PREQS User Guide

Carol A. San Soucie

September 9, 1996

1 Introduction

This manual describes the usage, installation and basic structure of PREQS, a parallel Richards' equation solver. Richards' equation [11] models the physical situation of water flowing into a porous medium containing both air and water [4]. The equation is given as,

$$\frac{\partial \theta(h)}{\partial t} - \nabla \cdot (K(h) \nabla h) = f, \text{ in } \Omega, \tag{1}$$

where $K(h) = \frac{k(x)\rho g}{\mu} k_r(h)$. Here, h is hydraulic head, θ is water content, $k(x)$ is the absolute permeability tensor, $k_r(h)$ is the relative permeability, μ is water viscosity, ρ is water density, g is gravity, f is a water source term and Ω is the flow domain. Boundary conditions of the form,

$$\sigma(x)(\mathbf{u} \cdot \mathbf{n}) + \nu(x) h = \gamma(x,t), \text{ on } \Gamma, \tag{2}$$

are considered, where Γ is the boundary of Ω, $\mathbf{u}$ is the flux $-K(h)\nabla h$, $\mathbf{n}$ is an outward pointing, unit, normal vector to Γ, σ and ν are functions of position only and γ is a function of both position and time.

PREQS uses a cell-centered finite difference scheme (or, equivalently, a lowest–order Raviart–Thomas expanded mixed finite element approximation [1]) for discretizing (1). Backward Euler is used for the time discretization, giving a fully implicit method. Application of these methods results in a system of nonlinear discrete equations which must be solved at each time step. This nonlinear system is solved with Newton's method and an optional backtracking globalization technique with dynamic forcing term selection. Each Newton iteration requires the solution of a large, nonsymmetric linear system to which a preconditioned GMRES method is applied. The system is preconditioned with a Jacobi preconditioner.

2 Usage

The PREQS user interface is based on Philip T. Keenan's *kScript* package, a flexible application scripting language. For a complete introduction to *kScript*, see the *kScript User Manual* [8], or the World Wide Web page at `http://www.ticam.utexas.edu/users/keenan/`. The *kScript* package can be obtained from the Web site, and used as the front end to other applications, but it is subject to the terms of the *kScript* copyright notice provided with the distribution and is not in the public domain. The PREQS user interface was created with `cmdGen` [7], a C++ code generation tool written by Keenan, and builds on the Keenan C++ Foundation Class Library, version 2.5.

PREQS reads commands from an input file written in the *kScript* language. Before describing the usage of the PREQS code, we present a brief summary of the core features of *kScript*.

2.1 *kScript*

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*. *kScript* is strongly typed and applications can add new data types as well.

For a complete and up-to-date list of commands, functions, types and objects available to the user interface, run the program to access on-line help. Once *kScript*, or programs based on it such as PREQS, is executed, type

```
help
```

to get started.

Commands specific to the `PREQS` program 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 square brackets are optional literal strings, typically 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 (`|`). For example, the syntax of the set command is

```
[set] nameExpr name [to|=] expression expr
```

Both the name `set` and the equal sign are optional, so the five commands

```
set x to 3.14
set x = 3.14
x = 3.14

set x 3.14
x 3.14
```

all assign the same value to a variable named x, but the first three versions are easier for a human reader to understand.

The keywords `optional` and `required` introduce alternative sets of arguments. Each set begins with a string literal which, if encountered while parsing the command, signals that the remainder of that clause will follow. Multiple cases can be separated by a vertical bar. In the required case, one alternative must be selected; in the optional case, zero or one may be chosen.

The `sequence` keyword introduces an argument pattern which may be repeated multiple times. A sequence argument can be an empty string (`{}`), a curly brace delimited list of one or more instances of the pattern, or, a single instance without the surrounding curly braces.

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 literals must be enclosed in curly braces, not quote marks. The curly braces can be nested and within them only the percent sign (`%`) is special — all other text is recorded verbatim. In all other contexts, white space (spaces, tabs, line breaks, and so on) serves only to delimit commands and their arguments.

2.2 Variables and Commands

In this section we describe each of the variables and commands which comprise the PREQS user interface. These variables and commands can be used throughout input files to set attributes of the computational grid, the physical problem and solvers for the nonlinear and linear discrete problems.

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 names are listed below; users can define additional ones using the `define` and `set` commands.

Some commands take an argument of type "ftype." This argument is a predefined function type (listed below) and is followed by two parameters, c1 and c2. The possible choices for ftype are,

1. `constant`

   For all values of its arguments, a constant function will return the value c1.

2. `vg_theta`

   Specifies the van Genuchten water content function [6],

   $$\theta(h) = \frac{\theta_s - \theta_r}{(1 + (\alpha h)^n)^m} + \theta_r, \tag{3}$$

   where $\alpha = c1$ and $m = c2$.

3. `vg_thetaDer`

   Specifies the derivative with respect to hydraulic head of the van Genuchten water content function.

4. `vg_relperm`

   Specifies the van Genuchten relative permeability function [6],

   $$k_r(h) = \frac{\left(1 - \frac{(\alpha h)^{n-1}}{(1+(\alpha h)^n)^m}\right)^2}{(1 + (\alpha h)^n)^{m/2}} \tag{4}$$

   where $\alpha = c1$ and $m = c2$.

5. `vg_relpermDer`

   Specifies the derivative with respect to hydraulic head of the van Genuchten relative permeability function.

6. `user`

   For user–defined functions, the user must specify a compiled routine which evaluates the function for various arguments. The parameters c1 and c2 allow some variability in the run–time specification of this function.

In the next four sections, we discuss the input commands and variables used for setting up a specific problem. Default values for variables are given in parentheses on the same line as the variable's name. Appendix A gives a sample input file along with a description of the problem and parameters it sets.

2.2.1 Grid Input

The first group of variables and commands are those associated with grid input. Note that the PREQS code runs three–dimensional problems. Lower dimensional problems should be modeled by taking one division in the unused directions.

