第十三讲：自适应有限元软件平台PHG

陈俊清

jqchen@math.tsinghua.edu.cn

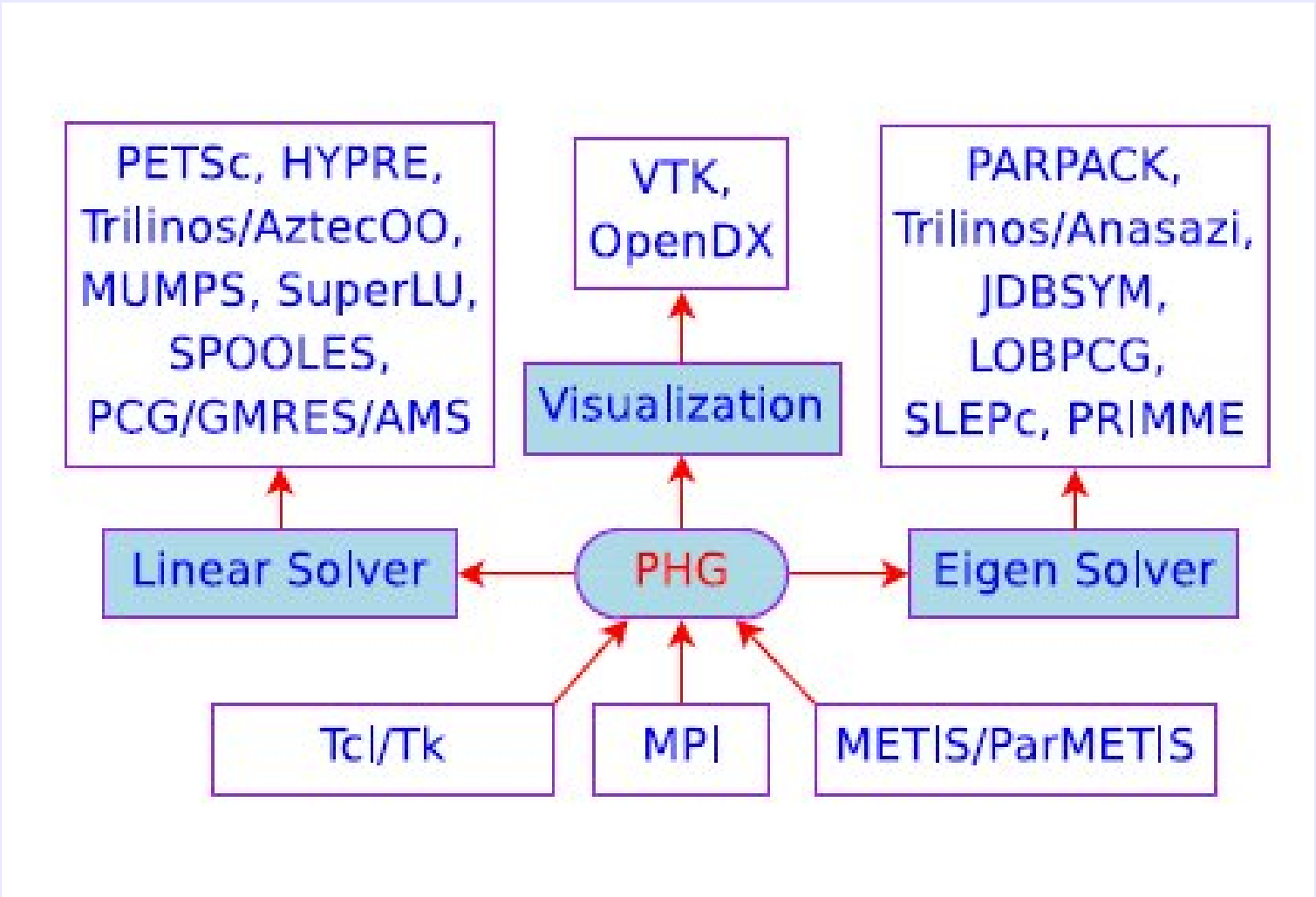
清华大学数学科学系

May 31, 2013

Outline

PHG简介

PHG（Parallel Hierarchical Grid）是中科院科学与工程计算国家重点实验室研制的一个三维自适应有限元软件开发平台。 采用C语言开发，基于MPI消息传递通信并行。



PHG简介-主要功能

主要功能:

- 面向三维四面体单元协调网格;
- 基于单元二分的网格局部加密与放粗;
- 基于网格剖分的并行实现、动态负载平衡;
- 自由度管理、有限元基函数、数值积分;
- 隐藏网格及并行细节;
- 线性解法器、特征值解法器接口;
- 可视化文件输出（vtk等）

PHG简介-安装

PHG的下载与配置、编译、安装

- 下载地址：http://lsec.cc.ac.cn/phg/download.htm
- 编译、安装:

```
tar xjpvf phg-x.x.x-xxxxx.tar.bz2
cd phg-x.x.x
./configure
make
make install
```

必要时需使用一些选项来指定某些软件的位置。运行./configure –help将显示所有选项。 如果发生问题，可在运行./configure 后查看文件config.log来找出某些模块配置不正确的原因。

PHG简介-configure 选项的例子

```
env FC="f77 -64" F77="f77 -64" CC="gcc -mabi=64" \
     CXX="g++ -mabi=64 -DMPI_NO_CPPBIND" \
     LDFLAGS="-L/usr/freeware/lib64 -L/usr/local/lib64" \
     ./configure --with-fc-libs="-lfortran -lftn" \
         --libdir=/usr/local/lib64 \
         --with-mumps-lib="-ldmumps -lpord -L/usr/local/SCALA
         -lscalapack-64 -L/usr/local/BLACS/LIB64 \
         -lblacs -lblacsF77init -lblacs -lpthread" \
         --with-superlu-incdir=/usr/local/superlu_dist/inclu
         --with-superlu-lib="-L/usr/local/superlu_dist/lib64
         -lsuperlu_dist_2.0"\
         --with-hypre-dir=/usr/local/hypre64 \
         --with-tcl-incdir=/usr/freeware/include \
         --with-tcl-libdir=/usr/freeware/lib64 \
         --with-metis-libdir=/usr/local/parmetis/lib64 \
         --with-metis-incdir=/usr/local/parmetis/include \
```

PHG简介-第三方软件

第三方软件:

- MPI：缺省使用mpicc、mpiCC编译， 可用环境变量 CC 和 CXX 指定 C 和 C++ 编译器,用选项 –with-mpi-libdir 或 –with-mpi-lib 以及 –with-mpi-incdir 分别指定 MPI 库 和头文件。
- 网格剖分：METIS/ParMETIS
  http://glaros.dtc.umn.edu/gkhome/views/metis 也可以从
  ftp://159.226.92.111/pub/RPMS/ 中下载、安装 parmetis 包。
- 可视化：paraview: http://www.paraview.org
  可以通过yum install paraview网络安装。

