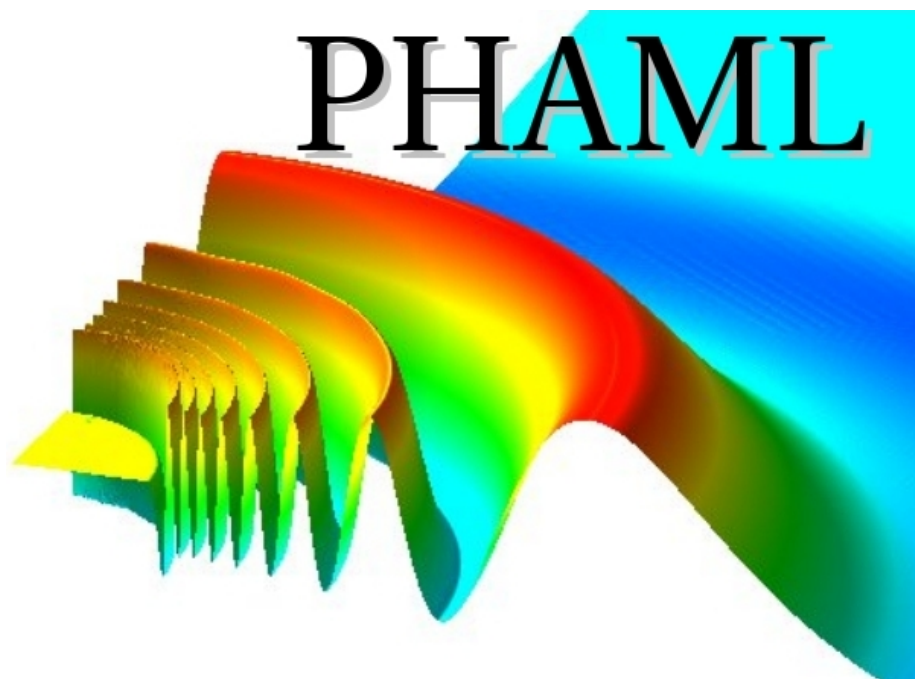


NISTIR 7374

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

Revised September 3, 2014 for Version 1.14.0



PHAML User's Guide, Version 1.14.0

William F. Mitchell¹
Applied and Computational Mathematics Division
100 Bureau Drive Stop 8910
National Institute of Standards and Technology
Gaithersburg, MD 20899-8910
email: phaml@nist.gov

¹Contribution 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.

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

1	Introduction	6
2	Software	8
2.1	Obtaining Software	8
2.1.1	PHAML	8
2.1.2	Fortran 90 and C Compilers	8
2.1.3	BLAS and LAPACK	9
2.1.4	MPI	9
2.1.5	OpenGL (or Mesa), GLUT and f90gl	9
2.1.6	Triangle	10
2.1.7	Gmsh	10
2.1.8	SLEPc	10
2.1.9	ARPACK	10
2.1.10	BLOPEX	10
2.1.11	PETSc	10
2.1.12	MUMPS	11
2.1.13	SuperLU	11
2.1.14	<i>hypre</i>	11
2.1.15	ML	11
2.1.16	Zoltan	11
2.2	Compiling PHAML	12
2.2.1	Creating the Makefiles	12
2.2.2	Compiling the Library	15
2.3	Testing the Library	15
2.4	Compiling and Running the Examples	16
3	Scalar Linear Elliptic Boundary Value Problems	19
3.1	Main program	19
3.1.1	Parallelism	19
3.1.2	Program structure	20
3.1.2.1	Master/slave and Sequential	20
3.1.2.2	SPMD	21
3.2	Defining the problem	23
3.2.1	Defining the PDE	23

3.2.2	Defining the boundary conditions	24
3.2.3	Defining the domain and initial grid	26
3.2.4	The true solution	28
3.3	Solution method	29
3.3.1	Discretization	29
3.3.2	Refinement	30
3.3.3	Error indicator	35
3.3.4	Linear system solver	37
3.3.4.1	Hierarchical basis multigrid solver	38
3.3.4.2	Krylov space solvers	39
3.3.4.3	Alternative direct solvers	40
3.3.4.4	Alternative iterative solvers	40
3.3.5	Load balancing	43
3.3.6	Termination	47
3.4	I/O	48
3.4.1	I/O files	48
3.4.2	Printed I/O	49
3.4.3	Pausing	50
3.5	Graphics	50
3.5.1	Overview	50
3.5.2	Example visualizations	52
3.5.3	View modifier	58
3.5.4	Colors	59
3.5.5	Functions	60
3.5.6	Lights	61
3.5.7	Contour plots	61
3.5.8	Multiple solutions	61
3.5.9	Miscellaneous features	62
3.5.10	Development aids	64
3.5.11	Postscript	64
3.6	Post-solution utilities	65
3.6.1	Store and Restore	65
3.6.2	Store Matrix	65
3.6.3	Store Grid	66
3.6.4	Query	67
3.6.5	Solution evaluation	67
3.6.6	Functionals	68
4	Problem Extensions	69
4.1	Eigenvalue Problems	69
4.2	Coupled Systems or Multicomponent Solutions	71
4.3	Parabolic, Nonlinear, Etc. Problems	72
4.4	3D Problems	73
5	Examples	78

6	Release notes	80
6.1	Version 1.14.0	80
6.1.1	Summary of changes	80
6.1.2	Major changes	82
6.1.3	Upgrading	82
6.2	Version 1.13.0	82
6.2.1	Summary of changes	83
6.2.2	Major changes	84
6.2.3	Upgrading	85
6.3	Version 1.12.1	85
6.3.1	Summary of changes	85
6.3.2	Major changes	85
6.3.3	Upgrading	85
6.4	Version 1.12.0	86
6.4.1	Summary of changes	86
6.4.2	Major changes	87
6.4.3	Upgrading	88
6.5	Version 1.11.0	88
6.5.1	Summary of changes	88
6.5.2	Major changes	89
6.5.3	Upgrading	90
6.6	Version 1.10.0	90
6.6.1	Summary of changes	90
6.6.2	Major changes	92
6.6.3	Upgrading	92
6.7	Earlier versions	93
7	Reference Manual	94
7.1	Quick Start	94
7.1.1	Obtaining the software	94
7.1.2	Compiling the PHAML library	95
7.1.3	Compiling an Example	95
7.1.4	Running the Example	95
7.1.5	Now what?	96
7.2	Public Entities in PHAML	96
7.2.1	phaml_solution_type	96
7.2.2	my_real	96
7.2.3	pde and my_pde_id	96
7.2.4	symbolic constants	97
7.3	User Provided Routines	97
7.3.1	bconds	97
7.3.2	boundary_point	98
7.3.3	boundary_npiece	98
7.3.4	boundary_param	98
7.3.5	iconds	99
7.3.6	pdecoefs	99

7.3.7	phaml_integral_kernel	100
7.3.8	regularity	100
7.3.9	truecs	101
7.3.10	truexs	101
7.3.11	trueys	101
7.3.12	update_usermod	102
7.4	PHAML procedures	102
7.4.1	phaml_compress	102
7.4.2	phaml_connect	102
7.4.3	phaml_copy_soln_to_old	104
7.4.4	phaml_create	104
7.4.5	phaml_destroy	106
7.4.6	phaml_evaluate	107
7.4.7	phaml_get_grid_soln	107
7.4.8	phaml_evaluate_old	108
7.4.9	phaml_integrate	108
7.4.10	phaml_pclose	109
7.4.11	phaml_popen	109
7.4.12	phaml_query	110
7.4.13	phaml_restore	113
7.4.14	phaml_scale	113
7.4.15	phaml_solve_pde	113
	7.4.15.1 Miscellaneous arguments	115
	7.4.15.2 Termination arguments	116
	7.4.15.3 Output control arguments	117
	7.4.15.4 Refinement arguments	123
	7.4.15.5 Load balancing arguments	127
	7.4.15.6 Assembly arguments	128
	7.4.15.7 Solver arguments	128
	7.4.15.8 Eigenvalue arguments	135
7.4.16	phaml_store	137
7.4.17	phaml_store_grid	137
7.4.18	phaml_store_matrix	138

Chapter 1

Introduction

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

Please note: This document changes with each release of PHAML. Changes made in the current release are printed in **red**. Changes made in a recent release are printed in **dark red**. Text that has been deleted is maintained for a few releases for comparison, but is printed in a tiny font size.

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

$$-\frac{\partial}{\partial x} \left(c_{xx} \frac{\partial u}{\partial x} \right) - \frac{\partial}{\partial x} \left(c_{xy} \frac{\partial u}{\partial y} \right) - \frac{\partial}{\partial y} \left(c_{yy} \frac{\partial u}{\partial y} \right) + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} + c_u u = f \quad \text{in } \Omega \quad (1.1)$$

where c_{xx} , c_{xy} , c_{yy} , c_x , c_y , c_u and f are functions of x and y , and the domain Ω is a bounded, connected, region in R^2 . The boundary conditions can be Dirichlet on part of the boundary,

$$u = g \quad \text{on } \partial\Omega_D \quad (1.2)$$

and natural or mixed on the remainder of the boundary,

$$\left(c_{xx} \frac{\partial u}{\partial x} + c_{xy} \frac{\partial u}{\partial y} \right) \frac{\partial y}{\partial s} - c_{yy} \frac{\partial u}{\partial y} \frac{\partial x}{\partial s} + c_{bc} u = g \quad \text{on } \partial\Omega_N \quad (1.3)$$

where g and c_{bc} are functions of x and y , c_{xx} , c_{xy} and c_{yy} are the same functions as in Equation 1.1, and 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 $c_{xx} = c_{yy} = 1$ and $c_{xy} = 0$ or the boundaries of the domain are parallel to the axes, the natural boundary condition reduces to the Neumann boundary condition

$$\frac{\partial u}{\partial n} = g$$

where n is the outward unit normal.

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 λu , and λ is an eigenvalue to be determined along with u . For eigenvalue problems, the boundary conditions must be homogeneous, i.e., $g = 0$.

PHAML also solves 3D elliptic PDEs, with the obvious extensions to Equations 1.1 and 1.3. See Section 4.4. Most of the rest of the user's guide will only address the 2D problem.

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. If the cross derivative or first order terms are included, i.e. $c_{xy} \neq 0$, $c_x \neq 0$ or $c_y \neq 0$, then one of the alternate solution methods for nonsymmetric matrices must be used. 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, an SPMD parallel program using MPI-1, MPI-2, an SMP program using OpenMP, or a hybrid parallel program using both MPI and OpenMP. PHAML optionally uses OpenGL to provide menu-driven interactive visualization.

Chapter 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 MPI

To run PHAML as a message passing parallel program, you must have an MPI library. 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>.

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>. Alternatively, if you are installing Mesa, then also get MesaGLUT. This distribution works with f90gl since version 7.0.3.

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 [41], to generate the initial grid for arbitrary geometry in 2D. 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 Gmsh

PHAML uses Gmsh [18, 19] to generate the initial grid for arbitrary geometry in 3D. If you are only solving problems on the unit cube, then you do not need Gmsh – you can use the Gmsh `.msh` files in the examples. You also do not need it if your domain has only planar faces and you have obtained `.msh` files elsewhere. Otherwise, you will need it. Gmsh is available at <http://geuz.org/gmsh>.

2.1.8 SLEPc

To solve elliptic eigenvalue problems, PHAML needs an algebraic eigensolver. SLEPc [10] is used for this. You can get SLEPc from <http://www.grycap.upv.es/slep>. SLEPc requires that you also have PETSc (sect. 2.1.11) installed.

2.1.9 ARPACK

ARPACK [24] is an external package supported by SLEPc, and can be used as an eigensolver for PHAML, accessed through SLEPc. It is optional, but if you choose to include it, ARPACK is compiled separate from SLEPc, and you must enable ARPACK support when you configure SLEPc for compilation. You can get ARPACK and PARPACK from <http://www.caam.rice.edu/software/ARPACK>. For a sequential compilation you do not need PARPACK.

2.1.10 BLOPEX

BLOPEX [22, 23] is an external package supported by SLEPc, and can be used as an eigensolver for PHAML, accessed through SLEPc. It is optional, but if you choose to include it, BLOPEX is compiled with SLEPc by using `--download-blopex` when you configure SLEPc for compilation.

2.1.11 PETSc

PETSc [7][8] 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>.

2.1.12 MUMPS

MUMPS [3][4] is a parallel direct linear system solver. It is not required, but can be included to provide an alternative linear system solver. It is accessed through PETSc, so you must also install PETSc and use `--download-mumps --download-parmetis --download-scalapack --download-blacs` when configuring PETSc.

2.1.13 SuperLU

SuperLU [25] is a parallel direct linear system solver from the Lawrence Berkeley National Laboratory. It is not required, but can be included to provide an alternative linear system solver. It is accessed through PETSc, so you must also install PETSc and use `--download-superlu` for a sequential compilation or `--download-superlu-dist --download-parmetis` for a parallel compilation when configuring PETSc.

2.1.14 *hypre*

hypre [15][16] is a package of iterative linear system solvers and preconditioners from the Lawrence Livermore National Laboratory. It is not required, but can be included linked with PHAML to provide alternative linear system solvers. It is accessed through PETSc, so you must also install PETSc and use `--download-hypre` when configuring PETSc. It is available at http://www.llnl.gov/CASC/linear_solvers.

2.1.15 ML

ML provides multi-level, multigrid-like preconditioners. It is part of the Trilinos Project [49] at Sandia National Laboratory. It is not required, but can be included to provide an alternative AMG preconditioner. It is accessed through PETSc, so you must also install PETSc and use `--download-ml` when configuring PETSc.

2.1.16 Zoltan

Zoltan [13][14] 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>.

Zoltan may be compiled with support for certain third party libraries. Two of these are supported in PHAML through Zoltan. ParMETIS is a static partitioning library. Zoltan includes a complimentary copy of the ParMETIS software. DRUM is utility for load balancing on heterogeneous or dynamically changing parallel computers. It has not yet been officially released and can only be obtained by requesting it from the author.

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`. This script will make all the **Makefiles** in all the PHAML subdirectories, so you do not need to run any of the other `mkmkfile.shs`. 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` `cygwin` `irixn32` `irix64` `linux` `solaris`. This defines the operating system running on your computer.
- `PHAML_F90` `absoft` `g95` `gfortran` `intel` `lahey` `mingw` `nag` `pathscale` `pgi` `sgi` `sun` `xl`. 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` `openmp` `hybrid_spawn` `hybrid_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, running an SPMD (single program multiple data) program in which all processes are identical and started from the command line, running an SMP program using OpenMP, and running a hybrid OpenMP/MPI program either as master/slave or SPMD. Note: not all compilers support OpenMP.

- PHAML_PARLIB `lam mpi mpich mpich2 myrinet openmpi none`. This defines the parallel library to use. `lam`, `mpich`, `mpich2` and `openmpi` 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` or `openmp`.
- PHAML_ELEMENT `triangle tetrahedron tetrahedron_fixdeg`. This defines what kind of elements will be used, which determines whether PHAML will be compiled for 2D or 3D problems. See Section 4.4 for the difference between `tetrahedron` and `tetrahedron_fixdeg`.
- 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_SLEPC `no yes`. This indicates whether or not to include SLEPc.
- PHAML_ARPACK `no yes`. Should be yes if and only if SLEPc was built with ARPACK support.
- PHAML_BLOPEX `no yes`. Should be yes if and only if SLEPc was built with BLOPEX support.
- PHAML_PETSC `no yes`. This indicates whether or not to include PETSc.
- PHAML_MUMPS `no yes`. Should be yes if and only if PETSc was built with MUMPS support.
- PHAML_SUPERLU `no yes`. Should be yes if and only if PETSc was built with SUPERLU support.

- **PHAML_HYPRE** no yes. Should be yes if and only if PETSc was built with hypre support. This indicates whether or not to include *hypre*.
- **PHAML_ML** no yes. Should be yes if and only if PETSc was built with ML support.
- **PHAML_ZOLTAN** no yes. This indicates whether or not to use Zoltan. When Zoltan was compiled, it may or may not have included support for third party libraries. The next few parameters indicate which of them were built into Zoltan. These must agree exactly with how Zoltan was built, or you will get error messages from the linker.
- **PHAML_PARMETIS** no yes. Should be yes if and only if Zoltan was built with ParMETIS support.
 - **PHAML_JOSTLE** no yes. Should be yes if and only if Zoltan was built with JOSTLE support.
 - **PHAML_PATOH** no yes. Should be yes if and only if Zoltan was built with PaToH support.
 - **PHAML_PARKWAY** no yes. Should be yes if and only if Zoltan was built with ParKway support.
 - **PHAML_NEMESIS** no yes. Should be yes if and only if Zoltan was built with Nemesis support.
- **PHAML_DRUM** no yes. Should be yes if and only if Zoltan was built with DRUM support.
- **PHAML_SYSTEM** none looney pepe raritan speedy suns. 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 MPI 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`, `messpass_spawn`, `openmp` or `hybrid_spawn` programs.

To run the full test suite on an interactive system, enter the command

```
make test
```

from either the PHAML root directory or the `testdir` subdirectory. If you are using a batch system like PBS or Torque/Maui, you can run the tests with one of the PBS scripts in the `testdir` directory. 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 in-

cluded, 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 sub-directory, with or without **test_**. For example, to run the PETSc tests you can use either

```
make test what=test_petsc, or
make test what=petsc
```

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**. These **Makefiles** were created when you ran **mkmkfile.sh** in the top PHAML directory. You should not run the **mkmkfile.sh** in the **examples** subdirectories.

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. 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
```

3. 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
```

4. For OpenMP or hybrid OpenMP/MPI programs, set the `OMP_NUM_THREADS` environment variable to the desired number of threads, probably equal to the number of cores on the SMP computer, or number of cores on each node of the distributed memory parallel computer, and run the same as a sequential or MPI program.

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

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.

Chapter 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 1.1 with Dirichlet, natural (often Neumann), or mixed boundary conditions given by Equations 1.2 and 1.3. 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 six 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 3.1.

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,

model	PHAML_PARALLEL
sequential	<code>sequential</code>
master/slave	<code>messpass_spawn</code>
SPMD	<code>messpass_nospawn</code>
OpenMP SMP	<code>openmp</code>
hybrid OpenMP/MPI master/slave	<code>hybrid_spawn</code>
hybrid OpenMP/MPI SPMD	<code>hybrid_nospawn</code>

Table 3.1: Correspondence between PHAML_PARALLEL in `mkmkfile.sh` and the parallel programming model.

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 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. With OpenMP, the parallelism is completely transparent to the user, except for setting the desired number of threads for each process through the environment variable `OMP_NUM_THREADS` or the `nthread` argument to `phaml.create`. Usually this is equal to the number of cores on an SMP computer, or, for hybrid OpenMP/MPI programs, the number of cores on each node of a distributed memory parallel computer or cluster. However, it is sometimes advantageous to use one less than the number of cores if the operating system is running a lot of background processes that would run on the unused core.

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 same program can be used for sequential, OpenMP and hybrid OpenMP/MPI master/slave models. 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 3.1 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.


```

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 3.1: A simple main program.

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 7.4.15. 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 3.2 for a program that uses an MPI library. The same program can be used for a hybrid OpenMP/MPI SPMD model.

In addition to using module `phaml`, it must use module `mpif_mod` to get

```

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 3.2: 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 3.3: `pdecoefs` for Laplace's equation.

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 3.2.

3.2 Defining the problem

3.2.1 Defining the PDE

The PDE in Equation 1.1 must be defined in subroutine `pdecoefs`. For convenience, the equation is repeated here.

$$-\frac{\partial}{\partial x} \left(c_{xx} \frac{\partial u}{\partial x} \right) - \frac{\partial}{\partial x} \left(c_{xy} \frac{\partial u}{\partial y} \right) - \frac{\partial}{\partial y} \left(c_{yx} \frac{\partial u}{\partial x} \right) + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} + c_u u = f \quad \text{in } \Omega$$

`pdecoefs` receives a point (x, y) and returns the value of the functions c_{xx} , c_{xy} , c_{yy} , c_x , c_y , c_u and f at that point in the arguments `cxx`, `cxy`, `cyy`, `cx`, `cy`, `c` and `rs` respectively. The interface for this routine is given in Section 7.3.6. For a scalar PDE, the return variables are 1×1 arrays (they have higher dimension for

systems of equations; see Section 4.2) and can be assigned with a whole array assignment statement, making their “arrayness” transparent. Figure 3.3 shows how `pdecoefs` could be written for Laplace’s equation $\nabla^2 u = 0$.

Certain characteristics of the PDE might have an effect on the solvers. For example, if there are first order terms then the matrix is not symmetric and a nonsymmetric solver must be used. Also, PHAML can take advantage of some characteristics, such as knowing the operator is the Laplacian, to get better performance. PHAML attempts to determine the relevant characteristics by sampling `pdecoefs` at the midpoints of the initial grid. It is possible for this test to fail. If so, you can force PHAML to know the characteristic through the following arguments to `phaml_solve_pde`:

- Set `pde_has_first_order_terms` to `.true.` if c_x or c_y is not identically zero.
- Set `pde_has_cross_derivative` to `.true.` if c_{xy} is not identically zero.
- Set `laplacian_operator` to `.true.` if c_{xx} and c_{yy} are identically one while c_{xy} , c_x , c_y and c are identically zero.
- Set `isosceles_right_triangles` to `.true.` if every initial element is an isosceles right triangle and they are paired up such that all the right angles are peaks.

In most cases where `task=SET_INITIAL` the automatic testing for these characteristics will call `pdecoefs` before the old solution has been set by `phaml_copy_soln_to_old` resulting in many warnings. You should include the above variables in that call to avoid these warnings.

