

NISTIR 7374

# PHAML User's Guide

---

William F. Mitchell  
U. S. Department of Commerce  
Technology Administration  
National Institute of Standards and Technology  
Information Technology Laboratory  
Gaithersburg, MD 20899 USA

October 2006



# PHAML User's Guide, Version 1.0

William F. Mitchell\*  
Mathematical and Computational Sciences Division  
100 Bureau Drive Stop 8910  
National Institute of Standards and Technology  
Gaithersburg, MD 20899-8910  
email: phaml@nist.gov

## Abstract

PHAML (Parallel Hierarchical Adaptive MultiLevel) is a Fortran module for the solution of elliptic partial differential equations. It uses finite elements, adaptive grid refinement ( $h$ ,  $p$  or  $hp$ ) and multigrid solution techniques in a message passing parallel program. It has interactive graphics via OpenGL. This document is the user's guide for PHAML. The first part tells how to obtain any needed software, how to build and test the PHAML library, and how to compile and run the example programs. The second part explains the use of PHAML, including program structure and the various options that are available. The third part is a reference manual which describes the API (application programming interface) of PHAML. The reference manual begins with a 2 page Quick Start section for the impatient.

**Keywords:** adaptive grid refinement, dynamic load balancing, elliptic eigenvalue problems, elliptic partial differential equations, high order finite elements, hp-adaptivity, multigrid, parallel programming

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Software</b>	<b>5</b>
2.1	Obtaining Software . . . . .	5
2.1.1	PHAML . . . . .	5
2.1.2	Fortran 90 and C Compilers . . . . .	5

---

\*Contribution of NIST, not subject to copyright in the United States. The mention of specific products, trademarks, or brand names is for purposes of identification only. Such mention is not to be interpreted in any way as an endorsement or certification of such products or brands by the National Institute of Standards and Technology. All trademarks mentioned herein belong to their respective owners.

2.1.3	BLAS and LAPACK . . . . .	6
2.1.4	PVM and MPI . . . . .	6
2.1.5	OpenGL (or Mesa), GLUT and f90gl . . . . .	6
2.1.6	Triangle . . . . .	7
2.1.7	ARPACK . . . . .	7
2.1.8	hypre . . . . .	7
2.1.9	MUMPS . . . . .	7
2.1.10	PETSc . . . . .	8
2.1.11	SuperLU . . . . .	8
2.1.12	Zoltan . . . . .	8
2.2	Compiling PHAML . . . . .	8
2.2.1	Creating the Makefiles . . . . .	8
2.2.2	Compiling the Library . . . . .	11
2.3	Testing the Library . . . . .	11
2.4	Compiling and Running the Examples . . . . .	12
<b>3</b>	<b>Scalar Linear Elliptic Boundary Value Problems</b>	<b>14</b>
3.1	Main program . . . . .	14
3.1.1	Parallelism . . . . .	14
3.1.2	Program structure . . . . .	15
3.2	Defining the problem . . . . .	16
3.2.1	Defining the PDE . . . . .	16
3.2.2	Defining the boundary conditions . . . . .	18
3.2.3	Defining the domain and initial grid . . . . .	20
3.2.4	The true solution . . . . .	22
3.3	Solution method . . . . .	22
3.3.1	Discretization . . . . .	23
3.3.2	Refinement . . . . .	24
3.3.3	Error indicator . . . . .	25
3.3.4	Linear system solver . . . . .	26
3.3.5	Load balancing . . . . .	31
3.3.6	Termination . . . . .	33
3.4	I/O . . . . .	33
3.4.1	I/O files . . . . .	33
3.4.2	Printed I/O . . . . .	34
3.4.3	Pausing . . . . .	35
3.5	Graphics . . . . .	36
3.5.1	Overview . . . . .	36
3.5.2	Example visualizations . . . . .	37
3.5.3	View modifier . . . . .	41
3.5.4	Colors . . . . .	44
3.5.5	Functions . . . . .	45
3.5.6	Lights . . . . .	46
3.5.7	Contour plots . . . . .	46
3.5.8	Multiple solutions . . . . .	47
3.5.9	Miscellaneous features . . . . .	47

3.5.10	Development aids . . . . .	49
3.5.11	Postscript . . . . .	49
3.6	Post-solution utilities . . . . .	49
3.6.1	Store and Restore . . . . .	49
3.6.2	Query . . . . .	50
3.6.3	Solution evaluation . . . . .	50
3.6.4	Functionals . . . . .	50
<b>4</b>	<b>Problem Extensions</b>	<b>51</b>
4.1	Eigenvalue Problems . . . . .	51
4.2	Coupled Systems or Multicomponent Solutions . . . . .	52
4.3	Parabolic, Nonlinear, Etc. Problems . . . . .	52
<b>5</b>	<b>Examples</b>	<b>54</b>
<b>6</b>	<b>Release notes</b>	<b>54</b>
<b>7</b>	<b>Reference Manual</b>	<b>55</b>
7.1	Quick Start . . . . .	55
7.1.1	Obtaining the software . . . . .	55
7.1.2	Compiling the PHAML library . . . . .	55
7.1.3	Compiling an Example . . . . .	56
7.1.4	Running the Example . . . . .	56
7.1.5	Now what? . . . . .	56
7.2	Public Entities in PHAML . . . . .	56
7.2.1	phaml_solution_type . . . . .	56
7.2.2	my_real . . . . .	57
7.2.3	pde and my_pde.id . . . . .	57
7.2.4	symbolic constants . . . . .	57
7.3	User Provided Routines . . . . .	57
7.3.1	bconds . . . . .	58
7.3.2	boundary_point . . . . .	58
7.3.3	boundary_npiece . . . . .	59
7.3.4	boundary_param . . . . .	59
7.3.5	iconds . . . . .	59
7.3.6	pdecoefs . . . . .	60
7.3.7	phaml_integral_kernel . . . . .	60
7.3.8	trues . . . . .	61
7.3.9	truexs . . . . .	61
7.3.10	trueys . . . . .	61
7.3.11	update_usermod . . . . .	62
7.4	PHAML procedures . . . . .	62
7.4.1	phaml_compress . . . . .	62
7.4.2	phaml_connect . . . . .	62
7.4.3	phaml_copy_soln_to_old . . . . .	64
7.4.4	phaml_create . . . . .	64

7.4.5	phaml_destroy	66
7.4.6	phaml_evaluate	66
7.4.7	phaml_evaluate_old	67
7.4.8	phaml_integrate	67
7.4.9	phaml_pclose	68
7.4.10	phaml_popen	68
7.4.11	phaml_query	69
7.4.12	phaml_restore	71
7.4.13	phaml_scale	72
7.4.14	phaml_solve_pde	72
7.4.15	phaml_store	88

## 1 Introduction

To start using PHAML immediately, see the Quick Start guide in Section 7.1.

PHAML stands for Parallel Hierarchical Adaptive MultiLevel method. It solves systems of linear elliptic partial differential equations (PDEs) of the form

$$-\frac{\partial}{\partial x}(p(x,y)\frac{\partial u}{\partial x}) - \frac{\partial}{\partial y}(q(x,y)\frac{\partial u}{\partial y}) + r(x,y)u = f(x,y) \text{ in } \Omega \quad (1)$$

where the domain  $\Omega$  is a bounded, connected, region in  $R^2$ . The boundary conditions can be Dirichlet on part of the boundary,

$$u = g(x,y) \text{ on } \partial\Omega_D \quad (2)$$

and natural or mixed on the remainder of the boundary,

$$p(x,y)\frac{\partial u}{\partial x}\frac{\partial y}{\partial s} - q(x,y)\frac{\partial u}{\partial y}\frac{\partial x}{\partial s} + c(x,y)u = g(x,y) \text{ on } \partial\Omega_N \quad (3)$$

where the boundary  $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$  and  $\partial\Omega_D \cap \partial\Omega_N = \emptyset$ . Periodic boundary conditions are also supported.

For natural boundary conditions, differentiation with respect to  $s$  is with respect to a counterclockwise parameterization of the boundary  $(x(s), y(s))$  with  $\|(dx/ds, dy/ds)\| = 1$ . Note that when  $p = q = 1$  or the boundaries of the domain are parallel to the axes, the natural boundary conditions reduce to Neumann conditions.

If the domain has curved boundaries, it is defined by subroutines that define the boundary parametrically. If it is polygonal, it can instead be defined by an initial triangulation given in data files created by the program Triangle.

PHAML also solves elliptic eigenvalue problems where the right hand side  $f(x,y)$  is  $\lambda u$ , and  $\lambda$  is an eigenvalue to be determined along with  $u$ . For eigenvalue problems, the boundary conditions must be homogeneous, i.e.,  $g = 0$ .

PHAML discretizes the PDE using the standard finite element method with piecewise polynomial functions over triangles. The approximating polynomial

degrees may be a fixed constant or adaptive. The grid is generated by beginning with a very coarse grid of fixed degree and using uniform or adaptive h-refinement in the form of newest node bisection, uniform or adaptive p-refinement in which the polynomial degree is increased, or hp-adaptive refinement which combines both forms of refinement.

The standard process alternates between phases of grid adaptation and solution of the discrete equations. The default solution method is a hierarchical basis multigrid method. Other solution methods are available through other optional software packages. All this is performed in parallel using the full domain partition to minimize communication. The default load balancing method is a refinement-tree based partitioning algorithm. Other load balancing methods are available through other optional software packages.

PHAML can be run as a sequential program, a master/slave parallel program using MPI-2 or PVM, or an SPMD parallel program using MPI-1, MPI-2, or PVM. PHAML optionally uses OpenGL to provide menu-driven interactive visualization.

## 2 Software

### 2.1 Obtaining Software

PHAML and all required or optional auxiliary software can be obtained by freely downloading them from the web, although some have commercial alternatives. This section explains what software is used with PHAML, when you need that software, and where to obtain it. Most of this software is optional and you do not need to install it unless you are using the relevant features as described below.

#### 2.1.1 PHAML

PHAML is available for download at <http://math.nist.gov/phaml>. It is a gzipped tar file. When unpacked, it creates a directory called `phaml-x.x.x` where the x's are the version number. It has been tested under many Unix systems, but not under MS Windows. It is standard conforming Fortran 90, so (theoretically) it should work under Windows (at least sequentially) without too much effort. For the remainder of this document, we will assume that the operating system is some variant of Unix, including Linux.

#### 2.1.2 Fortran 90 and C Compilers

A Fortran 90 (or later Fortran standard) compiler is required to compile PHAML. In some cases, a C compiler will also be required to compile some wrapper routines to C libraries. Most Unix systems already have a C compiler installed, and many also have a Fortran 90 compiler installed. Check your local documentation to find out, or look for a command like `f90` or `f95`.

If you do not have a Fortran 90 compiler, many good commercial compilers are available for most (if not all) Unix systems. To identify what is available for your system, see Michael Metcalf's *Fortran 90/95/HPF Information File* at <http://www.fortran.com/fortran/metcalf.htm>.

There are currently two free Fortran 90 compilers, g95 at <http://www.g95.org> and GNU's gfortran at <http://gcc.gnu.org/fortran>.

### 2.1.3 BLAS and LAPACK

BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKAGE) libraries are required for PHAML. The reference Fortran implementation for them can be obtained from Netlib at <http://www.netlib.org>. However, for best performance you should use a BLAS library that is optimized for your computer system. You may find that the BLAS and LAPACK libraries are already installed on your system, or that there are commercial optimized versions available for your system.

There are also freely available optimized BLAS packages such as ATLAS at <http://math-atlas.sourceforge.net> and GOTO BLAS at <http://www.tacc.utexas.edu/resources/software/>.

### 2.1.4 PVM and MPI

To run PHAML as a parallel program, you must have a message passing library (PHAML can be run as a sequential program, i.e. one processor, without a message passing library).

Message passing can be performed by either MPI (Message Passing Interface) or PVM (Parallel Virtual Machine). MPI is considered to be more of an industry standard, and is the recommended choice in most cases. PVM support in PHAML is maintained primarily for historical reasons, but PVM does have some features (for example, specifying the host on which to run the graphics process) that are not necessarily supported by an MPI implementation. Many of the other optional packages (e.g. PETSc, hypre, Zoltan, MUMPS, ARPACK) require MPI and cannot be used if PVM is chosen instead.

Many computer systems will already have an MPI library installed. Check your local documentation. If not, there are three freely available implementations of MPI. The Open MPI implementation is available at <http://www.open-mpi.org>. The LAM implementation is available at <http://www.lam-mpi.org>. The MPICH2 implementation is available at <http://www-unix.mcs.anl.gov/mpi/mpich2>.

PVM is available at [http://www.csm.ornl.gov/pvm/pvm\\_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html).

### 2.1.5 OpenGL (or Mesa), GLUT and f90gl

If you wish to use the visualization capabilities of PHAML, then you will need an OpenGL library (graphics library), GLUT (window, mouse, keyboard and menu management), and f90gl (the Fortran 90 interface to OpenGL and GLUT).



Some computers will already have an OpenGL library installed. If not, there are some commercial OpenGL libraries for some computer systems, and there is the freely available Mesa library. Mesa can be obtained from <http://www.mesa3d.org>, and further information about OpenGL can be found at <http://www.opengl.org>.

If your system already has an OpenGL library, then it is likely to also have GLUT already installed. However, many versions of GLUT are not compatible with f90gl, so you may need to reinstall GLUT anyway. If you (re)install GLUT, you should get it from the software section of the f90gl web pages at <http://math.nist.gov/f90gl>.

It is highly unlikely that f90gl will already be installed on your system. You can obtain f90gl from <http://math.nist.gov/f90gl>.

### **2.1.6 Triangle**

PHAML uses Jonathan Richard Shewchuk's mesh generator, Triangle [22], to generate the initial grid for arbitrary geometry. If you are only solving problems on the unit square, then you do not need Triangle – you can use the Triangle output files in the examples. You also do not need it if you have obtained Triangle data files elsewhere. Otherwise, you will need it. You also need it if you choose to define your domain through subroutines. Triangle is available at <http://www.cs.cmu.edu/~quake/triangle.html>.

### **2.1.7 ARPACK**

To solve elliptic eigenvalue problems, PHAML needs an algebraic eigensolver. Currently it only supports one eigensolver, ARPACK [12]. If you are going to solve eigenvalue problems, you must get ARPACK and PARPACK from <http://www.caam.rice.edu/software/ARPACK>. For a sequential compilation you do not need PARPACK.

### **2.1.8 hypre**

hypre [10][11] is a package of iterative linear system solvers and preconditioners from the Lawrence Livermore National Laboratory. It is not required, but can be linked with PHAML to provide alternative linear system solvers. It is available at [http://www.llnl.gov/CASC/linear\\_solvers](http://www.llnl.gov/CASC/linear_solvers).

### **2.1.9 MUMPS**

MUMPS [2][3] is a parallel direct linear system solver. It is not required, but can be linked with PHAML to provide an alternative linear system solver. It is available at <http://www.enseeiht.fr/lima/apo/MUMPS>.

### 2.1.10 PETSc

PETSc [5][6] is a package of iterative linear system solvers and preconditioners from Argonne National Laboratory. It is not required, but can be linked with PHAML to provide alternative linear system solvers. It is available at <http://www-unix.mcs.anl.gov/petsc/petsc-as>.

### 2.1.11 SuperLU

SuperLU [13] is a parallel direct linear system solver from the Lawrence Berkeley National Laboratory. It is not required, but can be linked with PHAML to provide an alternative linear system solver. It is available at <http://crd.lbl.gov/~xiaoye/SuperLU>.

### 2.1.12 Zoltan

Zoltan [8][9] is a library of dynamic load balancing methods. It is not required, but can be linked with PHAML to provide alternative methods for partitioning the grid for distribution over parallel processors. It can be obtained at <http://www.cs.sandia.gov/Zoltan>.

## 2.2 Compiling PHAML

This section gives instructions for compiling the PHAML library. Throughout this section it is assumed that the current working directory is the PHAML root directory.

### 2.2.1 Creating the Makefiles

Before compiling you must create a **Makefile**. This is done by running the shell script `mkmkfile.sh`. Before running this script you must edit it to set system dependent values, like the paths to certain libraries, and specify the configuration you wish to build a **Makefile** for (what kind of computer, what message passing library, etc.). Instructions for modifying `mkmkfile.sh` are contained in the file, but are also summarized here. There are three steps to modifying `mkmkfile.sh`: 1) set the default system configuration, 2) set the paths, library names, etc., and 3) set values for particular systems.

In step 1, the default system configuration is set. If you will only be running in one environment, then you can set the configuration here and forget it. In fact, you can probably just make the **Makefile** once and forget it. If you will be using more than one environment, then you can set the defaults here, and they can be overridden by one of the methods described later in this section.

The configuration variables (e.g. `PHAML_ARCH`) and legitimate values (e.g. `origin`) can be found under Step 1 in `mkmkfile.sh`. You can also get a list of all the configuration variables, their legitimate values, and current defaults by executing

```
./mkmkfile.sh help
```

They are:

- `PHAML_ARCH` `origin rs6k sgi sun tflop x86`. This defines the type of computer architecture you have.
- `PHAML_OS` `aix cougar irixn32 irix64 linux solaris`. This defines the operating system running on your computer.
- `PHAML_F90` `absoft g95 gfortran intel lahey nag pathscale pgi sgi sun xlf`. This defines the Fortran 90 compiler to use.
- `PHAML_C` `cc gcc`. This defines the C compiler to use.
- `PHAML_HASHSIZE` `1 2`. This defines the size of hash key to use for global identifiers. `1` uses one integer and `2` uses two integers. Using `1` saves memory and reduces message sizes; using `2` allows smaller elements (more refinement levels).
- `PHAML_PARALLEL` `messpass_spawn messpass_nospawn sequential`. This defines the type of parallelism. You can select between running a sequential program (no parallelism), having a master process that spawns the slave and graphics processes, and running an SPMD (single program multiple data) program in which all processes are identical and started from the command line.
- `PHAML_PARLIB` `lam mpi mpich mpich2 myrinet pvm none`. This defines the parallel library to use. `lam`, `mpich`, `mpich2` and `pvm` refer to the packages by those names, `mpi` to vendor implementations of MPI, `myrinet` to a special implementation of MPICH for myrinet networks, and `none` should be used if `PHAML_PARALLEL` is `sequential`.
- `PHAML_GRAPHICS` `metro mesa none opengl`. This defines what OpenGL library should be used. `opengl` refers to vendor OpenGL libraries, `mesa` to the MESA library, `metro` to a commercial OpenGL library for Linux, and `none` indicates that the graphics program should not be built.
- `PHAML_BLAS` `atlas compiler goto source standard vendor`. This defines the BLAS library to use. `atlas` and `goto` are as mentioned in Section 2.1.3. `compiler` refers to a BLAS library provided with the Fortran compiler defined in `PHAML_F90`. `vendor` refers to a BLAS library supplied by the hardware vendor defined in `PHAML_ARCH`. `standard` refers to a BLAS library in a standard location like `/usr/lib`, and often is not a high performance library. As a last resort, `source` will use BLAS source code provided with PHAML.
- `PHAML_LAPACK` `atlas compiler source standard vendor`. This defines the LAPACK library to use, and is usually given the same value as `PHAML_BLAS`.
- `PHAML_ARPACK` `no yes`. This indicates whether or not to include ARPACK.

- PHAML\_HYPRE `no yes 1.6.0 1.9.0b`. This indicates whether or not to include hypre. There were some incompatible changes between hypre versions 1.6.0 and 1.9.0b, so if you have an older version you should specify 1.6.0.
- PHAML\_MUMPS `no yes`. This indicates whether or not to include MUMPS.
- PHAML\_PETSC `no yes`. This indicates whether or not to include PETSc.
- PHAML\_SUPERLU `no yes`. This indicates whether or not to include SuperLU.
- PHAML\_ZOLTAN `no yes with_jostle with_parmetis with_jostle_and_parmetis`. This indicates whether or not to use Zoltan. If Zoltan was built with support for ParMETIS and/or Jostle, use the value that indicates that.
- PHAML\_SYSTEM `none dragon raritan looney looneyjr sgis suns tflop`. This designates a specific computer system for overriding configuration variables in Step 3. Usually it is the hostname of the system.

If you don't find a value that matches your system, you can either (1) add that value and modify `mkmkfile.sh` to handle it by mimicking what is done for other values, or (2) use a similar value and edit `Makefile` to correct it for your system. However, if you edit `Makefile` then you will lose your changes the next time you run `mkmkfile.sh`.

In step 2, set the paths, library names, etc. Here you set shell variables in `mkmkfile.sh` with the names of commands, flags, paths, library names, etc., for the configurations you will be using. Many of these will already be correct, but examine them because some of them are specific to the systems on which PHAML was developed. In particular, many of the compiler command names have been changed to avoid conflicts between multiple compilers on one system, and nearly all of the paths will vary between systems.

In step 3, you can override values set in step 2 for particular system configurations. For example, suppose your MPICH library has a different name on Linux than it has on all your other systems. Then in step 2 you set the name used on most systems, and in step 3 you override it if `PHAML_SYSTEM` is your linux box.

There are two other ways to override the default configuration variable values: 1) on the command line for `mkmkfile.sh` (described below), and 2) by setting environment variables in your shell. If an environment variable exists with the same name as a configuration variable, the value of that environment variable is used. Usually these variables are set in the shell startup file. For csh-type shell, an example is

```
setenv PHAML_OS linux
```

Once you finish modifying `mkmkfile.sh`, running it will create `Makefile` in the `src` directory and all the subdirectories under `example` and `testdir`. If

a file named `Makefile` already exists, it is moved to `Makefile.bak` before the new file is created. To accept the default configuration, just run

```
./mkmkfile.sh
```

To override the default values and values given by environment variables, specify them on the command line by giving a space-separated list of configuration variables and values. Here, including the “PHAML\_” part of the configuration variable is optional. For example,

```
./mkmkfile.sh PARALLEL messpass_spawn PARLIB lam
```

## 2.2.2 Compiling the Library

Once `Makefile` has been built, just type `make` in either the PHAML root directory or `src` subdirectory to compile the PHAML library. If this shows that `Makefile` has errors in it, you can either edit `Makefile` to fix them (but then they will appear again if you need to run `mkmkfile.sh` again), or fix the errors in `mkmkfile.sh` and rebuild `Makefile`. This will create the PHAML library `libphaml.a` and copy it to subdirectory `lib`. It also copies any module files (e.g., `*.mod`) to subdirectory `modules`, and creates a file `lib/CONFIG` that contains the values of the configuration variables used to create the library.

## 2.3 Testing the Library

After creating the PHAML library, you can test it by running the PHAML Test Suite. The suite contains over 100 short test programs to test nearly all aspects and options of PHAML. Tests that do not apply to the current compilation of the library are skipped. The tests have not yet been written for `PHAML_PARALLEL=messpass_nospawn` (i.e. SPMD programs). They can only be run with `sequential` or `messpass_spawn` (i.e. master/slave) programs.

To run the full test suite, enter the command

```
make test
```

from either the PHAML root directory or the `testdir` subdirectory. To delete all the files that were created by the tests, enter the command

```
make test what=clean
```

The tests can be grouped into three categories. The first tests consists of two very simple programs to verify that a program can be compiled, linked and run. The interactive tests test the use of the `pause` variables to `phaml_solve_pde`, spawning in a debug window, and graphics. The remaining tests are the non-interactive tests. Depending on how many optional auxiliary packages are included, the speed of your computer system, etc., these may take anywhere from a few minutes to a few hours to run.

You can run just the first tests, just the first and interactive tests, or just the first and noninteractive tests with the commands

```
make test what=first
```

```
make test what=interactive
```

```
make test what=noninteractive
```

If you are running the tests from a batch system where you cannot respond to interactive prompts, you should use `what=noninteractive`.