PHG简介

- 使用系统中的BLAS和LAPACK:

```
./configure --with-blas=-lblas \
                      --with-lapack=-llapack
./configure --with-lapack="-lblas -llapack"
```

- 使用Intel MKL:

```
./configure --with-lapack="-L/opt/intel/mkl/lib/32 \
        -lmkl_lapack -lmkl_def -lguide -lpthread"
```

如果使用 PETSc, configure 可自动通过 PETSc 的配置找到它所使用的 BLAS/LAPACK 库。 PHG 用到的所有软件包应该使用同样的 BLAS/LAPACK 库以免发生冲突。

PHG简介-线性解法器

- PETSc. configure 通过环境变量PETSC_DIR得到PETSc的安装路径及配置。（安装完PETSc后要在.bashrc中设置该环境变量。）
- Hypre.下载安装hypre源码（安装到/usr/local目录下），缺省情况configure会自动找到头文件和库， 必要时用–with-hypre-dir指定Hypre的安装目录。
- SuperLU,http://crd.lbl.gov/~ xiaoye/SuperLU/

特征值解法器： PARPACK、JDBSYM、BLOPEX (LOBPCG)等。

网格文件格式

PHG通过函数phgImport可以从多种文件格式中导入初始网格。
Albert格式的文件结构:

```
DIM:                        求解问题的维数
DIM_OF_WORLD:               空间维数
number of vertices:         顶点数 (nv)
number of elements:         单元数 (ne)

vetex coordinates:
顶点0坐标
顶点1坐标
。 。 。  。 。 。
顶点nv-1坐标
```

网格文件格式

element vertices:
单元0顶点编号
单元1顶点编号
。。。 。。。
单元ne-1顶点编号

element boundaries:
单元0边界类型
单元1边界类型
。。。 。。。
单元ne-1边界类型

网格文件格式

element type:
单元0类型
单元1类型
。。。 。。。
单元ne-1类型

element neighbours:
单元0的邻居单元
单元1的邻居单元
。。。 。。。
单元ne-1的邻居单元

curved boundaries:
曲面个数
曲面方程；x坐标投影；y坐标投影；z坐标投影

网格文件格式

PHG要求“求解问题维数”和“空间维数”均为3。

- “vertex coordinates”中每行3个数，给出顶点的x,y,z坐标。
- “element vertices”中每行包含四个整数，顺序给出单元的四个顶点的编号。
- “element boundaries”中每行包含4个整数，给出单元四个面的边界类型，1表示Dirichlet边界，< 0表示Neumann边界，2-11分别表示BDRY_USER0-BDRY_USER9, 0表示内部边。
- ”element neighbours”给出每个单元4个邻居的单元编号，-1表示边界面。PHG忽略albert输入文件中的邻居关系自行重新生成相关信息。

网格文件格式

curved boudaries是PHG的一个扩展，用来定义曲面边界，其数据由一个整数n和n组公式构成。 每组公式中包含4个用“；”隔开的表达式，它们均为x,y,z的函数，即每组公式取如下形式：

$$C(x,y,z); P(x,y,z); Q(x,y,z); R(x,y,z)$$

它表示曲面方程为$C(x,y,z)=0$,而$(P(x,y,z), Q(x,y,z), R(x,y,z))$则为点$(x,y,z)$投影到曲面上的坐标。（参看./test/sphere.dat）

网格文件格式

- 如果输入文件中不包含“element type”项,则 PHG 会根据特定的算法自动为每个单元指定一 种类型, 并且相应修改单元中四个顶点的顺序,以确保初始网格满足二分细化算法所要求的相容性条件。
- 如果输入文件中不包含 element boundaries” 则 PHG 将所有边界面的类型置为 UNDEFINED。

网格文件格式

Medit格式:

- Medit是一个网格显示与处理软件，一些免费的网格自动生成软件， 如Tetgen,Gmsh等可以输出Medit格式的为网格文件。具体Medit的格式细节可以参看Medit的手册。
- 如果使用 Netgen 生成网格的话,可以利用脚本 utils/netgen2medit 将 Netgen 的 .geo 或 neutral 格式文件转换为 Medit 格式,然后导入到 PHG 中,具体用法请参看脚本输出的帮助信息。

更多的关于网格文件格式的可以参阅PHG的手册。

边界类型

PHG提供的边界类型有：DIRICHLET、NEUMANN、BDRY_USER0-BDRY_USER9和UNDEFINED。 在导入文件时，PHG将网格文件中指定边界类型转换为上述类型之一或他们的组合。 在调用phgImport之前，用户可以调用phgImportSetBdryMapFunc为其指定一个边界类型转换函数，如下例所示：

```
static int bc_map(int bctype){
    switch(bctype){
        case 1: return DIRICHLET;
        case 2: return NEUMANN;
        case 3: return BDRY_USER1;
        case 4: return BDRY_USER2;
        default: return -1; /*invalid bctype*/
    }
}
... ...
phgImportSetBdryMapFunc(bc_map);
phgImport(..., ...);
```

边界类型

也可以用选项 “-bcmap_file filename” 指定一个边界类型转换文件, 该文件中每行包含两列,指定一个输入文件中的类型范围到一个 PHG 类型的转换,列间用空格或 <tab> 隔开。

- 第一列包含一个整数或两个用 “:” 隔开的整数,表示输入文件的一个类型范围,
- 第二列是一个字符串, 取值必须为 “Dirichlet”, “Neumann”, “BDRY_USER0”,..., “BDRY_USER9” 和 “Undefined” 之一, 大小写均可,表示相应的 PHG 边界类型。

例如

```
*:0   Dirichlet
3     Dirichlet
1:2   Neumann
5:    Undefined
```

浮点类型

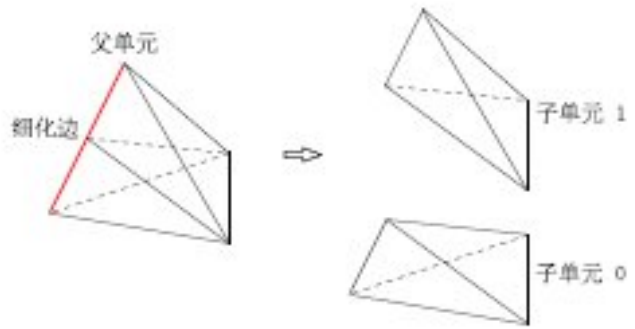
- PHG 中使用的浮点数类型为 FLOAT,默认定义为 C 的 double。在运行 configure 时可以用 –enable-long_double 选项将 FLOAT 指定为 long double。
- 用户程序中调用作用于 FLOAT 类型的数学函数时, 应该使用 PHG 定义的宏, 这些宏的名称通过将 libm 中 double 函数名的首字母改为大写得到, 如Sqrt, Sin，Log 等。
- 在 MPI 通信中, 应该使用 PHG_MPI_FLOAT 作为相应的 MPI 数据类型,不要直接用 MPI_DOUBLE，这样可以保证代码适用于不同 类型的 FLOAT (对其它类型, INT,PHG 也提供了相应的 MPI 数据类型,如 PHG_MPI_INT)。