`BoundingBox [(] <mathExpr xmin> <mathExpr ymin> <mathExpr zmin> [)][(] <mathExpr xmax> <mathExpr ymax> <mathExpr zmax> [)]`

Define the domain bounding box by specifying first the three coordinates of the left, front, bottom corner, (xmin, ymin, zmin), then the coordinates of the right, back, top corner (xmax, ymax, zmax).

The variables xmin, ymin, zmin, xmax, ymax and zmax can only be changed through the BoundingBox command. However, the user can see their values directly by echoing these variables. The default bounding box for the computation is the unit cube.

`xmin (0)`

constant double: The left coordinate of the bounding box.

`xmax (1)`

constant double: The right coordinate of the bounding box.

`ymin (0)`

constant double: The front coordinate of the bounding box.

`ymax (1)`

constant double: The back coordinate of the bounding box.

`zmin (0)`

constant double: The bottom coordinate of the bounding box.

`zmax (1)`

constant double: The top coordinate of the bounding box.

The following commands require a specified direction in which to act. The keyword argument specifies this direction and can be either x, y or z as described below.

`direction`

Keyword Type: Coordinate directions, x, y and z. Literal values are:

`x`

Left to right coordinate direction.

`y`

Front to back coordinate direction.

`z`

Bottom to top coordinate direction.

`nProcDivisions [in] <keyword(direction) dir> [is] <mathExpr num>`

Specify the number of processors in the coordinate directions.

`nCoarseDivisions [in] <keyword(direction) dir> [is] <mathExpr num>`

Specify the number of coarse divisions in the coordinate directions for the coarse grid. This command is valid only when the coarse grid is uniform. The coarse grid must be the same as or a refinement of the processor grid.

`CoarseDivide <keyword(direction) dir> [at] <doubleArray vals>`
: Define a dividing point for subdomains in coordinate directions. Each new dividing point must be greater than the previously set dividing point and less than the domain boundary for that direction. The coarse grid must be the same as or a refinement of the processor grid.

`nFineDivisions [in] <keyword(direction) dir> [is] <mathExpr num>`
: Specify the number of mesh cells in the coordinate directions for the fine grid. This command is valid only when the fine grid is uniform. The fine grid must be the same as or a refinement of the coarse grid.

`FineDivide <keyword(direction) dir> [at] <doubleArray vals>`
: Define a dividing point for mesh cells in coordinate directions. Each new dividing point must be greater than the previously set dividing point and less than the domain boundary for that direction. The fine grid must be the same as or a refinement of the coarse grid.

`isUniformCoarse (1)`
: int: 1 if coarse mesh is divided into uniform subdomains, and 0 otherwise.

`isUniformFine (1)`
: int: 1 if mesh is divided into uniform mesh cells, and 0 otherwise.

`Distribute`
: Distribute the coarse mesh among processors.

As of the time this is written, only the fine grid is used in the computation. However, further work will include two–level methods making use of both the coarse and fine grids.

2.2.2 Verification of Input

The next group are commands associated with printing out various characteristics of the problem and grid.

`PrintBoundingBox`
: Prints bounding box coordinates.

`PrintProcDivisions`
: Prints the number of processors in each coordinate direction.

`PrintCoarseDivisions`
: Prints the coarse grid divisions in each coordinate direction.

`PrintFineDivisions`
: Prints the fine grid divisions in each coordinate direction.

`VerifyGrid`
: Prints the bounding box, number of processors in each coordinate direction and coarse and fine grid divisions in each direction.

`showStatus`
: Prints input parameter information including parameters related to the physical properties of the problem being modeled.

`dumpGrid`
: Prints the computational grid.

2.2.3 Problem Specification

The physical problem that is modeled is specified with the following group of variables and commands.

`SetSigma sequence { <keyword(direction) dir> <int side> <nameExpr ftype> }`
Specifies the coefficient function for flux boundary conditions. Sigma is a function of (x, y, z, t).

`SetUpsilon sequence { <keyword(direction) dir> <int side> <nameExpr ftype> }`
Specifies the coefficient function for hydraulic head boundary conditions. Upsilon is a function of (x, y, z, t).

`SetGamma sequence { <keyword(direction) dir> <int side> <nameExpr ftype> }`
Specifies the right–hand–side function for boundary conditions. Gamma is a function of (x, y, z, t).

`SetF <nameExpr ftype>`
Specifies the forcing term function. F is a function of (x, y, z, t).

`SetPermTensor sequence { <int i> <int j> <nameExpr ftype> }`
Specifies entries of the absolute permeability tensor. Entries are functions of (x, y, z, t).

`SetInitH <nameExpr ftype>`
Specifies the initial hydraulic head function. InitH is a function of (x, y, z, h).

`SetSoln <nameExpr ftype>`
Specifies the exact problem solution if one exists. The solution is a function of (x, y, z, t).

`SetTheta <nameExpr ftype>`
Specifies the water content function. Water content is a function of (x, y, z, h).

`SetThetaDerivative <nameExpr ftype>`
Specifies the derivative of the water content function. The water content derivative is a function of (x, y, z, h).

`SetRelPerm <nameExpr ftype>`
Specifies the relative permeability function. The relative permeability is a function of (x, y, z, h).

`SetRelPermDerivative <nameExpr ftype>`
Specifies the derivative of the relative permebility function. i The relative permeability derivative is a function of (x, y, z, h).

`porosity (0.36)`
double: Porosity of the medium.

`density (1 g/`cm^3`)`
double: Density of water.

`viscosity (1.24 cP)`
double: Viscosity of water in g/(sm) = 10Ns/m^2 = 10cP.

`gravity`
double: Gravitational constant = $9.80665 * \mathrm{m}/(\mathrm{sec}^2)$.

`satr (0.27)`
double: Residual water saturation.

`sats (0.99)`
double: Maximal water saturation.

