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

June 6, 2013

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。)

有限元计算中的数值积分

PHG 的数值积分公式中使用归一化的权重，即将积分写成：

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

其中$V(e)$表示单元e的体积。显然，如果积分公式至少对0 次多项式是精确的，则积分权重必须满足：

$$\sum_{i=0}^{n-1} w_i = 1.$$

PHG 中用单元中的重心坐标表示积分点，易知这种表示下积分点和权重相对于仿射变换是不变的， 换言之，一个积分公式的积分点和权重与单元的形状、位置无关。

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。

第k($0 \le k < 6$)个边上的数值积分

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);
```

分别计算函数userfunc与自由度u在积分点上的值。

基函数、函数值的cache

```
FLOAT *phgQuadGetBasisValues(SIMPLEX *e, DOF *u, int n,
        QUAD *quad);
```

输入参数为单元e，自由度u，基函数的序号n，积分类型quad。该函数返回自由度u 在单元e上的第n个基函数在积分子quad 的求值点处的值。

```
FLOAT *phgQuadGetBasisGradient(SIMPLEX *e, DOF *u, int n,
        QUAD *quad);
```

参数含义同phgQuadGetBasisValues，计算的是基函数关于笛卡尔坐标的梯度。

```
FLOAT *phgQuadGetBasisCurl(SIMPLEX *e, DOF *u, int n,
        QUAD *quad);
```

参数含义同phgQuadGetBasisValues，计算的是基函数的旋度。

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

命令行选项

PHG 可以通过命令行选项的方式来设定、改变程序运行的参数。PHG 的命令行选项根据其来源及定义方式可以分成三类:

- PHG内部定义的选项
- PHG 调用的第三方软件(如PETSc) 所提供的选项
- 用户程序自行定义的选项

其中，前两类选项对任何PHG 程序都是通用的(比如 −help，"-oem_options"等)， 而第三类选项则取决于用户程序（参看程序examples/poisson.c）。

命令行选项

PHG 选项的形式为‘-选项名[参数]’ 或‘-选项名[=参数]’，选项之间用空格隔开，选项名与参数间用空格或等号隔开。

对于不带参数的选项，‘+选项名’ 表示与‘-选项名’ 相反的含义。

当同一个命令行选项多次出现时，以最后一次的值为准。运行任何一个PHG 程序时，都可以用’-help’ 列出它所支持的命令行选项及参数。

函数phgInit 负责对所有命令行选项进行处理。 调用phgInit 之后，命令行(argc, argv) 中将只留下非选项形式的命令行参数及oem_options 选项中的参数。

命令行选项

PHG 的每个选项都有一个与之相对应的变量，选项的作用就是根据选项的参数设定该变量的值。PHG 支持下述几种类型的命令行选项：

选项类型	参数类型	变量类型
无参数	无	BOOLEAN
整型	整数	INT
浮点型	浮点数	FLOAT
字符串	字符串	char *
文件名	文件名	char *
枚举型	关键字	int
函数型	字符串	函数指针(option handler)

命令行选项

选项'-options_file 文件名' 将指定文件(称为选项文件) 中的所有选项插入到当前位置。一个选项文件中可以包含任意数目的选项，选项间用空格或换行隔开。 如果存在名为"可执行文件名.options" 的选项文件，PHG 会自动插入该文件中的选项。
-oem_options 是一个特殊的选项，PHG 将其参数原封不动地传递给其它软件(如PETSc、Hypre等)。当有多个'-oem_options' 选项时，PHG 将它们的所有参数按顺序合并起来。例如，下例选用CG解法器和block Jacobi 预条件子做为PETSc 中的缺省解法器:

```
-oem_options "-ksp_type cg -pc_type bjacobi"
```

命令行选项

用户程序可以在调用phgInit 前调用phgOptionsPreset 来预设一些选项的值。可以多次调用phgOptionsPreset，每次调用phgOptionsPreset 可以给出多个选项。例如:

```
phgOptionsPreset("-dof_type P4");
phgOptionsPreset("-solver_maxit 1000 -solver_rtol 1e-5");
phgOptionsPreset("-solver hypre");
```

如果在运行一个PHG 程序时使用了'-help' 选项，则会显示出它所支持的命令行选项及说明然后立即退出。-help 的参数是一个类别名，如'-help hypre' 显示有关Hypre 的选项，'-help all' 显示所有选项，而'-help help' 则显示出所有类别名。

自定义命令行选项

用户可以调用函数phgOptionsRegisterXxxx 注册新的命令行选项。注册命令行选项必须在调用phgInit 之前进行。