The individual interactive tests can be run by giving `what` the value `pause`, `debug`, or `graphics`. The noninteractive tests are in the directories `testdir/test_*`. You can run individual noninteractive tests by giving `what` the name of the subdirectory, with or without `test_`. For example, to run the ARPACK tests you can use either

```
make test what=test_arpack, or
make test what=arpack
```

The results of the tests are printed both to the screen and to the file `testdir/testresults`. The file only contains standard output, not standard error, so things like messages from the compiler will not appear in the file. Each test consists of a short program that is compiled and run with the output directed to a file. The file is compared to an existing file that contains the expected output. Each test is identified as SUCCESS, FAILURE or WARNING. WARNING indicates that the output is not the same as the comparison file. Often this difference is just minor numerical differences (accumulated round off error) due to different processors, compilers, versions of auxiliary packages, etc. The warning message will direct you to a file that contains the differences, as identified by `diff`, which you should examine to see if the differences are significant.

## 2.4 Compiling and Running the Examples

Next you compile your application and link in the PHAML library. With most compilers you will need to specify the location of the module files when compiling program units that use `phaml`. Many compilers use the `-I` flag for this (e.g. `f90 -I $PHAML_HOME/modules myprog.f90`), but you should check your compiler's documentation. You also need to link with the PHAML library (e.g. `f90 myprog.f90 -L$PHAML_HOME/lib -lphaml`) and any other libraries your program needs (MPI, OpenGL, LAPACK, etc.). Your best start is to look at the examples in the `examples` directory, and their `Makefiles` (which were created by `mknkfile.sh`).

With the examples, a successful `make` will create up to three executables:

1. the primary executable `phaml`. This is the program you run.
2. `phaml_slave` if `PHAML_PARALLEL` is `messpass_spawn`. This is spawned by `phaml`.
3. `phaml_graphics` if `PHAML_GRAPHICS` is not `none` and `PARALLEL` is not `messpass_nospawn`. This is spawned by `phaml` and `phaml_slave` if `PARALLEL` is `messpass_spawn`, or run from the command line if `PARALLEL` is `sequential`.

In the PHAML root directory, `make clean` will remove any files created by compilation (e.g. `*.o`) in `src`, `examples/*`, and `testdir/*`. `make reallyclean` will also remove everything in `lib` and `modules`, and the `Makefiles`.

In general, you run `phaml` as you would any parallel program that uses the message passing library you have selected. Some examples are:

1. A sequential compilation. Run the primary program.

```
phaml
```

If you requested graphics, you must also run the graphics program.

```
phaml_graphics
```

2. PVM, as a master/slave program. Start the PVM demon on the nodes of the virtual machine. Then just run the master program, which will spawn the slaves and graphics.

```
phaml
```

3. LAM MPI, with spawning. You only start the master program, which spawns the slaves and graphics, so specify the number of processors to be 1.

```
mpirun -w -np 1 phaml
```

I have also found that in this case you don't need to use `mpirun`, you can just execute the master.

```
phaml
```

4. MPICH MPI, as an SPMD program. Since this does not spawn the slaves and graphics, you must use `mpirun` to specify the right number of processes. If there are to be  $n$  slaves, this consists of the sum of the following: One for the master,  $n$  for the slaves, one if the master is doing graphics, and  $n$  if the slaves are doing graphics. For example, if there are four slaves and the master is doing graphics,

```
mpirun -machinefile <file with list of hosts> -np 6 phaml
```

5. See also the file `doc/HINTS` for my notes on running under debuggers, using `ssh`, redirecting displays, etc.

For parallel code with PVM, you will need to have `phaml`, `phaml_slave` and `phaml_graphics` (if you are using graphics) in `$HOME/pvm3/bin/$PVM_ARCH`. I just keep a symbolic link in there for each one, which points to the executable in my working directory.

If you request graphics, a window should pop up with the graphical display. The following actions are defined by default:

- arrow keys - pan
- left mouse button - rotate
- middle mouse button - zoom
- right mouse button - a menu of actions

The menu contains a large number of ways to change the graphics. See Section 3.5 for further details.

model	PHAML_PARALLEL
sequential	sequential
master/slave	messpass_spawn
SPMD	messpass_nospawn

Table 1: Correspondence between PHAML\_PARALLEL in `mkmkfile.sh` and the parallel programming model.

### 3 Scalar Linear Elliptic Boundary Value Problems

The primary function of PHAML is to solve second-order scalar linear self-adjoint elliptic partial differential equations of the form given in Equation 1 with Dirichlet, natural (often Neumann), or mixed boundary conditions given by Equations 2 and 3. This section explains how to write a program that uses PHAML to solve these problems. Other classes of problems that can be solved using PHAML are addressed in subsequent sections. The information here is organized by concept. For an organization by subroutine and parameters, see the reference guide in Section 7.

#### 3.1 Main program

PHAML is a library of subroutines. The user must write a program that calls these subroutines to solve the application problem. Ordinarily this just requires a simple main program and subroutines that define the equations, but the program can be as complex as needed. Subroutines in the PHAML library can be called from either the main program or subroutines.

##### 3.1.1 Parallelism

There are three models of parallel processing supported by PHAML. The programming model used must correspond to the one specified by PHAML\_PARALLEL in `mkmkfile.sh` when the PHAML library was built (see Section 2.2.1). The correspondence is given in Table 1.

In the master/slave model, the parallelism is almost transparent to the user. The user only needs to specify `nproc` in `phaml_create`, and start the program in the manner specified by the message passing library, requesting 1 process. With some libraries, for example LAM and MPICH2, even this can be transparent, since you can just run the executable when you are only starting 1 process, unless your system uses a queuing program like PBS. Sequential programs can use the same program as a master/slave program, and `nproc` will be ignored. Single Program Multiple Data (SPMD) programs require a little more knowledge of the parallelism. Each instance of the program must determine if it is to be the master, a slave, or a graphics server, and act accordingly. This is explained



```

program simple
use phaml
type(phaml_solution_type) :: sol
call phaml_create(sol,nproc=2)
call phaml_solve_pde(sol,
                    max_vert=100000,
                    print_grid_when=PHASES,
                    print_grid_who=MASTER,
                    print_error_when=PHASES,
                    print_error_what=LINF_ERR,
                    print_error_who=MASTER)
call phaml_destroy(sol)
end program simple

```

Figure 1: A simple main program.

in the next section. Since SPMD PHAML programs behave as if they were a master/slave program, with process 0 acting as the master, this document will often make reference to the master and slaves, even though the program might be SPMD.

### 3.1.2 Program structure

#### 3.1.2.1 Master/slave and Sequential

In a master/slave program, the user writes a program for the master process. The slave and graphics programs are given in the `src` directory as `slave.f90` and `graphmain.f90`. To compile these programs, see the `Makefiles` in the `examples` directory.

Figure 1 illustrates a very simple main program for a master/slave model. More complicated examples can be found in the `examples` directory. This example illustrates the main steps in writing a PHAML program.

First it must use module `phaml`. This gives access to all the public subroutines, the defined constants that are used as values for subroutine arguments, the `phaml_solution_type` data structure, and the defined constant `my_real` which contains the kind number for reals in PHAML.

Second, there must be at least one variable of type `phaml_solution_type`. Variables of this type contain all the information known about the solution of the equation, including the current grid, solution, parallel processes, etc. The type is a public entity in module `phaml`, but the internals of the type are private. This means you cannot directly manipulate components of the variable, you can only pass it to the PHAML subroutines that operate on it.

Next, the solution variable is initialized by subroutine `phaml_create`. This creates an initial grid, allocates and initializes other components of the variable, and, for master/slave programs, spawns the slave and graphics processes. Here

we requested the use of 2 slave processors. Other arguments will be discussed in subsequent sections as they become relevant. A full list can be found in Section 7.4.4.

The PDE is solved by calling `phaml_solve_pde`. This subroutine requires that the first argument be a solution variable. There are over 100 more arguments to this subroutine, which will be discussed in subsequent sections and can be found in Section 7.4.14. To make this manageable, all arguments are optional, have reasonable defaults, and should be given as keyword arguments, i.e. *dummyarg=value*, as shown in the example.

Finally, the solution variable should be destroyed by `phaml_destroy` to free memory and terminate any processes that were spawned by `phaml_create`.

### 3.1.2.2 SPMD

An SPMD program is more complicated because the master, slave and graphics processes are identical. However, it still operates like a master/slave program, so each process must determine if it is the master, a slave, or a graphics server. This is illustrated in Figure 2 for a program that uses an MPI library.

In addition to using `module phaml`, it must use `module mpif_mod` to get access to the MPI include file.

The first step is to initialize MPI, determine how many processes there are, and determine the rank of this process. This is needed to determine what type of process this will be, so it cannot be hidden inside `phaml_create` like it is for master/slave programs.

Then the number of slaves is computed based on the total number of processes (which is specified when the program is launched, for example by `mpirun`) and how many graphics processes are needed.

Each process determines its type based on its rank (`my_processor`) and the number of slaves, and calls the corresponding main subroutine. The slave and graphics main subroutines are in the PHAML library. The main subroutine for the master is nearly identical to the main program in the master/slave model, as shown in Figure 2.

## 3.2 Defining the problem

### 3.2.1 Defining the PDE

The PDE in Equation 1 must be defined in subroutine `pdecoefs`. For convenience, the equation is repeated here.

$$-\frac{\partial}{\partial x}(p(x,y)\frac{\partial u}{\partial x}) - \frac{\partial}{\partial y}(q(x,y)\frac{\partial u}{\partial y}) + r(x,y)u = f(x,y)$$

`pdecoefs` receives a point  $(x, y)$  and returns the value of the functions  $p$ ,  $q$ ,  $r$  and  $f$  at that point in the arguments `cxx`, `cyy`, `c` and `rs` respectively. The interface for this routine is given in Section 7.3.6. For a scalar PDE, the return variables are  $1 \times 1$  arrays (they have higher dimension for systems of equations; see Section 4.2) and can be assigned with a whole array assignment statement,

```

program simple_spmd
use phaml
use mpif_mod
integer :: whodrawg
integer :: jerr
integer :: my_processor, total_nproc
integer :: nslave, subtract, divide

! initialize MPI, find out how many processors and what my rank is

call mpi_init(jerr)
call mpi_comm_size(MPI_COMM_WORLD,total_nproc,jerr)
call mpi_comm_rank(MPI_COMM_WORLD,my_processor,jerr)

! set the graphics options

whodrawg = NO_ONE

! determine how many processors for slaves and graphics

subtract = 1
if (whodrawg == MASTER .or. whodrawg == EVERYONE) subtract = 2
divide = 1
if (whodrawg == SLAVES .or. whodrawg == EVERYONE) divide = 2
nslave = (total_nproc-subtract)/divide

! call the master, slave or graphics program depending on my rank

if (my_processor == 0) then
  call phaml_master(whodrawg,nslave)
elseif (my_processor <= nslave) then
  call phaml_slave
else
  call phaml_graphics
endif
end program simple_spmd

subroutine phaml_master(whodrawg,nslave)
use phaml
integer, intent(in) :: whodrawg, nslave
type(phaml_solution_type) :: sol
call phaml_create(sol,nproc=nslave,draw_grid_who=whodrawg)
call phaml_solve_pde(sol, ... )
call phaml_destroy(sol)
end subroutine phaml_master

```

Figure 2: An SPMD main program using MPI.

```

subroutine pdecoefs(x,y,cxx,cxy,cyy,cx,cy,c,rs)
use phaml
real(my_real), intent(in) :: x,y
real(my_real), intent(out), dimension(:,:) :: cxx,cxy,cyy,cx,cy,c
real(my_real), intent(out), dimension(:) :: rs
cxx=1
cxy=0
cyy=1
cx=0
cy=0
c=0
rs=0
end subroutine pdecoefs

```

Figure 3: `pdecoefs` for Laplace’s equation.

making their “arrayness” transparent. Figure 3 shows how `pdecoefs` could be written for Laplace’s equation  $\nabla^2 u = 0$ .

Subroutine `pdecoefs` also takes three more return arguments: `cxy`, `cx` and `cy`. These are not currently used. They are included for a possible future extension of the class of problems that PHAML can solve. Strictly speaking they do not need to be set, but it would be wise to set them to 0 to avoid possible problems in the future if PHAML does start using them.

### 3.2.2 Defining the boundary conditions

The boundary conditions are defined in subroutine `bconds`. The interface for this routine is given in Section 7.3.1. On each piece of the boundary (Section 3.2.3 explains how to define the boundary piecewise and send a piece ID number to `bconds`) the boundary can be Dirichlet as defined by Equation 2

$$u = g(x, y),$$

natural (usually Neumann) or mixed as defined by Equation 3

$$p(x, y) \frac{\partial u}{\partial x} \frac{\partial y}{\partial s} - q(x, y) \frac{\partial u}{\partial y} \frac{\partial x}{\partial s} + c(x, y)u = g(x, y),$$

or periodic (usually on opposite sides of a rectangle).

`bconds` receives as input a point  $(x, y)$  at which to return the boundary conditions, and `bmark`, an integer ID number for the piece of the boundary that  $(x, y)$  is on. This ID is defined along with the boundary of the domain (Section 3.2.3).

The type of boundary condition for the indicated piece is returned in `itype` using a symbolic constant listed in Table 2. The functions  $p$  and  $q$  in Equation 3 are the same as those in the PDE, Equation 1, which are defined in subroutine

```

subroutine bconds(x,y,bmark,itype,c,rs)
use phaml
real(my_real), intent(in) :: x,y
integer, intent(in) :: bmark
integer, intent(out) :: itype
real(my_real), intent(out) :: c(:,:),rs(:)
if (bmark == 1) then
    itype = NATURAL
else
    itype = DIRICHLET
endif
c=0
rs=0
end subroutine bconds

```

Figure 4: `bconds` for homogeneous boundary conditions, natural on boundary piece 1 and Dirichlet elsewhere.

boundary condition	itype
Eq. 2	DIRICHLET
Eq. 3 with $c = 0$	NATURAL
Eq. 3 with $c \neq 0$	MIXED
periodic	PERIODIC

Table 2: Permitted values for `itype` to specify the type of boundary conditions.

`pdecoefs` (Section 3.2.1) and not repeated here. The function  $c$  in Equation 3 is returned in the variable `c`. Finally, the function  $g$  in Equations 2 and 3 is returned in `rs`.

`c` and `rs` are arrays, but for scalar problems the dimensions are all 1, and they can be assigned with a whole array assignment statement. (They have higher dimension for systems of equations; see Section 4.2.)

Assuming the domain is the unit square with the left side assigned 1 for the ID, Figure 4 shows a subroutine for the boundary conditions

$$\partial u / \partial n = 0 \quad \text{on } x = 0 \quad (4)$$

$$u = 0 \quad \text{elsewhere} \quad (5)$$

Periodic boundary conditions say that the value of the solution on one piece of the boundary is the same as the value on another piece. Usually the two pieces are opposite sides of a rectangle, but for generality we need a means of indicating which two pieces are matched up. This is done by making the piece ID of the two pieces have the same absolute value, with one positive and the other negative. `c` and `rs` are not used with periodic boundary conditions, but it is prudent to set them to 0. For an example of periodic boundary conditions, see `examples/periodic`.

Natural and periodic boundary conditions require that there be at least one point with Dirichlet boundary conditions to make the solution unique.

### 3.2.3 Defining the domain and initial grid

The domain and initial grid can be defined in one of two ways. The first is to supply subroutines that define the boundary explicitly as a piecewise curve in  $R^2$ . This approach is required if the boundary is not polygonal, i.e. if it has curved pieces, but can also be used for polygonal domains. The second is to provide triangle data files that were produced by the program Triangle [22]. These files can be created by writing a `.poly` file and running Triangle, or by using the first approach once and saving the triangle data files that PHAML creates, or by using the triangle data files from one of the PHAML examples. In either case, this gives a starting triangulation, which is preprocessed to create the initial grid.

To define the domain with subroutines, you must write three subroutines similar to those used by the domain processor of Rice [21].

`subroutine boundary_point(ipiece,s,x,y)` contains the definition of the boundary. Each piece of the boundary is given by a parametric curve  $(x(s), y(s))$  for  $s$  in some range of  $R$ . The ending point of one piece must be the same as the starting point of the next piece, including the last piece of the outer boundary connecting to the first piece of the outer boundary, and the last piece of each hole connecting to the first piece of the same hole. The input parameters `ipiece` and `s` indicate which piece of the boundary and parameter value to evaluate, and the output parameters `x` and `y` are the requested point on the boundary.

`ipiece` is also used as the boundary marker for the triangle data files, and will be passed to `bcond` (Section 7.3.1) to simplify evaluation of boundary conditions. The endpoints of the boundary pieces will always be vertices of the grid, and are assigned the boundary marker of the piece they start. To get a boundary marker that is unique to an individual vertex, define a boundary piece of length 0 at that vertex (i.e., starting and ending parameters are the same). The last boundary piece cannot be a single point. Instead, make that point be the first piece.

`function boundary_npiece(hole)` (Section 7.3.3) returns the number of pieces used to define the boundary. If `hole` is 0, it should return the number of pieces that define the outer boundary. Otherwise it should return the number of pieces that define the `holeth` hole. The holes are numbered consecutively starting with 1. `boundary_npiece(0)` must return a positive integer if and only if subroutines are used to define the boundary. If it returns 0 or a negative integer, the starting triangulation is read from triangle data files. If `hole` is larger than the number of holes in the domain, it should return 0.

`subroutine boundary_param(start,finish)` returns the range of parameter values for every piece of the boundary. The output variables `start` and `finish` are `real(my_real)` arrays of length equal to the total number of boundary pieces, and should be assigned the starting and ending parameters of each piece.

The starting grid is created by PHAML by creating a `.poly` file for input to Triangle, which is then run to create triangle data files. Running Triangle involves a call to the intrinsic subroutine `system`, which is not standard in Fortran 90. Most compilers provide this routine as an extension, and use the same behavior for it, but if you have trouble you may have to change the definition of `my_system` in `sysdep.f90`.

You can control the fineness of the starting grid with the parameter `max_blen` in `phaml_create`. No triangle boundary side in the starting triangulation will have length longer than `max_blen`, and Triangle is run with `-a max_blen2√3/4` to limit the area of the triangles to that of an equilateral triangle with side length `max_blen`.

The name of the `.poly` file is determined by the parameter `triangle_files` in `phaml_create`, which has the default value `"domain"`. The file name will be the character string with the added suffix `.poly`. Note that if the full path is not given in `triangle_files`, the location of the file may be compiler dependent.

The second method for defining the domain and initial grid is to use triangle data files. These are the edge (`.edge`), element (`.ele`), neighbor (`.neigh`), node (`.node`) and poly (`.poly`) files created by the program Triangle.

To indicate that the starting grid is given by triangle data files, have `function boundary_npiece(0)` return 0 or a negative number. The parameter `triangle_files` in `phaml_create`, which has the default value `"domain"`, then contains the root of the name of the five triangle data files. If Triangle inserted an iteration number (e.g. `.1`) into the name, then it should be included.

If you are solving a PDE on the unit square, or on one of the other domains in the examples, then you only need to copy the triangle data files from the

example and, if necessary, specify the name of the files.

Otherwise you must define the polygonal domain with a `.poly` file, which is basically a list of vertices and boundary line segments. Holes are allowed. See the documentation for Triangle [23] for the exact definition of a `.poly` file. You should specify a boundary marker for each vertex and boundary segment. This boundary marker will be passed to function `bcond` to simplify the definition of boundary conditions. With a `.poly` file defined, create the triangle data files with the command

```
triangle -pnej root_of_filename.poly
```

The `-p` flag indicates the input is a poly file. The `-n`, and `-e` flags force the writing of neighbor and edge files. The `-q` flag requests a quality mesh, which imposes bounds on the angles. The `-j` flag says to jettison (remove) vertices that are not part of the triangulation. Also, the `-a` flag may be useful to impose a maximum size (area) of the triangles, and the `-I` flag can be used to prevent the insertion of the iteration number in the filename. See the documentation for Triangle [23].

Regardless of whether they are created by Triangle from a `.poly` file, created by PHAML from the boundary subroutines, or copied from somewhere else, the triangle data files define a starting triangulation. PHAML requires an initial grid that satisfies certain conditions. This initial grid is determined from the starting triangulation automatically by PHAML through a process of refinement and grid smoothing as described in [20]. The triangles of the initial grid are obtained by bisection or trisection of the starting triangles, followed by some node movement.

### 3.2.4 The true solution

If the true (a.k.a. exact or analytical) solution is known, it and its first derivatives can be provided through subroutines `true`s, `true`xs and `true`ys. If these are provided, then the program can print norms of the error (Section 3.4.2) or choose the error as the function to visualize with the graphics. If `true`s is defined, but `true`xs and/or `true`ys is not, then the energy norm of the error cannot be printed, but all other norms and the graphics are still valid.

The interfaces for these routines are given in Sections 7.3.8, 7.3.9 and 7.3.10. They are function subroutines that return the value of the solution,  $x$  derivative of the solution, and  $y$  derivative of the solution, respectively. The input variables give the point  $(x, y)$  at which to return the solution. There are also two input variables, `comp` and `eigen`, that are not relevant for scalar boundary value problems (they are used for systems of equations and eigenvalue problems).

## 3.3 Solution method

There are many arguments to `phaml_solve_pde` that affect the details of the solution method. This section explains the options available. It is organized with sections on discretization, refinement, the error indicator, the linear system solver, load balancing and termination. Each section briefly describes the



methods used, but this is not intended to be a thorough treatment of parallel adaptive multilevel finite element methods. See the references for more details of the methods.

See also Section 7.4.14 which formally defines the arguments to `phaml_solve_pde`.

### 3.3.1 Discretization

PHAML uses a standard Galerkin finite element method (see, for example [24][25]) to approximate the solution of Equations 1-3, which we briefly describe here. The domain is partitioned into a set of conforming triangles,  $T$ , (the grid or mesh) and the approximation space is defined as the Hilbert space of continuous functions that are polynomials over each triangle. The degree of the polynomial can be different over different triangles, but PHAML restricts the polynomial degrees to differ by at most 1 between two triangles that share an edge. The  $p$ -hierarchical basis,  $\Phi = \{\phi_i\}$ , of either Szabo & Babuška [25] or Carnevali et al. [7] spans the space, and gives a representation for the approximate solution

$$u_T = \sum_{\phi_i \in \Phi} \alpha_i \phi_i \quad (6)$$

Basis functions can be categorized in three groups. The linear ( $p = 1$ ) basis functions are in one-to-one correspondence with the vertices of the grid. Higher order basis functions are associated with edges or faces (triangles). There are  $p - 1$  edge bases associated with an edge of degree  $p$ , and  $(p - 1)(p - 2)/2$  face bases associated with a face of degree  $p$ .

The space has the energy inner product defined by

$$\langle u, v \rangle = \int_{\Omega} p u_x v_x + q u_y v_y + r u v \quad (7)$$

where  $p$ ,  $q$  and  $r$  are from Equation 1, and the subordinate energy norm  $\|u\|^2 = \langle u, u \rangle$ . The approximate solution is the function in the space that minimizes the energy norm of the error, i.e. the error is orthogonal to the space under the energy inner product. Thus the approximate solution satisfies

$$\langle u_T, \phi_i \rangle = \langle f, \phi_i \rangle_2 \forall \phi_i \in \Phi \quad (8)$$

where  $\langle \cdot, \cdot \rangle_2$  is the  $L_2$  inner product  $\langle u, v \rangle_2 = \int_{\Omega} u v$ . Substituting Equation 6 into Equation 8 leads to the discretized form of the problem  $Ax = b$  with  $a_{ij} = \langle \phi_i, \phi_j \rangle$ ,  $b_i = \langle f, \phi_i \rangle_2$ , and  $x$  the vector of  $\alpha_i$ 's. Natural boundary conditions are imposed automatically by the energy inner product. Dirichlet boundary conditions are imposed by replacing the corresponding equations with equations that set the corresponding  $\alpha_i$ 's directly from the boundary condition.