PHG程序基本结构

```
#include"phg.h"
int main(int argc, char *argv[]){
      ... ...
    phgInit(&argc, &argv);
      ...
    phgFinalize();
    return 0;
}
```

网格加密（细化）

PHG采用二分网格单元局部细化算法，也称为“边细化”算法。对一个单元细化时， 将它的一条边(称为细化边)的中点与与之相对的两个顶点相连，将单元一分为二，如下图所示。 一个单元细化后，被称为细化所产生的单元的父单元，而细化所产生的单元则称为它的子单元。更详细的网格加密过程可以参看PHG文档。



Outline

基本数据类型和常量

- 为了方便使用不同精度进行计算，PHG定义了一组浮点型、整型、字符型和布尔型变量类型： FLOAT，INT，UNIT，SHORT，USHORT，CHAR，BYTE，BOOLEAN（TRUE／FALSE）
- 数学函数： 对应于FLOAT类型的数学函数名通过将标准C库中的数学函数首字母改为大写得到：如Sqrt、Pow、Fabs等。
- 常量(定义在phg.h中)

```
#define Dim 3
#define Nvert 4
#define NFace 4
#define NEdge 6
```

基本数据类型及常量

其它类型:

- GTYPE 用于描述几何对象的几何类型，也用于定义单元的加密类型，可取值为VERTEX、EDGE、FACE、DIAGONAL (ELEMENT)、OPPOSITE、MIXED 和UNKNOWN。
- BTYPE 用于描述边界类型，可应用于顶点、边、面或单元，它由一组标志位构成，这些标志位有：UNREFERENCED (未被叶子单元引用), OWNER (本进程所拥有), INTERIOR (区域内部), REMOTE (与其它进程共享), DIRICHLET (Dirichlet), NEUMANN (Neumann), BDRY_USER0 (用户类型0), ..., BDRY_USER9(用户类型9), UNDEFINED (未指定类型的边界)。

单元对象

```
typedef struct {
    SIMPLEX *children[2];       /* 指向两个子单元的指针*/
    void *neighbours[NFace];    /* 指向邻居单元的指针*/
    INT verts[NVert];           /* 4个顶点的本地编号*/
    INT edges[NEdge];           /* 6条边的本地编号*/
    INT faces[NFace];           /* 4个面的本地编号*/
    INT index;                  /* 单元编号*/
    SHORT mark;                 /* 用于标注加密、粗化*/
    BTYPE bound_type[NFace];    /* 单元面的边界类型*/
    ... ...
} SIMPLEX;
```

单元对象

单元的顶点、边、面的编号分为全局编号、本地(或局部) 编号和单元内编号三种。一个顶点和其 相对的面的单元内编号相同。6 条边的单元内编号顺序为0 (0-1), 1 (0-2), 2 (0-3), 3 (1-2), 4 (1-3), 5(2-3)。 下述宏计算各种单元内编号:

```
GetEdgeVertex(edge_no, v)    /* 边的顶点编号, v 取0 或1 */
GetFaceVertex(edge_no, v)    /* 面的顶点编号, v 取0、1 或2 */
GetEdgeNo(v0, v1)            /* 两个顶点对应的边编号*/
```

例如:

$$\text{GetEdgeVertex}(3,0) = 1$$
$$\text{GetEdgeVertex}(3,1) = 2$$
$$\text{GetFaceVertex}(2,1) = 1$$
$$\text{GetEdgeNo}(1,3) = 4$$

网格对象

```
typedef struct {
    MPI_Comm comm;                  /* 包含网格的MPI 通信器*/
    int nprocs, rank;               /* 进程数、进程编号*/
    FLOAT lif;                      /* 负载不平衡因子*/
    COORD *verts;                   /* 子网格中所有顶点的坐标*/
    BTYPE *types_vert;              /* 子网格中所有顶点的边界类型*/
    BTYPE *types_edge;              /* 子网格中所有边的边界类型*/
    BTYPE *types_face;              /* 子网格中所有面的边界类型*/
    BTYPE *types_elem;              /* 子网格中所有单元的边界类型*/
    INT nleaf;                      /* 子网格中的叶子单元数*/
    INT nvert, nvert_global;        /* 本地、全局顶点数*/
    INT nedge, nedge_global;        /* 本地、全局边数*/
    INT nface, nface_global;        /* 本地、全局面数*/
    INT nelem, nelem_global;        /* 本地、全局单元数*/
    ... ...
} GRID;
```

网格对象

- comm、nprocs、rank 分别为构成网格的通信器、进程数和当前进程在该通信器中的编号
- GRID 对象中的verts 成员包含(子) 网格中所有顶点的坐标。假设e是一个单元，则e的顶点k($0 \leq k < 4$) 的坐标为:

$$\begin{aligned} x &= g\rightarrow verts[e\rightarrow verts[k]][0] \\ y &= g\rightarrow verts[e\rightarrow verts[k]][1] \\ z &= g\rightarrow verts[e\rightarrow verts[k]][2] \end{aligned}$$

- lif 表示网格划分的负载不平衡因子(load imbalance factor):

$$lif = \frac{\text{子网格最大单元数}}{\text{子网格平均单元数}}$$

网格导入与销毁

```
GRID *g;
g = phgNewGrid(-1);
phgImport(g, "cube.dat", FALSE);
 ... ...
phgFreeGrid(&g);
```

phgImport 最后一个参数取TRUE 表示导入网格时强制对网格进行剖分并分布到MPI_COMM_WORLD的所有进程中(此时comm等于MPI_COMM_WORLD)；FALSE 表示导入网格时不对网格进行剖分，网格只存在于进程0 中(此时comm 等于MPI_COMM_SELF).

网格导出与遍历

导出网格

```
phgExportALBERT(GRID *g, const char *filename);
phgExportDX(GRID *g, const char *fn, DOF *dof, ...);
phgExportDXn(GRID *g, const char *fn, int ndof, DOF **dofs);
phgExportVTK(GRID *g, const char *fn, DOF *dof, ...);
phgExportVTKn(GRID *g, const char *fn, int ndof, DOF **dofs);
```

遍历网格

```
GRID *g;
SIMPLEX *e;
ForAllElements(g, e) {
    ... ...
}
```

网格剖分及负载平衡

```
phgBalanceGrid(GRID *g, FLOAT lif_threshold,
               INT submesh_threshold, DOF *weights, FLOAT powe
```

- lif_threshold: 当负载不平衡因子大于该值时重新划分网格
- submesh_threshold: 子网格最小单元数，用于确定子网格数目
- weights: 单元权重,P0类型的自由度对象
- power: 单元权重指数(真实单元权重为weights 的power 次方)

网格的加密与放粗

```
phgRefineAllElements(GRID *g, int level);
phgRefineMarkedElements(GRID *g);
phgCoarsenMarkedElements(GRID *g);
```

