ParFlow Documentation

Release 3.11.0

the ParFlow Developers

Sep 06, 2022

CONTENTS:

Integrated GroundWater Modeling Center

Reed M. Maxwell[1], **Stefan J. Kollet**[2], **Laura E. Condon**[3], **Steven G. Smith**[4], **Carol S. Woodward**[5], **Robert D. Falgout**[6], **Ian M. Ferguson**[7], **Nicholas Engdahl**[8], **Jaro Hokkanen**[9], **Basile Hector**[10], **James Gilbert**[11], **Lindsay Bearup**[7], **Jennifer Jefferson**[12], **Chuck Baldwin**, **William J. Bosl**[13], **Richard Hornung**[4], **Steven Ashby**[14], **Ketan B. Kulkarni**[15]

updated: Sep 06, 2022

[1] *Department of Civil and Environmental Engineering, The High Meadows Environmental Institute, and the Integrated GroundWater Modeling Center, Princeton University, Princeton, NJ, USA.* reedmaxwell@princeton.edu

[2] *Institute for Bio- and Geosciences, Agrosphere (IBG-3) and Centre for High-Performance Scientific Computing in Terrestrial Systems, Research Centre Jülich, Jülich, Germany.* s.kollet@fz-juelich.de

[3] *Department of Hydrology and Atmospheric Sciences, University of Arizona, Tucson, USA.* lecondon@arizona.edu

[4] *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA. USA.*

[5] *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA.* cswoodward@llnl.gov

[6] *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA.*

[7] *US Bureau of Reclamation, Denver, CO, USA.*

[8] *Civil and Environmental Engineering, Washington State University, Pullman, WA, USA.* nick.engdahl@wsu.edu

[9] *CSC - IT Center for Science, Espoo, Finland*

[10] *Université Grenoble-Alpes/IRD/IGE, Grenoble, FR.*

[11] *University of California, Santa Cruz & NOAA Fisheries Southwest Fisheries Science Center, Santa Cruz, CA, USA*

[12] *HNTB, Milwaukee, Wisconsin, United States.*

[13] *Children's Hospital Informatics Program, Harvard Medical School, Boston, MA, USA.*

[14] *Pacific Northwest National Laboratory, Richland, WA, USA.*

[15] *Jülich Supercomputing Centre and Centre for High-Performance Scientific Computing in Terrestrial Systems, Research Centre Jülich, Jülich, Germany.*

CHAPTER

ONE

INTRODUCTION

ParFlow (*PARallel FLOW*) is an integrated hydrology model that simulates surface and subsurface flow. ParFlow [AF96, JW01, KM06, Max13] is a parallel simulation platform that operates in three modes:

1. steady-state saturated;
2. variably saturated;
3. and integrated-watershed flow.

ParFlow is especially suitable for large scale problems on a range of single and multi-processor computing platforms. ParFlow simulates saturated and variably saturated subsurface flow in heterogeneous porous media in three spatial dimensions using a mulitgrid-preconditioned conjugate gradient solver [AF96] and a Newton-Krylov nonlinear solver [JW01]. ParFlow solves surface-subsurface flow to enable the simulation of hillslope runoff and channel routing in a truly integrated fashion [KM06]. ParFlow is also fully-coupled with the land surface model `CLM` [DZD+03] as described in Kollet and Maxwell [KM08a], Maxwell and Miller [MM05]. The development and application of ParFlow has been on-going for more than 20 years [AFBBP+02, AF96, AM11, CM13, CMG13, DMC10, dBRM09, FM10, FFKM09, JW01, KMW+13, KRM11, KCSchauttemeyer+09, KM06, KM08a, KM08b, KMW+10, Max10, Max13, MCT00, MCT08, MCK07, MK08a, MK08b, MLM+11, MM05, MTK09, MWH07, MWT03, MM11, MMGW14, MMR+14, MMF+13, RMC10, SSM+14, SM12a, SM12b, SNSMM12, SMP+10, TBP99, TCRM99, TFS+98, TMC+05, WM11, WMM13, WGM02] and resulted in some of the most advanced numerical solvers and multigrid preconditioners for massively parallel computer environments that are available today. Many of the numerical tools developed within the ParFlow platform have been turned into or are from libraries that are now distributed and maintained at LLNL (*Hypre* and *SUNDIALS*, for example). An additional advantage of ParFlow is the use of a sophisticated octree-space partitioning algorithm to depict complex structures in three-space, such as topography, different hydrologic facies, and watershed boundaries. All these components implemented into ParFlow enable large scale, high resolution watershed simulations.

ParFlow is primarily written in *C*, uses a modular architecture and contains a flexible communications layer to encapsulate parallel process interaction on a range of platforms. `CLM` is fully-integrated into ParFlow as a module and has been parallelized (including I/O) and is written in *FORTRAN 90/95*. ParFlow is organized into a main executable `pfdir/pfsimulator/parflow_exe` and a library `pfdir/pfsimulator/parflow_lib` (where `pfdir` is the main directory location) and is comprised of more than 190 separate source files. ParFlow is structured to allow it to be called from within another application (e.g. WRF, the Weather Research and Forecasting atmospheric model) or as a stand-alone application. There is also a directory structure for the message-passing layer `pfdir/pfsimulator/amps` for the associated tools `pfdir/pftools` for `CLM` `pfdir/pfsimulator/clm` and a directory of test cases `pfdir/test`.

1.1 How to use this manual

This manual describes how to use ParFlow, and is intended for hydrologists, geoscientists, environmental scientists and engineers. This manual is written assuming the reader has a basic understanding of Linux / UNIX environments, how to compose and execute scripts in various programming languages (e.g. TCL or Python), and is familiar with groundwater and surface water hydrology, parallel computing, and numerical modeling in general. In *Getting Started*, we describe how to install ParFlow, including building the code and associated libraries. Then, we lead the user through a simple ParFlow run and discuss the automated test suite. In *The ParFlow System*, we describe the ParFlow system in more detail. This chapter contains a lot of useful information regarding how a run is constructed and most importantly contains two detailed, annotated scripts that run two classical ParFlow problems, a fully saturated, heterogeneous aquifer and a variably saturated, transient, coupled watershed. Both test cases are published in the literature and are a terrific initial starting point for a new ParFlow user.

Manipulating Data: PFTools describes data analysis and processing. *Model Equations* provides the basic equations solved by ParFlow. *ParFlow Files* describes the formats of the various files used by ParFlow. These chapters are really intended to be used as reference material. This manual provides some overview of ParFlow some information on building the code, examples of scripts that solve certain classes of problems and a compendium of keys that are set for code options.

CHAPTER TWO

GETTING STARTED

This chapter is an introduction to setting up and running ParFlow. In *ParFlow Solvers* we describe the solver options available for use with ParFlow applications.