There are only three arguments that affect how discretization is performed. First, if  $p$ -refinement is not performed (see Section 3.3.2), then you can select the (constant) degree of the polynomials in the approximation space with the

argument `degree`. Second, the integrals of Equation 7 are computed by numerical quadrature, with the order of the quadrature rule determined such that integrals are exact for polynomials of the degree  $2(p - 1)$  where  $p$  is the degree of the basis over each triangle. Sometimes this may not be accurate enough, in which case you can supply an increase of the order through `inc_quad_order`. Finally, in triangles that are owned by a different processor (see Section 3.3.5), the order of the quadrature rule is increased to the maximum supported order. If this were not done, then the integrals over those triangles would be less accurate than the same integrals on the processor that owns the triangle, because it will have refined the triangle into many smaller triangles. However, sometimes this doesn't matter, such as when solving Laplace's equation (all the integrals are exact) or when using one of the auxiliary solvers (see 3.3.4) which don't use the unowned triangles. You can avoid the extra computation of the higher order quadrature with `ignore_quad_err=.true`.

### 3.3.2 Refinement

One of the major phases in `phaml_solve_pde` is refinement of the grid to increase the size of the approximation space, which reduces the norm of the discretization error.  $h$ -refinement refers to subdividing triangles by newest node bisection (see [14][16]).  $p$ -refinement refers to increasing the polynomial degree over a triangle. In either case the refinement can be undone, referred to as derefinement. Derefinement can be switched off with the argument `derefine=.false`. Refinement can be done uniformly, i.e. refine all triangles in the grid, or adaptively, i.e. refine a selected subset of the triangles.

The type of refinement that is performed is determined by the argument `reftype`. This argument can have one of the values `H_UNIFORM`, `H_ADAPTIVE`, `P_UNIFORM`, `P_ADAPTIVE` or `HP_ADAPTIVE`. With `HP_ADAPTIVE` an element may be refined by either  $h$ - or  $p$ -refinement. This is still experimental in PHAML.

There are several ways to determine how much refinement to perform in one refinement phase. This is controlled by the argument `refterm`. The allowed values for this argument are given in Table 3.

“Double” means that the refinement should approximately double the number of vertices, elements or equations in the grid. The factor 2 was chosen because this corresponds to the increase that would occur with one uniform  $h$ -refinement of the grid. If a different factor is desired, it can be specified with the real valued argument `inc_factor`.

Strictly doubling the number of entities may lead to grids that are nonsymmetric, which may be undesirable if the solution is symmetric. This can be improved by using the values that contain `SMOOTH`. With these values, after the doubling is completed the refinement will continue by refining all elements with a similar error indicator (Section 3.3.3) to the last element refined.

“Keep” attempts to change the grid by derefining some elements and refining others while keeping the total number of entities approximately the same. This is useful, for example, with time dependent problems where the grid should track the movement of some feature of the solution.

DOUBLE_NVERT
DOUBLE_NVERT_SMOOTH
DOUBLE_NELEM
DOUBLE_NELEM_SMOOTH
DOUBLE_NEQ
DOUBLE_NEQ_SMOOTH
HALVE_ERREST
KEEP_NVERT
KEEP_NVERT_SMOOTH
KEEP_NELEM
KEEP_NELEM_SMOOTH
KEEP_NEQ
KEEP_NEQ_SMOOTH
ONE_REF

Table 3: Permitted values for `refterm` to determine how much refinement to do.

`HALVE_ERREST` refines the grid until the maximum error indicator has been reduced by half. This corresponds to the expected reduction of error by a uniform  $h$ -refinement with `degree=1` and a smooth solution.

`ONE_REF` is a scheme in which no element gets refined more than once in a refinement phase. The argument `reftol` can provide a tolerance for which elements should be refined. All elements with an error indicator larger than  $\text{reftol}/\sqrt{n}$  are refined, where  $n$  is the starting number of equations.

### 3.3.3 Error indicator

The heart of an adaptive refinement strategy is the error estimator, or more properly, error indicator. An error indicator is computed for each triangle, and those with the largest error indicators are refined. PHAML currently contains three error indicators. The error indicator is selected by the argument `error_estimator` which takes one of the values `LOCAL_PROBLEM`, `HIERARCHICAL_COEFFICIENT` or `TRUE_DIFF`.

`LOCAL_PROBLEM` is the primary error indicator in PHAML. It is based on solving a small local problem. (See [1] for a more detailed treatment of error indicators.) Equation 1 is modified by replacing  $f$  with the residual  $f - Lu_T$  where  $L$  is the differential operator. For a linear PDE, the solution of this equation is the error. The modified equation is solved on a domain consisting of one or two triangles to get an estimate of the error over that small domain. To approximate the solution of this small problem with slightly higher accuracy than the current solution  $u_T$ , the triangle(s) are refined once either by  $h$ -refinement or  $p$ -refinement. The  $h$ -refinement form uses two triangles and refines them as a pair by bisection. Homogeneous Dirichlet boundary conditions are applied. The  $p$ -refinement form uses one triangle with polynomial degree one larger than that

used for  $u_T$ . Natural boundary conditions are applied based on the jump in the normal derivative of  $u_T$  across the triangle edges. Currently PHAML uses the  $h$ -refinement form as the error indicator for  $h$ -refinement and the  $p$ -refinement form as the error indicator for  $p$ -refinement. Since the refinement used for the error indicator is exactly what would occur if the triangle(s) was (were) chosen for refinement, this gives an estimate of how much the solution would change if this triangle(s) was (were) refined.

The local problem error indicator is very accurate, but it is relatively expensive, because of the computation required to set up and solve the elemental matrices for each triangle. A less expensive approach, given by

`HIERARCHICAL_COEFFICIENT`, is to simply examine the hierarchical coefficients (see [14]) of  $u_T$  over each triangle, which is similar to computing the local problem indicator one refinement back. This is a very fast indicator which is less accurate than the local problem indicator, but often adequate for creating a good adaptive grid. Currently, this indicator can only be used for `reftype=H_ADAPTIVE` and `degree=1`.

The `TRUE_DIFF` error indicator uses the difference between the true solution (if given, see 3.2.4) and  $u_T$ . This is primarily available for comparison with new error indicators and would not normally be used.

### 3.3.4 Linear system solver

Another major phase of `phaml_solve_pde` is the solution of the linear system of equations to get the coefficient vector of the solution. PHAML contains a hierarchical basis multigrid method (see [15][17][18]) as the primary linear system solver. It also contains hooks into several freely available software packages of direct and iterative parallel solvers for comparative studies and for situations where the multigrid algorithm is not appropriate (for example, indefinite problems). For information on obtaining the optional software discussed in this section, see Section 2.1. The solver is selected by the arguments `solver` and `preconditioner`.

#### 3.3.4.1 Hierarchical basis multigrid solver

The hierarchical basis multigrid solver (HBMG) is selected by `solver = MG_SOLVER`. The approach is to cycle through a sequence of nested grids. The grids come from the refinement process with the  $\ell^{\text{th}}$  grid consisting of triangles with refinement level up to  $\ell$ . Equations corresponding to high order face basis functions are removed from the system by static condensation [18] before the multigrid cycles, and solved directly after the multigrid cycles are complete. In each cycle, the equations corresponding to high order edge basis functions are first relaxed by some Gauss-Seidel iterations and the residual is injected into the equations corresponding to the linear basis functions. The equations corresponding to the linear basis functions are relaxed by some red-black Gauss-Seidel iterations. The black equations are those corresponding to equations in the coarse grid, and the red equations are from the fine grid but not the coarse grid. A basis change is then performed on those equations to convert from a

nodal basis to a 2-level  $h$ -hierarchical basis, and the coarse grid equations are extracted. The process of relaxation, basis change and extraction is repeated until the grid consists only of the initial grid, i.e. all elements have refinement level 1. The coarsest grid problem is solved with a LAPACK direct solver. The process is then reversed by performing relaxation followed by conversion from 2-level  $h$ -hierarchical basis to nodal basis to get the next finer grid. When the cycle has returned to the finest grid, another relaxation is performed on the equations corresponding to the high order edge basis functions. This constitutes one multigrid V-cycle. Cycles are repeated until some termination criterion is met.

There are several arguments to `phaml_solve_pde` that determine the specifics of the multigrid algorithm. `mg_prerelax_ho`, `mg_postrelax_ho`, `mg_prerelax` and `mg_postrelax` give the number of half-red-black Gauss-Seidel iterations to perform before and after solving the coarse grid problem for the high order equations and the linear equations, respectively. A half iteration relaxes the red equations but not the black equations. So, for example, `mg_prerelax=2` specifies one red-black iteration.

The termination of the multigrid cycles can be specified as a fixed number of cycles or by a tolerance on the residual of the linear system. To terminate by a tolerance, specify the tolerance in `mg_tol`. Iterations continue until the  $\ell^2$  norm of the residual of the scaled linear system is less than `mg_tol`. There are two symbolic constants that provide special values for `mg_tol`. `MG_ERREST_TOL` says to cycle until the residual is reduced to some fraction of the global error estimate. This avoids excess computation from solving the system more accurately than is necessary relative to the discretization error. `MG_NO_TOL` says to use a fixed number of iterations rather than a tolerance. The argument `mg_cycles` gives the fixed number of cycles to perform. Used in conjunction with a tolerance, `mg_cycles` provides an upper bound on the number of cycles to guarantee the iteration will terminate.

### 3.3.4.2 Alternative direct solvers

#### 3.3.4.2.1 LAPACK

The LAPACK library [4] is always linked with a PHAML program. In limited situations it can be used as the linear system solver. It is specified by `solver=LAPACK_SPD_SOLVER` for most problems, which generate a symmetric positive definite matrix, or by `solver=LAPACK_INDEFINITE_SOLVER` for problems that generate an indefinite symmetric matrix. However, it can only be used as the solver for sequential programs because it is not a parallel library. And it should only be used for relatively small problems.

#### 3.3.4.2.2 MUMPS

MUMPS [2][3] is a parallel direct solver from the European project PARASOL. Currently the PHAML interface to MUMPS only supports double precision, i.e. `my_real=kind(0.0d0)` in `global.f90`. MUMPS is specified with

PETSC_RICHARDSON_SOLVER	Richardson
PETSC_CHEBYCHEV_SOLVER	Chebyshev
PETSC_CG_SOLVER	conjugate gradients
PETSC_BICG_SOLVER	BiConjugate Gradients
PETSC_GMRES_SOLVER	generalized minimal residual
PETSC_BCGS_SOLVER	biconjugate gradients stabilized
PETSC_CGS_SOLVER	conjugate gradient squared
PETSC_TCQMR_SOLVER	transpose-free quasi-minimal residual
PETSC_TFQMR_SOLVER	transpose-free quasi-minimal residual
PETSC_CR_SOLVER	conjugate residual
PETSC_LSQR_SOLVER	least squares

Table 4: Available values of `solver` for the PETSc solvers.

NO_PRECONDITION	no preconditioning
MG_PRECONDITION	one hierarchical basis multigrid V-cycle
FMG_PRECONDITION	one F cycle of the MG preconditioner
FUDOP_DD_PRECONDITION	an experimental domain decomposition
COARSE_GRID_PRECONDITION	precondition with solution on a coarse grid
PETSC_JACOBI_PRECONDITION	Jacobi
PETSC_BJACOBI_PRECONDITION	block Jacobi
PETSC_SOR_PRECONDITION	SOR and SSOR
PETSC_EISENSTAT_PRECONDITION	SOR with Eisenstat trick
PETSC_ICC_PRECONDITION	incomplete Cholesky
PETSC_ILU_PRECONDITION	incomplete LU
PETSC_ASM_PRECONDITION	additive Schwarz

Table 5: Available values of `precondition` for the PETSc solvers.

`solver=MUMPS_SPD_SOLVER` or `solver=MUMPS_GEN_SOLVER` for the symmetric positive definite and general solvers, respectively.

### 3.3.4.2.3 SuperLU

SuperLU [13] is a parallel direct solver from the Lawrence Berkley National Laboratories. It is specified by `solver=SUPERLU_SOLVER`.

### 3.3.4.3 Alternative iterative solvers

#### 3.3.4.3.1 PETSc

PETSc [5][6] is a parallel library of preconditioners and Krylov space iterative solvers from Argonne National Laboratories. PHAML provides access to most of the methods in PETSc and several parameters to those methods. Table 4 contains the available PETSc solvers as specified by `solver`. Table 5 contains the

real(my_real)	petsc_richardson_damping_factor
real(my_real)	petsc_chebychev_emin
real(my_real)	petsc_chebychev_emax
integer	petsc_gmres_max_steps
real(my_real)	petsc_rtol
real(my_real)	petsc_atol
real(my_real)	petsc_dtol
integer	petsc_maxits
integer	petsc_ilu_levels
integer	petsc_icc_levels
real(my_real)	petsc_ilu_dt
real(my_real)	petsc_ilu_dtcol
integer	petsc_ilu_maxrowcount
real(my_real)	petsc_sor_omega
integer	petsc_sor_its
integer	petsc_sor_lits
logical	petsc_eisenstat_nodiagscaling
real(my_real)	petsc_eisenstat_omega
integer	petsc_asm_overlap

Table 6: Arguments that are passed to the PETSc solvers.

HYPRE_BOOMERAMG_SOLVER	algebraic multigrid
HYPRE_PCG_SOLVER	preconditioned conjugate gradients
HYPRE_GMRES_SOLVER	generalized minimal residual

Table 7: Available values of `solver` for the hypre solvers.

available preconditioners for the PETSc solvers specified by `preconditioner`. The FMG, FUDOP\_DD and COARSE\_GRID preconditioners are experimental and may be removed in a future release of PHAML. Their use is discouraged.

There are a number of arguments that are used as parameters to the PETSc methods. These are listed in Table 6. Refer to the PETSc User’s Manual [6] for explanations of these arguments.

If the preconditioner is not one of the ones starting with PETSC, then memory can be saved by not copying the matrix to the PETSc format. This is specified by `petsc_matrix_free=.true.`

### 3.3.4.3.2 hypre

hypre [10][11] is a package of parallel iterative solvers and preconditioners from Lawrence Livermore National Laboratories. The hypre solvers are listed in Table 7 and preconditioners are listed in Table 8. Note that the BoomerAMG solver does not use a preconditioner, and the ParaSails preconditioner cannot be used with the PCG solver. There are also several arguments that are passed

NO_PRECONDITION	no preconditioning
HYPRE_DS_PRECONDITION	diagonal scaling
HYPRE_BOOMERAMG_PRECONDITION	algebraic multigrid
HYPRE_PARASAILS_PRECONDITION	sparse approximate inverse (GMRES only)

Table 8: Available values of `precondition` for the hypre solvers.

integer	hypre_BoomerAMG_MaxLevels
integer	hypre_BoomerAMG_MaxIter
real(my_real)	hypre_BoomerAMG_Tol
real(my_real)	hypre_BoomerAMG_StrongThreshold
real(my_real)	hypre_BoomerAMG_MaxRowSum
integer	hypre_BoomerAMG_CoarsenType
integer	hypre_BoomerAMG_MeasureType
integer	hypre_BoomerAMG_CycleType
integer	hypre_BoomerAMG_NumGridSweeps(:)
integer	hypre_BoomerAMG_GridRelaxType(:)
integer	hypre_BoomerAMG_GridRelaxPoints(:,:)
real(my_real)	hypre_BoomerAMG_RelaxWeight(:)
integer	hypre_BoomerAMG_IOutDat (not available after hypre 1.6.0)
integer	hypre_BoomerAMG_DebugFlag
real(my_real)	hypre_ParaSails_thresh
integer	hypre_ParaSails_nlevels
real(my_real)	hypre_ParaSails_filter
integer	hypre_ParaSails_sym
real(my_real)	hypre_ParaSails_loadbal
integer	hypre_ParaSails_reuse
integer	hypre_ParaSails_logging
real(my_real)	hypre_PCG_Tol
integer	hypre_PCG_MaxIter
integer	hypre_PCG_TwoNorm
integer	hypre_PCG_RelChange
integer	hypre_PCG_Logging
integer	hypre_GMRES_KDim
real(my_real)	hypre_GMRES_Tol
integer	hypre_GMRES_MaxIter
integer	hypre_GMRES_Logging

Table 9: Arguments that are passed to the hypre solvers.



to the hybre methods listed in Table 9. See the hybre user’s guide (distributed with the software) for an explanation of these.

### 3.3.5 Load balancing

A parallel program with adaptive grid refinement must perform dynamic load balancing. When the grid is refined adaptively, some processors will perform more refinement than others, resulting in more grid elements, and hence more of the computational load, than others. Dynamic load balancing redistributes ownership of the elements to balance the load among the processors.

In PHAML, load balancing is performed by partitioning the grid into  $P$  sets where  $P$  is the number of slave processes. Each process is said to own the triangles in one of the partitions. Each vertex and edge has an associated triangle and is owned by the process that owns that triangle. PHAML uses a full domain partition (FuDoP) [16] in which each process also has additional coarse elements that cover the unowned part of the domain. Figure 5 illustrates an adaptively refined grid and the grid that would be seen by each of three processors, with the color indicating ownership.

One normally thinks of performing load balancing after refinement to redistribute the new grid. But it is also possible to perform predictive load balancing before refinement, which can reduce the amount of data to be redistributed. There also may be advantages to basing the balance on the number of elements, number of vertices or number of equations. PHAML provides the choice of balancing any of these entities before and/or after refinement. For balancing after refinement, it divides up the number of entities equally. For balancing before refinement, it uses the error indicator as a weight in the partitioning algorithm, so that those that are likely to be refined more times are given more weight. The selection of what to balance and when to balance is given by the arguments `prebalance` and `postbalance` which can take the value `BALANCE_NONE`, `BALANCE_ELEMENTS`, `BALANCE_VERTICES` or `BALANCE_EQUATIONS`. Usually, one of the two arguments is `BALANCE_NONE` so that either regular or predictive balancing is performed, but it is possible to perform balancing on both sides of refinement.

Grid partitioning algorithms have been the topic of extensive research, and there are several algorithms available. The method built into PHAML is the k-way refinement tree method [19]. This method uses the inherent refinement tree where the tree nodes correspond to triangles and the children of a tree node are the triangles created when a triangle is refined. Weights are attached to the leaf nodes and a tree traversal is performed to sum the weights. A second traversal is performed to partition the tree into  $P$  subtrees of equal summed weight. The children in the tree are ordered such that a tree traversal gives a space filling curve through the grid, and partitions are connected.

The choice of partitioning method is determined by the argument `partition_method`. The built in k-way refinement tree method is selected by the value `RTK`. Alternative partitioning methods are provided through the optional Zoltan dynamic load balancing library [8][9]. The values for selecting methods

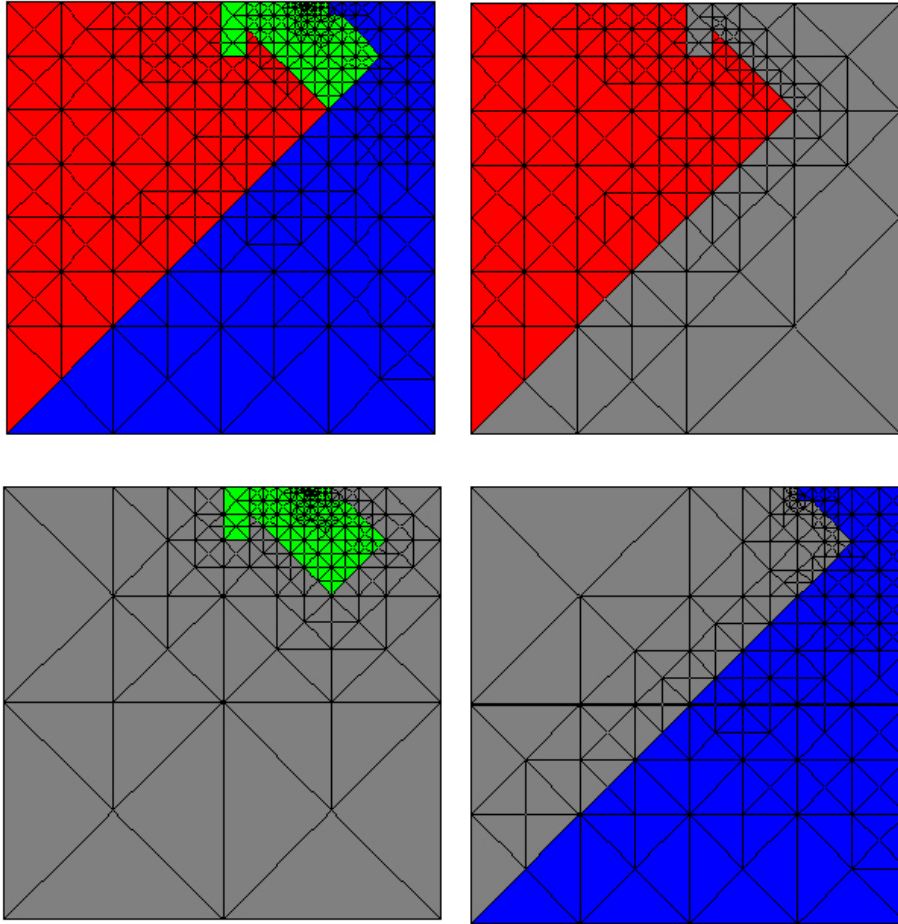


Figure 5: An adaptive grid partitioned for 3 processors, and the grid seen by each of the processors.

ZOLTAN_RCB	recursive coordinate bisection
ZOLTAN_OCT	RPI's Octree method
ZOLTAN_METIS	local diffusion method from ParMETIS
ZOLTAN_REFTREE	refinement tree
ZOLTAN_RIB	recursive inertial bisection
ZOLTAN_HSFC	Hilbert space filling curve

Table 10: Available values of `partition_method` for Zoltan partitioners.

from Zoltan are given in Table 10. See the Zoltan User's Guide for descriptions of these methods.

### 3.3.6 Termination

There are several ways to specify how long `phaml_solve_pde` should continue to work on the solution, i.e. when to terminate. This section presents them. At least one of them must be specified or else the subroutine will run forever. Multiple termination criteria can be specified, and the routine will return when any of them are satisfied.

The argument `task` selects what task should be performed by `phaml_solve_pde`. Three of the values for `task` invoke a single pass of one part of the program, and no further termination criterion is needed. These are `BALANCE_ONLY`, `REFINE_ONLY`, and `SOLVE_ONLY`. The value `BALANCE_REFINE_SOLVE` invokes a loop over the three phases until one of the termination criteria is met. `SET_INITIAL` also loops over the three phases, but for the solve phase it interpolates the function in subroutine `iconds` (see Section 7.3.5). This is usually used for setting initial conditions for a time dependent problem, or an initial guess for a nonlinear problem (see Section 4.3).

The termination criteria are given by optional arguments and fall into two categories. The first is those that specify an upper bound on the number of something. These arguments all take an `integer` value. The arguments are `max_elem` (elements), `max_vert` (vertices), `max_eq` (equations in the linear system), and `max_refsolveloop` (number of times through the refine/solve loop). You can also specify `max_lev` (refinement levels) and `max_deg` (polynomial degree), but these are not termination criteria – if the refinement of an element would exceed the given value, then it is simply not performed.

The other category is to terminate when an error estimate is small enough. The global error estimate is computed from the local error indicators and can be based on  $h$  or  $p$  as described in Section 3.3.3. The estimate can be an estimate of the energy norm,  $L^\infty$  norm, or  $L^2$  norm of the error. The arguments `term_h_energy_err`, `term_p_energy_err`, `term_h_Linf_err`, `term_p_Linf_err`, `term_h_L2_err` and `term_p_L2_err` all take a `real(my_real)` value.

