The RUF 2.3 User Manual

Philip Keenan

CRPC-TR95538
March 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

The RUF 2.3 User Manual[1]

Philip T. Keenan[2]

March 1, 1995

[1]This research was supported in part by the Department of Energy, the State of Texas Governor's Energy Office, and project grants from the National Science Foundation. The author was also supported in part by an NSF Postdoctoral Fellowship.

[2]Department of Computational and Applied Mathematics, Rice University, P.O. Box 1892, Houston, TX 77251-1892. WWW `http://www.cs.rice.edu/~keenan/`

Contents

Chapter 1

Introduction

This manual describes the user interface to `RUF`, the Rice Unstructured Flow program. RUF solves scalar linear second order elliptic equations on general unstructured meshes in two or three space dimensions using mixed finite element methods [1, 2, 3]. It is applicable to steady state flow calculations in porous media, such as arise in petroleum reservoir simulation and groundwater contaminant modeling. Extensions to nonlinear time dependent systems such as arise in multi-phase flow and transport are under development.

Version 2.3 uses the *kScript* application scripting language [8] as the user interface programming language. It also supports stochastic tensor coefficient fields and averaging methods, and Robin (Type III) boundary conditions. Version 2.0 added parallel computation and support for domain decomposition algorithms. Version 1.0 included many features including a command based user interface, full tensor coefficients, general boundary conditions, and general reference solutions for doing convergence studies, and general unstructured meshes.

1.1 What is RUF?

RUF models the flow of fluid through a porous medium. It can be applied to the study of both petroleum reservoirs and groundwater aquifers. RUF models single phase flow; future programs are planned which will handle multi-phase, multicomponent flow and transport. Reservoirs and aquifers are modeled geometrically as a grid or mesh consisting of polygonal or polyhedral elements. Unlike some models which require rectangular or brick elements arranged in a two or three dimensional lattice, RUF allows using unstructured meshes in which elements of a variety of shapes may be combined without restriction. In particular, it supports tetrahedral elements in three dimensions, as well as hexahedra and bricks, and triangular elements in two dimensions, as well as quadrilaterals and rectangles.

RUF has a number of other features. It handles general boundary conditions. It allows general permeability tensors, not just diagonal ones. It can refine the meshes it builds. It can set up test problems with known analytic solutions from a wide range of polynomial and non-polynomial reference solutions. It can output text or graphics files describing the mesh, the solution, the gradient and the flux; and when using a known reference solution it can also compute and plot errors and their norms. It allows wells to be represented directly as sources or sinks within elements; one can also use face boundary conditions to model a variety of wells. RUF also includes a flux postprocessor for triangular and tetrahedral elements, which increases the accuracy of the computed fluxes on unstructured meshes, as described in [6, 4].

RUF uses a very simple mesh geometry input language. While simple meshes can be constructed by hand, RUF is intended to be used in conjunction with a commercial mesh generation package. Any package will do, as long as its output mesh description can be translated into RUF's input format. The problem of chopping up a general unstructured mesh in an optimal way for parallel computation is a very difficult one computationally. In parallel settings RUF therefore expects a commercial mesh generator to supply an already chopped mesh, or else chops the mesh itself in a quite simplistic way.

1.1.1 Disclaimer

RUF is a research tool, not a commercial product. It is made available to other researchers subject to the following restrictions and disclaimers.

with him.

The user interface to this program was built with 'kScript', a flexible user interface generator and application scripting language, which is part of Philip T. Keenan's C++ Foundation Class Library.

While RUF has been extensively tested on hundreds of convergence studies with all sorts of meshes and other options, it is entirely possible that bugs remain. Bug reports may be sent to the author at `keenan@rice.edu`, but there is no promise that they will be fixed — the author must primarily concentrate on publishing papers so he can get tenure, so he is currently unable to provide support for the program.

1.2 Supported Platforms

RUF is written entirely in C++ and builds on the Keenan C++ Foundation Class Library, which is subject to the same terms and conditions as RUF. In the experience of the author [5], C++ currently provides a powerful and efficient mechanism for writing highly complex software for scientific computation. The resulting code is very portable and has been run on a variety of machines including Sun Sparc workstations, IBM RS6000 workstations, the Intel Hypercube and the Intel Paragon. The last two machines mentioned are distributed memory parallel supercomputers; RUF can read parallel mesh specifications, or distribute a sequential mesh (in a simplistic way) across multiple processors, to achieve substantial speedups in a parallel environment.

RUF does not require a native C++ compiler for the target machine: after using the standard `cfront` C++ to C translator, the resulting C code can be compiled on any machine with an ANSI C compiler.

On many modern machines C/C++ and FORTRAN achieve the same level of efficiency in numerical computation [5]. However, in the event that you wish to run on a machine for which the manufacturer did not put the same effort into the C compiler as the FORTRAN one, the Keenan C++ Foundation Class Library allows you to represent vectors and matrices in FORTRAN format, so that you can link with any previously developed FORTRAN linear algebra routines you wish. Indeed, FORTRAN and C/C++ can be mixed throughout the code, subject to the restriction that FORTRAN only understands very simple data structures and will therefore have a hard time with the trees and other pointer based data structures found in much of RUF.

RUF has been written in a modular style which should be easy to extend and modify. However, this manual addresses only the user interface and does not attempt to describe the source code itself. The user interface is flexible enough to describe a wide variety of application scenarios.

Chapter 2

The Single Phase Flow Equation

RUF 2.3 consists of a series of libraries which may be linked to form four executable program versions. The choices are between 2-D and 3-D, and between the hybrid mixed finite element method and the extended mixed finite element formulation.

The programs all use the lowest order Raviart-Thomas approximating spaces, just like the cell centered finite difference method when that is viewed as a mixed method with quadrature.

The programs are otherwise identical and build on a substantial C++ library of tools for partial differential equations, general geometry, linear algebra and user interfaces. The various numerical methods are defined in detail in [1, 2, 3], which also presents numerical examples and explains which methods are preferred. RUF 1.0 [7] included versions for the saddle-point formulation of the mixed finite element method, which has been dropped in subsequent versions because it is too inefficient. The stencil version in RUF 1.0 is now part of the enhanced or extended version in RUF 2.3.

All the versions solve the scalar linear elliptic partial differential equation

$$-\nabla \cdot (K(\mathbf{x})\nabla p(\mathbf{x})) + \alpha(\mathbf{x})p(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega.$$

In two dimensions, Ω is a two dimensional polygonal region defined by triangles, rectangles and quadrilaterals; in three dimensions it is a polyhedral region defined by tetrahedra or by hexahedra and bricks. Ω need not be convex; it also need not be simply connected; for instance wells may be represented as actual holes drilled in the domain.

α is a non-negative function of position. K is a symmetric positive definite tensor function of position. The scalar p represents a potential and is called pressure; $\mathbf{u} = -K\nabla p$ is called velocity. If $\hat{\mathbf{n}}$ is a unit normal to an edge then $\mathbf{u} \cdot \hat{\mathbf{n}}$ is called

the normal flux across the edge in the direction of the normal. On each external boundary edge, one of the following three boundary conditions must be supplied: either a scalar boundary condition

$$p(\mathbf{x}) = p_0(\mathbf{x}),$$

or a flux boundary condition

$$\mathbf{u}(\mathbf{x}) \cdot \hat{\mathbf{n}} = g_0(\mathbf{x}),$$

or a mixed boundary condition

$$\mathbf{u}(\mathbf{x}) \cdot \hat{\mathbf{n}} = g_0(\mathbf{x})(p(\mathbf{x}) - p_0(\mathbf{x})).$$

Note that all three types of boundary conditions may be mixed throughout the boundary of Ω. The boundary functions p_0 and g_0, the coefficient tensor K, and the source/sink function f must all be specified. The programs also allow "wells" to be specified as point sources and sinks within elements.