The ParFlow source code can be found on the ParFlow GitHub. Instructions for building ParFlow and Frequently-Asked Questions can be found on the ParFlow Wiki.

2.1 ParFlow Solvers

ParFlow can operate using a number of different solvers. Two of these solvers, IMPES (running in single-phase, fully-saturated mode, not multiphase) and RICHARDS (running in variably-saturated mode, not multiphase, with the options of land surface processes and coupled overland flow) are detailed below. This is a brief summary of solver settings used to simulate under three sets of conditions, fully-saturated, variably-saturated and variably-saturated with overland flow. A complete, detailed explanation of the solver parameters for ParFlow may be found later in this manual. To simulate fully saturated, steady-state conditions set the solver to IMPES, an example is given below. This is also the default solver in ParFlow, so if no solver is specified the code solves using IMPES.

```
pfset Solver               Impes
```

To simulate variably-saturated, transient conditions, using Richards' equation, variably/fully saturated, transient with compressible storage set the solver to RICHARDS. An example is below. This is also the solver used to simulate surface flow or coupled surface-subsurface flow.

```
pfset Solver             Richards
```

To simulate overland flow, using the kinematic wave approximation to the shallow-wave equations, set the solver to RICHARDS and set the upper patch boundary condition for the domain geometry to OverlandFlow, an example is below. This simulates overland flow, independently or coupled to Richards' Equation as detailed in [KM06]. The overland flow boundary condition can simulate both uniform and spatially-distributed sources, reading a distribution of fluxes from a binary file in the latter case.

```
pfset Patch.z-upper.BCPressure.Type  OverlandFlow
```

For this case, the solver needs to be set to RICHARDS:

```
pfset Solver         Richards
```

ParFlow may also be coupled with the land surface model `CLM` [[]]. This version of `CLM` has been extensively modified to be called from within ParFlow as a subroutine, to support parallel infrastructure including I/O and most importantly with modified physics to support coupled operation to best utilize the integrated hydrology in ParFlow[[MM05], [KM08a]]. To couple `CLM` into ParFlow first the `-with-clm` option is needed in the `./configure` command as indicated in

Installing ParFlow. Second, the `CLM` module needs to be called from within ParFlow, this is done using the following solver key:

```
pfset Solver.LSM CLM
```

Note that this key is used to call `CLM` from within the nonlinear solver time loop and requires that the solver bet set to RICHARDS to work. Note also that this key defaults to *not* call `CLM` so if this line is omitted `CLM` will not be called from within ParFlow even if compiled and linked. Currently, `CLM` gets some of it's information from ParFlow such as grid, topology and discretization, but also has some of it's own input files for land cover, land cover types and atmospheric forcing.

CHAPTER

THREE

THE PARFLOW SYSTEM

The ParFlow system is still evolving, but here we discuss how to define the problem in *Defining the Problem*, how to run ParFlow in *Running ParFlow*, and restart a simulation in *Restarting a Run*. We also cover options for visualizing the results in *Visualizing Output* and summarize the contents of a directory of test problems provided with ParFlow in *Directory of Test Cases*. Finally in *Annotated Input Scripts* we walk through two ParFlow input scripts in detail.

The reader is also referred to *Manipulating Data: PFTools* for a detailed listing of the of functions for manipulating ParFlow data.

3.1 Defining the Problem

There are many ways to define a problem in ParFlow, here we summarize the general approach for defining a domain (*Basic Domain Definition*) and simulating a real watershed (*Setting Up a Real Domain*).

The "main" ParFlow input file is one of the following: a `.tcl` TCL script, a `.py` Python script, or a `.ipynb` JuPyter notebook. This input script or notebook is used some special routines in PFTools to create a database which is used as the input for ParFlow. This database has the extension `.pfidb` and is the database of keys that ParFlow needs to define a run. See *Main Input Files (.tcl, .py, .ipynb)* and *ParFlow Input Keys* for details on the format of these files. The input values into ParFlow are defined by a key/value pair and are listed in *ParFlow Input Keys*. For each key you provide the associated value using either the `pfset` command in TCL or associated it with a named run (we use <*runname*> in this manual to denote that) inside the input script.

Since the input file is a script or notebook you can use any feature of TCL or Python to define the problem and to postprocess your run. This manual will make no effort to teach TCL or Python so refer to one of the available manuals or the wealth of online content for more information. This is NOT required, you can get along fine without understanding TCL or Python.

Looking at the example programs in the *Directory of Test Cases* and going through the annotated input scripts included in this manual (*Annotated Input Scripts*) is one of the best ways to understand what a ParFlow input file looks like.

3.1.1 Basic Domain Definition

ParFlow can handle complex geometries and defining the problem may involve several steps. Users can specify simple box domains directly in the input script. If a more complicated domain is required, the user may convert geometries into the `.pfsol` file format (*ParFlow Solid Files (.pfsol)*) using the appropriate PFTools conversion utility (*Manipulating Data: PFTools*). Alternatively, the topography can be specified using `.pfb` files of the slopes in the x and y directions.

Regardless of the approach the user must set the computational grid within the `.pfb` script as follows:

```
#-----------------------------------------------------------------------------
# Computational Grid
```

(continues on next page)

(continued from previous page)

```
#-----------------------------------------------------------------------------
pfset ComputationalGrid.Lower.X                -10.0
pfset ComputationalGrid.Lower.Y                 10.0
pfset ComputationalGrid.Lower.Z                  1.0

pfset ComputationalGrid.DX                       8.89
pfset ComputationalGrid.DY                      10.67
pfset ComputationalGrid.DZ                       1.0

pfset ComputationalGrid.NX                      18
pfset ComputationalGrid.NY                      15
pfset ComputationalGrid.NZ                       8
```

The value is normally a single string, double, or integer. In some cases, in particular for a list of names, you need to supply a space seperated sequence. This can be done using either a double quote or braces.

```
pfset Geom.domain.Patches "left right front back bottom top"

pfset Geom.domain.Patches {left right front back bottom top}
```

For commands longer than a single line, the TCL continuation character can be used,

```
pfset Geom.domain.Patches "very_long_name_1 very_long_name_2 very_long_name_3 \
                           very_long_name_4 very_long_name_5 very_long_name_6"
```

3.1.2 Setting Up a Real Domain

This section provides a brief outline of a sample workflow for setup ParFlow `CLM` simulation of a real domain. Of course there are many ways to accomplish this and users are encouraged to develop a workflow that works for them.

This example assumes that you are running with ParFlow `CLM` and it uses slope files and an indicator file to define the topography and geologic units of the domain. An alternate approach would be to define geometries by building a `.pfsol` file (*ParFlow Solid Files (.pfsol)*) using the appropriate PFTools conversion utility (*Manipulating Data: PFTools*).