## 3.4 I/O

### 3.4.1 I/O files

PHAML writes printed output to two units, which can be specified through the integer arguments `output_unit` and `error_unit` of subroutine `phaml_create`. Error messages and warnings are written to `error_unit` and all other printed output is written to `output_unit`. By default, `output_unit` is 6 which most compilers provide as a pre-connected unit, often referred to as `stdout`. Also by default, `error_unit` is 0 which most compilers provide as a pre-connected unit, often referred to as `stderr`. If your compiler uses different unit numbers for pre-connected units, the correct units can be specified through these arguments. The two units can be the same if you want to have error messages and

printed output intermixed. Warning messages from the master and slave processes can be suppressed by passing `print_warnings=.false.` to subroutine `phaml_solve_pde`. Error messages and warnings from graphics processes cannot be suppressed.

Usually `stdout` and `stderr` print to the terminal screen. In particular, they usually print to the window from which the program was started. However, the processes of a parallel program might not have an associated window, and the destination of `stdout` and `stderr` is determined by the parallel library. For example, if using a master/slave model with LAM as the parallel library, output from the master process will go to the window from which it was started, but output from the slave processes will go to the window from which `lamboot` was run. Moreover, output from all the slaves is intermixed in this window. To get more control over the destination of output from the slaves you can use `spawn_form=DEBUG_SLAVE` in `phaml_create`. This will open a window for each slave process and normally the printed output goes to those windows.

If you prefer to have the printed output directed to a file, then you can open a unit and specify that unit in `output_unit` and/or `error_unit`. PHAML provides “parallel open” and “parallel close” subroutines for this purpose. Subroutine `phaml_popen` (Section 7.4.10) opens a unit with a different file for the master and each of the slaves. If you specify the name of the file to be `root.suffix` then the actual filenames are `rootXXXX.suffix` for slave number `XXXX`, where the number of digits in `XXXX` is the minimum needed for the number of slaves. The master process is number 0. Subroutine `phaml_pclose` (Section 7.4.9) closes the unit.

Thus the usual process for directing printed output (or error) to files is: 1) call `phaml_create` with some non-pre-connected unit number in `output_unit`, 2) call `phaml_popen` with that unit number and some root file name, 3) call `phaml_solve_pde`, 4) call `phaml_pclose` with that unit number. Note that if there are any error messages printed by `phaml_create`, they cannot be written to this file because the unit has not yet been opened. These messages are written to unit 0 by default. You can specify a different unit through the argument `output_now`, but it must be a pre-connected unit.

### 3.4.2 Printed I/O

PHAML provides for the printing of various quantities at various times during the execution of `phaml_solve_pde`. By default, the only printed output is a header containing a summary of the input variables to `phaml_solve_pde` and a trailer containing termination information, printed by the master process. Other output is controlled through a series of “when” and “who” arguments. The “when” arguments indicate how often to print something. They take the values `NEVER` (don’t print that quantity), `FINAL` (print it just before returning), `PHASES` (print it once in each refine/solve loop), and `FREQUENTLY` (print more often, usually used for debugging). The “who” arguments indicate which processes should print something. They take the values `NO_ONE` (don’t print that quantity), `MASTER` (the master prints that quantity for the entire grid), `SLAVES` (each slave

prints that quantity for the part of the grid that it owns), `EVERYONE` (both the master and the slaves print), and `MASTER_ALL` (the master prints the quantity for each of the slaves).

`print_grid_when` and `print_grid_who` provide for printing a summary of the grid, such as the number of vertices, number of elements, etc. `print_linsys_when` and `print_linsys_who` provide for printing a summary of the linear system, such as the number of equations, etc.

`print_error_when` and `print_error_who` provide for printing certain norms of the error (if the true solution is provided) and error estimates.

`print_error_what` indicates what global norm(s) to print, with the available norms being the energy norm,  $L^\infty$  norm, and  $L^2$  norm. The values for this argument are `NEVER`, `ENERGY_ERR`, `LINF_ERR`, `L2_ERR`, `ENERGY_LINF_ERR`, `ENERGY_L2_ERR`, `LINF_L2_ERR`, and `ENERGY_LINF_L2_ERR`. The energy and  $L^2$  errors are approximated using a sixth order quadrature rule for the integrals. The  $L^\infty$  norm is approximated using the quadrature points of a sixth order quadrature rule.

`print_errest_what` controls which global error estimates are printed. It takes values that are similar to `print_error_what` except that “ERR” is replaced by `HERREST`, `PERREST` or `HPERREST`. `HERREST` indicates that the estimate of the requested norm(s) should be computed using the  $h$ -refinement error indicator, `PERREST` indicates the  $p$ -refinement error indicator should be used, and `HPERREST` indicates that both indicators should be printed.

`print_error_when` also provides for monitoring the convergence rate of the iterative linear system solver, if the solver is the built-in hierarchical basis multi-grid method or one of the solvers from PETSc. If it has the value `FREQUENTLY` then the  $\ell^2$  norm of the residual is printed after each iteration. It also accepts the value `TOO_MUCH` which additionally sets the solution to 0.0 before starting the iterations, to avoid convergence in 1 iteration.

`print_time_when` and `print_time_who` provide for measuring execution time of the program. The execution time of each section of the program (refinement, reconciliation, load balancing, matrix assembly, linear system solution, communication) is printed. Each time the time is printed it prints the time for the most recent pass through the refine/solve loop and the total time so far. `clocks` determines how the time is measured. It can be `CLOCK_C` for the cpu clock, `CLOCK_W` for the wall clock, or `CLOCK_CW` for both.

`print_header_who` and `print_trailer_who` specify which processes should print the header and trailer, respectively.

### 3.4.3 Pausing

In order to examine printed or graphical output while a PHAML program is running, it is often useful to have the program pause until you indicate that it may continue. There are several “pause” arguments to `phaml.solve_pde` that provide for this. They are logical arguments. When the program pauses, the master prints “press return to continue” to its printed output unit. It then reads from standard input, which is usually associated with the window from

which the master program was run. The pause arguments are `pause_at_start` (pause upon entering `phaml_solve_pde`), `pause_at_end` (pause before leaving `phaml_solve_pde`), `pause_after_phases` (pause at the end of each refine/solve loop), and `pause_after_draw` (pause each time the graphics is updated).

## 3.5 Graphics

### 3.5.1 Overview

Graphics, or visualization, in PHAML is provided using OpenGL, which is a platform-independent specification of a graphics application programming interface, and GLUT, which supplies window, keyboard and mouse usage for OpenGL. There are libraries for both of these specifications available for nearly every computer. See Section 2.1.5 for URLs for more information on OpenGL and GLUT and how to obtain the software.

PHAML's graphics are interactive and menu driven. There may be graphics windows associated with the master process and/or with each of the slave processes. The master's graphics shows the grid and solution as a whole entity, whereas the slaves each show the grid and solution as they know it. The selection of which processes will have graphics is made with the argument `draw_grid.who` to `phaml_create`, which can have the value `MASTER`, `SLAVES`, `EVERYONE`, or `NO_ONE`. There is no option to have only a subset of the slaves do graphics.

The graphics servers, which are separate processes in the parallel program, receive messages from the associated master or slave whenever the grid or solution changes. For a program under a sequential compilation of PHAML, the messages are passed by writing files to `/tmp`. For a master/slave program, the graphics servers are spawned automatically during the execution of subroutine `phaml_create`. For an SPMD program, the graphics servers should be included in the number of processes launched (see Section 2.4). For a sequential program, you must start the graphics server, `phaml_graphics`, from the command line. If a sequential program terminates abnormally, you must terminate `phaml_graphics` by hand, and should check `/tmp` for leftover files `phaml_message` and `phaml_lock`, and remove them if they exist.

All interaction with the graphics is performed with the mouse and arrow keys. The left and middle mouse buttons and arrow keys can be assigned various operations to change the view, as discussed in Section 3.5.3. By default, the left button rotates, the middle button zooms and the arrow keys pan. The right button brings up a menu of actions you can take to modify the graphics. The main menu is shown in Table 11. Menu items with a right arrow ( $\Rightarrow$ ) bring up submenus.

The next section will show some of the visualizations that are available. The subsequent sections will discuss how to manipulate the graphics through the submenus.

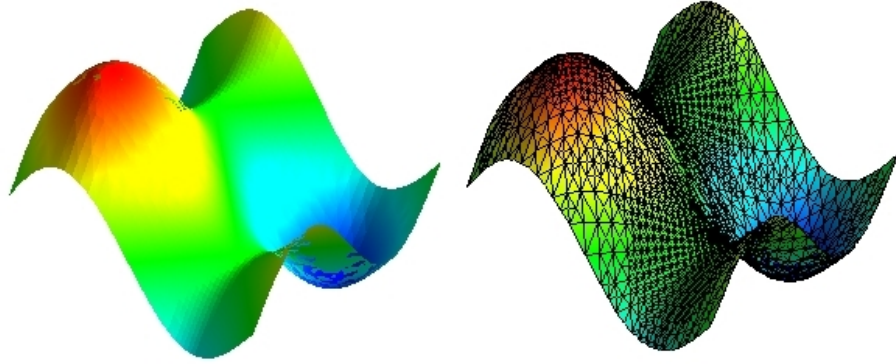
view modifier	⇒
element edge color	⇒
element interior color	⇒
function	⇒
contour plots	⇒
preprocess function	⇒
subelement resolution	⇒
color scheme	⇒
toggle lights	⇒
element label	⇒
edge label	⇒
vertex label	⇒
associated element	⇒
eigenfunction to use	⇒
component to use	⇒
space filling curve	⇒
grid offset	⇒
crop (debug window)	
toggle axes	
write postscript	⇒

Table 11: The main graphics menu.

### 3.5.2 Example visualizations

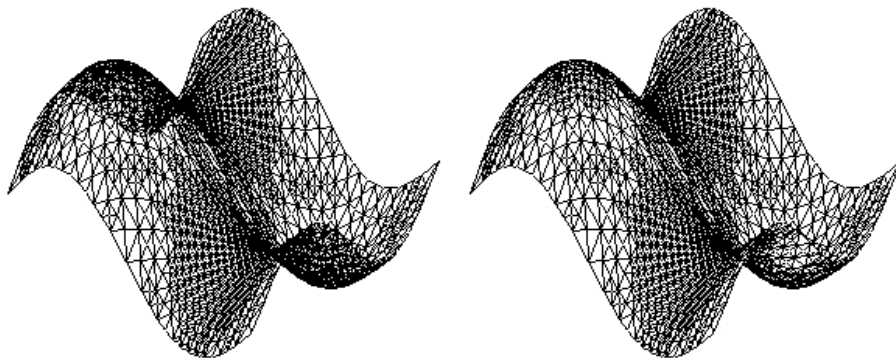
The PHAML graphics server has many options in the graphics menu. Primarily the graphics options are for different displays of the grid properties, functions over the grid, and the partitioning of the grid over the slaves. The functions that can be displayed include not only the computed solution, but also the true solution and error (if the true solution is given), and error estimates. In this section we present some examples of PHAML’s graphics capability with very brief descriptions. The subsequent sections will discuss how to use the submenus to create these and other graphical displays.

Figure 6 shows several of the different ways you can display a function. Parts (a)–(f) show surface plots of the solution, which comes from using the menu to select `function` to be `solution`. Part (a) shows the surface with the triangle interiors colored by the solution value using the rainbow color scheme where blue represents small values and red represents large values. In part (b) the same function is shown with the grid added to the surface, drawn in black. Part (c) displays the grid in black on the surface, but selects `element interior color` to be `transparent`. Part (d) is the same but with hidden lines removed, which is achieved by selecting `element interior color` to be `white` so that the triangle faces hide the grid lines behind them. Part (e) also has the grid on the surface and hidden lines removed, but with the grid lines colored by solution value. Part (f) is a surface plot using the striped color scheme and placing contour



(a)

(b)

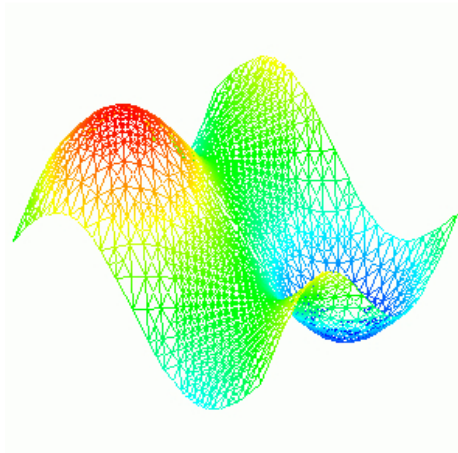


(c)

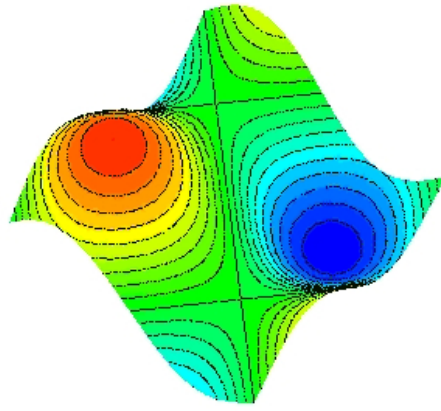
(d)

Figure 6: Visualizations of the solution.

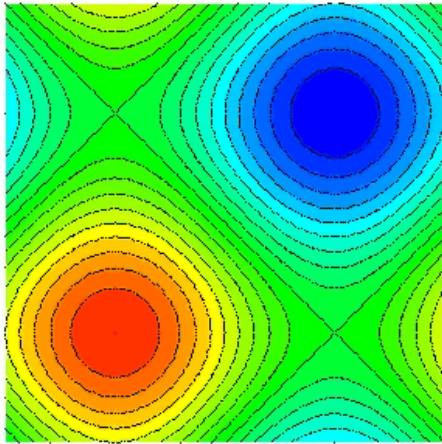




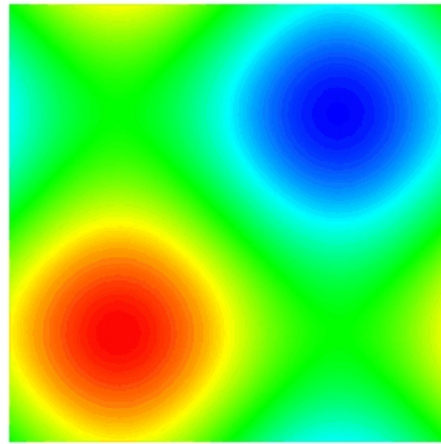
(e)



(f)

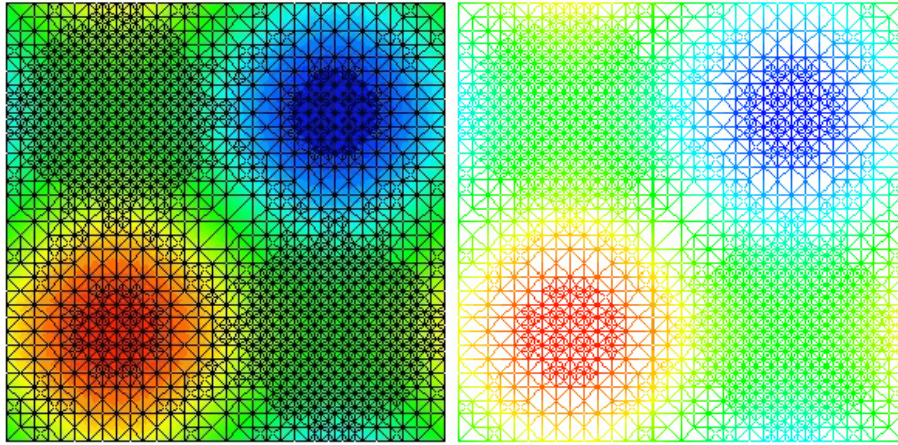


(g)



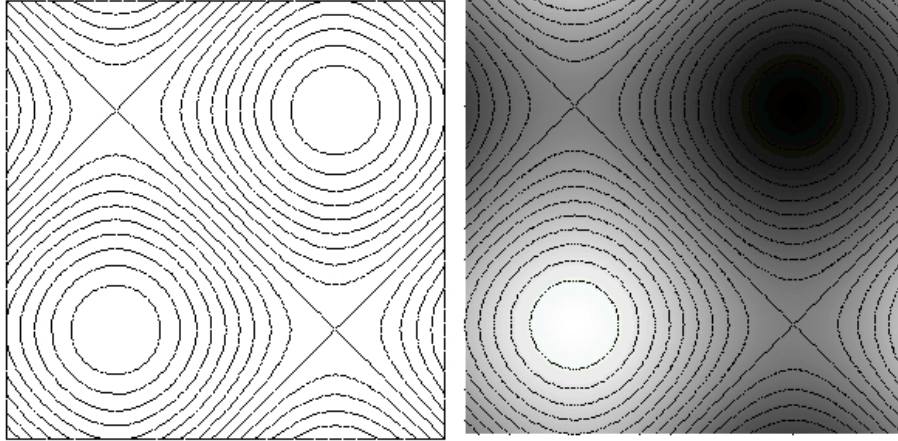
(h)

Figure 6: (continued) Visualizations of the solution.



(i)

(j)



(k)

(l)

Figure 6: (continued) Visualizations of the solution.

left mouse button	⇒
middle mouse button	⇒
arrow keys	⇒
reset to initial view	
view from above	
view from above origin	
quit	

Table 12: The view modifier submenu.

lines on the surface. Parts (g)–(l) show 2D displays of the solution obtained by selecting `function` to be `no function` and viewing from straight above. Part (g) uses the striped color scheme with contour lines drawn in the  $x$ - $y$  plane. Part (h) shows the solution using the rainbow color scheme. Part (i) adds the grid to the previous image, and part (j) colors the triangle interiors transparent and the grid lines with the solution value. Part (k) is simply a contour plot with no grid lines and the triangle interiors transparent. Finally, part (l) adds the coloring of the triangle interiors by the solution value using the gray scale color scheme.

Figure 7 shows some additional visualizations of the grid. Part (a) simply shows the grid drawn in black. Part (b) shows the regions in which the grid is coarse or fine by coloring the interiors of the triangles by the triangle size, using blue for small triangles and red for large. In part (c) the triangle interiors are colored by the polynomial degree of the triangle, for visualizing the effect of  $p$ - and  $hp$ -adaptive refinement. And in part (d) we use `function ⇒ levels` to create a hierarchy of grids based on refinement level.

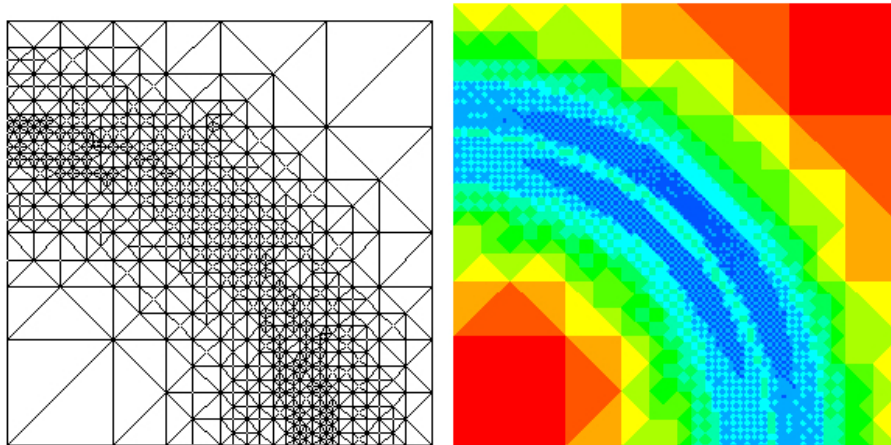
Figure 8 shows a few of the ways you can visualize how the grid has been partitioned for distribution over the slaves during the load balancing step. In part (a) the interior of the triangles is colored by owner and the grid lines are colored black. Part (b) colors the grid lines by owner and leaves the interior transparent. In part (c) the interiors are again colored by owner, but only the triangle edges that are on partition boundaries are drawn. Finally, part (d) shows an exploded grid where the partitions are physically separated.

### 3.5.3 View modifier

In this section, we discuss the view modifier. The view modifier submenu is shown in Table 12.

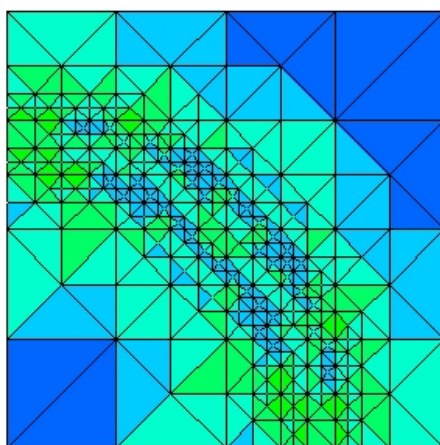
The first three items set the operation performed by the left mouse button, middle mouse button and arrow keys, respectively. The selected operation remains in effect until you select another one. The submenus below each of these are all the same. They contain `rotate`, `zoom`, `pan`, `scale x` (stretch or shrink along the  $x$  axis), `scale y`, `scale z`, `explode` (as in Figure 8(d)), and `move light` (see Section 3.5.6).

The next three entries are quick changes to a specific view of the grid. `reset`



(a)

(b)



(c)



(d)

Figure 7: Visualizations of the grid.

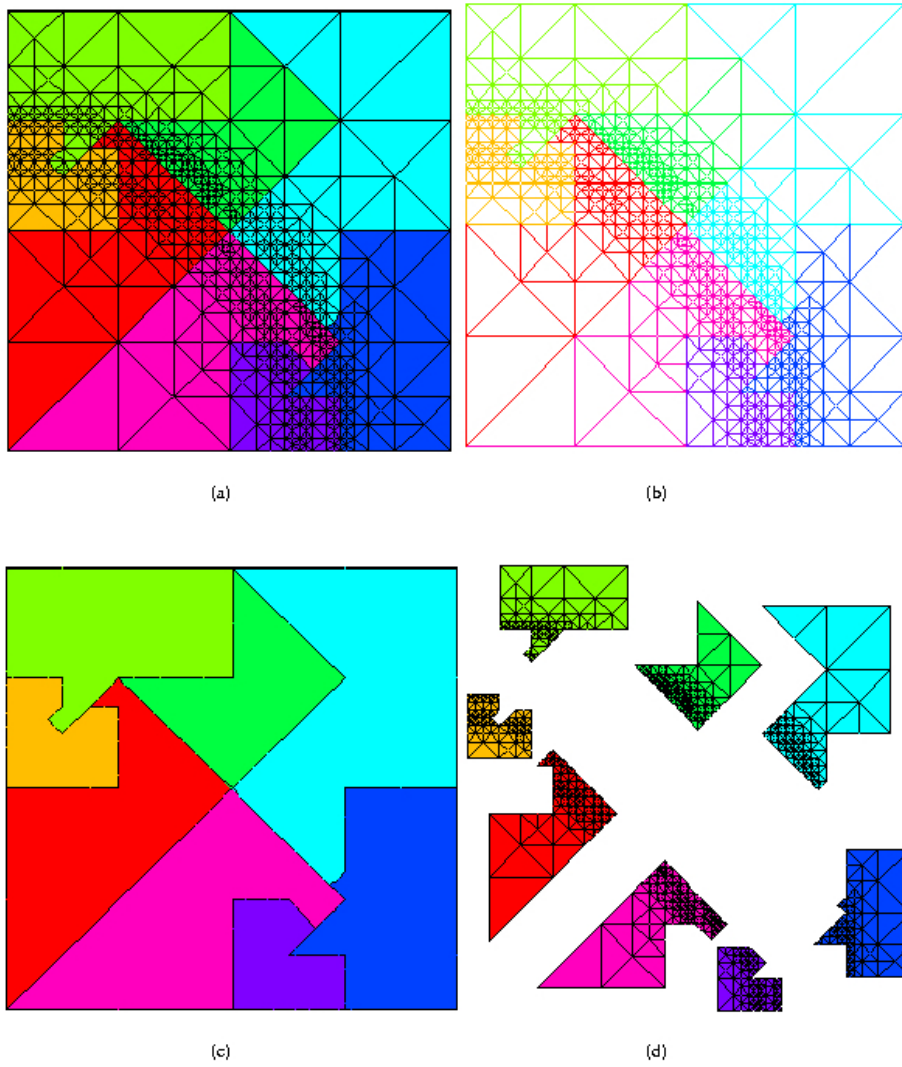


Figure 8: Visualizations of the partitions of the grid for load balancing.

to **initial view** resets the rotation, zoom and pan settings to the original settings. **view from above** sets a view that looks straight down and at the center of the grid, with the (xmin,ymin) corner of the domain at the lower left corner of the image. This is normally used with **function**  $\Rightarrow$  **no function** for visualizations of the grid and contour plots. **view from above origin** looks at the point (0,0,0) from straight above. This is useful if you want to zoom in on the origin without having to pan repeatedly.

