三维并行自适应有限元软件平台 PHG 0.8.0 版
参考手册、使用指南

张林波
中国科学院科学与工程计算国家重点实验室

2010 年 1 月 11 日

本手册介绍并行自适应有限元软件平台 PHG (Parallel Hierarchical Grid) 的函数接口、基本使用及内部数据结构。

目　录

第一章　基本记号 **1**

1.1 四面体网格 . . . 1
1.2 单元内部顶点、边、面的编号 . . . 1
1.3 二分单元细化 . . . 1
1.4 层次网格结构 . . . 1
1.5 分布式层次网格 . . . 3
1.6 单元内编号、本地编号和全局编号 . . . 3
1.7 一些常量 . . . 4
1.8 单元及网格对象数据结构 . . . 4
1.8.1 `SIMPLEX` 结构 . . . 4
1.8.2 `GRID` 结构 . . . 5
1.8.3 `types_xxxx` 数组 . . . 6

第二章　源码、配置、编译及文件格式 **9**

2.1 配置与编译 . . . 9
2.1.1 C/C++ 编译器和 MPI . . . 11
2.1.2 METIS/ParMETIS 接口 . . . 12
2.1.3 Zoltan 接口 . . . 12
2.1.4 解法器接口 . . . 12
2.1.5 Tcl/Tk、VTK . . . 12
2.1.6 BLAS 和 LAPACK 库 . . . 12
2.2 安装 . . . 13
2.3 制作 RPM 包 . . . 13
2.4 程序实例 . . . 14
2.5 Tcl/Tk 接口 . . . 14
2.6 实用程序 `phgdoc` . . . 14
2.7 用户程序的编译与链接 . . . 14
2.8 网格文件格式 . . . 15
2.8.1 ALBERT 格式 . . . 15
2.8.2 Medit 格式 . . . 16
2.8.3 边界类型 . . . 17
2.8.4 周期边界 . . . 18
2.8.5 镜像网格 . . . 18
2.9 浮点类型 . . . 19

第三章　程序实例 **21**

3.1 Poisson 方程 . . . 21
3.1.1 主程序 . . . 21

3.1.2 形成线性方程组 . . . 22
3.1.3 误差指示子的计算 . . . 22

第四章 自由度对象 25
4.1 自由度类型 . . . 25
4.1.1 粗网格到细网格插值函数 . . . 26
4.1.2 细网格到粗网格插值函数 . . . 27
4.1.3 赋值函数 . . . 28
4.1.4 基函数 . . . 29
4.1.5 基函数梯度 . . . 30
4.1.6 定义新的自由度类型 . . . 30
4.2 自由度对象数据结构 . . . 30
4.3 自由度对象的操作 . . . 31
4.4 直接访问自由度对象数据 . . . 31
4.5 特殊自由度类型 . . . 32
4.5.1 常量型自由度类型 . . . 32
4.5.2 解析型自由度类型 . . . 32
4.6 几何量自由度对象 . . . 32

第五章 数值积分 33
5.1 基本数据结构 . . . 33
5.2 基函数及其导数值的缓存 . . . 34
5.2.1 数据结构 . . . 34
5.2.2 内部接口 . . . 35
5.2.3 工作机制 . . . 36

第六章 参数控制及命令行选项 39
6.1 PHG 的命令行选项 . . . 39
6.2 列出可用的选项及帮助信息 . . . 39
6.3 传递给用户程序或第三方软件的命令行参数 . . . 39
6.4 用户自定义选项 . . . 40
6.5 在程序中获取或改变命令行选项的值 . . . 40

第七章 线性解法器接口 41
7.1 未知量及编号 . . . 41
7.2 用户接口函数 . . . 41
7.3 解法器参数的设定 . . . 42
7.4 PCG 和 GMRES 解法器 . . . 42
7.5 PETSc 解法器 . . . 43

第八章 映射、向量与矩阵 **45**
8.1 映射 . . . 45
8.1.1 简单映射 . . . 45
8.1.2 自由度映射 . . . 45
8.1.3 映射编号 . . . 45
8.1.4 映射的消毁 . . . 46
8.2 向量 . . . 46
8.2.1 向量的创建与消毁 . . . 46
8.2.2 向量的赋值与组装 . . . 46
8.2.3 自由度与向量间的数据传递 . . . 47
8.3 矩阵 . . . 47
8.3.1 添加矩阵元素 . . . 47
8.3.2 “无矩阵”矩阵 . . . 48
8.3.3 分块矩阵 . . . 48
8.3.4 矩阵的组装 . . . 48
8.4 矩阵、向量运算 . . . 49

第九章 特征值、特征向量计算 **51**

附录 A PHG 变量、函数与宏 **53**
A.1 命名规则 . . . 53
A.2 常量、全局变量 . . . 53
A.3 类型、对象 . . . 53
A.3.1 自由度类型中使用的函数接口类型 . . . 54
A.3.2 预定义的自由度类型 . . . 54
A.4 积分公式 . . . 55
A.5 映射、向量和矩阵 . . . 56
A.6 解法器 . . . 60
A.7 初始化、退出、错误处理及信息输出 . . . 61
A.8 内存管理 . . . 62
A.9 性能统计 . . . 62
A.10 命令行选项 . . . 62
A.11 网格管理 . . . 65
A.12 输入输出 . . . 70
A.13 自由度管理 . . . 71
A.14 数值积分 . . . 77
A.15 线性解法器 . . . 80
A.16 特征值及特征向量计算 . . . 83
A.16.1 常量 . . . 83
A.16.2 外部特征值解法器 . . . 83
A.16.3 接口函数 . . . 83
A.17 几何量的计算与管理 . . . 84

参考文献 85

函数、变量名索引 87

名词索引 93

插图目录

1.1 四面体单元二分细化 1
1.2 子单元类型及顶点编号 2
1.3 层次网格的二叉树结构，红色表示构成当前网格的叶子单元 3
1.4 分布式层次网格的树型结构 3
1.5 父单元、子单元中边的编号关系，括号外为本地编号，括号内为全局编号 6

4.1 四面体单元二分细化 27

5.1 Cache 机制数据流图 34
5.2 DOF_TYPE 中缓存数据示例 37

表格目录

4.1 插值函数中父单元自由度数据指针 27
4.2 插值函数中子单元自由度数据指针 27

A.1 phgDofMM 中的矩阵维数及运算关系 75

第一章　基本记号

1.1　四面体网格

PHG 处理的网格对象是一维、二维三角形和三维四面体协调网格。目前只实现了三维四面体网格，一、二维的情况留待将来实现。

1.2　单元内部顶点、边、面的编号

每个四面体单元中，四个顶点分别编号为 0、1、2、3，六条边分别编号为 0 (包含顶点 0–1)、1 (包含顶点 0–2)、2 (包含顶点 0–3)、3 (包含顶点 1–2)、4 (包含顶点 1–3) 和 5 (包含顶点 2–3)。四个面分别编号为 0, 1, 2, 3，其中面 i 指不包含顶点 i 的面。这些编号分别称为顶点、边和面的单元内编号

1.3　二分单元细化

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，如图 1.1 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。

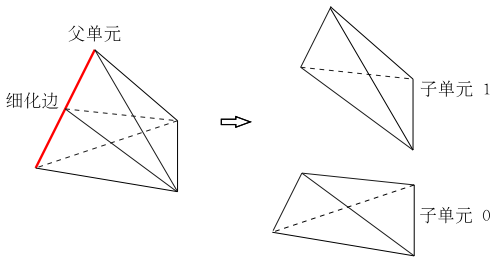


图 1.1　四面体单元二分细化

PHG 中采用的细化算法基于 [8, 7, 9, 1]。单元细化边的标注借用了自适应有限元软件包 ALBERTA [11] 中的记号，每个单元被赋予一种类型，总是将边 0 (包含顶点 0–1 的边) 作为细化边。初始网格中，单元的类型和顶点编号可以由用户指定，也可以由 PHG 自动产生。

单元细化后，包含父单元顶点 0 的子单元称为子单元 0，包含父单元顶点 1 的子单元称为子单元 1，子单元顶点的编号和类型根据父单元的顶点编号和类型确定。为了支持任意初始网格的细化，PHG 使用 5 种单元类型，分别称为 O (opposite)、M (mixed)、D (diagonal, 对应于 ALBERTA 的 type 0)、F (face, 对应于 ALBERTA 的 type 1) 和 E (edge, 对应于 ALBERTA 的 type 2)。这 5 种单元的细化流程如图 1.2 所示。

1.4　层次网格结构

四面体网格经过反复局部二分单元细化后，形成一个以单元为结点的二叉树结构。一个单元细化后便分出两条分枝，分别对应它的两个子单元。所有叶子单元构成当前网格层。PHG 保证当前网格层总是协调的。图 1.3 中用二维三角形单元给出了一个二叉树的示例 (三维四面体单元的二叉树结构与二维三角形单元相同)，在这个例子中，初始网格包含一个单元 e_0，将它细化一次得到两个新单元

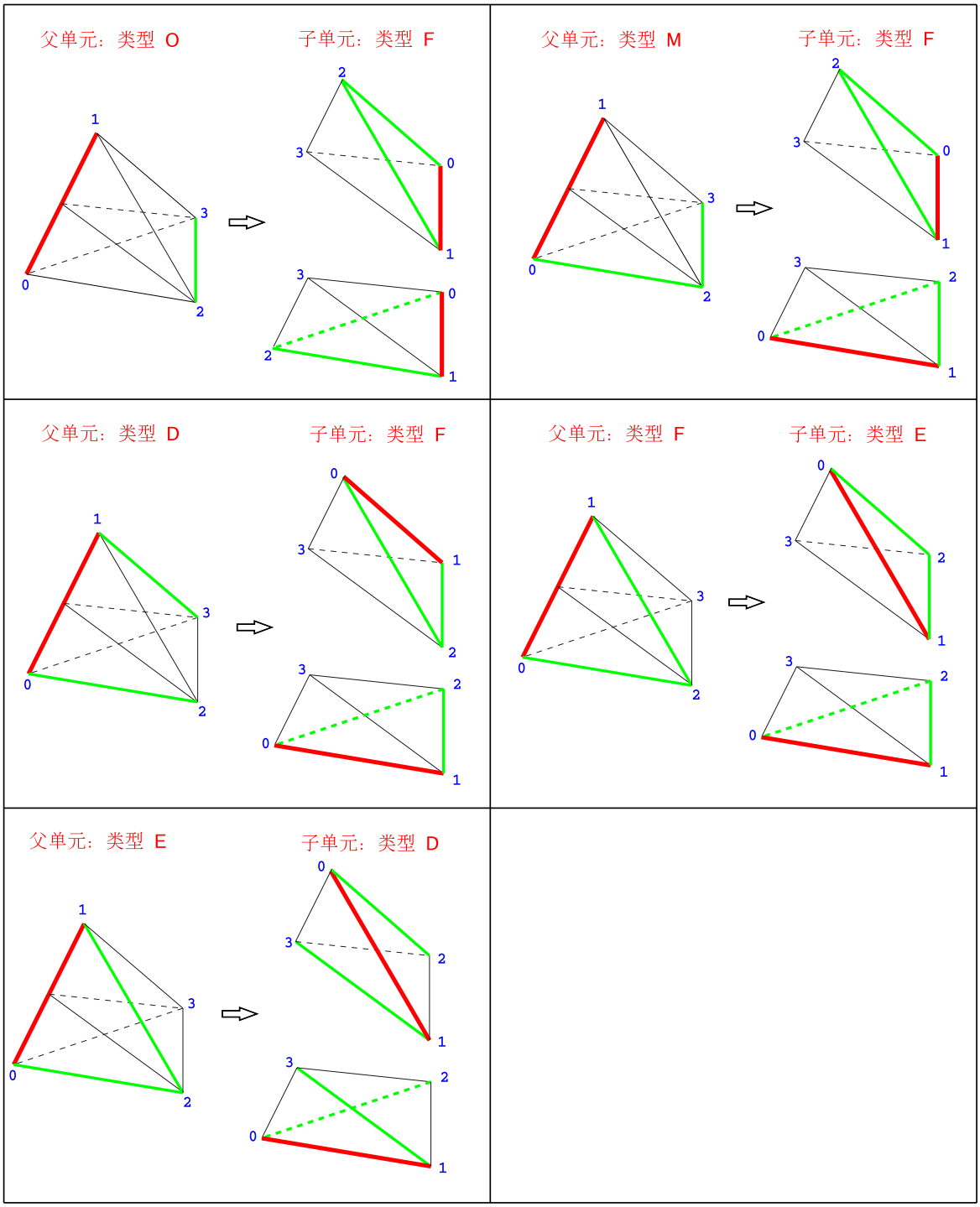


图 1.2 子单元类型及顶点编号

e_1 和 e_2，再将 e_1 和 e_2 各细化一次得到四个新单元 e_3, e_4, e_5, e_6, 最后将 e_3, e_4 和 e_5 各细化一次得到新单元 e_7, e_8, e_9, e_{10}, e_{11} 和 e_{12}，最终网格由 e_6, e_7, e_8, e_9, e_{10}, e_{11} 和 e_{12} 构成。

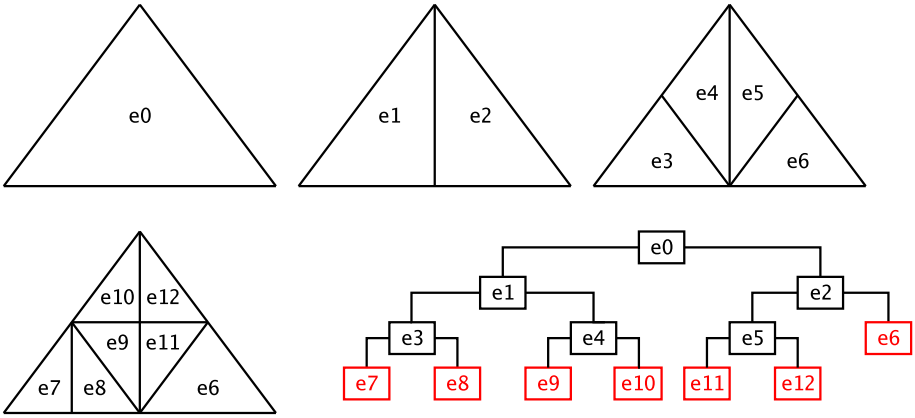


图 1.3 层次网格的二叉树结构，红色表示构成当前网格的叶子单元

后文中，当说到“网格”时一般是指由叶子单元构成的网格，不包括非叶子单元。

1.5 分布式层次网格

当用 p 个进程进行并行处理时，PHG 将网格剖分成 p 个子网格。PHG 的网格剖分基于单元进行，即将网格中单元的集合分解成 p 个互不相交的子集，每个子集构成一个子网格。相应地，描述网格层次结构的二叉树被划分成 p 个子树，每棵子树包含子网格中的叶子单元和它们所有前辈 (上层单元)。图 1.4 是将图 1.3 中的网格剖分为两个子网格 ($p = 2$) 时的子网格和子树示意图，其中第一个子网格包含叶子单元 e_7, e_8 和 e_9，第二个子网格包含叶子单元 e_6, e_{10}, e_{11} 和 e_{12}。注意，在不同子树中，叶子单元互不相同，但非叶子单元则可能有重复。这种子树结构既方便在网格层次间进行遍历，又具有并行可扩展性。

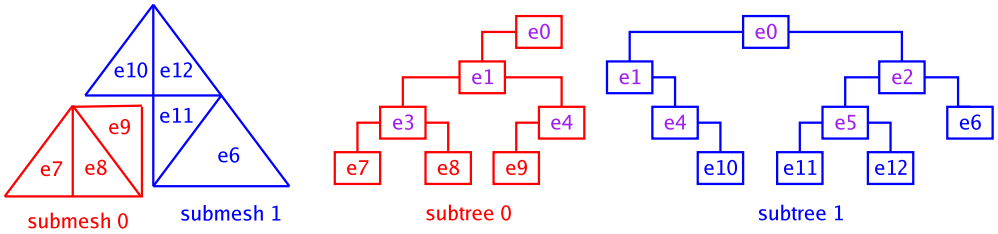


图 1.4 分布式层次网格的树型结构

1.6 单元内编号、本地编号和全局编号

PHG 中的几何对象，包括顶点、边、面、单元、自由度等，有三种不同的编号方式，分别称为单元内编号、本地编号和全局编号。单元内编号指在对象所在的单元内的编号，本地编号又称为子网格内编号或局部编号，它指对象在当前子网格中的编号，而全局编号则指对象在全局网格中的编号。所有

编号都从 0 开始。例如，顶点的单元内编号在 0 – 3 之间，本地编号在 0 – (nvert – 1) 之间 (nvert 为子树中的顶点数)，全局编号在 0 – (nvert_global – 1) 之间 (nvert_global 为全局网格中的顶点数)。

当一个单元被细化时，子单元继承属于父单元的顶点、边和面的本地和全局编号。新顶点 (即细化边的中点) 被分配一个新的编号。

子单元中包含四条新产生的边，包括两条切割边和两条新边。切割边指父单元的细化边被一分为二形成的两条边，而新边则指连接新顶点和与其相对的两个顶点的所形成的边。PHG 中约定包含细化边上全局编号较小的顶点的切割边继承父单元细化边的编号，而另一条切割边和两条新边会得到新的本地 (全局) 编号。

子单元中包含 5 个新产生的面，包括四个切割面和一个新面。切割面指包含切割边的面，它们是父单元中包含细化边的两个面被两条新边切割后产生的面，而新面则指包含两条新边的面。PHG 中约定包含细化边的两个顶点中全局编号较小的顶点的两个切割面继承父单元对应面的编号，而另外两个切割面和新面被分配新的编号。

对于单元编号，PHG 中约定由包含具有较小的全局编号的细化边上的顶点的子单元继父单元的编号，而为另一个子单元分配新的编号。

1.7 一些常量

下面是 PHG 中经常用到的一些常量，它们定义在头文件 phg.h 中。

Dim 代表空间维数，三维时为 3。

NVert、NEdge 和 NFace 分别代表一个单元中的顶点、边和面的数目。对于三维四面体单元而言，NVert = 4，NEdge = 6，NFace = 4。

1.8 单元及网格对象数据结构

1.8.1 SIMPLEX 结构

PHG 描述单元的数据结构是 SIMPLEX，其中包含如下几个主要成员：

```
typedef struct SIMPLEX_ {
    struct SIMPLEX_     *children[2];
    void                *neighbours[NFace];
    void                *parent;
    INT                 verts[NVert];
    INT                 edges[NEdge];
    INT                 faces[NFace];
    INT                 index;
    SHORT               mark;
    BTYPE               bound_type[NFace];
    ... ...
} SIMPLEX;
```

其中，children[0] 和 children[1] 分别指向两个子单元；neighbours[i] 指向第 i 个面上的邻居单元，如果面 i 为边界面则 neighbours[i] 为空指针，如果面 i 上的邻居不在本地，则 neighbours[i] 指向的是描述子网格间邻居关系的一个结构 (普通用户通常不必关心)；相应地，bound_type[i] 给出面 i 的边界类型，它是下面一些值的按位组合：INTERIOR (内部面)、DIRICHLET (Dirichlet 边界面)、NEUMANN (Neumann 边界面)、BDRY_USER0–BDRY_USER9 (用户类型 0-9)、UNDEFINED (未指定类型的边界面) 和

REMOTE (邻居在其它子网格中的内部面，亦即子网格的内边界面)；parent 指向父单元，对根单元而言 parent 为空指针。PHG 允许一个面同时包含多个标志，如 DIRICHLET 和 INTERIOR，这样在进行有限元计算时可以在内部面上指定边界类型。

verts[]、edges[]、faces[] 和 index 成员分别保存顶点、边、面和单元的本地编号，它们的全局编号可以分别调用宏 GlobalVertex、GlobalEdge、GlobalFace 和 GlobalElement 获得，例如，假设 g 为指向当前网格的指针 (GRID *)，e 为指向一个单元的指针，则 e->verts[0] 给出单元 e 中第 0 个顶点的本地编号，而 GlobalVertex(g, e->verts[0]) 则给出该顶点的全局编号 (对于非分布式的网格本地编号与全局编号是一样的)。

mark 成员用于在自适应计算中标注希望细化或粗化的单元，mark > 0 表示要求将该单元细化 mark 次，mark < 0 表示允许将该单元最多粗化 $-$mark 次。

1.8.2 GRID 结构

PHG 描述网格的数据结构是 GRID，其中包含如下几个主要成员：

```
typedef struct GRID_ {
    FLOAT       lif;            /* 负载不平衡因子 */
    COORD       *verts;         /* 顶点坐标 */
    ... ...
    BYTE        *types_vert;    /**< Types of vertices (bit flags) */
    BYTE        *types_edge;    /**< Types of edges (bit flags) */
    BYTE        *types_face;    /**< Types of faces (bit flags) */
    BYTE        *types_elem;    /**< Types of elements (bit flags) */
    ... ...
    INT         nleaf;
    INT         nvert;
    INT         nedge;
    INT         nface;
    INT         nelem;
    INT         nvert_global;
    INT         nedge_global;
    INT         nface_global;
    INT         nelem_global;
    INT         nroot;
    INT         ntree;
    ... ...
    int         rank;           /* 进程号 */
    int         nprocs;         /* 进程数 (子网格数) */
#if USE_MPI
    MPI_Comm    g->comm;        /* MPI 通信器 */
#endif
    ... ...
} GRID;
```

这里重点解释一下 verts 以及 nxxxx、nxxxx_global 成员。

verts 数组用于保存子网格中所有顶点的迪卡尔坐标，按顶点的本地编号顺序存放。例如，假设 g 为网格指针，e 为单元指针，则 e 的第 i 个顶点的 x、y、z 坐标分别为：

```
g->verts[e->verts[i]][0]
g->verts[e->verts[i]][1]
g->verts[e->verts[i]][2]
```

nleaf 给出当前子网格包含的单元数，即当前子树包含的叶子单元数。

nvert_global、nedge_global、nface_global 和 nelem_global 分别给出当前全局网格中的顶点数、边数、面数和单元数。显然，nelem_global 等于所有子网格中的 nleaf 值之和。这些量在所有进程中完全一样。

nvert、nedge、nface 和 nelem 的含义相对复杂一些，它们分别等于顶点、边、面和单元的最大本地编号加 1。对于非分布式的网格，它们分别等于叶子单元的总顶点数、总边数、总面数和总单元数。而对于分布式网格，nvert 等于子树中的顶点数，nedge、nface 和 nelem 的值则与网格的分布方式有关。这里以边的编号为例，用一个简单的二维例子来说明这些量的取法。假设初始网格包含一个单元 e_0，将其细化一次得到两个子单元 e_1 和 e_2，将这两个单元构成的网格划分为两个子网格，子网格 0 包含 e_1，子网格 1 包含 e_2，见图 1.5。图中括号外的数字代表边的本地编号，括号内的数字代表边的全局编号，两个子树中全局编号是一致的，本地编号则相互独立。子树 0 由单元 e_0 和 e_1 构成，e_0 的三条边的本地 (全局) 编号为 0(0)、1(1)、2(2)，e_1 的三条边的本地 (全局) 编号为 1(1)、2(2)、3(4)，因此 nedge 的值为 4。子树 1 由单元 e_0 和 e_2 构成，e_0 的三条边的本地 (全局) 编号为 0(0)、1(1)、2(2)，e_2 的三条边的本地 (全局) 编号为 0(0)、3(3)、4(4)，因此 nedge 的值为 5。有关父单元和子单元间的编号对应关系请参看 1.6 节中的说明。

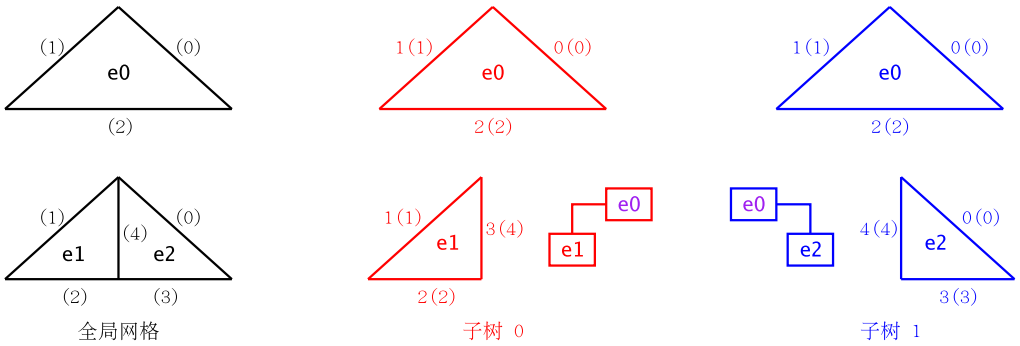


图 1.5 父单元、子单元中边的编号关系，括号外为本地编号，括号内为全局编号

这里，保留非叶子单元中顶点、边、面、和单元编号的目的是为了方便网格粗化算法的实现以及多重网格方法的构造。

1.8.3 types_xxxx 数组

GRID 组构中有四个成员 types_vert、types_edge、types_face 和 types_elem，它们是维数分别为 nvert、nedge、nface 和 nelem 的数组，分别给出当前子网格中的顶点、边、面和单元的边界属性。这些属性使用与 SIMPLEX 中的 bound_type 数组一样的标志位，并且是根据 bound_type 数组中的标志位构造的 (一个顶点或边的边界标志等于所有包含该顶点或边的面边界标志的按位“或”运算的结果，而面的边界标志则来源于 bound_type 数组)。

除了 bound_type 中使用的标志位外，types_xxxx 数组还有一个称为 OWNER 的位，表示当前子网格是否为属主。以顶点为例，一个顶点可能同时属于多个子网格，这些子网格中只有一个被指定为它的属主，在该子网格上 types_vert[i] 的 OWNER 位为 1, 在其余子网格上 OWNER 位均为 0，其中 i 表示顶点的本地编号。

当 types_vert[i] == 0 (程序中通常用常量 UNREFERENCED 表示) 时，表示顶点 i 不属于当前子网格的叶子单元，即该顶点没有被当前子网格所引用。很多情况下，在通过本地编号对顶点进行遍历

时，例如，在直接对自由度数组 (参看第四章) 循环时，应该注意跳过没被引用的顶点，在累计求和时，应该累加具有 OWNER 属性的项。例如：

```
for (i = 0; i < g->nvert; i++) {
    if (g->types_vert[i] == UNREFERENCED)
        continue;
    ... ...
}
```

对于边、面和单元而言，UNREFERENCED 具有相同的含义。

第二章　源码、配置、编译及文件格式

PHG 提供与其它一些软件的接口，包括 PETSc、HYPRE、Trilinos、MUMPS、SuperLU_Dist、SPC、LASPack、PARPACK、LOBPCG、SLEPC、Tcl/Tk、VTK、Zoltan、ParMETIS (或 METIS) 等。虽然这些软件都是可选的，但最好在编译 PHG 前安装好它们以便开启 PHG 的全部功能。如果使用 Linux 操作系统，则建议安装网址 ftp://159.226.92.111/pub/RPMS 处提供的一些 RPM 包，PHG 在配置时可以自动识别用这些 RPM 包安装的软件而不必使用任何参数。

2.1　配置与编译

在 PHG 源码的顶层目录中执行：

```
./configure
gmake
```

将编译生成 PHG 的库 `libphg.a`。注意，编译 PHG 时必须使用 GNU make，否则可能出错。PHG 的配置参数在运行 `configure` 时通过选项或环境变量指定，运行“`./configure --help`”可以得到有关 `configure` 的帮助信息。

`configure` 使用的环境变量主要有下面一些：

`CC`	指定 C 编译器 (默认为 `mpicc`)
`CFLAGS`	指定 C 编译选项
`CPP`	指定 C 预处理器
`CPPFLAGS`	指定 C/C++ 预处理选项 (如“`-I/opt/include`”)
`LDFLAGS`	指定链接选项
`LIBS`	指定链接时使用的库
`CXX`	指定 C++ 编译器 (默认为 `mpiCC` 或 `mpicxx`)
`CXXFLAGS`	指定 C++ 编译选项
`FC`	指定 Fortran (90) 编译器
`FCFLAGS`	指定 Fortran (90) 编译选项
`F77`	指定 Fortran 77 编译器
`FFLAGS`	指定 Fortran 77 编译选项

其中，Fortran 编译器及选项主要是为了在使用一些基于 Fortran 的外部软件 (如 PARPACK、MUMPS 等) 确定 Fortran 函数的命名规则及链接时所需要的与 Fortran 有关的库 (`configure` 会根据这些环境变量自动确定它们)。

`configure` 使用的主要选项有 (这里列出的不完整，具体请参考命令“`./configure --help`”的输出)：

`--prefix=`目录名	指定 PHG 的安装目录 (默认为 `/usr/local`)
`--enable-rpath`	链接时用 -rpath 指定某些动态库的路径 (默认值)
`--disable-rpath`	链接时禁用 -rpath
`--disable-shared`	编译生成静态库 libphg.a (默认值)
`--enable-shared`	编译生成共享库 libphg.so
`--enable-debug`	启用程序中的调试代码 (默认值)