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 1.2

$$u = g,$$

natural (usually Neumann) or mixed as defined by Equation 1.3

$$\left(c_{xx} \frac{\partial u}{\partial x} + c_{xy} \frac{\partial u}{\partial y} \right) \frac{\partial y}{\partial s} - c_{yy} \frac{\partial u}{\partial y} \frac{\partial x}{\partial s} + c_{bc} u = g$$

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).

```

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

boundary condition	itype
Eq. 1.2	DIRICHLET
Eq. 1.3 with $c_{bc} = 0$	NATURAL
Eq. 1.3 with $c_{bc} \neq 0$	MIXED
periodic	PERIODIC

Table 3.2: Permitted values for `itype` to specify the type of boundary conditions.

The type of boundary condition for the indicated piece is returned in `itype` using a symbolic constant listed in Table 3.2. The functions c_{xx} , c_{xy} and c_{yy} in Equation 1.3 are the same as those in the PDE, Equation 1.1, which are defined in subroutine `pdecoefs` (Section 3.2.1) and not repeated here. The function c_{bc} in Equation 1.3 is returned in the variable `c`. Finally, the function g in Equations 1.2 and 1.3 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 3.4 shows a subroutine for the boundary conditions

$$\partial u / \partial n = 0 \quad \text{on } x = 0 \quad (3.1)$$

$$u = 0 \quad \text{elsewhere} \quad (3.2)$$

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 [41]. 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 [39].

`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 **hole**th 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 [42] for the exact definition of a `.poly` file. You should specify a boundary marker for each vertex and boundary segment. The boundary marker must be 0 if and only if the vertex or edge is interior, i.e., not on the boundary. 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 -pneqj 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 [42].

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. 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 7.3.9, 7.3.10 and 7.3.11. 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 7.4.15 which formally defines the arguments to `phaml_solve_pde`.

3.3.1 Discretization

PHAML uses a standard Galerkin finite element method (see, for example [44][46]) to approximate the solution of Equations 1.1-1.3, 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. The p -hierarchical basis, $\Phi = \{\phi_i\}$, of either Szabo & Babuška [46] or Carnevali et al. [11] spans the space, and gives a representation for the approximate solution

$$u_T = \sum_{\phi_i \in \Phi} \alpha_i \phi_i \quad (3.3)$$

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

$$\langle u, v \rangle = \int_{\Omega} c_{xx} u_x v_x + c_{xy} u_y v_x + c_{yy} u_y v_y + c_x u_x v + c_y u_y v + c_u uv \quad (3.4)$$

where c_{xx} , c_{xy} , c_{yy} , c_x , c_y , and c_u are from Equation 1.1, and the subordinate energy norm $\|u\|^2 = \langle u, u \rangle$. (Note: We need $c_{xy} = c_x = c_y = 0$ for 3.4 to be an inner product because of symmetry, and $c_u \geq 0$ for $\langle u, u \rangle$ to be a norm because of positivity, but formally we proceed with these definitions knowing that some of the following statements are not strictly true.) 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

$$\langle u_T, \phi_i \rangle = \langle f, \phi_i \rangle_2 \quad \forall \phi_i \in \Phi \quad (3.5)$$

where $\langle \cdot, \cdot \rangle_2$ is the L_2 inner product $\langle u, v \rangle_2 = \int_{\Omega} uv$. Substituting Equation 3.3 into Equation 3.5 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 α_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 α_i 's directly from the boundary condition.

There are only four arguments that affect how discretization is performed. First, you can select the degree of the polynomials in the approximation space with the argument `degree`. This sets all elements to start with the given degree, even if you use p - or hp -refinement. Second, the integrals of Equation 3.4 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`. Third, in triangles that are owned by a different processor (see Section 3.3.5), the quadrature is performed by the owner and communicated by message passing. 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 a solver other than the (default) hierarchical basis multigrid solver 3.3.4.1, which don't use the unowned triangles. You can avoid the extra computation and communication with `ignore_quad_err=.true.` Finally, you can select how the degree of an edge is determined when the triangles that share that edge have different degrees through `edge_rule`. If you select `MINIMUM_RULE` (the default), then the edge gets the smaller of the two triangle degrees. If you select `MAXIMUM_RULE`, then the edge gets the larger of the two triangle degrees.

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 [28] [30] for 2D, and [6] for 3D). 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. hp -adaptive strategies are explained later in this section.

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 3.3.

DOUBLE_NVERT
DOUBLE_NELEM
DOUBLE_NEQ
HALVE_ERREST
KEEP_NVERT
KEEP_NELEM
KEEP_NEQ
ONE_REF
ONE_REF_HALF_ERRIND

Table 3.3: Permitted values for `refterm` to determine how much refinement to do.

“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`. The number of the specified entities is not doubled exactly. A set of elements is marked for refinement based on the error indicator (Section 3.3.3) and all elements in that set are refined. If the number of entities has not been doubled, the mark/refine process is repeated until the number is at least doubled.

“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. If the corresponding “max_” value is supplied (for example, `max_vert` along with `KEEP_NVERT`), then the attempt is to match the number supplied. Otherwise, the attempt is to match the number of entities currently in the grid, which can cause a drift in the number of entities after several iterations.

`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 estimate larger than reftol/\sqrt{n} are refined, where n is the starting number of elements.

`ONE_REF_HALF_ERRIND` is also a scheme in which no element gets refined more than once in a refinement phase. Those elements with an error indicator larger than half (actually, $1/\text{inc_factor}$) of the maximum error indicator get refined.

A large number of strategies for hp -adaptive refinement have been implemented in PHAML. Selection of which strategy to use is controlled by the argument `hp_strategy`. Some strategies may override the values of some arguments, in particular `refterm`, `derefine`, `inc_factor`, and `error_estimator`.

The strategies are briefly described here. For a more complete description, see [34, 35]. For a comparison of how the different strategies performed on a set of test problems, see [35, 36].

In all of the *hp* strategies, if an element that would be refined by *p* refinement has degree `max_deg`, then it is refined by *h* refinement (unless `stop_on_maxdeg` is `true`), and if an element that would be refined by *h* refinement has level `max_lev`, then it is refined by *p* refinement (unless `stop_on_maxlev` is `true`). If an element has both `max_deg` and `max_lev`, then it is not refined.

The following values are currently allowed for `hp_strategy`:

`HP_APRIORI` is an extension of the second strategy in Ainsworth & Senior [2]. The basic approach is to refine elements by *p* refinement, except elements that contain a known point singularity are refined by *h* refinement. The extension is that we do not limit irregularities to point singularities, and you can specify the strength of that irregularity.

This strategy requires that the user provide a function subroutine that indicates where the solution has an irregularity (i.e. is singular or nonsmooth). See the end of `examples/L-domain/pde.f90` for an example of this subroutine. Also see Section 7.3.8. In theory, this routine should return the largest value of *m* such that the solution is in $H^m(T)$, i.e. the derivative up to order *m* are in L^2 , where *T* is the triangle whose vertices are given as input to the function. For multicomponent solutions, it should return the worst (i.e. smallest) such *m* among the components.

In practice, it can be used to guide refinement in other *a priori* known trouble areas, such as sharp peaks, boundary layers and wave fronts. The actual use is that *p* refinement is performed if the current degree of the triangle is less than the returned value, and *h* refinement is performed otherwise. So, for example, if you know some region contains a boundary layer, you could return 3.1 for any triangle that intersects that region to perform *h*-refinement with cubic elements over the boundary layer.

Several *hp*-adaptive strategies use the same principle as the `HP_APRIORI` strategy, but attempt to estimate the regularity in various ways. `HP_NEXT3P` is an implementation of the first strategy given in Ainsworth & Senior [2]. Three error estimates are computed by the equilibrated residual method with spaces of degree *p* + 1, *p* + 2 and *p* + 3, where *p* is the current degree of the element. If ϕ is the actual error over the element, the error estimates η_{p+q} , *q* = 1, 2, 3, converge to ϕ with error bounds

$$\|\phi\|^2 - \eta_{p+q}^2 \leq C(p+q)^{-2\alpha}.$$

The three error estimates are used to determine the three unknown constants $\|\phi\|$, *C* and α . The regularity is then given by $m = 1 + \alpha$.

`HP_PRIOR2P_E` and `HP_PRIOR2P_H1` are inspired by a strategy of Suli, Houston and Schwab [45]. Two error estimates are computed using the local *p* error estimator with spaces of degree *p* − 1 and *p* − 2. The regularity is estimated by

$$m = 1 - \frac{\log(\eta_{p-1}/\eta_{p-2})}{\log((p-1)/(p-2))}$$

HP_PRIOR2P_E uses the energy norm of the error estimate, whereas HP_PRIOR2P_H1 uses the H^1 norm.

HP_TYPEPARAM selects the type parameter strategy of Gui and Babuška [20]. Perceived smoothness of the solution over element t is given by

$$R(t) = \begin{cases} \frac{e(t,p)}{e(t,p-1)} & e(t,p-1) \neq 0 \\ 0 & e(t,p-1) = 0 \end{cases}$$

where p is the degree of the element and e is the error indicator for the given degree. The type parameter, γ , determines the type of the element. If $R(t) > \gamma$ then t is of h -type and will be h refined; otherwise it is p -type and will be p refined. γ is specified with `tp_gamma`.

Another way of determining if the solution is smooth is to compare the local *a posteriori* error estimate with a prediction of what the error estimate should be, after refinement, under the assumption that the solution is smooth and has the proper asymptotic convergence rate [27]. If η_E is the error estimate on element E , and E' is a child of E after refinement, then the predicted error indicator for E' is

$$\eta_{E'}^{\text{pred}} = \begin{cases} \frac{1}{2} \gamma_h \left(\frac{1}{\sqrt{2}} \right)^{p_E} \eta_E & \text{if } E \text{ was } h \text{ refined} \\ \sqrt{\gamma_p} \eta_E & \text{if } E \text{ was } p \text{ refined} \end{cases}$$

where p_E is the degree of element E , and γ_h and γ_p are control parameters given in `sp_gamma_h` and `sp_gamma_p`. Then, if E' is selected for refinement, it is refined by h -refinement if $\eta_{E'} > \eta_{E'}^{\text{pred}}$, and by p -refinement otherwise. This strategy is selected with HP_SMOOTH_PRED.

HP_BIGGER_ERRIND selects a strategy in which both the LOCAL_PROBLEM_H and LOCAL_PROBLEM_P error indicators are computed. It is similar to the strategy for 1D problems in [40]. The error indicator specified by `error_estimator` is used to determine which elements will be refined (see 3.3.3). For each element that is selected for refinement, it is h refined if LOCAL_PROBLEM_H gives the larger error indicator, and p refined if LOCAL_PROBLEM_P is larger. The premise is that the local problem error indicators approximate how much the solution will change if the refinement is performed, and one should perform the type of refinement that will cause the largest change, and hence reduce the error the most. **Experiments have indicated that it is often beneficial to bias the refinement towards p -refinement, which can be done by setting `pbias` to a number greater than 1. We have found that generally `pbias=2` works best if the solution is singular or has exceptionally strong features, and `pbias=4` is best for smooth solutions or moderately strong features.**

The Texas 3 Step strategy [37] is selected by HP_T3S. The three steps are 1) uniform h refinement, 2) adaptive h refinement, and 3) adaptive p refinement. Steps 2 and 3 are repeated until a termination criterion is met. The first step is intended to create a starting grid for which the rate of convergence has reached the asymptotic region. The number of uniform refinements performed is controlled by `t3s_nunif`, which is 0 by default. If the method is performing poorly at the start, try increasing it.

Steps 2 and 3 attempt to reduce the error by a prescribed amount. Specifically, if the error estimate at the beginning of step 2 is θ , then the adaptive h refinement attempts to reduce it to $\gamma\eta\theta$, and the adaptive p refinement attempts to reduce it to $\eta\theta$. The parameters γ and η are specified by `t3s_gamma` and `t3s_eta`, and currently default to 6 and 0.1. This is still experimental and may change. In each step a formula is used to determine how much to refine each element. The regularity of the solution is used in this formula, and is specified by subroutine `regularity` the same as the `HP_APRIORI` strategy. To avoid overrefining in one step, an upper bound is placed on the number of times an element can be refined in one step. These bounds are given by `t3s_maxref` for the h refinement of step 2, and `t3s_maxdeginc` for the p refinement of step 3. Currently they both default to 3, but this is still experimental and may change. Currently derefinement is not allowed with this strategy. You should use `derefine=.false.` to avoid a warning message.

`HP_ALTERNATE` gives another strategy that alternates between h and p adaptive refinement. It is similar to `HP_T3S`, but instead of using a formula to estimate the correct amount of refinement of each element, it simply performs adaptive refinement until the error estimate reaches the target value of $\gamma\eta\theta$ for the h refinement step and $\eta\theta$ for the p refinement step. γ and η are again specified with `t3s_gamma` and `t3s_eta`. Unlike the Texas 3 Step strategy, it does not begin with uniform refinements.

`HP_NLP` attempts to optimize the grid by solving a nonlinear programming optimization problem. It is based on a method proposed by Patra and Gupta [38]. In essence, the optimization problem says to determine a number of h -refinements and p -refinements of each element to minimize the number of degrees of freedom subject to the predicted error estimate being less than one fourth of the current error estimate, plus other constraints that keep the number of refinements within feasible bounds, such as the degree being positive. Among these constraints are use-specified limits on how much change can occur in one refinement stage. The parameters `nlp_max_h_dec`, `nlp_max_p_dec`, `nlp_max_h_inc`, and `nlp_max_p_inc` specify the maximum amount of h and p coarsening and refinement to use at once. The defaults are 1, 1, 5 and 2. Solution of the nonlinear program takes a long time, except for very coarse grids, so we do not encourage the use of this strategy. This strategy requires the use of a third party optimization routine which is not included in the PHAML distribution. If you feel the need to use this strategy, contact phaml@nist.gov for instructions on how to obtain and configure this routine. This strategy cannot be used in parallel, because the optimization routine is not parallel.

A strategy for 1D problems based on the coefficients of the Legendre polynomial expansion of the solution was given in [26]. If p refinement is optimal, then the coefficients should decrease exponentially. A least squares fit of the last four coefficients is made to $ce^{-\sigma p_i}$, where p_i are the degrees of the last four Legendre polynomials. If $\sigma > 1$ then p refinement is performed, otherwise h refinement. To extend this to 2D, we use the coefficients of the p -hierarchical basis. The i^{th} "coefficient" is given by the sum of the absolute values of all coefficients of bases of degree i . This strategy is selected with `HP_COEF_DECAY`.

Another strategy based on Legendre coefficients is given in [21]. This strategy, selected with `HP_COEF_ROOT`, uses a “root test” on the coefficients to estimate the regularity of the solution. p refinement is used where the solution is sufficiently smooth, and h refinement elsewhere. Again, PHAML uses the coefficients of the p -hierarchical basis instead of the Legendre coefficients.

A strategy based on that of Šolín et al. [43] is implemented as `HP_REFSOLN_ELEM`. This strategy has its own overall structure to the adaptive refinement algorithm, so most of the refinement-related parameters to PHAML have no effect, including the error indicator. Also note that the global energy norm error estimate is actually a relative H^1 norm error estimate (which is equivalent), and the L_∞ and L_2 error estimates are not available. However, all the usual error estimates are available through `phaml_query`. The method begins by computing a reference solution, u_{ref} on a grid that has been globally refined by both h and p . The error indicator for element E is computed as $\|u_{\text{ref}} - u_T\|_{H^1(E)}$. Elements for which the error indicator is larger than the largest error indicator divided by `inc_factor` are selected for refinement. For each element to be refined, 11 candidate refinements are considered: p -refine to degree $p+1$, p -refine to degree $p+2$, and h -refine with the child degrees being all combinations of p_0 , p_0+1 and p_0+2 where p is the current degree of the element and $p_0 = \lfloor (p+1)/\sqrt{2} \rfloor$. For each candidate, the H^1 projection of the reference solution onto the corresponding space, $\Pi_{\text{candidate}}$, is computed, and the projection error is computed as $\|u_{\text{ref}} - \Pi_{\text{candidate}}\|_{H^1(E)}$. The projection error onto the unrefined space is also computed. The winning candidate refinement is selected based on the amount by which the logarithm of the projection error is reduced relative to the increase in the number of degrees of freedom. Experiments have indicated that it is often beneficial to bias the refinement towards p -refinement, which can be done by setting `pbias refsoln.pbias` to a number greater than 1. We have found that generally `pbias=2 refsoln.pbias=2` works best if the solution is singular or has exceptionally strong features, and `pbias=4 refsoln.pbias=4` is best for smooth solutions or moderately strong features. `HP_REFSOLN_ELEM` has not yet been implemented in parallel, so it currently requires `nproc=1`.