The general approach is as follows:

1. Gather input datasets to define the domain. First decide the resolution that you would like to simulate at. Then gather the following datasets at the appropriate resolution for your domain:

   a. Elevation (DEM)

   b. Soil data for the near surface layers

   c. Geologic maps for the deeper subsurface

   d. Land Cover

2. Create consistent gridded layers that are all clipped to your domain and have the same number of grid cells

3. Convert gridded files to `.pfb` (*ParFlow Binary Files (.pfb)*). One way to accomplish this is by reformatting the gridded outputs to the correct ParFlow `.sa` order (*ParFlow Simple ASCII and Simple Binary Files (.sa and .sb)*) and to convert the `.sa` file to `.pfb` using the conversion tools (see *Common examples using ParFlow TCL commands (PFTCL)* Example 1). If you have an elevation file in `.pfb` format, you may wish to preserve it as provenance for the slopes and for use in post-processing tools. You may point ParFlow to the elevation:

```
pfset TopoSlopes.Elevation.FileName "elevation.pfb"
```

4. Calculate slopes in the x and y directions from the elevation dataset. This can be done with the built in tools as shown in *Common examples using ParFlow TCL commands (PFTCL)* Example 5. In most cases some additional processing of the DEM will be required to ensure that the drainage patterns are correct. To check this you can run a "parking lot test" by setting the permeability of surface to almost zero and adding a flux to the top surface. If the results from this test don't look right (i.e. your runoff patterns don't match what you expect) you will need to go back and modify your DEM.

5. Create an indicator file for the subsurface. The indicator file is a 3D `.pfb` file with the same dimensions as your domain that has an integer for every cell designating which unit it belongs to. The units you define will correspond to the soil types and geologic units from your input datasets.

6. Determine the hydrologic properties for each of the subsurface units defined in the indicator file. You will need: Permeability, specific storage, porosity and van Genuchten parameters.

7. At this point you are ready to run a ParFlow model without `CLM` and if you don't need to include the land surface model in your simulations you can ignore the following steps. Either way, at this point it is advisable to run a "spinup" simulation to initialize the water table. There are several ways to approach this. One way is to start with the water table at a constant depth and run for a long time with a constant recharge forcing until the water table reaches a steady state. There are some additional key for spinup runs that are provided in *Spinup Options*.

8. Convert land cover classifications to the IGBP[1] land cover classes that are used in CLM.

- 1. Evergreen Needleleaf Forest
- 2. Evergreen Broadleaf Forest
- 3. Deciduous Needleleaf Forest
- 4. Deciduous Broadleaf Forest
- 5. Mixed Forests
- 6. Closed Shrublands
- 7. Open Shrublands
- 8. Woody Savannas
- 9. Savannas
- 10. Grasslands
- 11. Permanent Wetlands
- 12. Croplands
- 13. Urban and Built-Up
- 14. Cropland/Natural Vegetation Mosaic
- 15. Snow and Ice
- 16. Barren or Sparsely Vegetated
- 17. Water
- 18. Wooded Tundra

9. Create a `CLM` vegm file that designates the land cover fractions for every cell (Refer to the `clm input` directory in the Washita Example for an sample of what a `vegm` file should look like).

[1] http://www.igbp.net

10. Create a `CLM` driver file to set the parameters for the `CLM` model (Refer to the `clm input` directory in the Washita Example for a sample of a `CLM` driver file).

11. Assemble meteorological forcing data for your domain. CLM uses Greenwich Mean Time (GMT), not local time. The year, date and hour (in GMT) that the forcing begins should match the values in `drv_clmin.dat`. `CLM` requires the following variables (also described in *Main Input Files (.tcl, .py, .ipynb)*):

- DSWR: Visible or short-wave radiation $[W/m^2]$.
- DLWR: Long wave radiation $[W/m^2]$
- APCP: Precipitation $[mm/s]$
- Temp: Air Temperature $[K]$
- UGRD: East-west wind speed $[m/s]$
- VGRD: South-to-North wind speed $[m/s]$
- Press: Atmospheric pressure $[pa]$
- SPFH: Specific humidity $[kg/kg]$

If you choose to do spatially heterogenous forcings you will need to generate separate files for each variable. The files should be formatted in the standard ParFlow format with the third (i.e. z dimension) as time. If you are doing hourly simulations it is standard practice to put 24 hours in one file, but you can decide how many time steps per file. For an example of heterogenous forcing files refer to the `NLDAS` directory in the Washita Example).

Alternatively, if you would like to force the model with spatially homogenous forcings, then a single file can be provided where each variable is a column and rows designate time steps.

12. Run your simulation!

3.2 Running ParFlow

Once the problem input is defined, you need to add a few things to the script to make it execute ParFlow. First you need to add the TCL or Python commands to load the ParFlow command package. We will cover TCL first, then Python below.

TCL

To set up and run ParFlow using PFTools in TCL, you need the following header lines.

```
#
# Import the ParFlow TCL package
#
lappend auto_path $env(PARFLOW_DIR)/bin
package require parflow
namespace import Parflow::*
```

This loads the `pfset` and other ParFlow commands into the TCL shell.

Since this is a script you need to actually run ParFlow. These are normally the last lines of the input script.

```
#-----------------------------------------------------------------------------
# Run and Unload the ParFlow output files
#-----------------------------------------------------------------------------
pfrun default_single
pfundist default_single
```

The `pfrun` command runs ParFlow with the database as it exists at that point in the file. The argument is the name to give to the output files (which will normally be the same as the name of the script). Advanced users can set up multiple problems within the input script by using different output names.

The `pfundist` command takes the output files from the ParFlow run and undistributes them. ParFlow uses a virtual file system which allows files to be distributed across the processors. The `pfundist` takes these files and collapses them into a single file. On some machines if you don't do the `pfundist` you will see many files after the run. Each of these contains the output from a single node; before attempting using them you should undistribute them.

Since the input file is a TCL script run it using the TCL shell or command intepreter:

```
tclsh runname.tcl
```

NOTE: Make sure you are using TCL 8.0 or later. The script will not work with earlier releases.

Python

To run ParFlow via Python in either a Notebook or script you need to install PFTools. This makes the Python commands available within your environment. To do this you can either build ParFlow to include the building of PFTools in Python, or you can install the package from PyPi. This might look like:

```
pip install pftools
```

At a minimum you need to import the ParFlow Python package and name your run. There are a lot more tools that bring substantial functionality that are discussed in other sections of this manual.

```python
from parflow import Run
from parflow.tools.fs import mkdir, get_absolute_path

dsingle = Run("dsingle", __file__)
#-----------------------------------------------------------------------------
dsingle.FileVersion = 4
```

Then to build the key database and execute ParFlow you use the run command built into the Python PFTools structure.

```
dsingle.run()
```

From the command line you would execute your Python script using the command interpreter.

```
python default_single.py
```

A lot more detail, including several tutorials and examples, are given in the *Python* section of this manual.