`--disable-debug`	禁用程序中的调试代码
`--enable-fpetrap`	当编译环境允许时开启捕获浮点异常的功能 (默认值)
`--disable-fpetrap`	禁用捕获浮点异常的功能
`--enable-tcl`	启用 Tcl 脚本接口 (默认值)
`--disable-tcl`	禁用 Tcl 脚本接口
`--with-tcl-config=`文件名	指定 `tclConfig.sh` 文件名 (用于 Tcl 配置)
`--with-tcl-libdir=`目录名	指定 Tcl 库文件所在的目录
`--with-tcl-incdir=`目录名	指定 Tcl 头文件所在的目录
`--enable-tk`	启用 Tk 脚本接口 (默认值)
`--disable-tk`	禁用 Tk 脚本接口
`--with-tk-config=`文件名	指定 `tkConfig.sh` 文件名 (用于 Tk 配置)
`--with-tk-libdir=`目录名	指定 Tk 库文件所在的目录
`--with-tk-incdir=`目录名	指定 Tk 头文件所在的目录
`--enable-vtk`	启用 VTK 接口 (默认值)
`--disable-vtk`	禁用 VTK 接口
`--with-vtk-libdir=`目录名	指定 VTK 库文件所在的目录
`--with-vtk-incdir=`目录名	指定 VTK 头文件所在的目录
`--enable-mpi`	启用 MPI 消息传递 (默认值)
`--disable-mpi`	禁用 MPI 消息传递
`--with-mpi-libdir=`目录名	指定 MPI 库文件路径
`--with-mpi-incdir=`目录名	指定 MPI 头文件路径
`--with-mpi-lib=`库	指定 MPI 库 (如 `-lmpich`)
`--enable-mpiio`	启用 MPI–2 I/O 函数
`--disable-mpiio`	禁用 MPI–2 I/O 函数 (默认值)
`--enable-metis`	启用 METIS (用于网格剖分，默认值)
`--disable-metis`	禁用 METIS
`--with-metis-lib=`库	指定 METIS 库
`--with-metis-incdir=`目录名	指定 METIS 头文件所在的目录
`--enable-parmetis`	启用 ParMETIS (用于网格重剖分，默认值)
`--disable-parmetis`	禁用 ParMETIS
`--with-parmetis-lib=`库	指定 ParMETIS 库
`--with-parmetis-incdir=`目录名	指定 ParMETIS 头文件所在的目录
`--enable-zoltan`	启用 Zoltan 接口 (用于网格剖分，默认值)
`--disable-zoltan`	禁用 Zoltan 接口
`--with-zoltan-incdir=`目录名	Zoltan 头文件所在的目录
`--with-zoltan-libdir=`目录名	Zoltan 库文件所在的目录
`--enable-solver`	启用解法器接口 (默认值)
`--disable-solver`	禁用解法器接口
`--enable-spc`	启用 SPC 解法器 (默认值)
`--disable-spc`	禁用 SPC 解法器
`--enable-petsc`	启用 PETSc 解法器 (默认值)

--disable-petsc	禁用 PETSc 解法器
--enable-hypre	启用 HYPRE 解法器 (默认值)
--disable-hypre	禁用 HYPRE 解法器
--with-hypre-dir=目录名	HYPRE 安装目录 (默认值为 /usr/local/hypre)
--with-hypre-libs=库	HYPRE 库 (默认值为 -lHYPRE，必要时还需指定 BLAS 和 LAPACK 库)
--enable-trilinos	启用 Trilinos 接口
--disable-trilinos	禁用 Trilinos 接口 (默认值)
--with-trilinos-dir=目录名	给出 Trilinos 安装目录，用于寻找头文件及库
--with-trilinos-incdir=目录名	Trilinos 头文件目录
--with-trilinos-libdir=目录名	Trilinos 库文件目录
--enable-trilinos-anasazi	启用 Trilinos Anasazi 特征值解法器 (默认值)
--disable-trilinos-anasazi	禁用 Trilinos Anasazi 特征值解法器
--enable-superlu	启用 SuperLU_Dist 解法器 (默认值)
--disable-superlu	禁用 SuperLU_Dist 解法器
--with-superlu-lib=库	指定 SuperLu 库
--with-superlu-incdir=目录名	指定 SuperLu 头文件所在的目录
--enable-laspack	启用 LASPack (串行) 解法器 (默认值)
--disable-laspack	禁用 LASPack (串行) 解法器
--enable-papi	启用 PAPI 接口
--disable-papi	禁用 PAPI 接口
--with-papi-lib=库	指定 PAPI 库 (名及路径，默认为 -lpapi)
--with-papi-incdir=目录	指定 PAPI 头文件目录名
--enable-gzip	启用 gzip 支持 (支持读入 .gz 类型的文件，默认值)
--disable-gzip	禁用 gzip 解压缩支持
--with-gzip=程序名	gzip 程序名 (默认为 gzip)
--enable-bzip2	启用 bzip2 支持 (支持读入 .bz2 类型的文件，默认值)
--disable-bzip2	禁用 bzip2 解压缩支持
--with-bzip2=程序名	bzip2 程序名 (默认为 bzip2)
--with-blas=库	给出 BLAS 库
--with-lapack=库	给出 LAPACK 库
--with-f77-libs[=库]	Fortran 77 库 (省略“= 库”时由 PHG 自动检测)
--with-fc-libs[=库]	Fortran 库 (省略“= 库”时由 PHG 自动检测)
--enable-long-double	使用四倍精度浮点运算 (默认使用双精度浮点运算)

2.1.1 C/C++ 编译器和 MPI

PHG 主要用 C 语言编写，目前只有 VTK 和 Trilinos 接口是 C++ 的。因此如果不需要这些接口的话，只要有 C 编译器就可以了。PHG 通过 MPI 消息传递实现并行。如果没有 MPI 支持，则只能编译产生 PHG 的串行版本。

PHG 的 configure 脚本检测 C 和 C++ 编译器时优先检查 mpicc、mpiCC、mpicxx 等命令。因此，如果您的 MPI 系统提供了这样的编译命令，则在运行 configure 时不必指定任何关于 MPI 的参数。

如果您的 MPI 系统使用了其它名称的编译器前端，可以用环境变量 CC 和 CXX 来指定它们。其它情况下，configure 会试图自行找出有关 MPI 头文件、库等信息，如果失败，则需要用 --with-mpi-libdir 和 --with-mpi-incdir 选项分别指定 MPI 库和头文件的路径，以及用 --with-mpi-lib 选项指定 MPI 的库文件，如：

```
CC=gcc CXX=g++ ./configure \
        --with-mpi-libdir=/opt/mpi/lib \
        --with-mpi-incdir=/opt/mpi/include \
        --with-mpi-lib="-lpmpich -lmpich -lpmpich -lmpich -lpthread -lrt"
```

2.1.2 METIS/ParMETIS 接口

如果开启了 METIS/ParMETIS 支持，并且 configure 检测到了相应的头文件和库，则在运行程序时可以用选项 "-partitioner metis" 来指定调用 METIS/ParMETIS 进行网格划分或重划分。

2.1.3 Zoltan 接口

如果开启了 METIS/ParMETIS 支持，并且 configure 检测到了相应的头文件和库，则在运行程序时可以用选项 "-partitioner zoltan" 来指定使用 Zoltan 提供的一些算法进行网格划分或重划分，用选项 "-zoltan_method 算法" 来指定 Zoltan 的网格剖分算法，其中"算法"可以取："rcb" (recursive coordinate bisection, 默认值)、"rib" (recursive inertial bisection)、"hsfc" (Hilbert space filling curve) 和 "hypergraph" (hyper graph，慢!)。

2.1.4 解法器接口

除了 PHG 提供的内部解法器 PCG、GMRES 等之外，用户还可以调用外部解法器来求解线性方程组。目前 PHG 支持的外部解法器有 PETSc、Trilinos、HYPRE、MUMPS、SuperLU_Dist、SPC 和 LASPack 等。在运行 configure 时可以指定启用哪些解法器接口 (必要时需要提供头文件、库的位置等)。用户程序中具体使用哪种解法器由程序决定，或是在运行程序时由命令行选项"-solver"指定。LASPack 是一个串行解法器，适合在没有 MPI 的系统上使用。

2.1.5 Tcl/Tk、VTK

PHG 提供与 Tcl/Tk 脚本语言的接口。为了使用 Tcl/Tk 脚本功能，系统中必须安装有 Tcl/Tk 及相应的开发环境。如果 configure 无法自动检测到 Tcl/Tk，则需要通过适当的选项 (--with-tcl-* 和 --with-tk-*) 来指定它们。

PHG 的 VTK 接口目前是实验性的，它要求 VTK 4.5.0-2 以上版本。运行 configure 时通常只需要指定 cmake 程序的位置就可以了。

2.1.6 BLAS 和 LAPACK 库

一些外部软件包，如 PETSc, HYPRE 和 SuperLU_Dist 等需要用到 BLAS 或 LAPACK 库，其中 PETSc 和 HYPRE 需要 LAPACK 和 BLAS, 而 SuperLU_Dist 则仅需要 BLAS。如果希望在 PHG 中同时使用这些包，则它们必须引用相同的 BLAS 和 LAPACK。

运行 configure 时如果启用了 PETSc, 则 PHG 会自动从 PETSc 中获取有关 BLAS 和 LAPACK 的信息，此时用户不用在命令行上指定 BLAS 和 LAPACK 库 (如果指定反而可能出错)。

如果没有安装或启用 PETSc, 则用户可以通过 configure 的选项来指定 BLAS 或/和 LAPACK 库。下面是一些指定 BLAS/LAPACK 库的例子：

```
./configure --disable-petsc --with-blas=-lgoto --with-lapack=-llapack
./configure --disable-petsc \
        --with-lapack="-L/opt/intel/mkl/lib/32 -lmkl_lapack -lmkl_def -lguide"
```

(注：可以用 `--with-lapack` 同时指定 BLAS 和 LAPACK 库，但不应该用 `--with-blas` 来指定 LAPACK 库)。

当没有安装或启用 PETSc 并且用户没有指定 BLAS/LAPACK 库时，`configure` 会试图自动寻找适当的 BLAS/LAPACK 库。如果找到，则显示出 (并使用) 找到的库。如果找不到，则给出一条警告信息。

如果使用 Fedora Linux 并且没有其它合适的 BLAS/LAPACK 库时，可以考虑安装 `lapack-devel` 和 `blas-devel` 包。

2.2 安装

完成编译后，在源码目录中执行

```
gmake install
gmake install-doc
```

会将 PHG 的库和头文件 (及其它一些相关文件) 安装到 `--prefix` 选项指定的目录，其中后一条命令编译、安装 PHG 的手册 `manual.pdf` (需要 CCT 中文 TEX)。

PHG 安装后的主要文件和目录结构如下：

```
/prefix/ ┌ bin/phg
         │ bin/phg_tcl
         │ lib/libphg.a
         │ include/phg.h
         │ include/phg/{config.h,utils.h,...}
         │ share/phg/Makefile.inc
         │ share/phg/phg.tcl
         │ share/phg/phg-logo.gif
         │ share/doc/phg/{manual.pdf,README,...}
         └ share/doc/phg/examples/*
```

其中 `share/doc/phg/manual.pdf` 是 PHG 的手册，`share/doc/phg/examples` 中包含一些程序实例和 `Makefile`。

2.3 制作 RPM 包

PHG 的源码发布包可以直接用来制作 RPM 包。制作 RPM 包时，为了解决依赖关系，PHG 用到的其它软件，包括 VTK、ParMETIS、Zoltan 等等，最好也用 RPM 包的形式安装，网址

ftp://159.226.92.111/pub/RPMS

处提供了这些 RPM 包。

假定 PHG 源码包文件名为 phg-x.x.x-xxxxxxxx.tar.bz2，则命令：

```
rpmbuild --nodeps -ta phg-x.x.x-xxxxxxxx.tar.bz2
```

将会生成 PHG 的源码包

/usr/src/redhat/SRPMS/phg-x.x.x.src.rpm

和二进制包

/usr/src/redhat/RPMS/i386/phg-x.x.x.i386.rpm

由于 PHG 依赖于系统中安装的许多其它软件，因此所生成的二进制包通常只适合于安装在编译产生它的机器上。如果希望用 RPM 包在其它机器上安装，则需要在那些机器上重新对源码包进行编译，编译命令为：rpmbuild --nodeps --rebuild phg-x.x.x.src.rpm。

PHG 的 RPM 包的默认安装路径为 /usr/local。由于 PHG 的 RPM 包是 relocatable 的，可以在安装时指定其它安装路径，如：rpm --prefix=/usr -ivh phg-x.x.x.i386.rpm。

2.4 程序实例

examples 目录中包含一些有限元程序实例 (simplest.c、poisson.c、maxwell.c 等)。成功编译后会生成它们的可执行文件，可以运行它们来测试 PHG 的编译是否正确 (运行时可以用“-help all”选项得到它们支持的命令行选项)。

2.5 Tcl/Tk 接口

如果成功检测到 Tcl/Tk 和 VTK，“gmake all”命令将编译生成程序 phg_tcl，它是一个扩展的 Tcl/Tk 解释器，其中集成了 VTK 和 PHG 的 (部分) 功能。phg.tcl 是一个 phg_tcl 脚本实例，它实现了一个简单的图形界面，用法如下：

```
/prefix/share/phg/phg.tcl [网格文件]
```

(将“/prefix”替换成 PHG 的安装路径)。其中“网格文件”是一个 ALBERTA 或 Medit 格式的网格文件。另外，也可以通过 Shell 脚本 /prefix/bin/phg 来运行该界面。

2.6 实用程序 phgdoc

PHG 提供一条简单帮助命令 phgdoc 用于查询 PHG 函数的原型及宏。phgdoc 是一个简单的 Shell 脚本，它根据命令行参数在 PHG 的头文件中搜索相应的函数或宏名称并显示出来。例如：

```
% phgdoc phgImport
BOOLEAN phgImport(GRID *g, const char *filename, BOOLEAN distr);
% phgdoc phgDofCurl
#define phgDofCurl(src, dest, newtype, name) \
        phgDofCurl_(src, dest, newtype, name, __FILE__, __LINE__)
```

2.7 用户程序的编译与链接

假设用户源程序文件名为 mycode.c，只需在 Makefile 中加入下面一行：

```
include 目录/Makefile.inc
```

然后执行“gmake mycode”命令，便可编译生成可执行文件 mycode。其中“目录”是文件 Makefile.inc 所在的目录，它可以是 PHG 的源码目录, 也可以是 “/prefix/share/phg”。

用户程序的编译、链接可参考 /prefix/share/doc/phg/examples/Makefile。其中“/prefix”代表 PHG 的安装路径。

2.8 网格文件格式

PHG 通过函数 phgImport 可以从多种格式的网格文件中导入初始网格。目前，PHG 只能导入非分布的初始网格。本节简要介绍 PHG 所支持的网格文件格式。对于其它格式，可以通过将其转换为 PHG 所支持的格式来导入到 PHG 中，这种转换往往非常简单，一些情况下甚至用一个 Shell 脚本就能实现。

2.8.1 ALBERT 格式

这是 ALBERT 1.0 的初始网格文件格式，ALBERT 称之为 macro triangulations。ALBERT 中读、写它们的函数分别是 read_macro() 和 write_macro()。其文件结构如下：

```
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)

vertex coordinates:
顶点0坐标
顶点1坐标
... ...
顶点nv-1坐标

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

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
```

PHG 要求“求解问题的维数”和“空间维数”均等于 3。“vertex coordinates”中每行包含三个浮点数，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 (从 0 开始)。“element boundaries”中每行包含四个整数，给出相应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 表示区域内部。“element type”顺序给出每个单元的细化类型，0 表示 DIAGONAL，1 表示 FACE，2 表示 EDGE，PHG 中增加的两个类型分别用 3 (MIXED) 和 4 (OPPOSITE) 表示。“element neighbours”给出每个单元四个邻居的单元编号，-1 表示边界面，PHG 忽略 ALBERT 输入文件中给出的邻居关系而自行重新生成相关信息。

curved boundaries 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。每组公式中包含四个用 ‘;’ 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：

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

它表示曲面方程为 $C(x,y,z)=0$，而 $\big(P(x,y,z),\,Q(x,y,z),\,Q(x,y,z)\big)$ 则为点 (x,y,z) 投影到曲面上的坐标 (参看网格文件实例 test/sphere.dat)。输入文件中允许在公式之间分号后面换行，但分号不能省略。

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”项，则 PHG 将所有边界面的类型置为 UNDEFINED。

关于 ALBERT 输入文件格式的详细信息请参看 [11]。

2.8.2 Medit 格式

Medit [3] 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 [13]，Gmsh [5] 等，可以输出 Medit 格式的网格文件。关于 Medit 格式的细节请参看 Medit 的手册。

PHG 可以导入 Medit mesh format 格式的网格，但只读入 Vertices、Tetrahedra、Hexahedra、Triangles 和 Quadrilaterals 数据，忽略其它数据。PHG 要求输入文件中的网格是协调的。如果输入网格由六面体单元构成，PHG 自动将每个六面体单元转换为 5 个或 6 个四面体单元并保持网格的协调性。PHG 根据输入文件中的 Triangles 和 Quadrilaterals 类型来确定网格中边界面的边界类型。默认情况下，PHG 将 Medit 文件中的类型 1 转换为 DIRICHLET，类型 2 转换为 NEUMANN，其余均转换为 UNDEFINED。必要时，用户可以指定一个函数来将 Medit 文件中的边界类型转换为 PHG 的边界类型，参看 2.8.3，。

如果使用 Netgen [12] 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。

Medit 的 mesh 文件格式中，每个单元由 5 个数给出，前 4 个数是构成单元的顶点编号，PHG 将第 5 个数作为子区域编号保存在 SIMPLEX 结构的 region_mark 成员中，并且在网格加密时自动传递给新产生的单元。用户程序可以通过判断 region_mark 的值来确定单元所在的子区域。

PHG 提供了一个脚本 utils/tetgen2medit，它运行 tetgen，然后将其生成的网格转换为 Medit 格式 (tetgen 的 “-g” 选项生成的 .mesh 文件似乎缺少了边界信息)。

2.8.3 边界类型

PHG 提供的边界类型有：DIRICHLET、NEUMANN、BDRY_USER0–BDRY_USER9 和 UNDEFINED。在导入网格文件时，PHG 将网格文件中指定边界类型转换为上述类型之一或它们的组合。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(... ...);

... ...
```

上例所示的 bc_map 是 PHG 的默认边界类型转换函数，其中参数 bctype 为输入文件中的边界类型，函数返回值为相应的 PHG 边界类型，-1 表示非法或未知的边界类型。

除了指定一个边界类型转换函数来进行边界类型转换外，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。第一列中的两个数可以省略或用 “*” 代替，表示负无穷或正无穷。例如，下述边界类型转换文件：

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

表示将输入文件中 ≤ 0 的类型和类型 3 转换为 Dirichlet，类型 1 和 2 转换为 Neumann，≥ 5 的类型转换为 Undefined。边界类型转换文件中没有指定的类型一律转换为 Undefined。类型转换文件中可以用字符 “#” 引入说明，跟随在它后面的内容被忽略。如果用户程序中用 phgImportSetBdryMapFunc 指定了一个边界类型转换函数，则 PHG 将忽略选项 “-bcmap_file” 所指定的边界类型转换文件。边界类型转换文件一般用 “.bcmap” 做为扩展名。(TODO: 允许多个边界类型的组合)

当导入 Medit 格式的网格，并且用户没有给出选项 -bcmap_file 时，PHG 会自动检查与输入文件同名、扩展名为 “.bcmap” 的文件，如果该文件存在则自动做为边界类型转换文件读入。用户可以用命令选项 +auto_bcmap 来禁止 PHG 自动读入边界类型转换文件。

当导入 ALBERTA 格式的网格格式时，PHG 忽略边界类型转换函数和文件，而是直接按下表将 ALBERTA 的边界类型转换为 PHG 的边界类型：

ALBERTA 边界类型	PHG 边界类型
< 0	NEUMANN
1	DIRICHLET
2	BDRY_USER1
... ...	... ...
10	BDRY_USER9
11	BDRY_USER0
其它	UNDEFINED

此外，用户可以用命令行选项 -default_bdry_type, 或者在调用函数 phgImport 之前调用用函数 phgImportSetDefaultBdryType, 来指定默认边界类型。当一个网格面的类型为 UNDEFINED 时，如果该面在网格内部则作为内部面处理，否则其边界类型将被置为默认边界类型。

2.8.4 周期边界

用户可以调用函数 phgSetPeriodicity 设定周期边界，当周期方向与坐标方向不一致时，可以调用函数 phgSetPeriodicDirections 设定周期方向。这些函数必须在在调用函数 phgImport 导入网格之前调用。

PHG 要求周期边界面上的网格是协调的，并且任何一对周期顶点间至少要被三条边隔开 (很粗的初始网格可能会不满足该条件，此时可以预先对网格做适当加密)。

2.8.5 镜像网格

为了方便构造满足协调性要求的周期网格，utils 目录中提供了一个 mirror.c 程序，它对网格在指定空间方向上进行镜像拼接，从而形成满足周期协调性要求的网格，使用方法如下：

```
mirror [options] input_mesh output_mesh directions
```

其中 input_mesh 和 output_mesh 分别指定输入和输出网格文件名，directions 是一个由 'x'、'y' 和 'z' 构成的字符串，指定镜像的空间方向。例如：

```
mirror cube.dat out.mesh xyzx
```

表示依次对 x、y、z、x 方向做镜像 (输出网格的大小是输入网格的 16 倍)。mirror 支持下面几个选项 (可以用“mirror -help user”显示选项):

-refine_depth 整数
  镜像前对网格进行指定次数的一致加密。

-tolerance 浮点数
  用于判断坐标是否相等 (缺省为 1e-6)。

-symmetry_plane {min|max}
  指定镜像的对称面，min 表示用坐标值最小的面 (缺省值)，max 表示用坐标值最大的面。

-dir_file 文件名
  从指定的文件中读入周期方向 (用于处理周期方向不同于坐标轴的网格)。文件中应该包含 9 个浮点数，依次给出三个方向向量，三个向量不一定正交，但必须线性无关。

-output_format {medit|albert}
  指定输出格式，缺省为 Medit 格式。

`-vtk_file` **文件名**
生成指定名称的 VTK 文件。

`-opendx_file` **文件名**
生成指定名称的 OpenDX 文件。

2.9 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，如 INT，PHG 也提供了相应的 MPI 数据类型，如 PHG_MPI_INT)。

当用 phgPrintf 等函数输出 FLOAT 型变量时，建议转换为 double 输出，例如：

```
FLOAT a;
DOF *u;
... ...
phgPrintf("a = %lg, norm(u) = %lg\n", (double)a, (double)phgDofNormL2(u));
... ...
```

否则当 FLOAT 类型与 format 中的描述符不匹配时输出结果不对。

在 i686 和 x86_64 平台上，long double 实际只有 80 位精度 (仅比 double 多出约 3–4 位十进制精度)。在其它一些平台上，如 SGI Origin 3800，long double 可以达到 128 位精度 (约 32 位十进制精度)，但运算速度可能非常非常慢。

第三章　程序实例

3.1　Poisson 方程

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

$$\begin{cases} -\Delta u = f & x \in \Omega \\ u = g & x \in \partial\Omega \end{cases} \tag{3.1}$$

3.1.1　主程序

变量 g 为网格对象。DOF 对象 u_h 和 f_h 分别存放数值解和右端函数，类型为 DOF_DEFAULT (默认情况下为 DOF_P2, 即 2 阶 Lagrange 元，可在运行程序时通过命令行选项 -dof_type 设定为其它类型)。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);                          /* 初始化 PHG */
g = phgNewGrid(-1);                             /* 创建网格对象 */
phgImport(g, "cube.dat", FALSE);                /* 导入网格文件 */
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);
while (TRUE) {                                  /* 自适应循环 */
    phgBalanceGrid(g, 1.2, -1);                 /* 调整子网格数目与分布 */
    solver = phgSolverCreate(SOLVER_DEFAULT, u_h, NULL); /* 创建解法器 */
    build_linear_system(solver, u_h, f_h);      /* 形成线性方程组 */
    phgSolverSolve(solver, TRUE, u_h, NULL); /* 求解线性方程组 */
    phgSolverDestroy(&solver);                  /* 注销线性解法器 */
    grad_u = phgDofGradient(u_h, NULL, NULL, NULL); /* 计算数值解梯度 */
    estimate_error(u_h, f_h, grad_u, error);/* 计算误差指示子 */
    phgDofFree(&grad_u);                        /* 注销数值解梯度(不再需要) */
    if (满足终止条件) break;                   /* 判断是否结束计算 */
    mark_refine(est, ...);                      /* 根据误差指示子标注细化单元 */
    phgRefineMarkedElements(g);                 /* 网格局部细化 */
}
phgFreeGrid(&g);                                /* 注销网格对象 */
phgFinalize();                                  /* 退出 PHG */
```

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。

3.1.2 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 $B(I)$，$I,J=0,\dots,M-1$，M 为未知量个数，则计算过程如下：

```
ForAllElements(g, e) {                          /* 对单元进行遍历 */
    N = DofGetNBas(u_h, e);                     /* 单元基函数的个数 */
    for (i = 0; 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 ∫_e grad φ_i · grad φ_j 并累加到 A(I,J);
        }
        计算 ∫_e f φ_i 并累加到 B(I);
    }
}
```

其中，φ_i、φ_j 代表单元 e 中的局部基函数。phgSolverMapE2L(solver, 0, e, i) 计算 solver 的自由度对象 0 (即 u_h) 在单元 e 中的第 i 个未知量在方程组中的局部编号，phgDofGetElementBoundaryType 返回未知量的边界类型。$\int_e \operatorname{grad}\varphi_i\cdot\operatorname{grad}\varphi_j$ 的计算调用函数 phgQuadGradBasDotGradBas 完成，$\int_e f\varphi_i$ 的计算调用函数 phgQuadDofTimesBas 完成。程序中利用 $\int_e \operatorname{grad}\varphi_i\cdot\operatorname{grad}\varphi_j$ 对 i、j 的对称性减少计算量。

3.1.3 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。这里采用的误差指示子为：

$$\eta_e^2 = h_e^2\|\Delta u_h + f_h\|_{0,e}^2 + \sum_{f\in F(e),f\subset\Omega} h_f\|[\operatorname{grad} u_h\cdot n_f]\|_{0,f}^2$$

其中 h_e 为单元 e 的直径，$F(e)$ 为 e 的面的集合，h_f 为面 f 的直径，n_f 为面 f 的单位法向量，$[\cdot]$ 表示跨过面的跳量。具体计算代码如下：

```
DOF *jump, *residual;
jump = phgQuadFaceJump(grad_u, DOF_PROJ_DOT, NULL, -1);
residual = phgDofDivergence(grad_u, NULL, NULL, NULL); /* Δu_h */
phgDofAXPY(1., f_h, &residual);                 /* Δu_h + f_h */
ForAllElements(g, e) {
    int i;
    FLOAT eta, h;
    FLOAT diam = phgGeomGetDiameter(g, e);
```

```
        e->mark = 0;            /* clear refinement mark */
        eta = 0.0;
        /* for each face F compute [grad_u \cdot n] */
        for (i = 0; i < NFace; i++) {
            if (e->bound_type[i] & (DIRICHLET | NEUMANN))
                continue;       /* boundary face */
            h = phgGeomGetFaceDiameter(g, e, i);
            eta += *DofFaceData(jump, e->faces[i]) * h;
        }
        eta = eta*.5 + diam*diam*phgQuadDofDotDof(e, residual, residual, -1);
        *DofElementData(error, e->index) = Sqrt(eta);
    }
    phgDofFree(&jump);
```

上述代码中，首先调用函数 phgQuadFaceJump 计算 $\operatorname{grad} u_h$ 的面跳量，并存储在 DOF 对象 jump 中。接着调用函数 phgDofDivergence 计算 $\operatorname{grad} u_h$ 的散度 (即 Δu_h)，再调用函数 phgDofAXPY 将其与 f_h 相加从而得到 $\Delta u_h + f_h$。代码中通过函数 phgGeomGetDiameter 和 phgGeomGetFaceDiameter 分别得到单元直径和面直径。宏调用 DofFaceData(jump, e->faces[i]) 给出 jump 中对应于单元 e 的第 i 个面的数据地址，而宏调用 DofElementData(error, e->index) 则给出 error 中对应于单元 e 的数据地址。

第四章　自由度对象

自由度对象 (DOF) 是 PHG 的基本数据结构之一，用于描述和处理分布在网格上的各种数据。它既可用于定义有限元函数、空间，也可用于存储、处理任何其它分布在网格上的数据，例如几何信息 (面积、体积、直径、法向、Jacobian) 等。