Another strategy based on using a reference solution is that of Demkowicz [12]. `HP_REFSOLN_EDGE` is based on this strategy. It is similar to `HP_REFSOLN_ELEM`, but one first considers the refinement of edges by projecting the reference solution onto spaces corresponding to candidate refinements of the edge. Once it is determined how to refine the edge, the refinement of the element is determined by consistency of the h -refinement of the edge and the minimum edge rule for p . This method has not been implemented in parallel yet, so it currently requires `nproc=1`.

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 six error indicators. The error indicator is selected by the argument

`error_estimator` which takes one of the values `EXPLICIT_ERRIND`, `LOCAL_PROBLEM_H`, `LOCAL_PROBLEM_P`, `EQUILIBRATED_RESIDUAL`, `HIERARCHICAL_COEFFICIENT` or `TRUE_DIFF` to select one of the error indicators. It can also take the value `INITIAL_CONDITION`.

Many error indicators are based on the interior residual within elements

$$r = f - \mathcal{L}u_T \quad (3.6)$$

and the boundary residual on element edges

$$R = \begin{cases} g - \mathcal{B}u_T & \text{if the edge is on } \partial\Omega_N \\ 0 & \text{if the edge is on } \partial\Omega_D \\ -[\frac{\partial u_T}{\partial n}] & \text{if the edge is interior} \end{cases} \quad (3.7)$$

where \mathcal{L} is the operator defined in Equation 1.1, \mathcal{B} is the operator defined in Equation 1.3, f , g , $\partial\Omega_N$, and $\partial\Omega_D$ are defined in Equations 1.1-1.3, u_T is the approximate solution, and $[\frac{\partial u_T}{\partial n}]$ is the jump in the normal derivative across the element boundary. See [1] for a more detailed treatment of error indicators.

`EXPLICIT_ERRIND` selects the explicit error indicator defined in Chapter 2 of [1], with slight modifications for the more general operator and degree-based weights for high order elements as given in [27]. The error indicator is based on the norm of the residual. The explicit error indicator, η_e , for an element e is given by

$$\eta_e^2 = \frac{h_e^2}{p^2} \|r\|_{L_2(e)}^2 + \frac{h_e}{2p} \|R\|_{L_2(\partial e)} \quad (3.8)$$

where h_e is the longest edge length of the element and p is the degree of the element. The error indicator is used to guide adaptive refinement. The energy norm error estimate is given by an unknown constant times the square root of the sum of the squares of the error indicators. For the L_2 error estimate, the powers on the h 's are increased by 2. For the L_∞ error estimate, the L_∞ norm of the residuals is used and the maximum error indicator gives the error estimate. In PHAML the unknown constant has been chosen to be 1/20 for the energy norm and 1/100 for the L_2 and L_∞ norms based on the results obtained with the example and test problems, although these constants work better with high order or hp -adaptive elements than linear elements. The integrals for the energy and L_2 norms are approximated with a 4th order quadrature rule, and the L_∞ norm is approximated by values at the quadrature points of the same quadrature rule. The explicit error indicator is quite efficient to compute and generally provides good guidance for adaptive refinement, but one cannot be certain of the accuracy of the error estimates.

`LOCAL_PROBLEM_H` and `LOCAL_PROBLEM_P` are based on solving a small local problem. Equation 1.1 is modified by replacing f with the interior residual r . 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 using the boundary residual R . Since the refinement used for the error indicator is exactly what would occur if the triangle(s) was (were) chosen for refinement by h - or p -refinement, this gives an estimate of how much the solution would change if this triangle(s) was (were) refined. The local problem error indicators are very accurate, but relatively expensive, because of the computation required to set up and solve the elemental matrices for each triangle.

The `EQUILIBRATED_RESIDUAL` error indicator is similar to the `LOCAL_PROBLEM_P` indicator, except that the residual is equilibrated so that the boundary fluxes are in equilibrium with the interior residual loads (see Chapter 6 in [1]). This insures that the discrete Neumann problem has a solution, which in turn allows the use of the full local approximation space of degree p , whereas the `LOCAL_PROBLEM_P` uses a reduced space spanned by the bases of exact degree p . It can be shown that this method produces an error estimate that is a guaranteed upper bound on the actual energy norm of the error, if the local problem was solved exactly. Of course, the local problem is solved approximately so the guarantee does not hold, but the PHAML examples typically find the ratio of error estimate to actual error to be about 1.25. This is the most accurate and stable error indicator in PHAML, but also the most computationally expensive.

A less expensive approach, given by `HIERARCHICAL_COEFFICIENT`, is to simply examine the hierarchical coefficients (see [28]) of u_T over each triangle, which is similar to computing the local problem indicator one refinement back. For linear elements, the coefficient of the h -hierarchical basis is used. For higher order elements, the coefficients of the p -hierarchical bases are used. This indicator can provide a very inexpensive way to guide adaptive refinement, but is not likely to give an accurate estimate of the norm of the error.

The `TRUE_DIFF` error indicator uses the difference between the true solution (if given, see 3.2.4) and u_T . The energy and L^2 norms are estimated with a sixth order quadrature rule, and the L^∞ norm is estimated using the same quadrature points. This is primarily available for comparison with new error indicators and would not normally be used.

The `INITIAL_CONDITION` estimate uses the difference between the function given in subroutine `iconds` and u_T . Normally this is used to define an initial grid for time dependent or nonlinear problems (see Section 4.3), but it can also be used to provide a function that indicates where you believe the grid should be finer.

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 as the primary linear system solver, along with two basic Krylov space solvers, conjugate gradients and GMRES. It also

contains hooks into several freely available software packages of direct and iterative parallel solvers for comparative studies and for situations where the native solvers are not sufficient. 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 default solver is the hierarchical basis multigrid solver (HBMG), which can be explicitly selected with `solver = MG_SOLVER`. (Note: this solver must NOT be used if the linear system is not symmetric (i.e., c_{xy} , c_x or c_y from Equation 1.1 is nonzero, or a coefficient matrix in a coupled system of equations is nonsymmetric) or indefinite (i.e., c_u is negative with sufficiently large magnitude).) It combines the ideas of an hp -multigrid method [33] with the h -hierarchical basis multigrid method for linear elements [29]. The equations corresponding to high order face basis functions 3.3.1 are first removed by static condensation [50]. A p -multigrid cycle is applied to the equations corresponding to vertex and edge basis functions as follows. The cycle is like a normal multigrid V-cycle, except the levels are given by the polynomial degree. Some number of Gauss-Seidel iterations are performed on all equations up to degree p_{\max} where p_{\max} is the maximum degree. Then the Gauss-Seidel iterations are applied to all equations up to degree $p_{\max} - 1$, then $p_{\max} - 2$, etc., until only the linear and quadratic equations are used. With the p -hierarchical basis, the “intergrid transfers” are automatic. Then the “coarse grid” equations (equations corresponding to the vertices, i.e. linear elements) are solved “exactly” by using a standard h -multigrid method. In PHAML, the h -hierarchical basis multigrid method is used, as follows. 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 half iteration is allowed, which means relaxing the red equations but not the black. 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. After the finest level is reached, the second half of the p -multigrid cycle performs relaxations on all equations up to degree 2, then up to degree 3, etc. This constitutes one multigrid V-cycle. Cycles are repeated until some termination criterion is met. Finally, the equations corresponding to face basis functions are solved directly.

There are several arguments to `phaml_solve_pde` that determine the specifics of the multigrid algorithm. `mg_prerelax_ho` and `mg_postrelax_ho` determine the number of Gauss-Seidel iterations to perform on each level of the p -multigrid

cycle, before and after solving the linear basis equations. `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 linear equations. 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 ℓ^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.

The HBMG as implemented in PHAML uses the full domain partition approach to parallelism by default [31]. This approach allows the use of only two messages in each h -multigrid cycle, one at the coarsest grid and one at the finest grid, with a (usually very minor) reduction in the convergence rate of the h -hierarchical basis multigrid method, but still uses communication after each level of the p -multigrid cycle. PHAML also provides the option of using conventional parallelism with messages on each h -level, which gives exactly the same results as the sequential HBMG. This is selected by specifying `mg_comm = MGCOMM_CONVENTIONAL`.

3.3.4.2 Krylov space solvers

PHAML also contains two Krylov space solvers: conjugate gradients and GMRES. For a description of these methods, see [9]. The methods are selected by setting `solver` to be `CG_SOLVER` and `GMRES_SOLVER`, respectively. As with the HBMG method, PHAML begins by eliminating the equations associated with face basis function by static condensation. It also eliminates the equations associated with Dirichlet boundary conditions so that the working matrix is symmetric.

Either method can be used without preconditioning via `preconditioner=NO_PRECONDITION` or with HBMG as the preconditioner via `preconditioner=MG_PRECONDITION`. As a preconditioner, HBMG does not need to converge to the solution, just give an approximation. It is usually sufficient to use two iterations, so the default HBMG parameters are set appropriately when HBMG is used as a preconditioner instead of a solver. Also, the HBMG preconditioner uses, by default, `ignore_quad_err=.true.` to avoid reducing the larger quadrature errors in un-owned elements, and `mg_comm = MGCOMM_NONE` to skip the communication steps in HBMG.

There are three parameters that control the Krylov space solvers. `krylov_iter`

provides an upper bound on the number of iterations allowed, and `krylov_tol` gives a tolerance on the ℓ^2 norm of the residual. Convergence is declared when the residual is less than the tolerance. If the maximum number of iterations is achieved before convergence, the solution is accepted and a warning is printed. `krylov_restart` gives the number of restart vectors for GMRES.

3.3.4.3 Alternative direct solvers

3.3.4.3.1 LAPACK

The LAPACK library [5] 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.

3.3.4.3.2 MUMPS

MUMPS [3][4] is a parallel direct solver from the European project PARASOL. It is accessed through PETSc, so PETSc must also be installed and be configured for MUMPS support. MUMPS is specified with `solver=PETSC_MUMPS_SPD_SOLVER` for the symmetric positive definite solver, or `solver=PETSC_MUMPS_GEN_SOLVER` for the nonsymmetric solver.

3.3.4.3.3 SuperLU

SuperLU [25] is a parallel direct solver from the Lawrence Berkley National Laboratories. It is accessed through PETSc, so PETSc must also be installed and be configured for SuperLU support. SuperLU is specified with `solver=PETSC_SUPERLU_SOLVER`.

3.3.4.4 Alternative iterative solvers

3.3.4.4.1 PETSc

PETSc [7][8] 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 3.4 contains the available PETSc solvers as specified by `solver`. Table 3.5 contains the available preconditioners for the PETSc solvers specified by `preconditioner`. The 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 3.6. Refer to the PETSc User's Manual [8] for explanations of these arguments.

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 3.4: Available values of **solver** for the PETSc solvers.

NO_PRECONDITION	no preconditioning
MG_PRECONDITION	one hierarchical basis multigrid V-cycle
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
PETSC_GAMG_PRECONDITION	algebraic multigrid
PETSC_TRILINOS_ML_PRECONDITION	algebraic multigrid package ML from Trilinos

Table 3.5: Available values of **precondition** for the PETSc solvers.

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_dtol
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 3.6: Arguments that are passed to the PETSc solvers.

HYPRE_BOOMERANG_SOLVER	algebraic multigrid
HYPRE_PCG_SOLVER	preconditioned conjugate gradients
HYPRE_GMRES_SOLVER	generalized minimal residual

Table 3.7: Available values of solver for the *hypre* solvers.

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.`

If you use PETSc, you might have to make some changes to PHAML depending on what version of PETSc you have. See `mkmkfile.sh`, `petsc_init.F90`, and `petsc_interf.F90` and search for “before” to see if you need to make changes.

3.3.4.4.2 *hypre*

hypre [15][16] is a package of parallel iterative solvers and preconditioners from

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

Table 3.8: Available values of precondition for the *hypre* solvers.

integer	hypr_BoomerAMG_MaxLevels
integer	hypr_BoomerAMG_MaxIter
real(my_real)	hypr_BoomerAMG_Tol
real(my_real)	hypr_BoomerAMG_StrongThreshold
real(my_real)	hypr_BoomerAMG_MaxRowSum
integer	hypr_BoomerAMG_CoarsenType
integer	hypr_BoomerAMG_MeasureType
integer	hypr_BoomerAMG_CycleType
integer	hypr_BoomerAMG_NumGridSweeps(:)
integer	hypr_BoomerAMG_GridRelaxType(:)
integer	hypr_BoomerAMG_GridRelaxPoints(:, :)
real(my_real)	hypr_BoomerAMG_RelaxWeight(:)
integer	hypr_BoomerAMG_DebugFlag
real(my_real)	hypr_ParaSails_thresh
integer	hypr_ParaSails_nlevels
real(my_real)	hypr_ParaSails_filter
integer	hypr_ParaSails_sym
real(my_real)	hypr_ParaSails_loadbal
integer	hypr_ParaSails_reuse
integer	hypr_ParaSails_logging
real(my_real)	hypr_PCG_Tol
integer	hypr_PCG_MaxIter
integer	hypr_PCG_TwoNorm
integer	hypr_PCG_RelChange
integer	hypr_PCG_Logging
integer	hypr_GMRES_KDim
real(my_real)	hypr_GMRES_Tol
integer	hypr_GMRES_MaxIter
integer	hypr_GMRES_Logging

Table 3.9: Arguments that are passed to the *hypr* solvers.

Lawrence Livermore National Laboratories. It is accessed through PETSc, so PETSc must also be installed and be configured for *hypr* support. The four preconditioners from *hypr* that are available through PETSc are BoomerAMG, ParaSails, Pilut and Euclid. They are used in conjunction with a PETSc solver from Table 3.4. They are selected with `preconditioner = PETSC_HYPRE_XXX_PRECONDITION`, with `xxx` being `BOOMERAMG`, `PARASAILS`, `PILUT` or `EUCLID`. BoomerAMG can also be used as a solver via `solver = PETSC_HYPRE_BOOMERAMG_SOLVER`. There are several arguments that are passed to the *hypr* methods listed in Table 3.10. See the *hypr* reference manual (available at the *hypr* web site) for an explanation of these.

The *hypr* solvers are listed in Table 3.7 and preconditioners are listed in Table 3.8. 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 to the *hypr* methods listed in Table 3.9. See the *hypr* user's guide (distributed with the software) for an explanation of these.

If you use *hypr*, you might have to make some changes to PHAML depending on what version of *hypr* you have. See `hypr_fix.c` (instructions are at the beginning of the file) and `mkmkfile.sh` (search for “hypr version”) to see if you need to make changes.

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

real(my_real)	hypr_BoomerAMG_tol
real(my_real)	hypr_BoomerAMG_trunc_factor
real(my_real)	hypr_BoomerAMG_strong_threshold
real(my_real)	hypr_BoomerAMG_max_row_sum
real(my_real)	hypr_BoomerAMG_relax_weight
real(my_real)	hypr_BoomerAMG_outer_relax_weight
integer	hypr_BoomerAMG_cycle_type
integer	hypr_BoomerAMG_max_levels
integer	hypr_BoomerAMG_max_iter
integer	hypr_BoomerAMG_P_max_elmts
integer	hypr_BoomerAMG_agg_num_levels
integer	hypr_BoomerAMG_agg_num_paths
integer	hypr_BoomerAMG_num_sweeps_all
integer	hypr_BoomerAMG_num_sweeps_down
integer	hypr_BoomerAMG_num_sweeps_up
integer	hypr_BoomerAMG_num_sweeps_coarsest
integer	hypr_BoomerAMG_relax_type_all
integer	hypr_BoomerAMG_relax_type_down
integer	hypr_BoomerAMG_relax_type_up
integer	hypr_BoomerAMG_relax_type_coarsest
integer	hypr_BoomerAMG_relax_order
integer	hypr_BoomerAMG_measure_type
integer	hypr_BoomerAMG_coarsen_type
integer	hypr_BoomerAMG_interp_type
logical	hypr_BoomerAMG_print_statistics
logical	hypr_BoomerAMG_print_debug
real(my_real)	hypr_Pilut_tol
integer	hypr_Pilut_maxiter
integer	hypr_Pilut_factorrowsize
integer	hypr_Euclid_levels
integer	hypr_Euclid_bj
logical	hypr_Euclid_print_statistics
real(my_real)	hypr_ParaSails_thresh
real(my_real)	hypr_ParaSails_filter
real(my_real)	hypr_ParaSails_loadbal
integer	hypr_ParaSails_nlevels
integer	hypr_ParaSails_sym
logical	hypr_ParaSails_logging
logical	hypr_ParaSails_reuse

Table 3.10: Arguments that are passed to the *hypr* solvers and preconditioners.

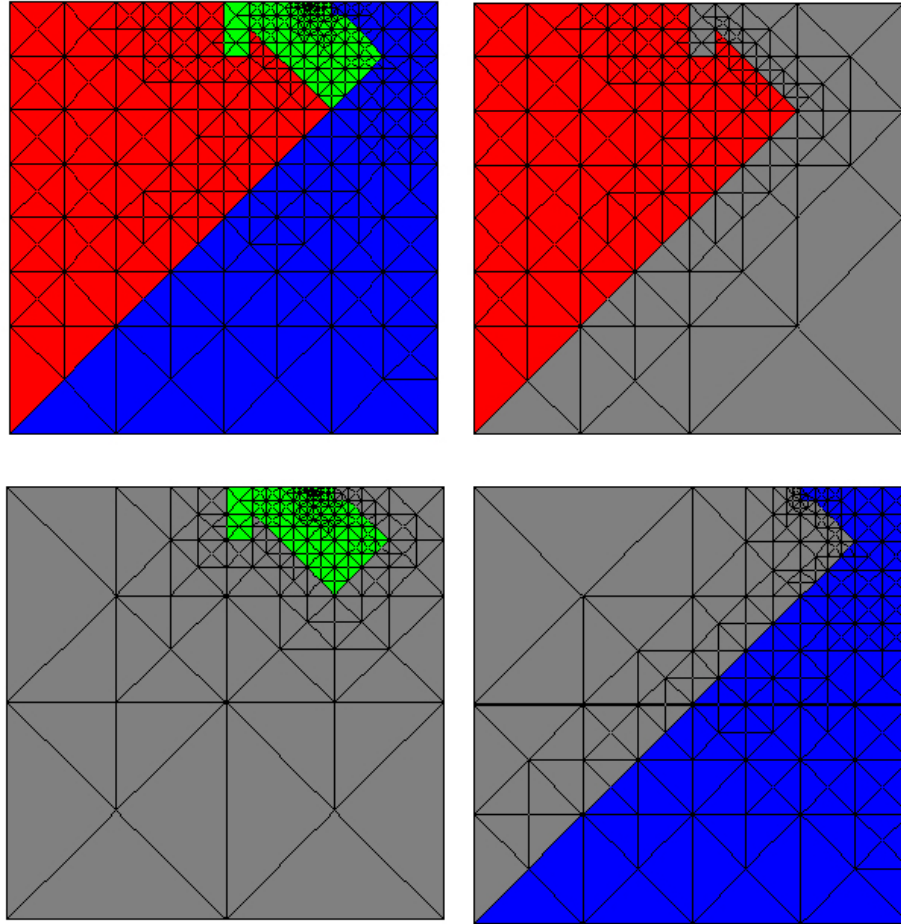


Figure 3.5: An adaptive grid partitioned for 3 processors, and the grid seen by each of the processors.

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
ZOLTAN_FILE	read Zoltan method and parameters from a file

Table 3.11: Available values of `partition_method` for Zoltan partitioners.

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) [30] in which each process also has additional coarse elements that cover the unowned part of the domain. Figure 3.5 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. The number of entities (elements, vertices or equations) that are associated with an element is used as the weight for the element in the partitioning algorithms. For balancing after refinement, it divides up the number of entities as equally as possible. For balancing before refinement, it estimates the number of entities that will be associated after refinement. If `refterm` is not `ONE_REF` or `ONE_REF_HALF_ERRIND`, it includes the error indicator in the weight, 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 [32]. 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 [13][14]. The values for selecting methods from Zoltan are given in Table 3.11. See the Zoltan User's Guide for descriptions of these methods, except `ZOLTAN_FILE`.

