elements actually has 11x11 nodes,*/
239   /*// so this mesh is significantly larger than the one in example 2.*/
240         mesh.print_info();
241         /*return 0;*/
242
243   // Create an equation systems object.
244   EquationSystems equation_systems (mesh);
245
246   // Declare the Poisson system and its variables.
247   // The Poisson system is another example of a steady system.
248   equation_systems.add_system<LinearImplicitSystem> ("Poisson");
249
250   // Adds the variable "u" to "Poisson".  "u"
251   // will be approximated using second-order approximation.
252         FEType fe_type(FIRST, LAGRANGE);
253
254   equation_systems.get_system("Poisson").add_variable("u", fe_type);
255
256   // Give the system a pointer to the matrix assembly
257   // function.  This will be called when needed by the
258   // library.
259
260   equation_systems.get_system("Poisson").attach_assemble_function (assemble_poisson);
261
262   // Initialize the data structures for the equation system.
263   equation_systems.init();
264         /*LinearImplicitSystem& system = equation_systems.get_system<LinearImplicitSystem>("Poisson");*/
265
266   // A reference to the  DofMap object for this system.  The  DofMap
267   // object handles the index translation from node and element numbers
268   // to degree of freedom numbers.  We will talk more about the  DofMap
269   // in future examples.
270   /*const DofMap& dof_map = system.get_dof_map();*/
271
272         /*dof_map.print_info();*/
273   // Prints information about the system to the screen.
274   equation_systems.print_info();
275
276   // Solve the system "Poisson".  Note that calling this
277   // member will assemble the linear system and invoke
278   // the default numerical solver.  With PETSc the solver can be
279   // controlled from the command line.  For example,
280   // you can invoke conjugate gradient with:
281   //
282   // ./ex3 -ksp_type cg
283   //
284   // You can also get a nice X-window that monitors the solver
285   // convergence with:
286   //
```

```
287   // ./ex3 -ksp_xmonitor
288   //
289   // if you linked against the appropriate X libraries when you
290   // built PETSc.
291   equation_systems.get_system("Poisson").solve();
292
293 #if defined(LIBMESH_HAVE_VTK) && !defined(LIBMESH_ENABLE_PARMESH)
294
295   // After solving the system write the solution
296   // to a VTK-formatted plot file.
297   VTKIO (mesh).write_equation_systems ("out.pvtu", equation_systems);
298
299 #endif // #ifdef LIBMESH_HAVE_VTK
300
301         TecplotIO(mesh).write_equation_systems ("squre_tri_res.plt",equation_systems);
302
303   // All done.
304   return 0;
305 }
306
307
308
309 // We now define the matrix assembly function for the
310 // Poisson system.  We need to first compute element
311 // matrices and right-hand sides, and then take into
312 // account the boundary conditions, which will be handled
313 // via a penalty method.
314 void assemble_poisson(EquationSystems& es,
315                       const std::string& system_name)
316 {
317
318   // It is a good idea to make sure we are assembling
319   // the proper system.
320   libmesh_assert (system_name == "Poisson");
321
322
323   // Get a constant reference to the mesh object.
324   const MeshBase& mesh = es.get_mesh();
325
326   // The dimension that we are running
327   const unsigned int dim = mesh.mesh_dimension();
328
329   // Get a reference to the LinearImplicitSystem we are solving
330   LinearImplicitSystem& system = es.get_system<LinearImplicitSystem> ("Poisson");
331
332   // A reference to the  DofMap object for this system.  The  DofMap
333   // object handles the index translation from node and element numbers
334   // to degree of freedom numbers.  We will talk more about the  DofMap
335   // in future examples.
336   const DofMap& dof_map = system.get_dof_map();
337
338   // Get a constant reference to the Finite Element type
339   // for the first (and only) variable in the system.
340   FEType fe_type = dof_map.variable_type(0);
341
342   // Build a Finite Element object of the specified type.  Since the
343   // FEBase::build() member dynamically creates memory we will
344   // store the object as an AutoPtr<FEBase>.  This can be thought
```