4.1　自由度类型

自由度类型 (DOF type) 描述自由度对象的基本特征，其数据结构中的主要成员如下：

```
typedef struct DOF_TYPE_ {
    ...                                     /* 积分点基函数值缓存区 */
    const char          *name;              /* 自由度类型的名称或描述 */
    FLOAT               *points;            /* 自由度位置 (重心坐标) */
    BYTE                *orders;            /* 各个基函数的多项式次数 */
    struct DOF_TYPE_    *grad_type;         /* 梯度值的自由度类型 */

    /* 函数指针 */
    DOF_INTERP_FUNC     InterpC2F;          /* 粗网格到细网格插值函数 */
    DOF_INTERP_FUNC     InterpF2C;          /* 细网格到粗网格插值函数 */
    DOF_INIT_FUNC       InitFunc;           /* 投影函数 (Π_h) */
    DOF_BASIS_FUNC      BasFuncs;           /* 基函数 */
    DOF_BASIS_GRAD      BasGrads;           /* 基函数梯度 */

    BOOLEAN             invariant;          /* 基函数是否与单元形状无关 */
    BOOLEAN             free_after_use;     /* 是否需要自动释放 */
    SHORT               id;                 /* 自由度类型编号 */
    SHORT               nbas;               /* 一个单元中的自由度个数 */
    BYTE                order;              /* 基函数的最高多项式次数 */
    CHAR                continuity;         /* 有限元函数的连续性 */
    SHORT               dim;                /* 基函数维数 */

    SHORT               np_vert;            /* 每个顶点上的自由度个数 */
    SHORT               np_edge;            /* 每条边上的自由度个数 */
    SHORT               np_face;            /* 每个面上的自由度个数 */
    SHORT               np_elem;            /* 每个单元中的自由度个数 */
} DOF_TYPE;
```

积分点基函数值缓存区包括一组指针，用于缓存基函数、基函数梯度等在 Gauss 积分点处的函数值，参看 5.2。

`name` 给出自由度类型的名称或描述信息。

`grad_type` 给出该自由度类型所定义的函数的梯度的自由度类型，用于自动生成函数的梯度、散度、curl 等自由度对象。例如，在描述 n 阶 Lagrange 元的自由度类型中，`grad_type` 被定义为 $n-1$ 阶 discontinuous Galerkin 元的自由度类型。

`np_vert`、`np_edge`、`np_face` 和 `np_elem` 分别给出定义在顶点、边、面和单元中的自由度个数，因此，一个单元上的自由度总数为

$$\texttt{NVert} \times \texttt{np\_vert} + \texttt{NEdge} \times \texttt{np\_edge} + \texttt{NFace} \times \texttt{np\_face} + \texttt{np\_elem} \tag{4.1}$$

nbas 给出一个单元中自由度 (局部基函数) 的个数，等于式 (4.1) 的值。

invariant 说明自由度类型的局部基函数是否与单元形状无关，即它们是否在所有单元中是一样的 (如 Lagrange 元)，主要用于减少重复计算；order 给出基函数的最高多项式次数 (如果数组 orders 为非空指针，则其中分别给出各个基函数的多项式次数，数组大小为 np_vert + np_edge + np_face + np_elem)，用于选择数值积分精度；dim 给出基函数的维数，例如，对 Lagrange 元 dim = 1，而对棱单元 dim = Dim。

数组 points 顺序给出顶点、边、面和单元自由度的插值点信息，分别用 0 维、1 维、2 维和 3 维重心坐标表示。points 的总长度应该是

$$\texttt{np\_vert} + 2 \times \texttt{np\_edge} + 3 \times \texttt{np\_face} + 4 \times \texttt{np\_elem}$$

例如，对于四阶 Lagrange 元，每个顶点上有 1 个自由度，每条边上有三个自由度，每个面上有三个自由度，每个单元体内有 1 个自由度，points 数组的内容如下：

```
static FLOAT points = {
    1.,                                  /* 顶点自由度位置 */
    .75,.25, .5,.5, .25,.75,             /* 边自由度位置 */
    .5,.25,.25, .25,.5,.25, .25,.25,.5, /* 面自由度位置 */
    .25,.25,.25,.25                      /* 体自由度位置 */
};
```

当自由度类型没有插值点时，应该将 points 置为 NULL。PHG 中假定，当 points 不等于 NULL 时，表明自由度的值等于其插值点处的函数值。

当一个自由度类型通过 phgDofNew 函数被引用时，PHG 为其分配一个非负的编号，保存在 id 中，该编号随着其它自由度类型的使用与释放会随时改变。id == -1 表示该自由度类型没被使用。

PHG 内部为每个正在使用的自由度类型保存着一个引用计数，当创建一个新的自由度对象时，该对象所引用的自由度类型的引用计数会被加上 1，而当释放一个自由度对象时，该对象所引用的自由度类型的引用计数则会被减去 1，当一个自由度类型的引用计数降为 0 时表明所有引用该类型的自由度对象均已被释放，此时如果 free_after_use 的值为 TRUE 的话，PHG 会释放该自由度类型中 name 和 points 所占用的内存块以及该自由度类型本身所占用的内存块。

continuity 给出自由度类型所描述的有限元函数的连续可导性，< 0 表示函数是间断的，0 表示函数属于 C^0，1 表示函数属于 C^1，依此类推。

4.1.1 粗网格到细网格插值函数

自由度类型中的成员 InterpC2F 指向网格细化时对自由度对象进行插值的函数，其接口类型为 DOF_INTERP_FUNC，具体如下：

```
void InterpC2F(DOF *dof, SIMPLEX *e, FLOAT **parent_data, FLOAT **children_data)
```

其中 dof 为自由度对象，e 为父单元。

数组 parent_data 中包含 15 个指针，指向父单元中的自由度数据，这里每个顶点、边、面和体中的自由度数据都是连续存放的。具体地，这些指针的含义在表 4.1 中给出，表中 dof->dim 为自由度对象的维数，参看 4.2。需要注意的是，当 np_edge > 1 或 np_face > 1 时，边或面中的数据排列顺序是根据顶点的全局编号确定的，访问时必须做相应调整。图 4.1 左是顶点和边数据的编号示意。

一个单元细化后产生一个新顶点、四条新边、5 个新面和两个新单元，相应地，数组 children_data 中包含 12 个指针，分别指向这些数据块，这些指针的含义在表 4.2 中给出，图 4.1 右是新顶点和新边

表 4.1 插值函数中父单元自由度数据指针

parent_data 项	数据块大小	数据块内容
parent_data[i]	[np_vert][dof->dim]	顶点 i 上的自由度值, $i = 0, 1, 2, 3$
parent_data[$4 + i$]	[np_edge][dof->dim]	边 i 上的自由度值, $i = 0, 1, 2, 3, 4, 5$
parent_data[$10 + i$]	[np_face][dof->dim]	面 i 上的自由度值, $i = 0, 1, 2, 3, 4$
parent_data[14]	[np_elem][dof->dim]	单元体中的自由度值

表 4.2 插值函数中子单元自由度数据指针

children_data 项	数据块大小	数据块内容
children_data[0]	[np_vert][dof->dim]	新顶点上的自由度值
children_data[1]	[np_edge][dof->dim]	新顶点和老顶点 0 构成的边上的自由度值
children_data[2]	[np_edge][dof->dim]	新顶点和老顶点 1 构成的边上的自由度值
children_data[3]	[np_edge][dof->dim]	新顶点和老顶点 2 构成的边上的自由度值
children_data[4]	[np_edge][dof->dim]	新顶点和老顶点 3 构成的边上的自由度值
children_data[5]	[np_face][dof->dim]	新顶点和老顶点 0, 3 构成的面上的自由度值
children_data[6]	[np_face][dof->dim]	新顶点和老顶点 1, 3 构成的面上的自由度值
children_data[7]	[np_face][dof->dim]	新顶点和老顶点 0, 2 构成的面上的自由度值
children_data[8]	[np_face][dof->dim]	新顶点和老顶点 1, 2 构成的面上的自由度值
children_data[9]	[np_face][dof->dim]	新顶点和老顶点 2, 3 构成的面上的自由度值
children_data[10]	[np_elem][dof->dim]	包含老顶点 0 的新单元体上的自由度值
children_data[11]	[np_elem][dof->dim]	包含老顶点 1 的新单元体上的自由度值

的示意。函数 InterpC2F 的任务就是进行插值计算，并将插值结果填在 children_data 指向的这些地址处。

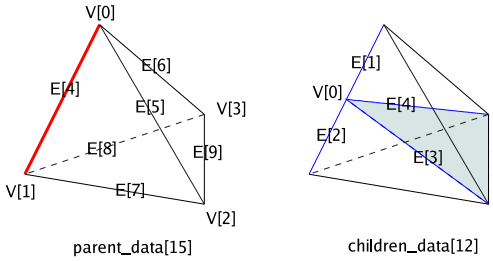


图 4.1 四面体单元二分细化

注：这里设计的接口会导致顶点、边、面处的自由度插值的一些重复计算，但这些计算在整个有限元计算中所占的计算时间比例很小。

PHG 提供了一个通用粗到细插值函数 phgDofInterpC2FGeneric 可用于任何自由度类型中的 InterC2F 成员，但其性能要差于专为特定自由度类型设计的插值函数。

4.1.2 细网格到粗网格插值函数

自由度类型中的成员 InterpF2C 指向网格粗化时对自由度对象进行插值的函数，它的接口参数与 InterpC2F 一样，但完成的操作正好相反。该函数进入时，细网格的自由度值由 children_data，以及 parent_data 中父、子单元共享的 4 个顶点、5 条边、两个面处的数据提供，函数负责计算 parent_data

中不属于子单元的一条边、两个面和单元中的自由度值。

PHG 提供了一个通用细到粗插值函数 phgDofInterpF2CGeneric 可用于任何自由度类型中的 InterpF2C 成员，但其性能通常差于专为特定自由度类型设计的插值函数。

注：对于分布式网格，子单元中可能只有一个在本地，而另一个在其它子网格，此时相应的 e->children[] 指针等于 NULL。当一个子单元在其它子网格时，需要通过通信得到它的自由度值。这些通信非常复杂，目前 PHG 没有实现它们，而是要求 InterpF2C 对这种情况进行特殊处理，即当一个子单元为空指针时只能使用 children_data[] 中属于另外一个子单元的数据，采用外插或其它手段来计算父单元的数据。

确切地，假设父单元为 e，两个子单元分别为 e_0 和 e_1。如果 e_0 和 e_1 均为非空指针，则 InterpF2C 只需要计算父单元的边 0、面 2/3 和单元自由度的值，它们分别对应 parent_data 中的第 4 (边 0)、第 12 (面 2)、第 13 (面 3) 和第 14 (单元) 项，parent_data 中的其它项和 children_data 中的所有项所指向的地址均已包含有效数据，可用于计算中。如果 e_0 为空指针，说明 e_0 不在本地，则 InterpF2C 除计算边 0、面 2/3 和单元自由度的值外，还要计算父单元的顶点 0、边 1/2 和面 1 处的自由度值，分别对应 parent_data 的第 0 (顶点 0)、第 5 (边 1)、第 6 (边 2) 和第 11 (面 1) 项，并且不能使用 children_data 中仅属于 e_0 的位置自由度值 (第 1、第 5、第 7 和第 10 项)。类似地，如果 e_1 为空指针，则 InterpF2C 除计算边 0、面 2/3 和单元自由度的值外，还需要计算父单元的顶点 1、边 3/4 和面 0 处的自由度值，分别对应 parent_data 第 1 (顶点 1)、第 7 (边 3)、第 8 (边 4) 和第 10 (面 0) 项，并且不能使用 children_data 中的第 2、第 6、第 8 和第 11 项的值。

4.1.3 赋值函数

自由度类型中的投影函数 (又叫初始化或赋值函数) InitFunc 根据函数值计算指定的自由度值，即将一个函数投影到指定的有限元空间，其接口类型为 DOF_INIT_FUNC，具体形式如下：

```
void InitFunc(DOF *dof, SIMPLEX *e, GTYPE type, int index,
          DOF_USER_FUNC userfunc, DOF_USER_FUNC_LAMBDA userfunc_lambda,
          const FLOAT *funcvalues, FLOAT *dofvalues, FLOAT **pdofvalues)
```

参数 type 指定要计算的自由度类别，VERTEX 表示顶点自由度，EDGE 表示边自由度，FACE 表示面自由度，ELEMENT 表示单元自由度。参数 index 给出顶点、边或面在单元中的编号，当 type 为 ELEMENT 时 index 值被忽略。InitFunc 计算指定位置的自由度值，并将结果放在参数 dofvalues 指向的缓冲区中返回给调用程序，该缓冲区由调用的程序提供，长度为 dof->dim × np_xxxx，数据在缓冲区中的的存放顺序为 FLOAT[np_xxxx][dof->dim] (xxxx 根据 type 的不同值分别代表 vert、edge、face 或 elem)。注意，对 Lagrange 型基函数，当 type 为 EDGE 或 FACE 并且 np_xxxx > 1 时，InitFunc 需要根据边或面的顶点的全局编号调整数据的存放顺序，对 np_xxxx 组数据 (每组包含 dof->dim 个数) 进行适当置换，以保证相邻单元间数据的一致性。

参数 userfunc、user_func_lambda 和 funcvalues 分别为两个函数指针和一个数组指针，它们给出计算函数值的函数指针或包含函数值的缓冲区地址，三个指针中必须有且仅有一个为非空指针。userfunc 指向一个关于 x, y, z 的函数，其接口类型为 DOF_USER_FUNC，具体如下：

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

而 userfunc_lambda 则指向一个关于重心坐标的函数，其接口类型为 DOF_USER_FUNC_LAMBDA，具体如下：

```
void userfunc_lambda(DOF *dof, simplex *e, int bno, const FLOAT lambda[],
```

```
                                FLOAT *values)
```

两个函数均需将指定坐标处的函数值，共计 DofDim(dof) 个数，填写在由 values 所指向的缓冲区中 (DOF_USER_FUNC_LAMBDA 中的 bno 参数给出相应位置的局部基函数编号)。

当 funcvalues 为非空指针时，它指向存有预先计算好的函数值的缓冲区。此时要求自由度类型中的 points 成员是非空指针，并且各个位置的自由度值完全由该位置的函数值确定。funcvalues 中包含指定顶点、边、面或单元处的所有 np_xxxx 个位置的函数值，共计 DofDim(dof) × np_xxxx 个数，这些位置根据 points 中的数据确定。

一些基函数，如 hierarchical basis 类的基函数，在计算边、面或体自由度时，需要用到低维位置处的自由度值。例如计算边自由度时需要用到该边的两个顶点处的自由度值，计算面自由度时需要用到该面的三个顶点和三条边处的自由度值。为了避免重复计算，PHG 中的函数在调用 InitFunc 时总是在一个单元内严格按从低维到高维的顺序进行。因此，在计算一个位置的自由度时，在其上的低维位置处的自由度值已经计算完毕，可以直接引用。如果参数 pdofvalues 为空指针，表明所需的低维位置处的自由度值可以从 dof->data 中得到。如果参数 pdofvalues 为非空指针，则表明 dof->data 中的数据不可用，需要从 pdofvalues 所指向的地址获得所需的低维位置处的自由度值。pdofvalues 中包含 15 个指针，分别指向 4 个顶点、6 条边、4 个面和体内的自由度数据，其含义与 InterpC2F 函数中的 parent_data 参数类似 (参看 4.1.1)。目前，pdofvalues != NULL 的情况仅用于通用插值函数中 (如 phgDofInterpC2FGeneric、phgDofInterpF2CGeneric)。

对于插值型的有限元基函数，如 Lagrange 元，DG 元等，其自由度的值正好就是相应点处的函数值。对这类基函数 PHG 提供了一个通用赋值函数 phgDofInitFuncPoint 作为自由度类型中的 InitFunc。

做为一个约定，如果 DOF_TYPE 中的 points 数组为非空指针，则 InitFunc 在调用 user_func 或 user_func_lambda 时必须严格按 points 数组中点的顺序来调用。PHG 中的一些函数，如 phgDofCopy，在实现时依据这一约定来避免一些重复计算 (如 cache 单元中的基函数值)。

注：设计 InitFunc 时可能需要调用 BasFuncs 计算基函数的值。此时，要特别注意 BasFuncs 返回的缓冲区在调用 user_func 或 user_func_lambda 之后有可能失效，因为后者可能调用 BasFuncs 来计算有限元函数值，这方面一个典型的例子是 *PHG* 的通用插值函数 *phgDofInterpC2FGeneric*。

4.1.4 基函数

自由度类型成员 BasFuncs 负责计算指定重心坐标位置的部分或全部基函数值，其接口类型为 DOF_BASIS_FUNC，接口参数如下：

```
const FLOAT *BasFuncs(DOF *dof, SIMPLEX *e, int no0, int no1,
                      const FLOAT *lambda)
```

其中 no0 和 no1 为局部基函数编号范围 (依次按照单元中顶点、边、面和体自由度的顺序编号，从 0 开始)，表示计算 no0 到 no1 - 1 之间的所有基函数，如果 no1 <= 0 则表示 no1 = dof->type->nbas。lambda[Dim + 1] 为重心坐标。该函数返回一个缓冲区地址，其中包含所指定范围的或全部基函数的值，该缓冲区由 BasFuncs 提供，通常是静态的，每次调用 BasFuncs 时缓冲区中的内容会被新值替换，或者缓冲区的地址会失效。BasFuncs 共返回 dof->type->dim * (no1 - no0) 个值，按 FLOAT[][dof->type->dim] 的顺序排列。

4.1.5 基函数梯度

自由度类型成员 BasGrads 负责计算指定重心坐标位置处部分或全部基函数关于重心坐标的梯度值，其接口类型为 DOF_BASIS_GRAD，接口参数为：

```
const FLOAT *BasGrads(DOF *dof, SIMPLEX *e, int no0, int no1,
                      const FLOAT *lambda)
```

各参数的含义与 BasFuncs 类似，函数返回值的个数正好是 BasFuncs 函数的 Dim + 1 倍，存储顺序为 FLOAT[][dof->type->dim][Dim + 1]。

4.1.6 定义新的自由度类型

定义一个新的自由度类型时，只需填写相应的数据类型，并实现相应的接口函数，可以参考 lagrange.c 中 Lagrange 元的定义，或 geom.c 中几何量的定义。接口函数中，只需提供需要用到的函数，用不到的可以不提供，直接写成 NULL 即可。

4.2 自由度对象数据结构

自由度对象数据结构中的主要成员如下：

```
typedef struct DOF_ {
    char        *name;          /* 名称或描述 */
    GRID        *g;             /* 网格对象 */
    DOF_TYPE    *type;          /* 自由度类型 */
    FLOAT       *data;          /* 存储自由度数据的缓冲区地址 */
    ... ...
    SHORT       dim;            /* 自由度对象的维数 */
} DOF;
```

一个自由度对象必须与一个网格对象相关联。一个网格对象中记录着所有与它相关联的自由度对象，当网格改变时 (如细化、粗化、重分布)，会自动更新这些自由度对象，而当网格释放时，会自动释放所有与其相关联的自由度对象。

dim 成员给出自由度对象的维数。一个自由度对象所对应的函数维数等于自由度类型的维数与自由度对象的维数之积 (dim × type->dim, 可通过宏 DofDim 得到)。

data 指向存储自由度数据的缓冲区，它的长度等于自由度对象在当前子网格中所有自由度的个数 (假设与自由度对象相关联的网格对象为 g)：

$$\begin{aligned}\texttt{dim} \times (&\texttt{g->nvert} \times \texttt{type->np\_vert} + \texttt{g->nedge} \times \texttt{type->np\_edge} + \\ &\texttt{g->nface} \times \texttt{type->np\_face} + \texttt{g->nelem} \times \texttt{type->np\_elem})\end{aligned} \tag{4.2}$$

每个进程只保存属于自己子网格部分的自由度数据。对于同时属于多个子网格的顶点、边或面自由度，它们重复存储在不同进程中并保持一致。自由度对象的数据分别按顶点、边、面和体自由度根据它们的本地编号顺序连续存放，当网格变化时 PHG 会自动对它们进行调整。确切地，一个子网格中自由度数据的存放顺序可以用下面的数组示意：

{FLOAT[g->nvert][type->np_vert][dim], FLOAT[g->nedge][type->np_edge][dim],
FLOAT[g->nface][type->np_face][dim], FLOAT[g->nelem][type->np_elem][dim]}

4.3 自由度对象的操作

自由度对象的数据结构使得我们可以对它们进行各种代数、微分、积分运算。例如，一个 dim = 1 的 n 阶 Lagrange 元所对应的函数的梯度是一个 dim = Dim 的 $n-1$ 阶 discontinuous Galerkin 元，对它的梯度取散度便得到它的 Laplacian，一个 dim = 1 的 $n-2$ 阶 DG 元。再如，对线性棱单元 (DOF_ND1) 进行微分运算 (梯度、散度、curl 等) 得到的是不同维数的 0 阶 Lagrange 元 (分片常数)。PHG 提供了一组通用的自由度对象数学运算操作，以方便各种有限元应用的开发，参看 A.13。

4.4 直接访问自由度对象数据

自由度对象的本地数据存储在一个数组中，参看 4.2。PHG 提供一组宏用于访问自由度对象数据，它们返回指定自由度数据的地址，用户可以对自由度数据直接进行读、写操作，包括：

```
DofVertexData(dof, 顶点本地编号)
DofEdgeData(dof, 边的本地编号)
DofFaceData(dof, 面的本地编号)
DofElementData(dof, 单元本地编号)
DofData(dof)
```

详见 A.13。

需要注意的是，存放自由度数据的数组中，部分边、面和单元的本地编号在子网格中是不存在的，它们的类型标志为 UNREFERENCED，当直接对自由度数据数组进行遍历时，应该注意跳过这些元素。可以调用函数 phgDofGetBoundaryType 或 phgDofGetElementBoundaryType 来获取自由度的类型标志。

另外，一个自由度可能同时存在于多个子网格中，有时需要检查自由度的 OWNER 标志来避免对同一个自由度在不同子网格中重复处理。PHG 提供一个宏 DofIsOwner，它返回自由度的 OWNER 标志位。

例如，下面一段代码计算所有自由度值的平方和：

```
    INT i, n;
    GRID *g;
    DOF *dof;
    FLOAT sum, *data;
    ... ...
    n = DofGetDataCount(dof);           /* 本地自由度数组大小 */
    data = DofGetData(dof);             /* 本地自由度数据地址 */
    sum = 0.0;
    for (i = 0; i < n; i++, data++) {
        if (!DofIsOwner(x, i))
            continue;
        sum += (*data) * (*data);
    }
#if USE_MPI
    if (g->nprocs > 1) {                /* 子网格间全局求和 */
        FLOAT tmp;
        tmp = sum;
        MPI_Allreduce(&tmp, &sum, 1, PHG_MPI_FLOAT, MPI_SUM, g->comm);
    }
#endif
```

4.5 特殊自由度类型

PHG 提供了两个特殊自由度类型，包括常量型自由度类型 DOF_CONSTANT 和解析型自由度类型 DOF_ANALYTIC。

4.5.1 常量型自由度类型

常量型自由度类型可用于存储取值为常数的函数。例如，下面的代码定义了一个表示取值为常向量 (1,2,3) 的自由度对象：

```
GRID *g;
DOF *u;
FLOAT values[] = {1., 2., 3.};
... ...
u = phgDofNew(g, DOF_CONSTANT, 3, "constant vector", DofNoAction);
phgDofSetDataByValues(u, values);
... ...
```

也可以用可变参数函数 phgDofSetDataByValuesV 来给常量型自由度对象赋值：

```
GRID *g;
DOF *u;
... ...
u = phgDofNew(g, DOF_CONSTANT, 3, "constant vector", DofNoAction);
phgDofSetDataByValuesV(u, 1.0, 2.0, 3.0);
... ...
```

PHG 允许常量型自由度对象参与数值积分运算。可以通过如宏 DofData、dof->data 指针等直接访问它的数据。对于常量型自由度对象，phgDofEval() 忽略参数 e 和 lambda。目前，PHG 不允许对常量型自由度对象进行微分运算。

4.5.2 解析型自由度类型

解析型自由度类型用于处理解析函数。一个解析型自由度对象既可以与一个基于迪卡尔坐标的函数相关联 (userfunc 成员), 也可以与一个基于重心坐标的函数相关联 (userfunc_lambda 成员)，其中 userfunc 既可以通过函数 phgDofNew 的参数设定，也可以调用函数 phgDofSetFunction 设定，而 userfunc_lambda 则只能调用函数 phgDofSetLambdaFunction 设定。自由度对象在指定点的值通过调用与之相关联的 userfunc 或 userfunc_lambda 函数得到。PHG 允许解析型自由度对象参与数值积分运算，以及通过函数 phgDofEval 来求值，但不允许对其进行微分运算，也不能做为函数 phgDofAXPY 中的 “y” 参数。

4.6 几何量自由度对象

文件 geom.c 中实现了一个特殊自由度对象，为用户提供有限元计算中经常用到的一些几何量，包括单元面的面积、法向量、直径，单元的体积、直径和重心坐标 Jacobian。geom.c 中提供了一组名为 phgGeomXxxxxx 的函数供用户调用来获取这些几何量，参看 A.17。

第五章　数值积分

有限元计算中需要计算单元基函数或其梯度、旋度等的积分。PHG 提供了一系列的数值积分函数接口（详见附录），这些积分函数均采用 Gauss 型数值积分方法 [15, 4, 10, 2, 6, 14]。

5.1　基本数据结构

数值积分最重要的部分是积分点的选取及其权重的确定。对于不同精度的求积公式，积分点的数量是不一样的。类似于 ALBERTA [11] 中的 QUAD 结构，PHG 中定义一个结构体，将给定阶的积分公式封装在结构体中，我们称该结构体为积分子（也叫积分类型）。具体定义如下：

```
typedef struct QUAD_ {
    const char  *name;
    int         dim;
    int         order;
    int         npoints;
    FLOAT       *points;
    FLOAT       *weights;
    SHORT       id;
 } QUAD;
```

结构体成员说明：

- `name`：定义积分子名称，可以由用户自己指定，如三维的四阶精度的积分可以命名为 ‘‘3D P4’’。
- `dim`：定义积分子维数，1 代表线段上的积分，2 代表三角形上的积分，3 代表四面体上的积分。
- `order`：定义积分子阶数。
- `npoints`：定义该积分子的积分点个数。
- `points`：定义该积分子的积分点在单元中的重心坐标。
- `weights`：定义每个积分点的权重。
- `id`：定义积分子的一个标识，与用户无关，初值为 -1，在 5.2 节中我们会作更详细的介绍。

PHG 中定义了一组任意阶一维（线段）积分子，二维（三角形）积分子和三维（四面体）积分子，它们的变量名为 `QUAD_[123]D_Pn`，采用的是 Legendre-Gauss 型积分公式 [10]，一些高维积分公式是利用一维 Legendre-Jacobi 积分公式通过张量积构造的。除了预定义的积分子，用户可以自定义其它类型的求积公式。例如，一维 3 阶的 Radau 积分公式可以如下定义

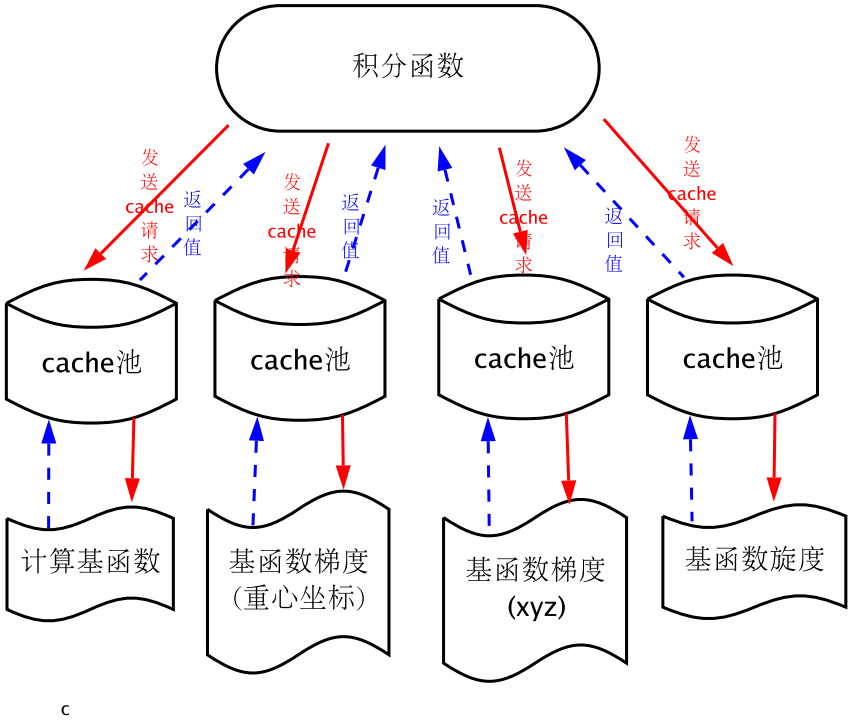
```
static FLOAT QUAD_1D_Radau_pts[ ]= {0, 1./6.};
static FLOAT QUAD_1D_Radau_wts[ ]= {.5, 1.5};
QUAD QUAD_1D_Radau3_ = {
     “1D Radau1” ,
    1,
    3,
    2,
    QUAD_1D_Radau_pts,
    QUAD_1D_Radau_wts,
```

```
    -1
};
#define QUAD_1D_Radau3 (&QUAD_1D_Radau3_)
```