The final entry, **quit**, should not normally be used. If you terminate the graphics program using this menu item, it may or may not kill the master and/or slaves. The graphics program will terminate properly when **phaml\_destroy** is called. Using the **quit** entry in the menu is only useful if the master or associated slave terminated early, and even then there might be a better way to terminate the graphics server (for example, with LAM the command **lamclean** will terminate all the processes).

### 3.5.4 Colors

By default, PHAML uses a rainbow color scheme. Colorization of items that have a continuous numerical value, like the solution, comes from a continuous spectrum with blue corresponding to small values and red to large values. Normally the spectrum is scaled to correspond to the minimum and maximum values of the item being drawn. Colorization of items that have a finite set of  $N$  discrete values, like the owner, comes from an equally spaced set of  $N$  colors in the spectrum from red to magenta with red assigned to the first item.

There are two other color schemes available. The color scheme is selected in the submenu under **color scheme** which has the selections **rainbow**, **gray scale**, and **striped**. The gray scale scheme uses a continuous gray scale from black to white with black corresponding to small values and white to large values. The striped scheme is like the rainbow scheme except the colors are from a discrete set of equally spaced colors from the spectrum from blue to red. The number of colors in the scheme is one less than the number of contour lines used for contour plots, so that contour lines will fall on the boundary between colors.

With a color scheme selected, you can choose how to color the grid lines (i.e. element edges) and triangles (i.e. element interiors) to get different information about the grid, partition, solution, error, etc. Table 13 shows the element edge color submenu for selecting how to color the element edges. **no lines** means don't draw the grid. **black** draws the edges black for a simple drawing of the grid. **edge owner** selects the color to indicate which slave process owns each edge. **vertex owner** selects the color to indicate the owner of the vertices at the ends of each edge. If the owners of the two endpoints are different, the color blends from one to the other along the length of the edge. **computed solution** uses the color corresponding to the value of the computed solution at the endpoints, and blends the colors along the length of the edge. **true solution** is similar but uses the true solution if it is given in function **trues**. Likewise, **error** uses the computed solution minus the true solution, if it is available. **size** colors the edges according to their length, using red for the largest edges and blue for

no lines
black
edge owner
vertex owner
computed solution
true solution
error
size
degree
partition boundary black

Table 13: The element edge color submenu.

the smallest. This is useful for distinguishing element sizes when the grid is so fine that drawing the edges black would result in large black areas. `degree` colors the edges by the polynomial degree of the approximation space along that edge. Finally, `partition boundary black` draws only the edges that fall between two triangles with different owners, and edges on the domain boundary. This results in the partition boundary being drawn black.

The submenu for element interior color is similar to the one for element edge color, with just a few differences. The `no lines` entry is replaced by `transparent`, but has the same meaning of don't draw the element. `black` is replaced by `white`. Coloring an element interior white, which is the same as the graphics window background, has the effect of blocking whatever is behind the element, and can be used to draw hidden line plots. There is only one `owner` entry, used to indicate which slave owns each element. `computed solution`, `true solution`, `error`, `size`, and `degree` all have the same meaning as with edge color. There is no partition boundary entry. It has an additional entry for coloring each element by its error indicator. This entry has a submenu to select whether to color according to the  $h$  error indicator or the  $p$  error indicator. It also has entries for  $h$  and  $p$  error estimates, but those are currently the same as the error indicators. You also have the choice of scaling the two types of indicators individually or collectively. By scaling collectively you can switch between the two to see whether the  $h$  or  $p$  error indicator is larger in an element.

### 3.5.5 Functions

The function submenu determines what function to draw as surface plots and contour plots. This menu has six entries. The first is `no function`, which simply draws the elements in the  $x$ - $y$  plane. This is usually used in conjunction with viewing from above. The next three entries are `computed solution`, `true solution`, and `error`. They draw the corresponding function. The fifth entry is `levels`. This draws the elements, including parent elements, in  $L$  discrete planes parallel to the  $x$ - $y$  plane, where  $L$  is the number of refinement levels in the grid (see Figure 7(d)). Each level contains the elements of that refinement level. You may find it

useful to use `scale z` from the view modifier (Section 3.5.3) with this. The final entry is `error estimates` with the same submenu as the error estimates submenu under `element interior color` (Section 3.5.4). This is a piecewise constant function with each element drawn at the height corresponding to its error indicator.

The functions (except levels) can be preprocessed in several ways with the submenu under `preprocess function`. The menu entries indicate what would be applied to a function  $f$ . They are `none` (no preprocessing, the default), `-f` (negate), `abs(f)` (absolute value), `f**2` (square), and `log(abs(f))` (logarithm of the absolute value).

### 3.5.6 Lights

When the function is `no function`, `levels` or an error estimate, or the striped color scheme is used, the image is rendered in flat light. Otherwise, it is rendered with a light source resulting in a 3D appearance with shadows. There are five lights available in the PHAML graphics. Four of them are fixed in position: one to the right, one to the left, one above and one below. The fifth light is movable. The default is that only the light to the right is turned on. Lights can be turned on and off with the submenu under `toggle lights`. The movable light can be moved with the mouse or arrow keys by selecting `move light` under the `view modifier` submenu.

### 3.5.7 Contour plots

In addition to being drawn as a surface or being represented by color, a function can be drawn as a contour plot. Contour plots are controlled by the submenu under `contour plots`. This submenu has four entries for selecting what function to plot, and three submenus for changing the properties of the contour plot.

The first four entries in the contour plot submenu select the function to plot. As usual, they are `no contour plot`, `computed solution`, `true solution`, and `error`. By default, no contour plot is drawn.

The next entry is `set number of uniform lines`. Initially PHAML uses 21 equally spaced contour values between the minimum and maximum values of the function. This submenu allows you to change the number of values while keeping them equally spaced. There are six entries that allow you to make the change directly from the menu: `increment by 1`, `decrement by 1`, `increment by 10`, `decrement by 10`, `double`, and `cut in half`. The final entry, `enter number in debug window`, lets you type in the number of contour lines you want. However, it prompts for this number from standard input, so the graphics process must have an associated window. This is achieved by using `spawn_form = DEBUG_GRAPHICS` or `spawn_form = DEBUG_BOTH` in the call to `phaml_create` (see Section 7.4.4).

You can also space the contour lines nonuniformly by using `set nonuniform lines`. This also requires a debug window. You will be prompted for the number of contour lines, and then to enter a comma separated list of the values for the contour lines.



The final menu entry gives two choices for the placement of the contours. They can either be placed on the  $x$ - $y$  plane, or be elevated by the function value, i.e., placed on the surface.

### 3.5.8 Multiple solutions

If you solve a coupled system of equations (multicomponent solution) or solve for more than one eigenvalue of an eigenvalue problem, then there is more than one solution to draw. Two submenus let you select which function to display. The default is to display the first one.

**eigenfunction to use** provides the selection of which eigenfunction to display. It has an entry of the form **eigenfunction 1**, **eigenfunction 2**, etc., up to the number of eigenvalues computed. If there are more than 9 eigenvalues, the last entry is **more**  $\Rightarrow$ . This brings up a submenu containing **10's**  $\Rightarrow$ , **20's**  $\Rightarrow$ , etc. Under these submenus you will find the eigenfunctions with larger indices. PHAML sorts the eigenvalues from smallest to largest, and orders the corresponding eigenfunctions the same way.

**component to use** provides the selection of which component of a multicomponent solution to display. It contains entries of the form **component 1**, **component 2**, etc. Like the eigenfunction selection, there are submenus for the higher indices if there are more than 9 components. The menu also contains two entries for displaying a composite of the components. If the components of the solution are  $u_1, u_2, \dots$ , then **L1 sum** will display  $|u_1| + |u_2| + \dots$ , and **L2 sum** will display  $u_1^2 + u_2^2 + \dots$ .

### 3.5.9 Miscellaneous features

This section addresses some menu items that are not big enough to warrant a section of their own.

The **subelement resolution** submenu is useful when high order elements are used. By default, a triangle is drawn as a piece of a plane defined by the three vertices of the triangle. Color blending, contour lines, etc., are also limited by this definition. This applies both to 3D surfaces and 2D drawings in the  $x$ - $y$  plane. This is fine for piecewise linear elements since the solution is a plane over each triangle. But with higher order element, any detail on the subelement level is lost. This menu entry defines how much subelement resolution to use. The drawing is still done by drawing pieces of a plane, i.e. triangles, but multiple triangles are drawn within a grid element. With subelement level 0, the element is drawn as a single triangle (the default). With subelement level 1, the element is drawn as 4 triangles, formed by connecting the midpoints of the element edges. At subelement level 2, each of those 4 triangles are drawn as 4 triangles by connecting the midpoints of their sides, resulting in 16 triangles in a grid element. In general, subelement level  $\ell$  results in drawing  $4^\ell$  triangles in each element. The effect of using a high level of subelement resolution can be seen in Figure 9.

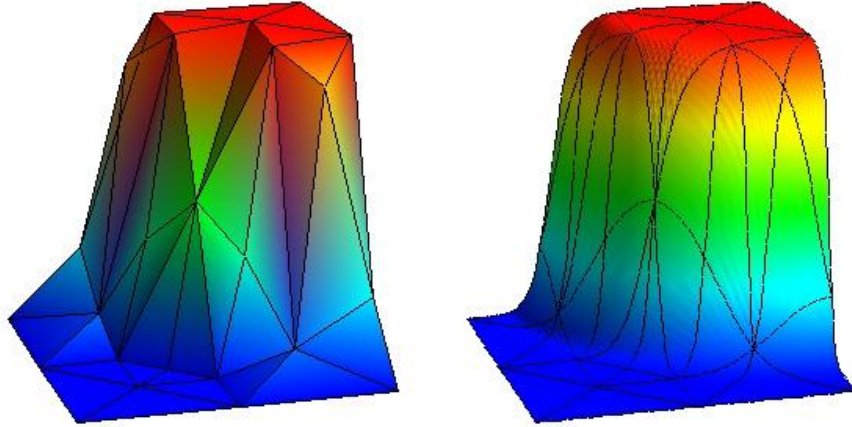


Figure 9: Improvement of a surface visualization by using subelement resolution.

`subelement resolution` has entries for 0, 1, 2 and 3 levels of subelement resolution. Since the number of triangles drawn grows exponentially with the number of subelement levels, using higher values of subelement resolution can be very slow. However, if needed, values larger than 3 are obtainable with the submenu entries `increase`, which adds 1, and `decrease`, which subtracts 1.

`x`, `y` and `z` axes can be added and removed from the plot with the `toggle axes` entry. The axes are fairly primitive with tic marks and values only at the ends and midpoint.

`crop (debug window)` provides a means of restricting how much of the domain is used in the display. Since it requires input from standard input, the graphics process must have an associated window. This is achieved by using `spawn_form = DEBUG_GRAPHICS` or `spawn_form = DEBUG_BOTH` in the call to `phaml.create` (see Section 7.4.4). When this menu entry is selected, it will prompt for the crop region to be entered as `xmin`, `xmax`, `ymin`, `ymax`.

The `grid offset` submenu helps to solve a potential problem with the rendering of the graphics. The elements are drawn as triangles and the grid lines are drawn as lines separately, but they occupy the same space. This can cause a problem in determining which should show when the image is rendered. Even if they are separated very slightly, there can be a problem because of machine roundoff error. This submenu allows you to change how far the grid lines are offset from the triangles. If you find that grid lines are disappearing, you should increase the offset. If you find that the grid lines appear to be separated from the surface, you should decrease the offset. The submenu entries let you increase or decrease by 1 or 10 at a time.

### 3.5.10 Development aids

Some of the graphics options were created as aids in the development and debugging of PHAML. These are probably not much use to the end user. They are the `element label`, `edge label`, and `vertex label`, which label each entity with its index in PHAML's data structure, `associated element` which displays the edge-element and vertex-element correspondence used to determine the owner of edges and vertices, and `space filling curve` which displays the space filling curve associated with the refinement tree partition method.

### 3.5.11 Postscript

The `write postscript` submenu lets you save the current visualization to an encapsulated postscript file. It writes vector graphics in the postscript language. This means the saved image is high quality and scalable, but it creates very large files. If you want smaller files you can use a screen capturing program and save the image in a raster graphics format like JPEG.

`write postscript` has two options in the submenu: `unsorted` and `sorted`. The `sorted` option sorts the entities of the image by distance from the viewer before writing the postscript file. There isn't really any reason to use the `unsorted` option, and it will probably be removed in the future.

When `write postscript` is selected, it creates a file called `render.eps`. The location of this file is compiler dependent, but it usually ends up in the directory where the master program was started or in the user's home directory. You may have to modify this file slightly in two ways. First, rename it to something more meaningful! Second, some printers need to have `showpage` added as the last command in the file, but that confuses some viewers. If you try to print the file and nothing happens, add this line.

## 3.6 Post-solution utilities

### 3.6.1 Store and Restore

There are times when it may be useful to save a PHAML solution and use it in some subsequent program. For example, one might solve a problem on a batch-only system and save the solution, and then later run a program on an interactive system that reads the solution and displays it with PHAML's graphics capability. PHAML provides a routine that writes the entire contents of a `phaml_solution_type` variable to a set of files (one file for the master and one for each slave), and a routine that reads those files into a `phaml_solution_type` variable. The files should be connected to a unit number using `phaml_popen` and `phaml_pclose` as described in section 3.4.1.

`phaml_store` takes two arguments: a `phaml_solution_type` variable and an integer specifying the unit to write to. `phaml_restore` also takes a `phaml_solution_type` variable (which should first be created with `phaml_create`, but otherwise be empty) and a unit to read from. It also takes two optional logical arguments. `do_draw_grid` indicates whether or not you want to invoke graph-

ical output immediately after reading the file, and `pause` indicates whether or not to pause after the graphics. A program that reads files with `phaml_restore` must have the same number of slaves as the program that created the files with `phaml_store`.

### 3.6.2 Query

PHAML provides for the recovery of many quantities of interest through subroutine `phaml_query`, which can be called after returning from `phaml_solve_pde`. This includes information about the grid, such as number of elements, number of vertices, minimum and maximum polynomial degree, etc. Most of them can be for the whole grid or for the grid that each slave has. A second category includes error estimates in various norms, and, if the true solution is provided, the norms of the error. A third category provides interesting quantities related to eigenvalue problems.

For a complete list of the quantities that can be recovered through `phaml_query`, see Section 7.4.11.

### 3.6.3 Solution evaluation

PHAML provides for evaluating the computed solution at a point in the domain through subroutine `phaml_evaluate` (see Section 7.4.6). After returning from `phaml_solve_pde`, you can pass the `phaml_solution_type` variable to `phaml_evaluate` to obtain the solution. You also pass two arrays, `x` and `y`, containing the points at which to evaluate the solution. The solution is returned in the array `soln`, which should have the same dimension as `x` and `y`. If you are solving a system of equations or an eigenvalue problems, there are optional arguments to specify which component or eigenfunction you want to evaluate. If a point outside the domain is given, the solution is returned as 0.

There is also a subroutine to evaluate an “old” solution, which can be used for time dependent and nonlinear problems. See Section 4.3 for information on this routine.

### 3.6.4 Functionals

In this context, a functional of the solution is some form of integral of the computed solution. Subroutine `phaml_integrate` (see Section 7.4.8) computes

$$\iint_{\Omega} k(x, y) u_i^p(x, y) u_j^q(x, y) dx dy.$$

$u_i$  and  $u_j$  are two components of the solution of a system of equations, or two eigenfunctions of an eigenvalue problem, or  $u_i$  is a solution and  $u_j$  is 1.  $p$  and  $q$  are integer powers.  $k$  is a kernel function defined by the user in function `phaml_integral_kernel` (Section 7.3.7). In addition to a point at which to evaluate the kernel, the function takes an integer, `kernel`, which allows for the definition of several kernel functions in the same subroutine.

## 4 Problem Extensions

### 4.1 Eigenvalue Problems

In addition to elliptic boundary value problems, PHAML can solve elliptic eigenvalue problems of the form

$$-\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + r(x,y)u = \lambda f(x,y)u \text{ in } \Omega \quad (9)$$

$$u = 0 \text{ on } \partial\Omega_D \quad (10)$$

$$p(x,y)\frac{\partial u}{\partial x}\frac{\partial y}{\partial s} - q(x,y)\frac{\partial u}{\partial y}\frac{\partial x}{\partial s} + c(x,y)u = 0 \text{ on } \partial\Omega_N \quad (11)$$

Note that the boundary conditions must be homogeneous. Usually the function  $f$  is identically 1. However there are some occasions where other  $f$  are useful. For example, to put the Laplacian operator in polar coordinates into the form of Equation 9, one may multiply the canonical form of the equation by  $x$  (a.k.a.  $r$  in polar coordinates) which results in  $f = x$ .

You indicate that an eigenvalue problem is being solved by specifying `eq_type = EIGENVALUE` in subroutine `phaml_create`. The solution returns eigenfunctions,  $u$ , and the corresponding eigenvalues  $\lambda$ . More than one eigenpair can be computed. The number of desired eigenpairs is specified through the argument `num_eval` to subroutine `phaml_solve_pde`. Subroutines `phaml_evaluate`, `phaml_evaluate_old`, `phaml_integrate`, `phaml_query`, and `phaml_scale` take an optional integer argument, `eigen`, which specifies which eigenfunction to use. Subroutine `phaml_query` returns the entire set of eigenvalues in the `real(my_real)` array argument `eigenvalues`. Printed output prints all of the eigenpairs.

By default PHAML computes the smallest eigenvalue(s). You can compute eigenvalues in the interior of the spectrum by using the argument `lambda0` to `phaml_solve_pde`. PHAML will compute the eigenvalues that are closest to `lambda0`, on both sides of it.

If  $u$  is a solution of Equation 9, then so is  $\alpha u$  for any scalar number  $\alpha$ . Thus the solution must be scaled to meet some condition to be unique. PHAML provides three options for how to scale the eigenfunctions through the argument `scale_vec` to subroutine `phaml_solve_pde`. `SCALE_LINF` requests that the  $\ell^\infty$  norm of the eigenvector,  $x$ , of the discrete problem be 1. With linear elements, this is the same as the  $L^\infty$  norm of the eigenfunction  $u$ . `SCALE_L2` scales such that the  $\ell^2$  norm of  $x$  is 1. `SCALE_M` scales such that the  $M$  norm of the eigenvector,  $\sqrt{x^T M x}$  where  $M$  is the mass matrix, or equivalently the  $L^2$  norm of the eigenfunction, is 1.

PHAML uses ARPACK to solve the discrete eigenproblem, so you must have PHAML configured with the optional ARPACK software (see Section 2.1.7 and 2.2.1). There are three arguments to `phaml_solve_pde` that affect ARPACK's

behavior: `arpack_ncv` sets the number of Lanczos basis vectors, `arpack_maxit` sets the maximum number of IRLM iterations, and `arpack_tol` sets the relative accuracy of the eigenvalues. See the ARPACK User's Guide [12] for a deeper explanation of these arguments.

## 4.2 Coupled Systems or Multicomponent Solutions

PHAML provides for the solution of certain types of coupled systems of elliptic PDEs, either boundary value problems or eigenvalue problems. These problems are also referred to as multicomponent problems if you prefer to think of it as a vector equation, rather than a system of equations, with a multicomponent solution, rather than multiple solutions. For these problems, Equations 1 - 3 are the same except the functions  $p$ ,  $q$  and  $c$  are  $n \times n$  arrays and the functions  $u$ ,  $f$  and  $g$  are vectors of length  $n$ , where  $n$  is the number of equations in the coupled system. An example of subroutine `pdecoefs` for a coupled system is given in Figure 10 for the equations

$$\begin{aligned} -\nabla^2 u + v &= f_1 \\ -\nabla^2 v + u &= f_2 \end{aligned}$$

The number of equations is specified by the argument `system_size` to subroutine `phaml_create`. Subroutines `phaml_evaluate`, `phaml_evaluate_old`, `phaml_integrate`, `phaml_query`, and `phaml_scale` take an optional integer argument `comp` which specifies which component to use.

## 4.3 Parabolic, Nonlinear, Etc. Problems

PHAML can be used to solve parabolic (time dependent) and nonlinear PDEs, but the user has to provide the iteration control in the main program. See the example `examples/parabolic` for an example of solving a time dependent parabolic problem using an implicit finite difference scheme in time, and `examples/nonlinear1` for solving a nonlinear PDE using a simple Picard iteration. Other approaches to handling the time dimension or nonlinearity can be implemented, but they must be single step methods, i.e., must only require one previous solution in the iterative step.

The feature of PHAML that facilitates these problems is the storage of an "old" solution. You create an old solution with subroutine `phaml_copy_soln_to_old` (Section 7.4.3). This copies the current solution component of the `phaml_solution_type` variable to another component called the old solution. This would normally be done at the beginning or end of each iteration. To use the old solution, call `phaml_evaluate_old` (Section 7.4.7). You can evaluate the solution and/or the first derivatives. This would normally be called from subroutine `pdecoefs`.

A time dependent problem requires setting an initial condition to start the solution. Similarly, a nonlinear problem requires an initial guess of the solution. These initial functions are set by calling `phaml_solve_pde` with

```

subroutine pdecoefs(x,y,cxx,cxy,cyy,cx,cy,c,rs)
use phaml
real(my_real), intent(in) :: x,y
real(my_real), intent(out), dimension(:,:) :: cxx,cxy,cyy,cx,cy,c
real(my_real), intent(out), dimension(:) :: rs

cxx(1,1) = 1.0_my_real;   cxx(1,2) = 0.0_my_real
cxx(2,1) = 0.0_my_real;   cxx(2,2) = 1.0_my_real

cyy(1,1) = 1.0_my_real;   cyy(1,2) = 0.0_my_real
cyy(2,1) = 0.0_my_real;   cyy(2,2) = 1.0_my_real

c(1,1) = 0.0_my_real;    c(1,2) = 1.0_my_real
c(2,1) = 1.0_my_real;    c(2,2) = 0.0_my_real

rs(1) = -(2.0_my_real*exp(x-y) - (x+y)**4/8.0_my_real)
rs(2) = -(3.0_my_real*(x+y)**2 - exp(x-y))

cxy=0
cx=0
cy=0
end subroutine pdecoefs

```

Figure 10: pdecoefs for a coupled system.

`task=SET_INITIAL` and an appropriate termination criterion. The solution will be set to be the function in subroutine `iconds` (Section 7.3.5). It should then be copied to the old solution before starting the iteration.