The file

```
user.C
```

gives a dummy definition of all the above functions. Even if the user chooses a non-user-defined ftype, such as constant or one of the van Genuchten types, in the input file, the dummy definition must be compiled. Functions which are user–defined can be defined in this file.

2.2.4 Solver Parameters

The last group of variables and commands sets various attributes of the nonlinear and linear discrete system solvers.

`verbosity (0)`
int: 0, 1, 2, ... produce increasingly detailed debugging information.

`nonlinTol (1e-4)`
double: Relative error tolerance to use as stopping criterion in nonlinear iterative solution processes.

`nonlinItMax (20)`
int: Maximum number of iterations to allow in nonlinear iterative solution processes.

`linTol (1e-8)`
double: Relative error tolerance to use as stopping criterion in linear iterative solution processes.

`linItMax (20)`
int: Maximum number of iterations to allow in linear iterative solution processes.

`doBackTrack (1)`
int: Set to 1 if backtracking globalization should be run for the nonlinear iteration.

`Restrt (15)`
int: Restart parameter for GMRES.

`FinalTime (1)`
double: Set the final simulation time in sec.

`TimeStep (0.001)`
double: Set the initial time step size.

`MinStep (0.0001)`
double: Set the minimum time step size.

`MaxStep (0.1)`
double: Set the maximum time step size.

`isUniformTimeStep (1)`
int: Set to 1 for uniform time steps, or 0 to select automatic time stepping.

`hasExactSoln (0)`
: int: Set to 1 if problem has an exact solution specified.

`Solve`
: Solve the partial differential equation.

3 Installation

In order to install PREQS, one should first obtain and install the Keenan C++ Foundation Class Library (KFCL). This library comes as part of the *kScript* package and can be obtained at the World Wide Web site, `http://www.ticam.utexas.edu/users/keenan/`.

In order to compile the KFCL, five environment variables must be set. These variables are,

- `KEENAN_src`

  The root directory of the KFCL source code.

- `KEENAN_inc`

  The directory where all include files for the KFCL will be stored. The Makefiles will automatically move these include files to the directory specified in this variable.

- `KEENAN_lib`

  A directory where the compiled libraries will be placed in subdirectories of the form "machine/method."

- `KEENAN_make`

  The location of the master Makefile, Makefile-C++.

- `KSCRIPT_INC_DIR`

  A directory for ".k" files which are included in *kScript* files.

The source for these libraries comes in a number of directories. Briefly, these are,

cont Array, vector and stack classes.

io Formatted I/O, similar to the standard C++ I/O streams library.

ks *kScript* routines, plus run time support classes, which handle command line argument parsing.

na Complex numbers, sorting routines, statistical routines and random number generators.

os Low level operating system interface and machine dependent function, memory management routines.

parallel Platform specific functions as well as source for routines which define a generic communications library for use on distributed memory parallel machines.

scripts Scripts for compiling the source.

Once the source code is in place for these libraries, they can be compiled by by setting the appropriate targets and platforms in the

`Targets`

file in the

```
KEENAN_src
```

directory. The "scripts" directory must be in the executable path.

Then, typing

```
Build method=''method type''
```

will compile the source code. The method refers to the compiler options such as checking, optimization or debugging. The values it can take are,

debug Sets compiler options to be compatible with the debugger for each platform.

check Turns on some error checking, such as run time checking of array indices, but uses -O2 as a compiler option.

opt Does not turn on extra array checking and uses -O2 as a compiler option.

The following directories will need to be created somewhere and pointed to with the above environment variables,

- inc
- lib
- lib/sparc
- lib/sparc/check
- lib/sparc/opt
- lib/sparc/debug
- lib/paragon/... etc., for other machines

After compiling, the ks, io, cont and os libraries must be in the lib directory under the subdirectory of the appropriate platform. This is where the PREQS makefile will look for them.

In order to compile the PREQS code, the above environment variables must be set. Then, in the PREQS source directory, typing

```
make PREQS
```

will build the PREQS executable.

Currently, the communications supported in the KFCL/comm directory are: PVM, PICL, NX (native Intel library) and a null library which conducts a no-opt for each communication call. The null library is particularly useful for debugging on a single processor. The PREQS makefile is currently set up to link in the null library. It can be changed to link in the PVM library by changing PVMDIR in the makefile to be the appropriate PVM directory and by putting in the appropriate library in the libs variable.

To run the code, the number of processors must be specified with the command line argument -nproc #, where # is the number of processors used in the computation. An input file must also be given. The sample input file given in this document is contained in the file input.k. Thus, to execute the code with this file, type

```
PREQS -nproc 1 input.k.
```

PREQS is primarily written in C++. However, the GMRES method and all routines it depends on are written in FORTRAN. In order to mix these languages, naming conventions with respect to underscores and capital letters must be specified. Currently, the code is set up for IBM RS6000 systems. In order to change to another system such as a SUN, the file namecat.h in KFCL/include.h must be modified. Also, declarations of the FORTRAN routine gmres and the C++ routine, matmult, called from gmres must be changed in the file step.h to appropriately account for underscores.

4 Numerical Methods

This section gives a brief description of some of the numerical methods used in the PREQS code.

4.1 Discretization

The discretiztion scheme employed is a cell–centered finite difference scheme equivalent to the expanded mixed method of Arbogast, Wheeler and Yotov [1], and analyzed for Richards' equation in [5].

We now describe the finite difference scheme employed in solving the system (1)-(2). We consider a rectangular two or three dimensional domain, Ω, with boundary $\partial\Omega$. Let $(.,.)$ denote the $L^2(\Omega)$ inner product, scalar and vector.

We will approximate the L^2 inner product with various quadrature rules, denoting these approximations by $(.,.)_{\mathrm{R}}$, where $\mathrm{R} = \mathrm{M}$, T and TM are application of the midpoint, trapezoidal and trapezoidal by midpoint rules, respectively.