```
phgOptionsRegisterTitle(const char *str, const char *help,
        const char *category);
phgOptionsRegisterNoArg(char *name, char *help, BOOLEAN *v
phgOptionsRegisterInt(char *name, char *help, INT *var);
phgOptionsRegisterFloat(char *name, char *help, FLOAT *var
phgOptionsRegisterString(char *name, char *help, char **va
phgOptionsRegisterFilename(char *name, char *help, char **
phgOptionsRegisterKeyword(char *name, char *help,
        char **keys, int *var);
phgOptionsRegisterHandler(char *name, char *help,
        OPTION_HANDLER func, BOOLEAN append);
```

这些函数的用法可参看poisson.c。

获取命令行参数的当前值

下面的函数可以用来在程序运行过程中获取命令行选项的当前值:

```
BOOLEAN phgOptionsGetNoArg(char *op_name);
INT phgOptionsGetInt(char *op_name);
FLOAT phgOptionsGetFloat(char *op_name);
char *phgOptionsGetKeyword(char *op_name);
char *phgOptionsGetString(char *op_name);
char *phgOptionsGetFilename(char *op_name);
```

例如:

```
phgPrintf("Default solver: %s\n",
        phgOptionsGetKeyword("default_solver"));
```

动态设置命令行选项的参数值

下列函数用于在程序运行过程中动态改变命令行选项对应的变量的值。需要注意的是，有些命令行选项是在程序启动之初发生作用，对于这些选项，程序运行中改变它们的设定不起作用，并且有可能导致程序异常。

```
void phgOptionsSetOptions(const char *str);
phgOptionsSetNoArg(char *op_name, BOOLEAN value);
phgOptionsSetInt(char *op_name, INT value);
phgOptionsSetFloat(char *op_name, FLOAT value);
phgOptionsSetKeyword(char *op_name, char *value);
phgOptionsSetString(char *op_name, char *value);
phgOptionsSetFilename(char *op_name, char *value);
phgOptionsSetHandler(char *op_name, char *value);
phgOptionsPush(void);
phgOptionsPop(void);
```

其中，phgOptionsPush 和phgOptionsPop 用于保存、恢复当前所有命令行选项的设置。

显示命令行及命令行选项

```
phgOptionsShowCmdline(void);
phgOptionsShowUsed(void);
```

- phgOptionsShowCmdline: 打印出运行程序的命令行。
- phgOptionsShowUsed: 打印出所有用户给出的命令行选项的最终值。

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*/
```

用户接口函数

PHG中生成、求解一个线性系统的过程如下：

- 调用phgSolverCreate创建解法器对象，其中需要指定作为未知量的一组自由度对象，PHG根据这些自由度对象建立自由度与线性系统未知量之间的关系。
- 调用phgSolverAddMatrixEntry或phgSolverAddMatrixEntries组装矩阵，phgSolverAddRHSEntry或phgSolverAddRHSEntries添加右端项。
- 调用phgSolverSolve求解线性系统，其中要提供一组自由度对象，它们的类型、维数必须与创建解法器的时候提供的自由度对象完全匹配。

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 *r
      INT ncols, INT *cols, FLOAT *values);
phgSolverAddRHSEntry(SOLVER *solver, INT index, FLOAT valu
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

PCG和GMRES解法器

PHG内部实现了两个迭代型求解器：PCG和GMRES，其名称 分别为SOLVER_PCG 和 SOLVER_GMRES，这两个解法器所支持的命令行参数可以通过 -help pcg 和 -help gmres获得，用户可以通过命令行选项或在程序中调用函数phgSolverSetPC来为它们指定预条件子。
例如，下述选项指定用单精度的MUMPS解法器为GMRES的预条件子：

```
-gmres_pc_type solver -gmres_pc_opts "-solver mumps \
            -mumps_precision single -mumps_symmetry unsym"
```

其中， -gmres_pc_type中的solver 参数表示在每一步迭代中调用另一个解法器作为预条件子， 而 -gmres_pc_opts选项则用来设定预条件子解法器及参数。

Outline

简单映射

MAP 用于管理连续分块存储的分布式向量，同时提供自由度到向量分量的映射关系。

```
MAP *phgMapCreateSimpleMap(GRID *g, INT m, INT M);
```