One output file of particular interest is the `<run name>.out.log` file. This file contains information about the run such as number of processes used, convergence history of algorithms, timings and MFLOP rates. For Richards' equation problems (including overland flow) the `<run name>.out.kinsol.log` file contains the nonlinear convergence information for each timestep. Additionally, the `<run name>.out.txt` contains all information routed to `standard out` of the machine you are running on and often contains error messages and other control information.

3.3 Restarting a Run

A ParFlow run may need to be restarted because either a system time limit has been reached, ParFlow has been prematurely terminated or the user specifically sets up a problem to run in segments. In order to restart a run the user needs to know the conditions under which ParFlow stopped. If ParFlow was prematurely terminated then the user must examine the output files from the last "timed dump" to see if they are complete. If not then those data files should be discarded and the output files from the next to last "timed dump" will be used in the restarting procedure. As an important note, if any set of "timed dump" files are deleted remember to also delete corresponding lines in the well output file or recombining the well output files from the individual segments afterwards will be difficult. It is not necessary to delete lines from the log file as you will only be noting information from it. To summarize, make sure all the important output data files are complete, accurate and consistent with each other.

Given a set of complete, consistent output files - to restart a run follow this procedure :

1. Note the important information for restarting :
   - Write down the dump sequence number for the last collection of "timed dump" data.
   - Examine the log file to find out what real time that "timed dump" data was written out at and write it down.
2. Prepare input data files from output data files :
   - Take the last pressure output file before the restart with the sequence number from above and format them for regular input using the keys detailed in 6.1.27 *Initial Conditions: Pressure* and possibly the `pfdist` utility in the input script.
3. Change the Main Input File 6.1 *Main Input Files (.tcl, .py, .ipynb)*:
   - Edit the .tcl file (you may want to save the old one) and utilize the pressure initial condition input file option (as referenced above) to specify the input files you created above as initial conditions for concentrations.
4. Restart the run :
   - Utilizing an editor recreate all the input parameters used in the run except for the following two items :
     - Use the dump sequence number from step 1 as the start_count.
     - Use the real time that the dump occured at from step 1 as the start_time.
     - To restart with `CLM`, use the `Solver.CLM.IstepStart` key described in *CLM Solver Parameters* with a value equal to the dump sequence plus one. Make sure this corresponds to changes to `drv_clmin.dat`.

3.4 Visualizing Output

While ParFlow does not have any visualization capabilities built-in, there are a number flexible, free options. Probably the best option is to use *VisIt*. *VisIt* is a powerful, free, open-source, rendering environment. It is multiplatform and may be downloaded directly from: https://visit.llnl.gov/. The most flexible option for using VisIt to view ParFlow output is to write files using the SILO format, which is available either as a direct output option (described in *Code Parameters*) or a conversion option using pftools. Many other output conversion options exist as described in *Manipulating Data: PFTools* and this allows ParFlow output to be converted into formats used by almost all visualization software.

3.5 Directory of Test Cases

ParFlow comes with a directory containing a few simple input files for use as templates in making new files and for use in testing the code. These files sit in the `/test` directory described earlier. This section gives a brief description of the problems in this directory.

`crater2D.tcl` An example of a two-dimensional, variably-saturated crater infiltration problem with time-varying boundary conditions. It uses the solid file `crater2D.pfsol`.

`default_richards.tcl` The default variably-saturated Richards' Equation simulation test script.

`default_single.tcl` The default parflow, single-processor, fully-saturated test script.

`forsyth2.tcl` An example two-dimensional, variably-saturated infiltration problem with layers of different hydraulic properties. It runs problem 2 in Forsyth *et al.* [FWP95] and uses the solid file `fors2_hf.pfsol`.

`harvey.flow.tcl` An example from Maxwell *et al.* [MWH07] for the Cape Cod bacterial injection site. This example is a three-dimensional, fully-saturated flow problem with spatially heterogeneous media (using a correlated, random field approach). It also provides examples of how tcl/tk scripts may be used in conjunction with ParFlow to loop iteratively or to run other scripts or programs. It uses the input text file `stats4.txt`. This input script is fully detailed in *Annotated Input Scripts*.

`default_overland.tcl` An overland flow boundary condition test and example script based loosely on the V-catchment problem in Kollet and Maxwell [KM06]. There are options provided to expand this problem into other overland flow-type, transient boundary-type problems included in the file as well.

`LW_var_dz_spinup.tcl` An example that uses the Little Washita domain to demonstrate a steady-state spinup initialization using P-E forcing. It also demonstrates the variable dz keys.

`LW_var_dz.tcl` An example that uses the Little Washita domain to demonstrate surface flow network development. It also uses the variable dz keys.

`Evap_Trans_test.tcl` An example that modifies the `default_overland.tcl` to demonstrate steady-state external flux `.pfb` files.

`overland_flux.tcl` An example that modifies the `default_overland.tcl` to demonstrate transient external flux `.pfb` files.

`/clm/clm.tcl` An example of how to use ParFlow coupled to `clm`. This directory also includes `clm`-specific input. Note: this problem will only run if `-with-clm` flag is used during the configure and build process.

`water_balance_x.tcl` and `water_balance_y.tcl`. An overland flow example script that uses the water-balance routines integrated into `pftools`. These two problems are based on simple overland flow conditions with slopes primarily in the x or y-directions. Note: this problem only will run if the Silo file capability is used, that is a `-with-silo=PATH` flag is used during the configure and build process.

`pfmg.tcl` and `pfmg_octree.tcl` Tests of the external Hypre preconditioner options. Note: this problem only will run if the Hypre capability is used, that is a `-with-hypre=PATH` flag is used during the configure and build process.

`test_x.tcl` A test problem for the Richards' solver that compares output to an analytical solution.

`/washita/tcl_scripts/LW_Test.tcl` A three day simulation of the Little Washita domain using ParFlow `CLM` with 3D forcings.

3.6 Annotated Input Scripts

This section contains two annotated input scripts:

- §3.6.1 *Harvey Flow Example* contains the harvey flow example (`harvey.flow.tcl`) which is an idealized domain with a heterogenous subsurface. The example also demonstrates how to generate multiple realizations of the subsurface and add pumping wells.
- §3.6.2 *Little Washita Example* contains the Little Washita example (`LW_Test.tcl`) which simulates a moderately sized (41km by 41km) real domain using ParFlow `CLM` with 3D meteorological forcings.

To run ParFlow, you use a script written in Tcl/TK. This script has a lot of flexibility, as it is somewhere in between a program and a user interface. The tcl script gives ParFlow the data it requires (or tells ParFlow where to find or read in that data) and also tells ParFlow to run.