Let $0 = t^0 < t^1 < \cdots < t^N = T$ be a given sequence of time steps, $\Delta t^n = t^n - t^{n-1}$, $\Delta t = \max_n \Delta t^n$, and for $\phi = \phi(t,.)$, let $\phi^n = \phi(t^n,.)$ with

$$d_t\phi^n = \frac{\phi^n - \phi^{n-1}}{\Delta t^n}.$$

Let $V = H(\Omega, div) = \{\mathbf{v} \in (L^2(\Omega))^d : \nabla\cdot\mathbf{v} \in L^2(\Omega)\}$ and $W = L^2(\Omega)$.

We will consider a quasi-uniform triangulation of Ω with mesh size h denoted by $\mathcal{T}$ and consisting of rectangles in two dimensions or rectangular solids in three dimensions. We consider the lowest order Raviart-Thomas-Nedelec space on bricks, [10, 9]. Thus, on an element $E \in \mathcal{T}$, we have

$$\begin{aligned} V_h(E) &= \{(\alpha_1 x_1 + \beta_1, \alpha_2 x_2 + \beta_2, \alpha_3 x_3 + \beta_3)^T : \alpha_i, \beta_i \in \mathbb{R}\}, \\ W_h(E) &= \{\alpha : \alpha \in \mathbb{R}\}. \end{aligned}$$

For an element on the boundary, $\partial E \cap \partial\Omega \neq$, we have the edge space,

$$\Lambda_h(\partial E) = \{\alpha : \alpha \in \mathbb{R}\}.$$

We use the standard nodal basis, where for V_h the nodes are at the midpoints of edges or faces of the elements, and for W_h the nodes are at the centers of the elements. The nodes for Λ_h are at midpoints of edges.

The expanded mixed finite element method simultaneously approximates, $H, \tilde{U} = -\nabla H$ and $U = K(H)\tilde{U}$. This method with quadrature is given as follows. Find $H^n \in W_h, \tilde{\mathbf{U}}^n \in V_h, \mathbf{U}^n \in V_h$

and $L^n \in \Lambda_h$ for each $n = 1, \ldots, N$ satisfying,

$$(d_t\theta(H)^n, w)_{\mathrm{M}} + (\nabla \cdot \mathbf{U}^n, w) = (f^n, w),\ \forall w \in W_h, \tag{5}$$

$$(\tilde{\mathbf{U}}^n, \mathbf{v})_{\mathrm{TM}} = (H^n, \nabla \cdot \mathbf{v}) - (L^n, \mathbf{v} \cdot \mathbf{n})_\Gamma,\ \forall \mathbf{v} \in V_h, \tag{6}$$

$$(\mathbf{U}^n, \mathbf{v})_{\mathrm{TM}} = (K(H^n)\tilde{\mathbf{U}}^n, \mathbf{v})_{\mathrm{T}},\ \forall \mathbf{v} \in V_h, \tag{7}$$

$$(\sigma \mathbf{U}^n \cdot \mathbf{n}, \beta)_\Gamma = (\gamma + \nu L^n, \beta)_\Gamma,\ \forall \beta \in \Lambda_h. \tag{8}$$

The system (5)-(8) reduces to a finite difference scheme for the hydraulic head approximations at each of the cell centers. To see this, consider first equation (5) and let $w = w_{ijk} \in W_h$ be the basis function,

$$w_{ijk} = \begin{cases} 1, & \text{in cell } ijk, \\ 0, & \text{otherwise.} \end{cases}$$

Then,

$$\begin{aligned} \Delta x_i \Delta y_j \Delta z_k \phi_{ijk}(\theta(H)^n_{ijk} - \theta(H)^{n-1}_{ijk}) =& \Delta t^n \Delta y_j \Delta z_k (\frac{U^n_{i+1/2jk} - U^n_{i-1/2jk}}{\Delta x_{i+1/2}}) \\ &+ \Delta t^n \Delta x_i \Delta z_k (\frac{U^n_{ij+1/2k} - U^n_{ij-1/2k}}{\Delta y_{j+1/2}}) \\ &+ \Delta t^n \Delta x_i \Delta y_j (\frac{U^n_{ijk+1/2} - U^n_{ijk-1/2}}{\Delta z_{k+1/2}}) \\ &+ \Delta t^n \Delta x_i \Delta y_j \Delta z_k f^n. \end{aligned} \tag{9}$$

Equation (6) gives $\tilde{\mathbf{U}}^n$ in terms of H^n; in particular, choosing $\mathbf{v} = (v_{i+1/2jk}, 0, 0)$, where $v_{i+1/2jk}$ is the basis function associated with node $(x_{i+1/2}, y_j, z_k)$,

$$v_{i+1/2jk} = \begin{cases} \frac{1}{\Delta x_i}(x - x_{i-1/2}) & x \in [x_{i-1/2}, x_{i+1/2}], y \in [y_{j-1/2}, y_{j+1/2}], z \in [z_{k-1/2}, z_{k+1/2}], \\ \frac{1}{\Delta x_{i+1}}(x - x_{i+3/2}) & x \in [x_{i+1/2}, x_{i+3/2}], y \in [y_{j-1/2}, y_{j+1/2}], z \in [z_{k-1/2}, z_{k+1/2}], \\ 0 & \text{otherwise,} \end{cases}$$

equation (6) reduces to (dropping temporal superscripts),

$$\tilde{\mathbf{U}}^x_{i+1/2jk} = \frac{H_{ijk} - H_{i+1jk}}{\Delta x_{i+1/2}}. \tag{10}$$

If $x_{i+1/2}$ is on the boundary, then the difference in hydraulic head values in equation (10) is replaced by the difference between the head in the nearest cell and the multiplier α on the boundary closest to the cell. The divisor for this difference will be half the cell width instead of $\Delta x_{i+1/2}$. The α term only plays a role on the outer boundary of the domain.