该函数定义一个全局长度为M、本进程中局部长度为m 的分布式向量。注意，调用该函数时，与网格g 相关联的所有进程必须同时调用，并且使用相同的M 参数。

简单映射

假设g 分布在p 个进程中，进程i 中的参数m 等于$m_i, i = 0, ..., p-1$，则m_i必须满足下述两个条件之一：

1. 所有m 之和等于M，即：

$$\sum_{i=0}^{p-1} m_i = M$$

此时定义的是一个全局长度为M、按进程序号分块顺序存储、进程i 中的分块大小为mi 的分布 式向量。

2. 所有进程中的m 等于M，即：

$$m_i = M, i = 0, 1, ..., p-1$$

此时定义的是一个非分布的、同时存储在所有进程中、长度为M 的向量。这种映射称为串行映 射。

基于自由度的映射

```
MAP *phgMapCreate(DOF *u, ...);
```

该函数所创建的向量分量对应于可变参数表中列出的所有自由度对象中的自由度，向量在进程 中的分布由当前子网格剖分决定。参数表必须以空指针NULL 结束。 例如:

```
DOF *u, *v;
MAP *map;
  ... ...
map = phgMapCreate(u, v, NULL);
```

创建一个MAP，它描述u 和v 中的所有自由度所构成的一个向量。

映射中的编号

- 局部编号
  基于本地自由度的编号。对简单映射而言，它等价于本地向量编号。对基于自由度的映射而言， 它等于将所有自由度对象中自由度的局部编号顺序联接起来得到的编号.
  例如，假设上例中u的自由度的局部编号为$0, ..., N-1$, v的自由度的局部编号为$0, ..., M-1$, 则映射中对应于u 的自由度的局部编号为$0, ..., N-1$, 对应于v 的自由度的局部编号为$N, ..., M+N-1$。
- 本地向量编号
  属于本地的向量分量的编号，编号范围为$0, ..., m-1$。
- 全局向量编号
  向量分量的全局编号，编号范围为$0, ..., M-1$。

编号转换

```
INT phgMapD2L(MAP *map, int dof_no, INT index);
INT phgMapE2L(MAP *map, int dof_no, SIMPLEX *e, int index)
INT phgMapL2V(MAP *map, INT index);
INT phgMapL2G(MAP *map, INT index);
```

其中dof_no 代表映射中自由度对象的序号，从0 开始，如前例中u 的序号为0, v 的序号为1。

- phgMapD2L 将自由度在自由度对象中的编号映射为MAP 中的局部编号。
- phgMapE2L 将自由度在单元中的编号(局部基函数编号) 映射为MAP 中的局部编号(参看simplest.c, phgSolverMapE2L)。
- phgMapL2V 将局部编号映射为本地向量编号。
- phgMapL2G 将局部编号映射为全局向量编号。

自由度与向量间的数据传递

```
int phgMapDofToLocalData(MAP *map, int ndof, DOF **dofs,
        FLOAT *data);
```

该函数用于将自由度对象中的数据传递给向量。

```
int phgMapLocalDataToDof(MAP *map, int ndof, DOF **dofs,
        FLOAT *data);
```

该函数用于将向量中的数据传递给与其相关联的自由度对象。

```
void phgMapVecToDofArrays(MAP *map, VEC *vec, BOOLEAN bdry
        DOF **u,...);
```

将一个多维向量的数据传递给一组自由度对象数组，每个自由度对象数组中必须包含vec->nvec个类型、维数完全一样的自由度对象。bdry 等于TRUE 时表示vec 中不包含边界自由度。该函数主要用于特征值计算的函数中。

自由度与向量间的数据传递

```
VEC *phgMapDofArraysToVec(MAP *map, int nvec, BOOLEAN bdry
        VEC **vecptr, DOF **u, ...);
```

将一组自由度对象数组的数据传递给一个多维向量。该函数主要用于特征值计算的函数中，其参数的含义与phgMapVecToDofArrays 类似。

```
VEC *phgMapDofArraysToVecN(MAP *map, int nvec, BOOLEAN bdr
        VEC **vecptr, int ndof, DOF ***dofs);
void phgMapVecToDofArraysN(MAP *map, VEC *vec, BOOLEAN bdr
        int ndof, DOF ***dofs);
```