`ZOLTAN_FILE` lets you specify the method to use in Zoltan, and also to specify other Zoltan parameters, in a file. Basically, any parameter that can be set with `Zoltan_Set_Param` (see the Zoltan User's Guide) can be set by including a line containing the parameter name and value. In particular, you can specify the method with, for example, `LB_METHOD RCB`. For a full description of this file, see the zoltanParams web page [48]. The Zoltan parameter file is specified by the `phaml_solve_pde` argument `zoltan_param_file`, with a default value of `"zoltan.params"`.

DRUM (the Dynamic Resource Utilization Model) [17][47] supports resource-aware, large-scale scientific computing in heterogeneous and hierarchical parallel computing environments. It can improve the load balance on, for example, clusters containing different kinds of processors. Within PHAML, DRUM is accessed with the partitioning method `ZOLTAN_FILE`. The parameters for DRUM are specified in the Zoltan parameter file. See the DRUM documentation for a description of the DRUM parameters.

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 by default these are not termination criteria – if the refinement of an element would exceed the given value, then it is simply not performed. To use these as termination criteria, use the logical arguments `stop_on_maxlev` and

`stop_on_maxdeg`.

The other category is to terminate when an error estimate is small enough. The global error estimate is computed from the local error indicator. The estimate can be an estimate of the energy norm, L^∞ norm, or L^2 norm of the error. The arguments `term_energy_err`, `term_Linf_err`, and `term_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 7.4.11) 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 7.4.10) 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_solver_when` and `print_solve_who` provide for printing a summary of the solver, such as the number of iterations used, the norm of the residual, 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^∞ 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^∞ 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 `ERREST`.

The norms of the error and error estimates can be either absolute error (the default) or relative error. This is controlled by `errtype` which takes the values `ABSOLUTE_ERROR` and `RELATIVE_ERROR`. If the relative error is selected, then the printed norm of the error is divided by the norm of the true solution, and the printed error estimate is divided by the norm of the computed solution.

`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 multigrid, conjugate gradients, or GMRES method or one of the solvers from PETSc. If it has the value `FREQUENTLY` then the ℓ^2 norm of the residual is printed after each iteration. For GMRES, it is printed at the restarts. 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

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	⇒
component scale	⇒
space filling curve	⇒
grid offset	⇒
crop (debug window)	
toggle axes	
toggle color key	
write postscript	⇒

Table 3.12: The main graphics menu.

`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 3.12. 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.

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 3.6 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 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 3.7 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** \Rightarrow **levels** to create a hierarchy of grids based on refinement level.

Figure 3.8 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

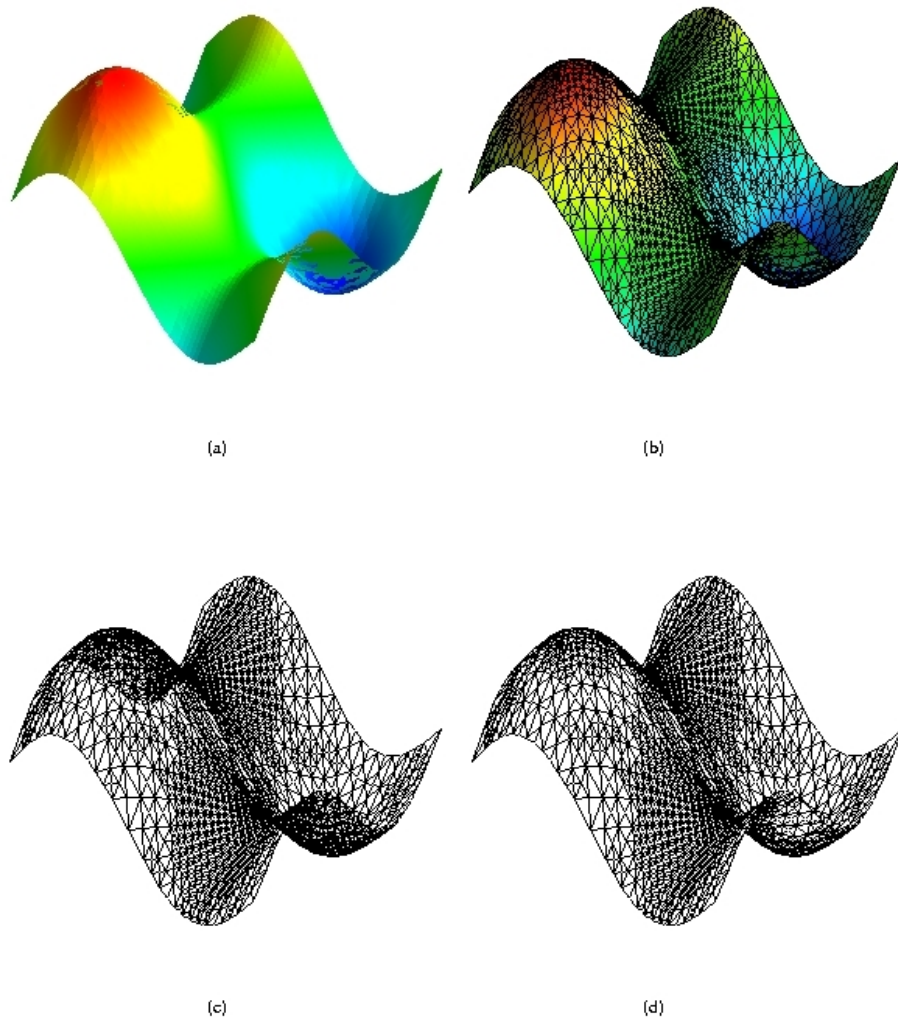
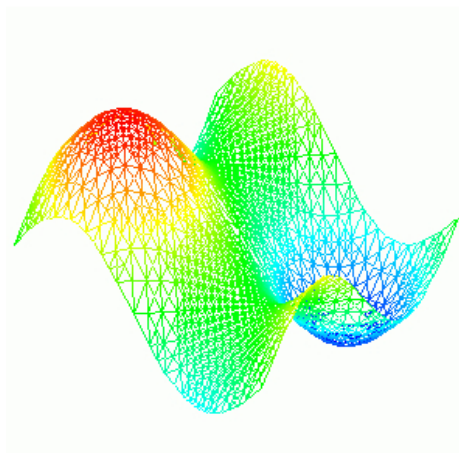
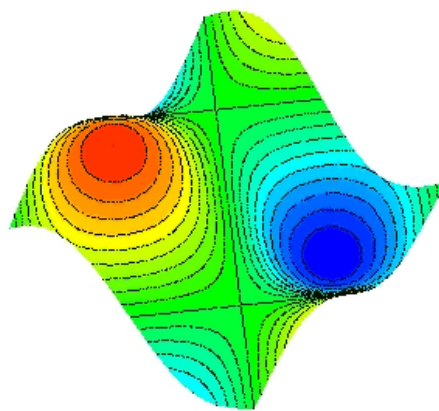


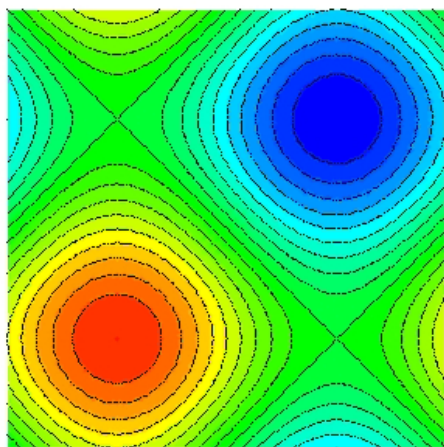
Figure 3.6: Visualizations of the solution.



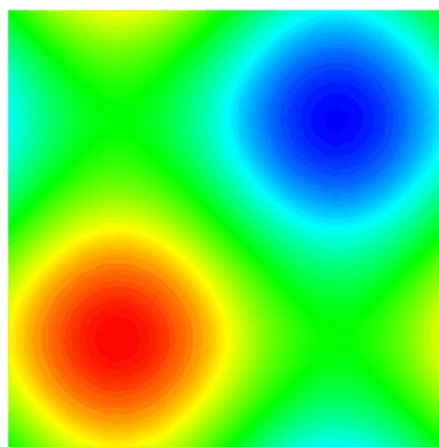
(e)



(f)



(g)



(h)

Figure 3.6: (continued) Visualizations of the solution.

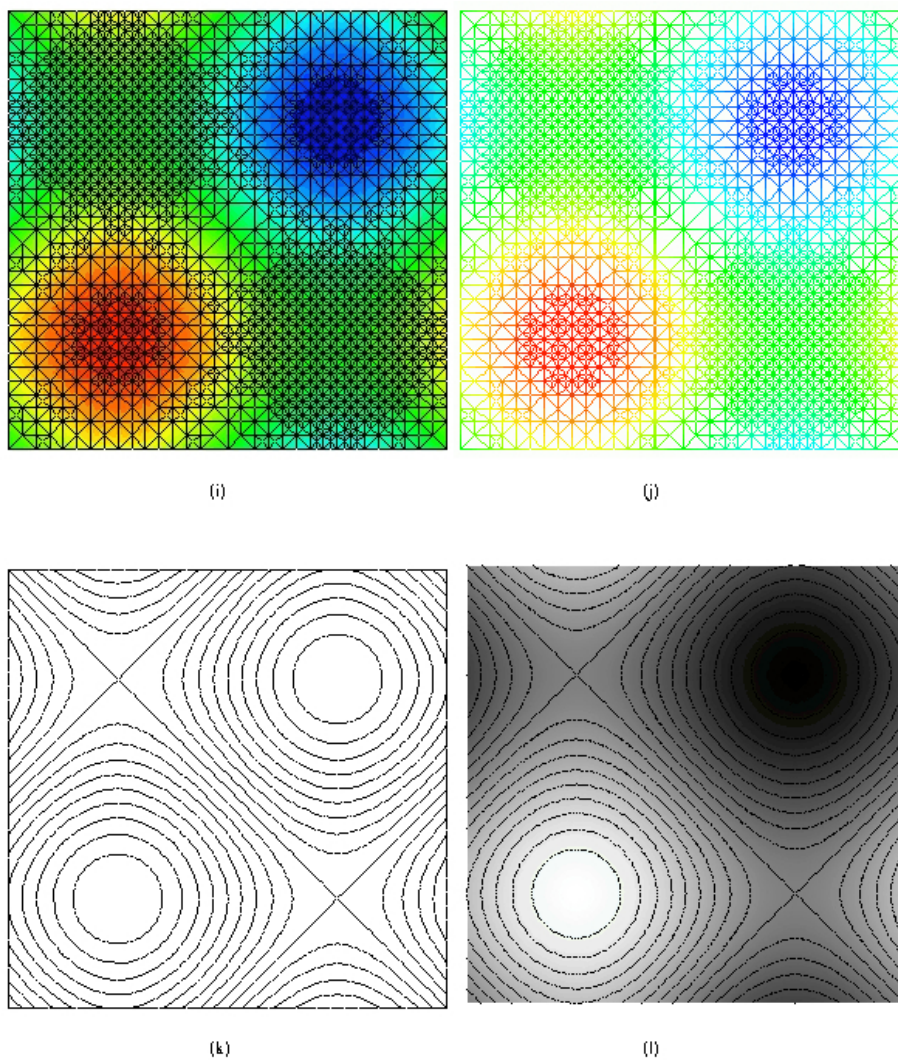


Figure 3.6: (continued) Visualizations of the solution.

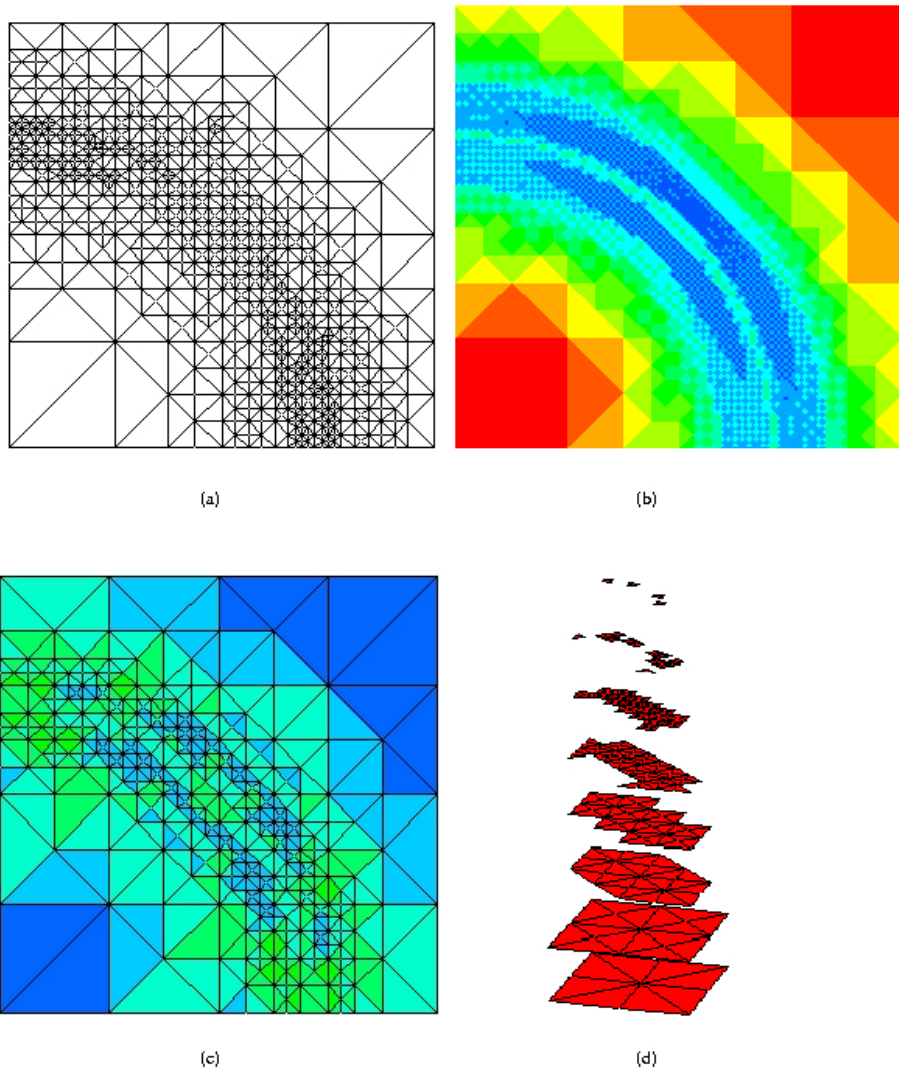


Figure 3.7: Visualizations of the grid.

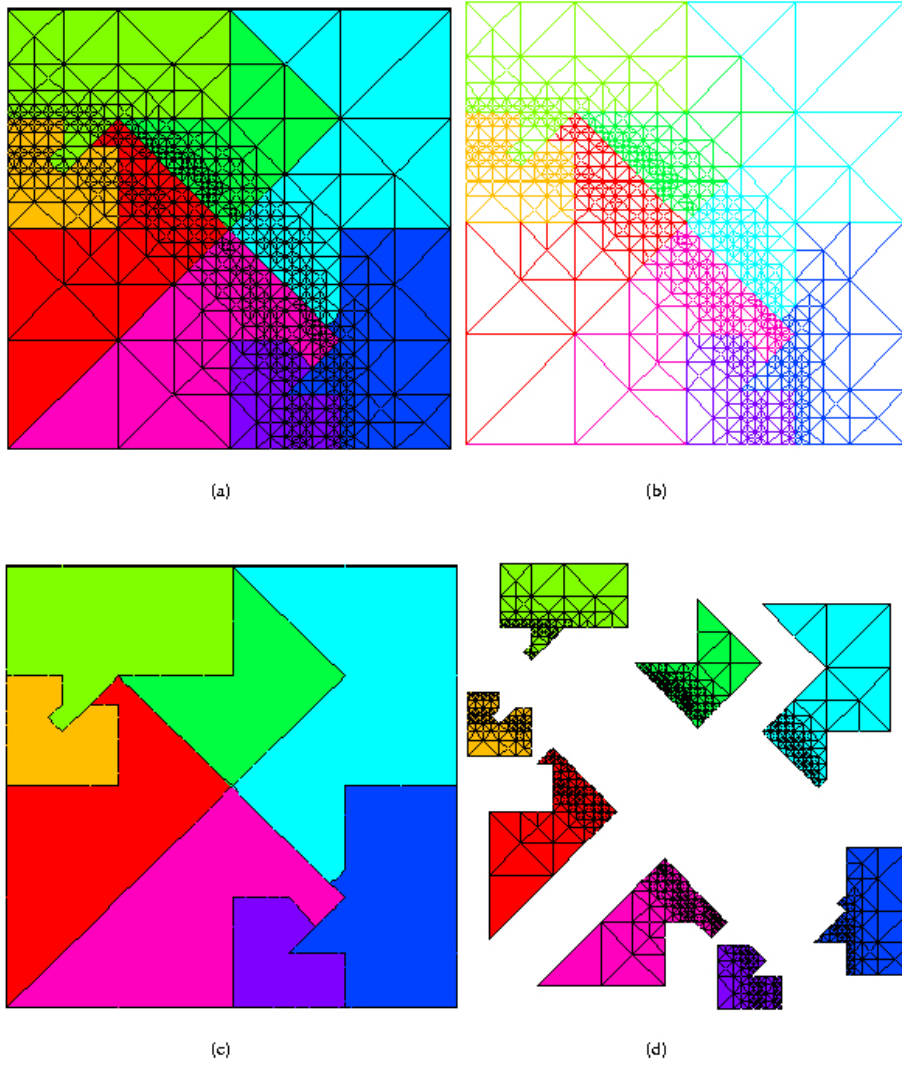


Figure 3.8: Visualizations of the partitions of the grid for load balancing.

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

Table 3.13: The view modifier submenu.

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 3.13.

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 3.8(d)), and **move light** (see Section 3.5.6).

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

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

Table 3.14: The element edge color submenu.

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 blue to red with blue assigned to the first item.

There are four other color schemes available. The color scheme is selected in the submenu under **color scheme** which has the selections **rainbow**, **double rainbow**, **gray scale**, **striped**, and **stepped sequential**. The double rainbow passes through the colors twice, with the second time a little more pastel. This is useful when discrete colors need more separation, such as when coloring by owner and there are a lot of processors, or when coloring by element degree with an hp -adaptive grid. 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. The stepped sequential scheme uses several steps of saturation/value within each of several hues ranging from red to violet. The number of steps and number of hues can be changed by use of a submenu. By default, there are 4 steps of 6 hues for a total of 24 colors. This scheme is particularly useful for discrete values such as owner and degree.

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 3.14 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 **true**s. 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 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.

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 3.7(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 indicator**. 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.

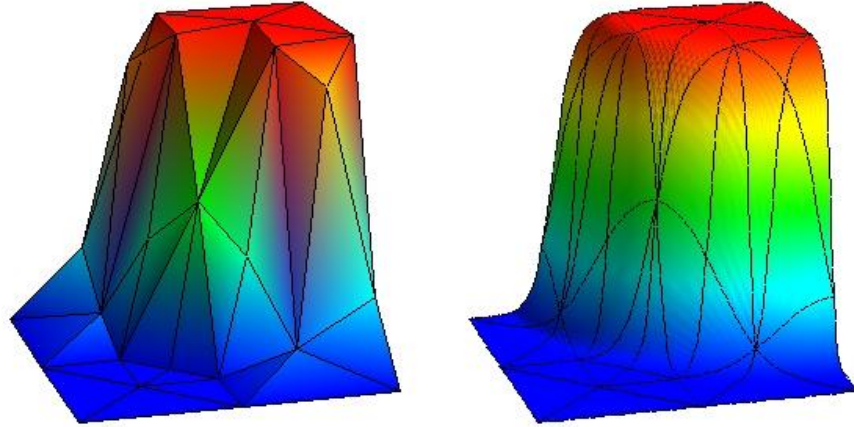


Figure 3.9: Improvement of a surface visualization by using subelement resolution.

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 multi-component 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 $u_1^2 + u_2^2 + \dots$.

component scale provides the selection of how to scale the components when the function is computed solution, true solution or error. Normally the function is scaled by its maximum absolute value. For multicomponent solutions, you can choose to have each component scaled by its own maximum absolute value by selecting **individual**, or you can choose to have all components scaled the same using the maximum absolute value over all components by selecting **all the same**.

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

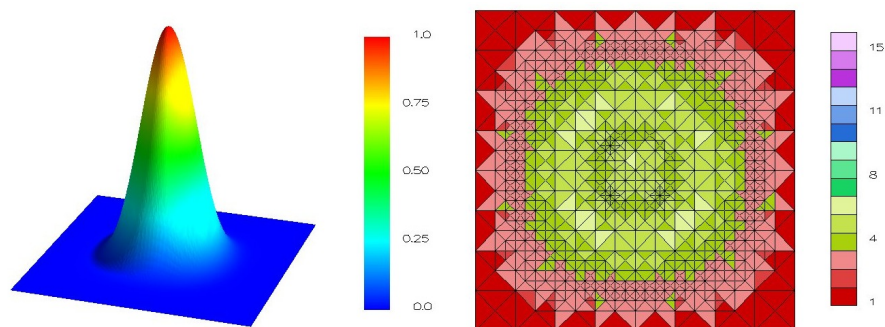


Figure 3.10: Examples of the color key.

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 ℓ results in drawing 4^ℓ triangles in each element. The effect of using a high level of subelement resolution can be seen in Figure 3.9.

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.

A color key can be added and removed with the **toggle color key** entry. The key gives the correspondence between the color of the interior of the elements and numeric values. Figure 3.10 illustrates two uses of the color key: a continuous color key for the function value using the rainbow color scheme, and a discrete color key showing the element degree using the stepped sequential color scheme.

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 `renderX.eps` where `X` is 0 for the master's graphics processor or the slave number for the slaves' graphics processors (note there cannot be more than 9 slaves for this to work properly). 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 graphical 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 Store Matrix

One might wish to store the linear system that represents the discretized PDE in a file for processing external to PHAML, for example solving the linear system with Matlab. Subroutine `phaml_store_matrix` does this. This routine stores the discretized PDE in `phaml_solution` so that it can be processed by an external program. In addition to a `phaml_solution_type` variable, it takes four optional arguments, `stiffness_unit`, `rhs_unit`, `mass_unit`, and `inc_quad_order`. `inc_quad_order` increases the order of the quadrature rule used for computing the matrix and right hand side entries, the same as in `phaml_solve_pde`.

For an elliptic boundary value problem, the discrete problem is a linear system of equations, $Ax = b$. `phaml_store_matrix` can store the stiffness matrix, A , and/or the right hand side, b . The presence of the optional arguments `stiffness_unit` and `rhs_unit` determines whether or not the stiffness matrix and right hand side will be stored, respectively. Either or both can be present. `mass_unit` should not be present.

Two details about the stored matrix: First, the matrix is stored without static condensation. Second, the Dirichlet boundary conditions are pre-eliminated from the stored linear system, although the corresponding rows of the matrix are kept to maintain the same indexing of the unknowns. A row in the matrix corresponding to a Dirichlet point is replaced by an identity row, i.e. 1 on the diagonal and 0 elsewhere, and the corresponding right hand side is replaced with the Dirichlet value. To maintain symmetry, the corresponding column is replaced with an identity column and the right hand side is adjusted

by subtracting the eliminated column entries times the Dirichlet values.

For an elliptic eigenvalue problem, the discrete problem is a generalized eigenvalue problem $Ax = \lambda Mx$. `phaml_store_matrix` can store the stiffness matrix, A , and/or the mass matrix, M . The presence of the optional arguments `stiffness_unit` and `mass_unit` determines whether or not the stiffness matrix and mass matrix will be stored, respectively. Either or both can be present. `rhs_unit` should not be present.

When present, `stiffness_unit`, `rhs_unit` and `mass_unit` should contain the I/O unit of a file. It is the responsibility of the calling program to open the unit for formatted, sequential writing before calling `phaml_store_matrix` and closing it after. This is done with the usual Fortran `open` and `close` statements, not the PHAML `phaml_popen` and `phaml_pclose` statements. If more than one is present, they must not be the same file.

The matrices are written in the Matrix Market exchange format. See <http://math.nist.gov/MatrixMarket/> for a description of the format and routines for reading the format in Fortran, C and Matlab. They are written in the real, general, coordinate format. The right hand side vector is written as an $N \times 1$ matrix in the real, general, coordinate Matrix Market format.

3.6.3 Store Grid

The grid, and optionally solution, can be stored to data files using `phaml_store_grid`. In 2D, the files can be stored in triangle's `.poly` format (see <http://www.cs.cmu.edu/~quake/triangle.poly.html>) by specifying `fmt=GRIDFILE.POLY` to store it without the solution, or `GRIDFILE.POLY.SOLN` to store it with the solution. The solution is stored as attributes of the vertices, so for high order solutions only the linear part is stored. The other triangle files (`.node`, `.neigh`, etc.) can be created from the `.poly` file using `triangle -pneQY filename.poly`.

If the domain has holes, then the `boundary_` routines (Sect. 3.2.3) must be available, i.e., the initial grid cannot be specified by triangle files.

In 2D or 3D, the files can be stored in Gmsh's `.msh` format (see <http://geuz.org/gmsh/doc/texinfo/gmsh.html#MSH-ASCII-file-format>) by specifying `fmt=GRIDFILE.MSH` to store it without the solution, or `GRIDFILE.MSH.SOLN` to store it with the solution. The solution is stored as a `NodeData` section, so for high order solutions only the linear part is stored. If there are any tags attached to the grid entities, they are included in the `.msh` file unless `notags` is present and `.true`. The Elements section contains not only the grid elements (triangles in 2D and tetrahedra in 3D), but also the boundary vertices, boundary edges, and, in 3D, boundary faces unless `noboundary` or `only_gmsh` is present and `.true`. (`only_gmsh` only applies to 3D grids.) If `noboundary` is `.true`, boundary entities are not included. If `noboundary` is `.false` or not present, and `only_gmsh` is `.true`, then vertices, edges and faces that were "elements" in the original geo or msh file, and their descendants, are included, but no other boundary entities. For boundary entities, the first tag is set to the boundary marker (Sects 3.2.2, 3.2.3). This tag is included even if `notags` is `.true`.

`comp` and `eigen` are optional arguments to control which components of a multicomponent solution and which eigenfunctions of an eigenvalue problem with multiple eigenvalues are stored. If `comp` is present, then it indicates which component to store. If it is not present, then all components are stored. If `eigen` is present, then it indicates which eigenfunction to store. If it is not present, then all eigenfunctions are stored. If neither `comp` nor `eigen` is present in a multicomponent, multieigenvalue problem with n components and m eigenvalues, then the order of the solution values in the data file is (comp 1, eigen 1), (comp 2, eigen 1), ..., (comp n , eigen 1), (comp 1, eigen 2), (comp 2, eigen 2), ... (comp n , eigen m).

In this release, storing the grid is restricted to one processor, i.e., PHAML must be built for sequential or OpenMP execution, or, if it is built for hybrid or MPI execution, you must specify `nproc=1`.

3.6.4 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 and true solution. The norms of the error are absolute error; you can get the relative error by dividing by the norm of the solution. 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 7.4.12.

3.6.5 Solution evaluation

PHAML provides for evaluating the computed solution and its derivatives 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 and/or first and second derivatives. You can request any subset of them. You also pass two arrays, `x` and `y`, containing the points at which to evaluate the solution. The solution is returned in the arrays `u`, `ux`, `uy`, `uxx` and `uyy`, which must have the same dimension as `x` and `y`. These are all optional arguments, and which ones are present determines which ones are evaluated. 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.

Another way to obtain the solution for post processing is through the subroutine `phaml_get_grid_soln` (see Section 7.4.7). This routine returns the vertices of the triangles in the grid in two arrays, `x` and `y`, and the solution at the vertices in the array `u`. The vertices are not returned in any particular order relative to

their placement in the domain. If you are solving a system of equations or an eigenvalue problem, there are optional arguments to specify which component or eigenfunction you want to evaluate.

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.

3.6.6 Functionals

In this context, a functional of the solution is some form of integral of the computed solution. Subroutine `phaml_integrate` (see Section 7.4.9) 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.

Chapter 4

Problem Extensions

4.1 Eigenvalue Problems

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

$$-\frac{\partial}{\partial x} \left(c_{xx} \frac{\partial u}{\partial x} \right) - \frac{\partial}{\partial x} \left(c_{xy} \frac{\partial u}{\partial y} \right) - \frac{\partial}{\partial y} \left(c_{yx} \frac{\partial u}{\partial x} \right) + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} + c_u u = \lambda f u \text{ in } \Omega \quad (4.1)$$

$$u = 0 \text{ on } \partial\Omega_D \quad (4.2)$$

$$\left(c_{xx} \frac{\partial u}{\partial x} + c_{xy} \frac{\partial u}{\partial y} \right) \frac{\partial y}{\partial s} - c_{yy} \frac{\partial u}{\partial y} \frac{\partial x}{\partial s} + c_{bc} u = 0 \text{ on } \partial\Omega_N \quad (4.3)$$

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 4.1, 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 λ . 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.

The discrete eigenvalue problem is solved using SLEPc or one of the external packages supported by SLEPc (ARPACK and BLOPEX). You should read the SLEPc User's Guide [10] to understand the arguments that effect the

SLEPc solvers. Not all methods are valid for all problems, and not all combinations of arguments are valid. The method to use for solving the eigenproblem is specified by `eigensolver`. Valid values are `SLEPC_POWER`, `SLEPC_SUBSPACE`, `SLEPC_ARNOLDI`, `SLEPC_LANCZOS`, `SLEPC_KRYLOV_SCHUR`, `SLEPC_GEN_DAVIDSON`, `SLEPC_JACOBI_DAVIDSON`, `SLEPC_LAPACK`, `SLEPC_ARKPACK`, and `SLEPC_BLOPEX`.

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`. Convergence to an interior eigenvalue can be accelerated by using harmonic extraction, specified by `harmonic_extraction=.true.`, or by using a spectral transformation, specified by `transformation`. Valid values are `ST_NONE`, `ST_SHIFT_ORIGIN`, `ST_FOLD`, `ST_SHIFT_INVERT` and `ST_CAYLEY`. The transformations use a shift which, by default, is `lambda0`, but can be specified to be a different number using `st_shift`. The Cayley spectral transform also has an antishift which, by default, is the same as the shift, but can be specified with `st_antishift`.

Some solvers (e.g., Krylov solvers) can avoid computing the residual by using a cheap estimate of the residual norm, but this may sometimes give inaccurate results, especially if a spectral transform is being used. If this is a problem, you can force computation of the true residual with `slepc_true_residual=.true.`

PHAML's default size of the subspace used by the solvers, a.k.a. number of column vectors, is the same as SLEPc's default. A different size can be specified with `eigen_ncv`. Termination of the eigensolver is controlled by two parameters, `eigen_tol` and `eigen_maxit`. `eigen_tol` is a tolerance for the error estimate of the eigenvalues and has a default value of 10^{-10} . `eigen_maxit` gives the maximum number of iterations allowed and has the same default value as SLEPc.

If u is a solution of Equation 4.1, then so is αu for any scalar number α . 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_evec` to subroutine `phaml_solve_pde`. `SCALE_LINF` requests that the ℓ^∞ norm of the eigenvector, x , of the discrete problem be 1. With linear elements, this is the same as the L^∞ norm of the eigenfunction u . `SCALE_L2` scales such that the ℓ^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.

The eigensolver requires the use of a linear system solver and sometimes a preconditioner. These are specified through `solver` and `preconditioner`. They must be solvers and preconditioners that are available through PETSc, including the `PETSC_MUMPS` and `PETSC_SUPERLU` solvers and the `NO_PRECONDITION` preconditioner, but not the `MG`, `FUDOP_DD` and `COARSE_GRID` preconditioners. The default is the same as SLEPc's default.