后两个函数根据单元中的mark 成员加密、放粗网格。单元每加密一次mark值会被减去1， 每放粗一次mark 值会被加上1。由于加密是强制性的，因此从phgRefineMarkedElements返回时所有单元的mark成员一定是非正值。

Outline

自由度类型与有限元基函数

```
typedef struct DOF_TYPE_ {
      const char *name;              /* 自由度类型的名称*/
      FLOAT *points;                 /* 自由度位置(插值点) */
      DOF_TYPE *grad_type;           /* 梯度的自由度类型*/
      DOF_INTERP_FUNC InterpC2F;     /* 父单元到子单元插值*/
      DOF_INTERP_FUNC InterpF2C;     /* 子单元到父单元插值*/
      DOF_INIT_FUNC InitFunc;        /* 初始化(投影) 函数*/
      DOF_BASIS_FUNC BasFuncs;       /* 计算基函数值*/
      DOF_BASIS_GRAD BasGrads;       /* 计算基函数梯度值*/
      BOOLEAN invariant;             /* 基函数与单元形状无关*/
```

自由度类型

```
    SHORT nbas;                 /* 一个单元中基函数个数*/
    BYTE order;                 /* 基函数多项式次数*/
    CHAR continuity;            /* 有限元函数的连续性*/
    SHORT dim;                  /* 基函数维数(1或Dim) */
    SHORT np_vert;              /* 顶点自由度个数*/
    SHORT np_edge;              /* 边自由度个数*/
    SHORT np_face;              /* 面自由度个数*/
    SHORT np_elem;              /* 单元自由度个数*/
    ... ...
} DOF_TYPE;
```

已定义的有限元类型

- DOF_P0–DOF_P4:0 至4 阶Lagrange 元(除DOF_P0 外均为H^1-conforming)
- DOF_HB1–DOF_HB15:1 至15 阶Hierarchical H^1-conforming 元
- DOF_HC0–DOF_HC15 1 至15 阶Hierarchical H(curl)-conforming 元
- DOF_DG0–DOF_DG15: 0 至15 阶Discontinuous Galerkin 元(L2) 其中DOF_HCn 为向量基(三维)，其余为标量基。 DOF_DGn 基于DOF_Pn ($n \leq 4$) 或DOF_HBn ($n > 4$) 构造。

另外，DOF_ND1 (线性Nédélec 元) 等价于DOF_HC0，DOF_HFEB1 等价于DOF_HC1，DOF_HFEB2 等价于DOF_HC2

自由度对象

```
typedef struct DOF_ {
      char *name;          /* name */
      GRID *g;             /* the mesh */
      DOF_TYPE *type;      /* type */
      SHORT dim;           /* # variables per location */
      BTYPE DB_mask;       /* Dirichlet boundary mask */
      FLOAT *data;         /* data array */
      FLOAT *data_vert;    /* pointer to vertex data */
      FLOAT *data_edge;    /* pointer to edge data */
      FLOAT *data_face;    /* pointer to face data */
      FLOAT *data_elem;    /* pointer to element data */
        ... ...
} DOF;
```

创建、导出和销毁自由度对象

```
DOF *u;
u = phgDofNew(GRID *g, DOF_TYPE *type, SHORT dim,
              const char *name, DOF_USER_FUNC userfunc);
... ...
phgDofFree(&u);
```

其中userfunc 可以取下述形式:

- DofNoData: 不保存自由度数据
- DofNoAction: 网格改变(加密或放粗) 时不对自由度数据进行插值
- DofInterpolation: 网格改变时自动对自由度数据进行插值
- 函数指针: 指向形为

  ```
  void userfunc(FLOAT x, FLOAT y, FLOAT z, FLOAT *values)
  ```

  的函数，PHG 根据该函数的函数值来计算自由度值(通过插值或L2 投影)

函数phgExportVTK 和phgExportDX 允许将任意数量的自由度对象写入到可视化文件中。

自由度编号

PHG 对自由度的编号采用局部编号，它基于子网格中的顶点、边、面和单元的局部编号。具体编号方式是:

- 对子网格中自由度以（点>边>面>体）顺序，按局部标号顺序进行编号。
- 对于同时属于多个子网格的自由度，它们在不同子网格中有着不同的编号，其中唯一一个子网格 为它的“属主”，该子网格便是它所在的顶点、边、面或单元属主。
- PHG不提供自由度的全局编号，需要自由度的全局编号时可以通过MAP 和VEC 对象来进行创建和管理。

直接访问自由度数据

对自由度u，DofData(u) 指向数据缓冲区，(等价于u→data)，
DofVertexData(u,vertex no) 指向指定顶点的自由度数据的起始地址，
DofEdgeData(u,edge no) 指向指定边的自由度数据的起始地址，
DofFaceData(u,face no) 指向指定面的自由度数据的起始地址，
DofElementData(u,elem no) 指向指定单元的自由度数据的起始地址。
对自由度数据的直接访问通常结合单元、顶点、边或面的遍历进行，例如：

```
int i;
DOF *u;
SIMPLEX *e;
FLOAT *data;
 ... ...
ForAllElements(u->g, e) {
    for (i = 0; i < NFace; i++) {
        data = DofFaceData(u, e->faces[i]);
        ... ...
    }
```

自由度对象的运算

```
phgDofCopy(DOF *src, DOF **dest,
           DOF_TYPE *newtype, char *name)
```

创建自由度对象dest，它是自由度对象src 的拷贝。newtype 取NULL 表示dest 的类型等于 src→type。如果newtype 指定的类型不同于src→type，则dest 等于src 到新的有限元空间上的投影。

```
phgDofAXPY(FLOAT a, DOF *x, DOF **y);
phgDofAXPBY(FLOAT a, DOF *x, FLOAT b, DOF **y);
phgDofMatVec(FLOAT alpha, DOF *A, DOF *x, FLOAT beta, DOF *
```

分别计算$y = ax + y, y = ax + by, y = \alpha Ax + \beta y$, A=NULL表示A为单位阵。

自由度对象的运算

```
phgDofMM(MAT_OP transa, MAT_OP transb, int M, int N, int K,
    FLOAT alpha, DOF *A, int blka, DOF *B, FLOAT beta, DOF **
```

该函数相当于 (按点) 计算

$$C := \alpha \mathrm{transa}(A)_{M\times K}\, transb(B)_{K\times N} + \beta C_{M\times N}$$

其中 transa 和 transb 可分别取 MAT_OP_N 或 MAT_OP_T。它涵盖了 phgDofAXPY、phgDofAXPBY 和 phgDofMatVec 的功能。

有限元函数求值

```
FLOAT *phgDofEval(DOF *u, SIMPLEX *e, const FLOAT lambda[],
        FLOAT *values);
FLOAT *phgDofEvalGradient(DOF *u, SIMPLEX *e,
        const FLOAT lambda[], FLOAT *values);
FLOAT *phgDofEvalDivergence(DOF *u, SIMPLEX *e,
        const FLOAT lambda[], FLOAT *values);
FLOAT *phgDofEvalCurl(DOF *u, SIMPLEX *e,
        const FLOAT lambda[], FLOAT *values);
```