Equation (7) gives $\mathbf{U}$ in terms of $\tilde{\mathbf{U}}$. Letting $\mathbf{v}$ be chosen as in (7) gives,

$$
\begin{aligned}
\mathbf{U}^x_{i+1/2jk}\Delta x_{i+1/2} = \frac{1}{8}\left(\frac{\rho k_r}{\mu}\right)_{i+1/2jk} &[(k_{11,i+1/2j-1/2k-1/2} + k_{11,i+1/2j+1/2k-1/2} \\
&+ k_{11,i+1/2j-1/2k+1/2} + k_{11,i+1/2j+1/2k+1/2}) \times (\Delta x_i \tilde{\mathbf{U}}^x_{i+1/2jk} + \Delta x_{i+1}\tilde{\mathbf{U}}^x_{i+1/2jk}) \\
&+ (k_{12,i+1/2j-1/2k-1/2} + k_{12,i+1/2j-1/2k+1/2}) \times (\Delta x_i \tilde{\mathbf{U}}^y_{ij-1/2k} + \Delta x_{i+1}\tilde{\mathbf{U}}^y_{i+1j-1/2k}) \\
&+ (k_{12,i+1/2j+1/2k-1/2} + k_{12,i+1/2j+1/2k+1/2}) \times (\Delta x_i \tilde{\mathbf{U}}^y_{ij+1/2k} + \Delta x_{i+1}\tilde{\mathbf{U}}^y_{i+1j+1/2k}) \\
&+ (k_{13,i+1/2j-1/2k-1/2} + k_{13,i+1/2j+1/2k-1/2}) \times (\Delta x_i \tilde{\mathbf{U}}^z_{ijk-1/2} + \Delta x_{i+1}\tilde{\mathbf{U}}^z_{i+1jk-1/2}) \\
&+ (k_{13,i+1/2j-1/2k+1/2} + k_{13,i+1/2j+1/2k+1/2}) \times (\Delta x_i \tilde{\mathbf{U}}^z_{ijk+1/2} + \Delta x_{i+1}\tilde{\mathbf{U}}^z_{i+1jk+1/2})] \\
&\equiv \left(\frac{\rho k_r}{\mu}\right)_{i+1/2jk} \bar{U}^x_{i+1/2jk}.
\end{aligned}
\tag{11}
$$

The coefficient $\left(\frac{\rho k_r}{\mu}\right)_{i+1/2jk}$ is approximated by upstream weighting as determined by the sign of $\bar{U}^x_{i+1/2jk}$, i.e.,

$$
\begin{aligned}
\left(\frac{\rho k_r}{\mu}\right)_{i+1/2jk} &= \left(\frac{\rho k_r}{\mu}\right)_{i+1jk}, &\quad \text{if } \bar{U}^x_{i+1/2jk} \le 0, \\
\left(\frac{\rho k_r}{\mu}\right)_{i+1/2jk} &= \left(\frac{\rho k_r}{\mu}\right)_{ijk}, &\quad \text{otherwise.}
\end{aligned}
$$

Lastly, equation (8) defines the multipliers α on the domain boundary. Let $\beta = \beta_{1/2jk}$ in equation (8). Then we have,

$$
-\sigma \mathbf{U}^x_{1/2jk} = \gamma_{1/2jk} + \nu\alpha_{1/2jk}. \tag{12}
$$

Combining equations (9)-(12) gives a finite difference method with a 19 point stencil for hydraulic head values.

In order to implement this finite difference scheme in parallel, each processor communicates with up to 18 neighbors. To reduce this requirement, we add extra unknowns along the interfaces between subdomains. Adding these unknowns allows for the normal fluxes at the interface points to be discontinuous, which is a nonphysical condition. Thus, extra equations which enforce continuity of normal fluxes at the interfaces are also introduced. Adding these extra unknowns corresponds to adding a single hydraulic head at the interface points. This value will be "owned" by one processor and communicated to the "non-owner" after updates. Let $\mathbf{V}_i = \mathbf{V}_h|_{\Omega_i}$. Then take, $\hat{\mathbf{V}}_h = \bigoplus \mathbf{V}_i$. The numerical scheme is defined as finding $(H^n, \tilde{\mathbf{U}}^n, \mathbf{U}^n, L^n) \in (W_h, \hat{\mathbf{V}}_h, \hat{\mathbf{V}}_h, \Lambda_h)$ at each time step $n = 1, \ldots, N$ satisfying,

$$
\begin{aligned}
(d_t\theta(H^n), w) + (\nabla \cdot \mathbf{U}^n, w) &= (f^n, w), &(13)\\
(\tilde{\mathbf{U}}^n, \mathbf{v})_{\Omega_i,TM} &= (H^n, \nabla \cdot \cdot \mathbf{v})_{\Omega_i} - (L^n, \mathbf{v}\cdot\mathbf{n})_{\Gamma_i}, &(14)\\
(\mathbf{U}^n, \mathbf{v})_{\Omega_i,TM} &= (K(H^n)\tilde{\mathbf{U}}^n, \mathbf{v})_{\Omega_i,T}, &(15)\\
(\sigma \mathbf{U}^n \cdot \mathbf{n}, \beta)_{\Gamma_i\cap\Gamma} &= (\gamma + \upsilon L^n, \beta)_{M,\Gamma_i\cap\Gamma}, &(16)\\
\sum_i (\mathbf{U}^n \cdot \mathbf{n}, \beta)_{\Gamma_{I_i}} &= 0. &(17)
\end{aligned}
$$

The extra unknowns provide a boundary condition for internal interfaces. Thus, the subdomains are coupled only through shared boundaries, and each subdomain will only need to communicate with neighbors sharing interfaces. Hence, subdomains communicate with up to 6 neighbors and not 18.

4.2 Nonlinear Equation Solver

The above finite difference scheme results in a coupled system of nonlinear discrete equations.