```
345   // of as a pointer that will clean up after itself.  Example 4
346   // describes some advantages of  AutoPtr's in the context of
347   // quadrature rules.
348   AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));
349
350   // A 5th order Gauss quadrature rule for numerical integration.
351   QGauss qrule (dim, FIFTH);
352
353   // Tell the finite element object to use our quadrature rule.
354   fe->attach_quadrature_rule (&qrule);
355
356   // Declare a special finite element object for
357   // boundary integration.
358   AutoPtr<FEBase> fe_face (FEBase::build(dim, fe_type));
359
360   // Boundary integration requires one quadraure rule,
361   // with dimensionality one less than the dimensionality
362   // of the element.
363   QGauss qface(dim-1, FIFTH);
364
365   // Tell the finite element object to use our
366   // quadrature rule.
367   fe_face->attach_quadrature_rule (&qface);
368
369   // Here we define some references to cell-specific data that
370   // will be used to assemble the linear system.
371   //
372   // The element Jacobian * quadrature weight at each integration point.
373   const std::vector<Real>& JxW = fe->get_JxW();
374
375   // The physical XY locations of the quadrature points on the element.
376   // These might be useful for evaluating spatially varying material
377   // properties at the quadrature points.
378   const std::vector<Point>& q_point = fe->get_xyz();
379
380   // The element shape functions evaluated at the quadrature points.
381   const std::vector<std::vector<Real> >& phi = fe->get_phi();
382
383   // The element shape function gradients evaluated at the quadrature
384   // points.
385   const std::vector<std::vector<RealGradient> >& dphi = fe->get_dphi();
386
387   // Define data structures to contain the element matrix
388   // and right-hand-side vector contribution.  Following
389   // basic finite element terminology we will denote these
390   // "Ke" and "Fe".  These datatypes are templated on
391   //  Number, which allows the same code to work for real
392   // or complex numbers.
393   DenseMatrix<Number> Ke;
394   DenseVector<Number> Fe;
395
396
397   // This vector will hold the degree of freedom indices for
398   // the element.  These define where in the global system
399   // the element degrees of freedom get mapped.
400   std::vector<unsigned int> dof_indices;
401
402   // Now we will loop over all the elements in the mesh.
```

```
403   // We will compute the element matrix and right-hand-side
404   // contribution.
405   //
406   // Element iterators are a nice way to iterate through all the
407   // elements, or all the elements that have some property.  The
408   // iterator el will iterate from the first to the last element on
409   // the local processor.  The iterator end_el tells us when to stop.
410   // It is smart to make this one const so that we don't accidentally
411   // mess it up!  In case users later modify this program to include
412   // refinement, we will be safe and will only consider the active
413   // elements; hence we use a variant of the \p active_elem_iterator.
414   MeshBase::const_element_iterator       el     = mesh.active_local_elements_begin();
415   const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();
416
417   // Loop over the elements.  Note that  ++el is preferred to
418   // el++ since the latter requires an unnecessary temporary
419   // object.
420   for ( ; el != end_el ; ++el)
421     {
422       // Store a pointer to the element we are currently
423       // working on.  This allows for nicer syntax later.
424       const Elem* elem = *el;
425
426       // Get the degree of freedom indices for the
427       // current element.  These define where in the global
428       // matrix and right-hand-side this element will
429       // contribute to.
430       dof_map.dof_indices (elem, dof_indices);
431
432       // Compute the element-specific data for the current
433       // element.  This involves computing the location of the
434       // quadrature points (q_point) and the shape functions
435       // (phi, dphi) for the current element.
436       fe->reinit (elem);
437
438
439       // Zero the element matrix and right-hand side before
440       // summing them.  We use the resize member here because
441       // the number of degrees of freedom might have changed from
442       // the last element.  Note that this will be the case if the
443       // element type is different (i.e. the last element was a
444       // triangle, now we are on a quadrilateral).
445
446       // The  DenseMatrix::resize() and the  DenseVector::resize()
447       // members will automatically zero out the matrix  and vector.
448       Ke.resize (dof_indices.size(),
449                  dof_indices.size());
450
451       Fe.resize (dof_indices.size());
452
453       // Now loop over the quadrature points.  This handles
454       // the numeric integration.
455       for (unsigned int qp=0; qp<qrule.n_points(); qp++)
456         {
457
458           // Now we will build the element matrix.  This involves
459           // a double loop to integrate the test funcions (i) against
460           // the trial functions (j).
```