```

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 4.1: `pdecoefs` for a coupled system.

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 1.1 - 1.3 are the same except the functions c_{xx} , c_{xy} , c_{yy} , c_x , c_y , c_u and c_{bc} 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 4.1 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.

If any of the matrices c_{xx} , c_{yy} , c_u or c_{bc} is nonsymmetric, or c_{xy} , c_x or c_y is nonzero, then you must use a nonsymmetric solver like LAPACK_INDEFINITE_SOLVER (single processor only) or a nonsymmetric solver from PETSC, including PETSC_MUMPS_GEN_SOLVER and PETSC_SUPERLU_SOLVER, if so configured. Do not use the built-in hierarchical basis multigrid method (the default) as a solver or as a preconditioner.

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/nonlinear` for solving a nonlinear PDE using a simple Picard iteration.

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 7.4.8). You can evaluate the solution and/or the first derivatives and/or the second derivatives. This would normally be called from subroutine `pdecoefs`.

PHAML only provides for saving one “old” solution. You would think this would prohibit the use of methods that require more than one old solution. However, in many cases such problems can be reformulated as a coupled system of equations. See `examples/waveeq` for an example of solving a problem with a second order time derivative, which requires two old solutions, as a coupled system of two first order equations. An example of a second order backward difference scheme, which requires two old solutions, implemented as a coupled system of two equations is given in `examples/bdf2`. And `examples/irk4` contains a fourth order 6-stage implicit Runge Kutta method implemented as a coupled system of six equations.

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 `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.

If the solution does not change very much from one time step (or iteration) to the next, it may be possible to take several steps without changing the grid. In this case, the matrix does not change from one step to the next (provided the differential operator is linear). By default, PHAML destroys the linear system before returning from `phaml_solve_pde`. Setting the parame-

ter `keep_linear_system=.true.` prevents this destruction so the matrix can be reused in the next call to `phaml_solve_pde` without recomputing it. Since the right hand side is likely to change, it is always recomputed. If the grid changes through refinement or load balancing, the kept linear system is automatically destroyed and recomputed. Also, the kept linear system is destroyed in `phaml_destroy`. See `examples/keeplinsys` for an example of how to use this feature.

4.4 3D Problems

Support for three dimensional problems in PHAML is currently under development. This release supports the following features:

- h -uniform refinement of tetrahedra
- h -adaptive refinement
- linear elements
- high order elements
- general PDE operator
- Dirichlet, natural, mixed and periodic boundary conditions
- domains with boundary edges that are straight line segments or circle arcs
- hierarchical coefficient error indicator
- initial condition error indicator
- optional third party solvers PETSc, hypre, ML, LAPACK, SuperLU, MUMPS
- all printed output
- all termination arguments
- coupled systems of equations
- eigenvalue problems
- `task=SET_INITIAL`
- PHAML procedures: `phaml_compress`, `phaml_connect`, `phaml_copy_soln_to_old`, `phaml_evaluate`, `phaml_evaluate_old`, `phaml_get_grid_soln`, `phaml_integrate`, `phaml_query`, `phaml_restore`, `phaml_scale`, `phaml_store`, `phaml_store_grid`, `phaml_store_matrix`

The 3D version of PHAML does not yet support these features:

- p -uniform refinement

- p -adaptive refinement
- hp -adaptive refinement
- derefinement
- high order elements
- explicit, local problem h , local problem p , equilibrated residual, and true diff and initial condition error indicators
- multigrid, CG and GMRES native solvers
- OpenMP prallelism
- MPI parallelism
- Zoltan partitioning
- RTK partitioning
- task=SET_INITIAL
- PHAML procedures: `phaml_compress`, `phaml_connect`, `phaml_copy_soln_to_old`, `phaml_evaluate`, `phaml_evaluate_old`, `phaml_get_grid_soln`, `phaml_integrate`, `phaml_query`, `phaml_restore`, `phaml_scale`, `phaml_store`, `phaml_store_matrix`

The 3D version of PHAML works exactly like the 2D version, with a few minor differences to handle the third dimension. The 2D and 3D versions of PHAML use common source code for most of the source files, with runtime selection of code sections that differ depending on the dimensionality. But there are a few files that have 2D and 3D versions, like the grid data structure module, and are selected at compile time. Thus PHAML is compiled for either 2D or 3D problems, but not both. To compile for 3D, use `tetrahedron` or `tetrahedron_fixdeg` for PHAML_ELEMENT when running `mkmkfile.sh`. `tetrahedron_fixdeg` will use a data structure with a fixed polynomial degree for the solution, and will save a considerable amount of memory when using h -refinement or hp -adaptive refinement with a small limit on the degree, as given by the argument `max_deg` of subroutine `phaml_solve_pde`. For p -refinement and hp -adaptive refinement with a large limit on the degree, use `tetrahedron`, in which the size of the solution component varies with the degree of the element.

The differences in the user interface for 3D are:

- Subroutine `pdcoefs` has additional arguments for the third dimension. The type specification for these arguments, and how they fit into the definition of the 3D PDE should be obvious from the 2D routine in Section 7.3.6. The argument list is `x,y,z,cxx,cyy,czz,cxy,cxz,cyz,cx,cy,cz,c,rs`.
- Subroutines `bconds`, `truecs`, `truexs`, `trueys`, `phaml_integral_kernel` and `regularity` have an argument `z` that follows `x,y`.
- An additional subroutine `truezs` must be supplied, in the same manner as `truexs`.

- Subroutines `boundary_point`, `boundary_npiece` and `boundary_param` are not needed.
- Default values of input parameters are different when the 2D default is not yet supported in 3D.

There are also many differences in the graphics menu. You are encouraged to experiment with it.

`triangle_files` in `phaml.create` is still used to define the initial grid, but the input formats are different than the one used in 2D. For 3D, PHAML uses files in the formats of the 3D mesh generator Gmsh [18, 19]. There are two types of files used by PHAML: geometry files with a `geo` suffix, and mesh files with a `msh` suffix. One normally defines the domain and boundary markers (to be passed to `bcond`, see Section 7.3.1) in a geometry file. A mesh file is usually created by applying Gmsh to a geometry file. There are examples of both types of files in `examples/domains3D`. For a full description of these file types, see the Gmsh reference manual at <http://geuz.org/gmsh>. The character string in `triangle_files` can take one of three forms. If it is `filename.msh` then the given `filename` with `msh` suffix is a mesh file. Mesh files can only be used for domains with planar faces and straight line edges, because geometry information is required to properly refine curved boundary edges. If it is `filename.geo` then the file is a geometry file. PHAML will run Gmsh, putting the output in `phaml_filename.msh`. If it is `filename.geomsh` then both the geometry file `filename.geo` and mesh file `filename.msh` exist and the mesh file was created from the geometry file. (Note that `.geomsh` is not the suffix of a file name.) PHAML will use the mesh file to generate the initial grid and the geometry file when refining curved boundary edges.

PHAML supports a limited form of Gmsh's geometry files. It currently allows domains that have edges that are straight line segments or circle arcs, and faces that are planar with an arbitrary number of edges, or a ruled surface with 3 or 4 edges. Planar faces can have holes. In general, a ruled surface is defined by transfinite interpolation of the edges. The exception is that if all the edges of a ruled surface are circle arcs with the same center, Gmsh has a hack to make that surface a segment of a sphere. The following features of a geometry file are currently supported:

- C and C++ style comments
- numeric expressions involving numbers (integer, fixed point, and exponential with e, E, d or D), variables, parentheses, and the operators `+`, `-`, `*`, `/`, `^` and `%`.
- `variable = expression`
- Include
- geometry commands: Point, Physical Point, Circle, Line, Physical Line, Line Loop, Plane Surface, Ruled Surface, Physical Surface, Volume, and Physical Volume

It does not support character expressions, operators not listed above, built-in functions (Log, Sin, etc.), user-defined functions, loops and conditionals, predefined expression identifiers (Pi, newp, etc.), general commands not listed above, Ellipse, Spline, BSpline, Compound Line, Compound Surface, the “In Sphere” clause of a Ruled Surface statement, extrusions, transformations and mesh commands. Also, an expression cannot be split over more than one line and cannot contain an embedded comment. For an Include statement, the *character-expression* must be a string enclosed in double quotes which gives a file name that can be passed to a Fortran OPEN statement. A ruled surface can only have one line loop, i.e. no holes. Note that most unsupported features are simply ignored, which may or may not produce the desired effect, but some might cause the parser to fail. If you need an unsupported feature, contact phaml@nist.gov.

Objects (vertices, edges, faces) on the boundary have an associated boundary marker, which can be used in subroutine `bconds` to determine which boundary condition to return. The boundary markers are defined by “Physical” objects in geometry files, and tags in mesh files. For example, a geometry file could define vertex 2 as a point at the origin by

```
Point(2) = {0,0,0};
```

and then define a physical point by

```
Physical Point(5) = {2};
```

Here 2 is the point index; 5 is the index of the physical point and will be the boundary marker for that point. When Gmsh processes the geometry file to create a mesh file, it will create an “element” for that physical point with 5 as the first tag and an indication it is vertex 2.

It is recommended that you assign a nonzero boundary marker (zero is used to indicate interior objects) to every boundary object, unless your boundary conditions are defined by a single equation so that you don’t need boundary markers. However, if there are boundary objects that do not appear in the mesh file as a Physical object or do not have tags, PHAML will assign a default boundary marker to it. The default boundary marker is the smallest positive integer that does not appear in the mesh file as a first tag. Boundary markers are assigned as follows:

- The boundary marker is set to 0 for interior vertices, edges and faces.
- If a boundary vertex, edge or face is in the mesh file with tags, the first tag is used as the boundary marker. By Gmsh’s default, this is the index of the physical region containing the entity.
- If a boundary face is not in the mesh file or has no tags, it is assigned the default boundary marker.
- If a boundary edge is not in the mesh file or has no tags, it is assigned the boundary marker of one of the two faces that share the edge. If one of the faces has the default boundary marker, the edge gets the boundary marker of the other face (which could be the default boundary marker). Otherwise, if one face has Dirichlet boundary conditions and the other does

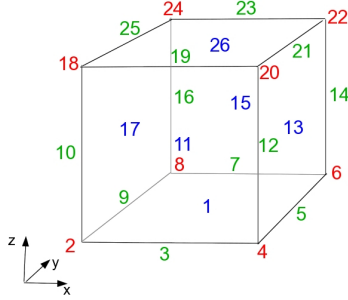


Figure 4.2: Assignment of boundary markers in `unitcube.msh`

not, the edge gets the boundary marker of the Dirichlet face. Otherwise, it gets the smaller of the two boundary markers.

- If a boundary vertex is not in the mesh file or has no tags, it is assigned the boundary marker of one of the edges that contains it. If any edges have Dirichlet boundary conditions, it is assigned the smallest boundary marker of those edges. Otherwise, it is assigned the smallest boundary marker of all the edges.

If you are solving a problem on the unit cube, you can use `examples3D/domains/unitcube.msh` as your mesh file. You might want to rename it `domain.msh`, which is the default value for `triangle_files`. Figure 4.2 shows the boundary markers for this mesh. Vertex boundary markers are shown in red, edge boundary markers in green, and face boundary markers in blue. For a more general parallelepiped, use `examples3D/domains/parpipe.geo`.

Periodic boundary conditions are supported only for parallelepipeds with faces parallel to the axes. They can be periodic in x, periodic in y and/or periodic in z. Since it is known what the matching periodic side is, you do not need to use the matching positive/negative boundary markers as in 2D, but you must indicate both sides are `PERIODIC` in subroutine `bconds`. Boundary edges and vertices that are at the intersection of a periodic side and a nonperiodic side can set the boundary condition type to either `PERIODIC` or whatever the type is on the nonperiodic side.

Chapter 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.

`holes` – defines a domain with holes.

`periodic` – periodic boundary conditions.

`eigenvalue` – an elliptic eigenvalue problem.

`parabolic` – solves a time-dependent parabolic PDE by an implicit finite difference scheme in t .

`keepinsys` – solves a parabolic PDE without recomputing the matrix at every time step.

`nonlinear` – 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.

runtimepde – determine user provided routines at run time

waveeq – solves the second order wave equations as a coupled system of two first order equations.

bdf2 – solves a parabolic PDE by second order backward differences implemented as a coupled system of two equations.

irk4 – solves a parabolic PDE by a 4th order, 6-stage implicit Runge Kutta method implemented as a coupled system of six equations.

3D examples are in **phaml-x.x.x/examples3D**:

simple – a trivial program for a 3D problem.

elliptic – several 3D elliptic boundary value problems.

eigenvalue – a 3D eigenvalue problem.

domains – domains defined by Gmsh **.geo** and **.msh** files.

periodic – periodic boundary conditions.

There are also several programs under the **testdir** and **testdir3D** directories, but they do not contain comments to help you understand them.

Chapter 6

Release notes

This section is reserved for listing the changes with each new release of PHAML. Changes in other parts of this document are indicated by font color and size. Changes made in the current release are printed in **red**. Changes made in a recent release are printed in **dark red**. Text that has been deleted is maintained for a few releases, but is printed in a tiny size.

6.1 Version 1.14.0

Released September 4, 2014

6.1.1 Summary of changes

- Add support for GAMG algebraic multigrid preconditioner in PETSc.
- Add support for ML algebraic multigrid preconditioner from Trilinos via PETSc.
- Changed `refsoln_pbias` to `pbias`.
- Allow use of `pbias` with `HP_BIGGER_ERRIND`.
- Change the default *hp*-adaptive strategy to `HP_COEF_DECAY`.
- Completed patch problem module.
- Completed new implementation of local *h* error indicator using the patch problem module.
- Check for exceeding maximum *h*-levels before refining a tetrahedron.
- If keeping the linear system between calls to `phaml_solve_pde` and using a PETSc solver, keep the PETSc matrix.
- Include the time to create the PETSc matrix in assembly instead of solve.

- Change `store_grid_msh` to 1) not use internal vertex lids 2) don't do vertex elements first.
- Change `store_grid_msh` again to order nodes and tetrahedra elements in the same order as the msh file.
- Have master return immediately from subroutine `refine`.
- Consolidate reductions and messages in `get_grid_info`.
- Added option to include `gmsh_element` vertices, edges and faces as elements (instead of all boundary entities) when using `phaml_store_grid` to save a 3D grid in msh format.
- Add check for no common surface between vertices in `define_boundary_vertex`.
- Use quad precision in `circle_crossing`.
- Use small logical kind for large logical arrays.
- Add interior Dirichlet conditions.
- Removed a few unnecessary MPI messages.
- Added additional output to the tests.
- Put time for `enforce_overlap` under reconciliation instead of `refine`.
- Add fatal error message to dummy SLEPc interface.
- Call `EPSDestroy` in SLEPc interface.
- Finalize SLEPc and PETSc in dummy `messpass`.
- Option to use 8-byte integers in `hash2`.
- Diagonal scaling of the linear system in certain situations.
- Include element tags in values prohibited for `default_bmark`.
- Skip copying periodic solutions in `solve` if no periodic boundary conditions.
- Skip looking for periodic master in `grid_to_eq` if no periodic boundary conditions.
- Don't reallocate solutions if they are the right size.
- New version of `create_column_index` that is simpler, faster and less memory.
- Added `bc_changed` to indicate boundary conditions do not need to be reset.
- Added a system of 3 PDEs to the tests.

- Bug: allocated bcrs in `elemental_matrix` at `nbasis` instead of `ss`.
- Bug: set `grid_modified` for slaves.
- Bug: increase the length of character string for `inline` in `read_msh`.
- Bug: missing `dim=2` when copying `desired_degree` in `more_elements`.
- Bug: needed OpenMP critical around initialization of Laplace elemental matrix.
- Bug: missing `cz` term in `norm_error`, `norm_solution` and `norm_true`.
- Bug: wasn't subtracting Dirichlet b.c. from PETSc rhs while updating rhs when keeping the linear system between calls.
- Bug: clear `relresid` when the linear system is kept.
- Bug: deallocated `lchild` instead of `rchild` in `destroy_search_tree`.
- Bug: set `finalize_mpi` from argument in `phaml_destroy` when sequential.
- Bug: test `finalize_mpi` before calling `slepc_finalize` and `petsc_finalize` in `sequential_terminate_comm`. `endif`

6.1.2 Major changes

Added support for GAMG algebraic multigrid preconditioner in PETSc (`preconditioner = PETSC_GAMG_PRECONDITION`) and ML algebraic multigrid preconditioner from Trilinos (`preconditioner = PETSC_TRILINOS_ML_PRECONDITION`).

The argument `refsoln_pbias` that biases *hp*-adaptive refinement toward using *p*-refinement in `HP_REFSOLN_ELEM` is changed to `pbias`, and its use extended to include `HP_BIGGER_ERRIND`.

Added an option for 8-byte integers in the hash keys, if the compiler supports 8-byte integers. This essentially doubles the number of levels of *h*-refinement that can be performed. To use this, copy `hash2_i8.f90` to `hash2.f90` and `hashext2_i8.f90` to `hashext2.f90`, and in `global.f90` change `int8` to the kind type parameter for 8-byte integers.

6.1.3 Upgrading

There are major changes in `mkmkfile.sh` so you will need to modify and use the new version of it.

The default *hp*-adaptive strategy was changed to `HP_COEF_DECAY`. To get the old behavior, use `hp_strategy=HP_SMOOTH_PRED`.

If you use `refsoln_pbias`, change it to `pbias`.

6.2 Version 1.13.0

Released September 23, 2013.