To run the simulation:

1. Make any modifications to the tcl input script (and give a new name, if you want to)
2. Save the tcl script
3. For Linux/Unix/OSX: invoke the script from the command line using the tcl-shell, this looks like: `>tclsh filename.tcl`
4. Wait patiently for the command prompt to return (Linux/Unix/OSX) indicating that ParFlow has finished. Intermediate files are written as the simulation runs, however there is no other indication that ParFlow is running.

To modify a tcl script, you right-click and select edit from the menu. If you select open, you will run the script.

Note: The units for **K** (ım/d, usually) are critical to the entire construction. These length and time units for **K** set the units for all other variables (input or generated, throughout the entire simulation) in the simulation. ParFlow can set to solve using hydraulic conductivity by literally setting density, viscosity and gravity to one (as is done in the script below). This means the pressure units are in length (meters), so pressure is now so-called pressure-head.

3.6.1 Harvey Flow Example

This tutorial matches the `harvey_flow.tcl` file found in the `/test` directory. This example is directly from Maxwell *et al.* [MWH07]. This example demonstrates how to set up and run a fully saturated flow problem with heterogeneous hydraulic conductivity using the turning bands approach [TAG89]. Given statistical parameters describing the geology of your site, this script can be easily modified to make as many realizations of the subsurface as you like, each different and yet having the same statistical parameters, useful for a Monte Carlo simulation. This example is the basis for several fully-saturated ParFlow applications [AMNS13a, AMNS13b, CWM14, SM12a, SM12b, SNSMM12].

When the script runs, it creates a new directory named `/flow` right in the directory where the tcl script is stored. ParFlow then puts all its output in `/flow`. Of course, you can change the name and location of this output directory by modifying the tcl script that runs ParFlow.

Now for the tcl script:

```
#
# Import the ParFlow TCL package
#
```

These first three lines are what link ParFlow and the tcl script, thus allowing you to use a set of commands seen later, such as `pfset`, etc.

```
lappend auto_path $env(PARFLOW_DIR)/bin
package require parflow
namespace import Parflow::*

#-----------------------------------------------------------------------------
# File input version number
#-----------------------------------------------------------------------------
pfset FileVersion 4
```

These next lines set the parallel process topology. The domain is divided in *x*, *y* and *z* by `P`, `Q` and `R`. The total number of processors is `P*Q*R` (see *Computing Topology*).

```
#-----------------------------------------------------------------------------
# Process Topology
#-----------------------------------------------------------------------------

pfset Process.Topology.P     1
pfset Process.Topology.Q     1
pfset Process.Topology.R     1
```

Next we set up the computational grid (*see Defining the Problem* and *Computational Grid*).

```
#-----------------------------------------------------------------------------
# Computational Grid
#-----------------------------------------------------------------------------
```

Locate the origin in the domain.

```
pfset ComputationalGrid.Lower.X    0.0
pfset ComputationalGrid.Lower.Y    0.0
pfset ComputationalGrid.Lower.Z    0.0
```

Define the size of the domain grid block. Length units, same as those on hydraulic conductivity.

```
pfset ComputationalGrid.DX     0.34
pfset ComputationalGrid.DY     0.34
pfset ComputationalGrid.DZ     0.038
```

Define the number of grid blocks in the domain.

```
pfset ComputationalGrid.NX      50
pfset ComputationalGrid.NY      30
pfset ComputationalGrid.NZ      100
```

This next piece is comparable to a pre-declaration of variables. These will be areas in our domain geometry. The regions themselves will be defined later. You must always have one that is the name of your entire domain. If you want subsections within your domain, you may declare these as well. For Cape Cod, we have the entire domain, and also the 2 (upper and lower) permeability zones in the aquifer.

```
#-----------------------------------------------------------------------------
# The Names of the GeomInputs
#-----------------------------------------------------------------------------
pfset GeomInput.Names "domain_input upper_aquifer_input lower_aquifer_input"
```

Now you characterize your domain that you just pre-declared to be a `box` (see *Geometries*), and you also give it a name, `domain`.

```
#-----------------------------------------------------------------------------
# Domain Geometry Input
#-----------------------------------------------------------------------------
pfset GeomInput.domain_input.InputType       Box
pfset GeomInput.domain_input.GeomName   domain
```

Here, you set the limits in space for your entire domain. The span from `Lower.X` to `Upper.X` will be equal to the product of `ComputationalGrid.DX` times `ComputationalGrid.NX`. Same for Y and Z (i.e. the number of grid elements times size of the grid element has to equal the size of the grid in each dimension). The `Patches` key assigns names to the outside edges, because the domain is the limit of the problem in space.

```
#-----------------------------------------------------------------------------
# Domain Geometry
#-----------------------------------------------------------------------------
pfset Geom.domain.Lower.X       0.0
pfset Geom.domain.Lower.Y       0.0
pfset Geom.domain.Lower.Z       0.0

pfset Geom.domain.Upper.X       17.0
pfset Geom.domain.Upper.Y       10.2
pfset Geom.domain.Upper.Z       3.8

pfset Geom.domain.Patches "left right front back bottom top"
```

Just like domain geometry, you also set the limits in space for the individual components (upper and lower, as defined in the Names of GeomInputs pre-declaration). There are no patches for these geometries as they are internal to the domain.

```
#-----------------------------------------------------------------------------
# Upper Aquifer Geometry Input
#-----------------------------------------------------------------------------
pfset GeomInput.upper_aquifer_input.InputType        Box
pfset GeomInput.upper_aquifer_input.GeomName   upper_aquifer

#-----------------------------------------------------------------------------
# Upper Aquifer Geometry
#-----------------------------------------------------------------------------
pfset Geom.upper_aquifer.Lower.X                      0.0
pfset Geom.upper_aquifer.Lower.Y                      0.0
pfset Geom.upper_aquifer.Lower.Z                      1.5

pfset Geom.upper_aquifer.Upper.X                      17.0
pfset Geom.upper_aquifer.Upper.Y                      10.2
pfset Geom.upper_aquifer.Upper.Z                      1.5

#-----------------------------------------------------------------------------
# Lower Aquifer Geometry Input
#-----------------------------------------------------------------------------
pfset GeomInput.lower_aquifer_input.InputType       Box
pfset GeomInput.lower_aquifer_input.GeomName   lower_aquifer
```

(continues on next page)

(continued from previous page)

```
#-----------------------------------------------------------------------------
# Lower Aquifer Geometry
#-----------------------------------------------------------------------------
pfset Geom.lower_aquifer.Lower.X      0.0
pfset Geom.lower_aquifer.Lower.Y      0.0
pfset Geom.lower_aquifer.Lower.Z      0.0

pfset Geom.lower_aquifer.Upper.X     17.0
pfset Geom.lower_aquifer.Upper.Y     10.2
pfset Geom.lower_aquifer.Upper.Z      1.5
```