Chapter 3

Input Commands

The user interface reads commands from a *kScript* input file. *kScript* is a complete programming language with comments, numeric and string variables, looping, branching and user defined commands. It includes predefined commands for online help, include file handling, arithmetic calculations and string concatenation, and communication with the UNIX shell. Applications can define additional commands and objects which enrich the vocabulary and power of *kScript*.

For a complete introduction to *kScript*, see the *kScript User Manual* [8]. The present manual primarily discusses those additional commands and objects defined by RUF.

For complete and up-to-date lists of commands and objects, run the program to access on-line help. Type

```
help
```

to get started.

Commands specific to RUF are listed below. Each command's name is followed by a list of arguments. Most arguments consist of a type name and a descriptive name, enclosed in angled brackets. These represent required arguments that must be of the stated type.

Arguments enclosed in single square brackets are optional prepositions. They can be used to create English sentence-like scripts which are easy to read, or they can be omitted with no change in the meaning of the script. Sometimes several alternatives are listed, separated by a vertical bar.

Arguments enclosed in double square brackets are optional keywords which do change the meaning of the script if they are supplied. Arguments enclosed in triple square brackets are lists of alternative keywords, exactly one of which must be used. Occasionally, triple square brackets contain type names rather than literal keywords.

In *kScript*, a space-delimited sharp or pound symbol comments out the rest of the line on which it occurs. Mathematical expressions must be written with no internal

spaces. String expressions must either have no internal spaces or be enclosed in curly braces. In all other contexts, white space (spaces, tabs, line breaks, and so on) serves only to delimit commands and their arguments.

3.1 Mesh Specification Commands

`v <char* name> <double x> <double y> <double z>`
Define a vertex by its coordinates.

`e <char* name> <char* id1> <char* id2>`
Define an edge by its vertices.

`tri <char* name> <char* id1> <char* id2> <char* id3>`
Define a triangle by its edges.

`rect <char* name> <char* id1> <char* id2> <char* id3> <char* id4>`
Define a rectangle by its edges.

`quad <char* name> <char* id1> <char* id2> <char* id3> <char* id4>`
Define a quadrilateral by its edges.

`tetra <char* name> <char* id1> <char* id2> <char* id3> <char* id4>`
Define a tetrahedron by its faces.

`brick <char* name> <char* id1> <char* id2> <char* id3> <char* id4> <char* id5> <char* id6>`
Define a brick by its faces.

`hexahedron <char* name> <char* id1> <char* id2> <char* id3> <char* id4> <char* id5> <char* id6>`
Define a hexahedron by its faces.

3.2 RUF Commands

`overlap <intArray procNumbers>`
Specify the set of processors sharing subsequent mesh objects. The first one listed is the owner for subsequent elements. The first two listed are the owner and other for subsequent faces; use -1 for the other on exterior boundary faces.

`endOverlap`
This must end each overlap section.

`distribute <distributionMethod* method>`
As an alternative to using the overlap command, use this command after globally reading a coarse mesh, to create a default processor assignment to elements.

`pad [by] <int nLevels>`
After calling distribute and possibly subdividing, call pad to specify 0 or more layers worth of padding for each processor's subdomain.

`bndy <bndyConds* bc> <mathExpr value>`
Select the type of boundary condition to impose on subsequent faces.

`setBndy [of] <stringExpr faceName> [=|to] <bndyConds* bc> <mathExpr value>`
Modify the boundary condition for a given face.

`pin`
If when using all flux boundary conditions, the solver does not converge, use this command to set one boundary to scalar=0.

`unpin`
This clears the effect of the pin command. Do this before subdividing a pinned mesh, otherwise after subdivision, multiple faces will share the scalar=0 condition.

`force [by] <mathExpr value>`
Specify a right-hand-side forcing value for subsequent elements.

`setForce [of] <stringExpr eltName> [to|=] <mathExpr value>`
Modify the right-hand-side forcing value for a given element.

`alpha [is] <mathExpr value>`
Specify a lowest-order-term coefficient value for subsequent elements.

`setAlpha [of] <stringExpr eltName> [=|to] <mathExpr value>`
Modify the lowest-order-term coefficient value for a given element.

`tensor [is] <doubleArray tensor>`
Specify the components of the symmetric coefficient tensor for subsequent elements.

`setTensor [of] <stringExpr eltName> [to|=] <doubleArray tensor>`
Modify the components of the symmetric coefficient tensor for a given element.

`subdivide <mathExpr N> [times]`
Globally subdivide the mesh N times.

`solve`
Solve the partial differential equation.

`solnMethod <solnMethods method>`
Select a solution method.

`preCond <preconditioners method>`
Select a preconditioner.

`postProc <postprocessors method>`
Select a method for postprocessing fluxes.

`colorRange <ranges* r>`
Select the range of values to show in color.

`plot <transforms* t> <colorVariables* var>`
Append a plot of the specified variable to the plot file.

`write <transforms* t> <colorVariables* var>`
Append transformed values to the log file.

`norms <transforms* t> <colorVariables* var>`
Append norms of the indicated variable to the log file.

`up`
Go up one level to a coarser mesh solution.

`down`
Go down one level to a finer mesh solution.

`averageTensor <tensorAveragingMethod* ave>`
Use the finer mesh solution to construct an averaged permeability tensor at the current mesh level.

`stochastic <stochasticMethod* method>`
Stochastically generate a tensor coefficient field.

`label <labels* obj>`
(2-D only) Label the specified top level objects in the current plot.

`plotCommands <stringExpr text>`
(2-D only) Append low level kplot commands to the plot file.

`redirect <outputFiles* file> [to] <stringExpr newFileName>`
Redirect an output file. Use '-' to restore the original output file as specified on the command line.

`refSoln <refSolnKinds* kind> <intArray maxIndices> <doubleArray coefficients>`
Specify a reference solution.

`info`
Print information about this program.

`showState`
Print the internal state variables in input format.

`dump`
Print the internal solution arrays.

`well <stringExpr name>`
Define a well. Use the wellHelp command for detailed information, since well definitions depend on the kind of PDEs being solved.

`wellHelp`
Explains how to define wells in the context of this program. Well definitions depend on the kind of PDEs being solved.

3.3 Arithmetic and String Variables

Many commands take arithmetic or string expressions as arguments. Math expressions can mix numbers, arithmetic and logical operators, and symbolic names. String expressions are enclosed in curly braces and can expand references to other string or numeric variables by preceding their names with a percent sign. Symbolic names can represent constant or variable values. Predefined ones are listed below; users can define additional ones using the `define` and `set` commands.

`verbosity`
0, 1, 2, ... produce increasingly detailed debugging information.

`dimension`
The number of space dimensions. (Constant)

`iterRelTol`
Relative error tolerance to use as stopping criterion in iterative solution processes.

`nElts`
The current number of elements in the global mesh. (Constant)

`nFaces`
The current number of faces in the global mesh. (Constant)

`lElts`
The current number of owned elements in the local mesh. (Constant)

`lFaces`
The current number of owned faces in the local mesh. (Constant)

`nProc`
The number of processors running the application. (Constant)

`use_quadrature`
If true, use the trapezoidal rule on rectangles, quadrilaterals, hexahedra and bricks, in the tensor inner product; otherwise use exact integration.

Currently, the mesh element and face counts are only available after a `solve` command. Future versions may make the `subdivide` command update them as well.