PHG 的数值积分接口中均有一个 order 参数，用于指定所要求的积分精度 (多项式阶数)。如果希望由 PHG (根据基函数的阶数) 自动确定积分精度，可将该参数取为负值，或是使用预定义的常量 QUAD_DEFAULT (其值为 -1) 以改善代码的可读性。

5.2 基函数及其导数值的缓存

计算单元刚度矩阵时是以单元为对象进行的，并且需要反复用到基函数及其导数在一些相同积分点上的值。同时对于一类基函数，如 Lagrange 型基函数，它们在给定积分点上的值在所有单元中是一样的。为了避免在数值积分计算中重复计算同一个基函数在同样位置的值，我们在 PHG 中设计了一种 cache 机制来解决这一问题，它将当前单元中所有计算过的基函数及其梯度值保存在一些内部缓存区中，这些缓存区是动态生成的并且记录在自由度类型 对象的数据结构中，供所有基于同一自由度类型的自由度对象所共用，如图 5.1 所示。这种 cache 机制对用户是透明的，用户程序中只需要直接调用数值积分接口函数即可，而不必关心如何保存并重复利用已经计算过的基函数及其梯度的值。数值积分接口函数自动判断缓存区中的数据是否可用的，如果不可用则调用相关内部函数计算。



c

图 5.1 Cache 机制数据流图

5.2.1 数据结构

为了实现这种 cache 机制，我们设计了两个内部数据结构 QUAD_CACHE 和 QUAD_CACHE_LIST 用于存储当前单元上的基函数或函数值；同时在 DOF_TYPE 和 QUAD 数据结构中设置了相应的成员项。

QUAD_CACHE 数据结构定义为：

```
typedef struct {
    SIMPLEX  *e;
    FLOAT    *data;
} QUAD_CACHE;
```

该数据结构主要用于存储缓存的具体数据，其中指针 e 指向缓存的数据对应的网格单元。指针 data 指向缓存数据的缓存区。

QUAD_CACHE_LIST 数据结构定义为：

```
typedef struct {
    QUAD_CACHE  **caches;
    SHORT       n;
} QUAD_CACHE_LIST;
```

该数据结构在逻辑上可以理解为图 5.1 中的 cache 池。n 代表该池中能存储的 QUAD_CACHE 最大数量。

DOF_TYPE 结构中添加了如下一些成员项：

```
...
 void      *cache_basfunc;
 void      *cache_basgrad;
 void      *cache_gradient;
 void      *cache_curl;
 BOOLEAN   invariant;
...
```

其中，cache_basfunc、cache_basgrad、cache_gradient 和 cache_curl 是指向 QUAD_CACHE_LIST 对象的指针，分别用于缓存基函数、基函数梯度（关于重心坐标）、基函数梯度（关于笛卡尔坐标）、基函数旋度的值。invariant 变量为 TRUE 时说明该自由度类型的基函数在所有网格单元上是一样的；反之则当单元改变时需要重新计算缓存中的值。

在 QUAD 对象中定义变量 id，该变量有两种状态：

- id = -1 时，表明该类型积分尚未使用过，缓存区中没有相关数据，需要向缓存区中添加记录。
- id > -1 时，表明该类型积分需要的相关数据可能存在于缓存区中（需要进一步判断），缓存该数据的 QUAD_CACHE 对象在缓存区（即 QUAD_CACHE_LIST 对象）中的位置为 id。

5.2.2 内部接口

本节中介绍的内部接口对于用户是完全透明的。这些接口会在用户调用 PHG 中定义的积分函数时自动的被调用。

get_cache 函数返回一个有效的 QUAD_CACHE 结构

```
static inline QUAD_CACHE *get_cache(void **clist_ptr, QUAD *quad)
```

函数说明：

输入参数为指向 QUAD_CACHE_LIST 对象的二重指针 clist_ptr 和 QUAD 对象的指针 quad。该函数返回一个指向 QUAD_CACHE 对象的指针。如果积分子 quad 已存在于 QUAD_CACHE_LIST 对象中，则返回的指针指向 QUAD_CACHE_LIST 对象中的某个 QUAD_CACHE 对象；如果没有，则在 QUAD_CACHE_LIST 对象中分配新的空间，用于存储该 quad 类型积分对应的 QUAD_CACHE 对象，新分配的 QUAD_CACHE 对象的 data 变量为 NULL，返回新分配的 QUAD_CACHE 对象地址。

以下四个函数调用 get_cache 函数获取相应的 QUAD_CACHE 对象：

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

函数说明：

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。

```
static FLOAT *get_grad_lambda(SIMPLEX *e, DOF *u, int n, QUAD *quad)
```

函数说明：

参数含义同 phgQuadGetBasisValues，计算的是基函数关于重心坐标的梯度。

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

函数说明：

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)
```

函数说明：

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

5.2.3 工作机制

所有的数值积分函数都调用 get_cache 得到一个指向 QUAD_CACHE 对象的指针。当遇到如下两种情况时 QUAD_CACHE 对象中缓存的数据是可用的，反之则不可用，必须重新计算。

(1) QUAD_CACHE 对象的 data 指针不为空而且该自由度类型的 invariant 标志为 TRUE, 即该自由度类型的基函数与单元无关；

(2) QUAD_CACHE 的 data 指针不为空而且 QUAD_CACHE 缓存的值刚好是单元 e 上的。

所有被使用了的积分类型都被保存在一个 QUAD 数组 quad_list 中。不同的缓存数据（如基函数的值、基函数梯度的值等）的 QUAD_CACHE 对象都被分别保存在不同的 QUAD_CACHE_LIST 中。对于同一个单元，每个 QUAD_CACHE_LIST 可以缓存多种积分类型的数据；但是不能缓存同一积分类型在不同单元上的数据。图 5.2 给出了一个例子。（图中“3DP1”代表 3 维 1 阶积分，其它依此类推）

当一个 QUAD 积分类型被缓存时，QUAD->id 代表它在 quad_list 以及相关 QUAD_CACHE_LIST 中的位置。这样在 QUAD_CACHE_LIST 中查找需要的 QUAD_CACHE 对象时，可以避免搜索，直接使用其 QUAD->id 定位。但是该做法可能会使 QUAD_CACHE_LIST 中产生“空洞”，如图 5.2 所示。这种 cache 结构的设计基于一个基本假设，即在有限元计算中，所涉及的不同基函数及积分类型是有限的，因此这些 cache 所占用的内存空间可以忽略不计，或者是可以容忍的。

get_cache 的具体算法，

```
procedure get_cache(clist_ptr, quad)
   if(quad 未被缓存); then
       在 quad_list 中记录该 quad;
       给 quad->id 赋值;
   fi

   if clist_ptr is NULL; then
```

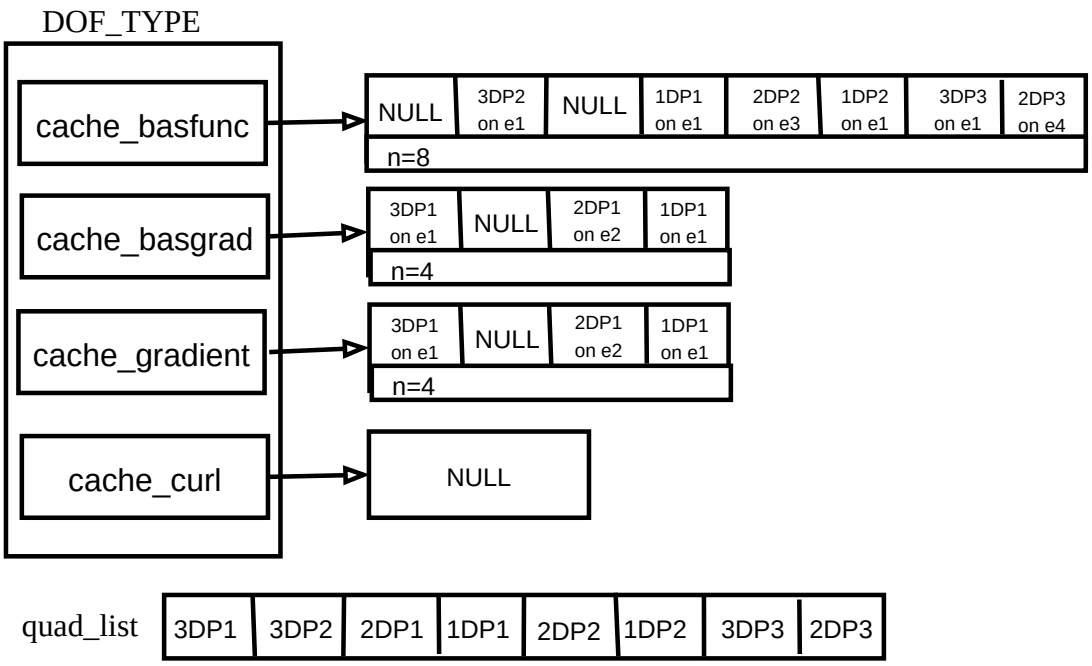


图 5.2 DOF_TYPE 中缓存数据示例

```
        初始化 clist_ptr;
    fi

    if clist->n <= quad->id; then
        将 clist->caches 的大小扩展到 quad->id + 1 项
        并且将 clist->caches[clist->n, \ldots, quad->id] 填充为 NULL;
        clist->n = quad->id;
    fi

    if clist->caches[quad->id] &=&  NULL; then
        初始化 clist->caches[quad->id];
    fi
    return cache = clist->caches[quad->id];
```

第六章　参数控制及命令行选项

6.1　PHG 的命令行选项

PHG 的程序可以通过命令行选项的形式来设定运行参数。PHG 的命令行选项根据其来源及定义方式可以分成三类：

(1) PHG 内部定义的选项

(2) PHG 调用的第三方软件 (如 PETSc) 所提供的选项

(3) 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，而第三类选项则取决于用户程序。

当用户程序调用 phgInit 时，PHG 从参数 argc 和 argv 中获得用户的命令行，并对其中的选项进行处理。从 phgInit 函数返回时，所有属于 PHG 的选项及参数将被从 argc 和 argv 中删除。

PHG 的选项一律采用 "-选项名 参数"、"-选项名" 或 "+选项名" 的形式。当一个选项在命令行中多次出现时，最后一次的值起作用。PHG 的选项分为带参数的和不带参数的两种，"+选项名" 的形式只能用于不带参数的选项，表示与 "-选项名" 相反含义的操作，通常用于取消一个前面出现过的选项。对于带参数的选项，选项名与参数间用空格或等号隔开，换言之，带参数的选项除可以写成 "-选项名 参数" 形式外，也可以写成 "-选项名=参数" 的形式。如果选项名或参数中包含空格或其它 Shell 特殊字符，则应该在命令行上用 (单或双) 引号将这些选项或参数括起来。

除可以直接在命令行上给出选项外，用户也可以将一组选项放在一个文件 (称为选项文件) 中，然后在命令行中用 "-options_file 文件名" 来调入文件中的选项。PHG 在从文件中读入选项时，以字符 '#' 开始的行被做为注释行忽略。命令行中允许出现多个 "-options_file" 选项，PHG 按照它们出现的顺序依次对每个选项文件中的选项进行处理。"-options_file" 选项也允许嵌套，即在选项文件中可以用 "-options_file" 插入其它的选项文件，PHG 会递归地处理嵌套的选项文件 (用户应该注意避免出现嵌套死循环)。

启动一个 PHG 程序时，如果在当前目录中存在文件 "可执行文件名.options"，则 PHG 会首先调入其中所定义的选项。用户可以用这种方式为一个程序设定默认选项。

用户程序中，还可以在 phgInit 之前调用 phgOptionsPreset 函数来定义一些预置的选项，这些选项会在所有其它选项之前处理。

6.2　列出可用的选项及帮助信息

PHG 提供了大量的命令行选项用于控制各种参数，我们不打算在文档中一一介绍所有的选项。运行任何一个 PHG 程序时，都可以用 "-help" 选项列出它所提供的所有选项及帮助信息。"-help" 选项要求一个类别名作为参数，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。

6.3　传递给用户程序或第三方软件的命令行参数

传递给第三方软件或用户程序的命令行参数必须通过 "-oem_options" 选项给出。目前 PHG 所支持的第三方软件中，只有 PETSc 和 HYPRE 的 Euclid 预条件子提供了标准的命令行选项，关于它

们的命令行选项请参看它们的文档。例如，可用下面的形式传递选项给 PETSc:

```
mpirun -np 2 poisson -oem_options "-ksp_type cg -pc_type bjacobi"
```

命令行上所有非选项形式的参数 (首字母不是 '-' 或 '+') 亦会被自动添加到 `-oem_options` 给出的参数中，传递给第三方软件或用户程序。此外，为了方便用户程序的处理，传递给第三方软件或用户程序的参数中，所有非选项形式的参数会被自动移到所有选项形式的参数之前。用户程序在调用 `phgInit` 后，可以通过 `argc` 和 `argv[]` 访问这些参数。

6.4 用户自定义选项

PHG 提供一组函数 `phgOptionsRegisterXXXX`，供用户程序添加自己的命令行选项。这些函数的参数中需要提供一个全局或静态变量的地址，当运行程序的命令行中给出相应的选项时，PHG 会相应修改该变量，使之对应于参数的值。

注意，用户自定义选项必须在调用 `phgInit` 之前进行。

关于用户自定义选项可以参看示例程序 `examples/poisson.c`。

6.5 在程序中获取或改变命令行选项的值

PHG 中的许多模块，例如解法器模块、网格剖分模块等，使用一些内部参数来控制其运行。这些参数通常使用静态变量，用户程序无从外部直接获取或修改它们。PHG 没有提供获取或修改这些控制参数的函数接口，因为用户程序可以方便地通过对应的命令行选项来获取或修改它们。

PHG 提供了函数 `phgOptionsGetNoArg`, `phgOptionsGetInt` 等供用户程序获取命令行选项的值，以及函数 `phgOptionsSetNoArg`, `phgOptionsSetInt`, `phgOptionsSetOptions` 等供用户程序修改命令行选项的值。这些函数提供了一种方便地设定各模块中的参数的手段，完全可以取代许多其它软件包中用于参数设定的大量函数接口。有关通过命令行选项在程序中设定模块参数的例子可参看 7.3。

第七章　线性解法器接口

PHG 提供一套统一的接口用于求解线性方程组，实际求解时，既可以调用 PHG 提供的 PCG、GMRES 等迭代法，也调用其它软件包如 PETSc、HYPRE、SuperLU、MUMPS、SPC、LASPack 等外部解法器。

PHG 的线性解法器对象为 SOLVER，用户程序通过它来定义、操作线性方程组的未知量、系数矩阵和右端项。用户通常不必关心 SOLVER 的内部结构，而只需调用 PHG 提供的线性解法器的接口函数来完成相关操作。关于解法器接口的基本使用可以参考程序 `examples/simplest.c`。

7.1　未知量及编号

PHG 的解法器中的未知量就是创建解法器时指定的自由度对象中的自由度。解法器中对未知量有三种编号方式，分别称为局部自由度编号、局部向量编号和全局向量编号。

局部自由度编号是指将子网格中的所有自由度从 0 开始按自由度对象及自由度本地编号的顺序依次编排产生的。矩阵、右端项操作函数 phgSolverAddMatrixEntry、phgSolverAddMatrixEntries、phgSolverAddRHSEntry 和 phgSolverAddRHSEntries 均使用局部自由度编号。在有限元计算形成刚度矩阵或右端项时，可以调用函数 phgSolverMapE2L 来方便地得到某个特定 DOF 对象在某个特定单元中的某个自由度的局部自由度编号，该函数的接口形式如下：

```
phgSolverMapE2L(SOLVER *solver, int dof_no, SIMPLEX *e, int index)
```

其中，`dof_no` 表示 DOF 对象在构成未知向量的全体 DOF 对象列表中 (从 0 开始) 的序号，例如，假设未知向量由三个 DOF 对象 `u`、`v`、`w` 依次构成，则 `u` 的序号为 0, `v` 的序号为 1, `w` 的序号为 2。`e` 为当前单元。`index` 为 `u` 中的自由度在单元 `e` 中的编号 (它等于 `u` 在单元 `e` 中的局部基函数的编号)。

PHG 线性解法器中包含一个系数矩阵 (`mat` 成员) 和一个右端项向量 (`rhs` 成员)。PHG 的向量按进程号的顺序分段存储在各进程中。局部向量编号指一个进程中，存储在本地的向量分量从 0 开始的顺序编号，全局向量编号则指向量的全体分量从 0 开始的顺序编号。一个向量分量的局部向量编号加上存储在本进程中的第一个向量分量的全局向量编号便是它的全局向量编号。除了上面提到的基于局部自由度编号的矩阵、右端项操作函数外，PHG 也提供了一组基于全局向量编号的矩阵、右端项操作函数，包括 phgSolverAddGlobalMatrixEntry、phgSolverAddGlobalMatrixEntries、phgSolverAddGlobalRHSEntry 和 phgSolverAddGlobalRHSEntries，它们的接口形式与基于局部自由度编号的函数完全相同，但矩阵行、列以及右端项分量采用全局向量编号。PHG 的矩阵、向量接口提供了更多的操作，参看第八章。

7.2　用户接口函数

PHG 中生成、求解一个线性系统的过程由以下几步构成：

(1) 调用 phgSolverCreate 创建解法器对象，其中需要指定做为未知量的一组 DOF 对象，PHG 根据这些 DOF 对象建立从 DOF 到线性系统解向量间的映射关系。

(2) 调用 phgSolverAddMatrixEntry 或 phgSolverAddMatrixEntries 向线性系统中添加矩阵元素，调用 phgSolverAddRHSEntry 或 phgSolverAddRHSEntries 添加右端项。PHG 的初始矩阵及右端

项均为 0，这些函数将新的元素累加到矩阵或右端项上。必要时，也可使用相应的基于全局向量编号的接口函数。

(3) (可选) 调用 phgSolverAssemble 完成线性系统的组装。

(4) 调用 phgSolverSolve 求解线性系统，其中需要提供一组 DOF 对象，它们的类型、维数必须与创建解法器对象时提供的 DOF 对象完全匹配，求解前包含初始近似解，求解完成后返回最终得到的解。

关于这些函数的具体形式参看 A.15。

7.3 解法器参数的设定

PHG 提供了一组命令行选项用于设定解法器及参数。通用的解法器选项可用 "-help solver" 显示出来，而各个特定解法器的选项则可用 "-help 解法器名" (如 "-help pcg" 等) 显示出来。

PHG 的解法器所使用的各种参数既可以直接通过相应的命令行选项设定，也可以在程序中调用 phgOptionsSetXXXX 函数来设定。一个解法器创建时，相关参数的当前值会被保存在解法器中，随后再对这些参数所做的修改不会影响到该解法器。因此，可以在创建解法器之前设定相关的参数值，同时结合使用 phgOptionsPush 和 phgOptionsPop 来保存、恢复参数原来的值，避免影响程序其它部分的运行。

例如，下面的代码创建一个作为预条件子的解法器，该解法器每次求解时仅调用一步 Hypre BoomerAMG 迭代：

```
MAT *A;
SOLVER *pc;
... ...
phgOptionsPush();
phgOptionsSetFloat("-solver_rtol", 0.);
phgOptionsSetInt("-solver_maxit", 1);
phgOptionsSetKeyword("-hypre_solver", "boomeramg");
phgOptionsSetKeyword("-hypre_pc", "none");
pc = phgMat2Solver(SOLVER_HYPRE, A);
phgOptionsPop();
... ...
```

上例中的参数设置也可以等效地写为：

```
phgOptionsSetOptions("-solver_rtol 0. -solver_maxit 1 "
                     "-hypre_solver boomeramg -hypre_pc none");
```

7.4 PCG 和 GMRES 解法器

PHG 内部实现了两个迭代型解法器：PCG (preconditioned conjugate gradient) 和 PGMRES (preconditioned generalized minimal residual)，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分别用 -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 解法器做为 GMRES 的预条件子：

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

上例中，gmres_pc_type 中的 “solver” 参数表示在 GMRES 每步迭代中调用另一个解法器做为预条件子，而 -gmres_pc_opts 选项则用来设定预条件子解法器及参数。

下面是另一个例子，它在每步 PCG 迭代中调用两步 Hypre BoomerAMG 迭代做为预条件子：

```
-pcg_pc_type solver -pcg_pc_opts "-solver hypre -solver_maxit 2 \
    -hypre_solver boomeramg -hypre_pc none"
```

(注：PHG 会自动将预条件子解法器的最大迭代次数设为 1，收敛准则设为 0)。

7.5 PETSc 解法器

PETSc 自身提供了一套完备的命令行参数、选项用于设定和控制解法器及参数。PHG 允许用户通过选项 -oem_options 来传递命令行参数给 PETSc (实际上，这组参数会被传递给所有外部解法器，但主要只对 PETSc 起作用，因为其它解法器基本上不支持通过命令行选项来设定求解参数)。

例如，下述命令行选项指定采用 PETSc 的 CG 迭代和 block Jacobi 预条件子求解：

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

函数 phgSolverSetPC 所设定的预条件子适用于 PETSc 解法器。除此之外，用户还可以用选项 “-petsc_pc_opts” 来指定将一个第三方解法器做为 PETSc 的预条件子 (当无此选项时使用 PETSc 的内部预条件子)。例如，下述选项使用 Hypre 的 BoomerAMG 做为 PETSc 的预条件子：

```
-solver petsc -petsc_pc_opts "-solver hypre -hypre_solver boomeramg \
    -hypre_pc none"
```

第八章　映射、向量与矩阵

PHG 提供一组函数用于管理连续分块存储的分布式向量，以及基于行划分、行压缩存储的稀疏矩阵。向量、矩阵在进程间的分布通过映射 (MAP) 进行描述与管理。

PHG 既可以处理由有限元函数、有限元离散形成的分布式向量、矩阵，也可以处理普通按块分布的向量及行压缩存储的矩阵。

8.1　映射

PHG 的映射 (MAP) 描述一个向量在进程间的分布，以及向量元素与自由度之间的对应关系。

8.1.1　简单映射

简单映射用于描述不与自由度对象相关联的向量。它直接指定向量的全局大小和各进程中的大小。创建简单 MAP 的函数如下：

```
MAP *phgMapCreateSimpleMap(MPI_Comm comm, INT m, INT M);
```

其中 M 给出全局长度，m 给出本进程中的长度。

当各进程中的 m 之和正好等于 M 时，该映射定义一个全局长度为 M、按进程序号分块存储、本进程中的分块的大小为 m 的分布式向量。

当所有进程中的 m 都等于 M 时，该映射定义一个非分布的、同时存储在所有进程中的向量。这种映射称为串行映射。

8.1.2　自由度映射

创建自由度映射的函数如下：

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

调用该函数时用一组自由度变量做为其参数，参数表必须以空指针 NULL 结束。所生成的 MAP 对应于参数表给出的自由度对象中的所有自由度构成的向量，向量的全局大小为所有自由度对象的自由度数之和，每个进程中的分块大小等于该进程所拥有的自由度个数 (注意，同时属于多个进程的自由度只被其中一个进程所“拥有”，由自由度所在的点、边、面或单元的 OWNER 标志确定)。

8.1.3　映射编号

我们称一个分布式向量中的元素在向量中的序号为元素的向量编号，向量中第一个元素编号为 0, 第二个元素编号为 1, 依次类推。向量元素的编号可以用全局向量编号 (编号范围为 0 — M-1)，也可以用局部向量编号 (编号范围为 0 — m-1)。将一个元素的局部向量编号加上本进程中第一个元素的全局编号便得到它的全局编号。

如果映射是由自由度定义的，则除了使用向量编号外，也可以用自由度的局部编号来标识一个元素。当映射由多个自由度对象构成时，将这些自由度对象依次连接起来构成一个数组，元素在数组中的编号称为它在映射中的的局部自由度编号，简称为局部编号 (注意与局部向量编号相区别)。PHG 提供一组函数用来转换这些编号，包括：

phgMapE2L 由一个单元中的自由度编号得到映射中的局部编号

phgMapD2L 由一个自由度对象中的自由度编号到映射中的局部编号

phgMapL2V 由元素的局部编号得到它的局部向量编号

phgMapL2G 由元素的局部编号得到它的全局向量编号

在 MAP 数据结构中，nlocal 成员为向量的本地分块大小 (对应于 m)。为了便于矩阵向量操作，向量的局部编号中还允许包含一部分不属于本地的向量分量，localsize 成员为向量的局部编号的总数，localsize ≥ nlocal, 局部编号在 [0:nlocal-1] 之间的分量为本地分量，而局部编号在 [nlocal:localsize-1] 之间的分量则为非本地分量，它们的全局向量编号由数组 O2Gmap[] 给出，该数组的大小为 localsize-nlocal。因此，假设一个元素的局部向量编号为 i，则它的全局向量编号为：

$$\begin{cases} i + n_0 & \text{如果 } i < \texttt{nlocal} \quad \text{(本地元素)} \\ \texttt{L2Gmap}[i - \texttt{nlocal}] & \text{如果 } i \geq \texttt{nlocal} \quad \text{(非本地元素)} \end{cases}$$

其中 n_0 表示本进程中的最小全局向量编号。

对于通过自由度对象创建的映射，localsize 成员等于位于本地子网格上的自由度的数目，而 nlocal 成员则等于本进程所拥有的自由度的数目。

在简单映射中，localsize = nlocal，映射中不存储不属于本地的元素，局部编号和局部向量编号是一样的。

8.1.4 映射的消毁

函数 phgMapDestroy 消毁一个映射，释放其所占用的资源。

由于一个映射通常会被多个向量、矩阵共用，为了避免重复存储，在每个映射中有一个计数器，称为映射的引用计数。一个映射被创建时其引用计数为 0。每当一个新的对象 (矩阵或向量) 引用该映射时，其引用计数会相应增加。调用 phgMapDestroy 时，如果映射的引用计数大于 0，则不会消毁该映射，而只是将它的引用计数减 1。只有当一个映射的引用计数为 0 时才会被消毁。此外，每个矩阵或向量被消毁时，亦会自动对其所引用的映射调用 phgMapDestroy。

8.2 向量

8.2.1 向量的创建与消毁

创建向量：

```
VEC *phgMapCreateVec(MAP *map, int nvec);
```

其中 nvec 指定向量的维数。

消毁向量：

```
void phgVecDestroy(VEC **vec_ptr);
```

8.2.2 向量的赋值与组装

一个向量创建后，既可以调用使用局部编号的函数 phgVecAddEntry 和 phgVecAddEntries 以及使用全局编号的函数 phgVecAddGlobalEntry 和 phgVecAddGlobalEntries 添加向量元素，也可以直接操作向量中的 data 和 offp_data 成员。完成对向量元素的赋值后，应该调用 phgVecAssemble 函数对向量进行组装。

需要注意的是，使用函数 phgVecAddXXXXXX 只能向一个尚未组装的向量添加元素，如果想向一个已经组装的向量添加元素，可以先调用 phgVecDisassemble 对向量进行卸装。PHG 中，通过函数 phgMapCreateVec 和 phgVecCreate 创建的向量默认是已组装的，因此在向其添加元素前必须先调用 phgVecDisassemble 将其卸装。

向量本地分块的数据存储在 VEC 结构的 data 成员中，数据长度为 map->nlocal × nvec。不属于本地分块，但位于本地子网格的自由度所对应的向量元素存储在 VEC 结构的 offp_data 成员中，数据长度为 (map->localsize − map->nlocal) × nvec。向量组装时，不属于本地的元素会被发送叠加到相应的进程。

向量结构中包含一个成员 mat，如果该向量是一个线性解法器中的右端项，则 mat 会指向线性解法器中的矩阵。当向量的 mat 成员为非空指针，并且 mat->handle_bdry_eqns 为 TRUE 时，向量组装程序会在组装向量之前求解 mat->bdry_eqns 与向量中的边界元素构成的小型线性方程组，并用这些方程组的解来更新向量中的相应元素。这主要是为了方便进行有限元离散时 Dirichlet 边界条件的处理。参看 8.3.4。

8.2.3 自由度与向量间的数据传递

函数 phgMapLocalDataToDof 和 phgMapDofToLocalData 用于在向量和与之相关联的自由度对象之间传递数据。而函数 phgMapVecToDofArrays 和 phgMapDofArraysToVec 则用于在多维向量与自由度对象数组之间传递数据。

8.3 矩阵

创建矩阵：