These nonlinear equations are solved with a backtracking line search globalized inexact Newton's method [3, 2]. Algorithm 4.1 describes an inexact Newton method applied to the problem of finding a root to the nonlinear equation $F(u) = 0$ where J is the Jacobian of F.

Algorithm 4.1 1. Let $u^{(0)}$ be an initial guess.

2. For $k = 0, 1, 2, \ldots$ until convergence, do

   (a) Choose $\eta^{(k)} \in [0, 1)$.

   (b) Using some Krylov iterative method, compute a vector $s^{(k)}$ satisfying

$$J^{(k)} s^{(k)} = -F^{(k)} + r^{(k)}, \tag{18}$$

   with $\frac{\|r^{(k)}\|}{\|F(u^{(k)})\|} \leq \eta^{(k)}$.

   (c) Set $u^{(k+1)} = u^{(k)} + \lambda^{(k)} s^{(k)}$.

The parameter η is chosen with a dynamic selection criteria, which reflects the agreement between F and its linear model at the previous iteration,

$$\eta^{(k)} = \min\{\eta_{\max}, \max\{\widetilde{\eta}^{(k)}, \left(\eta^{(k-1)}\right)^2\}\}, \tag{19}$$

where

$$\widetilde{\eta}^{(k)} = \frac{\left|\left\|F^{(k)}\right\| - \left\|F^{(k-1)} + J^{(k-1)} s^{(k-1)}\right\|\right|}{\left\|F^{(k-1)}\right\|}. \tag{20}$$

The parameter λ is chosen with a backtracking line–search globalization technique.

5 The PREQS Source Code

This section gives a brief overview of the code structure then discusses the contents of the PREQS source code files.

5.1 Code Overview

In this section, brief comments are made on the grid set–up, then a discussion of the equation solution driver function follows.

User input commands are used to set up the processor, coarse and fine grids, then distribute the coarse and fine grids over the processors. Source code which supports these commands is located in the files **Actions.C** and **GridInputUtil.C**. The grid is not distributed over the parallel machine until the *DistributeCmd* function is executed.

The user command *Solve* designates that the user–specified equation should be solved. The driver for the equation solve is the *SolveCmd* function in **Actions.C**. This function constructs a *ProblemState* class which holds all information including the computational grid for the given problem. The solve function then initializes the problem and starts a time loop. each iteration of the time loop calls the *step* function.

The *step* function determines the time step size based on whether the user specified automatic or uniform time stepping. It then calls *calcProps* which is a function that calculates physical properties such as relative permeabilities, water contents and derivatives that are needed for the nonlinear residual and linear system construction. After these properties are calculated, the *step* function initializes the linear system matrix data structures and calls various *setUp* functions which construct the linear system and nonlinear residual.

As mentioned in Section 4, the cell-centered finite difference scheme used in PREQS leads to a 19 point stencil. To reduce communication requirements, PREQS has extra hydraulic head unknowns along interfaces between subdomains. Each of these unknowns is "owned" by a processor (the lower number of two two processors that share the interface owns the unknowns along the interface). The owner processor calculates properties associated with this unknown and updates the "sharer" processor.

Physical data and unknowns are stored in the *dataArray* classes. Among its members, this class has a doubleVector, *lgvec*, a 3-D doubleArray, *intr*, and an array of 2-D doubleArrays. The *lgvec* vector holds all data for the interior cells and boundary unknowns. The *intr* array holds all data for the interior subdomain cells. This array is aliased to the beginning of *lgvec*. The 2-D arrays hold the boundary and interface data for the processors six boundaries. The owned unknowns on the boundaries are aliased to consecutive entries in *lgvec*. Thus, *lgvec* can be passed to the linear solver which expects a long vector of system unknowns.

The linear system matrix can be stored in blocks as can be seen in the **matrices.C, matrices.h** and **ProblemState.h** files. The AA block holds the 19 point stencil entries for the interior unknown dependencies on other interior unknowns. The AB block is actually an array of six 2-D arrays which store dependencies of interior unknowns on subdomain boundary unknowns. The BA block is also an array of six 2-D arrays storing the dependencies of the subdomain boundary unknowns on the interior unknowns. Lastly, the BB block is an array of six *bndryMat* classes. This is a class defined in **matrices.h** which stores boundary dependencies on other boundary unknowns. For a given subdomain face, boundary on boundary dependencies can be expressed as an unknown depending on itself (stored in the *bndryMat.diag* array) and as the boundary unknown depending on unknowns on different faces ("around" an edge dependencies). These last dependencies are stored in the *bndryMat.cross* array.

After the *setUp* functions conclude, *step* calls the *newtonCheck* function which checks for convergence of the Newton method. If convergence is not attained, a while loop is started where each iteration solves the linear system, backtracks if the user specified backtracking, constructs an updated system and nonlinear residual and checks for convergence.

The linear system is solved with the GMRES method. This solver is coded in Fortran, but the matrix–vector multiply and preconditioner (Jacobi for now) are coded in C++ to take advantage of the *dataArray* class structure. The matrix–vector multiply is done in a function called *matmult* contained in the **matmult.C** file. This function performs the multiply in block fashion and within each block, applies a band multiply.

After *step* converges the Newton iteration, it calculates the L^2 error in the problem solution if the user specified that an exact solution exists. Control is then returned to the *SolveCmd* function which updates the time and loops until the final time is reached.

5.2 File Contents

The following is a brief summary of the source code files and what is in each one.