3.4 Argument Types

The formal argument types in command descriptions generally correspond to C++ classes. The actual argument must be in the correct format for the specified type. For an explanation of the syntax for a particular type X, use the command `describe type X`. The command

```
describe all types
```

will list all of the type names for which on-line help is available. Some of these types are keyword types, which are lists of alternatives. These are described in the following subsections.

3.5 Boundary Conditions

An argument of type `bndyConds` can take any of the following values.

`scalar`
p0 : Type I (Dirichlet) condition: p = p0.

`flux`
g0 : Type II (Neumann) condition: u*n = g0.

`mixed`
k, p0: Type III (Robin) condition: u*n = k(p - p0).

In the expression $u * n$, n is the unit outward normal to $\partial\Omega$. In plots of boundary conditions, scalar faces are colored according to p_0, flux faces have a normal vector

drawn from their center with length based on g_0, and mixed faces do both, with the face colored by p_0 and the vector based on the case where $p - p_0 = 1$.

3.6 Variable Names for plot, write and norm

An argument of type `colorVariables` can take any of the following values.

`mesh`
: This draws the mesh.

`edges`
: 2-D only: This draws the mesh edges in black.

`measure`
: The measure (area or volume) of each element.

`map`
: The determinant of each element's map to the reference element.

`regularity`
: For each element, the shortest edge length divided by the longest.

`bndy`
: The boundary conditions on each face.

`force`
: This displays the forcing term value for each element.

`alpha`
: This displays the lowest order coefficient value for each element.

`stoch`
: The stochastic value for each element.

`tensor`
: The determinant of the coefficient tensor for each element.

`tensor11`
: The 1,1 entry of the tensor.

`tensor12`
: The 1,2 or 2,1 entry of the tensor.

`tensor22`
: The 2,2 entry of the tensor.

`tensor13`
: The 1,3 entry of the tensor in 3-D.

`tensor23`
: The 2,3 entry of the tensor in 3-D.

`tensor33`
: The 3,3 entry of the tensor in 3-D.

`scalar`
: The scalar solution at the element center.

`gradient`
: The gradient of the scalar solution: a vector at the element center.

`flux`
: The flux: the tensor coefficient times the gradient: a vector at the element center.

`normalFlux`
: The normal component of flux across each face: a vector at the face center.

`divergence`
: The divergence of the flux.

There is also an `owner` color variables which can be used to display the owning processor number of each element in a parallel mesh distribution.

3.7 Labels in plots

An argument of type `labels` can take any of the following values.

`vertices`
: Vertex names.

`edges`
: Edge names.

`faces`
: Faces names.

`boundary`
: Boundary faces only.

`elements`
: Element names.

`wells`
: Label wells.

Note: a label may be optionally preceded by a processor number, in which case only that portion of the mesh seen by that processor will be labeled. When a mesh is parallelized with the distribute and pad commands, however, any label commands must be executed before the pad command, so the `owner` color variable should be used instead to determine element ownership.

3.8 Output streams

An argument of type `outputFiles` can take any of the following values.

`plot`
: The plot file.

`log`
: The log file.

`out`
: The standard output.

3.9 Flux Postprocessors for Triangles and Tetrahedra

An argument of type `postprocessors` can take any of the following values.

`none`
: Use the ordinary RT-0 fluxes.

`l2`
: Use the Keenan-Dupont linear least squares flux postprocessor.

`l2_div`
: Use the Keenan-Dupont divergence preserving linear least squares flux postprocessor.

Other postprocessors may be defined as well; consult the online documentation for complete details.

3.10 Solver Preconditioners

An argument of type `preconditioners` can take any of the following values.

`none`
: No preconditioning.

`inherit`
: Use the previous coarser mesh solution as an initial guess.

Other preconditioners may be defined as well; consult the online documentation for complete details. Some preconditioning methods may be experimental in concept or implementation.

3.11 Reference solution kinds

An argument of type `refSolnKinds` can take any of the following values.

`poly`
: Sum of polynomials, like Taylor series.

`trigSum`
: Sum of trigonometric functions, like Fourier series.

`trigProd`
: Sum of products of trigonometric functions.

`bump`
: Sum of bump functions on an integer lattice.

`exp`
: Sum of exponential functions.

`property`
: Use the supplied mesh properties to determine the forcing term, coefficients and boundary data. No analytic reference solution is used.

`user`
: Use user defined functions for the forcing term, coefficients, boundary data and reference solution.

`approx`
: This uses the finest mesh solution as the reference solution. Use the 'up' command to move to a coarser mesh before calling 'norm'.

The `user` case only works if user defined coefficient functions have been written and linked into the code. The `property` case makes the forcing term f a piecewise constant function defined by the element property values, and makes the boundary functions p_0 and g_0 piecewise constants defined by the property values of boundary faces.

In addition to being used when the `user` reference solution has been chosen, externally linked user defined reference solution subroutines are called to provide reference solution values when the `property` solution is selected, so that the user may supply approximate analytic solutions for comparison. The user solution module

includes a hook for posting additional commands or variables to serve as parameters to these functions.

The array arguments to the `refSoln` command are omitted for the `property` and `approx` cases. To use the `approx` choice, first select another reference solution type so that appropriate boundary conditions (and right hand side function) are available during execution of the `solve` command. Then switch to the `approx` method before computing norms. The `approx` case compares the coarse mesh solution on an element or face to the *average* of the corresponding solution values on the finest available mesh.

The remaining cases, listed below, specify an analytic solution of the form (written here for n space dimensions)

$$p(x_1, \ldots, x_n) = \sum_{i_1=0}^{k_1} \cdots \sum_{i_n=0}^{k_n} c_{i_1 \ldots i_n} f_{i_1 \ldots i_n}(x_1, \ldots, x_n).$$

Here f is a primitive function selected by name:

`poly`

$$f_{i_1 \ldots i_n}(x_1, \ldots, x_n) = \prod_{k=1}^{n} {x_k}^{i_k}.$$

`trigSum`

$$f_{i_1 \ldots i_n}(x_1, \ldots, x_n) = \cos\left(\sum_{k=1}^{n} i_k x_k\right).$$

`trigProd`

$$f_{i_1 \ldots i_n}(x_1, \ldots, x_n) = \prod_{k=1}^{n} \cos\left(i_k x_k\right).$$

`bump`

$$f_{i_1 \ldots i_n}(x_1, \ldots, x_n) = \left(1 + \sum_{k=1}^{n} (x_k - i_k)^2\right)^{-1}.$$

`exp`

$$f_{i_1 \ldots i_n}(x_1, \ldots, x_n) = \exp\left(\sum_{k=1}^{n} i_k x_k\right).$$

The second argument to the `refSoln` command is the array of maximum indices $\{k_1 \ldots k_n\}$. The coefficients $c_{i_1 \ldots i_n}$ are supplied by the second array argument to the `refSoln` command, as a linear array, ordered lexicographically, as in the nested loop sequence

```
for i1 = 0 to k1
  for i2 = 0 to k2
    ...
      for in = 0 to kn
        read c(i1,...,in)
```

3.12 Stochastic Tensor Generation Methods

An argument of type `stochasticMethod` can take any of the following values.

`prepare`
: This should be used first, and after any subdivision, to ensure each element has its own stochastic value and tensor to adjust.

`scalar`
: mathExpr : This adjusts the stochastic value of each element using the supplied formula. You can use 'x', 'y', and 'z' as the element center coordinates, and 'val' for the current stochastic value,

`tensor`
: mathExpr11 mathExpr12 ... mathExprNN : This adjusts the tensor value in each element. The 3 (in 2-D) or 6 (in 3-D) formulas define the entries of the symmetric tensor and can use the same variables as in the scalar case.

3.13 Tensor Averaging Methods

An argument of type `tensorAveragingMethod` can take any of the following values.