这些函数计算有限元函数在指定重心坐标位置的函数值或导数值。

自由度对象的模

```
FLOAT phgDofNormL1(DOF *u);
FLOAT phgDofNormL1Vec(DOF *u);
FLOAT phgDofNormL2(DOF *u);
FLOAT phgDofNormL2Vec(DOF *u);
FLOAT phgDofNormH1(DOF *u);
FLOAT phgDofNormInftyVec(DOF *u);
```

分别计算自由度的各种范数（L^1, H^1, L^∞等）

自由度对象的微分

```
DOF *phgDofGradient(DOF *src, DOF **newdof,
      DOF_TYPE *newtype, const char *name);
DOF *phgDofDivergence(DOF *src, DOF **newdof,
      DOF_TYPE *newtype, const char *name);
DOF *phgDofCurl(DOF *src, DOF **newdof,
      DOF_TYPE *newtype, const char *name);
```

计算自由度的各种微分（如果存在的话）（梯度，散度，旋度）

特殊自由度对象

- 常量型自由度对象DOF_CONSTANT,，它相当于一个取值为常数的dim 维函数。

```
DOF *A;
A = phgDofNew(g, DOF_CONSTANT, 3, "A", DofNoAction);
phgDofSetDataByValuesV(A, 3, "A", 1., 2., 3.);
```

- 解析型自由度对象DOF_ANALYTIC,它调用一个用户提供的函数计算自由度对象的值

```
static void f(FLOAT x, FLOAT y, FLOAT z, FLOAT *value);
DOF *c;
c = phgDofNew(g, DOF_ANALYTIC, 1, "coefficient", f);
```

特殊自由度对象

- 一个分片常数函数可以用一个解析型自由度对象来表示,当网格面与间断面一致时也可以存储在一个类型为 DOF_P0 的自由度对象中。
- 解析型自由度对象可以和其它自由度对象一样参与代数运行、数值积分、进行求值等, 但不允许对其进行微分运算。

访问邻居单元的自由度数据

串行计算时，可以通过e→neighbours[i] 直接访问单元e 的第i 个邻居单元。 而并行计算时，邻居单元可能位于其它进程，对邻居单元的访问不便于直接用MPI 实现。

为避免对邻居单元的直接访问，一个方法是将计算顺序按照单元遍历的形式进行组织，将每个单元的贡献叠加到计算结果上。 可以利用PHG 的向量组装功能来完成计算结果的叠加.

另外一个方法是调用phgNeighbourData 来获取(远程) 邻居单元的数据(PHG 会自动完成必要的通信)。

访问邻居单元的自由度数据

假设自由度对象u 的类型为DOF_P0 (分片常数)，下述代码计算u 在所有面上的平均值。

```
NEIGHBOUR_DATA *nd;
SIMPLEX *e;
int i;   double a;
nd = phgDofInitNeighbourData(u, NULL);
ForAllElements(u->g, e) {
    for (i = 0; i < NFace; i++) {
        if (边界面) {
            a = *DofElementData(u, e->index);
        }
        else {
            a = (*DofElementData(u, e->index) +
            *phgDofNeighbourData(nd, e, i, 0, NULL)) * .5;
        }
        ... ...
    }
```

几何量自由度对象

PHG 内部自动维护一个特殊的自由度对象，称为几何量自由度对象，用于计算和存储与边、面 和单元相关的一些常用几何量，并提供一组函数供用户访问这些量。

```
FLOAT phgGeomGetVolume(GRID *g, SIMPLEX *e);
FLOAT phgGeomGetDiameter(GRID *g, SIMPLEX *e);
FLOAT *phgGeomGetJacobian(GRID *g, SIMPLEX *e);
FLOAT phgGeomGetFaceArea(GRID *g, SIMPLEX *e, int face);
FLOAT phgGeomGetFaceAreaByIndex(GRID *g, INT face_no);
FLOAT phgGeomGetFaceDiameter(GRID *g, SIMPLEX *e, int face);
FLOAT *phgGeomGetFaceNormal(GRID *g, SIMPLEX *e, int face);
FLOAT *phgGeomGetFaceOutNormal(GRID *g, SIMPLEX *e, int face);
FLOAT *phgGeomXYZ2Lambda(GRID *g, SIMPLEX *e,
        FLOAT x, FLOAT y, FLOAT z);
phgGeomLambda2XYZ(GRID *g ,SIMPLEX *e, const FLOAT *lambda,
        FLOAT *x, FLOAT *y, FLOAT *z);
```

Outline

有限元计算中的数值积分

有限元离散通常归结到计算自由度对象、基函数间的二次型、三次型（在单元上计算）。PHG中这些计算通常利用数值积分来完成。 以单元上的积分为例，假定e 是一个单元，f(x) 是定义在e 上的一个函数，则：

$$\int_e f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i)$$

其中x_i 称为积分点(abscissas)，w_i 称为权重(weights)。适当选取积分点和权重可以使得上述公式对不超过某个次数的任意多项式精确成立。一个积分公式的阶指积分公式对所有次数不超过该阶的多项式精确成立。

PHG 的数值积分公式

PHG 用结构QUAD 存储数值积分公式，这些公式存储在文件src/quad.c 中。该结构体的主要成员如下:

```
char *name;
int dim;          /* 1: 边, 2: 三角形, 3: 四面体*/
int order;        /* 积分公式的阶数*/
int npoints;      /* 积分公式的点数*/
FLOAT *points;    /* 积分点重心坐标*/
FLOAT *weights;   /* 积分点权重*/
```

函数phgQuadGetQuadnD(p) (n = 1; 2; 3) 返回指向一个n 维p 阶QUAD 结构的指针。如果指定阶 数的积分公式不存在，该函数会自动构造一个张量积形式的数值积分公式。

边上的数值积分

假设u 是一个(维数为1 的) 自由度对象，下述代码分别用3 阶积分公式计算u 在单元e 的 边、面和单元上的积分。

```
QUAD *q;
FLOAT lambda[] = {0., 0., 0., 0.}, sum = 0., value, *p, *w;
int i, v0, v1; FLOAT sum = 0., value, *p, *w;
v0 = GetEdgeVertex(k, 0);  v1 = GetEdgeVertex(k, 1);
q = phgQuadGetQuad1D(3);
p = q->points; w = q->weights;
for (i = 0; i < q->npoints; i++) {
     lambda[v0] = *(p++);
     lambda[v1] = *(p++);
     phgDofEval(u, e, lambda, &value);
     sum += *(w++) * value;
}
sum *= 边长度;
```

(边长度可以通过u→g→verts[e→verts[]] 计算)