## 5 Examples

Several examples of using PHAML are provided in directory `phaml-x.x.x/examples`. These can be used as tutorials to understand how PHAML works, or as templates for starting the implementation of your application. Depending on your set up, you might have to edit `master.f90` (or `spmd.f90` if you are using the SPMD model) slightly before running them, to change the termination criterion, number of processors, graphics choice, etc. The examples include:

- `simple` – a trivial program that should be the first one you try.
- `elliptic` – contains several linear scalar elliptic boundary value problems selected by a case statement.
- `rectangle` – defines a rectangular domain with an  $N \times M$  grid by writing a Triangle `.poly` file.
- `L-domain` – the classic L shaped domain with a singular solution commonly used for testing adaptive refinement codes.
- `domains` – contains several domains defined by Triangle `.poly` files.
- `curved` – defines a domain with curved boundaries.
- `periodic` – periodic boundary conditions.
- `eigenvalue` – an elliptic eigenvalue problem.
- `parabolic` – solves a time-dependent parabolic PDE by an implicit finite difference scheme in  $t$ .
- `nonlinear1` – solves a nonlinear equation by Picard iteration.
- `system` – solves a coupled system of 2 elliptic PDEs.
- `all` – illustrates the use of all of the `phaml_*` routines.

There are also several programs under the `testdir` directory, but they do not contain comments to help you understand them.

## 6 Release notes

This section is reserved for listing the changes with each new release of PHAML. See also the files `doc/HISTORY` and `doc/UPGRADING`.



## 7 Reference Manual

### 7.1 Quick Start

This section provides a minimum of information about getting PHAML up and running the example programs. For many people, this will be sufficient to get started using PHAML with your application, as many applications may be implemented by modifying one of the examples. If you encounter problems, refer to the appropriate section of the User's Guide for further details. Also read the remainder of the User's Guide for a better understanding of how PHAML works and what options may be useful to you.

The four steps in getting started with PHAML are

1. obtaining the software,
2. compiling the PHAML library,
3. compiling an example, and
4. running the example.

For this brief tutorial, we will build and run the `simple` example as a master/slave message passing MPI program with the master spawning the slave processes, and without graphics.

#### 7.1.1 Obtaining the software

PHAML can be obtained from the PHAML web page <http://math.nist.gov/phaml> by following the Software link. It comes as a gzipped tar file for Unix-like systems. (It has not yet been tested on MS Windows systems, but since it is written in standard Fortran 90, the adventurous may find that it works on MS Windows, too.) When unpacked, it will place everything in a new directory called `phaml-x.x.x`, where `x.x.x` is the current version number.

PHAML requires the BLAS and LAPACK libraries. You will probably find these are already installed on your computer, but if not, see Section 2.1.3.

For parallelism, you need an MPI library. You will probably find that LAM, MPICH, or a commercial MPI library is already installed on your computer, but if not see Section 2.1.4.

#### 7.1.2 Compiling the PHAML library

The first step in compiling the PHAML library is to create the `Makefile`. This is done with the shell script `mkmkfile.sh` in the top PHAML directory. Since PHAML allows so many options in terms of what compilers and libraries to use, it would be difficult to auto-locate these files. So, you must modify `mkmkfile.sh` to specify some paths, command names, and defaults for your computer system. Instructions for modifying it can be found within the file.

Now create the `Makefile` with

```
./mkmkfile.sh PARALLEL messpass_spawn PARLIB mpi GRAPHICS none
```

You should replace `mpi` with `mpich`, `mpich2`, or `lam` if you use an MPICH or LAM library. You may omit some of the arguments if your defaults are already set to these values; defaults can be determined with `mkmkfiles.sh help`.

`make` should now compile the library and place it in the `lib` subdirectory.

### 7.1.3 Compiling an Example

Go to the directory `examples/simple` and type `make`. (The Makefiles for the examples were also created by `mkmkfile.sh`.) This should create the executables `phaml` and `phaml_slave`.

### 7.1.4 Running the Example

The details of running an MPI program vary with the different MPI libraries. You may need to check your MPI documentation to find the correct command(s). It may also require starting some daemon before running the execution command.

Note that you should specify *one* process, because you are running the master processes which will spawn the slave processes. The number of slaves is specified in the main program, `master.f90`.

If you are using LAM, try

```
lamboot
```

```
mpirun -np 1 phaml
```

If you are using MPICH, try

```
mpirun -np 1 phaml
```

If you are using MPICH2, try

```
mpiexec -n 1 phaml
```

### 7.1.5 Now what?

If you have successfully run the first example, you are ready to install the graphics and any other optional software you desire (Section 2.1), run the other examples, and begin working on your own application!

## 7.2 Public Entities in PHAML

The statement `use phaml` in a program unit provides access to the public entities in PHAML. These consist of a derived type, variables for the user to use, symbolic constants, and procedures. The procedures are described in Section 7.4. The other entities are described in this section.

### 7.2.1 `phaml_solution_type`

`phaml_solution_type` is a type that contains all the data used for solving the PDE (grid, etc.). The type is public, but the contents are private. You can

declare one or more variables of this type and pass them to the PHAML procedures.

### 7.2.2 my\_real

`my_real` is a symbolic constant that determines the kind of real numbers used in PHAML. This is defined in `global.f90`. You can change that definition there to select the kind to use, either single or double precision. Other kinds of real, if supported by your Fortran compiler, are not supported by PHAML because of the reliance on BLAS and LAPACK routines, which only come in single and double precision. Currently `my_real` is set for default double precision. To be sure that your program is using the same `kind` values, you should declare your variables as `real(kind=my_real)` and attach the kind to constants, e.g. `0.0_my_real`.

### 7.2.3 pde and my\_pde\_id

`pde` is a rank 1 allocatable array of type `phaml_solution_type`. If you are going to use more than one `pde` and they must communicate with each other, then you must use this array for your `phaml_solution` variables. It must be allocated before using `phaml_create`, and should be deallocated when you are done with it.

`my_pde_id` is an identifier for which `pde` a `phaml_solution` is associated with, usually the index into the array `pde` described above. It is usually used as the case statement variable in the user provided routines that define the PDE and boundary conditions.

See `examples/system.ss` for an example of the use of `pde` and `my_pde_id`. However, having multiple `phaml_solution_type` variables communicate with each other is very slow, so use of this capability is discouraged.

### 7.2.4 symbolic constants

Except for `my_real`, the symbolic constants are used as values for the arguments of the procedures. In this document and in the PHAML source code they are written in capital letters, but Fortran is not case sensitive so this is not necessary. They are defined as they arise in the description of the PHAML procedures. You can find a list of them by looking at the `public` statements at the beginning of `phaml.f90`. If you have a problem with one of the names conflicting with an entity in your code, you can use Fortran's renaming capability to circumvent it, for example

```
use phaml, MY_EVAL_NAME => EIGENVALUE
```

## 7.3 User Provided Routines

The specifics of the problem to be solved are defined through a set of external subroutines that the user must provide. They must be external subroutines (i.e., not module or internal subroutines) because these are called from within

PHAML and are compiled after the PHAML library has been built. They should contain a `use phaml` statement to get access to `my_real` and possibly other entities from PHAML.

This section describes the purpose of each of these routines and defines the interface of each. Also see `pde.f90` in each subdirectory of the PHAML `examples` directory for examples of these routines.

### 7.3.1 bconds

Subroutine `bconds` returns the boundary conditions at a given point. At each point, the boundary conditions are of the form

$$u = g(x, y)$$

or

$$p(x, y) \frac{\partial u}{\partial x} \frac{\partial y}{\partial s} - q(x, y) \frac{\partial u}{\partial y} \frac{\partial x}{\partial s} + c(x, y)u = g(x, y)$$

as described in Section 3.2.2.

```
subroutine bconds(x,y,bmark,itype,c,rs)
```

`real(my_real), intent(in) :: x,y` – the point at which to evaluate the boundary conditions.

`integer, intent(in) :: bmark` – the boundary marker assigned in the Triangle data files.

`integer, intent(out) :: itype(:)` – the type of boundary condition at this point, given by symbolic constants from module `phaml`. It must be one of `DIRICHLET` (the first form above), `NATURAL` (the second form with  $c$  identically 0.0), or `MIXED` (the second form with  $c$  nonzero). The dimension of `itype` is `system_size`. The  $i^{th}$  value is the type of boundary condition for the  $i^{th}$  component.

`real(my_real), intent(out) :: c(:, :)` – the function  $c$  in the boundary condition. The dimension is `system_size` by `system_size`. The  $(i, j)^{th}$  entry is the coefficient of the  $j^{th}$  component in the boundary condition for the  $i^{th}$  component.

`real(my_real), intent(out) :: rs(:)` – the function  $g$  in the boundary condition. The dimension is `system_size`. The  $i^{th}$  entry is the right side of the boundary condition for the  $i^{th}$  component.

### 7.3.2 boundary\_point

Subroutine `boundary_point` defines the boundary of the domain, if it is given by subroutines rather than triangle data files.

```
subroutine boundary_point(ipiece,s,x,y)
```

`integer, intent(in) :: ipiece` – the piece of the boundary from which to return a point.

`real(my_real), intent(in) :: s` – the parameter for the point to be determined.

`real(my_real), intent(out) :: x,y` – the point  $(x(s), y(s))$  on piece `ipiece` of the boundary.

### 7.3.3 `boundary_npiece`

If the boundary of the domain is defined by the boundary subroutines, function `boundary_npiece` returns the number of boundary pieces in the definition. If the domain is defined by triangle data files, it returns 0 or a negative number. If the domain contains holes, it returns the number of pieces in the requested hole.

```
function boundary_npiece(hole)
```

`integer, intent(in) :: hole` – the hole for which to return the number of pieces, or 0 for the outer boundary. Holes are numbered consecutively starting with 1.

`integer :: boundary_npiece` – the number of boundary pieces in the requested hole or in the outer boundary if `hole` is 0.

### 7.3.4 `boundary_param`

Subroutine `boundary_param` gives the range of parameter values for each boundary piece.

```
subroutine boundary_param(start,finish)
```

`real(my_real), intent(out) :: start(:), finish(:)` – `start(i)` and `finish(i)` are the beginning and ending parameter values for the  $i^{th}$  piece of the boundary,  $i=1, \text{number of boundary pieces}$ .

### 7.3.5 `iconds`

Subroutine `iconds` is used for setting the solution to a given function. For example, it can be used for initial conditions for a time dependent problem or the initial guess for a nonlinear problem.

```
function iconds(x,y,comp,eigen)
```

`real(my_real), intent(in) :: x,y` – the point at which to evaluate the function.

`integer, intent(in) :: comp` – for multicomponent solutions, which component to return.

`integer, intent(in) :: eigen` – for eigenvalue problems in which more than one eigenvalue is computed, which eigenfunction to return.

`real(my_real) :: iconds` – the function value.

### 7.3.6 pdecoefs

Subroutine `pdecoefs` returns the values of the PDE coefficient functions and right side. The PDE is given by

$$-\frac{\partial}{\partial x}(p(x,y)\frac{\partial u}{\partial x}) - \frac{\partial}{\partial y}(q(x,y)\frac{\partial u}{\partial y}) + r(x,y)u = f(x,y) \text{ in } \Omega \quad (12)$$

as described in Section 3.2.1.

`subroutine pdecoefs(x,y,cxx,cxy,cyy,cx,cy,c,rs)`

`real(my_real), intent(in) :: x,y` – the point at which to evaluate the PDE.

`real(my_real), intent(out) :: cxx(:,:),cyy(:,:),c(:,:)` – the functions  $p$ ,  $q$  and  $r$  respectively. The dimension is `system_size` by `system_size`. The  $(i,j)^{th}$  entry is the coefficient of the  $j^{th}$  component in the PDE for the  $i^{th}$  component.

`real(my_real), intent(out) :: cxy(:,:),cx(:,:),cy(:,:)` – currently not used. They are included for a possible future extension. They should be set to 0 to avoid possible problems if they are used later.

`real(my_real), intent(out) :: rs(:)` – the function  $f$ . The dimension is `system_size`. The  $i^{th}$  value is the right side of the PDE for the  $i^{th}$  component.

### 7.3.7 phaml\_integral\_kernel

Subroutine `phaml_integrate` described in Sections 3.6.4 and 7.4.8 computes functionals of the computed solution of the form

$$\iint_{\Omega} k(x,y) u_i^p(x,y) u_j^q(x,y) dx dy$$

where  $k$  is a kernel function defined in `phaml_integral_kernel`. In addition to a point at which to evaluate the kernel, the function takes an integer, `kernel`, which allows for the definition of several kernel functions in the same subroutine.

`function phaml_integral_kernel(kernel,x,y)`

`integer, intent(in) :: kernel` – an integer passed through `phaml_integrate` to allow a choice of different kernels.

`real(my_real), intent(in) :: x,y` – the point at which to evaluate the kernel.

`real(my_real) :: phaml_integral_kernel` – the kernel value,  $k(x,y)$ .

### 7.3.8 trues

If the true solution of the PDE is known, you can provide it in function `trues`. This is used for printing norms of the error and for graphical display of the error and true solution. If you do not know the true solution, returning `huge(0.0_my_real)` will prevent printing and plotting of a bogus error.

```
function trues(x,y,comp,eigen)
```

```
real(my_real), intent(in) :: x,y – the point at which to evaluate the  
true solution
```

```
integer, intent(in) :: comp – for multicomponent solutions, which com-  
ponent to return.
```

```
integer, intent(in) :: eigen – for eigenvalue problems in which more  
than one eigenvalue is computed, which eigenfunction to return.
```

```
real(my_real) :: trues – the return value.
```

### 7.3.9 truexs

If the true solution of the PDE is known, you can provide the  $x$  derivative of it in function `truexs`. This is used for printing the energy norm of the error. If `trues` does not return `huge(0.0_my_real)` and you request the energy norm of the error, then you must provide the  $x$  derivative of the true solution in `truexs` or an incorrect value of the energy norm of the error will be printed.

```
function truexs(x,y,comp,eigen)
```

```
real(my_real), intent(in) :: x,y – the point at which to evaluate the  $x$   
derivative of the true solution
```

```
integer, intent(in) :: comp – for multicomponent solutions, which com-  
ponent to return.
```

```
integer, intent(in) :: eigen – for eigenvalue problems in which more  
than one eigenvalue is computed, which eigenfunction to return.
```

```
real(my_real) :: truexs – the return value.
```

### 7.3.10 trueys

If the true solution of the PDE is known, you can provide the  $y$  derivative of it in function `trueys`. This is used for printing the energy norm of the error. If `trues` does not return `huge(0.0_my_real)` and you request the energy norm of the error, then you must provide the  $y$  derivative of the true solution in `trueys` or an incorrect value of the energy norm of the error will be printed.

```
function trueys(x,y,comp,eigen)
```

```
real(my_real), intent(in) :: x,y – the point at which to evaluate the  $y$   
derivative of the true solution
```

`integer, intent(in) :: comp` – for multicomponent solutions, which component to return.

`integer, intent(in) :: eigen` – for eigenvalue problems in which more than one eigenvalue is computed, which eigenfunction to return.

`real(my_real) :: trueys` – the return value.

### 7.3.11 `update_usermod`

The application can include a module called `phaml_user_mod` to provide access to global variables in the user provided subroutines. However, values assigned to these variables in the main program are only available in the master process. Subroutine `update_usermod` passes the current value of these variables from the master to the slaves. For an example use of this facility, see `examples/parabolic` where it is used to pass the time step to the slaves. A working version of this subroutine is shown in Figure 11, or can be copied from `examples/parabolic/pde.f90`. In the example, the `integer` variable `ivar` and the `real(my_real)` variables `rvar1` and `rvar2` are declared in module `phaml_user_mod`. If you do not use this facility, you can copy a dummy version of the subroutine from `examples/simple/pde.f90`.

## 7.4 PHAML procedures

### 7.4.1 `phaml_compress`

`phaml_compress` compresses the `phaml_solution_type` data by moving all the unused elements, edges and vertices after the used ones. If considerable derefinement has occurred, this can significantly reduce the size of files created by `phaml_store`.

```
subroutine phaml_compress(phaml_solution)
```

`type(phaml_solution_type), intent(inout) :: phaml_solution` – the solution to compress.

### 7.4.2 `phaml_connect`

`phaml_connect` connects two `phaml_solution_type` variables so they can communicate. If one will request that another evaluate its solution, then they must be connected. See `examples/system_ss` and Section 7.2.3. However, having multiple `phaml_solution_type` variables communicate with each other is very slow, so use of this capability is discouraged.

```
subroutine phaml_connect(pde1, pde2)
```

`integer, intent(in) :: pde1, pde2` – indices into the pde array in module `phaml`.



```

subroutine update_usermod(phaml_solution)

!-----
! This routine updates the module variables on the slave processes
! by sending them from the master process
!-----

use phaml
use phaml_user_mod
!-----
! Dummy arguments

type(phaml_solution_type), intent(in) :: phaml_solution
!-----
! Local variables:

! Declare these arrays big enough to hold the variables to be sent

integer :: iparam(1)
real(my_real) :: rparam(2)
!-----
! Begin executable code

! Copy the module variables into the arrays, putting integer
! variables into iparam and real variables into rparam.

    iparam(1) = ivar
    rparam(1) = rvar1
    rparam(2) = rvar2

! Call the routine that performs the actual exchange. Don't change
! this line.

    call master_to_slaves(phaml_solution,iparam,rparam)

! Copy the arrays into the module variables, using the same
! correspondence between module variable and array index as
! was used above.

    ivar = iparam(1)
    rvar1 = rparam(1)
    rvar2 = rparam(2)

end subroutine update_usermod

```

Figure 11: An example of subroutine update\_usermod.

### 7.4.3 phaml\_copy\_soln\_to\_old

`phaml_copy_soln_to_old` makes a copy of the solution in `phaml_solution` which can be evaluated by `phaml_evaluate_old` (Section 7.4.7). The main purpose of this is to provide the “old” solution and derivatives for time stepping in time dependent problems and iterating in nonlinear problems.

```
subroutine phaml_copy_soln_to_old(phaml_solution)
```

```
type(phaml_solution_type), intent(inout) :: phaml_solution – the  
    phaml solution in which to make a copy of the solution.
```

### 7.4.4 phaml\_create

`phaml_create` creates a variable to contain a `phaml_solution`, i.e., it initializes the internals of a variable of type `phaml_solution_type`. It also spawns the associated slave and graphics processes. This should be called with each variable of type `phaml_solution_type` in your program.

All arguments except `phaml_solution` are optional.

```
subroutine phaml_create(phaml_solution, nproc, draw_grid_who,  
spawn_form, debug_command, display, graphics_host, output_unit,  
error_unit, output_now, id, system_size, eq_type, max_blen,  
triangle_files)
```

```
type (phaml_solution_type) phaml_solution – the variable to initialize.
```

```
integer nproc – the number of processes working in parallel. The default is  
    1.
```

`draw_grid_who` – which processes should display grid graphics. The grid graphics are not just the grid, but also the computed solution and many other useful displays. Valid values are:

**MASTER** – the graphics process associated with the master process draws a grid that is a consolidation of the slave’s grids.

**SLAVES** – each slave has an associated graphics process, which draws the grid as known to that slave. Note that there will be a graphics window for each slave.

**EVERYONE** – both **MASTER** and **SLAVES**.

**NO\_ONE** – no graphics are drawn.

The default is **NO\_ONE**.

`spawn_form` – whether or not to provide debugging capability with spawned processes. For MPI, an xterm running the slave under the debugger given by `debug_command` is brought up. For PVM the debugger is determined by the PVM environment (see the file `doc/HINTS`). Valid values are:

`NORMAL_SPAWN` – no debugging.

`DEBUG_SLAVE` – spawn the slave processes under the debugger.

`DEBUG_GRAPHICS` – spawn the graphics processes under the debugger.

`DEBUG_BOTH` – spawn both slaves and graphics under the debugger.

The default is `NORMAL_SPAWN`.

`character(len=*) debug_command` – the command name of the debugger to use in an `xterm` when `spawn_form` is not `NORMAL_SPAWN` and MPI is used. Limited to 64 characters. The default is `"gdb"`.

`character(len=*) display` – in some environments, the `-display` argument is needed for `xterm` when `spawn_form` is not `NORMAL_SPAWN`. This value is used for that argument. If it is `"default"` then the `-display` argument is not used for `xterm`. Limited to 64 characters. The default is `"default"`.

`character(len=*) graphics_host` – the name of the host on which to spawn the graphics processes. This is useful if you want to force the graphics processes to run on the workstation whose display is in front of you, or if you want graphics to run on a particular architecture. PVM will obey this request. For MPI, this is a “hint” which may be obeyed or ignored depending on the implementation of MPI. LAM, up to version 7.1.1 at least, ignores the request.

`integer output_unit` – the unit number for printed output. It should either be a pre-connected unit (e.g. standard output) or be opened as `FORMATTED` with a call to `phaml_popen` immediately after `call phaml_create`. The default is 6.

`integer error_unit` – the unit number for error messages. It should either be a pre-connected unit (e.g. standard error) or be opened as `FORMATTED` with a call to `phaml_popen` immediately after `call phaml_create`. The default is 0.

`integer output_now` – since `output_unit` and `error_unit` may not be available until after `phaml_create` is complete, a unit must be provided for any output (i.e. error messages) that occurs from subroutine `phaml_create`. This should be a pre-connected unit. The default is 6.

`integer id` – an identifier for the PDE, simply for use by the user in subroutines `pdecoefs`, `bconds`, etc. It is available as `my_pde_id` in module `phaml`. See Section 7.2.3. The default is 0.

`integer system_size` – number of equations in a coupled system of PDEs, or equivalently, number of components in a multicomponent solution. The default is 1.

`integer eq_type` – type of equation to create. Valid values are:

ELLIPTIC – solve an elliptic boundary value PDE.

EIGENVALUE – solve an elliptic eigenvalue PDE.

The default is ELLIPTIC.

real(my\_real) max\_blen – maximum length of a boundary segment if the domain is defined by the boundary subroutines.

The default is  $\infty$ , defined to be huge(0.0\_my\_real).

character(len=\*) triangle\_files – the root name of the .node, .ele, .neigh, .poly and .edge triangle data files. If the domain is defined by triangle data files and the file names include Triangle's iteration number (usually .1), include the iteration number. If the domain is defined by the boundary subroutines, this is used as the root of the created triangle data files.

The default is "domain".

logical update\_umod – run update\_usermod as soon as the communication has been initialized. This is needed if phaml\_user\_mod has any parameters that affect the definition of the domain, or need to be set before the first solution on the initial grid.

The default is .false.

#### 7.4.5 phaml\_destroy

phaml\_destroy destroys a variable of type phaml\_solution\_type, i.e., frees the memory contained in it. It also terminates the slave and graphics processes. If you have multiple phaml\_solution\_type variables that are not needed simultaneously, you should destroy those no longer needed before creating those not yet used, to reduce the possibility of running out of memory. Also, if a phaml\_solution\_type variable is used more than once and is to be recreated to give it a fresh start, it should be destroyed before being created again to avoid a memory leak.

```
subroutine phaml_destroy(phaml_solution, finalize_mpi)
```

type (phaml\_solution\_type) phaml\_solution – the variable to destroy

logical finalize\_mpi – if .false., do not call mpi\_finalize from the master. It should be .true. if and only if this is the last call to phaml\_destroy.

The default is .true.

#### 7.4.6 phaml\_evaluate

phaml\_evaluate evaluates the computed solution at the given points.

```
subroutine phaml_evaluate(phaml_solution, x, y, soln, comp,  
eigen)
```

`type(phaml_solution_type), intent(in) :: phaml_solution` – the solution to evaluate.

`real(my_real), intent(in) :: x(:), y(:)` – array of x and y coordinates at which to evaluate the solution.

`real(my_real), intent(out) :: soln(:)` – returned as the computed solution at the given points. Must have `size(soln) == size(x)`.

`integer, intent(in), optional :: comp` – for multicomponent solutions, which component to evaluate. The default is 1.