```
MAT *phgMapCreateMat(MAP *rmap, MAP *cmap);
```

rmap 给出矩阵的行映射，cmap 给出矩阵的列映射，它们可以是同一个映射 (行列具有相同大小和分布的方阵)，也可以是不同的映射。

消毁矩阵：

```
void phgMatDestroy(MAT **mat_ptr);
```

与映射类似，PHG 的矩阵对象中也有一个引用计数，以方便多个解法器共享同一个矩阵对象而不必对矩阵进行复制。消毁一个矩阵时，如果它的引用计数大于 0 则只是将引用计数减 1，只有当引用计数为 0 时才会真正消毁矩阵。

8.3.1 添加矩阵元素

PHG 提供下述函数用于向一个矩阵添加非 0 元素：

```
phgMatAddEntry          phgMatAddEntries
phgMatAddGlobalEntry    phgMatAddGlobalEntries
phgMatAddGLEntry        phgMatAddGLEntries
phgMatAddLGEntry        phgMatAddLGEntries
```

这些函数将指定的值累加到矩阵的指定元素上，它们的区别在于对矩阵的行和列分别使用了局部编号或全局编号 (注意一个新创建的矩阵的初始值为 0)。如果想用新的值替换矩阵中已有的值，则可以使用下述函数：

```
phgMatSetEntry          phgMatSetEntries
phgMatSetGlobalEntry    phgMatSetGlobalEntries
phgMatSetGLEntry        phgMatSetGLEntries
phgMatSetLGEntry        phgMatSetLGEntries
```

需要注意的是，只能向一个尚未组装的矩阵添加元素，如果想向一个已经组装的矩阵添加元素，则必须先调用 phgMatDisassemble 对其进行卸装。

PHG 中，通过函数 phgMapCreateMat* 和 phgMatCreate* 创建的矩阵默认是未组装的，可以直接向其中添加元素。

8.3.2 “无矩阵”矩阵

PHG 支持“无矩阵”(matrix-free) 形式的矩阵，即只有矩阵的大小和分布信息、没有矩阵数据的矩阵，它调用一个用户提供的函数来完成矩阵与向量的乘积操作。

```
MAT *phgMapCreateMatrixFreeMat(MAP *rmap, MAP *cmap, MV_FUNC mv_func,
                               void *mv_data0, ...);
```

MV_FUNC 是完成矩阵向量乘积的函数指针，接口形式如下：

```
typedef int (*MV_FUNC)(MAT_OP op, struct MAT_ *A, VEC *x, VEC *y);
```

其中 op 可以取 MAT_OP_N (矩阵乘向量)、MAT_OP_T (矩阵转置乘向量) 和 MAT_OP_D (矩阵的对角线乘以向量)。

可变参数 mv_data0 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。

8.3.3 分块矩阵

在 PHG 中，分块矩阵被做为一种特殊的“无矩阵”矩阵处理。当矩阵由一些相同或类似的块构成时，使用分块矩阵可以节省存储空间。PHG 的分块矩阵具有下述形式：

$$A = \begin{pmatrix} a_{1,1}\,\mathrm{op}_{1,1}(A_{1,1}) & a_{1,2}\,\mathrm{op}_{1,2}(A_{1,2}) & \cdots & a_{1,q}\,\mathrm{op}_{1,q}(A_{1,q}) \\ \vdots & \vdots & \vdots & \vdots \\ a_{p,1}\,\mathrm{op}_{p,1}(A_{p,1}) & a_{p,2}\,\mathrm{op}_{p,2}(A_{p,2}) & \cdots & a_{p,q}\,\mathrm{op}_{p,q}(A_{p,q}) \end{pmatrix}$$

其中，$A_{i,j}$ 为矩阵块，$a_{i,j}$ 为系数。目前，PHG 不支持矩阵块取转置的形式。

PHG 创建分块矩阵的函数为 phgMatCreateBlockMatrix，具体参数参看附录 A。分块矩阵的使用可参考程序实例 examples/maxwell-complex.c。

8.3.4 矩阵的组装

一个矩阵被创建并通过 phgMatAddEntry、phgMatAddEntries、phgMatAddGlobalEntry 等函数添加非 0 元素后，在使用前需要调用 phgMatAssemble 进行组装。一个矩阵组装后将不允许再直接修改其中的元素，否则可能会导致无法预料的结果。

PHG 的矩阵采用行压缩存储形式。矩阵的组装过程中不属于本地的行被发送并叠加到相应进程，同时对矩阵元素的列号进行统计及重新编号，属于本地列的元素采用 cmap 的局部向量编号，编号范围在 0 到 cmap->nlocal-1 之间。不属于本地列的元素如果它们在 cmap 中有局部编号，则使用该编号，编号范围在 cmap->nlocal 到 cmap->localsize-1 之间，否则则分配一个新的局部向量编号，编号范围在 cmap->localsize 到 localsize-1 之间，其中 MAT 的成员 localsize 等于所有在 cmap

中有编号的列向量元素 (编号范围在 0 到 cmap->localsize-1 之间) 以及所有其它被矩阵的本地部分引用的列元素 (编号范围在 cmap->localsize 到 localsize - 1 之间) 之和，显然 localsize ≥ cmap->localsize。与映射类似，矩阵中也用一个数组成员 L2Gmap 来保存不属于本地的列的全局编号，其中的前 cmap->localsize − cmap->nlocal 个编号与 cmap->L2Gmap 中完全一样。

为方便有限元计算中 Dirichlet 边界条件的处理，矩阵中有一个布尔型成员 handle_bdry_eqns，如果它的值为 TRUE，则在对矩阵进行组装前会先对本地的边界行进行处理。此时要求矩阵中所有边界行中的非 0 元素亦属于边界列，这些行、列构成一系列独立可解的小规模线性方程组，矩阵组装程序先将这些小方程组的 LU 分解保存在矩阵的 bdry_eqns 成员中，然后将所有边界行的对角线元素置为 1，非对角线元素置为 0。被保存的 LU 分解会被向量组装程序用来在向量组装前更新向量中的边界元素，参看 8.2.2。

8.4 矩阵、向量运算

函数 phgMatAXPBY 计算 $Y := \alpha X + \beta Y$，X、Y 均为矩阵，它们必须拥有相同大小和分布的行、列映射。

函数 phgMatVec 计算 $y := \alpha\,\mathrm{op}(A)x + \beta y$，$x$、$y$ 为向量，A 为矩阵，函数中的参数 op 可取为 MAT_OP_N ($\mathrm{op}(A) = A$)、MAT_OP_T ($\mathrm{op}(A) = \mathrm{trans}(A)$) 和 MAT_OP_D ($\mathrm{op}(A) = \mathrm{diag}(A)$)。当 op 为 MAT_OP_N 时，该函数要求 x->map 与 A->cmap 相同，y->map 与 A->rmap 相同。而当 op 为 MAT_OP_T 时，则要求 x->map 与 A->rmap，y->map 与 A->cmap 相同。

第九章　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。

PHG 的 JDBSYM 接口既支持原始的串行 JDBSYM 程序，也支持由戴小英完成的并行化的 JDBSYM 程序。运行 `configure` 时会自动检测所用的 JDBSYM 库是否是并行的。

PHG 的基本特征值计算接口函数如下：

```
int phgEigenSolve(MAT *A, MAT *B, int n, int which, FLOAT tau,
                  FLOAT *evals, VEC **evecs, int *nit)
```

它计算广义特征值问题 $Ax = \lambda Bx$ 的 n 个特征对 (特征值和相应的特征向量)。参数 `A` 和 `B` 给出矩阵，当 `B` 为空指针时表示 $B = I$，此时退化为标准特征值问题。`n` 为要求计算的特征对个数。`which` 指明计算哪 n 个特征对，可以取 `EIGEN_SMALLEST`、`EIGEN_LARGEST` 或 `EIGEN_CLOSEST`。`tau` 给出 shift 值。`evals` 指向保存计算得到的特征值的缓冲区 (长度至少为 `n`)。`evecs` 返回得到的特征向量。`nit` 返回计算过程中所使用的迭代次数 (其确切含义与所使用的解法器有关)。函数返回值为成功计算出的特征对的个数 ($\in [0, n]$)。

在有限元计算中更为方便的是下面基于自由度对象的接口：

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

它直接将计算出的特征向量存储在指定的自由度对象中。矩阵 `A` 和 `B` 通常应该是由 `map` 创建，然后通过函数 `phgMatRemoveBoundaryEntries` 删去边界行和列 (否则将会包含大量由 Dirichlet 边界条件产生的伪特征对)。映射 `map` 必须对应可变参数表 `u, ...` 给出的自由度对象。其余参数的含义与 `phgEigenSolve` 中完全一样。

附录 A　PHG 变量、函数与宏

A.1　命名规则

全局变量、函数的名称由 phg 前缀加一些英文单词或缩写构成，每个单词或缩写的首字母大写，其余字母小写。如 phgInit、phgRefineMarkedElements、phgRank 等。

PHG 中定义的 C 类型名由大写字母构成，如 GRID、DOF 等。

局部 (static) 变量或函数由小写字母、数字及下划线构成，例如 refine_path()。

PHG 中定义的宏、常量的名称通常与全局变量、函数的名称命名类似，但没有 phg 前缀，例如：NVert、DofElementData 等。但有时也全部用大写字母和下划线，如 TRUE、NEUMANN、DOF_P1 等。

A.2　常量、全局变量

Dim ▷ 空间维数，等于 3
NVert ▷ 一个单元中的顶点数，等于 4
NEdge ▷ 一个单元中的边数，等于 6
NFace ▷ 一个单元中的面数，等于 4
phgRank ▷ 本进程在 MPI_COMM_WORLD 中的进程号
phgNProcs ▷ MPI_COMM_WORLD 中的进程数

A.3　类型、对象

CHAR ▷ 带符号字节型变量
BYTE ▷ 无符号字节型变量
SHORT ▷ 短整型变量 (至少 2 字节)
USHORT ▷ 无符号短整型变量 (至少 2 字节)
INT ▷ 整型变量 (至少 4 字节)
UINT ▷ 无符号整型变量 (至少 4 字节)
FLOAT ▷ 浮点型变量
BOOLEAN ▷ 布尔型变量，取值 TRUE (1) 或 FALSE (0)
PHG_MPI_FLOAT ▷ FLOAT 的 MPI 数据类型
PHG_MPI_INT ▷ INT 的 MPI 数据类型
PHG_MPI_UINT ▷ UINT 的 MPI 数据类型
PHG_MPI_SHORT ▷ SHORT 的 MPI 数据类型
PHG_MPI_USHORT ▷ USHORT 的 MPI 数据类型
PHG_MPI_CHAR ▷ CHAR 的 MPI 数据类型
PHG_MPI_BYTE ▷ BYTE 的 MPI 数据类型
Pow ▷ 与 FLOAT 匹配的 pow 函数
Sqrt ▷ 与 FLOAT 匹配的 sqrt 函数
Fabs ▷ 与 FLOAT 匹配的 fabs 函数
Log ▷ 与 FLOAT 匹配的 log 函数

Exp ▷ 与 FLOAT 匹配的 exp 函数
Sin ▷ 与 FLOAT 匹配的 sin 函数
Asin ▷ 与 FLOAT 匹配的 asin 函数
Cos ▷ 与 FLOAT 匹配的 cos 函数
Acos ▷ 与 FLOAT 匹配的 acos 函数
Tan ▷ 与 FLOAT 匹配的 tan 函数
Atan ▷ 与 FLOAT 匹配的 atan 函数
GTYPE ▷ enum 类型，成员包括：VERTEX、EDGE、FACE、ELEMENT (DIAGONAL)、OPPOSITE、MIXED、UNKNOWN。前 4 个成员分别用于表示顶点或 0 维几何量 (VERTEX)、边或 1 维几何量 (EDGE)、面或 2 维几何量 (FACE)、以及体或三维几何量 (ELEMENT)。DIAGONAL、FACE、EDGE、OPPOSITE 和 MIXED 用于表示单元细化类型
BTYPE ▷ 位标志，用来表示点、边、面或体的类型，可取的值有 DIRICHLET (Dirichlet 边界)、NEUMANN (Neumann 边界)、BDRY_USER0 (用户类型 0)、、...、BDRY_USER9 (用户类型 9)、UNDEFINED (未定义边界)、INTERIOR (区域内部)、REMOTE (多个子区域共享)、OWNER (拥有者) 和 UNREFERENCED (0)。其中，UNDEFINED 表示未指定类型的边界。UNREFERENCED 表示对象没被叶子单元引用，只出现在非叶子单元中。一个对象可以同时拥有几个属性，例如一条边可以既属于 Dirichlet 边界，又属于 Neumann 边界，同时又为不同子网格所共享 (REMOTE)。一个点、边或面可以同时属于多个子网格，但只有唯一一个子网格为它的拥有者，相应的 OWNER 位在该子网格中为 1, 在其它子网格中为 0。

BTYPE 除了用于几何对象外，也用于自由度对象，其含义完全类似。
SIMPLEX ▷ 单元对象 (struct)
GRID ▷ 网格对象 (struct)
DOF ▷ 自由度 (DOF) 对象 (struct)
DOF_TYPE ▷ 自由度类型对象 (struct)
QUAD ▷ 积分公式对象 (struct)
SOLVER ▷ 求解器对象 (struct)
OEM_SOLVER ▷ 外部求解器接口对象 (struct)
COORD ▷ Dim 维 FLOAT 型数组，用于描述空间坐标

A.3.1 自由度类型中使用的函数接口类型

DOF_INTERP_FUNC ▷ 插值函数接口类型 (参看 4.1.1 和 4.1.2)
DOF_INIT_FUNC ▷ 自由度投影函数接口类型 (参看 4.1.3)
DOF_USER_FUNC ▷ 使用 x, y, z 的用户函数接口类型 (参看 4.1.3)
DOF_USER_FUNC_LAMBDA ▷ 使用重心坐标的用户函数接口类型 (参看 4.1.3)
DOF_BASIS_FUNC ▷ 计算基函数的函数接口类型 (参看 4.1.4)
DOF_BASIS_GRAD ▷ 计算基函数梯度的函数接口类型 (参看 4.1.5)

A.3.2 预定义的自由度类型

DOF_CONSTANT ▷ 特殊自由度类型，用于存储、处理常量。
DOF_ANALYTIC ▷ 特殊自由度类型，用于引用一个外部解析函数。

DOF_DEFAULT ▷ 默认自由度类型，可在运行程序时通过命令行选项 "-dof_type" 改变它的设定 (默认值为 DOF_P2)。

DOF_P0 ▷ 0 阶 Lagrange 元，分片常数，每个单元一个自由度

DOF_P1 ▷ 1 阶 Lagrange 元，连续分片线性函数，每个顶点一个自由度

DOF_P2 ▷ 2 阶 Lagrange 元，连续分片二次多项式，每个顶点、每条边各一个自由度

DOF_P3 ▷ 3 阶 Lagrange 元，连续分片三次多项式，每个顶点、每个面各一个自由度、每条边两个自由度

DOF_P4 ▷ 4 阶 Lagrange 元，连续分片四次多项式，每个顶点、每个单元各一个自由度，每条边、每个面各三个自由度

DOF_DG0 ▷ 0 阶 DG (discontinuous Galerkin) 元，等价于 DOF_P0

DOF_DGn ▷ n 阶 DG 元，$0 \le n \le 15$

DOF_ND1 ▷ 线性 Nédélec 元 (棱单元)

DOF_HFEB1 ▷ 1 阶棱单元 (= DOF_HC1)

DOF_HFEB2 ▷ 2 阶棱单元 (= DOF_HC2)

DOF_HBn ▷ n 阶 H^1 协层基 (hierarchical basis), $0 \le n \le 15$, 其中 DOF_HB0 等于 DOF_DG0

DOF_HCn ▷ n 阶 $H(\mathrm{curl})$ 协层基 (hierarchical basis), $0 \le n \le 15$, 其中 DOF_HC0 相当于 DOF_ND1

A.4 积分公式

QUAD_1D_P1 ▷ 1 阶精度一维 Gauss 积分公式 (1 点)

QUAD_1D_P2 ▷ 2 阶精度一维 Gauss 积分公式 (= QUAD_1D_P3)

QUAD_1D_P3 ▷ 3 阶精度一维 Gauss 积分公式 (2 点)

QUAD_1D_P4 ▷ 4 阶精度一维 Gauss 积分公式 (= QUAD_1D_P5)

QUAD_1D_P5 ▷ 5 阶精度一维 Gauss 积分公式 (3 点)

QUAD_1D_P6 ▷ 6 阶精度一维 Gauss 积分公式 (= QUAD_1D_P7)

QUAD_1D_P7 ▷ 7 阶精度一维 Gauss 积分公式 (4 点)

QUAD_1D_P8 ▷ 8 阶精度一维 Gauss 积分公式 (= QUAD_1D_P9)

QUAD_1D_P9 ▷ 9 阶精度一维 Gauss 积分公式 (5 点)

QUAD_1D_P10 ▷ 10 阶精度一维 Gauss 积分公式 (= QUAD_1D_P11)

QUAD_1D_P11 ▷ 11 阶精度一维 Gauss 积分公式 (6 点)

QUAD_2D_P1 ▷ 1 阶精度二维 Gauss 积分公式 (1 点)

QUAD_2D_P2 ▷ 2 阶精度二维 Gauss 积分公式 (3 点)

QUAD_2D_P3 ▷ 3 阶精度二维 Gauss 积分公式 (6 点)

QUAD_2D_P4 ▷ 4 阶精度二维 Gauss 积分公式 (6 点)

QUAD_2D_P5 ▷ 5 阶精度二维 Gauss 积分公式 (7 点)

QUAD_2D_P6 ▷ 6 阶精度二维 Gauss 积分公式 (12 点)

QUAD_2D_P7 ▷ 7 阶精度二维 Gauss 积分公式 (15 点)

QUAD_2D_P8 ▷ 8 阶精度二维 Gauss 积分公式 (16 点)

QUAD_2D_P9 ▷ 9 阶精度二维 Gauss 积分公式 (19 点)

QUAD_2D_P10 ▷ 10 阶精度二维 Gauss 积分公式 (25 点)

QUAD_3D_P1 ▷ 1 阶精度三维 Gauss 积分公式 (1 点)
QUAD_3D_P2 ▷ 2 阶精度三维 Gauss 积分公式 (4 点)
QUAD_3D_P3 ▷ 3 阶精度三维 Gauss 积分公式 (8 点)
QUAD_3D_P4 ▷ 4 阶精度三维 Gauss 积分公式 (14 点)
QUAD_3D_P5 ▷ 5 阶精度三维 Gauss 积分公式 (14 点)
QUAD_3D_P6 ▷ 6 阶精度三维 Gauss 积分公式 (24 点)
QUAD_3D_P7 ▷ 7 阶精度三维 Gauss 积分公式 (36 点)
QUAD_3D_P8 ▷ 8 阶精度三维 Gauss 积分公式 (46 点)
QUAD_3D_P9 ▷ 9 阶精度三维 Gauss 积分公式 (61 点)
QUAD_3D_P10 ▷ 10 阶精度三维 Gauss 积分公式 (73 或 81 点)

A.5 映射、向量和矩阵

MAP ▷ 映射对象，定义矩阵、向量的分布及与自由度间的对应关系。
VEC ▷ 向量对象 (顺序分块存储)。
MAT ▷ 矩阵对象 (稀疏行压缩存储)。
MV_FUNC ▷ “无矩阵”矩阵与向量乘积函数的接口类型。

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

创建基于一组自由度对象的映射。

```
MAP *phgMapCreateSimpleMap(MPI_Comm comm, INT m, INT M)
```

创建一个简单映射。

```
void phgMapDestroy(MAP **map_ptr)
```

销毁映射。

```
MAT *phgMapCreateMatrixFreeMat(MAP *rmap, MAP *cmap, MV_FUNC mv_func, void *mv_data,
                               ...)
```

创建“无矩阵”矩阵。mv_func 是完成矩阵、向量乘积的函数指针。参看 8.3.2。

```
INT phgMapE2L(MAP *map, int dof_no, SIMPLEX *e, int index)
```

返回 map 中的第 dof_no 个自由度对象在单元 e 中的第 index 个自由度在映射中的局部编号。dof_no 和 index 均从 0 开始。

```
INT phgMapD2L(MAP *map, int dof_no, INT index)
```

返回 map 中的第 dof_no 个自由度对象的第 index 个自由度在映射中的局部编号。

```
INT phgMapL2V(MAP *map, INT index)
```

返回 map 中局部编号为 index 的元素的局部向量编号。

```
INT phgMapL2G(MAP *map, INT index)
```

返回 map 中局部编号为 index 的元素的全局编号。

```
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。

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

该函数用于将自由度对象中的数据传递给向量。

```
void phgMapVecToDofArrays(MAP *map, VEC *vec, BOOLEAN remove_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。remove_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。

```
VEC *phgMapDofArraysToVec(MAP *map, int nvec, BOOLEAN remove_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。

```
MAT *phgMapCreateMat(MAP *rmap, MAP *cmap)
```

创建矩阵，rmap 和 cmap 分别给出行和列映射。

```
MAP *phgMatGetRowMap(MAT *mat)
```

返回矩阵 mat 中的行映射 (映射的引用计数增加 1)。

```
MAP *phgMatGetColumnMap(MAT *mat)
```

返回矩阵 mat 中的列映射 (映射的引用计数增加 1)。

```
MAT *phgMatCreateBlockMatrix(GRID *g, int p, int q, MAT *pmat[], FLOAT coeff[],
                             MAT_OP trans[])
```

创建一个 p×q 分块矩阵，其块 (i,j) 等于 coeff[iq+j]×trans[iq+j](pmat[iq+j]), $0 \le i < p$, $0 \le j < q$。

如果 coeff 等于 NULL 则表示所有系数均为 1。如果 trans 等于 NULL 则表示所有元素均为 MAT_OP_N。

注：PHG 中分块矩阵的实现基于 matrix-free 接口，它要求所有非对角块的 handle_bdry_eqns 值为 FALSE，而所有对角块的 handle_bdry_eqns 值同时为 TRUE 或同时为 FALSE。

```
void phgMatAddEntry(MAT mat, INT row, INT col, FLOAT value)
```

添加矩阵元素，行列均使用局部编号。

```
void phgMatAddGlobalEntry(MAT mat, INT row, INT col, FLOAT value)
```

添加矩阵元素，行列均使用全局向量编号。

```
void phgMatAddLGEntry(MAT mat, INT row, INT col, FLOAT value)
```

添加矩阵元素，使用局部行编号、全局向量列编号。

```
void phgMatAddGLEntry(MAT mat, INT row, INT col, FLOAT value)
```

添加矩阵元素，使用全局向量行编号、局部列编号。

```
void phgMatAddEntries(MAT *mat, INT nrows, INT *rows, INT ncols, INT *cols,
                      FLOAT *values)
```

添加 nrows × ncols 个矩阵元素，行列均使用局部编号。

```
void phgMatAddGlobalEntries(MAT *mat, INT nrows, INT *rows, INT ncols, INT *cols,
                            FLOAT *values)
```

添加 nrows × ncols 个矩阵元素，行列均使用全局向量编号。

```
void phgMatAddLGEntries(MAT *mat, INT nrows, INT *rows, INT ncols, INT *cols,
                        FLOAT *values)
```

添加 nrows × ncols 个矩阵元素，使用局部行编号、全局向量列编号。

```
void phgMatAddGLEntries(MAT *mat, INT nrows, INT *rows, INT ncols, INT *cols,
                        FLOAT *values)
```

添加 nrows × ncols 个矩阵元素，使用全局向量行编号、局部列编号。

```
void phgMatAssemble(MAT *mat)
```

组装矩阵，将不属于本地的行发送并叠加到相应进程。

```
void phgMatDisassemble(MAT *mat)
```

卸装装矩阵。

```
void phgMatDestroy(MAT **mat_ptr)
```

销毁矩阵。

```
MAT *phgMatRemoveBoundaryEntries(MAT *mat)
```

产生一个新的矩阵，它去掉了原矩阵中的所有边界行和边界列。矩阵的边界行和列通过定义映射的自由度对象中的 DB_mask 成员确定，当矩阵的映射不与自由度对象相关联、或所有自由度对象中的 DB_mask 均为 0 时该函数返回原矩阵。新产生的矩阵不再与原矩阵的自由度相关联。

```
SOLVER *phgMat2Solver(OEM_SOLVER *oem_solver, MAT *mat)
```

与函数 phgSolverMat2Solver 作用相同。

```
VEC *phgMapCreateVec(MAP *map, int nvec)
```

创建向量，nvec 为向量维数。

```
VEC *phgVecGetMap(VEC *vec)
```

返回向量 vec 的映射 (映射的引用计数增加 1)。

```
void phgVecAddEntry(VEC *vec, int which, INT index, FLOAT value)
```

添加向量元素 (使用局部编号)。which 指明添加到向量的哪个分量。

```
void phgVecAddEntries(VEC *vec, int which, INT n, INT *indices, FLOAT *values)
```

添加 n 个向量元素，indices 给出这些元素的局部编号。which 指明添加到向量的哪个分量。

```
void phgVecAddGlobalEntry(VEC *vec, int which, INT index, FLOAT value)
```

添加向量元素 (使用全局编号)。which 指明添加到向量的哪个分量。

```
void phgVecAddGlobalEntries(VEC *vec, int which, INT n, INT *indices, FLOAT *values)
```

添加 n 个向量元素，indices 给出这些元素的全局编号。which 指明添加到向量的哪个分量。

```
void phgVecAssemble(VEC *vec)
```

组装向量，将不属于本地的元素发送并叠加到相应进程。

```
void phgVecDisassemble(VEC *vec)
```

卸装向量。

```
void phgVecDestroy(VEC **vec_ptr)
```

销毁向量。

```
VEC *phgVecAXPBY(FLOAT a, VEC *x, FLOAT b, VEC **y)
```

计算 $y := a\vec{x} + b\vec{y}$。当 b == 0 时允许 y == NULL，此时将创建并返回一个新向量 y。

```
VEC *phgVecCopy(VEC *src, VEC **dest)
```

向量拷贝操作。该函数允许 dest == NULL，此时将创建并返回一个新向量 dest。

```
MAT *phgMatAXPBY(FLOAT a, MAT *x, FLOAT b, MAT **y)
```

计算矩阵的线性运算。参数的含义与 phgVecAXPBY 类似。

```
VEC *phgMatVec(MAT_OP op, FLOAT alpha, MAT *A, VEC *x, FLOAT beta, VEC **y)
```

计算 $y := \alpha Ax + \beta y$ (与 BLAS 函数 DGEMV 类似)。MAT_OP 指定矩阵 A 是否转置，可以取 MAT_OP_N (0) MAT_OP_T (1) 或 MAT_OP_D (2)。

该函数允许 A == NULL，表示 A 为单位阵。当 alpha == 0 时允许 x == NULL。当 beta == 0 时允许 y == NULL，此时将创建一个新的对象 y。函数中对 alpha 和 beta 的特殊值 $(0, 1, -1)$ 进行了特殊处理以减少不必要的浮点运算，因此可以用来完成许多类型的矩阵和向量运算，例如：

phgMatVec(0, 1.0, NULL, x, 0.0, &y)	$y := x$ (向量拷贝)
phgMatVec(0, -1.0, NULL, x, 0.0, &y)	$y := -x$
phgMatVec(0, alpha, NULL, x, 0.0, &y)	$y := \alpha x$
phgMatVec(0, 0.0, NULL, NULL, -1.0, &y)	$y := -y$
phgMatVec(0, 0.0, NULL, NULL, beta, &y)	$y := \beta y$
phgMatVec(0, alpha, NULL, x, beta, &y)	$y := \alpha x + \beta y$
phgMatVec(0, 1.0, A, x, 0.0, &y)	$y := Ax$
phgMatVec(0, -1.0, A, x, 0.0, &y)	$y := -Ax$

当 op = MAT_OP_D 时，该函数计算 A 的对角线部分与 x 的乘积。

```
FLOAT phgVecNorm2(VEC *vec, int which, FLOAT *result)
```

计算向量的 L2 模，如果指针 result 非空则它指向存放结果的地址。which 指明计算向量的哪个分量的模，如果 which < 0 则表明计算所有分量的模，此时如果 result 为非空指针则它指向的缓冲区长度应该不小于 vec->nvec，用于保存 vec->nvec 个分量的模。函数返回值为指定分量 (或第一个分量如果 which < 0) 的二模。

```
FLOAT phgVecNorm1(VEC *src)
```

计算向量的 L1 模。参数含义同 phgVecNorm2。

```
FLOAT phgVecNormInfty(VEC *vec, int which, FLOAT *result)
```

计算向量的最大模。参数含义同 phgVecNorm2。

```
FLOAT phgVecDotVec(VEC *x, int which_x, VEC *y, int which_y, FLOAT *result)
```

计算向量 x 和 y 的内积。参数 which_x、which_y 和 result 与 phgVecNorm2 中相同。

```
void phgMatDumpMATLAB(MAT *A, const char *var_name, const char *file_name)
```