`arithmetic`
: Arithmetic average.

`multiplicative`
: Multiplicative average (pseudo geometric mean).

`kw`
: The new Keenan-Wheeler averaging method.

Tensor averaging methods, also known as up-scaling or homogenization methods, are a focus of current research. Consult the on-line documentation for possible additional methods.

3.14 Transformations for variables in plot, write and norm commands

An argument of type `transforms` can take any of the following values.

`id`
: This is the default. No transformation is applied to the computed solution.

`ref`
: Use the reference solution rather than computed solution.

`err`
: The error = computed - reference solution.

`relErr`
: The relative error = (computed - reference)/reference.

`abs`
: The absolute value.

`log10Abs`
: The base-10 logarithm of the absolute value.

The `transforms` argument to the `plot`, `write` and `norms` commands is optional and defaults to `id`. Moreover, two transforms may be given instead of one, if the combination makes sense. For example,

```
abs err
```

yields the absolute error in whatever color variable is specified.

3.15 Color ranges

An argument of type `ranges` can take any of the following values.

`auto`
: The full range present in the data is used.

`range`
: min max : Values outside this range are converted to black.

3.16 Solution Methods

An argument of type `solnMethods` can take any of the following values.

`default`
: The default method for this program.

Other solution methods may be defined as well; consult the online documentation for complete details. For instance, the `enhanced` method defines three solution methods which control which faces receive Lagrange multipliers (see [1] for details).

`all-faces`
: Use multipliers on all faces.

`top-level`
: Use multipliers on top level mesh faces, and their refinements, only.

`external`
: Use multipliers on external boundary faces, and processor subdomain boundaries, only.

3.17 Parallel mesh distribution methods

An argument of type `distributionMethod` can take any of the following values.

`block`
: Assign elements to processors in large blocks by index.

`cyclic`
: Assign elements to processors by taking indices mod the number of processors.

Programmers can add custom mesh distributions easily, simply by supplying a function mapping elements to processor numbers.

Chapter 4

Some Mathematical Details

This section summarizes the hybrid formulation of the mixed finite element method and its implementation in RUF. For a more in-depth treatment of both the hybrid and enhanced methods, see [1].

The programs solve the elliptic partial differential equation

$$-\nabla \cdot (K(\mathbf{x})\nabla p(\mathbf{x})) + \alpha(\mathbf{x})p(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega,$$

with boundary conditions

$$p(\mathbf{x}) = p_0(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_{II},$$
$$\mathbf{u}(\mathbf{x}) \cdot \hat{\mathbf{n}} = g_0(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_I,$$
$$\mathbf{u}(\mathbf{x}) \cdot \hat{\mathbf{n}} = g_0(\mathbf{x})(p(\mathbf{x}) - p_0(\mathbf{x})), \quad \mathbf{x} \in \partial\Omega_{III},$$

where the three boundary subsets partition all of $\partial\Omega$. Here $\mathbf{u} = -K\nabla p$.

Let $(\cdot,\cdot)$ denote the L^2 inner product on Ω, and $<\cdot,\cdot>$ the L^2 inner product on $\partial\Omega$.

Multiplying by suitable test functions, integrating, applying the divergence theorem, one has the system:

$$(K^{-1}\mathbf{u}, \mathbf{v}) = (p, \nabla \cdot \mathbf{v}) - < p, \mathbf{v} \cdot \hat{\mathbf{n}} >,$$
$$(\nabla \cdot \mathbf{u}, w) = (f, w).$$

Under appropriate assumptions, this system of equations is equivalent to the original partial differential equation. In particular, $p \in L^2$ and $\mathbf{u} \in H(\mathrm{div})$, the system is equivalent provided it holds for all $\mathbf{v} \in H_0(\mathrm{div})$ and all $w \in L^2$. Here $H_0(\mathrm{div}) = \{\mathbf{v} \in H(\mathrm{div}) : \mathbf{v} \cdot \hat{\mathbf{n}} = 0 \text{ on } \partial\Omega_I \cup \partial\Omega_{III}\}$. Thus the term $< p, \mathbf{v} \cdot \hat{\mathbf{n}} >$ becomes $< p_0, \mathbf{v} \cdot \hat{\mathbf{n}} >_{\partial\Omega_{II}}$.

Now let $\{w_e : e \in E\}$ be a basis for a finite dimensional subspace of L^2, and $\{\mathbf{v}'_f : f \in F\}$ be a basis for a suitable corresponding finite dimensional subspace of $H(\mathrm{div})$. In RUF 2.3 we use the lowest order Raviart-Thomas spaces corresponding to a decomposition of Ω into mesh elements, so E is the set of elements, F is the set of faces, the w_e are piecewise constants element by element and the $\mathbf{v}'_f$ are certain discontinuous piecewise linear vector functions with continuous normal components across element faces. The support of w_e is the element e, and the support of $\mathbf{v}'_f$ consists of the at most two elements sharing face f.

To describe the hybrid formulation of the mixed finite element method we also introduce basis functions μ_f which piecewise constants on faces; the support of μ_f is face f. Next, we define a preferred normal direction to each face, and say that each interior face consists of two semi-faces, one for each element on either side of the face. We then define basis functions $\{\mathbf{v}_s : s \in S\}$, where S is the set of semi-faces. Each $\mathbf{v}_s$ is the restriction to e of $\mathbf{v}'_f$, where semi-face s is on the element e side of face f. These new basis functions have support in only one element each, and are fully discontinuous.

We write $[\mathbf{v}_s \cdot \hat{\mathbf{n}}]_f$ to mean the jump in the normal component of $\mathbf{v}_s$ across face f. We also let F_0 be the set of interior faces, and F_1, F_2, F_3 the sets of type I, II, and III boundary faces, respectively.

In the hybrid formulation, the unknowns are

$$\mathbf{U} = \sum_{s \in S} U_s \mathbf{v}_s,$$

$$P = \sum_{e \in E} P_e w_e,$$

and

$$\lambda = \sum_{f \in F} \lambda_f \mu_f.$$

The unknown coefficients U_s, P_e and λ_f must satisfy

$$(K^{-1}\mathbf{U}, \mathbf{v}_s) = (P, \nabla \cdot \mathbf{v}_s) - \sum_{f \in F} < \lambda, [\mathbf{v}_s \cdot \hat{\mathbf{n}}]_f >_f,$$

for all $s \in S$,

$$(\nabla \cdot \mathbf{U}, w_e) + (\alpha P, w_e) = (f, w_e),$$

for all $e \in E$,

$$< [\mathbf{U} \cdot \hat{\mathbf{n}}], \mu_f >_f = 0,$$

for all $f \in F_0$,

$$\lambda_f = p_0(f),$$

for all $f \in F_1$,

$$< \mathbf{U} \cdot \hat{\mathbf{n}}, \mu_f >_f = < g_0, \mu_f >_f,$$

for all $f \in F_2$, and

$$< \mathbf{U} \cdot \hat{\mathbf{n}}, \mu_f >_f - < g_0 \lambda, \mu_f > = < g_0 p_0, \mu_f >_f,$$

for all $f \in F_3$.

In matrix form, this discrete system takes the form

$$MU - BP + L\lambda = 0,$$

$$B^T U + AP = F,$$

$$L_a^T U + G_3 \lambda = G_2,$$

and

$$\lambda_b = p_0,$$

where L_a denotes the columns of L that are not associated with type I boundary faces, and λ_b denotes the rows that are associated with type I boundary faces. Note that G_2 and G_3 have non-zero entries only in association with Type II or III boundary faces, respectively.

One can eliminate all the unknowns except λ_a, which results in a sparse, symmetric and positive definite system for the λ_a, which can be solved using for instance Conjugate Gradients.