Now you add permeability data to the domain sections defined above (*Permeability*). You can reassign values simply by re-stating them – there is no need to comment out or delete the previous version – the final statement is the only one that counts.

```
#-----------------------------------------------------------------------------
# Perm
#-----------------------------------------------------------------------------
```

Name the permeability regions you will describe.

```
pfset Geom.Perm.Names "upper_aquifer lower_aquifer"
```

You can set, for example homogeneous, constant permeability, or you can generate a random field that meets your statistical requirements. To define a constant permeability for the entire domain:

```
#pfset Geom.domain.Perm.Type     Constant
#pfset Geom.domain.Perm.Value    4.0
```

However, for Cape Cod, we did not want a constant permeability field, so we instead generated a random permeability field meeting our statistical parameters for each the upper and lower zones. Third from the bottom is the `Seed`. This is a random starting point to generate the K field. Pick any large ODD number. First we do something tricky with Tcl/TK. We use the native commands within tcl to open a text file and read in locally set variables. Note we use set here and not pfset. One is a native tcl command, the other a ParFlow-specific command. For this problem, we are linking the parameter estimation code, PEST to ParFlow. PEST writes out the ascii file `stats4.txt` (also located in the `/test` directory) as the result of a calibration run. Since we are not coupled to PEST in this example, we just read in the file and use the values to assign statistical properties.

```
# we open a file, in this case from PEST to set upper and lower # kg and sigma
#
set fileId [open stats4.txt r 0600]
set kgu [gets $fileId]
set varu [gets $fileId]
set kgl [gets $fileId]
set varl [gets $fileId]
close $fileId
```

Now we set the heterogeneous parameters for the Upper and Lower aquifers (*see Permeability*). Note the special section at the very end of this block where we reset the geometric mean and standard deviation to our values we read in from a file. **Note:** ParFlow uses *Standard Deviation* not *Variance*.

```
pfset Geom.upper_aquifer.Perm.Type "TurnBands"
pfset Geom.upper_aquifer.Perm.LambdaX  3.60
```

(continues on next page)

(continued from previous page)

```
pfset Geom.upper_aquifer.Perm.LambdaY  3.60
pfset Geom.upper_aquifer.Perm.LambdaZ  0.19
pfset Geom.upper_aquifer.Perm.GeomMean  112.00

pfset Geom.upper_aquifer.Perm.Sigma   1.0
pfset Geom.upper_aquifer.Perm.Sigma   0.48989794
pfset Geom.upper_aquifer.Perm.NumLines 150
pfset Geom.upper_aquifer.Perm.RZeta  5.0
pfset Geom.upper_aquifer.Perm.KMax  100.0
pfset Geom.upper_aquifer.Perm.DelK  0.2
pfset Geom.upper_aquifer.Perm.Seed  33333
pfset Geom.upper_aquifer.Perm.LogNormal Log
pfset Geom.upper_aquifer.Perm.StratType Bottom
pfset Geom.lower_aquifer.Perm.Type "TurnBands"
pfset Geom.lower_aquifer.Perm.LambdaX  3.60
pfset Geom.lower_aquifer.Perm.LambdaY  3.60
pfset Geom.lower_aquifer.Perm.LambdaZ  0.19

pfset Geom.lower_aquifer.Perm.GeomMean  77.0
pfset Geom.lower_aquifer.Perm.Sigma   1.0
pfset Geom.lower_aquifer.Perm.Sigma   0.48989794
pfset Geom.lower_aquifer.Perm.NumLines 150
pfset Geom.lower_aquifer.Perm.RZeta  5.0
pfset Geom.lower_aquifer.Perm.KMax  100.0
pfset Geom.lower_aquifer.Perm.DelK  0.2
pfset Geom.lower_aquifer.Perm.Seed  33333
pfset Geom.lower_aquifer.Perm.LogNormal Log
pfset Geom.lower_aquifer.Perm.StratType Bottom

#pfset lower aqu and upper aq stats to pest/read in values

pfset Geom.upper_aquifer.Perm.GeomMean  $kgu
pfset Geom.upper_aquifer.Perm.Sigma  $varu

pfset Geom.lower_aquifer.Perm.GeomMean  $kgl
pfset Geom.lower_aquifer.Perm.Sigma  $varl
```

The following section allows you to specify the permeability tensor. In the case below, permeability is symmetric in all directions (x, y, and z) and therefore each is set to 1.0.

```
pfset Perm.TensorType               TensorByGeom

pfset Geom.Perm.TensorByGeom.Names  "domain"

pfset Geom.domain.Perm.TensorValX  1.0
pfset Geom.domain.Perm.TensorValY  1.0
pfset Geom.domain.Perm.TensorValZ  1.0
```

Next we set the specific storage, though this is not used in the IMPES/steady-state calculation.

```
#-----------------------------------------------------------------------------
# Specific Storage
```

(continues on next page)

(continued from previous page)

```
#-----------------------------------------------------------------------------
# specific storage does not figure into the impes (fully sat)
# case but we still need a key for it

pfset SpecificStorage.Type            Constant
pfset SpecificStorage.GeomNames       ""
pfset Geom.domain.SpecificStorage.Value 1.0e-4
```

ParFlow has the capability to deal with a multiphase system, but we only have one (water) at Cape Cod. As we stated earlier, we set density and viscosity artificially (and later gravity) both to 1.0. Again, this is merely a trick to solve for hydraulic conductivity and pressure head. If you were to set density and viscosity to their true values, the code would calculate **k** (permeability). By using the *normalized* values instead, you effectively embed the conversion of **k** to **K** (hydraulic conductivity). So this way, we get hydraulic conductivity, which is what we want for this problem.

```
#-----------------------------------------------------------------------------
# Phases
#-----------------------------------------------------------------------------

pfset Phase.Names "water"

pfset Phase.water.Density.Type       Constant
pfset Phase.water.Density.Value      1.0

pfset Phase.water.Viscosity.Type     Constant
pfset Phase.water.Viscosity.Value    1.0
```

We will not use the ParFlow grid based transport scheme. We will then leave contaminants blank because we will use a different code to model (virus, tracer) contamination.

```
#-----------------------------------------------------------------------------
# Contaminants
#-----------------------------------------------------------------------------
pfset Contaminants.Names                        ""
```

As with density and viscosity, gravity is normalized here. If we used the true value (in the *[L]* and *[T]* units of hydraulic conductivity) the code would be calculating permeability. Instead, we normalize so that the code calculates hydraulic conductivity.

```
#-----------------------------------------------------------------------------
# Gravity
#-----------------------------------------------------------------------------

pfset Gravity                               1.0

#-----------------------------------------------------------------------------
# Setup timing info
#-----------------------------------------------------------------------------
```

This basic time unit of 1.0 is used for transient boundary and well conditions. We are not using those features in this example.

```
pfset TimingInfo.BaseUnit            1.0
```