6.2.1 Summary of changes

- Extend `phaml_compress`, `phaml_evaluate_old`, `phaml_copy_soln_to_old`, `phaml_integrate`, `phaml_scale`, `phaml_get_grid_soln`, `phaml_store_matrix`, `phaml_query` and `phaml_connect` to 3D.
- 3D `task=SET_INITIAL` and `INITIAL_CONDITION` error indicator.
- Remove Dirichlet boundary conditions in `phaml_store_matrix`.
- Set PETSc initial guess from current solution.
- Options to not include boundary and tags in `phaml_store_grid .msh` file.
- Workaround for `dim=dim` compiler bug.
- Put quotes around filenames in `gmsh` command in `grid_init3D`.
- Replace recursive `make_leaf_list` with an iterative one.
- Change default number of SOR iterations from `PETSC_DEFAULT_INTEGER` to 1.
- Make default preconditioner for BoomerAMG solver with tetrahedra be `NO_PRECONDITION`.
- Preserve allocation in `evaluate_soln_local`, `p_hier_basis_func_a`, and `elemental_matrix`, and reallocate when the size is different.
- Removed unnecessary products in `fast_barycentric`.
- For hypre options, add a second call to `PetscOptionsSetValue` with a `st_` prefix if solving an eigenvalue problem.
- Send `solver_options` to `elemental_matrix` from local problem `h` error indicator.
- Don't evaluate derivatives of basis functions for elemental matrix when using the Laplacian table.
- Require `which_set == "a"` and `which_basis == "p"` to use Laplacian table.
- Move setting next for periodic edges and vertices from `before_h_refine` to `h_refine_element`.
- New version of 2D `get_neighbors` that uses neighbor hints like the 3D version.
- Create a module to create and solve problems on small patches of elements. Still under development.
- New version of `LOCAL_PROBLEM_H` error indicator based on patch problems. Still under development; old version still in use.

- Divide maximum error indicator by four instead of fourth root of two to determine which elements to refine.
- Remove JOSTLE, PaToH, ParKway and Nemesis (Zoltan add-ons that have never been supported in PHAML) from mkmkfile.sh.
- Remove release notes for versions 1.0.0 through 1.9.1
- Bug: not setting `pde_has_first_order_terms` etc in `store_matrix`.
- Bug: nullify solution components before first call to allocate them.
- Bug: always call `all_error_indicators` before `compute_global_max_errind` in `adapt`.
- Bug: avoid using 0 lcoef in `coef_decay`.
- Bug in eigenvalue error estimate in 3D.
- Bug: test for less than `-epsilon` instead of 0 in `find_containing_element`.
- Bug: using `size_edge_soln` for allocating vertex solution in 3D `copy_grid`.
- Bug: elemental mass matrix not set if `laplacian_operator` is true.
- Bug: duplicate element numbers in `store_grid_msh`.
- Bug: for `ELEMENT=tetrahedron`, bad use of `GRID_DIM` in `size_vert_soln`.
- Bug: error in choosing marked edge for face 5 in refinement of tetrahedron.
- Bug: real result instead of `real(my_real)` result in dummy version of
- Bug: nullify `jmatrix format_i_exists` for master in `create_linear_system`.
`phaml_global_sum_realv`.
- bug in `LOCAL_PROBLEM_H` error indicator with periodic boundary conditions.

6.2.2 Major changes

The major change is additional support for 3D problems. The post-solution utility routines were extended to support 3D problems. Support for initial conditions for 3D problems was added, allowing time-dependent and nonlinear 3D problems.

The criterion for refining an element with adaptive refinement was changed from the maximum error indicator divided by the fourth root of two to the maximum error indicator divided by four.

6.2.3 Upgrading

Application source code should not require any changes. But note that the grid and solution will be slightly different because of the change in how elements are selected for refinement. Also, there are major changes in `mkmkfile.sh` so you will need to modify and use the new version of it.

6.3 Version 1.12.1

Released February 8, 2013

6.3.1 Summary of changes

- subelement resolution for 3D graphics.
- `mkmkfile.sh` additions for Cygwin with gfortran or MinGW.
- overhall solution component function for better consistency and bug fixes.
- Bug: wrong size for fourth dimension in `reallocate_face_exact`.
- Bug: needed to transpose array in `get/set` solution routines for `tetrahedron_fixdeg`.
- Bug: bug in `face_exact` for `tetrahedron_fixdeg`.
- Bug: return a value from `size_*` and `allocated_*` for `tetrahedron_fixdeg` when the entity is present.
- Bug: use size of input for do loop limit in `set_*` for `tetrahedron_fixdeg`.
- Bug: wrong basis rank for face equations when building linear system for coupled systems.
- Bug: used edge degree instead of face degree in `face_exact`.

6.3.2 Major changes

This is primarily a bug fix release, but it also added subelement resolution in 3D graphics, and added `mkmkfile.sh` entries for Cygwin with gfortran or MinGW compilers.

6.3.3 Upgrading

No changes to user code are needed. The new `mkmkfile.sh` is needed only if you plan on running on Cygwin.

6.4 Version 1.12.0

Released January 7, 2013

6.4.1 Summary of changes

- Remove direct support for hypre.
- Add support for hypre via PETSc.
- Added `grid_modified`.
- Added `init_alloc` to `phaml_create`.
- Use functions to access solution, exact and `oldsoln` components instead of directly using them.
- Added element type `tetrahedron_fixdeg` with fixed degree solution component.
- `phaml_evaluate` for 3D.
- High order tetrahedral elements.
- Quadrature rules for high order tetrahedron.
- Quadrature rules for lines in 3-space.
- High order bases for tetrahedron.
- High order hierarchical coefficient error indicator for 3D.
- `edge_exact`, `face_exact` and `elem_exact` for 3D.
- Moved `linear_system` variable from subroutine `solve` to `phaml_solution_type` data structure.
- Added ability to keep the linear system between calls to `phaml_solve_pde`.
- Added an example to show how to use `keep_linear_system`.
- Added an example with holes in the domain.
- Keep the same order of vertices as the parent in faces.
- Order vertices in tetrahedra to be clockwise with newest vertex last.
- Order vertices in initial edges and faces the same as in `.msh` file.
- Order faces and `neighbor_hint` to be opposite the corresponding vertex in 3D.
- Order edges in tetrahedra in a canonical order.

- Optional which in `get_neighbors` in 3D.
- Extended `phaml_store` and `phaml_restore` to 3D.
- Don't set lids to 0 in 3D before `h_refine`.
- In `h_refine_grid` in `refine_adapt`, use constant-level sections of `elem_list`.
- Allow filenames to be up to 1024 characters in `phaml_popen`.
- Added `iostat` to `phaml_popen`.
- Changes in `petsc_interf.F90` due to changes in PETSc 3.3.
- In `more_elements` and `friends`, use actual size of each array for copy upper bound.
- Calling dummy PETSc routine is a fatal error, instead of warning, and stops.
- When requesting graphics with a sequential compilation, give warnings, and eventually quit, if it appears `phaml_graphics` is not running.
- Bug: divide tetrahedron volume by 6.
- Bug: incorrect right side in `examples3D/domains`.
- Bug: had row and col reversed for `row_block_t` in static condensation.
- Bug: could exceed allocation for edges or faces in `init_grid_msh`.

6.4.2 Major changes

High order 3D elements have been added. These can be used for high order h -refinement with uniform p , but p - and hp -refinement is not yet supported.

A new version of the 3D grid data structure was added. This version uses a fixed-sized array in the grid data structure for the solution instead of an allocatable array in each element, face and edge. This provides a large memory savings because the allocatable components have a large memory overhead with most Fortran compilers. However, this is not appropriate for refinement methods that result in a large range of polynomial degree, because the arrays must be dimensioned for the largest degree in the grid. In general, when running `mkmkfiles.sh` you should use "ELEMENT tetrahedron" when using p -adaptive refinement or most cases of hp -adaptive refinement (although neither of those are supported yet; keep this in mind for the future), and "ELEMENT tetrahedron_fixdeg" when using h -uniform, h -adaptive, and p -uniform refinements, and hp -adaptive refinement if the maximum degree is limited to a small value like 3 or 4.

It is now possible to save the linear system between calls to `phaml_solve_pde` by using `keep_linear_system=.true`. This can be useful, for example, when solving a parabolic PDE where the differential operator does not change from one

time step to the next, only the right side changes. The right side of the linear system is always recomputed. The matrix will be recomputed only if the grid changes, so this should be used in conjunction with `task=SOLVE_ONLY`. See the new example that illustrates this.

hypre is now accessed through PETSc instead of directly.

Added 3D support to `phaml_evaluate`, `phaml_store` and `phaml_restore`.

An initial grid allocation size can now be specified in `phaml_create`.

6.4.3 Upgrading

You will need to use the new version of `mkmkfiles.sh`.

If using the hypre solvers or preconditioners, you will need to change the values for solver and preconditioner in the call to `phaml_solve_pde`. See Section 7.4.15.7 for the available values, but in general you use the PETSc solver (for example, conjugate gradients) instead of the hypre solver (except BoomerAMG), and add `PETSC_` in front of the hypre preconditioner name.

6.5 Version 1.11.0

Released August 6, 2012

6.5.1 Summary of changes

- For 3D:
 - Subroutine `remove_hanging_nodes` for adaptive refinement.
 - Always set `mate=BOUNDARY` for tetrahedra to avoid compatibility chain.
 - Initial grid from Gmsh `.msh` file.
 - Output grid to Gmsh `.msh` file.
 - Domain defined by Gmsh `.geo` file.
 - Refinement of curved boundaries (with circular edges) defined by Gmsh `.geo` file.
 - Quadrature rules for triangles in 3-space.
 - Eigenvalue problems.
 - Periodic boundary conditions.
 - Face and edge transparency in graphics.
 - Isosurfaces in graphics.
 - View toward `xz` or `yz` plane.
 - Grid lines and contour plots on cutting planes.
 - Changed default solver to `PETSC_CG_SOLVER` and default preconditioner to `PETSC_SOR_PRECONDITION`.

- Removed `code:tetrahedron` and `code:cube` initial grids.
- New directory for 3D examples.
- Output grid to triangle `.poly` or Gmsh `.msh` file.
- Added `laplacian_operator` and `isosceles_right_triangles`.
- Added stack module.
- Added tags to grid data structures.
- With PETSc, if the linear system only has one equation, just set the solution.
- Added barrier subroutine that includes MASTER.
- Changed formula for area of a triangle in quadrules to stable Heron's formula.
- Reordered equations so edge or face rows with the same degree are grouped together.
- Initialize quadrature rule tables from `phaml_create`.
- Added option of specifying new size in grid reallocation routines.
- `graphics unpack_grid`: check for duplicates in hash table instead of searching list.
- Reset neighbor hint after finding actual neighbor.
- Change initial allocation of vertices, etc., in 3D to better reflect the relative number of objects.
- Many changes to improve performance; list available on request.
- bug: check for reallocation after `create_element` in `reconcile`
- bug: in Zoltan interface, array sections in `hash_[un]pack_key` too short.
- bug: forgot c_z term in `elemental_matrix`.
- bug: wasn't setting `petsc_matrix%global_eq` when `matrix-free`.

6.5.2 Major changes

Development of the 3D code continues. Major additions in this release are h-adaptive refinement, nonrectangular domains, eigenvalue problems and periodic boundary conditions. See Section 4.4. Major additions to 3D graphics are transparency, isosurfaces, grid lines and contour plots on the cutting planes, and setting the view toward the xz or yz plane.

Major changes that are not specific to 3D are the ability to save the grid to a `.poly` or `.msh` file, the addition of tags to the grid data structures, and the ability to indicate the operator is the Laplacian and the triangles are all isosceles right triangles for faster computation of elemental matrices.

6.5.3 Upgrading

mkmkfiles.sh contains several important changes. You will need to use the new version of it, after modifying it for your system.

triangle_files=code:tetrahedron and code:cube are no longer supported. Use tet.geo and unitcube.msh or unitcube.geo instead. These files are in examples3D/domains.

The default solver for 3D problems was changed to PETSc CG with SOR preconditioning. To get the old behavior, use solver=LAPACK_SPD_SOLVER.

6.6 Version 1.10.0

Released December 13, 2011

6.6.1 Summary of changes

- Additional OpenMP sections:
 - create_column_index
 - computation of norms
- Begin extension to 3D. This involved many changes. A list is available upon request.
- Add petsc_init_called to petsc_init_mod.
- Add degree-based weights to the explicit error indicator.
- Add pde.has.first_order_terms and pde.has.cross_derivative.
- Remove SMOOTH forms of refterm.
- Remove direct support (via PETSc) for BLOPEX.
- Remove direct support for ARPACK.
- Add support for SLEPc.
- Remove direct support for MUMPS and SuperLU.
- Add support for MUMPS and SuperLU via PETSc.
- Remove lambda0_side.
- Remove SHIFT_INVERT and SHIFT_SQUARE.
- Added support for sequential PETSc.
- Get PETSc number of iterations.
- Remove shifted matrix and rmin.

- Change default `hp_strategy` to `SMOOTH_PRED`.
- Let MASTER have the initial grid.
- Use boundary marker as test for whether a vertex, edge or face is on the boundary or interior.
- Add test for refinement stalling.
- Changed scaling of relative residual in CG.
- Start loop for static condensation at beginning of elements instead of edges.
- Consolidate multiple barycentric routines to one.
- Move `find_containing_element` to `grid_util.f90`.
- Don't add owner to graphics label for vertices, etc., when there is only one processor.
- Routine to compute the area or volume of an element.
- Print number of OpenMP threads in header.
- Option to set number of OpenMP threads in `phaml_create`.
- Sync'd makefile dependencies with USE statements.
- Use `phaml_alltoall` instead of `phaml_send` in `exchange_desired`.
- Master calls `copy_old`.
- Master has `initial_neighbor` in `phaml_store` and `phaml_restore`.
- Master calls `set_edge_mass`.
- Increased MUMPS fill-in size.
- Remove Dirichlet rows/columns from PETSc form of matrix.
- Change tag in call to `print_time_info` from 122 to 125 to avoid conflict with tag 121+1 in `print_error_info`
- Don't change rhs of Dirichlet rows in static condensation.
- Bug: Fixed f90 command for `showme` in `mkmkfile.sh`.
- Bug: In `hcoarsen_init_guess`, degree did not consider mate's degree.
- Bug: Passing undefined `child3` to `init_guess_ic`.
- Bug: work in call to `gmres` needs leading dimension at least 3.
- Bug: scaling error when printing relative error with `nproc > 1`.
- Bug: `reallocate_desired_level` etc. in `refine_adaptive` if `pred_load_balance` caused `grid%element` to be reallocated

6.6.2 Major changes

We have begun extending PHAML to solve 3D problems using bisected tetrahedra. This is very early and there is much work still to be done. At this point PHAML can solve an elliptic boundary value problem (including coupled systems) on the unit cube using uniform refinement of linear elements on a sequential computer and the third party linear system solvers. See Section 4.4 for details.

The optional third party linear system solvers MUMPS and SuperLU are no longer supported as stand-alone packages. Instead they can be accessed through PETSc. In addition, support was added for using PETSc in a sequential build.

Support was added for the eigensolver package SLEPc. Direct support for ARPACK and BLOPEX was removed. Instead they can be accessed through SLEPc. This also resulted in changes to several other eigenvalue related arguments to `phaml_solve_pde` and `phaml_query`. See Sections 4.1, 7.4.15 and 7.4.12.

6.6.3 Upgrading

Support for MUMPS and SuperLU is now done through PETSc. If you use these solvers you now access them with `solver = PETSC_MUMPS_GEN_SOLVER`, `PETSC_MUMPS_SPD_SOLVER`, or `PETSC_SUPERLU_SOLVER`. You must have PETSc installed and configured with support for MUMPS and/or SuperLU to use them.

Support for ARPACK and BLOPEX is now done through SLEPc. If you use these eigensolvers you now access them with `eigensolver = SLEPC_ARPACK` and `eigensolver = SLEPC_BLOPEX`. You must have SLEPc installed (which implies also having PETSc installed) and configured with support for ARPACK and/or BLOPEX to use them. In this release BLOPEX can only be used for computing the smallest eigenvalues. We anticipate adding BLOPEX computation of interior eigenvalues in a future release.

The default eigensolver has been changed to `SLEPC_KRYLOV_SCHUR`. To use the old default, use `eigensolver = SLEPC_ARPACK`.

The arguments to `phaml_solve_pde` that control ARPACK (`arpack_ncv`, `arpack_maxit` and `arpack_tol`) and BLOPEX (`blopex_maxit`, `blopex_atol` and `blopex_rtol`) have been replaced by the more generic names `eigen_ncv`, `eigen_maxit` and `eigen_tol`.

The arguments to `phaml_query` concerning ARPACK have been removed. Use these replacements: `eigen_l2_resid` \rightarrow `eigensolver_l2_residual`, `arpack_iter` \rightarrow `eigensolver_niter`, and `arpack_nconv` \rightarrow `eigensolver_nconv`. The following are no longer available: `max_linsys_resid`, `ave_linsys_resid`, `arpack_numop`, `arpack_numopb`, `arpack_numreo` and `arpack_info`. A new argument, `eigensolver_eval_errest`, gives an estimate of the error in each eigenvalue in the discrete eigenvalue problem.

The spectral transforms for computing interior eigenvalues, `SHIFT_INVERT` and `SHIFT_SQUARE`, have been replaced by `ST_SHIFT_INVERT` and `ST_FOLD`, respectively. There are also additional spectral transforms available. The default

spectral transform has been changed to `ST_NONE`. To get the old default, use `transformation = ST_SHIFT_INVERT`.

The specification of which side of the target to use for interior eigenvalues, `lambda0_side`, is no longer available. It appears that SLEPc always computes them on both sides of the target, except for `SLEPC_ARKPACK` which computes them on the right side of the target.

The SMOOTH form of the values for `refterm` have been removed. The adaptive refinement algorithm was changed slightly such that there is no longer any difference between the SMOOTH form and the regular form, with the result being closer to the SMOOTH form. If you used a SMOOTH form then use the same name without the SMOOTH suffix.

The default *hp*-adaptive strategy has been changed to `HP_SMOOTH_PRED`. To get the old default, use `hp_strategy = HP_PRIOR2P_H1`.

PHAML now checks whether the PDE has first order terms or a cross derivative term by sampling at the midpoint of each element in the initial grid. If you have a program where you use `phaml_evaluate_old` in subroutine `pdecoefs` and you call `phaml_solve_pde` before calling `phaml_copy_soln_to_old` (for example, to set the initial condition in a time dependent problem), then it will generate many warning messages about the old solution not being saved. To avoid this, add `pde_has_first_order_terms=.false.` and `pde_has_cross_derivative=.false.` (or `.true.` if it does have them) to the call to `phaml_solve_pde` that occurs before `phaml_copy_soln_to_old`.

The boundary marker from the triangle files is now used as a test for whether a vertex or edge is on the boundary or interior. This requires that the marker be 0 if and only if the vertex or edge is interior. If you wrote your own `.poly` file and included boundary markers, make sure this requirement is satisfied.

6.7 Earlier versions

Release notes for earlier versions are available upon request.

Chapter 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 Open MPI, 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`, `openmpi`, or `lam` if you use an MPICH, Open MPI, 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
```

If you are using Open MPI, try

```
mpiexec -np 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

$$u = g$$

or

$$\left(c_{xx} \frac{\partial u}{\partial x} + c_{xy} \frac{\partial u}{\partial y} \right) \frac{\partial y}{\partial s} - c_{yy} \frac{\partial u}{\partial y} \frac{\partial x}{\partial s} + c_{bc} u = g$$

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_{bc} 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,number of boundary pieces.`

7.3.5 icons

Subroutine `icons` 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 icons(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) :: icons – 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

$$-\frac{\partial}{\partial x} \left(c_{xx} \frac{\partial u}{\partial x} \right) - \frac{\partial}{\partial x} \left(c_{xy} \frac{\partial u}{\partial y} \right) - \frac{\partial}{\partial y} \left(c_{yy} \frac{\partial u}{\partial y} \right) + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} + c_u u = f \quad \text{in } \Omega$$

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(:, :), cxy(:, :), cyy(:, :), cx(:, :), cy(:, :), c(:, :)` – the functions c_{xx} , c_{xy} , c_{yy} , c_x , c_y and c_u 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) :: 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 3.6.6 and 7.4.9 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 regularity

One strategy for *hp*-adaptive refinement is to provide *a priori* knowledge about the singular nature of the solution. Then h refinement is done near singularities and p refinement is done where the solution is smooth. This strategy is selected with `HP_APRIORI`, and the function `regularity` provides the user's *a priori* knowledge. This function is also used by the Texas 3 Step strategy selected with `HP_T3S`.

In theory, this routine should return the largest value of m such that the solution is in $H^m(T)$, i.e. the derivatives up to order m are in L^2 , where T is the triangle whose vertices are given as input to the function. For multicomponent solutions, it should return the worst (i.e. smallest) such m among the components.

In practice, it can be used to guide refinement in other *a priori* known trouble areas, such as sharp peaks, boundary layers and wave fronts. The actual use is that p refinement is performed if the current degree of the triangle is less than the returned value, and h refinement is performed otherwise. So, for example, if you know some region contains a boundary layer, you could return 3.1 for any triangle that intersects that region to perform h -refinement with cubic elements over the boundary layer.

```
function regularity(x,y)
```

```
real(my_real), intent(in) :: x(3),y(3) – the  $x$  and  $y$  coordinates of
the three vertices of the triangle.
```

```
real(my_real) :: regularity – the returned regularity.
```


7.3.9 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.
```

7.3.10 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.
```

7.3.11 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 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) :: trueys – the return value.

```

7.3.12 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 7.1, 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 so-
lution 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 mod-
ule 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 7.1: 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 7.4.8). 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, nthread, 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, update_umod,  
init_alloc)
```

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

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

```
integer nthread – the number of OpenMP threads in each process. The de-  
fault is whatever is set outside of PHAML, for example via OMP_NUM_THREADS.
```

```
draw_grid_who – which processes should display grid graphics. The grid graph-  
ics 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. An xterm running the slave under the debugger given by `debug_command` is brought up. 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. This is a “hint” which may be obeyed or ignored depending on the implementation of MPI.

`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 ∞ , defined to be `huge(0.0_my_real)`.

character(len=*) triangle_files – in 2D, 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. In 3D, indicates the name of a `.msh` and/or `.geo` file to define the domain and initial grid (see Sect. 4.4)

The default is "domain" in 2D and "domain.msh" in 3D.

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.`

integer init_alloc – initially allocate the grid to be big enough for `init_alloc` vertices per processor. Using a number as large as possible will reduce the time spent in reallocation as the grid grows.

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 and/or derivatives of it at the given points.