第k 个面($0 \le k < 4$) 上的积分

```
QUAD *q;
FLOAT lambda[] = {0., 0., 0., 0.}, sum = 0., value, *p, *w;
int i, v0, v1, v2;
v0 = GetFaceVertex(k, 0);
v1 = GetFaceVertex(k, 1);
v2 = GetFaceVertex(k, 2);
q = phgQuadGetQuad2D(3);
p = q->points; w = q->weights;
for (i = 0; i < q->npoints; i++) {
     lambda[v0] = *(p++);
     lambda[v1] = *(p++);
     lambda[v2] = *(p++);
     phgDofEval(u, e, lambda, &value);
     sum += *(w++) * value;
}
sum *= phgGeomGetFaceArea(u->g, e, k);
```

单元上的积分

```
QUAD *q;
FLOAT sum = 0., value, *p, *w;
int i;
q = phgQuadGetQuad3D(3);
p = q->points;
w = q->weights;
for (i = 0; i < q->npoints; i++, p += 4) {
     phgDofEval(u, e, p, &value);
     sum += *(w++) * value;
}
sum *= phgGeomGetVolume(u->g, e);
```

双、三线性型的计算

有限元中经常需要计算双或三线性型，例如：

$$\int_e \phi_i \phi_j, \int_e \nabla\phi_i \cdot \nabla\phi_j, \int_e A\nabla\phi_i \cdot \nabla\phi_j, \int_e f\phi_i, \int_f g\phi_i$$

其中ϕ_i, ϕ_j 为基函数，f 为某个函数，A 是一个标量函数或3× 3 矩阵函数，e 为单元，f 为单元的一个面。

```
phgQuadBasDotBas(e, u, i, v, j, QUAD_DEFAULT);
phgQuadGradBasDotGradBas(e, u, n, v, m, QUAD_DEFAULT);
phgQuadGradBasAGradBas(e, u, n, A, v, m, QUAD_DEFAULT);
phgQuadDofTimesBas(e, f, u, n, QUAD_DEFAULT, result);
phgQuadFaceDofDotBas(e, k, g, DOF_PROJ_NONE, v, n,
        QUAD_DEFAULT);
```

计算面跳量

有限元后验误差估计中经常需要计算面跳量，PHG 提供一个通用的面跳量计算函数:

```
DOF *phgQuadFaceJump(DOF *u, DOF_PROJ proj, const char *name,
        int quad_order);
```

该函数返回一个自由度对象，每个面上一个自由度值，该值等于u 在该面上的跳量的模的平方的积分 (面上L^2 模的平方)。 proj 表示相对于面的外法向的投影。令n为面的外法向，$[\cdot]$表示函数跨过面的跳量，则:

$$\text{proj} = \text{DOF\_PROJ\_NONE 计算} \int_f |[u]|^2$$

$$\text{proj} = \text{DOF\_PROJ\_DOT 计算} \int_f |n \cdot [u]|^2$$

$$\text{proj} = \text{DOF\_PROJ\_CROSS 计算} \int_f |n \times [u]|^2$$

基函数、函数值的cache

许多情况下数值积分中基函数或函数的值会被重复用到。为避免重复计算，PHG 可以将计算过 的基函数值或基函数梯度值保存在特定的cache中。

编写数值积分函数时，如果调用下述函数计算被积函数或基函数值，则PHG 会根据函数或基函 数的性质自动cache 基函数或其梯度的值：

```
FLOAT *phgQuadGetFuncValues(GRID *g, SIMPLEX *e,
        int dim, DOF_USER_FUNC userfunc, QUAD *quad);
FLOAT *phgQuadGetDofValues(SIMPLEX *e, DOF *u, QUAD *quad);
FLOAT *phgQuadGetBasisValues(SIMPLEX *e, DOF *u, int n,
        QUAD *quad);
FLOAT *phgQuadGetBasisGradient(SIMPLEX *e, DOF *u, int n,
        QUAD *quad);
FLOAT *phgQuadGetBasisCurl(SIMPLEX *e, DOF *u, int n,
        QUAD *quad);
```

基函数、函数值的cache

例如，下面的代码计算 $\int_e \nabla\phi_n \cdot \nabla\phi_m$，其中$\phi_n$ 和ϕ_m 分别表示自由度对象u(这里假定其维数为1) 的第n 和第m 个基函数

```
QUAD *quad;
const FLOAT *g1, *g2, *w;
FLOAT d; int i;
quad = phgQuadGetQuad3D(2 * u->type->order - 2);
g1 = phgQuadGetBasisGradient(e, u, n, quad);
g2 = phgQuadGetBasisGradient(e, u, m, quad);
w = quad->weights;
d = 0.;
for (i = 0; i < quad->npoints; i++, g1 += 3, g2 += 3, w++)
     d += (g1[0] * g2[0]
               + g1[1] * g2[1] + g1[2] * g2[2]) * *w;
d *= phgGeomGetVolume(u->g, e);
```

Outline

线性解法器对象

```
typedef struct {
      OEM_SOLVER *oem_solver; /* 外部解法器*/
      MAT *mat;
      VEC *rhs;
      ... ...
} SOLVER;
```

其中，oem_solver 指所使用的“外部”解法器。当oem_solver 为NULL 时表示使用默认解法器， 默认解法器可以在命令行上用选项‘-solver’ 来指定。PHG 目前支持的“外部”解法器(OEM solver) 包括:

```
SOLVER_DEFAULT,
SOLVER_PCG, SOLVER_GMRES, SOLVER_AMS, /* 内部解法器*/
SOLVER_HYPRE, SOLVER_PETSC, SOLVER_TRILINOS, /* 迭代法*/
SOLVER_MUMPS, SOLVER_SUPERLU, SOLVER_SPOOLES, /* 直接法*/
SOLVER_SPC, SOLVER_LASPACK /* 其它解法器*/
```

创建和销毁解法器对象

```
SOLVER *phgSolverCreate(OEM_SOLVER *oem_solver, DOF *u, ...
int phgSolverDestroy(SOLVER **solver);
```

phgSolverCreate 中的可变参数给出一组构成未知量的自由度对象，可变参数表必须以空指针 NULL 结束。

组装线性系统的系数矩阵及右端项

在PHG 的线性解法器中，未知量通常对应一组自由度对象。组装系数矩阵或右端项时通常使用 未知量的局部编号。如果未知量由单个自由度对象构成，则未知量的局部编号就是自由度的局部编 号。 下述函数将自由度编号或单元基函数编号映射为未知量的局部编号:

```
phgSolverMapE2L(solver, dof_no, e, basis_no);
phgSolverMapD2L(solver, dof_no, dof_index);
```

下述函数用来将特定项叠加到系数矩阵或右端项中:

```
phgSolverAddMatrixEntry(SOLVER *solver, INT row, INT col,
        FLOAT value);
phgSolverAddMatrixEntries(SOLVER solver, INT nrows, INT *rows,
        INT ncols, INT *cols, FLOAT *values);
phgSolverAddRHSEntry(SOLVER *solver, INT index, FLOAT value);
phgSolverAddRHSEntries(SOLVER *solver, INT n, INT *indices,
        FLOAT *values)
```