映射的消毁

```
phgMapDestroy(MAP **map_ptr);
```

为支持同一个映射被多个矩阵、向量引用，在映射内部保存有一个引用计数器。一个映射被创建 时，它的引用计数器为0，而每当一个新的对象(矩阵、向量等) 引用该映射时，它的引用计数器会被 加1。调用phgMapDestroy 时，如果引用计数器为正，则只将引用计数器减1，只有当引用计数器为 0 时phgMapDestroy 才实际消毁该映射。一个引用了映射的对象被消毁时，它会自动对所引用的所有 映射调用phgMapDestroy。因此，phgMapDestroy 只有当一个映射没被任何其它对象引用时才会消毁 该映射。

映射的销毁

例如，下面的代码是错误的：

```
MAT *mat = phgMatCreate(......);
MAP *map = mat->rmap;
    ... ...
phgMatDestroy(&mat);
    ... ...
phgMapDestroy(&map);
```

正确的代码应该是：

```
MAT *mat = phgMatCreate(......);
MAP *map = phgMatGetRowMap(mat);
    ... ...
phgMatDestroy(&mat);
    ... ...
phgMapDestroy(&map);
```

向量(VEC)

```
VEC *phgVecCreate(GRID *g, INT m, INT M, int nvec);
VEC *phgMapCreateVec(MAP *map, int nvec);
void phgVecDestroy(VEC **vec_ptr);
void phgVecAddEntry(VEC *vec, int which, INT index, FLOAT
void phgVecAddEntries(VEC *vec, int which, INT n, INT *inc
        FLOAT *values);
void phgVecAssemble(VEC *vec);
```

其它关于VEC 的函数可用‘phgdoc phgVec’ 列出。
phgVecAddEntry 和phgVecAddEntries 中均使用向量分量的局部编号。

利用向量组装功能完成跨单元的量的叠加

利用PHG 的向量组装功能可以完成一些跨单元的量的叠加。假设自由度对象u 的类型为DOF_P0 (分片常数)，为了计算u 在面上的平均值，可以通过一个向量，对单元进行一次遍历完成。

```
static DOF_TYPE type = {... ..., 0, 0, 1, 0};
DOF *a; /* 存储面上平均值的自由度对象*/
MAP *m; VEC *v; SIMPLEX *e; int i; FLOAT d;
a = phgDofNew(u->g, &type, 1, "average", DofNoAction);
m = phgMapCreate(a, NULL);
v = phgMapCreateVec(m, 1);
phgVecDisassemble(v);
ForAllElements(g, e) {
    for (i = 0; i < NFace; i++) {
        d = *DofElementData(u, e->index);
```

利用向量组装功能完成跨单元的量的叠加

```
            if (边界面) {
                if (!子网格拥有该面) continue;
                phgVecAddEntry(v, 0, e->faces[i], d);
            } else
                phgVecAddEntry(v, 0, e->faces[i], 0.5 * d);
        }
    }
    phgVecAssemble(v);
    phgMapVecToDofArrays(m, v, FALSE, a, NULL);
    phgVecDestroy(&v); phgMapDestroy(&m);
```

矩阵(MAT)

```
MAT *phgMapCreateMat(MAP *rmap, MAP *cmap);
void phgMatDestroy(MAT **Mat);
MAP *phgMatGetRowMap(MAT *mat);
MAP *phgMatGetColumnMap(MAT *mat);
phgMatAddEntry phgMatAddEntries
phgMatAddGlobalEntry phgMatAddGlobalEntries
phgMatAddGLEntry phgMatAddGLEntries
phgMatAddLGEntry phgMatAddLGEntries
phgMatSetEntry phgMatSetEntries
phgMatSetGlobalEntry phgMatSetGlobalEntries
phgMatSetGLEntry phgMatSetGLEntries
phgMatSetLGEntry phgMatSetLGEntries
void phgMatAssemble(MAT *mat);
MAT *phgMatRemoveBoundaryEntries(MAT *mat);
```

与映射类似，MAT 中保存一个引用计数(主要供不同解法器引用同一矩阵)。只有当引用计数为0时phgMatDestroy 才消毁矩阵对象。

Dirichlet 边界条件

如果一个矩阵对象中的BOOLEAN 成员handle_bdry_eqns 的值为TRUE，则phgMatAssemble 函数会根据自由度对象中的DB_mask 成员确定哪些元素对应于Dirichlet 边界自由度，在组装前将这些元素对应的行、列保存在其它地方，然后对矩阵中的这些行列进行对角化处理，这样的处理有助于保持矩阵的对称性。

phgSolverAssembleRHS 函数在组装右端项之前利用保存的边界行列先将对应于Dirichlet 边界自由度的未知量求解出来，并相应更新方程的右端项。