Cape Cod is a steady state problem, so these timing features are again unused, but need to be included.

```
pfset TimingInfo.StartCount   -1
pfset TimingInfo.StartTime     0.0
pfset TimingInfo.StopTime     0.0
```

Set the `dump interval` to -1 to report info at the end of every calculation, which in this case is only when steady state has been reached.

```
pfset TimingInfo.DumpInterval              -1
```

Next, we assign the porosity (*see* §6.1.12 *Porosity*). For the Cape Cod, the porosity is 0.39.

```
#-----------------------------------------------------------------------------
# Porosity
#-----------------------------------------------------------------------------

pfset Geom.Porosity.GeomNames          domain

pfset Geom.domain.Porosity.Type    Constant
pfset Geom.domain.Porosity.Value   0.390
```

Having defined the geometry of our problem before and named it `domain`, we are now ready to report/upload that problem, which we do here.

```
#-----------------------------------------------------------------------------
# Domain
#-----------------------------------------------------------------------------
pfset Domain.GeomName domain
```

Mobility between phases is set to 1.0 because we only have one phase (water).

```
#-----------------------------------------------------------------------------
# Mobility
#-----------------------------------------------------------------------------
pfset Phase.water.Mobility.Type        Constant
pfset Phase.water.Mobility.Value       1.0
```

Again, ParFlow has more capabilities than we are using here in the Cape Cod example. For this example, we handle monitoring wells in a separate code as we assume they do not remove a significant amount of water from the domain. Note that since there are no well names listed here, ParFlow assumes we have no wells. If we had pumping wells, we would have to include them here, because they would affect the head distribution throughout our domain. See below for an example of how to include pumping wells in this script.

```
#-----------------------------------------------------------------------------
# Wells
#-----------------------------------------------------------------------------
pfset Wells.Names ""
```

You can give certain periods of time names if you want to (ie. Pre-injection, post-injection, etc). Here, however we do not have multiple time intervals and are simulating in steady state, so time cycle keys are simple. We have only one time cycle and it's constant for the duration of the simulation. We accomplish this by giving it a repeat value of -1, which repeats indefinitely. The length of the cycle is the length specified below (an integer) multiplied by the base unit value we specified earlier.

```
#-----------------------------------------------------------------------------
# Time Cycles
#-----------------------------------------------------------------------------
pfset Cycle.Names constant
pfset Cycle.constant.Names           "alltime"
pfset Cycle.constant.alltime.Length   1
pfset Cycle.constant.Repeat          -1
```

Now, we assign Boundary Conditions for each face (each of the Patches in the domain defined before). Recall the previously stated Patches and associate them with the boundary conditions that follow.

```
pfset BCPressure.PatchNames "left right front back bottom top"
```

These are Dirichlet BCs (i.e. constant head over cell so the pressure head is set to hydrostatic– *see Boundary Conditions: Pressure*). There is no time dependence, so use the `constant` time cycle we defined previously. `RefGeom` links this to the established domain geometry and tells ParFlow what to use for a datum when calculating hydrostatic head conditions.

```
pfset Patch.left.BCPressure.Type          DirEquilRefPatch
pfset Patch.left.BCPressure.Cycle        "constant"
pfset Patch.left.BCPressure.RefGeom domain
```

Reference the current (left) patch to the bottom to define the line of intersection between the two.

```
pfset Patch.left.BCPressure.RefPatch  bottom
```

Set the head permanently to 10.0m. Pressure-head will of course vary top to bottom because of hydrostatics, but head potential will be constant.

```
pfset Patch.left.BCPressure.alltime.Value  10.0
```

Repeat the declarations for the rest of the faces of the domain. The left to right (X) dimension is aligned with the hydraulic gradient. The difference between the values assigned to right and left divided by the length of the domain corresponds to the correct hydraulic gradient.

```
pfset Patch.right.BCPressure.Type              DirEquilRefPatch
pfset Patch.right.BCPressure.Cycle             "constant"
pfset Patch.right.BCPressure.RefGeom       domain
pfset Patch.right.BCPressure.RefPatch       bottom
pfset Patch.right.BCPressure.alltime.Value 9.97501

pfset Patch.front.BCPressure.Type               FluxConst
pfset Patch.front.BCPressure.Cycle             "constant"
pfset Patch.front.BCPressure.alltime.Value 0.0

pfset Patch.back.BCPressure.Type               FluxConst
pfset Patch.back.BCPressure.Cycle              "constant"
pfset Patch.back.BCPressure.alltime.Value 0.0

pfset Patch.bottom.BCPressure.Type              FluxConst
pfset Patch.bottom.BCPressure.Cycle             "constant"
pfset Patch.bottom.BCPressure.alltime.Value 0.0

pfset Patch.top.BCPressure.Type                     FluxConst
```

(continues on next page)

(continued from previous page)

```
pfset Patch.top.BCPressure.Cycle                         "constant"
pfset Patch.top.BCPressure.alltime.Value                 0.0
```

Next we define topographic slopes and Mannings *n* values. These are not used, since we do not solve for overland flow. However, the keys still need to appear in the input script.

```
#---------------------------------------------------------
# Topo slopes in x-direction
#---------------------------------------------------------
# topo slopes do not figure into the impes (fully sat) case but we still
# need keys for them

pfset TopoSlopesX.Type "Constant"
pfset TopoSlopesX.GeomNames ""

pfset TopoSlopesX.Geom.domain.Value 0.0

#---------------------------------------------------------
# Topo slopes in y-direction
#---------------------------------------------------------

pfset TopoSlopesY.Type "Constant"
pfset TopoSlopesY.GeomNames ""

pfset TopoSlopesY.Geom.domain.Value 0.0

# You may also indicate an elevation file used to derive the slopes.
# This is optional but can be useful when post-processing terrain-
# following grids:
pfset TopoSlopes.Elevation.FileName "elevation.pfb"

#---------------------------------------------------------
# Mannings coefficient
#---------------------------------------------------------
# mannings roughnesses do not figure into the impes (fully sat) case but we still
# need a key for them

pfset Mannings.Type "Constant"
pfset Mannings.GeomNames ""
pfset Mannings.Geom.domain.Value 0.
```

Phase sources allows you to add sources other than wells and boundaries, but we do not have any so this key is constant, 0.0 over entire domain.

```
#-----------------------------------------------------------------------------
# Phase sources:
#-----------------------------------------------------------------------------

pfset PhaseSources.water.Type                             Constant
pfset PhaseSources.water.GeomNames                        domain
pfset PhaseSources.water.Geom.domain.Value        0.0
```

Next we define solver parameters for **IMPES**. Since this is the default solver, we do not need a solver key.

```
#-----------------------------------------------------------
#  Solver Impes
#-----------------------------------------------------------
```

We allow up to 50 iterations of the linear solver before it quits or converges.

```
pfset Solver.MaxIter 50
```

