A Tutorial on PHG

ZHANG Lin-bo

Academy of Mathematics and Systems Science,
Chinese Academy of Sciences

zlb@lsec.cc.ac.cn

14 March, 2019

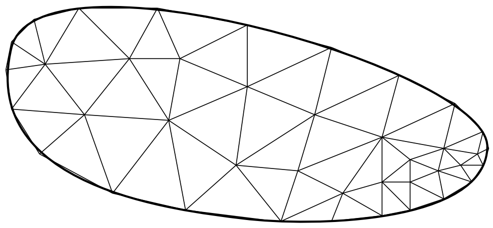
Contents

Part I Introduction

§1.1 Finite element methods

Solving partial differential equations (PDE) using finite element methods generally consists of the following steps:

- Partition the domain into a mesh consisting of elements
- Discretize the PDE on the mesh
- Solve algebraic equations



Main factors affecting accuracy of FE computations:

- Discretization schemes (aka FE spaces)
- Shape and distribution (size) of elements
- Regularity of the solution

§1.2 Example: elleptic boundary value problem

$$\begin{cases} -\operatorname{div}\big(\alpha(x)\,\mathbf{grad}\,u(x)\big) = f(x), & x \in \Omega \\ u(x) = 0, & x \in \partial\Omega \end{cases}$$

§1.2.1 Weak formulation

Find $u \in H_0^1(\Omega)$ which satisfies:

$$\int_\Omega (\alpha\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,v\,\mathrm{d}x = \int_\Omega f v\,\mathrm{d}x, \quad \forall v \in H_0^1(\Omega)$$

§1.2.2 Finite element discretization

Let $\mathscr{M}_h$ be a triangular mesh on Ω, $\mathscr{V}_h \subset H_0^1(\Omega)$ a conforming piecewise polynomial finite element space on $\mathscr{M}_h$. Here h indicates the size of the mesh.

Find $u_h \in \mathscr{V}_h$ which satisfies:

$$\int_\Omega (\alpha\,\mathbf{grad}\,u_h)\cdot\mathbf{grad}\,v_h\,\mathrm{d}x = \int_\Omega f v_h\,\mathrm{d}x, \quad \forall v_h \in \mathscr{V}_h$$

§1.2.3 *A priori* error estimate

If $u \in H^m(\Omega)$ $(m > 1)$, then the error of the numerical solution can be estimated by (Babuška & Suri, 1987; Ainsworth & Senior, 1998):

$$\|u - u_h\|_{H^1(\Omega)} \le C\frac{h^\mu}{p^m}\|u\|_{H^m(\Omega)}$$

where p is the polynomial order of the FE space, $\mu = \min(p, m-1)$.

§1.2.4 The case of linear element

For linear element ($p = 1$), suppose u is smooth, e.g., $u \in H^2(\Omega)$, then we have:

$$\|u - u_h\|_{H^1(\Omega)} \leq Ch\|u\|_{H^2(\Omega)}$$

On a uniform mesh, $h \approx O(N^{-\frac{1}{d}})$, where N is the number of degrees of freedom (DOF), d is the space dimension (2 or 3). Thus when u is smooth we have:

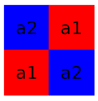
$$\|u - u_h\|_{H^1(\Omega)} \leq CN^{-\frac{1}{d}}\|u\|_{H^2(\Omega)}$$

However if u is not smooth, e.g., $u \in H^{1+\varepsilon}(\Omega)$ ($\varepsilon \ll 1$), then we have:

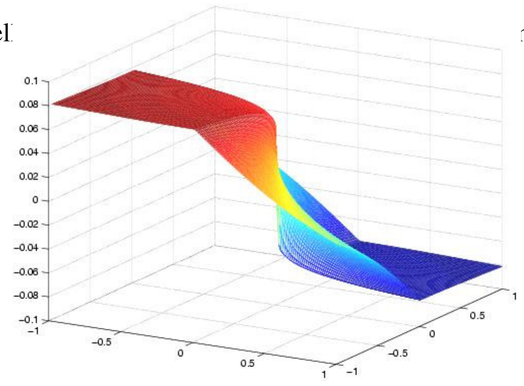
$$\|u - u_h\|_{H^1(\Omega)} \leq CN^{-\frac{\varepsilon}{d}}\|u\|_{H^2(\Omega)}$$

§1.2.5 An example with a singular solution (source: Z. Chen)

$$d = 2,\ \Omega = (-1,1)^2,\ f(x) = 0,\ \alpha = \begin{cases} a_1 = 161.45, & \text{if } x_1x_2 > 0, \\ a_2 = 1, & \text{if } x_1x_2 < 0 \end{cases}$$



The analytic solution (Kel nooth function $\Longrightarrow u \in H^{1+\varepsilon}(\Omega)$ ($\varepsilon < 0.1$)

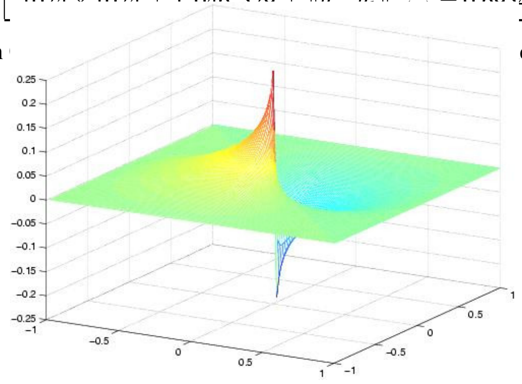


Using uniform meshes the error decay rate is slower than $N^{-0.05}$!

§1.2.6 Some numerical results

Mesh size	DOF (N)	Energy error
128×128	$16,129$	$\|u - u_h\|_{E(\Omega)} = 0.8547$
512×512	$261,121$	$\|u - u_h\|_{E(\Omega)} = 0.7981$
1024×1024	$1,046,529$	$\|u - u_h\|_{E(\Omega)} = 0.6954$

For the energy error to be less than ements!



Error distribution on a uniform mesh

Question: *how to achieve the optimal convergence rate $N^{-\frac{1}{d}}$ for this problem?*

Answer: *use an adaptive finite element method*

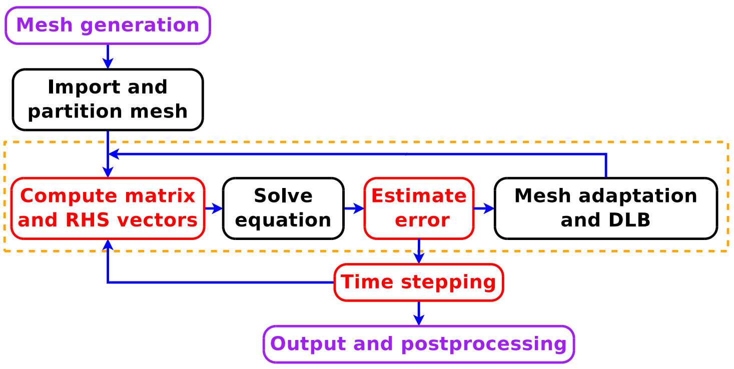
§1.3 Adaptative finite element methods

Adaptive finite element methods (AFEM) consist of repeatedly adjusting the mesh and/or the discretization schemes used on different elements according to some estimated error using the current numerical solution and input data, in order to either:

- improve accuracy for a given amount of computations and memory size, or
- reduce amount of computations and memory size for a given accuracy.

- Principle: *equidistribution of numerical error across elements*
- Means: *adjust either size of elements (h-adaptive) or orders of finite element bases (p-adaptive), or both (hp-adaptive)*
- Ingredients: *local mesh refinement, error estimator (indicator), hierarchical bases for hp-adaptivity*

§1.4 A general diagram of AFEM


Mesh generation
Import and partition mesh
Compute matrix and RHS vectors
Solve equation
Estimate error
Mesh adaptation and DLB
Time stepping
Output and postprocessing

$$\begin{tikzcd}\textbf{Solve} \arrow[r] & \textbf{Estimate} \arrow[r] & \textbf{Mark} \arrow[r] & \textbf{Refine} \arrow[lll, bend left]\end{tikzcd}$$

§1.5 Mesh adaptation methods

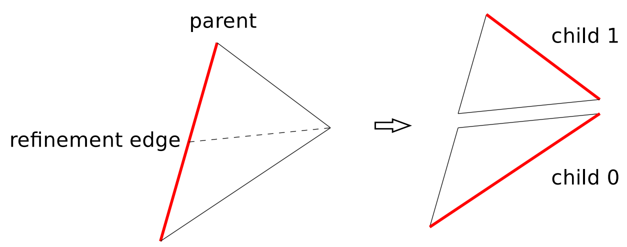
- r-adaptive: change position of mesh points (aka *moving mesh method*)
- h-adaptive: subdivide selected elements
- p-adaptive: change (polynomial) order of bases in each element
- hp-adaptive: h-adaptive and p-adaptive combined

Adaptive finite element methods based on appropriate *a posteriori* error estimates and mesh adaptation strategies can produce quasi-optimal meshes for a given problem.