其中行、列编号使用未知量的局部编号。

Dirichlet 边界条件

PHG 提供一个通用的处理Dirichlet 边界条件的函数，它将一个指定位置的Dirichlet 边界条件转化为一个关于某些自由度的线性方程。

```
BOOLEAN phgDofDirichletBC(DOF *u, SIMPLEX *e, int index,
        DOF_USER_FUNC f, FLOAT mat[],
        FLOAT rhs[], DOF_PROJ proj);
```

其中index 代表单元基函数编号。返回值为FALSE 表示该基函数对应的位置不是Dirichlet 边界(哪 些边界属于Dirichlet 边界可通过自由度对象中的DB_mask 成员指定)，否则在mat 中返回边界方程的 系数，在rhs 中返回边界方程右端项。数组mat 的长度应该等于单元基函数的个数，而rhs 的长度则 应该等于u→dim。

用同一个系数矩阵反复求解不同右端项

创建解法器对象solver;
生成系数矩阵和右端项;
组装系数矩阵和右端项;

```
phgSolverSolve(solver, FALSE, ... ...);
phgSolverResetRHS(solver);
```

生成、组装新的右端项;

```
phgSolverSolve(solver, FALSE, ... ...);
... ...
```

销毁解法器对象;

PHG 支持的解法器

- PCG/GMRES
- LASPack
- PETSc
- HYPRE
- MUMPS
- SuperLU
- SPOOLES
- Trilinos/AztecOO

Outline

Poisson方程

example/simplest.c是PHG中最简单的自适应有限元程序实例。它求解下述Dirichlet边界条件的Poisson方程:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= g \quad \text{on } \partial\Omega. \end{aligned}$$

主程序

变量g为网格对象。DOF对象u_h, f_h分别存放数值解和右端项，类型为DOF_DEFAULT(默认情况下为DOF_P2,即二阶Lagrange元，可在运行程序是通过命令行选项-dof_type设定其他类型)。grad_u用于计算和保存数值解的梯度。error用于保存误差指示子，类型为DOF_P0(分片常数)。

```
GRID *g;
DOF *u_h, *f_h, &grad_u, *error;
SOLVER *solver;

phgInit(&argc, &argv);
g=phgNewGrid(-1);
phgImport(g,"cube.dat", FALSE);
u_h=phgDofNew(g,DOF_DEFAULT, 1, "u_h",DofInterpolation);
phgDofSetDataByvalue(u_h,0.0);
f_h=phgDofNew(g,DOF_DEFAULT,1,"f_h",func_f);
error=phgDofNew(g,DOF_P0,1,"error indicator",DofNoAction);
```

主程序

```
while(TRUE){
    phgBalanceGrid(g,1.2,-1);
    solver=phgSolverCreate(SOLVER_DEFAULT,u_h,NULL);
    build_linear_system(solver,u_h,f_h);
    phgSolverSolve(solver,TRUE,u_h,NULL);
    phgSolverDestroy(&solver);
    grad_u=phgDofGradient(u_h,NULL,NULL,NULL);
    estimate_error(u_h,f_h,grad_u,error);
    phgDofFree(&grad_u);
    if(满足收敛条件)break;
    mark_refine(est,...);
    phgRefineMarkedElements(g);
}
phgFreGrid(&g);
phgFinalize();
```

主程序

创建自由度对象u_h时使用参数DofInterpolation，它使得当网格细化或粗化时自动对u_h进行插值，创建f_h 时指明调用函数func_f() 对其进行赋值，而创建error 时使用了参数DofNoAction，表示不对自由度对象进行任何自动的赋值或插值处理，只是为它开辟相应的存储空间。
grad_u 由函数phgDofGradient 创建，每次使用完毕后立即释放。
phgSolverCreate(SOLVER_DEFAULT, u_h, NULL)创建一个以u_h 为未知量的解法器对象，使用的解法器为SOLVER_DEFAULT (默认为PCG，用户可以用命令行选项"-solver" 来指定使用其它解法器)。函数phgSolverCreate 用一个可变参数表列出作为未知量的自由度对象，这些自由度对象在解法器中依次编号为0, 1, 等等。

形成线性方程组

函数build_linear_system() 形成线性方程组，它对当前网格的所有单元进行遍历，计算每个单元的单元刚度矩阵和右端项，然后迭加到线性系统中去。对网格中单元的遍历通过宏ForAllElements来进行。假设线性方程组的系数矩阵为$A(I;J)$，右端向量为B(I)，$I, J = 0, ..., M-1$，M为未知量个数，则计算过程如下

```
ForAllElements(g,e){
    N=DofGetNBas(u_h,e);
    for(i=1;i<N;i++){
        I=phgSolverMapE2L(solver,0,e,i);
        type=phgDofGetElementBoundaryType(u_h,e,i);
        if(type &DIRICHLET){
            将1.0累加到A(I,I); 将u的边界值累加到B(I);
            continue;
        }
        for(j=0;j<N;j++){
            J=phgSolverMapE2L(solver,0,e,j);
            计算(grad \phi_j,grad \phi_i )e并累加到A(I,J);
```

误差指示子的计算

函数estimate_error()计算每个单元上的误差指示子，并存储在DOF对象error中，这里应用的误差指示子为：

$$\eta_e^2 = h_e^2 \|\Delta u_h + f_h\|_{L^2(e)}^2 + \sum_{f \subset \partial e, f \subset \Omega} h_f \|[gradu_h \cdot n_f]\|_{L^2(f)}^2$$

其中h_e为单元e的直径，h_f为面f的直径，n_f为面f的单位外法向。

误差指示子的计算

```
DOF *jump, *residual;
jump=phgQuadFaceJump(grad_u, DOF_PROJ_DOT, NULL,-1);
residual = phgDofDivergence(grad_u, NULL, NULL, NULL);
phgDofAXPY(1.0,f_h,&redidual); /*\Delta u_h+f_h*/
ForAllElements(g,e){
    int i; FLOAT eta,h;
    FLOAT diam = phgGeomGetDiameter(g,e);
    e->mark = 0; eta = 0.0;
    for(i=0;i<NFace;i++){
        if(e->bound_type[i] &(DIRICHLET|NEUMANN))
            continue;
        h=phgGeomGetFaceDiameter(g,e,i);
        eta=+*DofFaceData(jump,e->faces[i])*h;
    }
    eta=eta*0.5+diam*diam*phgQuadDofDotDof(e,redisual,redi
    *DofElementData(error,e->index)=Sqrt(eta);
}
```

Outline

数值例子

- 方程:

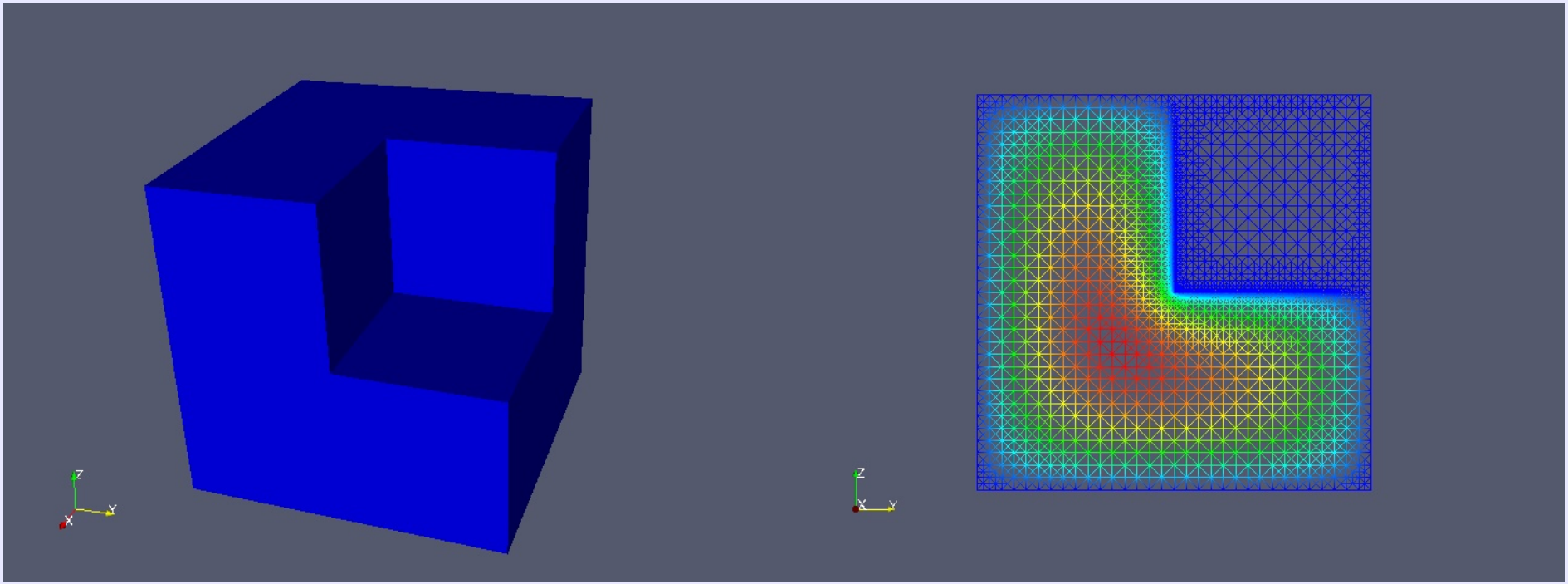
$$\begin{aligned} -\Delta u &= 1 \text{ in } \Omega, \\ u &= 1 \text{ on } \partial\Omega. \end{aligned}$$

- 区域Ω:

$$\Omega = [-1,1] \times [-1,1] \times [-1,1] \backslash [0,1] \times [0,1] \times [0,1].$$

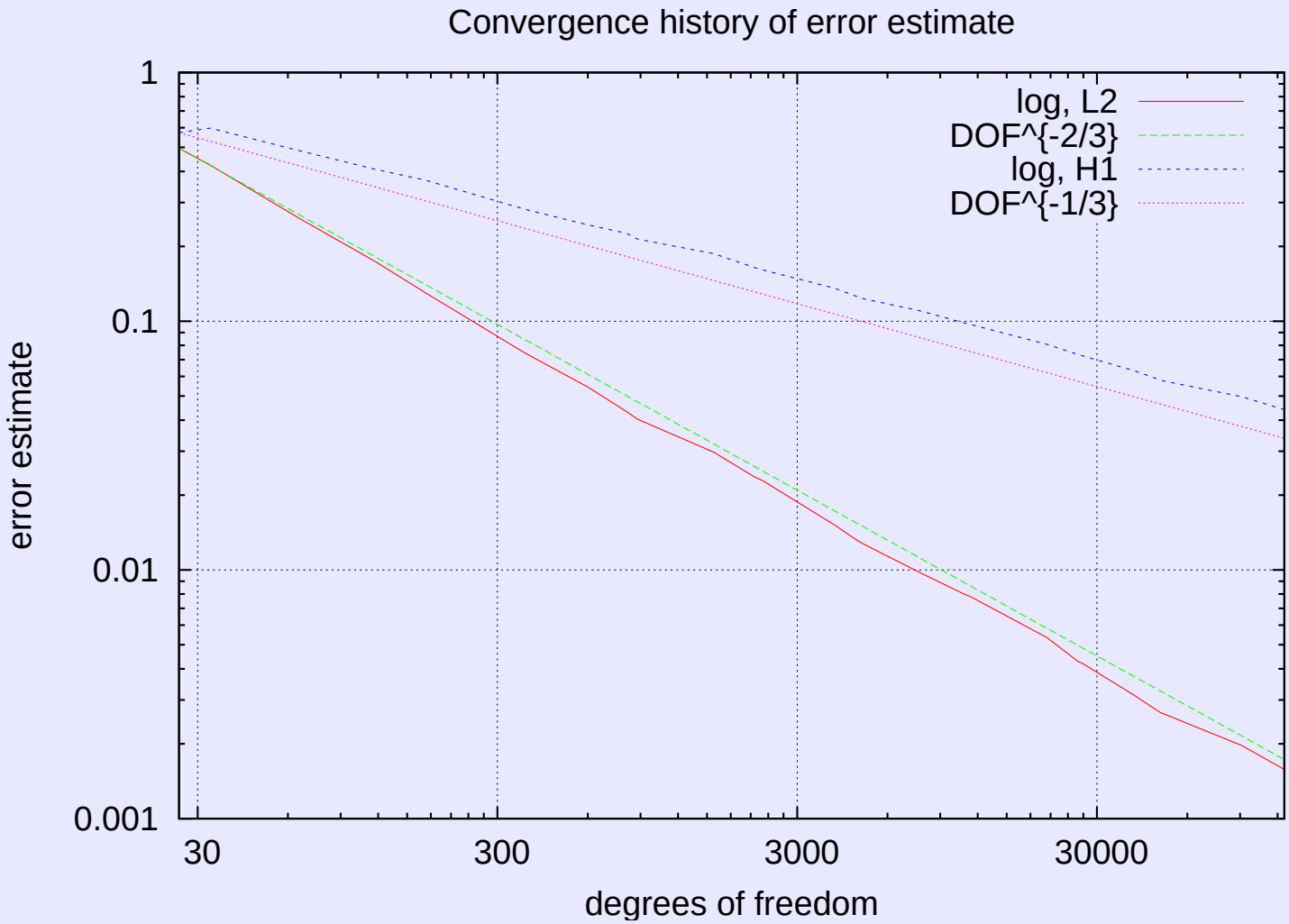
数值例子

网格(参看文件test/cube1o.dat):



数值例子

误差下降阶：


Convergence history of error estimate
log, L2
DOF^{-2/3}
log, H1
DOF^{-1/3}
error estimate
1
0.1
0.01
0.001
30
300
3000
30000
degrees of freedom