```
461           for (unsigned int i=0; i<phi.size(); i++)
462             for (unsigned int j=0; j<phi.size(); j++)
463               {
464                 Ke(i,j) += JxW[qp]*(dphi[i][qp]*dphi[j][qp]);
465               }
466
467           // This is the end of the matrix summation loop
468           // Now we build the element right-hand-side contribution.
469           // This involves a single loop in which we integrate the
470           // "forcing function" in the PDE against the test functions.
471           {
472             const Real x = q_point[qp](0);
473             const Real y = q_point[qp](1);
474             const Real eps = 1.e-3;
475
476
477             // "fxy" is the forcing function for the Poisson equation.
478             // In this case we set fxy to be a finite difference
479             // Laplacian approximation to the (known) exact solution.
480             //
481             // We will use the second-order accurate FD Laplacian
482             // approximation, which in 2D is
483             //
484             // u_xx + u_yy = (u(i,j-1) + u(i,j+1) +
485             //                u(i-1,j) + u(i+1,j) +
486             //                -4*u(i,j))/h^2
487             //
488             // Since the value of the forcing function depends only
489             // on the location of the quadrature point (q_point[qp])
490             // we will compute it here, outside of the i-loop
491             const Real fxy = -(exact_solution(x,y-eps) +
492                                exact_solution(x,y+eps) +
493                                exact_solution(x-eps,y) +
494                                exact_solution(x+eps,y) -
495                                4.*exact_solution(x,y))/eps/eps;
496
497                                                 /*std::cout << fxy << std::endl;        */
498             for (unsigned int i=0; i<phi.size(); i++)
499               Fe(i) += JxW[qp]*fxy*phi[i][qp];
500           }
501         }
502
503
504
505       // We have now reached the end of the RHS summation,
506       // and the end of quadrature point loop, so
507       // the interior element integration has
508       // been completed.  However, we have not yet addressed
509       // boundary conditions.  For this example we will only
510       // consider simple Dirichlet boundary conditions.
511       //
512       // There are several ways Dirichlet boundary conditions
513       // can be imposed.  A simple approach, which works for
514       // interpolary bases like the standard Lagrange polynomials,
515       // is to assign function values to the
516       // degrees of freedom living on the domain boundary. This
517       // works well for interpolary bases, but is more difficult
518       // when non-interpolary (e.g Legendre or Hierarchic) bases
```

```
519       // are used.
520       //
521       // Dirichlet boundary conditions can also be imposed with a
522       // "penalty" method.  In this case essentially the L2 projection
523       // of the boundary values are added to the matrix. The
524       // projection is multiplied by some large factor so that, in
525       // floating point arithmetic, the existing (smaller) entries
526       // in the matrix and right-hand-side are effectively ignored.
527       //
528       // This amounts to adding a term of the form (in latex notation)
529       //
530       // \frac{1}{\epsilon} \int_{\delta \Omega} \phi_i \phi_j = \frac{1}{\epsilon} \int_{\delta \Omega} u \phi_i
531       //
532       // where
533       //
534       // \frac{1}{\epsilon} is the penalty parameter, defined such that \epsilon << 1
535       {
536
537         // The following loop is over the sides of the element.
538         // If the element has no neighbor on a side then that
539         // side MUST live on a boundary of the domain.
540         for (unsigned int side=0; side<elem->n_sides(); side++)
541           if (elem->neighbor(side) == NULL)
542             {
543               // The value of the shape functions at the quadrature
544               // points.
545               const std::vector<std::vector<Real> >&  phi_face = fe_face->get_phi();
546
547               // The Jacobian * Quadrature Weight at the quadrature
548               // points on the face.
549               const std::vector<Real>& JxW_face = fe_face->get_JxW();
550
551               // The XYZ locations (in physical space) of the
552               // quadrature points on the face.  This is where
553               // we will interpolate the boundary value function.
554               const std::vector<Point >& qface_point = fe_face->get_xyz();
555
556               // Compute the shape function values on the element
557               // face.
558               fe_face->reinit(elem, side);
559
560               // Loop over the face quadrature points for integration.
561               for (unsigned int qp=0; qp<qface.n_points(); qp++)
562                 {
563
564                   // The location on the boundary of the current
565                   // face quadrature point.
566                   const Real xf = qface_point[qp](0);
567                   const Real yf = qface_point[qp](1);
568
569                   // The penalty value.  \frac{1}{\epsilon}
570                   // in the discussion above.
571                   const Real penalty = 1.e10;
572
573                   // The boundary value.
574                   const Real value = exact_solution(xf, yf);
575
576                   // 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