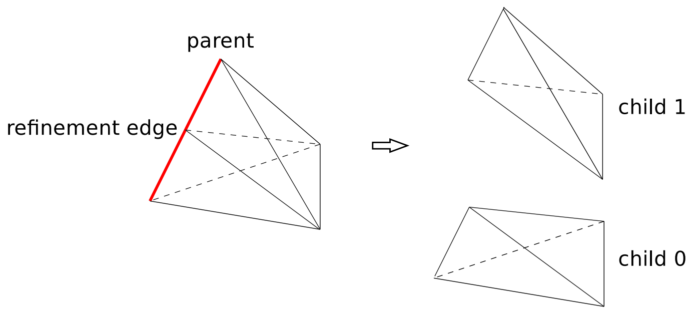
hp-adaptive finite element methods can produce exponential convergence, even the solution may contain singularities.

§1.6 Bisection based element subdivision

Triangle:



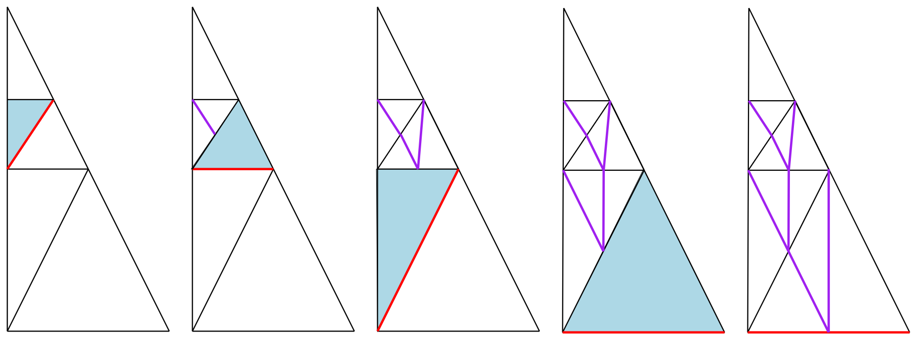
Tetrahedron:



§1.6.1 Algorithms for selecting the bisection edge

- *Longest edge* algorithm: bisect the longest edge
- *Newest vertex* algorithm: bisect the edge opposite to the newest vertex

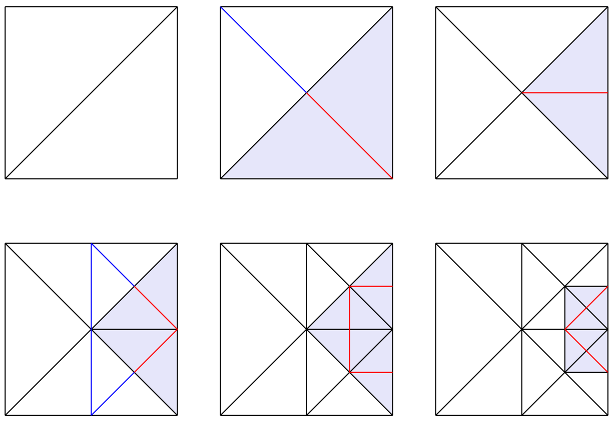
§1.7 Propagation of bisected elements



Neighbouring elements need to be recursively bisected to maintain mesh conformity (red lines representing the bisection edge)

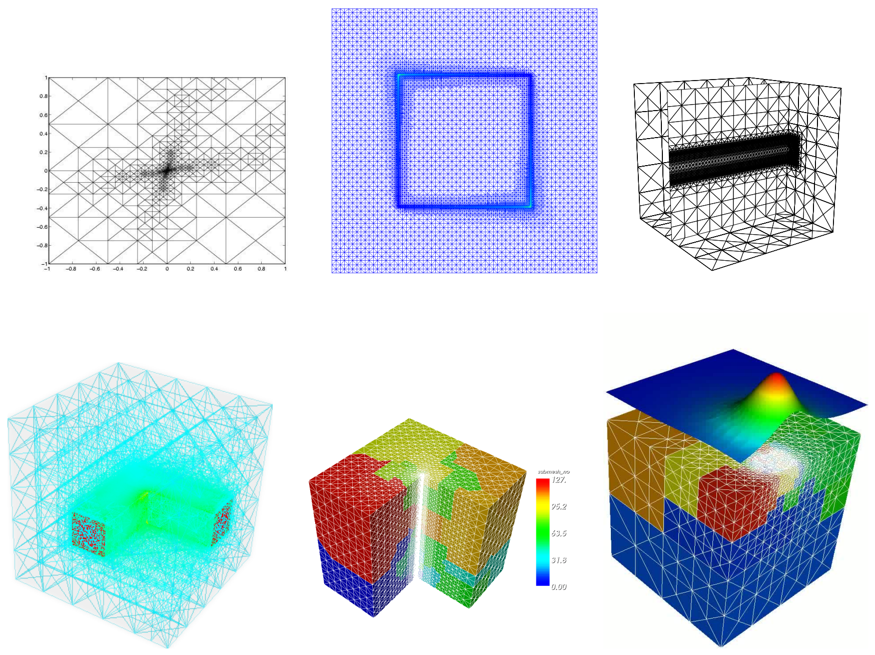
- Finite termination of the propagation process can be theoretically proved for both the longest edge and the newest vertex algorithms.
- Shape regularity of the resulting meshes after repeated refinements has been proved for the newest vertex algorithm in both 2d and 3d.
- For the longest edge algorithm, shape regularity of the resulting meshes can only be proved in the 2d case and is still open in the 3d case!

§1.8 Bisection based local mesh refinement



希望加密的单元用底色标示，红线表示主动加密，蓝线表示被动加密

§1.9 Some examples of adaptive meshes



§1.10 A posteriori error estimate

A residual type a *posteriori* error estimate (Babuska & Miller, 1987):

$$\|u-u_h\|^2_{H^1(\Omega)} \le C \sum_{K\in\mathscr{M}_h} \eta_K^2$$

where K denotes an element and η_K is called the *error indicator* of the element K and is given by:

$$\eta_K^2 = h_K^2 \left\|\Delta u_h + f\right\|^2_{L_2(K)} + \sum_{F\in\partial K, F\not\subset\partial\Omega} h_F \left\|[\alpha\, \mathbf{grad}\, u_h \cdot n_F]_F\right\|^2_{L_2(F)}$$

h_K denotes the diameter of K.

F denotes a face of K, h_F denotes the diameter of F, n denotes the normal vector of F, and $[\cdot]_F$ denotes the jump of a function across F.

§1.11 Marking strategies

A marking strategy is an algorithm for selecting a subset $\mathscr{M}'_h$ of the $\mathscr{M}_h$, which is the set of elements to be refined.

General rule: mark the elements which have larger error indicators.

Some widely used marking strategies:

- MAX strategy: $\mathscr{M}'_h = \{K | K \geq \theta \eta_{\max}\}$, where $\eta_{\max} = \max_{K \in \mathscr{M}_h} \eta_K$.
- GERS strategy (Garanteed Error Reduction Strategy): $\mathscr{M}'_h$ is a subset with fewest elements which satisfies $\sum_{K \in \mathscr{M}'_h} \eta_K^2 \geq \theta \sum_{K \in \mathscr{M}'_h} \eta_K^2$.
- EQDIST strategy (error equidistribution): $\mathscr{M}'_h = \{K | K \geq \theta \frac{\eta}{N^{\frac{1}{2}}}\}$, where $N = |\mathscr{M}_h|$ and $\eta = (\sum_{K \in \mathscr{M}_h} \eta_K^2)^{\frac{1}{2}}$.

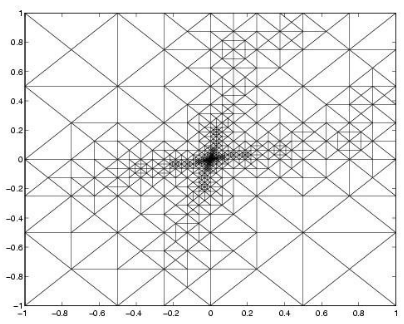
where $\theta \in (0,1)$ is a parameter.

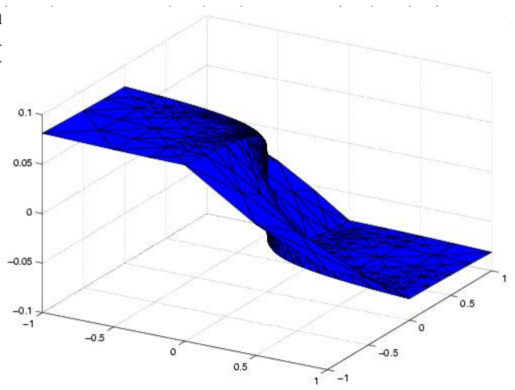
§1.12 Mesh adaptation loop

1. Initial mesh: $\mathscr{M}_0$. Set $n = 0$;
2. Compute the finite element solution u_n on the mesh $\mathscr{M}_n$ and the error indicators $\{\eta_K \mid K \in \mathscr{M}_n\}$. Set $\eta = (\sum_{K \in \mathscr{M}_n} \eta_K^2)^{\frac{1}{2}}$;
3. Stop if η satisfies the convergence criterion;
4. Select a subset of $\mathscr{M}_n$ $\mathscr{M}'_n := \{K \mid \eta_K \text{ is large}\}$ using a given marking strategy;
5. Bisect elements in $\mathscr{M}'_n$ plus some more elements for ensuring mesh conformity to produce a new (refined) mesh $\mathscr{M}_{n+1}$;
6. Set $n := n+1$ and got 2.