`integer, intent(in), optional :: eigen` – for eigenvalue problems, which eigenfunction to evaluate. The default is 1.

#### 7.4.7 `phaml_evaluate_old`

`phaml_evaluate_old` evaluates the “old” solution and/or derivatives at the given point(s). Unlike all the other `phaml` subroutines, it should not be called from the main program. It is intended to be called from the user routines that define the problem (`pdecoefs`, etc.) to provide the “old” solution and derivatives for time stepping in time dependent problems and iterating in nonlinear problems.

subroutine `phaml_evaluate_old(x, y, u, ux, uy, comp, eigen)`

`real(my_real), intent(in) :: x, y` – x and y coordinates of the point at which to evaluate the solution.

`real(my_real), intent(out), optional :: u, ux, uy` – returned as the old solution, x derivative and y derivative at the given point. Any combination of the three may be present.

`integer, intent(in), optional :: comp` – for multicomponent solutions, which component to evaluate. The default is 1.

`integer, intent(in), optional :: eigen` – for eigenvalue problems, which eigenfunction to evaluate. The default is 1.

#### 7.4.8 `phaml_integrate`

`phaml_integrate` returns a functional of the computed solution, i.e. an integral of a computed solution or product of two computed solutions or powers of computed solutions, weighted by a kernel function. It computes

$$\iint_{\Omega} k(x, y) u_{comp1, eigen1}^p(x, y) u_{comp2, eigen2}^q(x, y) dx dy$$

where  $k$  is a kernel function defined in `phaml_integral_kernel` (see Section 7.3.7).

```
function phaml_integrate(phaml_solution, kernel, comp1, eigen1,
comp2, eigen2, p, q)
```

`type(phaml_solution_type), intent(in) :: phaml_solution` – the solution to integrate.

`integer, intent(in) :: kernel` – allows you to select among different kernel functions. It is passed to `phaml_integral_kernel` where it can be used, for example, in a case statement to determine the kernel to use.

`integer, intent(in), optional :: comp1, eigen1, comp2, eigen2` – which component(s) of a multicomponent solution and which eigenfunction(s) of an eigenvalue problem to use. All of them default to 1. If `comp1` is omitted then `comp2` must also be omitted, and likewise for `eigen1` and `eigen2`. If `comp2` and `eigen2` are both omitted, then  $u_{comp2,eigen2}$  is omitted from the integral.

`integer, intent(in), optional :: p, q` – the powers to which to raise the computed solutions in the integral. The default is 1.

#### 7.4.9 `phaml_pclose`

`phaml_pclose` closes unit number `unit` on all processors in `phaml_solution`. See also `phaml_popen` in Section 7.4.10.

```
subroutine phaml_pclose(phaml_solution, unit)
```

`type (phaml_solution_type), intent(in) :: phaml_solution` – the solution for which the files are being closed.

`integer, intent(in) :: unit` – the unit number to close.

#### 7.4.10 `phaml_popen`

`phaml_popen` opens unit number `unit` on all processors in `phaml_solution`. If `file` is of the form “root.suffix” then the actual filenames are `rootXXXX.suffix` for processor number `XXXX`, where the number of digits in `XXXX` is the minimum needed for the number of processors in `phaml_solution`. The master processor is number 0. If there is no “.” in `file`, then there is no suffix and `XXXX` is appended to the filename. The filename is limited to 128 characters and the number of processors is limited to 9999. If the filename does not contain the full path, the location of the file is compiler dependent. Good places to look for it are the current working directory and the user’s home directory.

```
subroutine phaml_popen(phaml_solution, unit, file, form)
```

`type (phaml_solution_type), intent(in) :: phaml_solution` – the solution for which the files are being opened.

`integer, intent(in) :: unit` – unit number to open.

`character(len=*) , intent(in) :: file` – base file name for the files.

`character(len=*) , intent(in), optional :: form` – must take the value "FORMATTED" or "UNFORMATTED" to determine the type of file to open. Default is "FORMATTED".

#### 7.4.11 `phaml_query`

`phaml_query` returns values of interest about the grid and solution. All arguments except `phaml_solution`, `comp` and `eigen` are optional and `intent(out)`. Include those you wish to receive as keyword arguments. `phaml_solution` is mandatory and `intent(in)`. `comp` and `eigen` are optional and `intent(in)`.

```
subroutine phaml_query(phaml_solution, nvert, nvert_proc,
nvert_own, nelem, nelem_proc, nelem_own, neq, neq_proc, neq_own,
nlev, min_degree, max_degree, linf_error, energy_error, l2_error,
max_h_error_indicator, max_p_error_indicator, linf_h_error_estimate,
energy_h_error_estimate, l2_h_error_estimate, linf_p_error_estimate,
energy_p_error_estimate, l2_p_error_estimate, linf_solution,
l2_solution, eigenvalues, max_linsys_resid, ave_linsys_resid,
eigen_l2_resid, arpack_iter, arpack_nconv, arpack_numop,
arpack_numopb, arpack_numreo, arpack_info, comp, eigen)
```

`type(phaml_solution_type) :: phaml_solution` – the solution to query

`integer :: nvert` – the number of vertices in the grid

`integer, dimension(nproc) :: nvert_proc` – the number of vertices in the grid of each processor

`integer, dimension(nproc) :: nvert_own` – the number of vertices owned by each processor

`integer :: nelem` – the number of elements in the grid

`integer, dimension(nproc) :: nelem_proc` – the number of elements in the grid of each processor

`integer, dimension(nproc) :: nelem_own` – the number of elements owned by each processor

`integer :: neq` – the number of equations in the linear system

`integer, dimension(nproc) :: neq_proc` – the number of equations in the linear system of each processor

**integer, dimension(nproc) :: neq\_own** – the number of equations owned by each processor  
**integer :: nlev** – the number of levels of refinement  
**integer :: min\_degree** – the smallest polynomial degree of an element  
**integer :: max\_degree** – the largest polynomial degree of an element  
**real(my\_real) :: linf\_error** – the  $L^\infty$  norm of the error (at the vertices and quadrature points of a sixth order quadrature rule), if the true solution is known  
**real(my\_real) :: energy\_error** – the energy norm of the error (approximated by a sixth order quadrature rule), if the true solution and its derivatives are known  
**real(my\_real) :: l2\_error** – the  $L^2$  norm of the error (approximated by a sixth order quadrature rule), if the true solution is known  
**real(my\_real) :: max\_h\_error\_indicator** – the largest h error indicator  
**real(my\_real) :: max\_p\_error\_indicator** – the largest p error indicator  
**real(my\_real) :: linf\_h\_error\_estimate** – an estimate of the  $L^\infty$  norm of the error based on h error indicators  
**real(my\_real) :: energy\_h\_error\_estimate** – an estimate of the energy norm of the error based on h error indicators  
**real(my\_real) :: l2\_h\_error\_estimate** – an estimate of the  $L^2$  norm of the error based on h error indicators  
**real(my\_real) :: linf\_p\_error\_estimate** – an estimate of the  $L^\infty$  norm of the error based on p error indicators  
**real(my\_real) :: energy\_p\_error\_estimate** – an estimate of the energy norm of the error based on p error indicators  
**real(my\_real) :: l2\_p\_error\_estimate** – an estimate of the  $L^2$  norm of the error based on p error indicators  
**real(my\_real) :: linf\_solution** – the discrete  $\ell^\infty$  norm of the solution vector  
**real(my\_real) :: l2\_solution** – the discrete  $\ell^2$  norm of the solution vector  
**real(my\_real) :: energy\_solution** – the discrete energy norm of the solution vector  
**real(my\_real) :: linf\_u** – the continuous  $L^\infty$  norm of the solution, approximated using the vertices and the quadrature points of a sixth order quadrature rule



`real(my_real) :: l2_u` – the continuous  $L^2$  norm of the solution, approximated with a sixth order quadrature rule  
`real(my_real) :: energy_u` – the continuous energy norm of the solution, approximated with a sixth order quadrature rule  
`real(my_real), dimension(num_eval) :: eigenvalues` – the computed eigenvalues, for eigenvalue problems  
`real(my_real) :: max_linsys_resid` – for eigenvalue problems, the largest  $\ell^2$  norm of the residual of the linear systems solved (scaled by the norm of the right hand side)  
`real(my_real) :: ave_linsys_resid` – for eigenvalue problems, the average of the  $\ell^2$  norm of the residuals of the linear systems solved (scaled by the norm of the right hand side)  
`real(my_real), dimension(num_eval) :: eigen_l2_resid` – the  $\ell^2$  norm of the residual of the eigensystem, for each eigenvalue  $\|Ax - \lambda Mx\|/\|\lambda Mx\|$   
`integer :: arpack_iter` – from ARPACK, number of iterations used  
`integer :: arpack_nconv` – from ARPACK, number of converged Ritz values  
`integer :: arpack_numop` – from ARPACK, number of OP\*x operations  
`integer :: arpack_numopb` – from ARPACK, number of B\*x operations  
`integer :: arpack_numreo` – from ARPACK, number of reorthogonalizations  
`integer :: arpack_info` – from ARPACK, info (error flag)  
`integer :: comp` – for multicomponent solutions, which component to report. Energy norms cover all components in a single norm.  $L^\infty$  and  $L^2$  norms cover each component in individual norms. The default is 1.  
`integer :: eigen` – for eigenvalue problems, which eigenfunction to report. The default is 1.

#### 7.4.12 phaml\_restore

`phaml_restore` restores information for `phaml_solution` from files created by subroutine `phaml_store` (see Section 7.4.15).

```

subroutine phaml_restore(phaml_solution, unit, do_draw_grid,
pause)

```

`type (phaml_solution_type), intent(inout) :: phaml_solution` – the solution into which the data will be read. If `phaml_solution` was previously used, it should be destroyed first with `phaml_destroy` (see Section 7.4.5) to avoid a memory leak. Whether or not it was previously used, it must be created with `phaml_create` (see Section 7.4.4) before passing it to `phaml_restore`.

`integer, intent(in) :: unit` – the unit number to read from, which should have been opened with `phaml_popen` using the same form as was used when `phaml_store` created the files.

`logical, intent(in), optional :: do_draw_grid` – whether or not to draw the solution immediately after reading it.

`logical, intent(in), optional :: pause` – if `do_draw_grid` is true, whether or not to pause after drawing the grid.

#### 7.4.13 `phaml_scale`

`phaml_scale` scales the computed solution by multiplying by `factor`.

```
subroutine phaml_scale(phaml_solution, factor, comp, eigen)
```

`type(phaml_solution_type), intent(inout) :: phaml_solution` – the solution to scale.

`real(my_real), intent(in) :: factor` – the factor by which to multiply.

`integer, intent(in), optional :: comp` – for multicomponent solutions, which component to scale. The default is 1.

`integer, intent(in), optional :: eigen` – for eigenvalue problems, which eigenfunction to scale. The default is 1.

#### 7.4.14 `phaml_solve_pde`

`phaml_solve_pde` solves the PDE. All arguments are optional, except `phaml_solution`, so you only need to provide those for which you do not want to take the default. All arguments are `intent(in)` except `phaml_solution` which is `intent(inout)` and `iterm` which is `intent(out)`. It is recommended that the call use keyword arguments for all arguments except possibly `phaml_solution`.

```
subroutine phaml_solve_pde(phaml_solution, iterm, max_elem,  
max_vert, max_eq, max_lev, max_deg, max_refsolve_loop,  
term_h_energy_err, term_p_energy_err, term_h_Linf_err, term_p_Linf_err,  
term_h_L2_err, term_p_L2_err, task, print_grid_when, print_grid_who,  
print_error_when, print_error_who, print_error_what, print_errest_what,  
print_linsys_when, print_linsys_who, print_time_when, print_time_who,  
print_eval_when, print_eval_who, print_header_who, print_trailer_who,
```

```

print_warnings, clocks, draw_grid_when, pause_after_draw,
pause_after_phases, pause_at_start, pause_at_end, sequential_vert,
inc_factor, error_estimator, reftype, refterm, reftol, derefine,
partition_method, prebalance, postbalance, petsc_matrix_free, solver,
preconditioner, mg_cycles, mg_tol, mg_prerelax, mg_postrelax,
mg_prerelax_ho, mg_postrelax_ho, dd_iterations, ignore_quad_err,
num_eval, lambda0, scale_evec, arpack_ncv, arpack_maxit, arpack_tol,
degree, inc_quad_order, hypre_BoomerAMG_MaxLevels,
hypre_BoomerAMG_MaxIter, hypre_BoomerAMG_Tol,
hypre_BoomerAMG_StrongThreshold, hypre_BoomerAMG_MaxRowSum,
hypre_BoomerAMG_CoarsenType, hypre_BoomerAMG_MeasureType,
hypre_BoomerAMG_CycleType, hypre_BoomerAMG_NumGridSweeps,
hypre_BoomerAMG_GridRelaxType, hypre_BoomerAMG_GridRelaxPoints,
hypre_BoomerAMG_RelaxWeight, hypre_BoomerAMG_IOutDat,
hypre_BoomerAMG_DebugFlag, hypre_ParaSails_thresh,
hypre_ParaSails_nlevels, hypre_ParaSails_filter, hypre_ParaSails_sym,
hypre_ParaSails_loadbal, hypre_ParaSails_reuse,
hypre_ParaSails_logging, hypre_PCG_Tol, hypre_PCG_MaxIter,
hypre_PCG_TwoNorm, hypre_PCG_RelChange, hypre_PCG_Logging,
hypre_GMRES_KDim, hypre_GMRES_Tol, hypre_GMRES_MaxIter,
hypre_GMRES_Logging, petsc_richardson_damping_factor,
petsc_chebychev_emin, petsc_chebychev_emax, petsc_gmres_max_steps,
petsc_rtol, petsc_atol, petsc_dtol, petsc_maxits, petsc_ilu_levels,
petsc_icc_levels, petsc_ilu_dt, petsc_ilu_dtol, petsc_ilu_maxrowcount,
petsc_sor_omega, petsc_sor_its, petsc_sor_lits,
petsc_eisenstat_nodiagscaling, petsc_eisenstat_omega,
petsc_asm_overlap, coarse_size, coarse_method)

```

`type (phaml_solution_type) phaml_solution` – the variable that contains the main data structures. It must be created by subroutine `phaml_create` before passing it to `phaml_solve_pde`.

`integer item` – termination code. If positive, a termination criterion was met. If negative, an error occurred. For the current meaning of the codes, see `global.f90` and look for the string `termination`.

`integer max_elem` – the maximum number of elements to use (termination criterion).

The default is  $\infty$ , defined to be `huge(0)`.

`integer max_vert` – maximum number of vertices to use (termination criterion).

The default is  $\infty$ , defined to be `huge(0)`.

`integer max_eq` – maximum number of equations in the linear system (a.k.a. degrees of freedom) to use (termination criterion).

The default is  $\infty$ , defined to be `huge(0)`.

`integer max_lev` – the maximum number of  $h$  refinement levels to use. If an element is flagged for  $h$  refinement and the number of levels would exceed `max_lev`, then the element is quietly not refined. The number of levels is limited by the size of the hash keys, which is set at the time the PHAML library is compiled (see Section 2.2.1). For 32 bit integers it cannot exceed 30 for `PHAML_HASHSIZE=1` and 60 for `PHAML_HASHSIZE=2`. If refinement would exceed this limit and `max_lev` is larger than the limit, the element is quietly not refined.

The default is  $\infty$ , defined to be `huge(0)`.

`integer max_deg` – the maximum polynomial degree for the approximation space. If an element is flagged for  $p$  refinement and the degree would exceed `max_deg`, then the element is quietly not refined.

The default is 10, which corresponds to the maximum degree for which the currently implemented quadrature rules will give the exact solution if the solution is a polynomial of degree `max_deg`.

`integer max_refsolve_loop` – number of times to go through the refine/solve loop (termination criteria).

The default is  $\infty$ , defined to be `huge(0)`.

`real(my_real) term_h_energy_err` – terminate when the  $h$  energy error estimate is less than this value (termination criteria).

The default is 0.0.

`real(my_real) term_p_energy_err` – terminate when the  $p$  energy error estimate is less than this value (termination criteria).

The default is 0.0.

`real(my_real) term_h_Linf_err` – terminate when the  $h$   $L^\infty$  error estimate is less than this value (termination criteria).

The default is 0.0.

`real(my_real) term_p_Linf_err` – terminate when the  $p$   $L^\infty$  error estimate is less than this value (termination criteria).

The default is 0.0.

`real(my_real) term_h_L2_err` – terminate when the  $h$   $L^2$  error estimate is less than this value (termination criteria).

The default is 0.0.

`real(my_real) term_p_L2_err` – terminate when the  $p$   $L^2$  error estimate is less than this value (termination criteria).

The default is 0.0.

`integer task` – what task to perform. Valid values are:

`BALANCE_REFINE_SOLVE` – go through a loop repeatedly doing load balance, refinement and solution phases.

`SET_INITIAL` – loop through the phases, but for the solution phase use interpolation of the function in `iconds`.

`BALANCE_ONLY` – just do one load balancing phase.

`REFINE_ONLY` – just do one refinement phase.

`SOLVE_ONLY` – just do one solution phase.

The default is `BALANCE_REFINE_SOLVE`.

`integer print_grid_when` – how often to produce a printed summary of the grid (number of vertices and elements, etc.). Valid values are:

`NEVER` – don't print.

`PHASES` – after each refinement phase.

`FINAL` – only at the end.

`FREQUENTLY` – possibly more often than `PHASES` (used for debugging).

The default is `NEVER`.

`integer print_grid_who` – which processes should print the summary of the grid. Valid values are:

`NO_ONE` – don't print.

`SLAVES` – slave processes print a summary of the grid as they know it.

`MASTER` – master process prints the composite grid.

`EVERYONE` – both `SLAVES` and `MASTER`.

`MASTER_ALL` – master prints the individual grids of each slave process.

The default is `NO_ONE`.

`integer print_linsys_when` – how often to produce a printed summary of the linear system (number of equations, sparsity, etc.). Valid values are:

`NEVER` – don't print.

`PHASES` – after each linear system solution phase.

`FREQUENTLY` – possibly more often than `PHASES` (used for debugging).

The default is `NEVER`.

`integer print_linsys_who` – which processes should print the summary of the linear system. Valid values are:

`NO_ONE` – don't print.

`SLAVES` – slave processes print a summary of the system as they know it.

**MASTER** – master process prints the composite linear system.  
**EVERYONE** – both **SLAVES** and **MASTER**.  
**MASTER\_ALL** – master prints the individual systems of each slave process.

The default is **NO\_ONE**.

**integer print\_error\_when** – how often to print the norms of the error (defined to be the difference between the computed solution and the function defined in function **true**) and error estimates. What norms of the error and error estimates are printed is determined by **print\_error\_what** and **print\_errest\_what**. It also prints the factor by which they have been reduced since the last time they were printed, and the effectivity index of the error estimate (ratio of the error estimate to the error) if both the error and error estimate are printed. If **true** returns **huge(0.0\_my\_real)** to indicate that the true solution is not known, then the error is not printed. If **true\_x** or **true\_y** return **huge(0.0\_my\_real)** and **true** does not, then the energy norm of the error (if requested) is printed as **huge(0.0\_my\_real)**. Valid values are:

**NEVER** – don't print.  
**PHASES** – after each solution phase.  
**FINAL** – only at the end.  
**FREQUENTLY** – possibly more often than **PHASES** (used for debugging).  
With the hierarchical basis multigrid solver and the PETSc solvers, this causes an estimate of the  $\ell^2$  norm of the residual to be printed after each iteration. See **KSPDefaultMonitor** in the PETSc documentation.  
**TOO\_MUCH** – possibly more often than **FREQUENTLY**. Also this sets the solution to 0.0 before the solution phase as well as printing the residual after each iteration of some solvers as with **FREQUENTLY**.

The default is **NEVER**.

**integer print\_error\_who** – which processes should print the error. Individual processor energy norms of the error cannot be computed, so the energy norm of the error is printed only by the master. Valid values are:

**NO\_ONE** – don't print.  
**SLAVES** – slave processes print the error over the grid as they know it.  
**MASTER** – master process prints the error over the composite grid.  
**EVERYONE** – both **SLAVES** and **MASTER**.  
**MASTER\_ALL** – master prints the individual errors of each slave process.

The default is **NO\_ONE**.

`integer print_error_what` – what norms of the error to print, selected from energy,  $L^\infty$ , and  $L^2$ . The  $L^\infty$  norm of the error is an approximation given by the maximum error at the vertices and the quadrature points of a sixth order quadrature rule. The  $L^2$  norm and energy norm are approximated using a sixth order quadrature rule over the triangles of the grid. Valid values are:

NEVER – don't print any.

ENERGY\_ERR – energy norm.

LINF\_ERR –  $L^\infty$  norm.

L2\_ERR –  $L^2$  norm.

ENERGY\_LINF\_ERR – energy and  $L^\infty$  norms.

ENERGY\_L2\_ERR – energy and  $L^2$  norms.

LINF\_L2\_ERR –  $L^\infty$  and  $L^2$  norms.

ENERGY\_LINF\_L2\_ERR – all three norms.

The default is NEVER.

`integer print_errest_what` – what norms of the error estimate to print, selected from energy,  $L^\infty$ , and  $L^2$ . Valid values are:

NEVER – don't print any.

ENERGY\_HERREST – energy norm based on error indicators for  $h$  refinement.

LINF\_HERREST –  $L^\infty$  norm based on error indicators for  $h$  refinement.

L2\_HERREST –  $L^2$  norm based on error indicators for  $h$  refinement.

ENERGY\_LINF\_HERREST – energy and  $L^\infty$  norms based on error indicators for  $h$  refinement.

ENERGY\_L2\_HERREST – energy and  $L^2$  norms based on error indicators for  $h$  refinement.

LINF\_L2\_HERREST –  $L^\infty$  and  $L^2$  norms based on error indicators for  $h$  refinement.

ENERGY\_LINF\_L2\_HERREST – all three norms based on error indicators for  $h$  refinement.

Also the same forms with PERREST for norms based on error indicators for  $p$  refinement, and the same forms with HPERREST for both the norms based on error indicators for  $h$  refinement and for  $p$  refinement.

The default is NEVER.

`integer print_time_when` – how often to print the amount of time used by the program. Valid values are:

NEVER – don't print.

PHASES – after each refinement/solve loop.

FINAL – only at the end.

FREQUENTLY – possibly more often than PHASES (used for debugging).

The default is NEVER.

`integer print_time_who` – which processes should print the time. Valid values are:

NO\_ONE – don't print.

SLAVES – slave processes print their own times.

MASTER – master process prints maximum time over all slaves.

EVERYONE – both SLAVES and MASTER.

MASTER\_ALL – master prints the individual times of each slave process.

The default is NO\_ONE.

`integer print_eval_when` – for eigenvalue problems, how often to print the eigenvalues. Valid values are:

NEVER – don't print.

PHASES – after each refinement/solve loop.

FINAL – only at the end.

The default is NEVER.

`integer print_eval_who` – for eigenvalue problems, which processes should print the eigenvalues. Valid values are:

NO\_ONE – don't print.

SLAVES – slaves print the eigenvalues.

MASTER – master process prints the eigenvalues.

EVERYONE – both SLAVES and MASTER.

The default is NO\_ONE.

`integer print_header_who` – which processes should print a header message and the values of the parameters when the subroutine starts. Valid values are:

NO\_ONE – don't print.

SLAVES – slaves print the header.

MASTER – master prints the header.

EVERYONE – both SLAVES and MASTER.

The default is MASTER.



`integer print_trailer_who` – which processes should print a trailer message when the subroutine completes. Valid values are:

- `NO_ONE` – don't print.
- `SLAVES` – slaves print the trailer.
- `MASTER` – master prints the trailer.
- `EVERYONE` – both `SLAVES` and `MASTER`.