- GridInputUtil.C Utility functions used for grid and problem input.
  Functions included: ProcessDivisions, isRefinement, PartitionGrid, inputFn, PrintHFunc

- InitMass.C Contains InitMass function which calculates the initial water mass in the system - used in mass balance calculations. Some code is duplicated from the setUp routines.
  Function included: InitMass

- InitProblem.C Contains InitProblem function which initializes the pressure head, absolute permeability, sigma and upsilon variables.
  Function included: InitProblem

- actions.C Contains definitions of input commands.
  Functions included: BoundingBoxCmd, nProcDivisionsCmd, nCoarseDivisionsCmd, CoarseDivideCmd, nFineDivisionsCmd, FineDivideCmd, PrintBoundingBoxCmd, PrintProcDivisionsCmd, PrintCoarseDivisionsCmd, PrintFineDivisionsCmd, VerifyGridCmd, DistributeCmd, SetSigmaCmd, SetUpsilonCmd, SetGammaCmd, SetFCmd, SetPermTensorCmd, SetInitHCmd, SetSolnCmd, SetThetaCmd, SetThetaDerivativeCmd, SetRelPermCmd, SetRelPermDerivativeCmd, SolveCmd, ShowStatusCmd, dumpGridCmd

- backtrack.C Performs the backtracking line-search globalization while updating reduction factor used in linear system tolerance selection.
  Function included: backtrack

- cState.C Declares the global ProblemState pointer cState.

- calcProps.C Calculates physical properties used in matrix set up - water content, relative permeability and their derivatives.
  Function included: calcProps

- computeError.C Computes the discrete L^2 error for the case that an exact solution is known.
  Function included: computeError

- dataArray.C Implements dataArray, bndryArray and tensor classes.

- direction.C Defines the numerical value for indexing x, y and z as coordinate directions.
  Function included: initDirections

- dumpGrid.C Prints global grid into file *grid.eye* in EYE format.
  Function included: dumpGrid

- feval.C Evaluates nonlinear function at a given pressure head.
  Function included: feval

- fn.C Implements data function classes, constant, van Genuchten and user defined.

- gFns.C Global norm and dot product functions for dataArrays.
  Functions included: gnorm and gdot.

- globalVars.C Declares pointers to various global functions.
  Functions included: globalVars::globalVars

- grid.C Implementation of constructor and print function for grid class.
  Functions included: grid::grid(), grid::PrintGrid()
- jacobi.C Implementation of Jacobi preconditioner (diagonal scaling PC).
  Functions included: jacobi
- matmult.C Performs matrix-vector multiply and update, $y = b * y + a * A * x$.
  Functions included: matmult, matmultAA, matmultAB, matmultBA, matmultBB.
- matrices.C Implements 3-D and 2-D banded matrix classes as well as a specialized boundary matrix class.
  Functions included: constructor, resize, operator= and print for bandMat3d, bandMat2d and bndryMat classes.
- newtonCheck.C Checks for convergence of the Newton method.
  Functions included: newtonCheck
- noprec.C Performs no preconditioning action - just copies src to dest.
  Functions included: noprec
- printMatrix.C Prints the system matrix.
  Functions included: printMatrix
- problemState.C Constructor for the problemState class.
  Functions included: problemState::problemState.
- richardsMain.C This is the driver for the PREQS code.
  Functions included: userMain
- richardsPost.C Posts commands to the *kScript* tables.
  Functions included: richardsCmdsPost
- richardsPostObjs.C Posts objects to *kScript* tables.
  Functions included: richardsPostObjects
- setUpBndMat.C Functions in this file calculate matrix and right-hand side entries corresponding to flux values on subdomain boundary edges.
  Functions included: setUpBndMatx, setUpBndMaty, setUpBndMatz.
- setUpForceTerm.C Calculates contribution to right-hand-side of forcing term. Updates contribution to mass from forcing term.
  Function included: setUpForceTerm
- setUpIntMat.C Calculates matrix entries from fluxes across edges interior to the subdomain.
  Functions included: setUpIntMat
- setUpTimeMat.C Calculates contribution to right-hand-side and diagonal matrix entries due to the time derivative term. Also updates contribution to mass from time change in mass.
  Functions included: setUpTimeMat
- step.C Solves the nonlinear system for a given time.
  Functions included: step

- transferData.C Transfers data in arrays pointed to in stuff from owner to sharer.
  Functions included: transferData

- user.C Default definitions of input functions.
  Functions included: userSigx0, userSigx1, userSigy0, userSigy1, userSigz0, userSigz1, userUpsx0, userUpsx1, userUpsy0, userUpsy1, userUpsz0, userUpsz1, userGamx0, userGamx1, userGamy0, userGamy1, userGamz0, userGamz1, userF, userInitH, userSoln, userTheta, userRelPerm, userThetaDer, userRelPermDer, userK11, userK12, userK13, userK22, userK23, userK33

- whereAmI.C This file contains a routine which finds the indices for the calling processor's coordinates in the processor grid as well as its neighbors.
  Functions included: whereAmI

- gmres.f Contains routines associated with the GMRES iterative solver.
  Functions included: gmres, update, basis

- gdblas.f Global parallel functions for use in gmres.
  Functions included: Gdnorm2, Gddot

- dblas.f Double precision blas routines.

Header files contained in the PREQS code are: GridInputUtil.h, InitProblem.h, cState.h, dataArray.h, direction.h, fn.h, gFns.h, globalVars.h, grid.h, matrices.h, msgtypes.h, printMatrix.h, problemState.h, richards.h, step.h, user.h, verbosity.h and whereAmI.h.