§1.13 Optimal convergence of adaptive finite element methods

For solutions with singularities, with adaptive fin achieve the same convergence rate with respect to the number of I





For the previous example using an adaptive finite element method an error decay rate of $N^{-\frac{1}{2}}$ has been achieved and an actual error of 0.07451 has been obtained on a mesh with 2673 DOF using linear element.

§1.14 The toolbox **PHG**

Parallel Hierarchical Grid is a toolbox for writing scalable parallel adaptive finite element programs.

PHG provides functions which perform *common* and *difficult* tasks in parallel adaptive finite element programs, such as:

- management of unstructured parallel (distributed) meshes,
- parallel adaptive mesh refinement and coarsening,
- dynamic load balancing via mesh repartitioning and redistribution,
- efficient, scalable implementation using MPI and OpenMP,
- finite element computations (bases, quadrature, etc.),
- linear/eigen solvers and preconditioners.

§1.15 Main features of **PHG**

- Deals with *conforming* tetrahedral meshes
- Supports AFEM with full *hp adaptivity*
- Adaptive mesh refinement/coarsening based on *newest vertex bisection*
- Fully parallel with transparent *dynamical load balancing*, scalable to thousands of MPI processes
- Transparent MPI+OpenMP *two-level parallelism*
- Stack based *cmdline options database*
- Flexible *linear solver interface* with a rich set of direct or iterative solvers and preconditioners

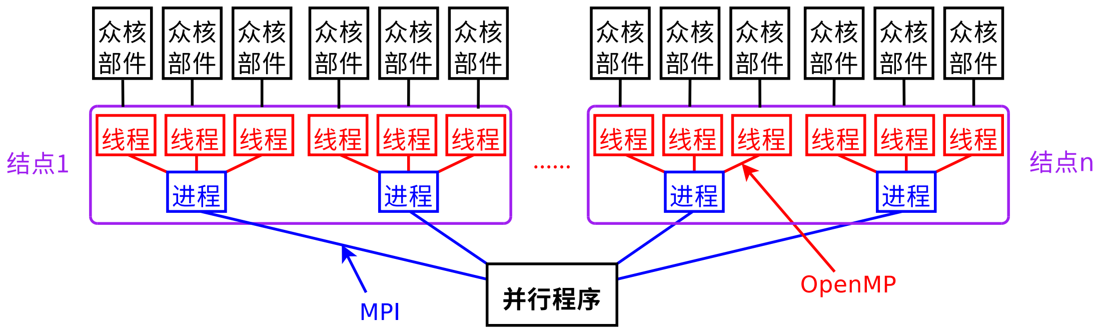
§1.16 Core modules of **PHG**

- Parallel adaptive mesh management
- DOF administration, FE bases and numerical quadrature
- Sparse matrices, linear and eigen solvers
- Management of parameters (the cmdline options interface)

§1.17 Some applications of **PHG**

- Ice-sheet simulation (FSU, SC and AMSS)
- Parasitic extraction: ParAFEMImp/ParAFEMCap (AMSS and Fudan Univ.)
  `http://lsec.cc.ac.cn/~tcui/ParAFEMIMP.tar.gz`
- Structural analysis: **PHG**-Solid (AMSS)
  `http://lsec.cc.ac.cn/phg/download/phg-solid.tar.gz`
- Nonlinear eddy current simulation in power transformers(AMSS and Tian Wei Group Inc.)
- Electronic structure calculation: RealSPACES (AMSS and Fudan Univ.)
- Numerical relativity: CaPHG (LSU) and iPHG (AMSS)
  `https://www.cct.lsu.edu/~jtao/cct/CaPHG/CaPHG.html`
- Biomolecular, ion channel: iChannel (AMSS)
  `http://www.continuummodel.org/`
- Elastic wave propagation simulation (AMSS)

§1.18 Toward exa-scale computation with PHG



Key to exa-scale: many-core acceleration.

- Transparently through many core accelerated preconditioners: DDM+ILU, AMG
- User-coded many core computation such as the seismic simulation code.

Part II Compiling and Installing PHG

§2.1 System requirements

- Minimum requirements:
  - UNIX-like OS (LINUX recommended)
  - C/CXX compiler (C99 compliant, GCC recommended)
  - Windows: MinGW/MSYS (`http://www.mingw.org/`)
- Optional tools:
  - Compilers and MPI: Fortran, Yacc/Lex
  - Math libraries: BLAS, LAPACK, ScaLAPACK, etc. (MKL)
  - System libraries: GMP, libmatheval, PAPI, etc.
  - Solvers: HYPRE, PETSc, Trilinos, MUMPS, SuperLU, etc.
  - Eigen solvers: PARPACK, LOBPCG, SLEPc, Trilinos, etc.
- Linux: RPM packages for the optional tools can be downloaded from: `ftp://159.226.92.111/pub/RPMS`
- Apple OS X: MacPorts is recommended: `https://www.macports.org/`

§2.2 Downloading and configuring PHG

- Downloading: `http://lsec.cc.ac.cn/phg/download.htm`
- Configuring and compiling

  ```
  % tar xjpvf phg-x.x.x-xxxxxxxx.tar.bz2
  % cd phg-x.x.x
  % ./configure
  % gmake
  % gmake install
  ```

  To get cmdline options and variables: `./configure --help`
- Command-line options and environment variables:

  ```
  % ./configure --help
  % env CPPFLAGS="-I/opt/local/include" \
        LDFLAGS="-L/opt/local/lib" \
        ./configure --with-hypre-dir=/usr/local/hypre
  ```

  After running configure, check error messages in `config.log` if something goes wrong.

Part III Programming with PHG

§3.1 Getting started with using PHG

To get started:

- Start by looking at the sample programs, which are closest to the program you intend to write, in the directories `examples/` and `test/`.
- Look at the header files in the directory `include/` for function prototypes and data structures.
- The script `utils/phgdoc` can show function prototypes and macros, for example:

```
% ~/phg/utils/phgdoc phgImport
BOOLEAN phgImport(GRID *g, const char *fn, BOOLEAN distr);
```

  (it mainly works on Linux systems)
- Look at the source code or the comments in the source code, in the directory `src/`, for understanding a function.
- Sample codes: `http://lsec.cc.ac.cn/phg/download/sample.zip`

§3.2 A simple example: `simple.c`

"`simple.c`" is a simple program in `sample.zip` which solves the following Poisson's equation with homogeneous Dirichlet boundary condition:

$$\begin{cases} -\operatorname{div}\left(\mathbf{grad}\left(u(x)\right)\right) = f(x), & x \in \Omega \\ u(x) = 0, & x \in \partial\Omega \end{cases}$$

In the program $f(x) \equiv 1$ and Ω is the unit cube $(0,1)^3$.

Let Ω_h be a conforming tetrahedral mesh on Ω, V_h an $H_0^1(\Omega_h)$ conforming finite element space. Hereafter we will use Ω_h to denote both the polyhedral *domain* covered by the mesh, as well as the *set of elements* in the mesh.

§3.2.1 Finite element discretization

Find $u_h \in V_h$ satisfying:

$$\int_{\Omega_h} \mathbf{grad}\, u_h \cdot \mathbf{grad}\, v_h \,\mathrm{d}x = \int_{\Omega_h} f v_h \,\mathrm{d}x, \quad \forall v_h \in V_h$$

Let $\{\varphi_i \mid i = 1, \ldots, N\}$ be a basis of V_h and $u_h = \sum_{i=1}^N u_i \varphi_i$. Then the equations above are equivalent to:

$$\sum_{i=1}^N \left(\int_{\Omega_h} \mathbf{grad}\, \varphi_i \cdot \mathbf{grad}\, \varphi_j \,\mathrm{d}x \right) u_i = \int_{\Omega_h} f \varphi_j \,\mathrm{d}x, \quad j = 1, \ldots, N$$

Denote $A = [a_{i,j}]_{N\times N}$, $b = [b_1, \ldots, b_N]^T$, $u_h = [u_1, \ldots, u_N]^T$, where:

$$a_{i,j} = \int_{\Omega_h} \mathbf{grad}\, \varphi_i \cdot \mathbf{grad}\, \varphi_j \,\mathrm{d}x \text{ and } b_j = \int_{\Omega_h} f \varphi_j \,\mathrm{d}x$$

A and b are called the *stiffness matrix* and the *load vector*, respectively.

We get the following linear system of equations for u_h:

$$A u_h = b$$

which is to be solved to get the finite element solution.

§3.2.2 Element(-wise) stiffness matrix and load vector

In practice the stiffness matrix and load vector are often computed in an elementwise way. Let e be an element of Ω_h and $\{\varphi_i^e \mid i = 1, \ldots, n\}$ be the local basis (aka element basis) on e, i.e., the set of basis functions with non empty support on e. Denote:

$$A_e = [a_{i,j}^e]_{n\times n}, \quad a_{i,j}^e = \int_e \mathbf{grad}\,\varphi_i^e \cdot \mathbf{grad}\,\varphi_j^e \,\mathrm{d}x,$$
$$b_e = [b_1^e, \ldots, b_n^e]^T, \quad b_j^e = \int_e f \varphi_j^e \,\mathrm{d}x$$

A_e and b_e are called the *element stiffness matrix* and the *element load vector*, respectively.

Then we have:

$$A = \sum_{e\in\Omega_h} P_e A_e P_e^T, \quad b = \sum_{e\in\Omega_h} P_e b_e$$

where P_e is an $N \times n$ matrix of 1s and 0s which maps local (element) basis numbers to global basis numbers (each row of P_e has exactly one 1). A and b can be computed conveniently using the formulae above by looping over the elements in the mesh Ω_h.