将矩阵以 MATLAB 稀疏矩阵的格式输出到文件中，var_name 给出 MATLAB 变量名，file_name 给出输出文件名 (通常以 .m 为扩展名)，输出形式为“var_name = spconvert([...]);”。

```
void phgVecDumpMATLAB(VEC *v, const char *var_name, const char *file_name)
```

将向量的所有 (v->nvec 个) 分量以 MATLAB 稠密矩阵的格式输出到文件中，矩阵的每列对应于向量的一个分量。var_name 给出 MATLAB 变量名，file_name 给出输出文件名 (通常以 .m 为扩展名)，输出形式为“var_name = [...];”。

A.6 解法器

本节给出 PHG 所支持的解法器，它们可用于函数 phgSolverCreate。

SOLVER_PETSC ▷ PETSc 解法器
SOLVER_SPC ▷ SPC 解法器
SOLVER_SUPERLU ▷ SuperLU 解法器
SOLVER_HYPRE ▷ HYPRE 解法器
SOLVER_LASPACK ▷ LASPack 解法器 (仅支持串行)
SOLVER_PCG ▷ PHG 的 PCG 解法器
SOLVER_GMRES ▷ PHG 的 GMRES 解法器
SOLVER_AMS ▷ PHG 的 AMS (Auxiliary space Maxwell Solver) 解法器
SOLVER_DEFAULT ▷ 默认解法器，表示让 PHG 选择解法器
phgSolverList ▷ 该数组列出 PHG 的所有解法器，格式如下：

```
extern OEM_SOLVER *phgSolverList[];
```

phgSolverNames ▷ 该数组列出 PHG 的所有解法器的名称，数组最后是一个 NULL，其格式如下：

```
extern const char *phgSolverNames[];
```

PC_PROC ▷ PHG 的 PCG 和 GMRES 解法器的预条件子接口，具体如下：

```
VEC *(*PC_PROC)(SOLVER *M, VEC *b, VEC *x_1, VEC **u);
```

该函数实现预条件过程，并将结果 *u 返回。

A.7 初始化、退出、错误处理及信息输出

```
void phgInit(int *argc, char ***argv)
```

初始化 PHG (及 MPI、解法器等)。必须在调用其它 PHG 函数前调用。argc 和 argv 必须是真实命令行参数的地址。

```
void phgFinalize(void)
```

退出 PHG。必须在程序结束前调用。

```
int phgSetVerbosity(int verbosity)
```

控制 PHG 的内部信息输出：0 (默认值) 表示不输出内部信息。

```
void phgPause(void)
```

暂停程序，按回车键继续。用于程序调试。

```
int phgPrintf(char *fmt, ...)
```

显示信息。同 printf 函数，但只在进程 0 中起作用。

```
void phgInfo(int verbose_level, char *fmt, ...)
```

显示信息，供内部程序调试用。除参数 verbose_level 外，格式与 printf 类似。该函数只有当用户用 -log_file 选项指定了日志文件时才输出信息。

```
void phgWarning(char *fmt, ...)
```

显示警告信息。格式同 printf。

```
void phgError(int code, char *fmt, ...)
```

显示错误信息。如果 code != 0 则显示信息后中止程序执行。

```
void phgAbort(int code)
```

强制中止程序执行，code 为程序返回码。

```
double phgGetTime(double tarray[])
```

获取时间。tarray 是一个长度不小于 3 的 double 型数组，当 tarray != NULL 时，tarray[0] 返回用户时间，tarray[1] 返回系统时间，tarray[2] 返回墙上时间，以秒为单位。函数返回值为当前墙上时间，以秒为单位。

A.8 内存管理

```
void *phgAlloc(size_t size)
```

相当于 malloc()，但如果内存申请失败则中止程序执行。

```
void *phgCalloc(size_t nmemb, size_t size)
```

相当于 calloc()，但如果内存申请失败则中止程序执行。

```
void *phgRealloc(void *ptr, size_t size)
```

相当于 realloc()，但如果内存申请失败则中止程序执行。

```
void phgFree(void *ptr)
```

相当于 free()，但它允许 ptr == NULL。

A.9 性能统计

```
double phgPerfGetMflops(GRID *g, double *aggregate,
                        double *since_last_call)
```

获取当前浮点性能值 (Mflops) (需要其它软件包如 PAPI 的支持)。aggregate 返回从程序开始执行到当前的平均性能，since_last_call 返回从上次调用该函数 (第 1 次调用的话从程序开始执行) 到当前的平均性能。函数返回值等于 *since_last_call。这里统计的性能是 g->comm 中所有进程的性能之和。

```
size_t phgMemoryUsage(GRID *g, size_t *peak)
```

返回程序当前使用的内存 (RSS) 字节数，如果 peak 为非空指针，则在 *peak 中返回程序使用过的最大内存量 (以特定频率对内存的使用进行采样的结果)。

如果 g == NULL，则返回的是本进程的内存使用情况，反之则返回 g 中所有进程中最大内存使用情况。

```
size_t phgMemoryPeakReset(size_t value)
```

将 PHG 内部的最大内存计数置为 value，返回最大内存计数在此之前的值。

```
size_t phgMemoryPeakRestore(size_t value)
```

将 PHG 内部的最大内存计数置为现有的值与 value 中的较大者，返回在此之前最大内存计数的值。

A.10 命令行选项

```
void phgOptionsRegisterNoArg(const char *name, const char *help,
                             BOOLEAN *var)
```

注册一个不带参数的命令行选项。name 给出选项名称 (不带 "-" 号)。help 给出关于该选项的帮助信息。var 为一个 BOOLEAN 型 (全局或静态) 变量的地址，该变量的初始值应为 FALSE，当命令行中包含该选项时，PHG 将它的值改变为 TRUE。

```
void phgOptionsRegisterInt(const char *name, const char *help,          INT *var)
```

注册一个带整型参数的命令行选项。name 给出选项名称 (不带 "-" 号)。help 给出关于该选项的帮助信息。var 为一个 INT 型 (全局或静态) 变量的地址，当命令行中包含该选项时，PHG 将相应的参数值赋给 *var。

```
void phgOptionsRegisterFloat(const char *name, const char *help,          FLOAT *var)
```

注册一个带浮点型参数的命令行选项。name 给出选项名称 (不带 "-" 号)。help 给出关于该选项的帮助信息。var 为一个 FLOAT 型 (全局或静态) 变量的地址，当命令行中包含该选项时，PHG 将相应的参数值赋给 *var。

```
void phgOptionsRegisterString(const char *name, const char *help,          char **var)
```

注册一个带字符串参数的命令行选项。name 给出选项名称 (不带 "-" 号)。help 给出关于该选项的帮助信息。var 为一个 char * 型 (全局或静态) 变量的地址，当命令行中包含该选项时，PHG 将相应的参数值赋给 *var，PHG 为字符串动态分配内存块并负责内存块的释放，用户程序不能自行释放它。

```
void phgOptionsRegisterFilename(const char *name, const char *help,
                                char **var)
```

注册一个带文件名参数的命令行选项。name 给出选项名称 (不带 "-" 号)。help 给出关于该选项的帮助信息。var 为一个 char * 型 (全局或静态) 变量的地址，当命令行中包含该选项时，PHG 将相应的参数值赋给 *var，PHG 为文件名字符串动态分配内存块并负责内存块的释放，用户程序不能自行释放它。

```
void phgOptionsRegisterKeyword(const char *name, const char *help,
                               const char **keys, int *var)
```

注册一个带关键字参数的命令行选项。name 给出选项名称 (不带 "-" 号)。help 给出关于该选项的帮助信息。keys 是一个以 NULL 结尾的字符串指针数组，列出该选项所允许的参数。var 为一个 int 型 (全局或静态) 变量的地址，该变量的值为 keys 数组的索引 (从 0 开始)，当命令行中包含该选项时，PHG 将相应修改 var 的值。

```
void phgOptionsRegisterHandler(const char *name, const char *help,
                               OPTION_HANDLER func)
```

注册一个用指定函数 func 处理的带参数的命令行选项，func 为具有如下接口的函数：

```
int func(const char *name, const char *string)
```

其中 name 为选项名，string 为选项后面所跟随的参数，如果函数返回非 0 值则表示参数中有错误 (如非法参数)。当参数值 string 为空指针时，该函数应该显示帮助信息 (如参数的合法格式等)。

```
void phgOptionsRegisterTitle(const char *str, const char *help, const char *category)
```

该函数并不定义新的命令选项，而用于为一组命令行选项注册一个标题，显示在命令行选项的帮助信息中。category 用于命令行选项的分类。

```
void phgOptionsPreset(const char *string)
```

该函数用来预设一些参数的值，它必须在 phgInit 之前调用。例如可以用

```
phgOptionsPreset("-oem_options '-ksp_type cg -pc_type bjacobi'")
```

来将 PETSc 的默认解法器和预条件子分别设为 CG 和 block Jacobi。

```
void phgOptionsShowCmdline(void)
```

列出用户输入的命令行。

```
void phgOptionsShowUsed(void)
```

列出所有通过命令行选项指定 (即出现在命令行中的选项) 的参数值。

```
void phgOptionsHelp(void)
```

列出所有命令行选项及相应的帮助信息。

```
BOOLEAN phgOptionsSetOptions(const char *str)
```

根据 str 中给出的选项设定相应变量的值。

```
BOOLEAN phgOptionsSetNoArg(const char *op_name, BOOLEAN value)
```

将选项 op_name 所对应的变量值置为 value，op_name 必须是一个无参数的选项名，不含 '-' 或 '+' 号。

```
BOOLEAN phgOptionsSetInt(const char *op_name, INT value)
```

将选项 op_name 所对应的变量值置为 value，op_name 必须是一个带整型参数的选项名，不含 '-' 号。

```
BOOLEAN phgOptionsSetFloat(const char *op_name, FLOAT value)
```

将选项 op_name 所对应的变量值置为 value，op_name 必须是一个带实型参数的选项名，不含 '-' 号。

```
BOOLEAN phgOptionsSetKeyword(const char *op_name, const char *value)
```

将选项 op_name 所对应的变量值置为 value，op_name 必须是一个带关键字参数的选项名，不含 '-' 号。

```
BOOLEAN phgOptionsSetString(const char *op_name, const char *value)
```

将选项 op_name 所对应的变量值置为 value，op_name 必须是一个带字符串参数的选项名，不含 '-' 号。

```
BOOLEAN phgOptionsSetFilename(const char *op_name, const char *value)
```

将选项 op_name 所对应的变量值置为 value，op_name 必须是一个带文件名参数的选项名，不含 '-' 号。

```
BOOLEAN phgOptionsGetNoArg(const char *op_name)
```

返回选项 op_name 所对应的变量值，op_name 必须是一个无参数的选项名，不含 '-' 或 '+' 号。

```
INT phgOptionsGetInt(const char *op_name)
```

返回选项 op_name 所对应的变量值，op_name 必须是一个带整型参数的选项名，不含 '-' 号。

```
FLOAT phgOptionsGetFloat(const char *op_name)
```

返回选项 op_name 所对应的变量值，op_name 必须是一个带实型参数的选项名，不含 '-' 号。

```
const char *phgOptionsGetKeyword(const char *op_name)
```

返回选项 op_name 所对应的变量值，op_name 必须是一个带关键字参数的选项名，不含 '-' 号。

```
const char *phgOptionsGetString(const char *op_name)
```

返回选项 op_name 所对应的变量值，op_name 必须是一个带字符串参数的选项名，不含 '-' 号。

```
const char *phgOptionsGetFilename(const char *op_name)
```

返回选项 op_name 所对应的变量值，op_name 必须是一个带文件名参数的选项名，不含 '-' 号。

```
void phgOptionsPush(void)
```

保存所有选项的当前值。

```
void phgOptionsPop(void)
```

恢复上一次用 phgOptionsPush 保存的选项值。

A.11 网格管理

```
GRID *phgNewGrid(int flags)
```

创建新的网格对象，返回创建的网格的指针。flags 包含一些标志位，用于控制当网格改变时是否更新特定的几何对象，包括如下一些标志：VERT_FLAG：对网格顶点进行编号；EDGE_FLAG：对网格边进行编号；FACE_FLAG：对网格面进行编号；ELEM_FLAG：对网格单元进行编号；GEOM_FLAG：自动计算几何量 (面/体积、法向、直径、Jacobian 等)，参看 A.17 中的 phgGeom* 函数。通常取 flags = -1。

```
void phgFreeGrid(GRID **gptr)
```

释放网格对象 (及其与之相关联的自由度对象)。

```
void phgSetPeriodicity(GRID *g, int flags)
```

用于设定网格在一个或多个空间方向上的周期性，其中 flags 取 X_MASK、Y_MASK 及 Z_MASK 的任意按位组合。如：phgSetPeriodicity(g, X_MASK | Z_MASK) 表示网格在 x 和 z 方向上是周期的。

该函数的调用必须在 phgImport 之前进行，并且它对初始网格有下面一些要求：

(1) 网格在周期面上必须是协调的。

(2) 初始网格不能太粗，要求任意连接一对周期结点的由单元边构成的路径必须至少包含三条边，否则程序会出错。如果初始网格不满足这一条件，可以通过对其适当加密生成一个新的满足该条件的网格。例如，对网格 test/cube.dat 进行 6 次一致加密后得到的包含 384 个单元的网格便满足该条件。

```
void phgSetPeriodicDirections(GRID *g, const FLOAT dirs[])
```

指定周期方向。数组 dirs 中包含 9 个数，给出三个线性无关的向量，代表网格的三个周期方向，这三个方向不要求正交。phgSetPeriodicity 中将第一个方向做为 x 方向，第二个方向做为 y 方向，第三个方向做为 z 方向。该函数的调用必须在 phgImport 之前进行。

```
void phgCheckConformity(GRID *g)
```

检查网格的协调性，用于程序调试。

```
void phgMarkElements(STRATEGY strategy, DOF *error, FLOAT theta, DOF *osc,
                     FLOAT zeta, int depth, STRATEGY coarsen_strategy,
                     DOF *coarsen_error, FLOAT gamma, int coarsen_depth, FLOAT tol)
```

标记网格的细化和粗化。strategy 指定网格细化策略。用 M 表示网格中所有单元的集合，则该函数根据指定的策略计算细化单元集合 M_r (M 的子集)。用 $\eta(s)$ 表示单元 (或面) s 上的误差指示子，θ 表示细化比例，PHG 提供下述细化策略：

(1) MARK_MAX
Max 策略：$M_r = \{e \mid e \in M, \eta(e) > \theta \cdot \max_{s\in M} \eta(s)\}$

(2) MARK_GERS
GERS 策略: M_r 为满足 $\sum_{e\in M_r} \eta(e) > \theta \cdot \sum_{e\in M} \eta(e)$ 的最小子集。

(3) MARK_MNS
MNS 策略：$M_r = M_{\text{error}} \cup M_{\text{osc}}$，其中 M_{error} 是相对于 error 和 theta 的 GERS 细化单元集合，M_{osc} 是相对于 osc 和 zeta 的 GERS 细化单元集合。该函数要求误差指示子 error 定义在面上而不是体上。

(4) MARK_EQDIST
EQDIST 策略：$M_r = \{e \mid e \in M, \eta(e) > \theta \cdot \texttt{tol}\}$

(5) MARK_ALL
细化所有单元：$M_r = M$

(6) MARK_NONE
不细化任何单元：$M_r = \emptyset$

(7) MARK_DEFAULT
使用命令行选项 -refine_strategy 或 -strategy 指定的策略

函数返回时，M_r 中所有单元的 mark 成员的值将被置为 depth (函数 phgRefineMarkedElements 会将这些单元至少二分细化 depth 次)。

类似地，coarsen_strategy 指定网格粗化策略，粗化策略的名称与细化相同。函数根据指定的策略计算粗化单元集合 M_c，用 $\eta(s)$ 表示单元 (或面) s 上的粗化误差指示子，γ 表示粗化比例，则各策略在标记粗化时的含义如下：

(1) MARK_MAX
Max 策略：$M_c = \{e \mid e \in M, \eta(e) \le \gamma \cdot \max_{s\in M} \eta(s)\}$

(2) MARK_GERS
GERS 策略：M_c 为满足 $\sum_{e\in M_c} \eta(e) \le \gamma \cdot \sum_{e\in M} \eta(e)$ 的最大子集。

(3) MARK_EQDIST
EQDIST 策略：$M_c = \{e \mid e \in M, \eta(e) \leq \gamma \cdot \texttt{tol}\}$

(4) MARK_ALL
粗化所有单元：$M_c = M$

(5) MARK_NONE
不粗化任何单元：$M_c = \emptyset$

(6) MARK_DEFAULT
使用命令行选项 -coarsen_strategy 或 -strategy 指定的策略.

函数返回时，M_c 中所有单元的 mark 成员的值将被置为 -coarsen_depth (函数 phgCoarsenMarkedElements 会将这些单元粗化至多 coarsen_depth 次)。

自由度对象 error 为细化误差指示子，osc 为 data oscillation，coarsen_error 为是用来标记粗化的自由度对象。实型参数 theta 为 error 的细化比例，zeta 为 osc 的细化比例，gamma 为粗化比例，它们必须是非负数并且小于或等于 1。整型参数 depth 以及 coarsen_depth 分别为细化和粗化的次数，它们必须大于 0。实型参数 tol 为 tolerance, 仅被 MARK_EQDIST 用到。

所有其它单元，即集合 $M \setminus (M_r \cup M_c)$ 中的单元，其 mark 成员的值将被置为 0。

对误差子需要注意的是：

(1) 三个自由度对象不能同时为 NULL, error 可以是面上的自由度对象，或者体上的自由度，但 **不能** 同时含有面和体上的数据；osc 只要求体上有数据即可, coarsen_error 只要求体上有数据即可，对于 osc 以及 coarsen_error 建议使用的自由度类型是 DOF_P0。对细化，MARK_MAX 以及 MARK_EQDIST, error 的数据必须存在体上, 建议使用 DOF_P0。需要注意的是无论是面上或者体上的数据，只能含有 **一个** 数据。

(2) 使用者需要自己对自由度的类型与标记策略对应起来。默认行为是如果 osc 不是 NULL，就会被标记。

(3) 对于细化，标记策略如果是 MARK_ALL 或 MARK_NONE, 则 error 可以为 NULL; 对于粗化，标记策略如果是 MARK_ALL 或者 MARK_NONE, 则 coarsen_error 可以为 NULL。

(4) 细化的时候用到的自由度对象是 error 以及 osc, 粗化的用到的是 coarsen_error。

标记子函数有一些默认行为:

(1) 子函数是先进行细化的标记，然后进行粗化的标记。

(2) 对细化，osc 不是 NULL 的时候,，则对 osc 采用同样的策略。对 MARK_GERS 而言，osc 在 error 标记的基础上扩大被标记的子集而满足 MARK_GERS 策略；如果 error 的数据存放在面上，则策略等价于 MARK_MNS; 如果策略是 MARK_MAX，则对 osc 直接使用 MARK_MAX 策略。如果 osc 是 NULL, 直接对 error 操作。osc 是可选的, 但对 MARK_MNS 则是必须的。

(3) 对细化, MARK_EQDIST, 如果 osc 不是 NULL, 使用一样的策略。

(4) 对细化，如果策略为 MARK_ALL, 则细化标记完毕以后直接退出标记子函数，不进行粗化。

(5) 粗化仅仅使用 coarsen_error, 如果粗化策略不是 MARK_ALL 或者 MARK_NONE, coarsen_error 不能为 NULL。

(6) 如果粗化的策略是 MARK_ALL, 之前标记的细化的全部被覆盖。

(7) 如果细化策略不是 MARK_ALL 并且粗化策略不是 MARK_NONE, 则 coarsen_error 不能为 NULL, 所以如果不准备作粗化的话，粗化策略一定要选 MARK_NONE。

(8) 如果策略选择 MARK_DEFAULT，这个策略用在细化以及粗化上，默认的策略是 MARK_GERS, 用在细化的标记上没有问题，但粗化需要注意，如果不做粗化的话，不要将粗化策略选择为 MARK_DEFAULT。使用者可以在运行的时候用选项来选择满足自己需要的策略, 用选项 -strategy 选择。

(9) 如果连续使用两次标记函数，则第二次的有效. 连续的意思是标记了网格，但没有进行粗化以及细化的操作，然后又使用了标记策略，两个标记函数之间即使含有很多代码也看作连续。

(10) 命令行选项 -refine_strategy、-coarsen_strategy 和 -strategy 的合法参数为 max, gers, mns, eqdist, all, none。

(11) 警告: 对于 PHG, 只认 error, osc 以及 coarsen_error, 标记方式按照上面的描述来做，对于不同的模，用户要负责做好转换工作, 来满足自己的需求。

```
void phgMarkRefine(STRATEGY strategy, DOF *error, FLOAT theta, DOF *osc, FLOAT zeta,
                   int depth, FLOAT tol)
```

基于 phgMarkElements 定义的宏，只标记细化单元。

```
void phgMarkCoarsen(STRATEGY strategy, DOF *error, FLOAT gamma, int depth, FLOAT tol)
```

基于 phgMarkElements 定义的宏，只标记粗化单元。

```
void phgRefineMarkedElements(GRID *g)
```

细化网格中的指定单元。所有 mark 大于 0 的叶子单元至少被细化 mark 次。PHG 会自动对其它单元进行必要的细化以保持网格的协调性。细化完成后，新网格中所叶子单元的 mark 成员的值等于原网格叶子单元中的值减去该单元被细化的次数。

```
void phgRefineAllElements(GRID *g, int level)
```

将所有叶子单元细化 level 次 (即一致网格加密，相当于将所有叶子单元的 mark 设为 level，然后调用一次 phgRefineMarkedElements)。

```
void phgCoarsenMarkedElements(GRID *g)
```

粗化网格中的指定单元。叶子单元的 mark 值为该单元最多允许被粗化的次数反号，即一个叶子单元最多允许被粗化 −mark 次。PHG 保证粗化后，由新的叶子单元所构成的网格是协调的，并且在满足 mark 值及网格协调性约束的前提下粗化尽量多的单元。粗化完成后，新网格中所叶子单元的 mark 成员的值等于原网格叶子单元中的值减去该单元被粗化的次数。

```
void phgPartitionGrid(GRID *g)
```

分布网格：将当前网格划分成子网格，分布到各进程中。该函数为 PHG 内部使用，用户程序应该调用 phgBalanceGrid。

```
void phgRedistributeGrid(GRID *g)
```

重分网格：对子网格进行调整以达到更好的负载平衡。该函数为 PHG 内部使用，用户程序应该调用 phgBalanceGrid。

```
int phgBalanceGrid(GRID *g, FLOAT lif_threshold,        INT submesh_threshold,
                   DOF *weights, FLOAT power)
```

根据阈值 lif_threshold 和 submesh_threshold 调整子网格的数目与分布。其中 lif_threshold 用于控制负载不平衡因子，表示当负载不平衡因子大于或等于该值时重分网格，submesh_threshold 用于控制子网格数目，使得子网格的平均单元数目不少于该值 (做为特殊约定，当 submesh_threshold 取负数时表示使用 PHG 内置的默认值)。该函数返回 0 表示网格不变，非 0 表示对网格进行了重划分。weights 和 power 给出子网格划分时的单元权重 (取 weights 的 power 次方)，当 weights 为 NULL 或 power 为 0 时表示所有单元的权重均相等。

```
GRID *phgSurfaceCut(GRID *g, const FLOAT cuts[])
```

该函数返回网格 g 与一个曲面相交产生的新网格，新网格包含老网格的所有顶点以及曲面与老网格边的所有交点。数组 cuts 的长度为 g->nedge，它按照边的局部编号顺序依次给出曲面与每条边的交点位置，取值小于等于 0 或大于等于 1 表示与该边没有交点，否则表示交点位于从全局编号较小的顶点到全局编号较大的顶点的相对位置。可以用函数 phgDofTransfer 来在两个网格间传递自由度数据。

注：目前该函数要求曲面与每条边最多只能有一个交点，并且每个单元上各交点的位置必须满足一些合理的限制条件。同时，不允许对所产生的新网格进行单元细化。此外，该函数的接口形式是临时性的，将来可能会改变。

```
FLOAT phgGetBoundaryError(GRID *g, SIMPLEX *e)
```

当存在曲面边界时，返回单元 e 所有位于曲面边界的边的中点与其映射到曲面上的点之间的距离的平方的最大值 (用于估计边界上的曲线误差)。

```
ForAllElements(g, e)
```

用于对网格中所有叶子单元进行遍历的宏，通常用于有限元离散或后验误差估计的计算中，每步循环中 e 指向当前叶子单元，循环结束时 e 被置为 NULL。

```
GlobalVertex(g, index)
```

给出本地编号为 index 的顶点的全局编号。

```
GlobalEdge(g, index)
```

给出本地编号为 index 的边的全局编号。

```
GlobalFace(g, index)
```

给出本地编号为 index 的面的全局编号。

```
GlobalElement(g, index)
```

给出本地编号为 index 的单元的全局编号。

A.12 输入输出

```
void phgImportSetBdryMapFunc(BDRY_MAP_FUNC func)
```

指定导入网格文件时，输入文件边界类型到 PHG 边界类型的转换函数。该函数应该调用在 phgImport 之前调用，所设定的边界类型转换函数仅对后续的 phgImport 函数起作用。func 是具有如下接口形式的用户自定义函数：

```
int func(int bctype)
```

该函数返回与输入边界类型 bctype 相对应的 PHG 边界类型 (如 DIRICHLET、NEUMANN 等)，返回 -1 表示未知或非法的边界类型。该函数中 bc_map 参数可以为空指针，表示使用默认的边界类型转换函数。参看 2.8.3。

```
BTYPE phgImportSetDefaultBdryType(BTYPE type)
```

指定默认边界类型。该函数应该调用在 phgImport 之前调用，所设定的默认边界类型仅对后续的 phgImport 函数起作用。type 给出新的默认边界类型，函数返回值为之前的默认边界类型。type 参数取 0 等价于 UNDEFINED。参看 2.8.3。

```
BOOLEAN phgImport(GRID *g, const char *filename, BOOLEAN distr)
```

从文件中导入网格数据。g 是网格指针，它必须是用 phgNewGrid 新创建的；filename 是文件名；如果 distr 为 TRUE，则在网格导入后立即对其进行剖分并将子网格分布到所有进程中去 (如果不清楚该参数的确切作用，建议用 TRUE)。

目前 PHG 支持的网格文件格式有：ALBERTA 格式和 Medit 格式。

```
BOOLEAN phgExportALBERT(GRID *g, const char *filename)
```

将当前网格输出为 ALBERTA 的网格文件格式。如果当前网格包含多个子网格，该函数目前只输出第 0 号子网格。

```
BOOLEAN phgExportMedit(GRID *g, const char *filename)
```

将当前网格输出为 MEDIT (http://www.ann.jussieu.fr/~frey/logiciels/medit.html) 格式。如果当前网格包含多个子网格，该函数目前只输出第 0 号子网格。

```
const char *phgExportVTK(GRID *g, const char *filename, DOF *dof1, ...)
```

将网格以 VTK vtkUnstructuredGrid 的形式输出到指定文件，以便用其它基于 VTK 的可视化软件如 ParaView (http://www.paraview.org/)、MayaVi (http://mayavi.sourceforge.net/) 等处理，文件格式为二进制 legacy VTK file。可选参数指定要输出的自由度对象，以 NULL 结束。类型为 DOF_P0 (分片常数) 的自由度对象作为 cell data 输出，而其它类型的自由度对象则作为 point data 输出，后者需要调用自由度类型中的 BasFuncs 函数来计算网格顶点处的值。如果参数中给出的文件名不含扩展，该函数会自动加上扩展名 .vtk。函数返回输出文件 (加了扩展以后) 的文件名。该函数目前不支持输出类型为 DOF_ANALYTIC 的自由度对象。

```
const char *phgExportDX(GRID *g, const char *filename, DOF *dof1, ...)
```

将网格以 IBM Data Explorer 的 native file format 格式输出到指定文件，以便用 OpenDX 等软件 (http://www.opendx.org/) 进行可视化处理。可选参数指定要输出的自由度对象，以 NULL 结束。如

果参数中给出的文件名不含扩展，该函数会自动加上扩展名 `.dx`。函数返回输出文件 (加了扩展以后) 的文件名。该函数目前不支持输出类型为 `DOF_ANALYTIC` 的自由度对象。