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

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

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

7 关于边界元网格的考虑

在边界元计算中，首先要生成边界元网格，这样的网格可以通过有限元网格的生成来实现。一般来说，只要提取出有限元网格的表面就可以当作边界元计算的网格了。

7.1 gmsh 对边界元网格生成的支持

下面给出 gmsh 网格数据文件的例子 [1].

```
                $MeshFormat
    2.2 0 8
    $EndMeshFormat
    $Nodes
    6                       six mesh nodes:
    1 0.0 0.0 0.0             node #1: coordinates (0.0, 0.0, 0.0)
    2 1.0 0.0 0.0             node #2: coordinates (1.0, 0.0, 0.0)
    3 1.0 1.0 0.0             etc.
    4 0.0 1.0 0.0
    5 2.0 0.0 0.0
    6 2.0 1.0 0.0
    $EndNodes
    $Elements
    2                       two elements:
    1 3 2 99 2 1 2 3 4        quad #1: type 3, physical 99, elementary 2, nodes 1 2 3 4
    2 3 2 99 2 2 5 6 3        quad #2: type 3, physical 99, elementary 2, nodes 2 5 6 3
    $EndElements
    $NodeData
    1                       one string tag:
    ”A scalar view”           the name of the view (”A scalar view”)
    1                       one real tag:
    0.0                       the time value (0.0)
    3                       three integer tags:
    0                         the time step (0; time steps always start at 0)
    1                         1-component (scalar) field
    6                         six associated nodal values
    1 0.0                   value associated with node #1 (0.0)
    2 0.1                   value associated with node #2 (0.1)
    3 0.2                   etc.
    4 0.0
    5 0.2
    6 0.4
    $EndNodeData
```

这是一个 gmsh2 的文件格式，这个文件将所有的单元都写在文件里面了。

7.2 netgen 对边界元网格生成的支持

下面给出 netgen 的两种网格文件的例子 [2]。

```
surfacemesh
8
        0         0         0
        0         0         1
        1         0         0
        0         1         0
        1         0         1
```

```
        0         1         1
        1         1         0
        1         1         1
12
      1       4       7
      7       3       1
      4       1       2
      2       6       4
      8       7       4
      6       8       4
      7       8       5
      3       7       5
      8       6       2
      2       5       8
      3       5       2
      1       3       2
```

这一个文件是表面网格数据，他的包含的是 8 个点和 12 个三角形面，覆盖了一个正方体。

```
8
  0.000000  0.000000  0.000000
  0.000000  0.000000  1.000000
  1.000000  0.000000  0.000000
  0.000000  1.000000  0.000000
  1.000000  0.000000  1.000000
  0.000000  1.000000  1.000000
  1.000000  1.000000  0.000000
  1.000000  1.000000  1.000000
6
   1         4       2       6       8
   1         8       7       2       5
   1         3       2       1       7
   1         3       5       2       7
   1         1       2       4       7
   1         8       7       4       2
12
   1           1       4       7
   1           7       3       1
   2           4       1       2
   2           2       6       4
   3           8       7       4
   3           6       8       4
   4           7       8       5
   4           3       7       5
   5           8       6       2
   5           2       5       8
   6           3       5       2
   6           1       3       2
```

这一个文件是体网格数据，他的包含的是 8 个点和 12 个三角形面以及 6 个四面体单元，这些单元剖分了一个正方体, 其中单元的数据行开始都是以其标记开始的。

7.3 总结

需要注意的是，对于生成的表面网格，都是外法向反时针的顺序。因此就不需要再对得到的外法向做方向调整了。

今后的主要支持方向还是 gmsh, 因为还有数据文件，数据文件还指望 gmsh。

参考文献

[1] Geuzaine C, Remacle J. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities [J]. International Journal for Numerical Methods in Engineering. 2009, 79 (11): 1309–1331.

[2] Schöberl J. NETGEN An advancing front 2D/3D-mesh generator based on abstract rules [J]. Computing and visualization in science. 1997, 1 (1): 41–52.