Appendix A: Test Program Input file

This section describes a sample input file used in the PREQS program. This sample file is given below.

```
include /u10/carol/richards/units.k

BoundingBox ( 0, 0, 0 ) ( 100*cm, 100*cm, 20*cm ) ;
nProcDivisions in x is 2
nProcDivisions in y is 2
nProcDivisions in z is 1
isUniformCoarse = 1
nCoarseDivisions in x is 2
nCoarseDivisions in y is 2
nCoarseDivisions in z is 2
isUniformFine  = 1
nFineDivisions in x is 20
nFineDivisions in y is 20
nFineDivisions in z is 10
Distribute
VerifyGrid

nonlinTol = 1e-8
nonlinItMax = 50
linTol = 1e-7
Restrt = 50
linItMax = 300
doBackTrack = 0
MinStep = 0.0001*day
MaxStep = 6*hr
TimeStep = 0.0005*day
FinalTime = 0.5*day
isUniformTimeStep = 0
verbosity = 4
hasExactSoln = 0

double hres = -734*cm

SetSigma { x 0 constant 1
           x 1 constant 1
           y 0 constant 1
           y 1 constant 1
           z 0 user 0 0
           z 1 user 0 0 }
SetUpsilon { x 0 constant 0
             x 1 constant 0
             y 0 constant 0
             y 1 constant 0
```

```
              z 0 user 0 0
              z 1 user 0 0 }
SetGamma { x 0 constant 0
           x 1 constant 0
           y 0 constant 0
           y 1 constant 0
           z 0 user -1000*cm 0
           z 1 user -70*cm 0 }

SetF constant 0
SetPermTensor { 1 1 user 9.33e-12*m*m 9.33e-10*m*m
                2 2 user 9.33e-12*m*m 9.33e-10*m*m
                3 3 user 9.33e-12*m*m 9.33e-10*m*m
                1 2 user 9.33e-13*m*m 9.33e-11*m*m
                1 3 user 9.33e-13*m*m 9.33e-11*m*m
                2 3 user 9.33e-13*m*m 9.33e-11*m*m }
SetInitH user hres 0

SetTheta vg_theta 0.0334 0.5
SetRelPerm vg_relperm 0.0334 0.5
SetThetaDerivative vg_thetaDer 0.0334 0.5
SetRelPermDerivative vg_relpermDer 0.0334 0.5

porosity = 0.368
density = 1*gm/cm^3
viscosity = 1.124*cP
gravity = 9.80665*m/sec^2
comp = 4.48-10*m^2/nt
satr = 0.2771
sats = 0.98

Solve
```

The first line specifies that the program should include the file **units.k**. This file uses the cgs units system as base and defines a number of units in terms of these. These unit definitions allow PREQS input to be in any unit system, not even a consistent system. The BoundingBox command specifies that the flow domain will be the rectangular solid where $x \in [0, 100]$, $y \in [0, 100]$ and $z \in [0, 20]$. The processor mesh is $2 \times 2 \times 1$. The coarse grid is uniform with 2 divisions in each coordinate direction. The fine grid is also uniform, but has 20 divisions in each of the x and y directions and 10 divisions in the z direction. The user has then asked to distribute the grid over the processors and verify the resulting grid.

The Newton method will iterate until a discrete l^2 norm of the residual is less than $1e-8$ or until 50 iterations have executed, whichever comes first. The GMRES method will iterate until a relative residual reduction of $1e-7$ is achieved or until 300 iterations have been executed. The restart parameter is 50. The backtracking line–search globalization will not be applied. The minimum time step is 0.0001 day, the maximum step is 6 hours and the initial time step is 0.0005 day. The final time is 0.5 day. Automatic time step selection will be used so that the time step will grow after the initial step. The verbosity level is 4 and an exact solution is not specified.

A temporary variable *hres* is specified to be double with the value of -734 cm. No flow boundary conditions are set for all x and y external domain boundaries. User–specified Dirichlet conditions are set for the domain top and bottom boundaries. The source/sink term is 0. The absolute permeability tensor entries are all user–defined functions with two parameters. The file **user.C** defines all user–defined functions for this case. Lastly, the initial hydraulic head function is also specified to be a user–defined function.

The water content and relative permeability functions along with their derivatives are specified to be of van Genuchten form with $\alpha = 0.0334$ and $m = 0.5$. Physical constant are set as given.

The last of the file tells PREQS to solve the specified equation with the above defined parameters.

Acknowledgements

I wish to thank Philip T. Keenan for his advice and help in the design of the matrix classes and in the design of the matrix-vector multiply routines.

References

[1] T. Arbogast, M. F. Wheeler, and I. Yotov, *Mixed finite elements for elliptic problems with tensor coefficients as cell-centered finite differences*, Dept. Comp. Appl. Math. TR95-06, Rice University, Houston, TX 77251, Mar. 1995. To appear SIAM J. Numer. Anal., 1997, vol. 34.

[2] S. C. Eisenstat and H. F. Walker, *Globally convergent inexact Newton methods*, SIAM J. Optimization, 4 (1994), pp. 393–422.

[3] ——, *Choosing the forcing terms in an inexact Newton method*, SIAM J. Sci. Comp., 17 (1996), pp. 16–32.

[4] R. A. Freeze and J. A. Cherry, *Groundwater*, Prentice Hall, Inc., New Jersey, 1979.

[5] C. A. San Soucie, *Mixed finite element methods for variably saturated subsurface flow*, Dept. Comp. Appl. Math. TR96-10, Rice University, Houston, TX 77251, Apr. 1996.

[6] M. T. van Genuchten, *A closed form equation for predicting the hydraulic conductivity of unsaturated soils*, Soil Sci. Soc. Am. J., 44 (1980), pp. 892–898.

[7] P. T. Keenan, *cmdGen 2.5 user manual*, Texas Inst. for Comp. and Applied Math. 96-09, University of Texas, Austin, TX, Feb. 1996.

[8] ——, *kScript 2.5 user manual*, Texas Inst. for Comp. and Applied Math. 96-08, University of Texas, Austin, TX, Feb. 1996.

[9] J. Nedelec, *Mixed finite elements in* $\mathbb{R}^3$, Numer. Math., 35 (1980), pp. 315–341.

[10] P. A. Raviart and J. M. Thomas, *A mixed finite element method for second order elliptic problems*, in Mathematical Aspects of Finite Element Methods: Lecture Notes in Mathematics 606, I. Galligani and E. Magenes, eds., Berlin, 1977, Springer-Verlag, pp. 292–315.

[11] L. A. Richards, *Capillary conduction of liquids through porous mediums*, Physics, 1 (1931), pp. 318–333.