For additional details and a description of the relationship between mixed finite element methods and cell centered finite differences, see [1].

Chapter 5

Running the Programs

5.1 Command Line Arguments

Executing a command like

```
elliptic_hybrid_2d -usage
```

will bring up a complete list of the command line options and C-shell environment variables used by the program. In particular, the `-echo` option displays input commands as they are processed, which may help with debugging input files. The standard command line is

```
elliptic_X_Nd inputFileName plotFileName logFileName
```

Using - in place of a file name makes the program read from the keyboard or send output to the screen, which also happens if the output files are omitted. Two dimensional versions of RUF write graphics files suited for display with Phil Keenan's `kplot` program, which runs under X-11/Motif on workstations; three dimensional versions write graphics files designed for Wavefront's commercial *Data Visualizer* program, which runs on Silicon Graphics machines. Both kinds of graphics files are simply text files in a straightforward format, so user conversion for other display programs should be possible. Moreover, the `write` command provides an easy way to write out the solution in numeric form, which (at least when using rectangular grids) can then be imaged using commercial products such as Matlab.

5.2 Advanced Features

5.2.1 Parallelizing Mesh Descriptions

This section explains how to take a sequential mesh description and decompose it for use with RUF 2.3 for computation on a parallel architecture.

There are two ways to decompose the mesh. The easy way is to use the `distribute` and `pad` commands. This requires that the top level (coarsest) mesh be small enough that a complete copy fits on each processor. The `distribute` command then assigns each element to a unique "owner" processor, currently in a very simplistic way. The `pad` command then deletes all mesh objects on a given processor not needed for that processor's subdomain, which is the owned elements plus any padded ones plus supporting faces, edges and vertices. The result is reasonable for experimentation, but the subdomains may be far from optimal in terms of surface to volume ratio, so this is mainly for simple cases and timing studies.

The number of layers of padding to use depends on the solver algorithm. In RUF 2.3, none of the solvers require an overlap region and some might fail if one is used, so `pad 0` is the appropriate command. Future versions may incorporate other preconditioning schemes or solver algorithms which do require additional levels of padding.

The harder, but more general, way to specify a parallel mesh requires the user (or mesh generator) to determine what the subdomains should look like. The program is then informed of the decomposition through the `overlap` command. The `overlap` command is used on vertices, edges, faces, and elements to tell the processors which ones need to pay attention to the following objects, until a matching `endOverlap` is encountered.

To understand arbitrarily overlapping subdomains, first picture the complete mesh, with each element assigned to a unique processor called its "owner". Every face has two element neighbors, unless it is on the external boundary of the domain Ω. We look at the processors which own the neighbor elements and use the smaller as the face's owner. The other (or -1 for external boundary faces) is not surprisingly called the face's "other" processor.

The convention for element definitions is that the first processor in the `overlap` command's list is the owner. Any additional processors in the list simply get a copy of the element in their overlap region.

The convention for face definitions is that the first two processors in the `overlap` command's list are the owner and the other processor for the face, respectively. Any additional processors in the list simply get a copy of the face in their overlap region. If only one processor is listed, or if the first two processor numbers are identical, the face is an interior face for that processor's owned region. If the second processor number is -1, the face is an exterior boundary face.

Example: Non-overlapping subdomains

Both mechanisms are general enough to handle both overlapping and non-overlapping subdomains. With `distribute` and `pad`, all processors know the global layout. With

`overlap`, the user must ensure that each processor builds all the objects in its subdomain, and can determine the connections between subdomains. That is, the owner of each element, and the owner and other of each face, must be known to every processor which shares a given element of face.

Here is a very simple 2 triangle mesh in which each element is in exactly one subdomain. Interface faces are in exactly two; all other faces are in exactly one.

```
overlap { 0 }
  v a -1 0
endOverlap
overlap { 0 1 }
  v b 0 1
  v c 0 -1
endOverlap
overlap { 1 }
  v d 1 0
endOverlap
overlap { 0 -1 }
  e ab a b
  e ac a c
endOverlap
overlap { 0 1 }
  e bc b c
endOverlap
overlap { 1 -1 }
  e db d b
  e dc d c
endOverlap
overlap { 0 }
  tri abc ab bc ac
endOverlap
overlap { 1 }
  tri dbc db bc dc
endOverlap
```

The mesh is illustrated in Figure 5.1. The `distribute/pad` version is much shorter: remove all `overlap`/`endOverlap` commands, and at the end append `distribute block pad 0`. Note that for a large mesh, the overlap commands will take up a much smaller fraction of the input file than they appear to here.

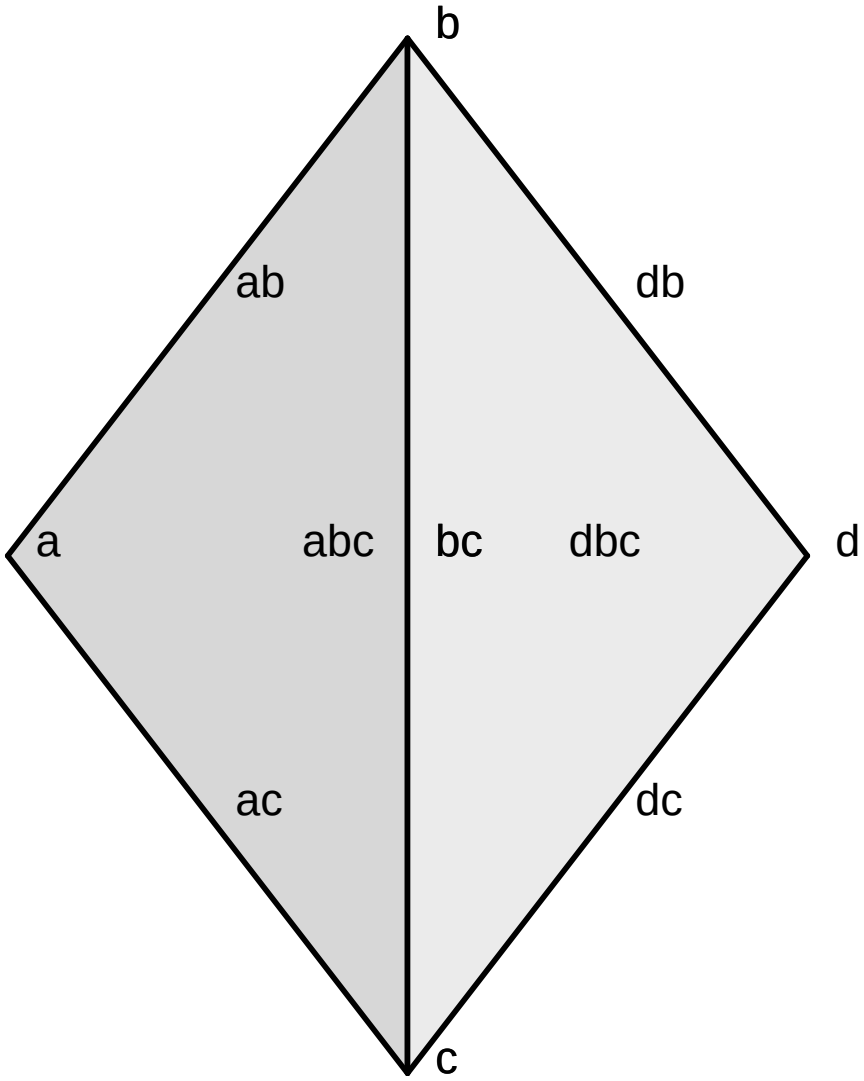

Figure 5.1: Non-overlapping mesh decomposition

5.2.2 Running Multiple Scenarios

RUF allows related multiple scenarios to be run from within the same input file. While unrelated scenarios can always be handled by a shell script which repeatedly runs the program, running related scenarios together offers some advantages, primarily related to doing convergence studies.