A.13 自由度管理

`SpecialDofType(type)`

用于判断一个自由度类型是否特殊类型 (`DOF_CONSTANT` 或 `DOF_ANALYTIC`) 的宏。

`DofTypeDim(type)`

获取自由度类型的维数。`type` 为自由度类型对象。

`DofTypeName(type)`

获取自由度类型的名称。`type` 为自由度类型对象。

`DofTypeOrder(type)`

获取自由度类型的多项式阶数。`type` 为自由度类型对象。

`DofNoData`

用于 DOF 的 `userfunc` 成员，表示创建自由度对象时不申请数据缓冲区。

`DofNoAction`

用于 DOF 的 `userfunc` 成员，表示不对自由度对象进行自动赋值、插值等处理。

`DofInterpolation`

用于 DOF 的 `userfunc` 成员，表示当网格细化、粗化时自动对自由度对象进行插值。

`DofDim(dof)`

返回自由度对象的维数 (等于 `dof->dim` × `dof->type->dim`)。

`DofData(dof)`

返回本地自由度数据的起始地址，`dof` 为自由度对象。根据自由度数据的排列方式它实际等价于 `DofVertexData(dof, 0)`。

`DofVertexData(dof, index)`

返回对应于给定顶点的自由度数据的起始地址，`dof` 为自由度对象，`index` 为顶点的本地编号。

`DofEdgeData(dof, index)`

返回对应于给定边的自由度数据的起始地址，`dof` 为自由度对象，`index` 为边的本地编号。

`DofFaceData(dof, index)`

返回对应于给定面的自由度数据的起始地址，`dof` 为自由度对象，`index` 为面的本地编号。

`DofElementData(dof, index)`

返回对应于给定单元的自由度数据的起始地址，`dof` 为自由度对象，`index` 为单元的本地编号。

```
DofGetVertexDataCount(dof)
```

返回子网格中所有顶点自由度数据的长度。

```
DofGetEdgeDataCount(dof)
```

返回子网格中所有边自由度数据的长度。

```
DofGetFaceDataCount(dof)
```

返回子网格中所有面自由度数据的长度。

```
DofGetElementDataCount(dof)
```

返回子网格中所有单元 (体) 自由度数据的长度。

```
DofGetDataCount(dof)
```

返回子网格中自由度数据的总长度。

```
DofGetVertexDataCountGlobal(dof)
```

返回全局网格中所有顶点自由度数据的长度。

```
DofGetEdgeDataCountGlobal(dof)
```

返回全局网格中所有边自由度数据的长度。

```
DofGetFaceDataCountGlobal(dof)
```

返回全局网格中所有面自由度数据的长度。

```
DofGetElementDataCountGlobal(dof)
```

返回全局网格中所有单元 (体) 自由度数据的长度。

```
DofGetDataCountGlobal(dof)
```

返回全局网格中自由度数据的总长度。

```
DOF *phgDofNew(GRID *g, DOF_TYPE *type, SHORT dim, const char *name,
               DOF_USER_FUNC userfunc)
```

创建新自由度对象。g 为网格对象；type 为自由度类型 (DOF_TYPE)；dim 为自由度对象的维数 (每个位置 dim 个变量)；name 为自由度对象的名称；userfunc 指定与该自由度对象相关联的用户函数名 (接口类型为 DOF_USER_FUNC)，用于给该自由度对象赋值，userfunc 可以取特殊值 DofNoAction 或 DofInterpolation，前者表示不对自由度对象进行自动赋值、插值，后者表示当网格细化或粗化时进行自动插值。

```
void phgDofFree(DOF **dof)
```

释放自由度对象。

```
void phgDofSetFunction(DOF *u, DOF_USER_FUNC func)
```

设定与由度对象相关联的用户函数，该函数将用于给该自由度对象赋值。

```
void phgDofSetLambdaFunction(DOF *u, DOF_USER_FUNC_LAMBDA func)
```

设定与由度对象相关联的基于重心坐标的用户函数，该函数将用于给该自由度对象赋值。

```
BTYPE phgDofSetDirichletBoundaryMask(DOF *u, BTYPE mask)
```

设定 u->DB_mask 的值, 函数返回值为 u->DB_mask 原有的值。

```
void phgDofDump(DOF *dof)
```

打印自由度对象的内容。通常用于程序调试。

```
void phgDofSetDataByValue(DOF *dof, FLOAT value)
```

将自由度对象的数据初始化为指定值 value。

```
void phgDofSetDataByValues(DOF *dof, const FLOAT array_of_values[])
```

将自由度对象的数据初始化为指定值。array_of_values 中包含 DofDim(dof) 个数，与函数值的分量个数相对应。

```
void phgDofSetDataByValuesV(DOF *dof, FLOAT v0, ...)
```

将自由度对象的数据初始化为由 v0 和随后的可变参数给出的值, 可变参数个数加 1 必须等于 DofDim(dof)。

```
void phgDofSetDataByFunction(DOF *dof, DOF_USER_FUNC userfunc)
```

用指定函数为自由度对象赋值。参看 DOF_USER_FUNC。

```
size_t phgDofGetDataCount(DOF *dof)
```

返回指定自由度对象本地数据的长度 (与宏 DofGetDataCount 功能一样)。

```
size_t phgDofGetDataCountGlobal(DOF *dof)
```

返回指定自由度对象全局数据的长度 (与宏 DofGetDataCountGlobal 功能一样)。

```
FLOAT *phgDofEval(DOF *dof, SIMPLEX *e, const FLOAT lambda[], FLOAT *values)
```

计算自由度对象在指定单元中指定位置的值。dof 为自由度对象；e 为单元；lambda 为重心坐标；values 为输出缓冲区，用于存放计算出的自由度值，调用程序应该保证缓冲区的大小。返回输出缓冲区地址。

```
FLOAT *phgDofEvalGradient(DOF *dof, SIMPLEX *e, const FLOAT lambda[],
                          const FLOAT *gradbas, FLOAT *values)
```

与 phgDofEval 类似，但计算自由度对象的梯度在给定点的值。gradbas 是一个数组，包含自由度类型在单元中的所有基函数的梯度值，如果 gradbas 为空指针则 phgDofEvalGradient 将不使用它而自行计算基函数的梯度值。gradbas 参数的引入主要为了方便避免一些冗余的基函数梯度值的计算。

```
FLOAT *phgDofEvalDivergence(DOF *dof, SIMPLEX *e, const FLOAT lambda[],
                            const FLOAT *gradbas, FLOAT *values)
```

与 phgDofEval 类似，但计算自由度对象的散度在给定点的值。参数 gradbas 的内容与作用与函数 phgDofEvalGradient 中完全一样。

```
FLOAT *phgDofEvalCurl(DOF *dof, SIMPLEX *e, const FLOAT lambda[],
                      const FLOAT *gradbas, FLOAT *values)
```

与 phgDofEval 类似，但计算自由度对象的 Curl 在给定点的值。参数 gradbas 的内容与作用与函数 phgDofEvalGradient 中完全一样。

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

创建一个新自由度对象，它等于 src 的梯度。name 给出新自由度对象的名称。newtype 给出新自由度对象的类型，该参数如果取为 NULL 则表示新自由度对象的类型等于 src->type->grad_type。

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

创建一个新自由度对象，它等于 src 的散度。name 给出新自由度对象的名称。注意 DofDim(src) 必须是 3 的倍数。newtype 给出新自由度对象的类型，该参数如果取为 NULL 则表示新自由度对象的类型等于 src->type->grad_type。

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

创建一个新自由度对象，它等于 src 的 Curl。name 给出新自由度对象的名称。注意 DofDim(src) 必须是 3 的倍数。newtype 给出新自由度对象的类型，该参数如果取为 NULL 则表示新自由度对象的类型等于 src->type->grad_type。

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

复制自由度对象。该函数允许改变自由度类型，例如，可以用它将一个类型为三阶 Lagrange 元的自由度对象的值赋给一个类型为二阶 Lagrange 元的自由度对象。newtype 给出新自由度对象的类型，该参数如果取为 NULL 则表示新自由度对象的类型等于 src->type。

```
DofIsOwner(u, index)
```

检查指定的自由度是否为当前子网格所拥有。

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

计算 C := α op(A) op(B) + β C (相当于 BLAS 中的 GEMM 函数，其中 op 表示转置或不转置)。

当 Cptr 不等于 NULL 时，计算结果 C 保存在 *Cptr 中，并且 C 的类型与 *Cptr 的类型相同。

当 Cptr 等于 NULL 时，计算结果 C 是一个新创建的、类型为 B->type 的 DOF 对象 (此时要求 B 为非空指针)。

该函数要求 DofDim(B) = N × K 及 DofDim(C) = M × N。transa 和 transb 可以分别取 MAT_OP_N (不转置) 和 MAT_OP_T (转置)，各种情况下的矩阵维数及运算关系由表 A.1 给出。

当 blka ≤ 0 时表示 A 为满阵，此时要求 DofDim(A) = M × K。

当 blka > 0 时，表示 A 是一个分块对角矩阵，对角块大小为 blka × blka。此时要求 M = K，blka 整除 M，并且 DofDim(A) = M × blka。一种常用的情况是 blka = 1，此时 A 表示一个 M × M 的对角矩阵，DofDim(A) = M，A 中的分量对应矩阵的对角元素。

A 可以为 NULL，表示 A 为单位阵，此时要求 M = K。当 $\alpha = 0$ 时 B 可以为 NULL。当 $\beta = 0$ 时 C 可以为 NULL，此时，函数将返回一个新自由度对象，其类型与 B 的类型相同，维数为 M × N。

注：该函数涵盖了函数 phgDofAXPY 和 phgDofAXPBY 的功能。

表 A.1 phgDofMM 中的矩阵维数及运算关系

transa 取值	transb 取值	矩阵维数及矩阵运算
MAT_OP_N	MAT_OP_N	$C_{MN} = \alpha A_{MK} B_{KN} + \beta C_{MN}$
MAT_OP_T	MAT_OP_N	$C_{MN} = \alpha A^T_{KM} B_{KN} + \beta C_{MN}$
MAT_OP_N	MAT_OP_T	$C_{MN} = \alpha A_{MK} B^T_{NK} + \beta C_{MN}$
MAT_OP_T	MAT_OP_T	$C_{MN} = \alpha A^T_{KM} B^T_{NK} + \beta C_{MN}$

```
DOF *phgDofAXPY(FLOAT a, DOF *x, DOF **y)
```

计算 `y := a * x + y`。`a == 0` 相当于空操作，`a == 1` 时等价于自由度相加，`a == -1` 时等价于自由度相减。

```
DOF *phgDofAXPBY(FLOAT a, DOF *x, FLOAT b, DOF **y)
```

计算 `y := a * x + b * y`。`a == 0` 或 `b == 0` 时相当于自由度乘以常数。

```
DOF *phgDofAFXPBY(FLOAT a, void (*f)
```

(FLOAT *in, FLOAT *out), DOF *x, FLOAT b, DOF **y) 计算 `y := a * f(x) + b * y`。函数 `f` 的输入参数 `in` 为 `x` 函数值，其维数等于 `x` 的维数；输出参数 `out` 为 `y` 函数值，其维数等于 `y` 的维数。

```
DOF *phgDofAFXPBY1(FLOAT a, FLOAT (*f)
```

(FLOAT xvalue), DOF *x, FLOAT b, DOF **y) 计算 `y := a * f(x) + b * y`。与 phgDofAFXPBY 的区别在于这里函数 `f` 为标量函数，它分别作用在 `x` 的每个分量上。该函数要求 `x` 和 `y` 具有相同的维数。

```
DOF *phgDofMatVec(FLOAT alpha, DOF *A, DOF *x, FLOAT beta, DOF **y_ptr)
```

计算 `y := alpha*A*x + beta*y`，相当于 BLAS 子程序 DGEMV。当 `A == NULL` 时表示 `A` 为单位阵，当 `DofDim(A) == 1` 时表示 `A` 为标量乘以单位阵，当 `DofDim(A) == DofDim(x)` 时表示 `A` 为对角阵，而当 `DofDim(A) == DofDim(x)`^{2} 时表示 `A` 为满阵。

```
FLOAT phgDofNormL2Vec(DOF *x)
```

计算自由度对象的向量 2 范数，即所有自由度的平方和再开方。

```
FLOAT phgDofNormL1Vec(DOF *x)
```

计算自由度对象的向量 1 范数，即所有自由度的绝对值之和。

```
FLOAT phgDofNormInftyVec(DOF *x)
```

计算自由度对象的向量无穷范数，即所有自由度数据的最大绝对值。

```
FLOAT phgDofNormL1(DOF *u)
```

计算自由度对象所对应的有限元函数的 L^1 模。

```
FLOAT phgDofNormL2(DOF *u)
```

计算自由度对象所对应的有限元函数的 L^2 模。

```
FLOAT phgDofNormH1(DOF *u)
```

计算自由度对象所对应的有限元函数的 H^1 模 (积分)。

```
INT phgDofMapE2D(DOF *dof, SIMPLEX *e, int index)
```

返回 dof 在单元 e 上单元内编号为 index 的自由度在自由度对象中的编号。

```
BTYPE phgDofGetBoundaryType(DOF *u, INT index)
```

获取自由度变量的边界类型，index 是自由度变量的本地编号。

```
BTYPE phgDofGetElementBoundaryType(DOF *u, SIMPLEX *e, int index)
```

获取自由度变量的边界类型，index 是自由度变量的单元内编号。

```
FLOAT *phgDofGetElementCoordinates(DOF *u, SIMPLEX *e, int index)
```

获取自由度变量的坐标位置，index 是自由度变量的单元内编号。

```
void phgDofInitFuncPoint(DOF *dof, SIMPLEX *e, GTYPE type, int index,
                         DOF_USER_FUNC userfunc,
                         DOF_USER_FUNC_LAMBDA userfunc_lambda,
                         const FLOAT *funcvalues, FLOAT *dofvalues)
```

适用于一类特定自由度类型，包括 Lagrange 元、DG 元等，的通用投影函数，可做为这些自由度类型中的 InitFunc，参看 4.1.3。

```
void phgDofInterpC2FGeneric(DOF *dof, SIMPLEX *e, FLOAT **parent_data,
                            FLOAT **children_data)
```

通用粗网格到细网格的插值函数，用于自由度类型中的 InterpC2F 成员，参看 4.1.1。

```
void phgDofInterpF2CGeneric(DOF *dof, SIMPLEX *e, FLOAT **parent_data,
                            FLOAT **children_data)
```

通用细网格到粗网格的插值函数，用于自由度类型中的 InterpF2C 成员，参看 4.1.2。

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

对于自由度对象 u 在单元 e 中的第 bas_index 个局部基函数，如果相应位置的边界标志中不包含 u->DB_mask 中的位，则该函数不做任何处理，函数返回值为 FALSE (此时没用到 mat 和 rhs)。否则，根据 func 给出的边值条件形成边界方程的系数及右端项，并通过 mat 和 rhs 返回给调用程序，函数返回值为 TRUE。

下面用 i 表示 bas_index，n 表示单元基函数的个数 ($n =$ DofGetNBas(u, e))，φ_j 表示第 j 个基函数，u_j 表示相应的自由度，$0 \le j < n$。

对 Dirichlet 边界条件的处理采用直接插值或 L^2 投影。

直接插值要求 u->type->points != NULL，此时隐含假定自由度值就是函数值，边界方程为：

$$u_i = \texttt{func}(x)$$

其中 x 为自由度位置的坐标。

如果 u->type->points == NULL，则边界方程通过 L^2 投影构造，即：

$$\sum_{j=0}^{n-1}\left(\int_V \varphi_j\cdot\varphi_i\,\mathrm{d}V\right)u_j=\int_V \texttt{func}\cdot\varphi_i\,\mathrm{d}V$$

其中 V 表示自由度所在的顶点 (顶点自由度)、边 (边自由度)、面 (面自由度)、或单元 (单元自由度)。

当 u->dim > 1 时，边界方程一共有 u->dim 个，由于所有方程的系数是一样的，因此 mat 中依然只返回 n 个系数，由调用程序根据需要对这些系数进行复制。rhs 中返回 u->dim 个右端项。

参数 proj 指定边界条件的投影方式，目前只能取为 DOF_PROJ_NONE (不投影) 和 DOF_PROJ_CROSS，其中后者仅能用在 u->type->dim % 3 == 0 的情况，用于处理 Maxwell 方程边界条件 $\boldsymbol{u}\times\boldsymbol{n}=\boldsymbol{g}\times\boldsymbol{n}$。

```
NEIGHBOUR_DATA *phgDofInitNeighbourData(DOF *dof, struct MAP_ *map)
```

初始化邻居数据。该函数通过消息传递得到远程邻居中的自由度数据，并将数据存储在一个类型为 NEIGHBOUR_DATA 的结构中。函数返回值为该结构的指针。

```
FLOAT *phgDofNeighbourData(NEIGHBOUR_DATA *nd, SIMPLEX *e, int neighbour,int bas_no,
                           INT *gindex)
```

返回邻居单元中位于与本单元交界面上的自由度数据的指针。如果是本地邻居，则指针直接指向自由度对象数据的相应位置，否则指针指向 nd 中存储邻居数据的相应位置。neighbour 指定哪个邻居 (面编号)。bas_no 指定自由度位置 (顶点、边、面或单元自由度)。gindex 为非空指针时，返回自由度的全局向量编号。

```
int phgDofNeighbourNBas(NEIGHBOUR_DATA *nd, SIMPLEX *e, int face_no,
                        DOF_TYPE **peer_type)
```

返回邻居单元与本单元交界面上的基函数个数 (相当于在邻居单元中调用 phgDofGetBasesOnFace 的返回值)。当 peer_type 为非空指针时，它返回邻居单元上的自由度类型 (用于 hp 支持，目前只允许 DG 类型)。

```
void phgDofReleaseNeighbourData(NEIGHBOUR_DATA **nd_ptr)
```

销毁保存邻居数据的数据结构。

```
int phgDofGetBasesOnFace(DOF *dof, SIMPLEX *e, int face_no,   SHORT bases[])
```

得到自由度对象 dof 在单元 e 的面 face_no 上的基函数 (指在该面上不恒为 0 的单元 e 的基函数)。数组 bases 返回这些基函数在单元内的编号，它们的排列顺序由面的三个顶点的全局编号确定，因此，对共享该面的两个单元调用该函数所得到的两组基函数正好是一一对应的。函数返回值为面上基函数的个数。

A.14 数值积分

这里列出的 PHG 数值积分用户函数是不完整的，它们根据需要随时扩充。

```
QUAD *phgQuadGetQuad1D(int order)
```

返回符合指定精度 (多项式阶) 的一维 (线段上) 求积公式。

```
QUAD *phgQuadGetQuad2D(int order)
```

返回符合指定精度 (多项式阶) 的二维 (三角形中) 求积公式。

```
QUAD *phgQuadGetQuad3D(int order)
```

返回符合指定精度 (多项式阶) 的三维 (四面体中) 求积公式。

```
FLOAT *phgQuadGetFuncValues(GRID *g, SIMPLEX *e, int dim, DOF_USER_FUNC userfunc,
                            QUAD *quad)
```

返回一个静态缓冲区地址，其中包含函数 userfunc 在积分子 quad 的所有积分点处的值，dim 是函数的维数。—

该函数采用了一个简单的 cache 机制，当反复使用同一个函数在同一个单元中同一个积分子的积分点处值时可以避免重复计算。应用示例：phgQuadFuncDotBas。

```
FLOAT *phgQuadGetDofValues(SIMPLEX *e, DOF *u, QUAD *quad)
```

返回一个静态缓冲区地址，其中包含自由度对象 u 在积分子 quad 的所有积分点处的值。

该函数内置 cache 机制，当反复使用同一个自由度对象在同一个单元中同一个积分子的积分点处值时可以避免重复计算 (例如函数 phgQuadDofDotBas)。另外，当 u->type->invariant 为 TRUE 它还可以避免重复计算 u 的基函数 (参看 第五章)。应用示例参看 phgQuadDofDotBas。

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

返回一个静态缓冲区地址，其中包含自由度类型 u->type 的第 n 个基函数在积分子 quad 的所有积分点处的值。

该函数内置 cache 机制，当反复使用同一个自由度类型的基函数在同一个单元中同一个积分子的积分点处值时可以避免重复计算 (例如函数 phgQuadDofDotBas)。另外，当 u->type->invariant 为 TRUE 它还可以避免重复计算 u 的基函数 (参看 第五章)。应用示例参看 phgQuadBasABas。

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

返回一个静态缓冲区地址，其中包含自由度类型 u->type 的第 n 个基函数的梯度在积分子 quad 的所有积分点处的值。

功能及用法与 phgQuadGetBasisValues 类似。应用示例参看 phgQuadGradBasAGradBas。

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

返回一个静态缓冲区地址，其中包含自由度类型 u->type 的第 n 个基函数的 curl 在积分子 quad 的所有积分点处的值。

功能及用法与 phgQuadGetBasisValues 类似。应用示例参看 phgQuadCurlBasACurlBas。

```
FLOAT phgQuadDofDotDof(SIMPLEX *e, DOF *u, DOF *v, int order)
```

计算自由度对象 u 与 v 的乘 (内) 积在单元 e 上的积分。order 指定求积公式的阶数，当它小于 0 或等于 QUAD_DEFAULT 时，表示由函数自动选择适当精度的求积公式。

```
FLOAT phgQuadBasDotBas(SIMPLEX *e, DOF *u, int n, DOF *v, int m,
                       int order)
```

计算单元 e 上自由度对象 u 的第 n 个局部基函数与自由度对象 v 的第 m 个局部基函数的内积在单元 e 上的积分。order 参数的含义同 phgQuadDofDotDof。自由度对象 v 不能为 DOF_CONSTANT 和 DOF_ANALYTIC 类型。

```
FLOAT phgQuadFuncDotBas(SIMPLEX *e, DOF_USER_FUNC userfunc, DOF *u, int n, int order)
```

计算函数 userfunc 与自由度对象 u 的第 n 个基函数的乘 (内) 积在单元 e 上的积分。order 参数的含义同 phgQuadDofDotDof。

```
FLOAT phgQuadGradBasDotGradBas(SIMPLEX *e, DOF *u, int n, DOF *v,         int m,
                               int order)
```

计算自由度对象 u 与自由度对象 v 的局部基函数的梯度的内积在单元 e 上的积分。n 和 m 分别给出 u 和 v 的局部基函数的编号。order 参数的含义同 phgQuadDofDotDof。自由度对象 u 和 v 不能为 DOF_CONSTANT 和 DOF_ANALYTIC 类型。

```
FLOAT phgQuadCurlBasDotCurlBas (SIMPLEX *e, DOF *u, int n,                  DOF *v,
                                int m, int order)
```

计算自由度对象 u 与自由度对象 v 的局部基函数的 curl 的内积在单元 e 上的积分。n 和 m 分别给出 u 和 v 的局部基函数的编号。order 参数的含义同 phgQuadDofDotDof。自由度对象 u 和 v 不能为 DOF_CONSTANT 和 DOF_ANALYTIC 类型。

```
FLOAT *phgQuadDofTimesBas(SIMPLEX *e, DOF *u, DOF *v, int n,         int order,
                          FLOAT *res)
```

计算自由度对象 u 与自由度对象 v 的局部基函数的乘积在单元 e 上的积分。n 给出 v 的基函数编号，计算结果放在 res 指定的缓冲区并返回缓冲区地址 (当 u 为向量函数时计算结果是向量值)。order 参数的含义同 phgQuadDofDotDof。自由度对象 v 不能为 DOF_CONSTANT 和 DOF_ANALYTIC 类型。

```
FLOAT phgQuadDofDotBas (SIMPLEX *e, DOF *u, DOF *v, int n, int order)
```

计算由度对象 u 与由度对象 v 的第 n 个局部基函数的内积在单元 e 上的积分，它要求 v 的基函数是矢量函数，这是与 phgQuadDofTimesBas 不同之处。

```
FLOAT phgQuadDofDotGradBas(SIMPLEX *e,DOF *u,DOF *v, int m, int order)
```

计算自由度 u 的梯度与自由度 v 的第 m 个基函数的梯度的内积在单元 e 上的积分。

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

计算指定自由度对象 u 的 (面的) 跳量，通常用于后验误差估计。它返回的是一个新自由度对象，每个面一个值，该值即是 u 在该面上的跳量的 L^2 模的平方。参数 proj 指定如何对 u 进行投影：DOF_PROJ_NONE 表示不投影、DOF_PROJ_DOT 表示与面单位法向量进行内积、DOF_PROJ_CROSS 表示与面单位法向量进行外积 (叉乘)。order 为积分公式的阶数。

```
DOF *phgQuadFaceJumpN(DOF *u, DOF_PROJ proj, const char *name, int order, DOF *g)
```

计算指定自由度对象 u 的 (面的) 跳量，功能类似 phgQuadFaceJump, 区别在于后者只计算内部面的跳量，不计算边界面的。phgQuadFaceJumpN 计算 Neumann 边界的跳量，自由度对象 g 就是 u 在 Neumann 边界的外法向导数。别的参数意思与 phgQuadFaceJump 相同。

```
FLOAT phgQuadFaceDofDotBas(SIMPLEX *e, int face, DOF *u, DOF_PROJ proj, DOF *v,
                           int N, int order)
```

计算指定面上自由度 u 与 v 的基函数点乘的积分。proj 指定 u 在面外法向上的投影方式，可取 DOF_PROJ_NONE、DOF_PROJ_DOT 或 DOF_PROJ_CROSS。order 指定积分精度。

```
FLOAT phgQuadFaceDofDotDof(SIMPLEX *e, int face, DOF *u, DOF_PROJ proj, DOF *v,
                           int order)
```

计算指定面上自由度 u 与 v 点乘的积分。proj 指定 u 的投影方式，可取 DOF_PROJ_NONE、DOF_PROJ_DOT 或 DOF_PROJ_CROSS。order 指定积分精度。

```
FLOAT phgQuadDofNormP(SIMPLEX *e, DOF *u, int order, int p)
```

计算 $\int_e |u|^p$，order 指定积分精度。

```
FLOAT phgQuadGradBasAGradBas(SIMPLEX *e, DOF *u, int n, DOF *A, DOF *v,  int m,
                             int order)
```

计算自由度 u 第 n 个基函数的梯度乘以系数矩阵 A 点乘自由度 v 的第 m 个基函数的梯度子单元 e 上的积分。当 A 为 NULL 表示其为单位阵，此时该函数等同于 phgQuadGradBasDotGradBas。A 为非空指针时，DofDim(A) 必须等于 1 (表示常数乘以单位阵) 或 DofDim(u) * Dim (表示对角阵) 或 DofDim(u) * Dim 的平方 (表示满阵)，后两种情况尚未彻底测试。

```
FLOAT phgQuadBasABas(SIMPLEX *e, DOF *u, int n, DOF *A, DOF *v, int m, int order)
```

计算自由度 u 第 n 个基函数乘系数矩阵 A 点乘自由度 v 的第 m 个基函数在单元 e 上的积分。当 A 为 NULL 表示其为单位矩阵，此时该函数等同于函数 phgQuadBasDotBas。A 为非空指针时，DofDim(A) 必须等于 1 (表示常数乘以单位阵) 或 DofDim(u) (表示对角阵) 或 DofDim(u) 的平方 (表示满阵)，后两种情况尚未彻底测试。

```
FLOAT phgQuadCurlBasACurlBas(SIMPLEX *e, DOF *u, int n, DOF *A, DOF *v,int m,
                             int order)
```

计算自由度 u 第 n 个基函数的旋度与矩阵 A 的乘积与自由度 v 的第 m 个基函数的旋度在单元 e 上的积分。对 A 的约定及维数限制同函数 phgQuadGradBasAGradBas。

```
FLOAT phgQuadDofAGradBas(SIMPLEX *e, DOF *u, DOF *A, DOF *v, int m, int order)
```

计算自由度 u 的梯度与矩阵 A 的乘积与自由度 v 的第 m 个基函数的梯度的内积在单元 e 上的积分。对 A 的约定及维数限制同函数 phgQuadBasABas，这里要求 DofDim(u) 等于 DofDim(v) * Dim。

A.15 线性解法器

目前的线性解法器接口只是初步的，根据需要会随时改变、调整。

```
INT phgSolverMapD2L(SOLVER *solver, int dof_no, int index)
```

返回 solver 的第 dof_no 个自由度对象中的第 index 个自由度在线性系统中的局部自由度编号。

```
INT phgSolverMapE2L(SOLVER *solver, int dof_no, SIMPLEX *e, int index)
```

返回 solver 中第 dof_no 个自由度对象在单元 e 上的第 index 个自由度在线性系统中的局部自由度编号。

```
int phgSolverInitialize(int *argc, char ***argv)
```

(内部调用)

```
int phgSolverFinalize(void)
```

(内部调用)

```
SOLVER *phgSolverCreate(OEM_SOLVER *oem_solver, DOF *u, ...)
```

创建解法器对象。oem_solver 指定解法器，可取 SOLVER_PETSC、SOLVER_SPC、SOLVER_SUPERLU、SOLVER_HYPRE、SOLVER_LASPACK、SOLVER_PCG、SOLVER_GMRES、SOLVER_AMS 或 SOLVER_DEFAULT。可变参数 (从 u 开始) 给出未知量列表，以 NULL 结束。返回创建的解法器对象指针。

```
int phgSolverDestroy(SOLVER **solver)
```

销毁解法器对象，释放占用的资源。

```
int phgSolverAddMatrixEntry(SOLVER *solver, INT row, INT col,         FLOAT value)
```