```
subroutine phaml_evaluate(phaml_solution, x, y, u, ux, uy, uxx, uyy,  
comp, eigen, z, uz, uzz)
```

`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), optional :: u(:), ux(:), uy(:), uxx(:), uyy(:)` – returned as the computed solution and derivatives at the given points. Those that are present determine what is evaluated. Must have `size(u) == size(x)`, etc.

`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.

`real(my_real), intent(in), optional :: z(:)` – array of z coordinates at which to evaluate the solution. Must be present if and only if PHAML was compiled for 3D problems.

`real(my_real), intent(out), optional :: uz(:), uzz(:)` – returned as the derivatives at the given points. Those that are present determine what is evaluated. Must have `size(uz) == size(x)`, etc.

7.4.7 phaml_get_grid_soln

`phaml_get_grid_soln` returns the vertices of the grid and the solution at those vertices.

```
subroutine phaml_get_grid_soln(phaml_solution, x, y, u, comp, eigen,  
z)
```

`type(phaml_solution_type), intent(in) :: phaml_solution` – the solution from which to get the grid and solution.

`real(my_real), pointer :: x(:), y(:), u(:)` – x and y are returned as the coordinates of the vertices of the triangles (tetrahedra in 3D) of the grid; u is the computed solution at those vertices. They should be disassociated on input and will be allocated to the correct size. You should deallocate them when done with them. To determine the number of points returned, use `size(x)`.

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

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

`real(my_real), pointer, optional :: z(:)` – returned as the z coordinates of the tetrahedra. Must be present if and only if PHAML was compiled for 3D problems.

7.4.8 `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, uxx, uyy, comp, eigen, z, uz, uzz)`

`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, uxx, uyy` – returned as the old solution, x derivative, y derivative, second x derivative and second y derivative at the given point. Any combination of the five 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.

`real(my_real), intent(in), optional :: z` – z coordinate at which to evaluate the solution. Must be present if and only if PHAML was compiled for 3D problems.

`real(my_real), intent(out), optional :: uz, uzz` – returned as the first and second z derivative at the given point. May be present only if PHAML was compiled for 3D problems.

7.4.9 `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 solu-
tion to integrate.
```

```
integer, intent(in) :: kernel – allows you to select among different ker-
nel 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.
```

7.4.10 `phaml_pclose`

`phaml_pclose` closes unit number `unit` on all processors in `phaml_solution`. See also `phaml_popen` in Section 7.4.11.

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

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

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

7.4.11 `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 1024₁₂₈ 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, iostat )
```

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".

integer, intent(out), optional :: iostat – 0 if the file was successfully opened on all processors. -10 if form is present with a value other than "FORMATTED" or "UNFORMATTED". -11 if unit is negative. Any other value is the value of iostat returned by the Fortran OPEN statement.

7.4.12 phaml_query

phaml_query returns values of interest about the grid and solution. All arguments except phaml_solution, comp, eigen and error_estimator are optional and intent(out). Include those you wish to receive as keyword arguments. phaml_solution is mandatory and intent(in). comp, eigen and error_estimator 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_error_indicator,
linf_error_estimate, energy_error_estimate, l2_error_estimate, linf_solution,
l2_solution, energy_solution, linf_u, l2_u, energy_u, linf_true, l2_true,
energy_true, eigenvalues, eigenvalue_error_estimate, eigensolver_eval_errest,
eigensolver_l2_residual, eigensolver_niter, eigensolver_nconv, comp,
eigen, error_estimator)
```

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^∞ 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_error_indicator` – the largest error indicator
`real(my_real) :: linf_error_estimate` – an estimate of the L^∞ norm of the error
`real(my_real) :: energy_error_estimate` – an estimate of the energy norm of the error
`real(my_real) :: l2_error_estimate` – an estimate of the L^2 norm of the error
`real(my_real) :: linf_solution` – the discrete ℓ^∞ norm of the solution vector
`real(my_real) :: l2_solution` – the discrete ℓ^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^∞ 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) :: linf_true` – the continuous L^∞ norm of the true solution, approximated using the vertices and the quadrature points of a sixth order quadrature rule
`real(my_real) :: l2_true` – the continuous L^2 norm of the true solution, approximated with a sixth order quadrature rule
`real(my_real) :: energy_true` – the continuous energy norm of the true solution, approximated with a sixth order quadrature rule
`real(my_real), dimension(num_eval) :: eigenvalues` – the computed eigenvalues, for eigenvalue problems
`real(my_real), dimension(num_eval) :: eigenvalue_error_estimate` – an estimate of the error in each eigenvalue in the continuous problem
`real(my_real), dimension(num_eval) :: eigensolver_eval_errest` – an estimate of the error in each eigenvalue in the discrete eigenvalue problem
`real(my_real), dimension(num_eval) :: eigensolver_l2_residual` – the ℓ^2 norm of the residual of the discrete eigenvalue problem for each eigenpair
`integer :: eigensolver_niter` – number of iterations used by the eigensolver
`integer :: eigensolver_nconv` – number of converged eigenvalues
`integer :: comp` – for multicomponent solutions, which component to report. Energy norms cover all components in a single norm. L^∞ 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.
`integer :: error_estimator` – which error indicator to use for error estimates and error indicators. The default is EXPLICIT_ERRIND.

7.4.13 phaml_restore

`phaml_restore` restores information for `phaml_solution` from files created by subroutine `phaml_store` (see Section 7.4.16).

```
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.

7.4.14 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.

7.4.15 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` and `grid_modified` which are `intent(out)`. It is recommended that the call use keyword arguments for all arguments except possibly `phaml_solution`. Since there is such a large number of arguments to this routine, they are placed

into subsections by category.

```

subroutine phaml_solve_pde(phaml_solution, item, grid_modified,
max_elem, max_vert, max_eq, max_lev, max_deg, stop_on_maxlev, stop_on_maxdeg,
max_refsolveloop, term_energy_err, term_Linf_err, term_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_solver_when,
print_solver_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, solve_init,
keep_linear_system, sequential_vert, inc_factor, error_estimator, errtype,
reftype, refterm, reftol, hp_strategy, t3s_gamma, t3s_eta, t3s_nunif,
t3s_maxref, t3s_maxdeginc, tp_gamma, sp_gamma_h, sp_gamma_p, nlp_max_h_dec,
nlp_max_h_inc, nlp_max_p_dec, nlp_max_p_inc, pbias, refsoln_pbias, derefine,
partition_method, edge_rule, zoltan_param_file, prebalance, postbalance,
petsc_matrix_free, solver, preconditioner, mg_cycles, mg_tol, mg_prerelax,
mg_postrelax, mg_prerelax_ho, mg_postrelax_ho, dd_iterations, krylov_iter,
krylov_restart, krylov_tol, mg_comm, ignore_quad_err, eigensolver, num_eval,
lambda0, st_shift, st_antishift, harmonic_extraction, slepc_true_residual,
scale_evec, eigen_ncv, eigen_maxit, eigen_tol, degree, inc_quad_order,
pde_has_first_order_terms, pde_has_cross_derivative, laplacian_operator,
isosceles_right_triangles, bc_changed, hypre_BoomerAMG_tol, hypre_BoomerAMG_trunc_factor,
hypre_BoomerAMG_strong_threshold, hypre_BoomerAMG_max_row_sum, hypre_BoomerAMG_relax_weight,
hypre_BoomerAMG_outer_relax_weight, hypre_BoomerAMG_cycle_type, hypre_BoomerAMG_max_levels,
hypre_BoomerAMG_max_iter, hypre_BoomerAMG_P_max_elmts, hypre_BoomerAMG_agg_num_levels,
hypre_BoomerAMG_agg_num_paths, hypre_BoomerAMG_num_sweeps_all, hypre_BoomerAMG_num_sweeps_down,
hypre_BoomerAMG_num_sweeps_up, hypre_BoomerAMG_num_sweeps_coarsest, hypre_BoomerAMG_relax_type,
hypre_BoomerAMG_relax_type_down, hypre_BoomerAMG_relax_type_up, hypre_BoomerAMG_relax_type_coar,
hypre_BoomerAMG_relax_order, hypre_BoomerAMG_measure_type, hypre_BoomerAMG_coarsen_type,
hypre_BoomerAMG_interp_type, hypre_BoomerAMG_print_statistics, hypre_BoomerAMG_print_debug,
hypre_Pilut_tol, hypre_Pilut_maxiter, hypre_Pilut_factorrowsize, hypre_Euclid_levels,
hypre_Euclid_bj, hypre_Euclid_print_statistics, hypre_ParaSails_thresh,
hypre_ParaSails_filter, hypre_ParaSails_loadbal, hypre_ParaSails_nlevels,
hypre_ParaSails_sym, hypre_ParaSails_logging, hypre_ParaSails_reuse,
    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_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)

```

7.4.15.1 Miscellaneous arguments

`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 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`.

`logical solve_init` – determines whether or not to compute the solution on the initial grid. Sometimes you might not want to if the grid comes from a previous call.

The default is `.true.`

`logical keep_linear_system` – don't destroy the last linear system before returning.

The default is `.false.`

`logical pde_has_first_order_terms` – indicates whether or not one of c_x or c_y is not identically zero.

The default is `.true.` if either c_x or c_y is nonzero at the midpoint of one of the initial elements, and `.false.` otherwise.

`logical pde_has_cross_derivative` – indicates whether or not c_{xy} is not identically zero.

The default is `.true.` if c_{xy} is nonzero at the midpoint of one of the initial elements, and `.false.` otherwise.

`logical laplacian_operator` – indicates whether or not the PDE operator is the Laplacian, i.e. c_{xx} and c_{yy} are identically one while all other coefficients are identically zero (the right side can be anything).

The default is `.true.` if c_{xx} and c_{yy} are 1.0 at the midpoint of every initial element and c_{xy} , c_x , c_y and c are 0.0 at the midpoint of every initial element, and `.false.` otherwise.

`logical isosceles_right_triangles` – indicates whether or not every triangle in the initial grid is an isosceles right triangle with the peaks at the right angles.

The default is `.true.` if every initial triangle appears to be an isosceles right triangle with the right angle at the third vertex, within roundoff error, and `.false.` otherwise.

`bc_changed` – on calls after the first, setting this to false can save time if it is known that the boundary conditions and true solution (if provided) have not changed since the previous call.

The default is `.true.`

7.4.15.2 Termination arguments

`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`.

`logical grid_modified` – True if and only if changes were made to the grid in this call, excluding changes in the solution value and load balancing.

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

The default is ∞ , defined to be `huge(0)`.

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

The default is ∞ , 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 ∞ , 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, except for some hp strategies where it is p -refined instead. The number of levels is also limited by the size of the hash keys, which is set at the time the PHAML library is compiled (see Section 2.2.1) and the number of element in the initial grid. For 32 bit integers and a very coarse initial grid the maximum is about 25-30 for `PHAML_HASHSIZE=1` and 55-60 for `PHAML_HASHSIZE=2`. If h refinement would cause the hash to overflow, the element is quietly not refined, except for some hp strategies where it is p -refined instead.

The default is ∞ , 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, except for some hp strategies where it is h -refined instead.

In most cases the default is 22, 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`. If `reftype` is `HP_ADAPTIVE` and `hp_strategy` is `HP_NEXT3P`, the default is 19. Otherwise, if `error_estimator` is `LOCAL_PROBLEM_P` or `EQUILIBRATED_RESIDUAL`, or `reftype` is `HP_ADAPTIVE` and `hp_strategy` is `HP_REFSOLN_ELEM` or `HP_REFSOLN_EDGE`, the default is 21. If the element kind from running `mkmkfile.sh` is `tetrahedron` then the default is 10. If the element kind from running `mkmkfile.sh` is `tetrahedron_fixdeg` then if `degree` is present, then the default is `degree`, and if `degree` is not present, then the default is the maximum degree in the current grid.

`logical stop_on_maxlev` – if `.true.`, then `max_lev` is used as one of the termination criteria.

The default is `.false.`

`logical stop_on_maxdeg` – if `.true.`, then `max_deg` is used as one of the termination criteria.

The default is `.false.`

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

The default is ∞ , defined to be `huge(0)`.

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

The default is 0.0.

`real(my_real) term_Linf_err` – terminate when the L^∞ error estimate is less than this value (termination criteria).

The default is 0.0.

`real(my_real) term_L2_err` – terminate when the L^2 error estimate is less than this value (termination criteria).

The default is 0.0.

7.4.15.3 Output control arguments

`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 assembly.

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_solver_when` – how often to produce a printed summary of the solver performance (number of iterations, residual, 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_solver_who` – which processes should print the summary of the solver performance. Valid values are:

`NO_ONE` – don't print.

`SLAVES` – slave processes.

`MASTER` – master process.

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

`MASTER_ALL` – master 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 `trueex` or `trueey` 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, built in conjugate gradient and GMRES solvers, and the PETSc solvers, this causes an estimate of the ℓ^2 norm of the residual to be printed after each iteration. See `KSPDefaultMonitor` in the PETSc documentation. For `GMRES_SOLVER` it is only printed at the restarts.

`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^∞ , and L^2 . The L^∞ 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^∞ norm.

L2_ERR – L^2 norm.

ENERGY_LINF_ERR – energy and L^∞ norms.

ENERGY_L2_ERR – energy and L^2 norms.

LINF_L2_ERR – L^∞ 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^∞ , and L^2 . Valid values are:

NEVER – don't print any.

ENERGY_ERREST – energy norm.

LINF_ERREST – L^∞ norm.

L2_ERREST – L^2 norm.

ENERGY_LINF_ERREST – energy and L^∞ norms.

ENERGY_L2_ERREST – energy and L^2 norms.

LINF_L2_ERREST – L^∞ and L^2 norms.

ENERGY_LINF_L2_ERREST – all three norms.

The default is NEVER.

integer errtype – type of error and error estimates to use in printed output. Valid values are:

ABSOLUTE_ERROR – absolute error.

RELATIVE_ERROR – relative error. Norms of the error are scaled by the norm of the true solution. Error estimates are scaled by the norm of the computed solution.

The default is **ABSOLUTE_ERROR**.

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.

LAST – the time for the last cycle.

FINAL – only at the end.

LAST_AND_FINAL – the time of the last cycle, and the total time.

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 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 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 `.true.`.

`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.`

7.4.15.4 Refinement arguments

`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.

In most cases the default is to use the existing degree in each element. In `phaml_create`, all elements are initialized to have degree 3¹. If the element kind from running `mkmkfile.sh` is `tetrahedron_fixdeg` and `max_deg` is present, then the default is `max_deg`.

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

The default is `.true.`

`integer error_estimator` – select what to use for an error estimate to guide adaptive refinement and estimate global norms of the error. Valid values are:

`EXPLICIT_ERRIND` – error indicator based on the norm of the residual and jump.

`LOCAL_PROBLEM_H` – computes an estimate over a pair of elements by performing one h refinement and solving a local Diriclet residual problem.

`LOCAL_PROBLEM_P` – computes an estimate over a triangle by performing one p refinement and solving a local Neumann residual problem.

`EQUILIBRATED_RESIDUAL` – computes an estimate over a triangle by performing one p refinement and solving a local Neumann residual problem with an equilibrated residual.

`HIERARCHICAL_COEFFICIENT` – use the coefficient of the h -hierarchical basis for linear elements, or the highest order p -hierarchical bases for high order elements. It is significantly faster than the local problem estimates, 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). If `task` is SET_INITIAL then it must be INITIAL_CONDITION.

The default is EXPLICIT_ERRIND, but it is always set to INITIAL_CONDITION if `task` is SET_INITIAL.

`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_NELEM – double the number of elements.

DOUBLE_NEQ – double the number of equations.

HALVE_ERREST – reduce the maximum error indicator by half.

KEEP_NVERT – keep the same number of vertices or, if `max_vert` is present, try to make it close to `max_vert`.

KEEP_NELEM – keep the same number of elements or, if `max_elem` is present, try to make it close to `max_elem`.

KEEP_NEQ – keep the same number of equations or, if `max_eq` is present, try to make it close to `max_eq`.

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

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

ONE_REF_HALF_ERRIND – perform one refinement of all elements with an error indicator larger than half (or $1/\text{inc_factor}$) of the maximum error indicator.

The default is `DOUBLE_NEQ`

`real(my_real) inc_factor` – usually the factor by which to increase the size of the grid during one refinement phase. For *hp*-adaptive refinement with `REFSOLN_EDGE` or `REFSOLN_ELEM`, it is the factor by which the maximum error indicator is divided to get the error indicator cut-off that determines which elements are refined.

The default is usually 2.0. For *hp*-adaptive refinement with `REFSOLN_EDGE` or `REFSOLN_ELEM` it is 10.0.

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

The default is `term_energy_err/2`.

`integer hp_strategy` – select strategy for `reftype=HP_ADAPTIVE`. See Section 3.3.2. Valid values are:

`HP_APRIORI` – use *h* refinement around *a priori* known trouble spots, and *p* refinement elsewhere. The regularity, *m*, of the solution is given by the user in function subroutine `regularity` (Sect. 7.3.8). *h* refinement is used if the degree of the element is larger than *m*.

`HP_NEXT3P` – *hp* strategy in which the regularity is estimated using error estimators computed with spaces of degree *p* + 1, *p* + 2, and *p* + 3.

`HP_PRIOR2P_E` – *hp* strategy in which the regularity is estimated using energy norm error estimators computed with spaces of degree *p* – 1 and *p* – 2.

`HP_PRIOR2P_H1` – *hp* strategy in which the regularity is estimated using *H*¹ error estimators computed with spaces of degree *p* – 1 and *p* – 2.

`HP_TYPEPARAM` – the type parameter strategy.

`HP_BIGGER_ERRIND` – compute both `LOCAL_PROBLEM_H` and `LOCAL_PROBLEM_P` error indicators and refine an element by *h* or *p* depending on which is bigger.

`HP_T3S` – the Texas 3 Step strategy.

`HP_ALTERNATE` – alternate between *h* and *p* adaptive refinement, reducing the error estimate to a prescribed level at each step.

`HP_NLP` – determine what refinements to do using nonlinear programming.

`HP_COEF_DECAY` – strategy based on the decay rate of the *p*-hierarchical coefficients.

`HP_COEF_ROOT` – strategy that estimates the regularity using a root test with the *p*-hierarchical coefficients.

`HP_SMOOTH_PRED` – a strategy that compares the local *a posteriori* error estimate with a predicted error estimate under the assumption that the solution is smooth.

HP_REFSOLN_ELEM – a strategy that considers several possible refinements of an element and picks one that reduces the difference from a reference solution the most.

HP_REFSOLN_EDGE – a strategy that considers several possible refinements of an edge and picks one that reduces the difference from a reference solution the most, and refines elements to be consistent with the chosen edge refinement.

The default is **HP_COEF_DECAY** HP_SMOOTH_PRED.

`real(my_real) tp_gamma` – the type parameter γ for HP_TYPEPARAM.

The default is 0.2.

`integer t3s_nunif` – number of uniform h refinements to start HP_T3S.

The default is 0.

`real(my_real) t3s_gamma` – the parameter γ for HP_T3S and HP_ALTERNATE.

The default is 6.0.

`real(my_real) t3s_eta` – the parameter η for HP_T3S and HP_ALTERNATE.

The default is 0.1.

`integer t3s_maxref` – upper bound on the number of h refinements to perform in one step of HP_T3S.

The default is 3.

`integer t3s_maxdeginc` – upper bound on the number of p refinements to perform in one step of HP_T3S.

The default is 3.

`real(my_real) sp_gamma_h` – parameter γ_h for SMOOTH_PRED.

The default is 4.0.

`real(my_real) sp_gamma_p` – parameter γ_p for SMOOTH_PRED.

The default is 0.4.

`integer nlp_max_h_dec` – maximum number of h coarsenings at one time in HP_NLP.

The default is 1.

`integer nlp_max_p_dec` – maximum number of p coarsenings at one time in HP_NLP.

The default is 1.

`integer nlp_max_h_inc` – maximum number of h refinements at one time in HP_NLP.

The default is 5.

integer nlp_max_p_inc – maximum number of p refinements at one time in HP_NLP.

The default is 2.

real(my_real) pbias refsoln.pbias – bias the HP_REFSOLN_ELEM or HP_BIGGER_ERRIND strategy toward p -refinement. Recommended values are 2.0 for singular or nearly singular solutions and 4.0 for smooth solutions or moderately strong features. 1.0 is no bias.

The default is 4.0.

integer edge_rule – the rule for determining how to assign the polynomial degree for edges. Valid values are:

MINIMUM_RULE – use the minimum of the degrees of the triangles sharing the edge.

MAXIMUM_RULE – use the maximum of the degrees of the triangles sharing the edge.

The default is MINIMUM_RULE.

7.4.15.5 Load balancing arguments

integer sequential_vert – number of vertices in the grid before it starts running in parallel.

The default is 100.

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 refinement. 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 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.

ZOLTAN_FILE – give Zoltan method and parameters in a file.

The default is RTK.

`character(len=*) zoltan_param_file` – name of the file containing parameters for ZOLTAN_FILE. The default is `zoltan.params`.

7.4.15.6 Assembly arguments

`integer inc_quad_order` – increment the order of the quadrature rules by this amount.

The default is 0.

`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), or when using a solver other than HBMG, but will reduce the convergence of the discretization error otherwise.

The default is `.false.` if the solver is HBMG and `.true.` otherwise.

7.4.15.7 Solver arguments

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

MG_SOLVER – the hierarchical basis multigrid method implemented in PHAML.

CG_SOLVER – conjugate gradient solver implemented in PHAML.

GMRES_SOLVER – GMRES solver 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.

PETSC_MUMPS_GEN_SOLVER – Nonsymmetric direct solver from MUMPS.

PETSC_MUMPS_SPD_SOLVER – Symmetric positive definite direct solver from MUMPS.

PETSC_SUPERLU_SOLVER – Direct solver from SuperLU.

PETSC_HYPRE_BOOMERAMG_SOLVER – BoomerAMG algebraic multigrid from *hypre*.

HYPRE_PCG_SOLVER – Preconditioned Conjugate Gradients from *hypre*.

HYPRE_GMRES_SOLVER – GMRES from *hypre*.

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* solvers are replaced by a LAPACK solver for the whole matrix on each processor.)

The default is `MG_SOLVER` in 2D and `PETSC_CG_SOLVER` in 3D.

`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 preconditioner.

`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.
 PETSC_GAMG_PRECONDITION – Algebraic multigrid preconditioner from PETSc.
 PETSC_TRILINOS_ML_PRECONDITION – Algebraic multigrid preconditioner from the ML package from Trilinos.
 PETSC_HYPRE_BOOMERAMG_PRECONDITION – BoomerAMG algebraic multigrid from *hypre*.
 PETSC_HYPRE_PARASAILS_PRECONDITION – ParaSails preconditioner from *hypre*.
 PETSC_HYPRE_PILUT_PRECONDITION – Pilut preconditioner from *hypre*.
 PETSC_HYPRE_EUCLID_PRECONDITION – Euclid preconditioner from *hypre*.
 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=CG_SOLVER` and `solver=GMRES_SOLVER`, the preconditioner must be one of `NO_PRECONDITION` or `MG_PRECONDITION`. 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

`PETSC_SOR_PRECONDITION` – in 3D

`MG_PRECONDITION` – otherwise

`integer mg_cycles` – number of multigrid V-cycles to use in one solution phase or as a preconditioner.

If HBMG is the solver, then the default is 1 if `mg_tol` is `MG_NO_TOL` and infinite otherwise. If HBMG is the preconditioner, then the default is 2.

`real(my_real) mg_tol` – perform multigrid cycles until the ℓ^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, currently 1/1000.

The default is **MG_ERREST_TOL** if HBMG is the solver, and **MG_NO_TOL** if HBMG is the preconditioner.

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 relaxation sweeps to make on each level of the high order p -multigrid cycle before coarse grid correction.

The default is 1.

integer mg_postrelax_ho – number of relaxation sweeps to make on each level of the high order p -multigrid cycle after coarse grid correction.

The default is 1.

integer mg_comm – selects how much communication to do in the parallel hierarchical basis multigrid method. Permitted values are

MGCOMM_FUDOP – the full domain partition approach with two messages per cycle.

MGCOMM_CONVENTIONAL – conventional communication at each level.

MGCOMM_NONE – no communication. Should only be used as a preconditioner.

The default is **MGCOMM_FUDOP** if HBMG is the solver and **MGCOMM_NONE** if HBMG is the preconditioner.

integer krylov_iter – the maximum number of iterations to use with the native Krylov space solvers **CG_SOLVER** and **GMRES_SOLVER**.

The default is 100.

integer krylov_restart – the GMRES restart parameter for **GMRES_SOLVER**.

The default is 20.

real(my_real) krylov_tol – convergence tolerance on the ℓ^2 norm of the linear system residual for the native Krylov space solvers. In addition to positive real numbers, it can have the following values:

`KRYLOV_ERREST_TOL` – reduce the residual to some fraction of the error estimate, currently 1/100.

The default is `KRYLOV_ERREST_TOL`.

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

The default is 1.

`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.

The default is `LAPACK_INDEFINITE_SOLVER`.

`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.`

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

real(my_real)	<code>petsc_richardson_damping_factor</code>
real(my_real)	<code>petsc_chebychev_emin</code>
real(my_real)	<code>petsc_chebychev_emax</code>
integer	<code>petsc_gmres_max_steps</code>
real(my_real)	<code>petsc_rtol</code>
real(my_real)	<code>petsc_atol</code>
real(my_real)	<code>petsc_dtol</code>
integer	<code>petsc_maxits</code>
integer	<code>petsc_ilu_levels</code>
integer	<code>petsc_icc_levels</code>
real(my_real)	<code>petsc_ilu_dt</code>
real(my_real)	<code>petsc_ilu_dtc0l</code>
integer	<code>petsc_ilu_maxrowcount</code>
real(my_real)	<code>petsc_sor_omega</code>
integer	<code>petsc_sor_its</code>
integer	<code>petsc_sor_lits</code>
logical	<code>petsc_eisenstat_nodiagscaling</code>
real(my_real)	<code>petsc_eisenstat_omega</code>
integer	<code>petsc_asm_overlap</code>

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

real(my_real)	hypre_BoomerAMG_tol
real(my_real)	hypre_BoomerAMG_trunc_factor
real(my_real)	hypre_BoomerAMG_strong_threshold
real(my_real)	hypre_BoomerAMG_max_row_sum
real(my_real)	hypre_BoomerAMG_relax_weight
real(my_real)	hypre_BoomerAMG_outer_relax_weight
integer	hypre_BoomerAMG_cycle_type
integer	hypre_BoomerAMG_max_levels
integer	hypre_BoomerAMG_max_iter
integer	hypre_BoomerAMG_P_max_elmts
integer	hypre_BoomerAMG_agg_num_levels
integer	hypre_BoomerAMG_agg_num_paths
integer	hypre_BoomerAMG_num_sweeps_all
integer	hypre_BoomerAMG_num_sweeps_down
integer	hypre_BoomerAMG_num_sweeps_up
integer	hypre_BoomerAMG_num_sweeps_coarsest
integer	hypre_BoomerAMG_relax_type_all
integer	hypre_BoomerAMG_relax_type_down
integer	hypre_BoomerAMG_relax_type_up
integer	hypre_BoomerAMG_relax_type_coarsest
integer	hypre_BoomerAMG_relax_order
integer	hypre_BoomerAMG_measure_type
integer	hypre_BoomerAMG_coarsen_type
integer	hypre_BoomerAMG_interp_type
logical	hypre_BoomerAMG_print_statistics
logical	hypre_BoomerAMG_print_debug
real(my_real)	hypre_Pilut_tol
integer	hypre_Pilut_maxiter
integer	hypre_Pilut_factorrowsize
integer	hypre_Euclid_levels
integer	hypre_Euclid_bj
logical	hypre_Euclid_print_statistics
real(my_real)	hypre_ParaSails_thresh
real(my_real)	hypre_ParaSails_filter
real(my_real)	hypre_ParaSails_loadbal
integer	hypre_ParaSails_nlevels
integer	hypre_ParaSails_sym
logical	hypre_ParaSails_logging
logical	hypre_ParaSails_reuse

integer	hypr_BoomerAMG_MaxLevels
integer	hypr_BoomerAMG_MaxIter
real(my_real)	hypr_BoomerAMG_Tol
real(my_real)	hypr_BoomerAMG_StrongThreshold
real(my_real)	hypr_BoomerAMG_MaxRowSum
integer	hypr_BoomerAMG_CoarsenType
integer	hypr_BoomerAMG_MeasureType
integer	hypr_BoomerAMG_CycleType
integer	hypr_BoomerAMG_NumGridSweeps(:)
integer	hypr_BoomerAMG_GridRelaxType(:)
integer	hypr_BoomerAMG_GridRelaxPoints(:, :)
real(my_real)	hypr_BoomerAMG_RelaxWeight(:)
integer	hypr_BoomerAMG_DebugFlag
real(my_real)	hypr_ParaSails_thresh
integer	hypr_ParaSails_nlevels
real(my_real)	hypr_ParaSails_filter
integer	hypr_ParaSails_sym
real(my_real)	hypr_ParaSails_loadbal
integer	hypr_ParaSails_reuse
integer	hypr_ParaSails_logging
real(my_real)	hypr_PCG_Tol
integer	hypr_PCG_MaxIter
integer	hypr_PCG_TwoNorm
integer	hypr_PCG_RelChange
integer	hypr_PCG_Logging
integer	hypr_GMRES_KDim
real(my_real)	hypr_GMRES_Tol
integer	hypr_GMRES_MaxIter
integer	hypr_GMRES_Logging

7.4.15.8 Eigenvalue arguments

integer eigensolver – What method to use to solve the discrete eigenproblem. Valid values are:

SLEPC_POWER – SLEPc power iteration
 SLEPC_SUBSPACE – SLEPc subspace iteration
 SLEPC_ARNOLDI – SLEPc Arnoldi method
 SLEPC_LANCZOS – SLEPc Lanczos method
 SLEPC_KRYLOV_SCHUR – SLEPc Krylov-Schur method
 SLEPC_GEN_DAVIDSON – SLEPc generalized Davidson
 SLEPC_JACOBI_DAVIDSON – SLEPc Jacobi-Davidson
 SLEPC_LAPACK – SLEPc wrapper to LAPACK
 SLEPC_ARKPACK – SLEPc wrapper to ARPACK
 SLEPC_BLOPEX – SLEPc wrapper to BLOPEX

The default is SLEPC_KRYLOV_SCHUR.

integer num_eval – The number of eigenvalues to find.

The default is 1.

real(my_real) lambda0 – Find the eigenvalues closest to **lambda0**.

The default is **-huge(0.0_my_real)**, i.e., find the smallest eigenvalues.

integer transformation – What spectral transformation to use. Valid values are:

ST_NONE – no spectral transformation
 ST_SHIFT_ORIGIN – shift of origin

ST_FOLD – spectrum folding
 ST_SHIFT_INVERT – shift and invert
 ST_CAYLEY – generalized Cayley

The default is ST_NONE.

real(my_real) st_shift – the shift for the SLEPc spectral transformations
 The default is lambda0.

real(my_real) st_antishift – the antishift for the ST_CAYLEY spectral transformations
 The default is to be the same as st_shift.

integer scale_evec – The eigenvectors can be multiplied (scaled) by an arbitrary constant. These are the choices for scaling:

SCALE_LINF – scale so the (discrete) ℓ^∞ norm is 1. For linear elements, this is also the L^∞ norm.

SCALE_L2 – scale so the (discrete) ℓ^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.

integer eigen_ncv – The number of column vectors to use in the eigensolver
 The default is given in the SLEPc user's guide.

integer eigen_maxit – The maximum number of iterations to use in the eigensolver
 The default is given in the SLEPc user's guide.

real(my_real) eigen_tol – The tolerance on the residual of the eigensystem
 The default is 10^{-10} .

logical harmonic_extraction – If true, use harmonic extraction to accelerate convergence of interior eigenvalues.
 The default is .false.

logical slepc_true_residual – For Krylov solvers use the true residual instead of a cheap approximation.
 The default is .false.

7.4.16 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 solution to store.

`integer, intent(in) :: unit` – the unit number to write to, which should have been opened as either `FORMATTED` or `UNFORMATTED` with subroutine `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.

7.4.17 phaml_store_grid

`phaml_store_grid` stores the grid, and optionally the solution, to data files.