The code `simple.c`

```
/*
 * This code solves the following Poisson's equation with zero Dirichlet BC:
 *       - \div (\grad u(x)) = 1
 */

#include "phg.h"

#include <string.h>
#include <math.h>

static void
build_linear_system(SOLVER *solver, DOF *u_h, DOF *f)
{
    GRID *g = u_h->g;
    ELEMENT *e;

    assert(u_h->dim == 1);
    ForAllElements(g, e) {      /* loop on elements */
        int N = DofGetNBas(u_h, e);     /* number of bases in the element */
        int ii, jj;
        INT i, j;
        FLOAT mat, rhs, buf[N];

        for (ii = 0; ii < N; ii++) {    /* loop on bases in current element */
            i = phgSolverMapE2L(solver, 0, e, ii);
            if (phgDofDirichletBC(u_h, e, ii, NULL, buf, &rhs, DOF_PROJ_NONE)) {
                /* current node is on the Dirichlet boundary */
                for (jj = 0; jj < N; jj++) {
                    j = phgSolverMapE2L(solver, 0, e, jj);
                    phgSolverAddMatrixEntry(solver, i, j, buf[jj]);
                }
            }
            else {
                /* current node is not on the Dirichlet boundary */
                for (jj = 0; jj < N; jj++) {
                    j = phgSolverMapE2L(solver, 0, e, jj);
                    mat = phgQuadGradBasDotGradBas(e, u_h, jj, u_h, ii, -1);
                    phgSolverAddMatrixEntry(solver, i, j, mat);
                }
                phgQuadDofTimesBas(e, f, u_h, ii, -1, &rhs);
            }
            phgSolverAddRHSEntry(solver, i, rhs);
        }
    }
}

int
main(int argc, char *argv[])
{
    char *fn = "cube4.dat";     /* input mesh file */
    INT refines = 0;            /* grid refinement levels */

    GRID *g;
    DOF *u_h, *f;
    SOLVER *solver;

    phgOptionsRegisterFilename("-mesh_file", "Mesh file (input)", (char **)&fn);
    phgOptionsRegisterInt("-refines", "Refinement levels", &refines);

    phgInit(&argc, &argv);

    g = phgNewGrid(-1);
    if (!phgImport(g, fn, TRUE))
        phgError(1, "can't read mesh file \"%s\".\n", fn);
    phgRefineAllElements(g, refines);

    /* The numerical solution, default type */
    u_h = phgDofNew(g, DOF_DEFAULT, 1, "u_h", DofInterpolation);
    /* RHS function */
    f = phgDofNew(g, DOF_CONSTANT, 1, "f",  DofNoAction);
    phgDofSetDataByValue(f, 1.0);

    phgPrintf("*** %ld DOF, %ld elements, %d submesh%s\n",
                        (long)DofGetDataCountGlobal(u_h),
                        (long)g->nleaf_global,
                        g->nprocs, g->nprocs <= 1 ? "" : "es");

    /* create solver, with DOF of u_h as its unknowns */
    solver = phgSolverCreate(SOLVER_DEFAULT, u_h, NULL);
    /* compute matrix and RHS */
    build_linear_system(solver, u_h, f);
    /* solve linear system */
    phgSolverSolve(solver, TRUE, u_h, NULL);
    phgPrintf("Solve linear system, nits = %d, residual = %le\n",
                        solver->nits, (double)solver->residual);
    /* destroy solver */
    phgSolverDestroy(&solver);

    phgExportVTK(g, "simple.vtk", u_h, NULL);
    phgPrintf("Solution written to \"simple.vtk\".\n");

    phgDofFree(&u_h);
    phgDofFree(&f);
    phgFreeGrid(&g);

    phgFinalize();

    return 0;
}
```

§3.2.3 Compiling the program

- To compile the program in **PHG**'s source directory:

  Put `simple.c` in either `test/` or `examples/`, and type `gmake simple` in the subdirectory. **PHG** should have been properly configured and compiled.

- To compile the program using an installation of **PHG**:

  ```
  gmake -f install_prefix/share/phg/Makefile.inc simple
  ```

  (or use a `Makefile` which includes `Makefile.inc`)

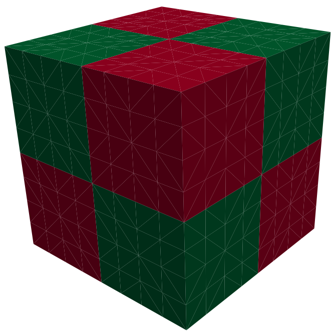
§3.2.4 Running the program

- Get help on available command-line options: `./simple -help`
- Use a solver other than PCG: `./simple -solver gmres`
- Change finite element basis: `./simple -dof_type P5`
- MPI parallel execution: `mpirun -np 8 ./simple -refines 9`

§3.3 A sample code with mesh adaptation: `sample.c`

Consider the following Poisson's equation with a piecewise constant coefficient:

$$\begin{cases} -\operatorname{div}\big(a(x)\,\mathbf{grad}\big(u(x)\big)\big) = f(x), & x \in \Omega \\ u(x) = 0, & x \in \partial\Omega \end{cases}$$



$f(x) \equiv 1$, Ω is the unit cube $(0,1)^3$, and $a(x)$ is a piecewise-constant scalar function with a checkerboard configuration:

$$a(x) = \begin{cases} \alpha, & x \in \text{ red regions} \\ 1/\alpha, & x \in \text{ green regions} \end{cases}$$

where $\alpha > 0$ is a constant.

For this problem the element stiffness matrix becomes:

$$A_e = [a^e_{i,j}]_{n\times n}, \quad a^e_{i,j} = \int_e a(x)\,\mathbf{grad}\,\varphi^e_i \cdot \mathbf{grad}\,\varphi^e_j\,\mathrm{d}x.$$

§3.3.1 A posteriori error estimate

The sample program uses the following *a posteriori* error estimate:

$$\eta^2 = \sum_{e\in\Omega_h} \eta_e^2$$

where

$$\eta_e^2 = h_e^2 \left| \operatorname{div} a(x) \, \mathbf{grad}\, u_h + f \right|_e^2 + \frac{1}{2} \sum_{s\in F(e)\cap F(\Omega_h)} h_s \left| [a(x) \, \mathbf{grad}\, u_h \cdot n_s]_s \right|_s^2$$

- $F(\Omega_h)$ – the set of non-boundary faces in the mesh
- $F(e)$ – the set of faces of the element e
- h_e – the diameter of the element e
- $|\cdot|_e$ – L_2 norm on the element e: $\left(\int_e |\cdot|^2\right)^{\frac{1}{2}}$
- n_s – unit normal vector of the face s
- h_s – the diameter of the face s
- $[\cdot]_s$ – jump across the face s
- $|\cdot|_s$ – L_2 norm on the face s: $\left(\int_s |\cdot|^2\right)^{\frac{1}{2}}$

§3.3.2 Mesh adaptation loop

$$\begin{tikzcd}
\text{Init} \arrow[r] & \text{Solve} \arrow[r] & \text{Estimate} \arrow[r] \arrow[d] & \text{Stop} \\
& \text{Refine} \arrow[u] & \text{Mark} \arrow[l] &
\end{tikzcd}$$

Given an initial mesh Ω^0, the mesh adaptation loop produces a sequence of meshes Ω_h^k, $k = 0, 1, \ldots$.

(0) Import the initial mesh Ω_h^0. Set $k := 0$.
(1) Compute u_h^k on Ω_h^k (Solve)
(2) Compute the error estimate $\{\eta_e^k \mid e \in \Omega_h^k\}$ for u_h^k (Estimate)
(3) Stop if stopping criteria (error tolerance, resource limits, etc.) are met.
(4) Mark a subset of elements to be refined in Ω_h^k (Mark).
(5) Refine the marked elements, plus a few more necessary to ensure mesh conformity, producing a refined mesh which is denoted by Ω_h^{k+1} (Refine).
(6) Set $k := k+1$ and goto (1)