A typical convergence study looks like this: one defines a coarse mesh and chooses an analytic reference solution. One then subdivides the mesh, solves the PDE, computes norms of errors, and repeats several times. This is straightforward to do with RUF, as the `repeat`, `subdivide`, `solve` and `norms` commands work exactly as one would expect in implementing this construction.

In addition, however, RUF allows more complex collections of scenarios. This section explains what kinds of scenarios can be run together.

Related scenarios must use the same domain Ω. In particular, once `subdivide` has been called, no further changes in the domain are allowed: commands such as `v`, `e`, `tri` and `rect` cannot be used once the mesh has been subdivided. This ensures that all levels of the mesh correspond to the same domain, thereby allowing one to compare solutions obtained at different levels of refinement.

Coefficient and boundary data are propagated by the `subdivide` command and so should generally not be changed thereafter, except by using the `pin`, `unpin`, and `averageTensor` commands. Otherwise the changes may propagate further, or less far, than you expected. Of course, user defined reference solutions and coefficient functions can always be used to provide complete control.

The `subdivide` command applies to the finest mesh yet constructed and refines it, making the result the new current mesh. The `solve` command applies to the current mesh. The `up` and `down` commands change the current mesh, allowing one to solve on a coarser level after having created a finer one. This is useful in studying effective parameters, such as the averaged tensor coefficient, in which fine scale data is averaged up to coarse scale values.

The `approx` reference solution choice uses the finest solution available as the reference solution. We say "available", because one can create intermediate mesh levels with `subdivide` without necessarily calling `solve` on each one. The `up` and `down` commands are used to select the coarse grid.

The `inherit` preconditioning method uses the solution from the mesh one level up, if available, as the initial guess for the solver on the current mesh, thereby speeding up convergence in the iterative solver.

The `plot`, `norms`, and `write` commands all apply to the solution corresponding to the current mesh; it is an error to call these functions if there is no solution attached to the current mesh, except when plotting the mesh itself.

The selected postprocessor only impacts `plot`, `norms`, and `write` commands

involving the vector flux variable. Thus it can be repeatedly adjusted to observe the impact of different postprocessing methods on the same solution.

5.2.3 Generating Stochastic Tensor Fields

The `stochastic` command family enables one to define very general stochastic tensor fields. The basic model is as follows. Each element stores a tensor coefficient K and a stochastic value s. Initially, groups of elements share common values of the tensor as set by the `tensor` and `setTensor` commands, with the identity tensor as the default. The command

```
stochastic prepare
```

ensures that each element gets its own independent copies of K and s to work with. The command

```
stochastic scalar normal(0,1)
```

assigns a standard normal variate to each s value — a different one for each element. The command

```
stochastic scalar exp((val>0)*val)
```

sets all negative s values to zero, and then exponentiates the result. One can also refer to the element's center coordinates in the formula as x, y, and z. Finally, the command

```
stochastic tensor val 0.1*val 2*val
```

in the two space dimensional setting creates a tensor field; on each element, the tensor will look like

$$T = \begin{pmatrix} v & 0.1v \\ 0.1v & 2v \end{pmatrix},$$

where v is the scalar stochastic value constructed by the previous commands. Again, one can use x, y, and z in the formulas to define non-stationary fields.

If you now subdivide the mesh, the K and s values are inherited, providing a certain amount of local correlation in space. Alternatively, one can repeatedly subdivide, prepare, adjust the scalar, and subdivide again, until the scalar field has the required spatial statistics; then construct the corresponding tensor field. The tensor field formulas can contain calls to the `normal` function, or any other *kScript* mathematical function, as well.

The `averageTensor` command produces effective tensor coefficients on a coarse mesh from data on a finer scale. It is experimental and is intended to facilitate ongoing research of the author's. Results for various averaging methods will be described in a future research report.

5.2.4 User Defined Coefficients and Reference Solutions

The `user` reference solution choice provides access to custom reference solutions, boundary data and coefficient values. Library link overloading means that a programmer need only recompile the main executable driver and one new C++ source file defining the user functions. The presence of this file will cause the linker to ignore the default user functions in the library, when the program is linked.

This one new file is quite shielded from the messy details of the code internals. In it, the programmer must define 10 functions, based on the model in `userSoln.C`. In the default case, they all do nothing, or return zero.

Applications of this technique include non-smooth reference solutions arising from jumps in the coefficient tensor: in simple geometries one can work out the solution and its derivatives and plug it in, allowing convergence studies in this interesting non-smooth case.

5.3 Sample Input Files

Suppose the file `twistM` contains the following lines:

```
# a pair of triangles stretched along the normal direction
 v a     -1 0
 v b     0 1
 v c     0 -1
 v d     2 1.5

# boundary edges
 e ab     a b
 e ac     a c

 bndy flux 0 # the default is scalar 0
 e db     d b
 e dc     d c

# internal edges
 e bc     b c

 tri t     ab ac bc
 tri tt     db dc bc
```

It defines a domain made from two triangles, as shown in Figure 5.2. Two boundary edges use Type I boundary conditions, and two others use Type II ones.

Next, suppose the file `demo` contains the following lines:

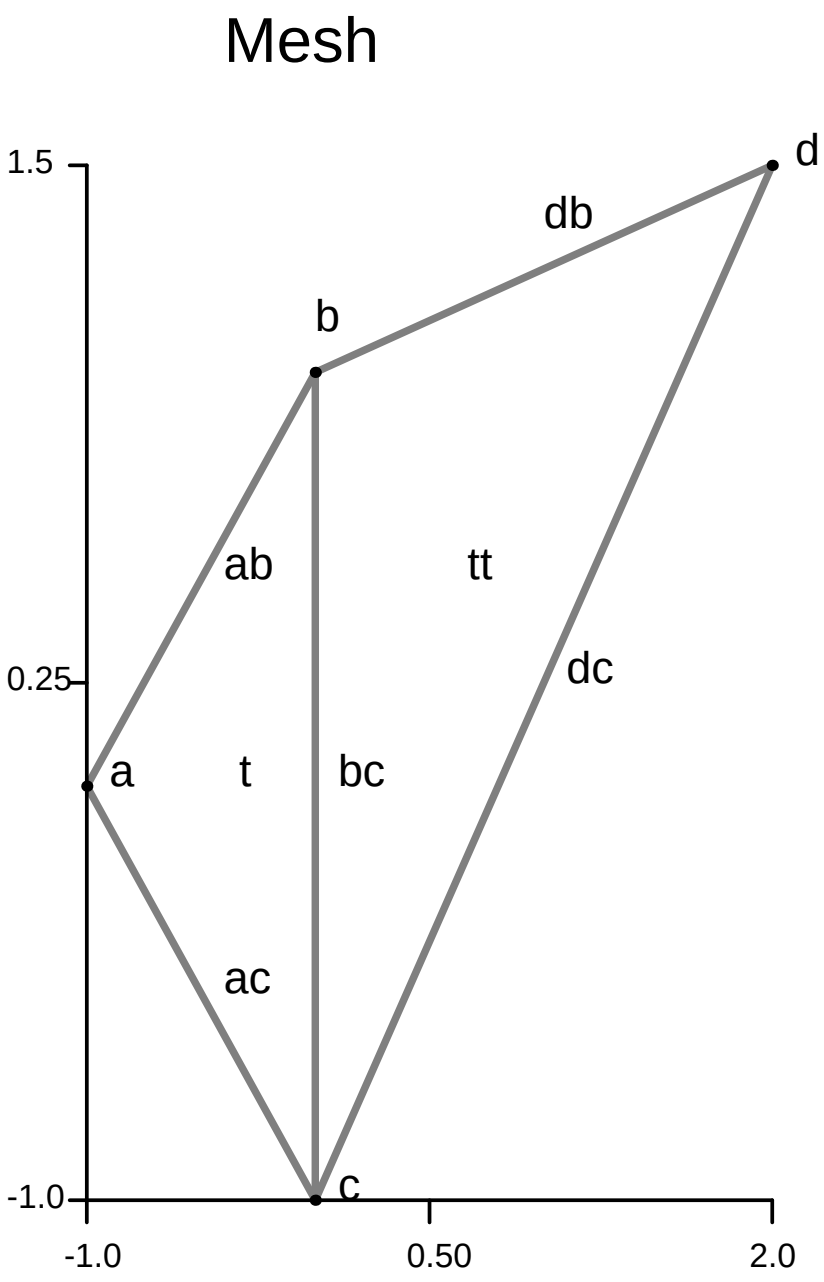

Figure 5.2: Sample coarse mesh

```
# a sample driver file

tensor { 1 0.5 3 }

include twistM # this reads in the above mesh description

plot mesh
plotCommands { new }

refSoln poly { 3 3 } {
        1  -3  1.7 -4.1
        2  2.4 3.1   0
      -1.1 2.1  0    0
       1.2  0   0    0  }

plot edges
plot bndy
plotCommands { new }

subdivide 3 times
solve

plot scalar
plot edges
plotCommands { new }

plot ref scalar
plot edges
plotCommands { new }

plot abs err scalar
plot edges
plotCommands { new }

norms abs err scalar
norms abs err flux
```

The script subdivides the mesh and solves the PDE using a cubic polynomial as a reference solution, with a non-diagonal tensor for K. We set the tensor before reading the mesh to avoid having to use setTensor commands afterward on each

element to override the identity tensor default. This is because settings for alpha, tensor, and so on apply only to *subsequently* defined mesh objects. The script produces several informative plots as well as discrete norms for the error in both pressure and velocity. If run via a command like

```
elliptic_enhanced_2d demo demo.plot demo.log
```

the plots will be in the file `demo.plot`, while the norms and other convergence information will be in `demo.log`. For instance, Figure 5.3 is a grayscale rendition of the color plot produced for the pressure solution.

The output log shows, for instance, that the error in the pressure on this mesh, which is still a relatively coarse mesh, is already fairly small: a maximum error of 0.456 (as shown by the l^∞ norm), while pressure itself is on the order of 20. Subdividing the mesh further yields more accurate answers, at the cost of additional computing time. For complete details on the convergence rates for each numerical method in RUF, see [2, 3].

Switching topics for a moment, Figure 5.4 shows a realization of a non-stationary permeability field constructed with RUF for use in geostatistical simulations. This particular field is spatially uncorrelated but with spatially dependent variance increasing to the right.

Finally, as a further example of the code's flexibility in representing general geometry, Figure 5.5 shows the pressure and velocity fields surrounding a two dimensional horizontal well. The injection well is perforated only along the horizontal segment; the other well is a production well.