8 Libmesh 导入网格数据的几种方法

libmesh 提供了数种导入网格的方法，包括

1. 自己定义的 xda 格式的网格文件。
2. Gmsh 网格剖分器支持的文件。
3. Abaqus 软件的网格文件,AbaqusIO 只有读
4. TECPLOT 软件的数据结果写出。

总得来说其最主要的是两个类 Meshinput 和 Meshoutput。

9 曙光机群中 MPI 的使用方法

1. 配置运行环境确认你要用的 mpi

```
source /public/software/mpi/openmpi1.3.4-gnu.sh
or
source /public/software/mpi/openmpi1.3.4-intel.sh
```

2. 确认 mpicc 所调用的编译器

```
mpicc -show
```

3. 编译 cpi.c

```
mpicc -o cpi-openmpi cpi.c
```

4 运行

```
mpirun -np 16 -machinefile ma --mca btl self,tcp  ./cpi-openmpi
```

参数说明
-np 16 启 16 个进程
-machinefile ma 文件写出在哪些节点上执行该命令，格式如下
ma 的格式如下
node1 slots=8
node2 slots=8
5 查看该 openmpi 支持的网络协议

```
ompi_info | grep btl
```

6 运行 OpenMPI 时选择网络协议的方法

```
--mca btl self,tcp              使用以太网 TCP/IP 通讯
               self,sm                  当单节点运行的时候，使用内存通讯，效率高
               self,openib           有 infiniband 设备时，使用 IB 通讯
               self,sm,openib    有 infiniband 设备时，节点间使用 IB 通讯，节点内使用内存通信
```

```
--mca btl_tcp_if_include eth0 以太网通讯时用 eth0 通讯
```