```
subroutine phaml_store_grid(phaml_solution, unit, fmt, comp, eigen,  
noboundary, notags, only_gmsh)
```

`type (phaml_solution_type), intent(in) :: phaml_solution` – the solution containing the grid to store.

`integer, intent(in) :: unit` – the I/O unit number to which to write the grid. It should be open for formatted sequential writing prior to calling `phaml_store_grid` and closed afterwards.

`integer, intent(in) :: fmt` – the format for the output file. Valid values are:

`GRIDFILE_POLY` – for 2D, store just the grid in triangle’s `.poly` format,
<http://www.cs.cmu.edu/~quake/triangle.poly.html>

`GRIDFILE_POLY_SOLN` – for 2D, store both the grid and the solution in triangle’s `.poly` format.

`GRIDFILE_MSH` – for 2D or 3D, store just the grid in Gmsh’s `.msh` format,
<http://geuz.org/gmsh/doc/texinfo/gmsh.html#MSH-ASCII-file-format>

`GRIDFILE_MSH_SOLN` – for 2D or 3D, store both the grid and the solution in Gmsh’s `.msh` format.

`integer, intent(in), optional :: comp` – for multicomponent solutions, which component to store. The default is all of them.

`integer, intent(in), optional :: eigen` – for eigenvalue problems, which eigenfunction to store. The default is all of them.

`logical, intent(in), optional :: noboundary` – if `.true.`, only the elements are saved, not the boundary entities. The default is `.false.`

`logical, intent(in), optional :: notags` – if `.true.`, tags are not saved, except for a tag containing `bmark` for boundary entities. The default is `.false.`

`logical, intent(in), optional :: only_gmsh` – in 3D msh format, if `.true.`, only the elements plus the vertices, edges and faces that were “elements” in the original geo or msh file, and their descendants, are saved. The default is `.false.`

7.4.18 `phaml_store_matrix`

`phaml_store_matrix` stores the stiffness matrix, right hand side, and/or mass matrix in a file in Matrix Market exchange format (<http://math.nist.gov/MatrixMarket/formats.html>). For details, see Section 3.6.2.

```
subroutine phaml_store_matrix(phaml_solution, stiffness_unit, rhs_unit,
mass_unit, inc_quad_order)
```

`type (phaml_solution_type), intent(in) :: phaml_solution` – the solution containing the linear system to store.

`integer, intent(in), optional :: stiffness_unit, rhs_unit, mass_unit` – the I/O unit numbers for writing the stiffness matrix, right hand side, and mass matrix, respectively. They should be open for formatted sequential writing prior to calling `phaml_store_matrix` and closed afterwards. Which ones are present determines which are written.

`integer, intent(in), optional :: inc_quad_order` – increase the order of the quadrature rule by this amount. Default is 0.

Bibliography

- [1] M. Ainsworth and J.T. Oden, *A Posteriori Error Analysis in Finite Element Analysis*, Wiley Interscience Publishers, New York, 2000.
- [2] M. Ainsworth and B. Senior, *hp*-finite element procedures on non-uniform geometric meshes: adaptivity and constrained approximation, in *Grid Generation and Adaptive Algorithms*, M. W. Bern and J. E. Flaherty and M. Luskin, eds, Vol. 113, IMA Volumes in Mathematics and its Applications, Springer-Verlag, New York, (1999), pp. 1-28.
- [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*, **23**, (1), (2001), pp. 15-41.
- [4] P.R. Amestoy, A. Guermouche, J.-Y. L'Excellent and S. Pralet, Hybrid scheduling for the parallel solution of linear systems, *Parallel Computing* **32** (2), (2006), pp. 136-156.
- [5] 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.
- [6] D. Arnold, A. Mukherjee and L. Pouly, Locally Adapted Tetrahedral Meshes Using Bisection, *SIAM Journal on Scientific Computing* **22** (2), (2000), pp. 431-448.
- [7] S. Balay, V. Eijkhout, W.D. Gropp, L. CurfmanMcInnes and B.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., pp. 163-202, Birkhäuser Press, 1997.
- [8] S. Balay, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L. Curfman McInnes, B.F. Smith and H. Zhang, *PETSc Users Manual*, ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.
- [9] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst, *Templates for the*

Solution of Linear Systems: Building Blocks for Iterative Methods, Second Edition, SIAM, Philadelphia, PA, 1994.

- [10] C. Campos, J.E. Román, E. Romero and A. Tomás, SLEPc Users Manual, available at <http://www.grycap.upv.es/slepc>
- [11] 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.
- [12] L. Demkowicz, Computing with hp-Adaptive Finite Elements, Volume 1, One and Two Dimensional Elliptic and Maxwell Problems, Chapman & Hall/CRC, Boca Raton, FL, 2007.
- [13] 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.
- [14] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Zoltan Data Management Services for Parallel Dynamic Applications, *Computing in Science and Engineering*, **4**, (2), (2002), pp. 90–97.
- [15] 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.
- [16] 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. Bjørstad, and A. Tveito, eds., *Lecture Notes in Computational Science and Engineering*, Springer-Verlag, **51**, (2006), pp. 267–294.
- [17] J. Faik, A Model for Resource-Aware Load Balancing on Heterogeneous and Non-Dedicated Clusters, PhD Thesis, Rensselaer Polytechnic Institute, Troy, NY, 2005.
- [18] C. Geuzaine and J.-F. Remacle, Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities, *Int. J. Num. Meth. Eng.*, **79**, 11, (2009), pp. 1309–1331.
- [19] C. Geuzaine and J.-F. Remacle, Gmsh web page, <http://geuz.org/gmsh/>
- [20] W. Gui and I. Babuška, The h , p and h - p Versions of the Finite Element Method in 1 Dimension. Part 3: The Adaptive h - p Version, *Numer. Math.*, **49**, (1986), pp. 659–683.

- [21] P. Houston, B. Senior and E. Süli, Sobolev Regularity Estimation for *hp*-Adaptive Finite Element Methods, Numerical Mathematics and Advanced Applications (Berlin) (F. Brezzi, A. Buffa, S. Corsaro and A. Murli, eds.), Springer-Verlag, 2003, pp. 619–644.
- [22] A. Knyazev, Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method, SIAM J. Sci. Comp., **23**, (2001), pp. 517–541.
- [23] A. Knyazev, BLOPEX web page, <http://www-math.cudenver.edu/~aknyazev/software/BLOPEX/>
- [24] R.B. Lehoucq, D.C. Sorensen, and C. Yang, ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods, SIAM, Philadelphia, 1998.
- [25] X.S. Li and J.W. Demmel, SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems, ACM Trans. Math. Soft., **29**, (2), (2003), pp. 110–140.
- [26] C. Mavriplis, Adaptive Mesh Strategies for the Spectral Element Method, Comput. Methods Appl. Mech. Engrg., **116**, (1994), pp. 77–86.
- [27] J.M. Melenk and B.I. Wohlmuth, On Residual-based a-posteriori Error Estimation in *hp*-FEM, Adv. Comput. Math. **15** (2001), pp. 311–331.
- [28] W.F. Mitchell, Adaptive refinement for arbitrary finite element spaces with hierarchical bases, J. Comp. Appl. Math., **36**, (1991), pp. 65–78.
- [29] W.F. Mitchell, Optimal multilevel iterative methods for adaptive grids, SIAM J. Sci. Statist. Comput., **13**, (1992), pp. 146–167.
- [30] 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.
- [31] W.F. Mitchell, A Parallel Multigrid Method Using the Full Domain Partition, Electronic Transactions on Numerical Analysis, **6**, (1998), pp. 224–233.
- [32] W.F. Mitchell, A Refinement-tree Based Partitioning Method for Dynamic Load Balancing with Adaptively Refined Grids, J. Par. Dist. Comput., **67** (4), (2007), pp. 417–429.
- [33] W.F. Mitchell, The *hp*-Multigrid Method Applied to *hp*-Adaptive Refinement of Triangular Grids, Num. Lin. Alg. Appl., **17** (2010), pp. 211–228.

- [34] W.F. Mitchell and M.A. McClain, A Survey of hp -Adaptive Strategies for Elliptic Partial Differential Equations, Annals of the European Academy of Sciences, in Recent Advances in Computational and Applied Mathematics (T.E. Simos, ed.), Springer, 2011, pp. 227–258.
- [35] W.F. Mitchell and M.A. McClain, A Comparison of hp -Adaptive Strategies for Elliptic Partial Differential Equations (long version), NISTIR 7824, 2011.
- [36] W.F. Mitchell and M.A. McClain, A Comparison of hp -Adaptive Strategies for Elliptic Partial Differential Equations, submitted. Available at <http://math.nist.gov/~WMitchell>
- [37] J.T. Oden and A. Patra, A parallel adaptive strategy for hp finite element computations, Comput. Methods Appl. Mech. Engrg., **121**, (1995), pp. 449–470.
- [38] A. Patra and A. Gupta, A systematic strategy for simultaneous adaptive hp finite element mesh modification using nonlinear programming, Comput. Methods Appl. Mech. Engrg., **190**, (2001), pp. 3797–3818.
- [39] J.R. Rice, Algorithm 625: A Two Dimensional Domain Processor, ACM Trans. Math. Soft., **10**, (1984), pp. 443–452.
- [40] A. Schmidt and K.G. Siebert, a Posteriori Estimators for the $h - p$ Version of the Finite Element Method in 1D, Appl. Numer. Math., **35**, (2000), pp. 43–66.
- [41] J.R. 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, pp. 203–222, Springer-Verlag, Berlin, May 1996.
- [42] J.R. Shewchuk, Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator, <http://www.cs.cmu.edu/~quake/triangle.html>
- [43] P. Šolín, J. Červený and I. Doležal, Arbitrary-level hanging nodes and automatic adaptivity in the hp -FEM, Math. Comput. Simulation **77** (2008), 117–132.
- [44] G. Strang and G. Fix, An Analysis of the Finite Element Method, Prentice-Hall, Englewood Cliffs, NJ, 1973.
- [45] E. Süli, P. Houston and Ch. Schwab, hp -Finite Element Methods for Hyperbolic Problems, in The Mathematics of Finite Elements and Applications X. MAFELAP, J.R. Whiteman (ed.), Elsevier, 2000, 143–162.
- [46] B. Szabo and I. Babuška, Finite Element Analysis, John Wiley and Sons, New York, 1991.

- [47] J.D. Teresco, J. Faik, and J.E. Flaherty, Resource-Aware Scientific Computation on a Heterogeneous Cluster, *Computing in Science & Engineering*, **7** (2), (2005), 40–50.
- [48] J.D. Teresco, zoltanParams: Library for Parsing Zoltan Parameters, <http://www.cs.williams.edu/~terescoj/research/zoltanParams/>
- [49] Trilinos web page, <http://trilinos.sandia.gov/>
- [50] E.L. Wilson, The static condensation algorithm, *Int. J. Num. Meth. Engrg.*, **8**, (1974), pp. 198–203.