一个普通矩阵的handle_bdry_eqns 成员的值通常为FALSE，而通过函数phgSolverCreate 所创建的矩阵的handle_bdry_eqns 成员的值缺省为TRUE。

"无矩阵" 矩阵

PHG 支持一类特殊的矩阵，称为"matrix-free" matrix，其中用户不用提供矩阵元素，而只需提供一个负责完成矩阵与向量乘积的函数(MV_FUNC)。

```
typedef int (*MV_FUNC)(MAT_OP op, MAT *A, VEC *x, VEC *y);
MAT *phgMapCreateMatrixFreeMat(MAP *rmap, MAP *cmap,
        MV_FUNC mv_func, void *mv_data, ...);
```

可变参数mv_data 等为可选参数，它们被存储在MAT 的指针数组成员mv_data 中，可用来传递 参数给MV_FUNC。

"无矩阵" 矩阵

函数MV_FUNC 应该处理MAT_OP 等于MAT_OP_N (矩阵乘以向量)、MAT_OP_T (矩阵转置乘以向量) 和MAT_OP_D (矩阵的对角线部分乘以向量) 三种情况。

```
static int mat_vec(MAT_OP op, MAT *mat, VEC *x, VEC *y)
{
    switch (op) {
        case MAT_OP_N: ...; break; /* y := A * x */
        case MAT_OP_T: ...; break; /* y := trans(A) * x */
        case MAT_OP_D: ...; break; /* y := diag(A) * x */
    }
    return 0;
}
```

MAT、VEC 与解法器

下述函数负责矩阵与解法器之间的转换:

```
SOLVER *phgMat2Solver(OEM_SOLVER *oem_solver, MAT *mat);
SOLVER *phgSolverMat2Solver(OEM_SOLVER *oem_solver, MAT *m
MAT *phgSolverGetMat(SOLVER *solver);
```

函数phgSolverGetMat 会使得矩阵的引用计数增加1, 这是它与直接访问mat 成员的主要区别。

MAT、VEC 与解法器

例：下面的调用是错误的：

```
SOLVER *solver = phgSolverCreate(......);
MAT *mat = solver->mat;
      ... ...
phgSolverDestroy(&solver);
      ... ...
phgMatDestroy(&mat);
```

MAT、VEC 与解法器

正确的调用应该是:

```
SOLVER *solver;
MAT *mat;
      ... ...
SOLVER *solver = phgSolverCreate(......);
MAT *mat = phgSolverGetMat(solver);
      ... ...
phgSolverDestroy(&solver);
      ... ...
phgMatDestroy(&mat);
```

特征值、特征向量计算

PHG 提供
与PARPACK、JDBSYM、LOBPCG、SLEPC、Trilinos/Anasazi
和PRIMME 的接口用于计算广义特征值和特征向量(本征值)。特征值、特征向量的计算采用统一接口，可在运行程序时通过命令行选项来选择任何一个可用的特征值求解器。

```
int phgEigenSolve(MAT *A, MAT *B, int n, int which,
FLOAT tau, FLOAT *evals, VEC **evecs, int *nit);
int phgDofEigenSolve(MAT *A, MAT *B, int n, int which,
FLOAT tau, int *nit, FLOAT *evals, MAP *map,
DOF **u, ...);
```

上述函数计算广义特征值问题$Ax = \lambda Bx$的n个特征对。参数which 可取为EIGEN_SMALLEST、EIGEN_LARGEST 或EIGEN_CLOSEST(最接近tau的n个)。

Outline

软件的目录结构

- include: 头文件，包括数据对象的定义，函数的接口定义等
- src: 各函数实现的源程序
- utils: 一些有用的脚本文件，phgdoc， valgrind.sh等
- examples: 程序实例
- doc: 文档
- test: 用于测试的一些程序
- ...

examples 目录

目录examples 中提供了一些简单的自适应计算程序实例，包括:

simplest.c	Poisson 方程，2 阶Lagrange 元
poisson.c	Poisson 方程，任意H^1元
eigen.c	Laplace 算子的特征值和特征向量
heat.c	热传导方程
non-smooth.c	二阶间断系数椭圆型问题
elastic.c	弹性力学方程
maxwell.c	时谐Maxwell 方程，任意阶棱单元
maxwell-eigen.c	时谐Maxwell 方程，特征值问题