10 热传导方程的渐进均匀化过程

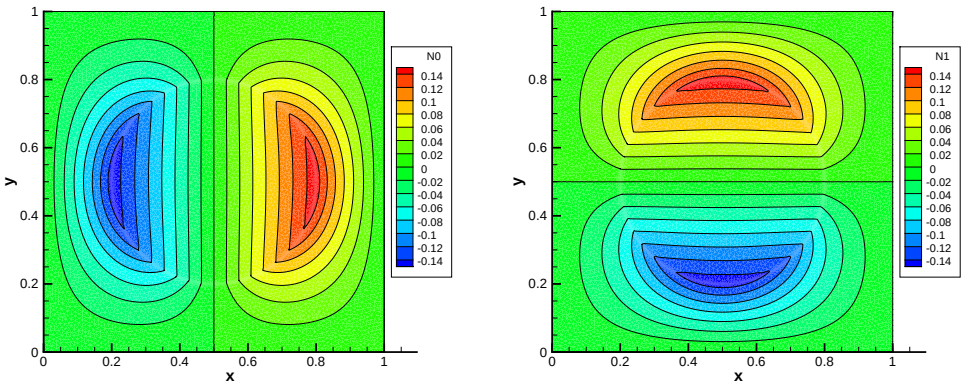


图 2: 一阶均匀化函数

11 骨重建的简单算例

完成时间 2012-12-06

11.1 模拟结果

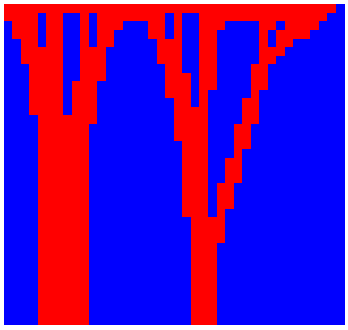


图 3: 40× 40 网格结果

11.2 ANSYS 模拟程序

```
FINISH  ! Make sure we are at BEGIN level
/CLEAR,NOSTART  ! Clear model since no SAVE found

FINISH

!! 计算中的参数

B = 1.0
C = 100
ka = 0.25e3
gamma = 2
rho_init = 0.8e3
rho_0 = 0.01e3
rho_cb = 1.74e3
possion_ratio = 0.3
ediv = 100
square_lenth = 0.1

!! 定义上表面的表载荷

*dim,toppres,table,2,1,,x, ,
*set,toppres(1,0) , 0
*set,toppres(1,1) , 10e6
*set,toppres(2,0) , square_lenth

/prep7
```

```
rectng,0,square_lenth,0,square_lenth,

!* 选取单元，设置单元参数
et,1,plane183
!et,1,plane42

keyopt,1,1,0
keyopt,1,2,0
keyopt,1,3,0
keyopt,1,5,0
keyopt,1,6,0

!! 剖分网格
mshape,0,2d
mshkey,1
asel,all
esize, ,ediv,
amesh,all

!# 准备数据
!! 获取最大单元数
*get,emax,elem,,num,max
*get,nmax,node,,num,max
*dim,rho,,emax
*dim,esed,,emax

!! 设置材料参数
*do, eindex , 1,emax,1
    rho(eindex) =  rho_init    !! 设定每一个单元的密度初始值
                mp,prxy,eindex, possion_ratio !! 设定每一个单元的 'poissons ratio
    mp,ex,eindex,C*rho(eindex)**gamma
                mpchg,eindex,eindex
*enddo
finish

!! 清空数据文件

*cfopen,density,plt
*cfclos
*cfopen,obj,plt
*cfclos

!! 时间步
*do,t,0,200,1
    /sol
    lsel,s,line,,1
    dl,all, ,uy,0

    lsel,s,line,,3
    sfl,all,pres, %toppres%

    allsel,all
    solve
    finish

    !! 画图
                /post1
                /show,jpeg
```

```
                plnsol, u,y, 0,1.0
                /show,close
                /rename,file000,jpg,,uy%t%,jpg
                finish

    obj_function = 0

!! 单元循环更新单元材料参数,
    *do,eindex,1,emax,1
         !! 计算单元的应变能密度N
         *get,esen,elem,eindex,tene 提取单元应变,    !中写的是“helpstiffness ”energy
         *get,evol,elem,eindex,volu 提取单元的体积   !
         *set,esed(eindex),esen/evol 得到单元的应变能密度     !

         !! 更新密度如果密度值在  rho_0 和rho_cb 之间则更新密度。
         *if,rho(eindex),lt,rho_0,then
            rho(eindex) = rho_0
         *endif

         *if,rho(eindex),gt,rho_cb,then
            rho(eindex) = rho_cb
         *endif

         *if,rho(eindex),gt,rho_0,and,rho(eindex),lt,rho_cb,then
            !! 密度更新方程
            rho(eindex) = rho(eindex) + B*(esed(eindex)/rho(eindex) - ka)计算目标函数
                  !!
            obj_function = obj_function + abs(esed(eindex)/rho(eindex) - ka)
         *endif

         !! 更新单元的材料参数
         /prep7
         mp,ex,eindex,C*rho(eindex)**gamma   !! 模量Young
    *enddo !! 单元循环

    obj_function = obj_function/emax

    !! 输出目标函数值

    *cfopen,obj,plt,,append
    *vwrite,t,obj_function
%12.8f %12.8f
    *cfclos

!! 打开密度数据文件
*cfopen,density,plt,,append

*vwrite,ediv+1,ediv+1
zone i=%8I,j=%8I
*do,i,1,ediv,1
        *do,j,1,ediv,1
            ielem = (i-1)*ediv + j
                        ildnode = nelem(ielem,1)
      p = rho(ielem)*1e-3
      *vwrite,nx(ildnode),ny(ildnode),esed(ielem)*1e-6/p,p
%12.8f %12.8f %18.8f %18.8f
  *enddo
            ielem = (i-1)*ediv + j
```

```
                        ildnode = nelem(ielem,2)
      p = rho(ielem)*1e-3
      *vwrite,nx(ildnode),ny(ildnode),esed(ielem)*1e-6/p,p
%12.8f %12.8f %18.8f %18.8f

*enddo

*do,j,1,ediv,1
    ielem = (ediv-1)*ediv + j
    ildnode = nelem(ielem,4)
    *vwrite,nx(ildnode),ny(ildnode),esed(ielem)*1e-6/p,p
%12.8f %12.8f %18.8f %18.8f
*enddo

ielem = (ediv-1)*ediv + j
ildnode = nelem(ielem,3)
*vwrite,nx(ildnode),ny(ildnode),esed(ielem)*1e-6/p,p
%12.8f %12.8f %18.8f %18.8f
*cfclos !! 结束文件输入

*enddo !! 时间步

FINISH
```

12 H 矩阵的快速算法

??

参考文献