The default is `MASTER`.

`logical print_warnings` – if `.false.`, warning messages are not printed.

The default is `.false.`

`integer clocks` – which clock(s) (cpu and/or wall) to use for timing. Valid values are:

- `CLOCK_C` – cpu clock.
- `CLOCK_W` – wall clock.
- `CLOCK_CW` – both cpu and wall clock.

The default is `CLOCK_W`.

`integer draw_grid_when` – how often to update the graphics. Valid values are:

- `NEVER` – don't draw.
- `PHASES` – after each refinement and solve phase.
- `FINAL` – only at the end.
- `FREQUENTLY` – possibly more often than `PHASES` (used for debugging).

The default is `NEVER`.

`logical pause_after_draw` – if `.true.`, the program will prompt for keyboard input after updating the graphics.

The default is `.false.`

`logical pause_after_phases` – if `.true.`, the program will prompt for keyboard input after each refinement/solve loop.

The default is `.false.`

`logical pause_at_start` – if `.true.`, the program will prompt for keyboard input before starting subroutine `phaml_solve_pde`.

The default is `.false.`

`logical pause_at_end` – if `.true.`, the program will prompt for keyboard input before returning from subroutine `phaml_solve_pde`.

The default is `.false.`

**integer degree** – sets the initial degree of the polynomials in the finite element space. If no  $p$  refinement is performed, it is the fixed degree of the space.

The default is to use the existing degree in each element (designated by 0). In `phaml_create`, all elements are initialized to have degree 1.

**integer inc\_quad\_order** – increment the order of the quadrature rules by this amount.

The default is 0.

**integer sequential\_vert** – number of vertices in the grid before it starts running in parallel.

The default is 100.

**real(my\_real) inc\_factor** – the factor by which to increase the size of the grid during one refinement phase.

The default is 2.0.

**integer error\_estimator** – select what to use for an error estimate to guide adaptive refinement. Valid values are:

**LOCAL\_PROBLEM** – computes  $h$  and  $p$  error indicators and estimates by solving a local Dirichlet ( $h$ ) and Neumann ( $p$ ) residual problem.

**HIERARCHICAL\_COEFFICIENT** – use the coefficient of the hierarchical basis at the newest vertex of the element. This only provides an estimate for  $h$  refinement of linear elements, and cannot be used with  $p$  refinement,  $hp$  refinement, or degree  $> 1$ . It is significantly faster than **LOCAL\_PROBLEM**, but of lower quality.

**TRUE\_DIFF** – use the difference between the true solution at the newest vertex of the element and surrounding vertices of the parent element. Can only be used if the true solution is known and supplied, and only for  $h$  refinement of linear elements.

**INITIAL\_CONDITION** – an error estimate based on interpolation of the function in subroutine `iconds` (Section 7.3.5).

The default is **INITIAL\_CONDITION** if `task` is **SET\_INITIAL**, and **LOCAL\_PROBLEM** otherwise.

**integer reftype** – type of refinement to perform. Valid values are:

**H\_UNIFORM** – uniform  $h$  refinement.

**H\_ADAPTIVE** – adaptive  $h$  refinement.

**P\_UNIFORM** – uniform  $p$  refinement.

**P\_ADAPTIVE** – adaptive  $p$  refinement.

**HP\_ADAPTIVE** – adaptive  $h$  and  $p$  refinement.

The default is H\_ADAPTIVE.

`integer refterm` – termination criteria for a refinement phase (`DOUBLE` or `HALVE` really mean to multiply or divide by `inc_factor`). Valid values are:

`DOUBLE_NVERT` – double the number of vertices.

`DOUBLE_NVERT_SMOOTH` – double the number of vertices, and then refine all remaining elements in the same error estimate bin (helps to maintain symmetries in the grid).

`DOUBLE_NELEM` – double the number of elements.

`DOUBLE_NELEM_SMOOTH` – double the number of elements, and then refine all remaining elements in the same error estimate bin.

`DOUBLE_NEQ` – double the number of equations.

`DOUBLE_NEQ_SMOOTH` – double the number of equations, and then refine all remaining elements in the same error estimate bin.

`HALVE_ERREST` – reduce the maximum error indicator by half.

`KEEP_NVERT` – keep the same number of vertices or reduce number to `max_vert`, and adjust the grid through derefinement and refinement.

`KEEP_NVERT_SMOOTH` – keep the same number of vertices or reduce number to `max_vert`, but then refine all remaining elements in the same error estimate bin.

`KEEP_NELEM` – keep the same number of elements or reduce number to `max_elem`.

`KEEP_NELEM_SMOOTH` – keep the same number of elements or reduce number to `max_elem`, but then refine all remaining elements in the same error estimate bin.

`KEEP_NEQ` – keep the same number of equations or reduce number to `max_eq`.

`KEEP_NEQ_SMOOTH` – keep the same number of equations or reduce number to `max_eq`, but then refine all remaining elements in the same error estimate bin.

`KEEP_ERREST` – keep the same maximum error indicator. Currently not supported.

`ONE_REF` – refine all elements with an error indicator larger than  $\text{reftol}/\sqrt{n}$  where  $n$  is the number of equations (degrees of freedom), but refine each element only once (just  $h$  or  $p$ , and don't refine children).

The default is `DOUBLE_NEQ_SMOOTH`.

`real(my_real) reftol` – tolerance for refining elements if `refterm` is `ONE_REF`.  
If `refterm` is `ONE_REF`, then at least one of `reftol`, `term_h_energy_err`,  
or `term_p_energy_err` must be given.

The default is `max(term_p_energy_err,term_h_energy_err)/2`.

`logical derefine` – if `.true.`, perform derefinement as well as refinement  
during adaptive refinement.

The default is `.true.`

`integer partition_method` – what method to use for partitioning the grid.  
Valid values are:

- `RTK` – the k-way refinement tree method implemented in PHAML.
- `ZOLTAN_RCB` – recursive coordinate bisection, from Zoltan.
- `ZOLTAN_OCT` – RPI's Octree method, from Zoltan.
- `ZOLTAN_METIS` – the local diffusion method from ParMETIS via Zoltan.
- `ZOLTAN_REFTREE` – the refinement tree method, from Zoltan.
- `ZOLTAN_RIB` – recursive inertial bisection, from Zoltan.
- `ZOLTAN_HSFC` – Hilbert space filling curve, from Zoltan.

The default is `RTK`.

`integer prebalance` – what to balance when partitioning the grid before  
refinement. Valid values are:

- `BALANCE_NONE` – do not partition the grid before refinement.
- `BALANCE_ELEMENTS` – equal number of elements in each partition.
- `BALANCE_VERTICES` – equal number of vertices in each partition.
- `BALANCE_EQUATIONS` – equal number of equations in each partition.

The default is `BALANCE_ELEMENTS`.

`integer postbalance` – what to balance when partitioning the grid after re-  
finement. Valid values are:

- `BALANCE_NONE` – do not partition the grid after refinement.
- `BALANCE_ELEMENTS` – equal number of elements in each partition.
- `BALANCE_VERTICES` – equal number of vertices in each partition.
- `BALANCE_EQUATIONS` – equal number of equations in each partition.

The default is `BALANCE_NONE`.

`integer solver` – what method to use as the solver. Valid values are:

- `MG_SOLVER` – the hierarchical basis multigrid method implemented in  
PHAML.

PETSC\_RICHARDSON\_SOLVER – Richardson solver from PETSc.

PETSC\_CHEBYCHEV\_SOLVER – Chebychev solver from PETSc.

PETSC\_CG\_SOLVER – Conjugate Gradients from PETSc.

PETSC\_GMRES\_SOLVER – Generalized Minimal Residual (GMRES) from PETSc.

PETSC\_TCQMR\_SOLVER – Transpose-Free Quasi-Minimal Residual (QMR) from PETSc.

PETSC\_BCGS\_SOLVER – BiConjugate Gradients Stabilized (BiCGSTAB) from PETSc.

PETSC\_CGS\_SOLVER – Conjugate Gradient Squared from PETSc.

PETSC\_TFQMR\_SOLVER – Transpose-Free Quasi-Minimal Residual (QMR) from PETSc.

PETSC\_CR\_SOLVER – Conjugate Residual from PETSc.

PETSC\_LSQR\_SOLVER – Least Squares from PETSc.

PETSC\_BICG\_SOLVER – BiConjugate Gradients from PETSc.

HYPRE\_BOOMERAMG\_SOLVER – BoomerAMG algebraic multigrid from hypre.

HYPRE\_PCG\_SOLVER – Preconditioned Conjugate Gradients from hypre.

HYPRE\_GMRES\_SOLVER – GMRES from hypre.

MUMPS\_SPD\_SOLVER – Symmetric positive definite solver from MUMPS (a parallel sparse direct solver). Only double precision is supported, i.e. `my_real=kind(0.0d0)` in `global.f90`.

MUMPS\_GEN\_SOLVER – General symmetric solver from MUMPS, for when the matrix might not be positive definite. Only double precision is supported, i.e. `my_real=kind(0.0d0)` in `global.f90`.

SUPERLU\_SOLVER – parallel sparse direct solver SuperLU.

LAPACK\_INDEFINITE\_SOLVER – the indefinite solver from LAPACK. This is available for certain debugging and testing operations, and can only be used for very small problem sizes and only with one processor.

LAPACK\_SPD\_SOLVER – the LAPACK solver for symmetric positive definite band matrices. This is available for certain debugging and testing operations, and can only be used for very small problem sizes and only with one processor.

Note: when the number of vertices is less than `sequential_vert`, the hypre and SuperLU solvers are replaced by a LAPACK solver for the whole matrix on each processor.)

The default is `MG_SOLVER`.

`integer preconditioner` – what method to use as a preconditioner for the Krylov methods. Valid values are:

**NO\_PRECONDITION** – no preconditioning.  
**MG\_PRECONDITION** – a hierarchical basis multigrid V-cycle as preconditioner.  
**FMG\_PRECONDITION** – an F-cycle of the MG preconditioner (full mg).  
**FUDOP\_DD\_PRECONDITION** – a domain decomposition preconditioner with FuDoP.  
**COARSE\_GRID\_PRECONDITION** – exact solver on a coarse grid as preconditioner.  
**PETSC\_JACOBI\_PRECONDITION** – Jacobi preconditioner from PETSc.  
**PETSC\_BJACOBI\_PRECONDITION** – Block Jacobi preconditioner from PETSc.  
**PETSC\_SOR\_PRECONDITION** – SOR (and SSOR) preconditioner from PETSc.  
**PETSC\_EISENSTAT\_PRECONDITION** – SOR with Eisenstat trick from PETSc.  
**PETSC\_ICC\_PRECONDITION** – Incomplete Cholesky preconditioner from PETSc.  
**PETSC\_ILU\_PRECONDITION** – Incomplete LU preconditioner from PETSc.  
**PETSC\_ASM\_PRECONDITION** – Additive Schwarz preconditioner from PETSc.  
**HYPRE\_DS\_PRECONDITION** – Diagonal scaling preconditioner from hypre.  
**HYPRE\_BOOMERAMG\_PRECONDITION** – BoomerAMG algebraic multigrid from hypre.  
**HYPRE\_PARASAILS\_PRECONDITION** – ParaSails sparse approximate inverse from hypre.

Note: For `solver=HYPRE_PCG_SOLVER`, the preconditioner must be one of `NO_PRECONDITION`, `HYPRE_DS_PRECONDITION`, or `HYPRE_BOOMERAMG_PRECONDITION`. For `solver=HYPRE_GMRES_SOLVER`, the preconditioner must be one of `NO_PRECONDITION`, `HYPRE_DS_PRECONDITION`, `HYPRE_BOOMERAMG_PRECONDITION`, or `HYPRE_PARASAILS_PRECONDITION`. The hypre preconditioners can only be used with the hypre PCG and GMRES solvers.

The default is:

**NO\_PRECONDITION** – when not applicable  
**HYPRE\_BOOMERAMG\_PRECONDITION** – for the hypre solvers  
**MG\_PRECONDITION** – otherwise

**integer coarse\_size** – for the coarse grid preconditioner, maximum size of the coarsened grid.

The default is 5000.

**integer coarse\_method** – for the coarse grid preconditioner, the method to use to solve the coarse grid problem. Permitted values are

LAPACK\_INDEFINITE\_SOLVER – the indefinite solver from LAPACK.

MUMPS\_GEN\_SOLVER – general symmetric solver from MUMPS.

SUPERLU\_SOLVER – parallel sparse direct solver SuperLU.

The default is LAPACK\_INDEFINITE\_SOLVER.

`integer mg_cycles` – number of multigrid V-cycles to use in one solution phase, or as a preconditioner, or number of V-cycles on each level during the F-cycle preconditioner.

If `mg_tol` is MG\_NO\_TOL, the default is 1; otherwise it is infinite.

`real(my_real) mg_tol` – perform multigrid cycles until the  $\ell^2$  norm of the scaled linear system residual is less than `mg_tol` (up to a maximum of `mg_cycles` cycles). In addition to positive real numbers, it can have the following values:

MG\_NO\_TOL – do not use the tolerance test for ending the multigrid cycles, use a fixed number of cycles given by `mg_cycles` instead.

MG\_ERREST\_TOL – reduce the residual to some fraction of the error estimate.

The default is MG\_ERREST\_TOL.

`integer mg_prerelax` – number of half red-black relaxation sweeps to make before coarse grid correction in the  $h$ -hierarchical linear bases.

The default is 1.

`integer mg_postrelax` – number of half red-black relaxation sweeps to make after coarse grid correction in the  $h$ -hierarchical linear bases.

The default is 2.

`integer mg_prerelax_ho` – number of half red-black relaxation sweeps to make before coarse grid correction in the  $p$ -hierarchical high order bases. In addition to nonnegative integers, it can have the following values:

MG\_RELAX\_HO\_MAXDEG – number of full red-black relaxations sweeps equals the maximum degree of any element.

The default is MG\_RELAX\_HO\_MAXDEG.

`integer mg_postrelax_ho` – number of half red-black relaxation sweeps to make after coarse grid correction in the  $p$ -hierarchical high order bases. In addition to nonnegative integers, it can have the following values:

MG\_RELAX\_HO\_MAXDEG – number of full red-black relaxations sweeps equals the maximum degree of any element.

The default is MG\_RELAX\_HO\_MAXDEG.

`integer dd_iterations` – number of iterations to use for the FuDoP domain decomposition preconditioner.

The default is 1.

`logical petsc_matrix_free` – if the solver is one of the PETSc methods and this parameter is `.true.`, memory is saved by using a matrix-free approach that does not copy the matrix to a PETSc data structure. If `.true.`, then you cannot use the PETSc preconditioners.

The default is `.false.`

`logical ignore_quad_err` – if `.true.`, when setting up the linear system, ignore the quadrature errors in the large triangles outside the owned region. This is acceptable (and reduces computation and communication) when the PDE coefficients and right hand side are constant (for example, Laplace’s equation), but will reduce the convergence of the discretization error otherwise.

The default is `.false.`

`integer num_eval` – For eigenvalue problems, the number of eigenvalues to find.

The default is 1.

`real(my_real) lambda0` – For eigenvalue problems, find the eigenvalues closest to `lambda0`.

The default is  $-\infty$  defined as `-huge(0.0_my_real)`, i.e., find the smallest eigenvalues.

`integer scale_evec` – For eigenvalue problems, the eigenvectors can be multiplied (scaled) by an arbitrary constant. These are the choices for scaling:

`SCALE_LINF` – scale so the (discrete)  $\ell^\infty$  norm is 1. For linear elements, this is also the  $L^\infty$  norm.

`SCALE_L2` – scale so the (discrete)  $\ell^2$  norm is 1.

`SCALE_M` – scale so the  $M$  norm,  $\sqrt{x^T M x}$ , is 1, where  $M$  is the mass matrix. The  $M$  norm is also the (continuous)  $L^2$  norm.

The default is `SCALE_LINF`.

The following parameters are passed to ARPACK. See the ARPACK documentation for more information on them.

`integer arpack_ncv` – number of Lanczos basis vectors to use

The default is 20.

`integer arpack_maxit` – maximum number of IRLM iterations

The default is 100.



real(my\_real) arpack\_tol – relative accuracy to which eigenvalues are computed

The default is  $10^{-10}$ .

The following parameters are passed to subroutines in hypre. See the documentation for hypre for explanations and default values.

integer	hypre_BoomerAMG_MaxLevels
integer	hypre_BoomerAMG_MaxIter
real(my_real)	hypre_BoomerAMG_Tol
real(my_real)	hypre_BoomerAMG_StrongThreshold
real(my_real)	hypre_BoomerAMG_MaxRowSum
integer	hypre_BoomerAMG_CoarsenType
integer	hypre_BoomerAMG_MeasureType
integer	hypre_BoomerAMG_CycleType
integer	hypre_BoomerAMG_NumGridSweeps(:)
integer	hypre_BoomerAMG_GridRelaxType(:)
integer	hypre_BoomerAMG_GridRelaxPoints(:, :)
real(my_real)	hypre_BoomerAMG_RelaxWeight(:)
integer	hypre_BoomerAMG_IOutDat (not available after hypre 1.6.0)
integer	hypre_BoomerAMG_DebugFlag
real(my_real)	hypre_ParaSails_thresh
integer	hypre_ParaSails_nlevels
real(my_real)	hypre_ParaSails_filter
integer	hypre_ParaSails_sym
real(my_real)	hypre_ParaSails_loadbal
integer	hypre_ParaSails_reuse
integer	hypre_ParaSails_logging
real(my_real)	hypre_PCG_Tol
integer	hypre_PCG_MaxIter
integer	hypre_PCG_TwoNorm
integer	hypre_PCG_RelChange
integer	hypre_PCG_Logging
integer	hypre_GMRES_KDim
real(my_real)	hypre_GMRES_Tol
integer	hypre_GMRES_MaxIter
integer	hypre_GMRES_Logging

The following parameters are passed to subroutines in PETSc. See the documentation for PETSc for explanations and default values.

real(my_real)	petsc_richardson_damping_factor
real(my_real)	petsc_chebychev_emin
real(my_real)	petsc_chebychev_emax
integer	petsc_gmres_max_steps
real(my_real)	petsc_rtol
real(my_real)	petsc_atol
real(my_real)	petsc_dtol
integer	petsc_maxits
integer	petsc_ilu_levels
integer	petsc_icc_levels
real(my_real)	petsc_ilu_dt
real(my_real)	petsc_ilu_dtcol
integer	petsc_ilu_maxrowcount
real(my_real)	petsc_sor_omega
integer	petsc_sor_its
integer	petsc_sor_lits
logical	petsc_eisenstat_nodiagscaling
real(my_real)	petsc_eisenstat_omega
integer	petsc_asm_overlap

#### 7.4.15 phaml\_store

phaml\_store stores information from phaml\_solution into files for later use.

```

subroutine phaml_store(phaml_solution, unit)
type (phaml_solution_type), intent(in) :: phaml_solution - the so-
  lution to store.
integer, intent(in) :: unit - the unit number to write to, which should
  have been opened as either FORMATTED or UNFORMATTED with sub-
  routine phaml_popen. UNFORMATTED is likely to be faster and create
  smaller data files. FORMATTED may be required if you will be restoring
  with a different compiler or architecture, and might not reproduce floating
  point numbers exactly.

```

## References

- [1] M. Ainsworth and J.T. Oden, A Posteriori Error Analysis in Finite Element Analysis, Wiley Interscience Publishers, New York, 2000.
- [2] P. R. Amestoy, I. S. Duff and J.-Y. L'Excellent (1998), Multifrontal parallel distributed symmetric and unsymmetric solvers, in Comput. Methods in Appl. Mech. Eng., 184, 501-520 (2000).
- [3] P. R. Amestoy, I. S. Duff, J. Koster and J.-Y. L'Excellent, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal of Matrix Analysis and Applications, Vol 23, No 1, pp 15-41 (2001).

- [4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK User's Guide, Third Edition, SIAM, Philadelphia, PA, USA, 1999.
- [5] Satish Balay, Victor Eijkhout, William D. Gropp, Lois Curfman McInnes and Barry F. Smith, Efficient Management of Parallelism in Object Oriented Numerical Software Libraries, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset and H. P. Langtangen, eds., 163–202, Birkhäuser Press, 1997.
- [6] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith and Hong Zhang, PETSc Users Manual, ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.
- [7] P. Carnevali, R. B. Morris, Y. Tsuji and G. Taylor, New Basis Functions and Computational Procedures for the p-Version Finite Element Analysis, *Int. J. Num. Meth. Engng.*, 36, (1993), pp. 3759-3779.
- [8] K. Devine, B. Hendrickson, E. Boman, M. St. John, C. Vaughan and W. F. Mitchell, Zoltan: A dynamic load-balancing library for parallel applications, User's Guide, Sandia Technical Report SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 2000.
- [9] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, Courtenay Vaughan, Zoltan Data Management Services for Parallel Dynamic Applications, *Computing in Science and Engineering*, 2002, 4, 2, 90–97.
- [10] R.D. Falgout and U.M. Yang, hypre: a Library of High Performance Preconditioners, in Computational Science - ICCS 2002 Part III, P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, and A.G. Hoekstra, eds., vol. 2331 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 632-641.
- [11] R.D. Falgout, J.E. Jones, and U.M. Yang, The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners, in Numerical Solution of Partial Differential Equations on Parallel Computers, A.M. Bruaset, P. Bjrstad, and A. Tveito, eds., Lecture Notes in Computational Science and Engineering, Springer-Verlag, 51 (2006), pp. 267-294.
- [12] R. B. Lehoucq, D. C. Sorensen, and C. Yang, ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods, SIAM.
- [13] Xiaoye S. Li and James W. Demmel, SuperLU\_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems, *ACM Trans. Mathematical Software*, 2003, 29, 2, 110–140.
- [14] W.F. Mitchell, Adaptive refinement for arbitrary finite element spaces with hierarchical bases, *J. Comp. Appl. Math.* 36 (1991), pp. 65–78.

- [15] W.F. Mitchell, Optimal multilevel iterative methods for adaptive grids, *SIAM J. Sci. Statist. Comput.* 13 (1992), pp. 146–167.
- [16] W.F. Mitchell, The Full Domain Partition Approach to Parallel Adaptive Refinement, in *Grid Generation and Adaptive Algorithms, IMA Volumes in Mathematics and its Applications*, 113, Springer-Verlag, 1998, pp. 151–162.
- [17] W.F. Mitchell, A Parallel Multigrid Method Using the Full Domain Partition, *Electronic Transactions on Numerical Analysis*, 6, 224–233 (1998).
- [18] W.F. Mitchell, Multigrid Methods for the hp Version of the Finite Element Method, *8th U.S. National Congress on Computational Mechanics*, July 2005.
- [19] W.F. Mitchell, A Refinement-tree Based Partitioning Method for Dynamic Load Balancing with Adaptively Refined Grids, *J. Par. Dist. Comput.*, to appear.
- [20] W.F. Mitchell, Initial Grid Generation for Newest Node Bisection Refinement of Triangles, in process.
- [21] J.R. Rice, Algorithm 625: A Two Dimensional Domain Processor, *ACM Trans. Math. Software*, 10, (1984), 443–452.
- [22] Jonathan Richard Shewchuk, Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, in “Applied Computational Geometry: Towards Geometric Engineering” (Ming C. Lin and Dinesh Manocha, editors), volume 1148 of Lecture Notes in Computer Science, pages 203–222, Springer-Verlag, Berlin, May 1996.
- [23] Jonathan Richard Shewchuk, Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator, <http://www.cs.cmu.edu/~quake/triangle.html>
- [24] G. Strang and G. Fix, *An Analysis of the Finite Element Method*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
- [25] B. Szabo and I. Babuška, *Introduction to Finite Element Analysis*, John Wiley and Sons, New York, 1989.