将 value 值累加到线性方程组系数矩阵 A[row][col] 上，其中 row 和 col 为局部自由度编号。

```
int phgSolverAddGlobalMatrixEntry(SOLVER *solver, INT row, INT col,
                                  FLOAT value)
```

将 value 值累加到线性方程组系数矩阵 A[row][col] 上，其中 row 和 col 为全局向量编号。

```
int phgSolverAddMatrixEntries(SOLVER *solver, INT nrows, INT *rows,
                              INT ncols, INT *cols, FLOAT *values)
```

将 values 中的 nrows × ncols 个系数值累加到线性方程组系数矩阵中，rows 和 cols 数组分别给出系数矩阵的行和列的局部自由度编号。

```
int phgSolverAddGlobalMatrixEntries(SOLVER *solver, INT nrows, INT *rows, INT ncols,
                                    INT *cols, FLOAT *values)
```

将 values 中的 nrows × ncols 个系数值累加到线性方程组系数矩阵中，rows 和 cols 数组分别给出系数矩阵的行和列的全局向量编号。

```
int phgSolverAddRHSEntry(SOLVER *solver, INT index, FLOAT value)
```

将 value 值累加到线性方程组第 index 个右端项，index 为局部自由度编号。

```
int phgSolverAddGlobalRHSEntry(SOLVER *solver, INT index, FLOAT value)
```

将 value 值累加到线性方程组第 index 个右端项，index 为全局向量编号。

```
int phgSolverAddRHSEntries(SOLVER *solver, INT n, INT *indices,
                           FLOAT *values)
```

将 values 中的 n 个值累加到线性方程组右端项，数组 indices 给出这些右端项的局部自由度编号。

```
int phgSolverAddGlobalRHSEntries(SOLVER *solver, INT n, INT *indices,
                                 FLOAT *values)
```

将 values 中的 n 个值累加到线性方程组右端项，数组 indices 给出这些右端项的全局向量编号。

```
int phgSolverAssemble(SOLVER *solver)
```

组装线性系统。用户程序完成所有对 phgSolverAddMatrixEntry、phgSolverAddMatrixEntries、phgSolverAddRHSEntry 和 phgSolverAddRHSEntries 函数的调用后，可以在求解前调用该函数来完成线性系统的组装。该函数的调用是可选的：必要时 phgSolverSolve 会自动调用它来完成线性系统的组装。

```
int phgSolverDumpCSR(SOLVER *solver, const char *matrix_fn,
                     const char *rhs_fn)
```

将线性系统的系数矩阵及右端项按行压缩稀疏存储格式 (CSR) 输出到指定文件。matrix_fn 给出系统矩阵输出文件名，matrix_fn 给出右端项输出文件名。matrix_fn 或 matrix_fn 可以是空指针，表示不输出相应文件。该函数目前不支持分布式的线性系统。

```
MAT *phgSolverGetMat(SOLVER *solver)
```

获取解法器中的系数矩阵。该函数返回 solver->mat，并将 solver->mat 的引用计数加 1。

```
SOLVER *phgSolverMat2Solver(OEM_SOLVER *oem_solver, MAT *mat)
```

创建一个以 mat 为系数矩阵的解法器，并将 mat 的引用计数加 1。

```
int phgSolverVecSolve(SOLVER *solver, BOOLEAN destroy, VEC *x)
```

求解线性方程组，x 返回方程组的解。该函数主要被其它一些函数如 phgSolverSolve 等调用。

```
int phgSolverSolve(SOLVER *solver, BOOLEAN destroy, DOF *u, ...)
```

求解线性方程组。可变参数 (从 u 开始) 给出未知量列表，以 NULL 结束，它们进入时包含初始解，退出时包含线性系统的解。注意未知量列表中的个数和类型必须与 phgSolverCreate 中的一致。如果 destroy == TRUE 的话，则求解完毕后立即销毁系数矩阵和右端项以释放内存。该函数返回最终迭代次数 (返回值 ≤ 0 表示求解中发生错误)。

```
void phgSolverSetMaxIt(SOLVER *solver, INT user_maxit)
```

设置指定解法器对象的最大迭代次数。若调用该函数则通过命令行设置的最大迭代次数将失效。

```
void phgSolverSetTol(SOLVER *solver, FLOAT user_tol)
```

设置指定解法器对象的迭代终止阀值。若调用该函数则通过命令行设置的迭代终止阀值将失效。

```
void phgSolverSetPC(SOLVER *solver, SOLVER *pc_solver, PC_PROC pc_func)
```

设置解法器对象 solver 的预条件子。用户可以通过该函数指针 PC_PROC pc_func 实现复杂的预条件子。当 pc_func 为 NULL 时，默认调用 phgSolverVecSolve(pc_solver, FALSE, *u) 求解 $Mu = b$, 并将 u 返回, 其中 M 矩阵由 pc_solver 定义。

注：该函数设置的预条件子目前仅对 PCG、GMRES 以及 PETSc 解法器有效，PCG 和 GMRES 默认使用 jacobi 预条件子。

```
int phgJacobiSolver(MAT *A, VEC *b, VEC **x, int max_its, FLOAT rtol)
```

使用 Jacobi 方法，求解 Ax=b, 将结果保存在 x 指向的 VEC 对象（该对象需要在调用该函数之前通过 phgMapCreateVec 创建）。函数返回值为实际迭代次数。max_its 代表最大迭代步数，rtol 表示迭代终止阀值。

```
void phgSolverHypreAMSSetPoisson(SOLVER *solver, DOF *alpha, DOF *beta)
```

（需要 HYPRE 2.0.0 或以上版本支持）定义 HYPRE AMS 预条件子使用中的 A_α 和 A_β 矩阵。

```
void phgSolverHypreAMSSetConstantPoisson(SOLVER *solver, FLOAT alpha, FLOAT beta)
```

同 phgSolverHypreAMSSetPoisson，用于 α 和 β 均为常数的情形。

A.16 特征值及特征向量计算

A.16.1 常量

EIGEN_SMALLEST ▷ 计算最小的 n 个特征值和特征向量
EIGEN_LARGEST ▷ 计算最大的 n 个特征值和特征向量
EIGEN_CLOSEST ▷ 计算最接近 tau 的 n 个特征值和特征向量

A.16.2 外部特征值解法器

phgEigenSolverARPACK ▷ PARPACK 特征值解法器
phgEigenSolverJDBSYM ▷ JDBSYM 特征值解法器
phgEigenSolverBLOPEX ▷ BLOPEX (LOBPCG) 特征值解法器
phgEigenSolverSLEPc ▷ SLEPC 特征值解法器
phgEigenSolverTrilinos ▷ Trilinos/Anasazi 特征值解法器

A.16.3 接口函数

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

计算广义特征值问题 $Ax = \lambda Bx$ 的 n 个特征值/特征向量。which 可以取 EIGEN_SMALLEST (最小特征值)、EIGEN_LARGEST (最大特征值) 和 EIGEN_CLOSEST (最靠近 tau 的特征值)。nit 返回迭代次数，可变参数 u, ... 为一组 DOF 对象的数组，数组的个数和数组中 DOF 对象的类型必须与 A 和 B 中的 DOF 对象一致，每个数组包含 n 个 DOF 对象，返回得到的特征向量。

```
int phgEigenSolve(MAT *A, MAT *B, int n, int which, FLOAT tau,
                  FLOAT *evals, VEC **evecs, int *nit)
```

计算广义特征值问题 $Ax = \lambda Bx$ 的 n 个特征值和特征向量。特征向量通过 *evecs 返回 (如果进入时 *evecs == NULL，则该函数将自动创建一个向量对象)。其余参数与 phgDofEigenSolve 中的对应参数相同。

A.17 几何量的计算与管理

```
void  phgGeomInit(GRID *g)
```

初始化几何数据 (内部调用)。

```
FLOAT phgGeomGetVolume(GRID *g, SIMPLEX *e)
```

返回指定单元的体积。

```
FLOAT phgGeomGetDiameter(GRID *g, SIMPLEX *e)
```

返回指定单元的直径。

```
FLOAT *phgGeomGetJacobian(GRID *g, SIMPLEX *e)
```

返回指定单元的重心坐标的 Jacobian。

```
FLOAT phgGeomGetFaceArea(GRID *g, SIMPLEX *e, int face)
```

返回指定面的面积，face 为面的单元内编号。

```
FLOAT phgGeomGetFaceDiameter(GRID *g, SIMPLEX *e, int face)
```

返回指定面的直径，face 为面的单元内编号。

```
FLOAT *phgGeomGetFaceNormal(GRID *g, SIMPLEX *e, int face)
```

返回指定面的单位法向量，face 为面的单元内编号。

```
FLOAT *phgGeomGetFaceOutNormal(GRID *g, SIMPLEX *e, int face)
```

返回指定面的单位外法向量，face 为面的单元内编号。

```
const FLOAT *phgGeomXYZ2Lambda(GRID *g, SIMPLEX *e, FLOAT x, FLOAT y, FLOAT z)
```

返回指定迪卡尔坐标 (x, y, z) 在单元 e 中的重心坐标。

```
void  phgGeomLambda2XYZ(GRID *g ,SIMPLEX *e, const FLOAT *lambda, FLOAT *x, FLOAT *y,
                        FLOAT *z)
```

返回单元 e 中指定重心坐标的迪卡尔坐标。

参考文献

[1] D. N. Arnold, A. Mukherjee, and L. Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM J. Sci. Comput.*, 22(2):431–448, 2000.

[2] D. Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle. *Int. J. Numer. Method Eng.*, 21:1129–114, 1995.

[3] Pascal J. FREY. Medit: an interactive mesh visualization software. http://www.ann.jussieu.fr/~frey/logiciels/medit.html.

[4] C. F. Gauss. *Methodus nova integralium valores per approximationem inveniendi*, page 163. Reprinted in Werke, Vol. 3. New York: George Olms, 1981.

[5] Christophe Geuzaine and Jean-Franǫis Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. http://www.geuz.org/gmsh/.

[6] H. Kardestuncer, editor. *Finite Element Handbook*. McGraw-Hill, New York, 1987.

[7] I. Kossaczky. A recursive approach to local mesh refinement in two and three dimensions. *J. Comput. Appl. Math.*, 55:275–288, 1994.

[8] Zhang Lin-bo. A parallel algorithm for adaptive local refinement of tetrahedral meshes using bisection. *Numer. Math. Theor. Meth. Appl.*, 2(1):65–89, 2009. http://icmsec/05research_report/0509.pdf.

[9] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM J. Sci. Comput.*, 16(6):1269–1291, 1995.

[10] Walfram Research. Gaussian quadrature. http://mathworld.wolfram.com/Legendre-GaussQuadrature.html.

[11] A. Schmidt and K. G. Siebert. ALBERTA: An adaptive hierarchical finite element toolbox. http://www.alberta-fem.de/.

[12] J. Schöberl, H. Gerstmayr, and R. Gaisbauer. http://www.hpfem.jku.at/netgen/.

[13] Hang Si. Tetgen: a quality tetrahedral mesh generator and three-dimensional delaunay triangulator. http://tetgen.berlios.de/index.html.

[14] A. H. Stroud. *Approximate Calculation of Multiple Integrals*. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[15] Linbo Zhang, Tao Cui, and Hui Liu. A set of symmetric quadrature rules on triangles and tetrahedra. *J. Comp. Math.*, 27(1):89–96, 2009.

函数、变量名索引

A

phgAbort, *61*
Acos, *54*
phgAlloc, *62*
Asin, *54*
Atan, *54*

B

phgBalanceGrid, 68, *69*
BDRY_USER0, 4, 17, 18, *54*, 54
BDRY_USER1, 18
BDRY_USER9, 4, 17, 18, *54*, 54
BOOLEAN, *53*, 62
BTYPE, *54*, 54
BYTE, *53*, 53

C

phgCalloc, *62*
CHAR, *53*, 53
char*phgExportDX, *70*
char*phgExportVTK, *70*
char*phgOptionsGetFilename, *65*
char*phgOptionsGetKeyword, *65*
char*phgOptionsGetString, *65*
phgCheckConformity, *66*
phgCoarsenMarkedElements, *68*
COORD, *54*
Cos, *54*

D

DIAGONAL, *54*, 54
Dim, 4, 26, 29–31, *53*, 54
DIRICHLET, 4, 5, 16–18, *54*, 54, 70
DOF, 53, *54*, 83
DOF_ANALYTIC, 32, *54*, 70, 71, 79
DOF_BASIS_FUNC, 29, *54*
DOF_BASIS_GRAD, 30, *54*
DOF_CONSTANT, 32, *54*, 71, 79
DOF_DEFAULT, 21, *55*
DOF_DG0, *55*, 55
DOF_DGn, *55*
DOF_HB0, 55
DOF_HBn, *55*
DOF_HC0, 55
DOF_HC1, 55
DOF_HC2, 55
DOF_HCn, *55*
DOF_HFEB1, *55*
DOF_HFEB2, *55*
DOF_INIT_FUNC, 28, *54*
DOF_INTERP_FUNC, 26, *54*
DOF_ND1, 31, *55*, 55
DOF_P0, 21, *55*, 55, 70
DOF_P1, 53, *55*
DOF_P2, 21, *55*, 55
DOF_P3, *55*
DOF_P4, *55*
DOF_PROJ_CROSS, 77, *79*, 79, 80
DOF_PROJ_DOT, *79*, 79, 80
DOF_PROJ_NONE, 77, *79*, 79, 80
DOF_TYPE, 29, 34, 35, *54*, 72
DOF_USER_FUNC, 28, *54*, 72, 73
DOF_USER_FUNC_LAMBDA, 28, 29, *54*
phgDofAFXPBY, *75*, 75
phgDofAFXPBY1, *75*
phgDofAXPBY, 74, *75*
phgDofAXPY, 23, 32, 74, *75*
phgDofCopy, 29, *74*
phgDofCurl, *74*
DofData, 31, 32, *71*
DofDim, 29, 30, *71*, 73–75, 80
phgDofDirichletBC, *76*
phgDofDivergence, 23, *74*
phgDofDump, *73*

DofEdgeData, 31, *71*
phgDofEigenSolve, *83*, 84
DofElementData, 23, 31, 53, *71*
phgDofEval, 32, *73*, 73, 74
phgDofEvalCurl, *74*
phgDofEvalDivergence, *73*
phgDofEvalGradient, *73*, 73, 74
DofFaceData, 23, 31, *71*
phgDofFree, *72*
phgDofGetBasesOnFace, *77*, 77
phgDofGetBoundaryType, 31, *76*
DofGetDataCount, *72*, 73
phgDofGetDataCount, *73*
DofGetDataCountGlobal, *72*, 73
phgDofGetDataCountGlobal, *73*
DofGetEdgeDataCount, *72*
DofGetEdgeDataCountGlobal, *72*
phgDofGetElementBoundaryType, 22, 31, *76*
phgDofGetElementCoordinates, *76*
DofGetElementDataCount, *72*
DofGetElementDataCountGlobal, *72*
DofGetFaceDataCount, *72*
DofGetFaceDataCountGlobal, *72*
DofGetVertexDataCount, *72*
DofGetVertexDataCountGlobal, *72*
phgDofGradient, 21, *74*
phgDofInitFuncPoint, 29, *76*
phgDofInitNeighbourData, *77*
phgDofInterpC2FGeneric, 27, 29, *76*
phgDofInterpF2CGeneric, 28, 29, *76*
DofInterpolation, 21, *71*, 72
DofIsOwner, 31, *74*
phgDofMapE2D, *76*
phgDofMatVec, *75*
phgDofMM, ix, *74*, 75
phgDofNeighbourData, *77*
phgDofNeighbourNBas, *77*
phgDofNew, 26, 32, *72*
DofNoAction, 21, *71*, 72
DofNoData, *71*
phgDofNormH1, *76*
phgDofNormInftyVec, *75*
phgDofNormL1, *75*
phgDofNormL1Vec, *75*
phgDofNormL2, *75*
phgDofNormL2Vec, *75*
phgDofReleaseNeighbourData, *77*
phgDofSetDataByFunction, *73*
phgDofSetDataByValue, *73*
phgDofSetDataByValues, *73*
phgDofSetDataByValuesV, 32, *73*
phgDofSetDirichletBoundaryMask, *73*
phgDofSetFunction, 32, *72*
phgDofSetLambdaFunction, 32, *73*
phgDofTransfer, *69*, 69
DofTypeDim, *71*
DofTypeName, *71*
DofTypeOrder, *71*
DofVertexData, 31, *71*

E

EDGE, 28, *54*, 54
EDGE_FLAG, 65
EIGEN_CLOSEST, 51, *83*, 83
EIGEN_LARGEST, 51, *83*, 83
EIGEN_SMALLEST, 51, *83*, 83
phgEigenSolve, 51, *84*
phgEigenSolverARPACK, *83*
phgEigenSolverBLOPEX, *83*
phgEigenSolverJDBSYM, *83*
phgEigenSolverSLEPc, *83*
phgEigenSolverTrilinos, *83*
ELEM_FLAG, 65
ELEMENT, 28, *54*, 54
phgError, *61*
Exp, *54*
phgExportALBERT, *70*
phgExportMedit, *70*

F

Fabs, *53*
FACE, 28, *54*, 54

FACE_FLAG, 65
FALSE, *53*, 53, 57, 62, 76
phgFinalize, *61*
FLOAT, 19, *53*, 53, 54, 63
FLOAT*phgGeomXYZ2Lambda, *84*
ForAllElements, 22, *69*
phgFree, *62*
phgFreeGrid, *65*

G

GEOM_FLAG, 65
phgGeomGetDiameter, 23, *84*
phgGeomGetFaceArea, *84*
phgGeomGetFaceDiameter, 23, *84*
phgGeomGetFaceNormal, *84*
phgGeomGetFaceOutNormal, *84*
phgGeomGetJacobian, *84*
phgGeomGetVolume, *84*
phgGeomInit, *84*
phgGeomLambda2XYZ, *84*
get_cache, *35*, 35, 36
phgGetBoundaryError, *69*
phgGetTime, *61*
GlobalEdge, 5, *69*
GlobalElement, 5, *69*
GlobalFace, 5, *69*
GlobalVertex, 5, *69*
GRID, 5, 6, 53, *54*
GTYPE, *54*

I

phgImport, 15, 17, 18, 65, 66, *70*, 70
phgImportSetBdryMapFunc, 17, *70*
phgImportSetDefaultBdryType, 18, *70*
phgInfo, *61*
phgInit, 39, 40, 53, *61*, 63
INT, 19, *53*, 53, 63
INTERIOR, 4, 5, *54*, 54

J

phgJacobiSolver, *83*

L

Log, 19, *53*

M

MAP, 45, 46, *56*
phgMapCreate, *56*
phgMapCreateMat, *57*
phgMapCreateMatrixFreeMat, *56*
phgMapCreateSimpleMap, *56*
phgMapCreateVec, *58*, 83
phgMapD2L, 46, *56*
phgMapDestroy, 46, *56*
phgMapDofArraysToVec, 47, *57*
phgMapDofToLocalData, 47, *57*
phgMapE2L, 45, *56*
phgMapL2G, 46, *56*
phgMapL2V, 46, *56*
phgMapLocalDataToDof, 47, *57*
phgMapVecToDofArrays, 47, *57*, 57
phgMarkCoarsen, *68*
phgMarkElements, *66*, 68
phgMarkRefine, *68*
MAT, 48, *56*
MAT_OP, *59*, 59
MAT_OP_D, 48, 49, *59*, 59
MAT_OP_N, 48, 49, 57, *59*, 59, 74
MAT_OP_T, 48, 49, *59*, 59, 74
phgMat2Solver, *58*
phgMatAddEntries, 48, *58*
phgMatAddEntry, 48, *57*
phgMatAddGLEntries, *58*
phgMatAddGLEntry, *58*
phgMatAddGlobalEntries, *58*
phgMatAddGlobalEntry, 48, *57*
phgMatAddLGEntries, *58*
phgMatAddLGEntry, *57*
phgMatAssemble, 48, *58*
phgMatAXPBY, 49, *59*
phgMatCreateBlockMatrix, 48, *57*
phgMatDestroy, *58*
phgMatDisassemble, 48, *58*

phgMatDumpMATLAB, *60*
phgMatGetColumnMap, *57*
phgMatGetRowMap, *57*
phgMatRemoveBoundaryEntries, 51, *58*
phgMatVec, 49, *59*
phgMemoryPeakReset, *62*
phgMemoryPeakRestore, *62*
phgMemoryUsage, *62*
MIXED, *54*, 54
MV_FUNC, 48, *56*

N

NEdge, 4, 25, *53*
NEIGHBOUR_DATA, *77*, 77
NEUMANN, 4, 16–18, 53, *54*, 54, 70
phgNewGrid, *65*, 70
NFace, 4, 25, *53*
phgNProcs, *53*
NVert, 4, 25, *53*, 53

O

OEM_SOLVER, *54*, 60
OPPOSITE, *54*, 54
phgOptionsGetFloat, *65*
phgOptionsGetInt, 40, *64*
phgOptionsGetNoArg, 40, *64*
phgOptionsHelp, *64*
phgOptionsPop, 42, *65*
phgOptionsPreset, 39, *63*
phgOptionsPush, 42, *65*, 65
phgOptionsRegisterFilename, *63*
phgOptionsRegisterFloat, *63*
phgOptionsRegisterHandler, *63*
phgOptionsRegisterInt, *63*
phgOptionsRegisterKeyword, *63*
phgOptionsRegisterNoArg, *62*
phgOptionsRegisterString, *63*
phgOptionsRegisterTitle, *63*
phgOptionsSetFilename, *64*
phgOptionsSetFloat, *64*
phgOptionsSetInt, 40, *64*
phgOptionsSetKeyword, *64*
phgOptionsSetNoArg, 40, *64*
phgOptionsSetOptions, 40, *64*
phgOptionsSetString, *64*
phgOptionsShowCmdline, *64*
phgOptionsShowUsed, *64*
OWNER, 6, 7, 31, 45, *54*, 54

P

phgPartitionGrid, *68*
phgPause, *61*
PC_PROC, *61*, 83
phgPerfGetMflops, *62*
PHG_MPI_BYTE, *53*
PHG_MPI_CHAR, *53*
PHG_MPI_FLOAT, 19, *53*
PHG_MPI_INT, 19, *53*
PHG_MPI_SHORT, *53*
PHG_MPI_UINT, *53*
PHG_MPI_USHORT, *53*
Pow, *53*
phgPrintf, 19, *61*

Q

QUAD, 34–36, *54*
QUAD_1D_P1, *55*
QUAD_1D_P10, *55*
QUAD_1D_P11, *55*, 55
QUAD_1D_P2, *55*
QUAD_1D_P3, *55*, 55
QUAD_1D_P4, *55*
QUAD_1D_P5, *55*, 55
QUAD_1D_P6, *55*
QUAD_1D_P7, *55*, 55
QUAD_1D_P8, *55*
QUAD_1D_P9, *55*, 55
QUAD_2D_P1, *55*
QUAD_2D_P10, *55*
QUAD_2D_P2, *55*
QUAD_2D_P3, *55*
QUAD_2D_P4, *55*

QUAD_2D_P5, *55*
QUAD_2D_P6, *55*
QUAD_2D_P7, *55*
QUAD_2D_P8, *55*
QUAD_2D_P9, *55*
QUAD_3D_P1, *56*
QUAD_3D_P10, *56*
QUAD_3D_P2, *56*
QUAD_3D_P3, *56*
QUAD_3D_P4, *56*
QUAD_3D_P5, *56*
QUAD_3D_P6, *56*
QUAD_3D_P7, *56*
QUAD_3D_P8, *56*
QUAD_3D_P9, *56*
QUAD_CACHE, 34, *35*, 35, 36
QUAD_CACHE_LIST, 34, *35*, 35, 36
QUAD_DEFAULT, *34*, 34, 78
phgQuadBasABas, 78, *80*, 80
phgQuadBasDotBas, *78*, 80
phgQuadCurlBasACurlBas, 78, *80*
phgQuadCurlBasDotCurlBas, *79*
phgQuadDofAGradBas, *80*
phgQuadDofDotBas, 78, *79*
phgQuadDofDotDof, *78*, 79
phgQuadDofDotGradBas, *79*
phgQuadDofNormP, *80*
phgQuadDofTimesBas, 22, *79*
phgQuadFaceDofDotBas, *80*
phgQuadFaceDofDotDof, *80*
phgQuadFaceJump, 23, *79*
phgQuadFaceJumpN, *79*
phgQuadFuncDotBas, 78, *79*
phgQuadGetBasisCurl, *78*
phgQuadGetBasisGradient, *78*
phgQuadGetBasisValues, *78*, 78
phgQuadGetDofValues, *78*
phgQuadGetFuncValues, *78*
phgQuadGetQuad1D, *77*
phgQuadGetQuad2D, *78*
phgQuadGetQuad3D, *78*
phgQuadGradBasAGradBas, 78, *80*, 80
phgQuadGradBasDotGradBas, 22, *79*, 80

R

phgRank, *53*, 53
phgRealloc, *62*
phgRedistributeGrid, *68*
phgRefineAllElements, *68*
phgRefineMarkedElements, 53, *68*, 68
REMOTE, 5, *54*, 54

S

phgSetPeriodicDirections, 18, *66*
phgSetPeriodicity, 18, *65*, 66
phgSetVerbosity, *61*
SHORT, *53*, 53
SIMPLEX, 4, 6, 16, *54*
Sin, 19, *54*
SOLVER, 41, *54*
SOLVER_AMS, *60*, 81
SOLVER_DEFAULT, 21, *60*, 81
SOLVER_GMRES, 42, *60*, 81
SOLVER_HYPRE, *60*, 81
SOLVER_LASPACK, *60*, 81
SOLVER_PCG, 42, *60*, 81
SOLVER_PETSC, *60*, 81
SOLVER_SPC, *60*, 81
SOLVER_SUPERLU, *60*, 81
phgSolverAddGlobalMatrixEntries, 41, *81*
phgSolverAddGlobalMatrixEntry, 41, *81*
phgSolverAddGlobalRHSEntries, 41, *82*
phgSolverAddGlobalRHSEntry, 41, *81*
phgSolverAddMatrixEntries, 41, *81*, 82
phgSolverAddMatrixEntry, 41, *81*, 82
phgSolverAddRHSEntries, 41, *82*, 82
phgSolverAddRHSEntry, 41, *81*, 82
phgSolverAssemble, 42, *82*
phgSolverCreate, 21, 41, 60, *81*, 82
phgSolverDestroy, *81*
phgSolverDumpCSR, *82*
phgSolverFinalize, *81*

phgSolverGetMat, *82*
phgSolverHypreAMSSetConstantPoisson, *83*
phgSolverHypreAMSSetPoisson, *83*, 83
phgSolverInitialize, *81*
phgSolverList, *60*
phgSolverMapD2L, *81*
phgSolverMapE2L, 22, 41, *81*
phgSolverMat2Solver, 58, *82*
phgSolverNames, *60*
phgSolverSetMaxIt, *82*
phgSolverSetPC, 42, 43, *83*
phgSolverSetTol, *82*
phgSolverSolve, 42, *82*, 82
phgSolverVecSolve, *82*, 83
SpecialDofType, *71*
Sqrt, 19, *53*
phgSurfaceCut, *69*

T

Tan, *54*
TRUE, 47, 49, *53*, 53, 57, 62, 70, 76, 78

U

UINT, *53*, 53
UNDEFINED, 4, 16–18, *54*, 54, 70
UNKNOWN, *54*, 54
UNREFERENCED, 6, 7, 31, *54*, 54
USHORT, *53*, 53

V

VEC, 47, *56*, 83
phgVecAddEntries, 46, *59*
phgVecAddEntry, 46, *58*
phgVecAddGlobalEntries, 46, *59*
phgVecAddGlobalEntry, 46, *59*
phgVecAssemble, 46, *59*
phgVecAXPBY, *59*, 59
phgVecCopy, *59*
phgVecDestroy, *59*
phgVecDisassemble, 47, *59*
phgVecDotVec, *60*
phgVecDumpMATLAB, *60*
phgVecGetMap, *58*
phgVecNorm1, *60*
phgVecNorm2, *60*, 60
phgVecNormInfty, *60*
VERT_FLAG, 65
VERTEX, 28, *54*, 54

W

phgWarning, *61*

名词索引

B
本地编号, *3*

C
串行映射, *45*

D
单元内编号, *1*, *3*

F
父单元, *1*

J
积分类型, *33*
积分子, *33*
结构体成员说明：, *33*
局部编号, *3*, *45*
局部向量编号, *41*, *45*
局部自由度编号, *41*, *45*

P
phgdoc, 14

Q
切割边, *4*
切割面, *4*
全局编号, *3*
全局向量编号, *41*, *45*

W
网格文件
ALBERT, 15
Medit, 16
Netgen, 16
Tetgen, 16

X
细化边, *1*
向量编号, *45*
新边, *4*
新面, *4*

Y
引用计数, *46*, *47*

Z
子单元, *1*
子网格, *3*
子网格内编号, *3*
自由度对象, *25*
自由度类型, *25*, *34*