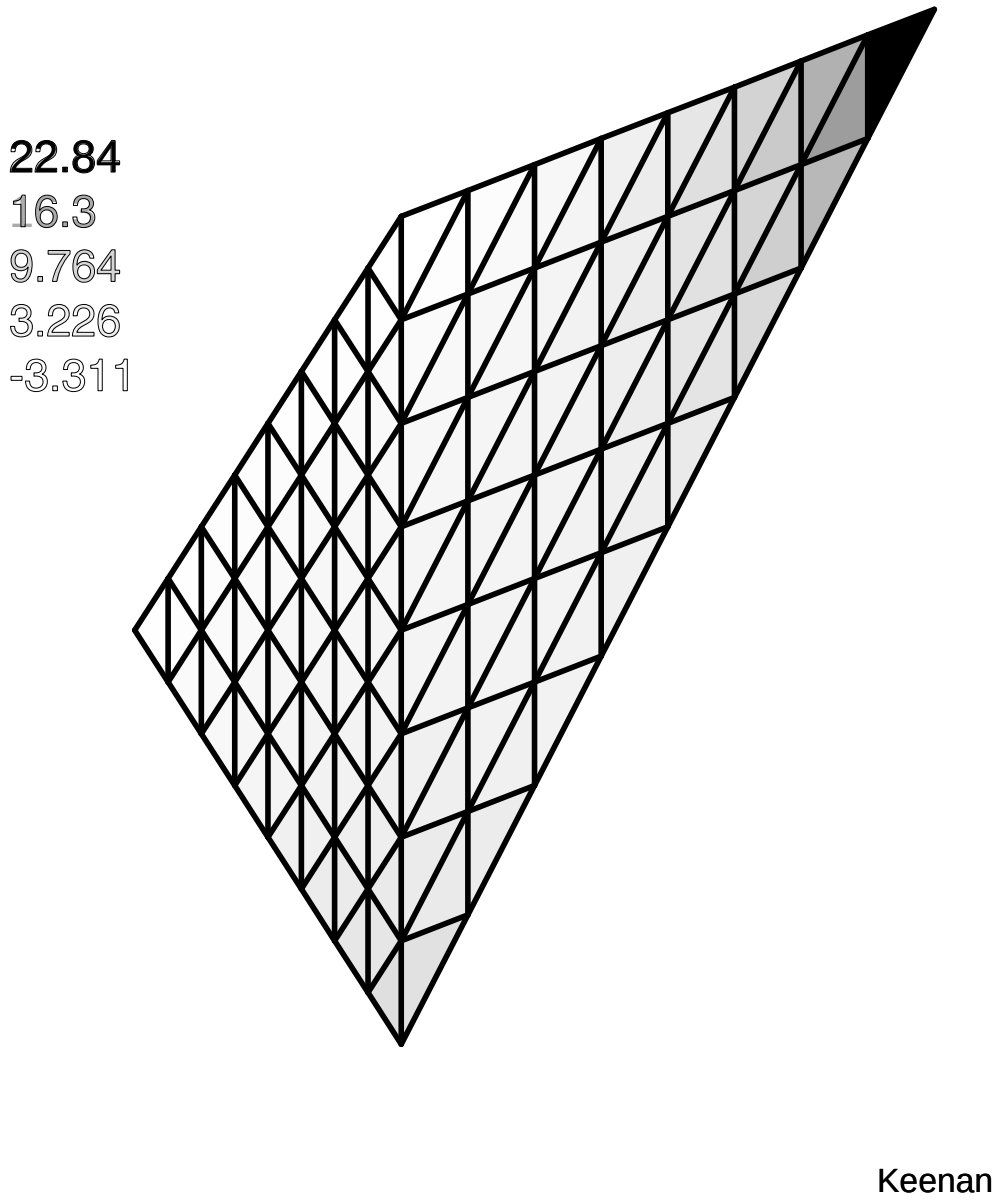

Figure 5.3: Pressure Solution

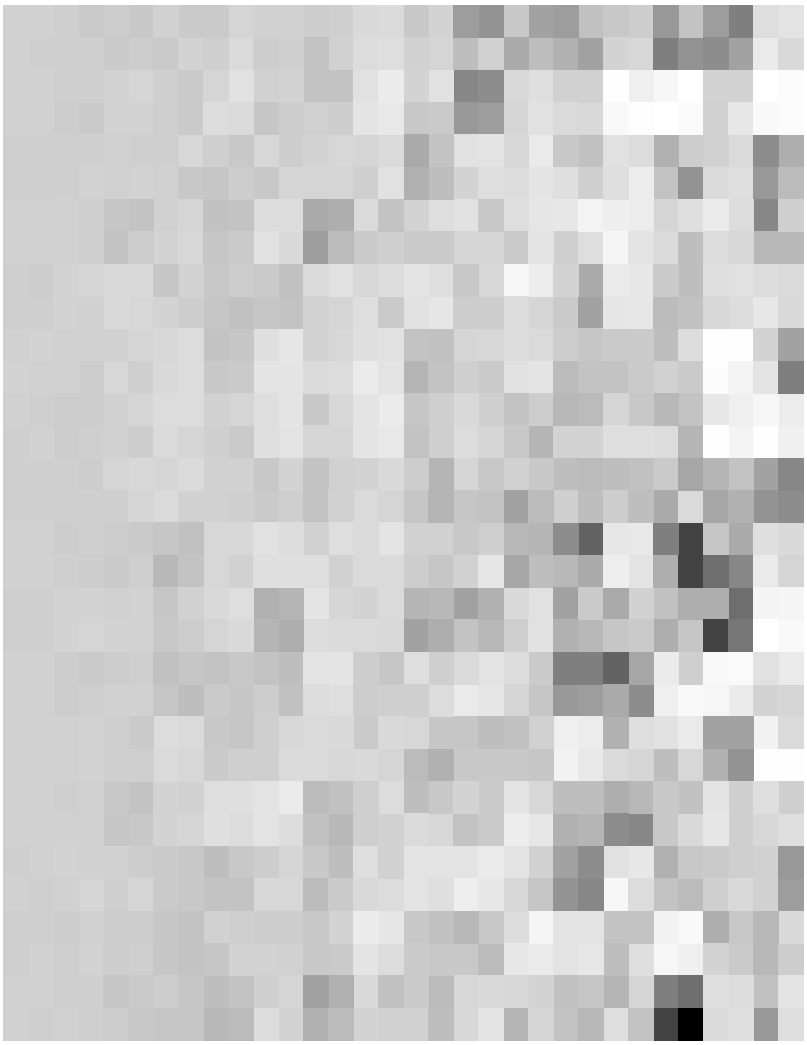

Figure 5.4: A non-stationary permeability field

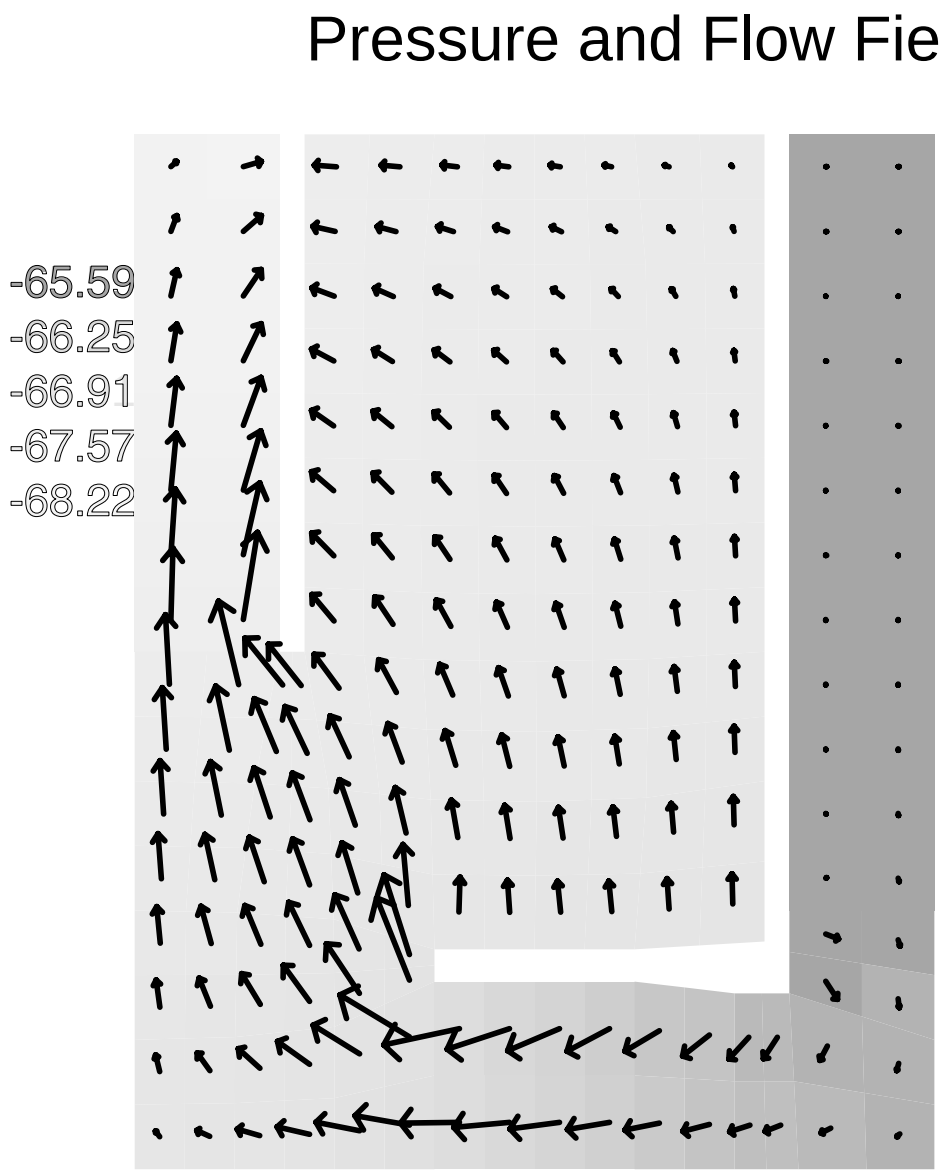

Figure 5.5: A Partially Perforated Horizontal Well Example

Bibliography

[1] Arbogast, T., Dawson, C., and Keenan, P. T., *Efficient Mixed Methods for Groundwater Flow on Triangular or Tetrahedral Meshes*, Computational Methods in Water Resources X, (Peters et. al., editors), Kluwer (1994), pp. 3–10.

[2] Arbogast, T., Dawson, C., and Keenan, P. T., *Mixed Finite Element as Finite Differences for Elliptic Equations on Triangular Elements*, Dept. of Computational and Applied Mathematics Tech. Report #93–53, Rice University, 1993.

[3] Arbogast, T., Dawson, C., Keenan, P. T., Wheeler, M. F., and Yotov, I., *Implementation of Mixed Finite Element Methods for Elliptic Equations on General Geometry*, Dept. of Computational and Applied Mathematics Tech. Report #95–??, Rice University, 1995, and To Appear.

[4] Dupont, T. F. and Keenan, P. T., *Superconvergence and Postprocessing of Fluxes from Lowest Order Mixed Methods on Triangles*, Dept. of Computational and Applied Mathematics Tech. Report #95–03, Rice University, 1995, and To Appear.

[5] Keenan, P. T., *C++ and FORTRAN Timing Comparisons*, Dept. of Computational and Applied Mathematics Tech. Report #93–03, Rice University, 1993.

[6] Keenan, P. T., *An Efficient Postprocessor for Velocities from Mixed Methods on Triangular Elements*, Dept. of Computational and Applied Mathematics Tech. Report #94–22, Rice University, 1994.

[7] Keenan, P. T., *RUF 1.0 User Manual: The Rice Unstructured Flow Code*, Dept. of Computational and Applied Mathematics Tech. Report #94–30, Rice University, 1994.

[8] Keenan, P. T., *kScript User Manual*, Dept. of Computational and Applied Mathematics Tech. Report #95–02, Rice University, 1995.