The code `sample.c`

```
/*
 * This code solves the following Poisson's equation with zero Dirichlet BC:
 *        - \div (a(x) \grad u(x)) = f(x)
 *
 * Note: to extract DOFs and errors from the output:
 *       awk '/DOF/ {n=$2} /Error/ {print n, $4}'
 */

#include "phg.h"

#include <string.h>
#include <math.h>
```

```


static void
build_linear_system(SOLVER *solver, DOF *u_h, DOF *a, DOF *f)
{
    GRID *g = u_h->g;
    ELEMENT *e;

    assert(u_h->dim == 1);
    ForAllElements(g, e) {      /* loop on elements */
        int N = DofGetNBas(u_h, e);     /* number of bases in the element */
        int ii, jj;
        INT i, j;
        FLOAT mat, rhs, buf[N];

        for (ii = 0; ii < N; ii++) {    /* loop on bases in current element */
            i = phgSolverMapE2L(solver, 0, e, ii);
            if (phgDofDirichletBC(u_h, e, ii, NULL, buf, &rhs, DOF_PROJ_NONE)) {
                /* current node is on the Dirichlet boundary */
                for (jj = 0; jj < N; jj++) {
                    j = phgSolverMapE2L(solver, 0, e, jj);
                    phgSolverAddMatrixEntry(solver, i, j, buf[jj]);
                }
            }
            else {
                /* current node is not on the Dirichlet boundary */
                for (jj = 0; jj < N; jj++) {
                    j = phgSolverMapE2L(solver, 0, e, jj);
                    mat = phgQuadGradBasAGradBas(e, u_h, jj, a, u_h, ii, -1);
                    phgSolverAddMatrixEntry(solver, i, j, mat);
                }
                phgQuadDofTimesBas(e, f, u_h, ii, -1, &rhs);
            }
            phgSolverAddRHSEntry(solver, i, rhs);
        }
    }
}

static void
estimate_error(DOF *u_h, DOF *a, DOF *f, DOF *error)
/* compute H1 error indicator */
{
    GRID *g = u_h->g;
    ELEMENT *e;
    DOF *agrad, *jump, *residual, *tmp;

    /* tmp = \grad u_h */
    tmp = phgDofGradient(u_h, NULL, NULL, NULL);
    /* agrtad = a \grad u_h */
    agrad = phgDofMM(MAT_OP_N, MAT_OP_N, 1, 3, 1, 1., a, 1, tmp, 0., NULL);
    phgDofFree(&tmp);
    /* jump := \int_f [a\grad u_h \cdot n], one value on each face */
    jump = phgQuadFaceJump(agrad, DOF_PROJ_DOT, NULL, QUAD_DEFAULT);
    residual = phgDofDivergence(agrad, NULL, NULL, NULL);
    phgDofAXPY(1.0, f, &residual);

    ForAllElements(g, e) {
        int i;
        FLOAT eta1, eta2, h;
        /* compute sum of face jumps */
        eta1 = 0.0;
        /* for each face F compute [a\grad u_h \cdot n] */
        for (i = 0; i < NFace; i++) {
            if (e->bound_type[i] & u_h->DB_mask)
                continue;       /* boundary face */
            h = phgGeomGetFaceDiameter(g, e, i);        /* face diameter */
            eta1 += *DofFaceData(jump, e->faces[i]) * h;
        }
        /* compute residual */
        h = phgGeomGetDiameter(g, e);   /* element diameter */
        eta2 = h * h * phgQuadDofDotDof(e, residual, residual, -1);
        *DofElementData(error, e->index) = eta1 * 0.5 + eta2;
    }
    phgDofFree(&jump);
    phgDofFree(&residual);
    phgDofFree(&agrad);

    return;
}

int
main(int argc, char *argv[])
{
    char *fn = "cube4.dat";     /* input mesh file */
    char *vtk = NULL;           /* VTK file (NULL -> don't write VTK file) */
    FLOAT tol = 0.0;            /* convergence criterion */
    INT mem_max = 600;          /* memory (per process) limit (MB) */
    FLOAT alpha = 1000.0;       /* the constant alpha */

    GRID *g;
    ELEMENT *e;
    DOF *u_h, *a, *f, *error;
    SOLVER *solver;
    FLOAT total_error;
    size_t mem;

    phgOptionsRegisterFilename("-mesh_file", "Mesh file (input)", (char **)&fn);
    phgOptionsRegisterFilename("-vtk_file", "VTK file (output)", (char **)&vtk);
    phgOptionsRegisterFloat("-tol", "Convergence criterion", &tol);
    phgOptionsRegisterInt("-mem_max", "Maximum memory (MB)", &mem_max);
    phgOptionsRegisterFloat("-alpha", "alpha", &alpha);

    phgOptionsPreset("-dof_type P4");

    phgInit(&argc, &argv);

    g = phgNewGrid(-1);
    if (!phgImport(g, fn, FALSE))
        phgError(1, "can't read mesh file \"%s\".\n", fn);

    /* The numerical solution, default type */
    u_h = phgDofNew(g, DOF_DEFAULT, 1, "u_h", DofInterpolation);
    phgPrintf("DOF type: %s, mesh file: %s, alpha = %lg\n",
                u_h->type->name, fn, alpha);

    /* The coefficient (material or medium), P0 type (piecewise constant) */
    a = phgDofNew(g, DOF_P0, 1, "a", DofInterpolation);
    ForAllElements(g, e) {
        int k;
        FLOAT x, y, z, barycenter[] = {0.25, 0.25, 0.25, 0.25};
        phgGeomLambda2XYZ(g, e, barycenter, &x, &y, &z);
        k = (x < 0.5 ? 1 : 0) + (y < 0.5 ? 1 : 0) + (z < 0.5 ? 1 : 0);
        *DofElementData(a, e->index) = (k % 2) ? alpha : 1.0 / alpha;
    }

    /* RHS function */
    f = phgDofNew(g, DOF_CONSTANT, 1, "f",  DofNoAction);
    phgDofSetDataByValue(f, 1.0);

    /* DOF for storing error indicators, P0 type (1 indicator per element) */
    error = phgDofNew(g, DOF_P0, 1, "error indicator", DofNoAction);

    while (TRUE) {      /* mesh adaptation loop */
        phgPrintf("*** %ld DOF, %ld elements, %d submesh%s\n",
                    (long)DofGetDataCountGlobal(u_h),
                    (long)g->nleaf_global,
                    g->nprocs, g->nprocs <= 1 ? "" : "es");
        /* This is the only line needed for MPI parallel execution */
        if (phgBalanceGrid(g, 1.2, 1, NULL, 0.))
            phgPrintf("    Repartition mesh, LIF: %lg\n", (double)g->lif);
        /* create solver, with DOF of u_h as its unknowns */
        solver = phgSolverCreate(SOLVER_DEFAULT, u_h, NULL);
        /* compute matrix and RHS */
        build_linear_system(solver, u_h, a, f);
        /* solve linear system */
        phgSolverSolve(solver, TRUE, u_h, NULL);
        phgPrintf("    Solve linear system, nits = %d, residual = %le\n",
                    solver->nits, (double)solver->residual);
        /* destroy solver */
        phgSolverDestroy(&solver);
        /* compute error indicators */
        estimate_error(u_h, a, f, error);
        total_error = Sqrt(phgDofNormL1Vec(error));
        phgMemoryUsage(g, &mem);
        phgPrintf("    Error indicator = %0.3le, mem = %0.2lfMB\n",
                    (double)total_error, (double)mem / (1024.0 * 1024.0));
        if (total_error <= tol || mem > 1024 * (size_t)mem_max * 1024)
            break;
        /* mark elements to be refined */
        phgMarkRefine(MARK_DEFAULT, error, 0.5, NULL, 0., 1,
                                Pow(tol, 2) / g->nelem_global);
        /* refine marked elements */
        phgRefineMarkedElements(g);
    }   /* while */

    if (vtk != NULL) {
        /* output a VTK file for postprocessing/visualization */
        phgPrintf("Write final solution to \"%s\".\n", vtk);
        phgExportVTK(g, vtk, u_h, a, error, NULL);
    }

    phgDofFree(&u_h);
    phgDofFree(&a);
    phgDofFree(&f);
    phgDofFree(&error);

    phgFreeGrid(&g);

    phgFinalize();

    return 0;
}
```

§3.3.3 Compiling the program

- To compile the program in **PHG**'s source directory:

  Put `sample.c` in either `test/` or `examples/`, and type `gmake sample` in the subdirectory. **PHG** should have been properly configured and compiled.

- To compile the program using an installation of **PHG**:

  ```
  gmake -f install_prefix/share/phg/Makefile.inc sample
  ```

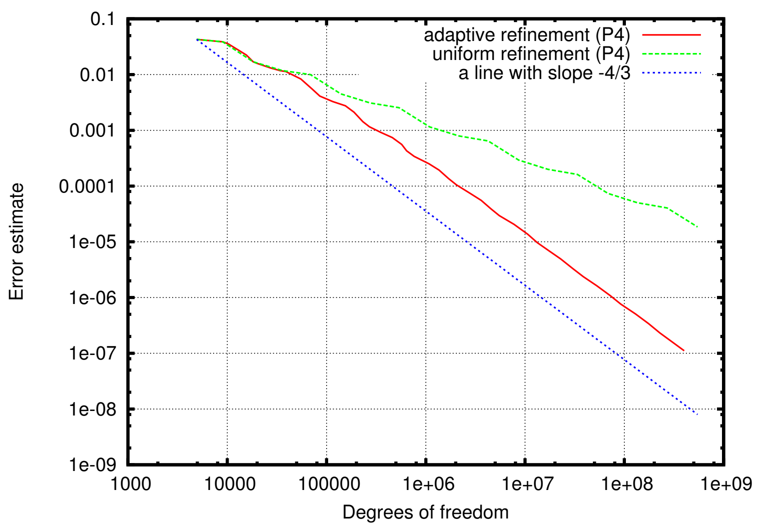
  (or use a `Makefile` which includes `Makefile.inc`)

§3.3.4 Running the program

- Get help on available command-line options: `./sample -help`
- Use a solver other than PCG: `./sample -solver gmres`
- Use uniform mesh refinement: `./sample -strategy all`
- Change finite element basis: `./sample -dof_type P5`
- MPI parallel execution: `mpirun -np 8 ./sample`
- MPI+OpenMP: `env OMP_NUM_THREADS=4 mpirun -np 2 ./sample`

§3.3.5 Sample results obtained with the program

Comparison of uniform refinement vs adaptive refinement



§3.4 Preprocessing: mesh generation

PHG supports the following formats for the input mesh file:

- The ALBERT format – used by the AFEM package ALBERTA:
  `http://www.alberta-fem.de/`
- The Medit format – the *mesh* format described in:
  `http://www.ann.jussieu.fr/frey/publications/RT-0253.pdf`
  It's a widely supported format used by the meshing tool Medit:
  `http://www.ann.jussieu.fr/frey/software.html`
- The Gambit format: the Gambit neutral file format, see, e.g.:
  `http://web.stanford.edu/class/me469b/handouts/gambit_write.pdf`

Recommended mesh generation tools:

- Tetgen: `http://wias-berlin.de/software/tetgen/`
  (see script: utils/tetgen2medit)
- Netgen: `http://sourceforge.net/projects/netgen-mesher/`
  (see script: utils/netgen2medit)

§3.5 Postprocessing and visualization

PHG can output the following file formats for postprocessing or visualization:

- `phgExportALBERT`: the ALBERTA format (mesh only)
  Can be loaded back into **PHG**.
- `phgExportMedit`: the Medit format (mesh only)
  Can be loaded back into **PHG**.
- `phgExportVTK`: the VTK format (mesh and DOFs)
  Can be visualized by visualization tools like Paraview and VisIt:
  - `http://www.paraview.org/`

  - `https://wci.llnl.gov/simulation/computer-codes/visit`
- `phgExportDX`: the IBM Data Explorer format (mesh and DOFs)
  Can be visualized by OpenDX: `http://www.opendx.org/`
- `phgExportEnsight`: the Ensight format (mesh and DOFs).
- `phgMatDumpMATLAB`: output sparse matrices as MATLAB `.m` files.

§3.6 Debugging: what to do if the program fails?

Hints on debugging **PHG** programs:

- Pay attention to compiler warnings and run time error messages.
- For MPI related errors try options "`+mpi_2level`" or "`-mpi_trap_error`".
- Use option "`-log_file tmp`" and examine the error messages in the files `tmp.xxx` after program execution.
- Use option "`-verbosity #`" (with # ≥ 1) to get more verbose outputs.
- Print debugging messages in the program using "`phgInfo(-1,...)`". If the option "`-log_file tmp`" is used then the messages will go into the files `tmp.xxxx`.
- Make the program produce core dumps (e.g., "`ulimit -c unlimited`") and examine them using a debugger like GDB.
- Use option "`-pause`". Then after initialization the program will stop, print PIDs of the processes, and wait for the user to hit "`Enter`" before continuing. The user can then attach debuggers to the running processes.
- Use 3rd party tools, e.g., Valgrind (`http://valgrind.org/`).

Part IV PHG's Application Programming Interface

§4.1 Naming convention and program structure

Basic naming conventions:

- Global function names start with "phg" and with the 1st letter of each word capitalized (e.g.: `phgInit`, `phgSolverCreate`)
- Macro names are like the function names but without the "phg" prefix (e.g.: `GlobalVertex`, `NFace`)
- Letters in data type names are all capitalized (e.g.: `INT`, `GRID`, `ELEMENT`)

Program structure:

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

§4.2 Data types and intrinsic math functions

- Floating-point type: `FLOAT` (configurable, default to `double`)
- Integer types: `INT` (configurable, default to `int`), `SHORT`, `CHAR`, `BYTE`
- Boolean type: `BOOLEAN`
- The `INT` type limits the largest problem size and can be changed with, for example:
  `./configure --with-int="long long"`
- The `FLOAT` type defines the floating point numbers used in PHG and can be changed with, for example,
  `./configure --with-float="long double"`
- Constants matching `FLOAT`: `FLOAT_MAX`, `FLOAT_MIN`, `FLOAT_EPSILON`
- Math functions matching the `FLOAT` type (with 1st letter capitalized):
  `Pow`, `Sqrt`, `Fabs`, `Log`, `Exp`, `Sin`, `Asin`, `Cos`, `Acos`, `Tan`, `Atan`, `Floor`, `Ceil`
- Corresponding `MPI_Datatype` is defined with the prefix `PHG_MPI_`. For example: `PHG_MPI_INT`, `PHG_MPI_FLOAT`
- PHG defines the `printf` conversion specifier for the type `INT` in the macro `dFMT`. For example:
  `INT a; ... printf("a=%"dFMT"\n", a);`

§4.3 Objects and data structures

§4.3.1 The ELEMENT object

```
typedef struct {
    ELEMENT     *children[2];           /* pointers to children */
    void        *neighbours[NFace];     /* neighbours */
    BTYPE       bound_type[NFace];      /* boundary masks */
    INT         verts[NVert];           /* list of vertices */
    INT         edges[NEdge];           /* list of edges  */
    INT         faces[NFace];           /* list of faces */
    INT         index;                  /* element index */
    SHORT       region_mark;            /* material mark */
    SHORT       mark;                   /* refinement mark */
    ... ...
} ELEMENT;
```

§4.3.2 The GRID object

```
typedef struct {
    MPI_Comm    comm;
    int         nprocs, rank;
    FLOAT       lif;            /* load imbalance factor */
    COORD       *verts;         /* coordinates of vertices */
    ELEMENT     *roots;         /* list of root elements */
    ELEMENT     **elems;        /* list of leaf elements */
    INT         nvert, nvert_global;    /* number of vertices */
    INT         nedge, nedge_global;    /* number of edges */
    INT         nface, nface_global;    /* number of faces */
    INT         nelem, nelem_global;    /* number of elements */
    ... ...
} GRID;
```

Functions and macros:

- I/O: `phgNewGrid`, `phgImport`, `phgExportALBERT`, `phgExportMedit`,`phgExportVTK`, `phgExportDX`
- Traversal: `GRID *g; ELEMENT *e; ForAllElements(g,e) { ... }`

§4.4 Mesh manipulation

§4.4.1 Mesh partitioning and dynamic load balancing

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

where:

- `lif_threshold`: threshold for repartitioning.
- `submesh_threshold`: minimum number of elements in a submesh.
- `weights, power`: weights.

§4.4.2 Mesh refinement and coarsening

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

§4.5 Degrees of freedom and finite element bases

§4.5.1 The DOF_TYPE object

The DOF_TYPE object specifies how a dataset is distributed in a mesh, and optionally defines the finite element bases.

```
typedef struct DOF_TYPE_ {
    const char         *name;          /* name of the DOF type */
    SHORT              np_vert;        /* DOF per vertex */
    SHORT              np_edge;        /* DOF per edge */
    SHORT              np_face;        /* DOF per face */
    SHORT              np_elem;        /* DOF per element */
    DOF_TYPE           *grad_type;     /* gradient type */
    SHORT              nbas;           /* # element bases */
    BYTE               order;          /* polynomial order */
    CHAR               continuity;     /* continuity */
    SHORT              dim;            /* dimension of bases */
    ... ...
} DOF_TYPE;
```

To get the list of DOF_TYPEs with a **PHG** program: `./prog -help generic`

§4.5.2 The DOF object

The DOF object holds actual data defined by a DOF_TYPE on a mesh. If the underlying DOF_TYPE also defines finite element bases, then the DOF object can be regarded as a finite element function.

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

Functions for creating and destroying DOFs: `phgDofNew`, `phgDofFree`.

§4.5.3 Macros for accessing DOF data

The DOF object stores DOF data in the order *vertex data*, *edge data*, *face data* and *element data*. The macros `DofData`, `DofVertexData`, `DofEdgeData`, `DofFaceData` and `DofElementData` can be used to access the data, for example:

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

§4.5.4 DOF functions

- phgDofEval, phgDofCopy, phgDofAXPBY, phgDofMatVec, phgDofMM
- phgDofGradient, phgDofDivergence, phgDofCurl

§4.6 Special DOFs

§4.6.1 Constant DOF

Constant DOFs have the type DOF_CONSTANT and represent constant functions.
E.g.: `DOF *one = phgDofNew(g, DOF_CONSTANT, 1, "one", DofNoAction);`

§4.6.2 Analytic DOF

Analytic DOFs have the type DOF_ANALYTIC. They are associated with a function. The following example defines the function $\sin(x)\cos(y)\exp(z)$:

```
static void func(FLOAT x, FLOAT y, FLOAT z, FLOAT *value)
{
    *value = Sin(x) * Cos(y) * Exp(z);
}
DOF *c = phgDofNew(g, DOF_ANALYTIC, 1, "coefficient", func);
```

Note: barycentric coordinates should be used when dealing with discontinuous functions. **PHG** provides a function phgDofSetLambdaFunction to associate an analytic DOF to a function using barycentric coordinates.

§4.6.3 Geometric data

PHG uses an internal DOF called geom to manage and store geometric data which are frequently needed in finite element computations. The following functions can be used to access the geometric data.

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

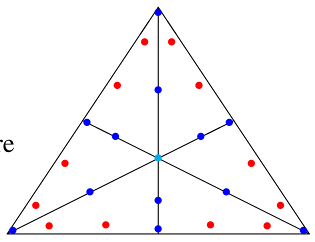
§4.7 Numerical quadrature

An n-point *numerical quadrature rule* is defined as:

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

where w_i are called (unified) quadrature *weights* and x_i are called quadrature *nodes*.

A quadrature rule is said to have *algebraic order p* if it's exact for all polynomials of order p.


A 25-point order-10 rule on triangles

PHG defines numerical quadrature rules on d-simplex for $d = 1$ (line segment), 2 (triangle) and 3 (tetrahedron). The nodes are represented in barycentric coordinates and thus are independent of the element e on which they are defined.

PHG provides numerical quadrature rules of arbitrary orders, either pre-stored or computed on demand. The pre-stored rules are usually *symmetric*, i.e., their set of nodes is invariant under permutations of the barycentric coordinates.

§4.7.1 API for numerical quadrature

- Numerical quadrature rules are defined with the object `QUAD`:

```
typedef struct QUAD_ {
    char *name;         /* name of the quadrature rule */
    int dim;            /* 1: edge, 2: face, 3: tetra */
    int order;          /* algebraic order */
    int npoints;        /* number of points */
    FLOAT *points;      /* list of nodes */
    FLOAT *weights;     /* list of weights */
    SHORT id;           /* id (used with reference count) */
} QUAD;
```

  The nodes are stored in barycentric coordinates, with $d+1$ coordinates for each point, thus the length of the `points` array is $(d+1)\times$ `npoints`.

- Functions for getting a quadrature rule:

```
QUAD *phgQuadGetQuad1D(int order);
QUAD *phgQuadGetQuad2D(int order);
QUAD *phgQuadGetQuad3D(int order);
```

  New rules are created on demand and stored for later reference.

§4.7.2 An example for integration on an edge

The following code computes $\int_{\mathscr{E}} u(x)$, where $\mathscr{E}$ is the `k`-th edge of the element `e` and $u(x)$ is defined by the `DOF` object `u`. An order-4 rule is used.

```
FLOAT lambda[] = {0.,0.,0.,0.}, res, ux, *p, *w;
int i, v0, v1;

v0 = GetEdgeVertex(k, 0);   /* 1st vertex of the edge */
v1 = GetEdgeVertex(k, 1);   /* 2nd vertex of the edge */
QUAD *q = phgQuadGetQuad1D(4); /* order-4 quadrature rule */
p = q->points; w = q->weights;
for (i = 0, res = 0.; i < q->npoints; i++) {
    lambda[v0] = *(p++);
    lambda[v1] = *(p++);
    phgDofEval(u, e, lambda, &ux); /* u(x) on i-th node */
    res += *(w++) * ux;
}
res *= edge length;
```

§4.7.3 An example for integration on a face

The following code computes $\int_{\mathscr{F}} u(x)$, where $\mathscr{F}$ is the k-th face of the element e and $u(x)$ is defined by the DOF object u. An order-4 rule is used.

```
FLOAT lambda[] = {0.,0.,0.,0.}, res, ux, *p, *w;
int i, v0, v1, v2;

v0 = GetFaceVertex(k, 0); /* 1st vertex of the face */
v1 = GetFaceVertex(k, 1); /* 2nd vertex of the face */
v2 = GetFaceVertex(k, 2); /* 3rd vertex of the face */
QUAD *q = phgQuadGetQuad2D(4); /* order-4 quadrature rule */
p = q->points; w = q->weights;
for (i = 0, res = 0.; i < q->npoints; i++) {
    lambda[v0] = *(p++);
    lambda[v1] = *(p++);
    lambda[v2] = *(p++);
    phgDofEval(u, e, lambda, &ux); /* u(x) on i-th node */
    res += *(w++) * ux;
}
res *= phgGeomGetFaceArea(u->g, e, k); /* times face area */
```

§4.7.4 An example for integration on an element

The following code computes $\int_{\mathtt{e}} u(x)$, where e is an element and $u(x)$ is defined by the DOF object u. An order-4 rule is used.

```
FLOAT res, ux, *p, *w;
int i;

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

§4.7.5 Computation of bi- and tri-linear forms

With FEM one often needs to compute bi- and tri-linear forms. E.g.:

$$\int_e \varphi_i \varphi_j, \quad \int_e \nabla\varphi_i \cdot \nabla\varphi_j, \quad \int_e A\nabla\varphi_i \cdot \nabla\varphi_j, \quad \int_e f\varphi_i, \quad \int_f g\varphi_i$$

where φ_i, φ_j are basis functions, f and A are given functions (A may be a scalar function or a 3×3 matrix function). e denotes an element and f denotes a face.

PHG provides functions for computing such kind of integrals. E.g.:

```
phgQuadBasDotBas(ELEMENT *e, DOF *u, int i, DOF *v, int j, int order)
phgQuadGradBasDotGradBas(ELEMENT *e, DOF *u, int i, DOF *v, int i, int order)
phgQuadGradBasAGradBas(ELEMENT *e, DOF *u, int i, DOF *A, DOF *v, int j, int order)
phgQuadDofTimesBas(ELEMENT *e, DOF *f, DOF *u, int i, int order, FLOAT *res)
phgQuadFaceDofDotBas(ELEMENT *e, int k, DOF *f, DOF_PROJ prj, DOF *v, int i, int o)
```

Users often need to write their own functions of similar forms.

§4.7.6 Computing face jumps

Residual type error estimates often involve computing jumps of a certain function $u(x)$ across a face $\mathscr{F}$ in the following form:

$$\int_{\mathscr{F}} \left| [\mathscr{P}(u(x))] \right|^2$$

where $\mathscr{P}(u(x))$ denotes some projection of $u(x)$ onto the face $\mathscr{F}$, e.g., $u(x) \times n$ or $u(x) \cdot n$, where n denotes the unit normal vector of the face. $[\cdot]$ denotes the jump of a function across $\mathscr{F}$, i.e., $[f(x)] = f(x_+) - f(x_-)$, where $f(x_+)$ and $f(x_-)$ denote respectively the values of $f(x)$ on the two sides of $\mathscr{F}$.

PHG provides the following general function for computing face jumps:

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

It returns a `DOF` object named `name`, which has one value per face which is the jump on the face. The argument `proj` can be set to `DOF_PROJ_NONE` (compute jump of u), `DOF_PROJ_DOT` (compute jump of $u \cdot n$) or `DOF_PROJ_CROSS` (compute jump of $u \times n$).

§4.8 Command-line options

PHG provides command-line options (or simply options), which can be specified on the command-line when running a program for controlling program behaviour or defining parameters.

- If an option has an argument, it can be invoked as either '`-opname arg`' or '`-opname=arg`', with optional spaces around '='.
- If an option has a string argument, it can also be invoked as '`-opname+=str`', in this case the string '`str`' is appended to the current value.
- An option without argument can also be invoked as '`+opname`', with an effect opposite to '`-opname`'.
- Curl braces can be used to group the argument to facilitate nesting of options, for example:
  `-pcg_pc_opts="{-solver hypre -hypre_pc boomeramg}"`
- Cmdline options are processed by `phgInit`. After calling `phgInit`, all cmdline options are removed from `argc` and `argv`.
- New options can be defined with `phgOptionsRegisterXXXX`, but only before calling `phgInit`.

§4.8.1 Types of options

Each option has a matching variable, specifying an option sets the value of the variable. The types of options provided by **PHG** are listed below.

Option type	Argument type	Variable type	phgOptionsRegister$x$
boolean	none	BOOLEAN	$x=$ NoArg
integer	integer	INT	$x=$ Int
float	float	FLOAT	$x=$ Float
string	string	char*	$x=$ String
filename	string	char*	$x=$ Filename
keywords	keywords	int	$x=$ Keyword
handler	string	func. ptr	$x=$ Handler

§4.8.2 The option `-oem_options`

This is a special option. Its argument is passed as command line options to 3rd-party softwares such as PETSc and HYPRE. Unlike other string options, the argument of '`-oem_options`' is always appended, i.e., as if '`-oem_options+=arg`' was specified.

§4.8.3 Specification and processing of options

Below are the various ways in which options can be specified, listed in the order in which they are processed. Note that if an option is specified multiple times, the last specification is effective.

- in the program with the `phgOptionsPreset` function (before `phgInit`),
- in the environment variable `PHG_OPTIONS`
- in the file '`progname.options`',
- in the files specified by '`-options_file filename`' options,
- in the command-line,
- in the program with the `phgOptionsSetXXXX` function.

If the option '`-help`' is given, the program will list all options and exit. Use '`-help all`' to list all options and '`-help category`' to list options in a given category (e.g., '`-help solver`').

§4.8.4 Using options in the program

Options can be set in a program. It provides a convenient way of setting some internal parameters without needing an API for them.

PHG provides an options stack for saving and restoring the current state of the options database, which is very useful in this aspect. For example:

```
SOLVER *aux_solver; char *aux_opts = NULL;
phgOptionsRegisterString("-aux_solver_opts", "Aux solver options", &aux_opts);
phgOptionsPush();
phgOptionsSetOptions("-solver hypre -hypre_solver boomeramg -hypre_pc none "
        "-solver_atol 0 -solver_rtol 0 -solver_btol 0 -solver_maxit 1");
phgOptionsSetOptions(aux_opts);
aux_solver = phgSolverCreate(......);
phgOptionsPop();
```

```
./myprog -aux_solver_opts "-solver mumps"
```

§4.9 Linear solver

Linear systems of equations are described and solved with the SOLVER object, which mainly consists of three parts:

```
typedef struct {
    OEM_SOLVER  *oem_solver;    /* the actual solver */
    MAT         *mat;           /* matrix */
    VEC         *rhs;           /* RHS */
    SOLVER_PC   *pc;            /* preconditioner */
    ... ...
} SOLVER;

SOLVER *phgSolverCreate(OEM_SOLVER *oem_solver, DOF *u, ...);
SOLVER *phgSolverMat2Solver(OEM_SOLVER *oem_solver, MAT *mat);
```

The OEM_SOLVER object describes the actual solver used in a SOLVER object, which corresponds to either an external solver (package), or a built-in solver.

All external solvers are optional. The command-line option '-help solver' can be used to list available solvers in a installation.

§4.9.1 List of some OEM_SOLVERs implemented in PHG

OEM Solver	Package, origin	Type	MPI	External PC
SOLVER_PCG	PCG, PHG	iterative	yes	yes
SOLVER_GMRES	PGMRES, PHG	iterative	yes	yes
SOLVER_PREONLY	Precon. only, PHG	iterative	yes	no
SOLVER_MINRES	MINRES, Stanford/USA	iterative	yes	yes
SOLVER_PETSC	PETSc, ANL/USA	iterative	yes	yes
SOLVER_HYPRE	Hypre, LLNL/USA	iterative	yes	no
SOLVER_TRILINOS	Trilinos, SNL/USA	iterative	yes	no
SOLVER_MUMPS	MUMPS, Inria/Europe	direct	yes	no
SOLVER_PASTIX	PasTix, Inria/Europe	direct	yes	no
SOLVER_SPOOLES	DARPA/USA	direct	yes	no
SOLVER_SUPERLU	SuperLU, LBL/USA	direct	yes	no
SOLVER_PARDISO	PARDISO, Europe	direct	no	no
SOLVER_SSPARSE	SuiteSparse, Florida	direct	no	no

Note: "External PC" means the solver spports an external preconditioner.

When SOLVER_DEFAULT is used in place of an OEM_SOLVER, the actual solver used is controlled by the command-line option '-solver'.

§4.9.2 Preconditioners (PC)

At present PHG only implements preconditioners of the form $B^{-1}Ax = B^{-1}b$ (left preconditioners), where B (or B^{-1}) is called the preconditioning matrix.

PHG's preconditioner is handled by the object SOLVER_PC, which has the following form:

```
typedef void (*PC_PROC)(void *ctx, VEC *r, VEC **x);

typedef struct SOLVER_PC_ {
    char        *name;
    PC_PROC     pc_proc; /* function performing the PC */
    void        *ctx;    /* context data (passed to PC_PROC) */
    ... ...
} SOLVER_PC;
```

where `pc_proc` is the function performing actual preconditioning process, in which the input vector `r` is the residual of the current approximate solution, and the output vector `*x` contains the result of $B^{-1}r$.

The `ctx` member points to user defined data for use by `pc_proc` It is often a pointer to a `SOLVER` object. It is passed to `pc_proc` as an argument.

Some solvers have built-in preconditioners, which can be selected with command line options.

Four of the solvers, namely PCG, GMRES, MINRES, Preonly and PETSc, support the so called *external preconditioners*, in which another solver is used as the preconditioner.

A preconditioner can be specified in two ways:

(1) with the function `phgSolverSetPC` (external preconditioners only)
(2) with the options `-xxx_pc_type`, `-xxx_pc_solver` and `-xxx_pc_opts`.

Command-line options take precedence over the function `phgSolverSetPC`.

If a solver has the `-xxx_pc_type` option, then '`-xxx_pc_type solver`' must be used in conjuntion with `-xxx_pc_solver` and `-xxx_pc_opts` for the latters to take effect. For example:

```
./sample -solver pcg -pcg_pc_type solver -pcg_pc_solver mumps
```

PHG has some built-in solvers which are intended to be only used as preconditioners, including

- `SOLVER_ASM` (Additive Schwarz Method),
- `SOLVER_AMS` (Auxiliary space Maxwell Solver) and
- `SOLVER_SMOOTHER` (smoothers).

§4.10 Map, matrix and vector

§4.10.1 Map

A `MAP` object describes the partitioning of a distributed vector, as well as the map between vector components and degrees of freedom.

- Simple map: `MAP *phgMapCreateSimpleMap(GRID *g, INT m, INT M)`
  Create a `MAP` of global size `M` and local size `m`.
- DOF based map: `MAP *phgMapCreate(DOF *u, ..., NULL)`
  Create a `MAP` for the list of `NULL` terminated `DOF`s.

There are 3 types of numberings in a map:

(1) Local numbering: for a simple map it is $0,\dots,\mathtt{m}-1$; for a `DOF` based map it is the concatenated `DOF` numbering.
(2) Local vector numbering: continuous numbering of local vector components. For a simple map it is the same as local numbering.

(3) Global numbering: continuous numbering of all vector components with respect to process rank and local vector numbering.

§4.10.2 Mapping of different numberings

- `INT phgMapD2L(MAP *map, int dof_no, INT index)`
  (DOF number, number within the DOF) to local number
- `INT phgMapE2L(MAP *map, int dof_no, ELEMENT *e, int index)`
  (DOF number, element, number within the element) to local number
- `INT phgMapL2V(MAP *map, INT index)`
  Local number to local vector number
- `INT phgMapL2G(MAP *map, INT index)`
  Local number to global vector number

§4.10.3 Data transfer between DOFs and vector

- `phgMapDofToLocalData` and `phgMapLocalDataToDof`
- `phgMapDofArraysToVec` and `phgMapVecToDofArrays`,
  `phgMapDofArraysToVecN` and `phgMapVecToDofArraysN`

§4.10.4 The reference count of a map

Since a map may be used in multiple matrices and vectors, a *reference count* is maintained. Each time the map is reference by another object, the reference count is increased by 1. When the function `phgMapDestroy` is called, if the reference count is not zero, then the reference count is decremented by 1 and the map is not destroyed. So the following code is erroneous:

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

The correct code should look like:

```
    MAT *mat = phgMatCreate(phgComm, m, M);
    MAP *map = phgMatGetRowMap(mat); /*  ref. count incremented */
    ... ...
    phgMatDestroy(&mat);
    phgMapDestroy(&map);
```

§4.10.5 Vector

PHG's distributed vectors are handled by the object `VEC`.

```
typedef struct VEC_ {
    struct MAP_ *map;
    FLOAT       *data, *offp_data;
    ... ...
} VEC;

VEC *phgVecCreate(GRID *g, INT m, INT M, int nvec)
VEC *phgMapCreateVec(MAP *map, int nvec)
```

```
void phgVecDestroy(VEC **vec_ptr)
void phgVecAddEntry(VEC *vec, int which, INT index, FLOAT value)
void phgVecAddEntries(VEC *vec, int which, INT n, INT *indices, FLOAT *values)
void phgVecAddGlobalEntry(VEC *vec, int which, INT index, FLOAT value)
void phgVecAddGlobalEntries(VEC *vec, int which, INT n, INT *indices, FLOAT *values)
void phgVecAssemble(VEC *vec)
```

§4.10.6 Matrix

PHG's sparse matrices are distributed row-wise and stored in CSR format.

```
typedef struct MAT_ {
    MAP         *rmap, *cmap;
    MAT_TYPE    type;   /* packed, unpacked, dense, matrix-free */
    ... ...
} MAT;

MAT *phgMapCreateMat(MAP *rmap, MAP *cmap)
MAT *phgMatCreate(MPI_Comm comm, INT m, INT M)
MAT *phgMatCreateNonSquare(MPI_Comm c, INT m, INT M, INT n, INT N)
void phgMatDestroy(MAT **Mat)

phgMatAddEntry          phgMatAddEntries
phgMatAddGlobalEntry    phgMatAddGlobalEntries
phgMatAddGLEntry        phgMatAddGLEntries
phgMatAddLGEntry        phgMatAddLGEntries

phgMatSetEntry          phgMatSetEntries
... ...                 ... ...
```

§4.10.7 Matrix-free matrix

A matrix-free matrix is a special type of the `MAT` object in which the matrix entries are not stored, instead a pointer to a function which performs the matrix-vector multiplication operation (`MV_FUNC`) is provided.

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

The function `mv_func` performs $y := \mathrm{op}(A)x$. It takes the following form:

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

The matrix-free matrix is the equivalent of PETSc's *shell matrix*.

§4.10.8 Block matrix

PHG's block matrix is implemented as a special kind of matrix-free matrix.

```
MAT *phgMatCreateBlockMatrix(MPI_Comm comm, int p, int q,
                MAT *pmat[], FLOAT coeff[], MAT_OP trans[]);
```

The function above created a block matrix which consists of $\mathtt{p} \times \mathtt{q}$ submatrices in the following form:

$$\begin{pmatrix} \mathtt{B[1*p-p]} & \cdots & \mathtt{B[1*p-1]} \\ \mathtt{B[2*p-p]} & \cdots & \mathtt{B[2*p-1]} \\ \vdots & \vdots & \vdots \\ \mathtt{B[q*p-p]} & \cdots & \mathtt{B[q*p-1]} \end{pmatrix}$$

where $\mathtt{B[i]} = \mathtt{coeff[i]} * \mathtt{trans}(\mathtt{pmat[i]})$.

The arrays `pmat`, `coeff` and `trans` are all of length `p*q` and are respectively the list of pointers to the submatrices, the list of coefficients and the list of transpose operations (`MAT_OP_N` or `MAT_OP_T`).

Block matrices are allowed to be nested, i.e., the blocks can themselves be block matrices.