The solution must be accurate to this level

```
pfset Solver.AbsTol  1E-10
```

We drop significant digits beyond E-15

```
pfset Solver.Drop   1E-15

#----------------------------------------------------------
# Run and Unload the ParFlow output files
#-----------------------------------------------------------
```

Here you set the number of realizations again using a local tcl variable. We have set only one run but by setting the `n_runs` variable to something else we can run more than one realization of hydraulic conductivity.

```
# this script is setup to run 100 realizations, for testing we just run one
###set n_runs 100
set n_runs 1
```

Here is where you tell ParFlow where to put the output. In this case, it is a directory called flow. Then you cd (change directory) into that new directory. If you wanted to put an entire path rather than just a name, you would have more control over where your output file goes. For example, you would put `file mkdir “/cape_cod/revised_statistics/flow"` and then change into that directory.

```
file mkdir "flow"
cd "flow"
```

Now we loop through the realizations, again using tcl. `k` is the integer counter that is incremented for each realization. When you use a variable (rather than define it), you precede it with `$`. The hanging character `{` opens the do loop for `k`.

```
#
#  Loop through runs
#
for {set k 1} {$k <= $n_runs} {incr k 1} {
```

The following expressions sets the variable `seed` equal to the expression in brackets, which increments with each turn of the do loop and each seed will produce a different random field of K. You set upper and lower aquifer, because in the Cape Cod site, these are the two subsets of the domain. Note the seed starts at a different point to allow for different random field generation for the upper and lower zones.

```
#
# set the random seed to be different for every run
#
pfset Geom.upper_aquifer.Perm.Seed  [ expr 33333+2*$k ]
pfset Geom.lower_aquifer.Perm.Seed  [ expr 31313+2*$k ]
```

The following command runs ParFlow and gives you a suite of output files for each realization. The file names will begin `harvey_flow.1.xxxxx`, `harvey_flow.2.xxxx`, etc up to as many realizations as you run. The .xxxxx part will designate x, y, and z permeability, etc. Recall that in this case, since we normalized gravity, viscosity, and density, remember that we are really getting hydraulic conductivity.

```
pfrun harvey_flow.$k
```

This command removes a large number of superfluous dummy files or un-distributes parallel files back into a single file. If you compile with the `-with-amps-sequential-io` option then a single ParFlow file is written with corresponding `XXXX.dist` files and the `pfundist` command just removes these `.dist` files (though you don't really need to remove them if you don't want to).

```
pfundist harvey_flow.$k
```

The following commands take advantage of PFTools (*see PFTCL Commands*) and load pressure head output of the /parflow model into a pressure matrix.

```
# we use pf tools to convert from pressure to head
# we could do a number of other things here like copy files to different
# format
set press [pfload harvey_flow.$k.out.press.pfb]
```

The next command takes the pressures that were just loaded and converts it to head and loads them into a head matrix tcl variable.

```
set head [pfhhead $press]
```

Finally, the head matrix is saved as a ParFlow binary file (.pfb) and the k do loop is closed by the `}` character. Then we move up to the root directory when we are finished

```
 pfsave $head -pfb harvey_flow.$k.head.pfb
}

cd ".."
```

Once you have modified the tcl input script (if necessary) and run ParFlow, you will have as many realizations of your subsurface as you specified. Each of these realizations will be used as input for a particle or streamline calculation in the future. We can see below, that since we have a tcl script as input, we can do a lot of different operations, for example, we might run a particle tracking transport code simulation using the results of the ParFlow runs. This actually corresponds to the example presented in the `SLIM` user's manual.

```
# this could run other tcl scripts now an example is below
#puts stdout "running SLIM"
#source bromide_trans.sm.tcl
```

We can add options to this script. For example if we wanted to add a pumping well these additions are described below.

3.6.2 Adding a Pumping Well

Let us change the input problem by adding a pumping well:

Add the following lines to the input file near where the existing well information is in the input file. You need to replace the "Wells.Names" line with the one included here to get both wells activated (this value lists the names of the wells):

```
pfset Wells.Names {new_well}

pfset Wells.new_well.InputType                Recirc

pfset Wells.new_well.Cycle                 constant

pfset Wells.new_well.ExtractionType        Flux
pfset Wells.new_well.InjectionType            Flux

pfset Wells.new_well.X                     10.0
pfset Wells.new_well.Y                     10.0
pfset Wells.new_well.ExtractionZLower       0.5
pfset Wells.new_well.ExtractionZUpper       0.5
pfset Wells.new_well.InjectionZLower        0.2
pfset Wells.new_well.InjectionZUpper        0.2

pfset Wells.new_well.ExtractionMethod      Standard
pfset Wells.new_well.InjectionMethod          Standard

pfset Wells.new_well.alltime.Extraction.Flux.water.Value                     0.50
pfset Wells.new_well.alltime.Injection.Flux.water.Value                      0.75
```

For more information on defining the problem, see *Defining the Problem*.

We could also visualize the results of the ParFlow simulations, using *VisIt*. For example, we can turn on *SILO* file output which allows these files to be directly read and visualized. We would do this by adding the following `pfset` commands, I usually add them to t he solver section:

```
pfset Solver.WriteSiloSubsurfData True
pfset Solver.WriteSiloPressure True
pfset Solver.WriteSiloSaturation True
```

You can then directly open the file `harvey_flow.#.out.perm_x.silo` (where `#` is the realization number). The resulting image will be the hydraulic conductivity field of your domain, showing the variation in x-permeability in 3-D space. You can also generate representations of head or pressure (or y or z permeability) throughout your domain using ParFlow output files. See the section on visualization for more details.

3.6.3 Little Washita Example

This tutorial matches the `LW_Test.tcl` file found in the `/test/washita/tcl_scripts` directory and corresponds to Condon and Maxwell [CM14a], Condon and Maxwell [CM14b]. This script runs the Little Washita domain for three days using ParFlow `CLM` with 3D forcings. The domain is setup using terrain following grid (*Terrain Following Grid*) and subsurface geologes are specified using a `.pfb` indicator file. Input files were generated using the workflow detailed in *Setting Up a Real Domain*.

Now for the tcl script:

```
#
# Import the ParFlow TCL package
#
```

These first three lines are what link ParFlow and the tcl script, thus allowing you to use a set of commands seen later, such as `pfset`, etc.

```
lappend auto_path $env(PARFLOW_DIR)/bin
package require parflow
namespace import Parflow::*

#-----------------------------------------------------------------------------
# File input version number
#-----------------------------------------------------------------------------
pfset FileVersion 4
```

These next lines set the parallel process topology. The domain is divided in *x*, *y* and *z* by `P`, `Q` and `R`. The total number of processors is `P*Q*R` (see *Computing Topology*).

```
#-----------------------------------------------------------------------------
# Process Topology
#-----------------------------------------------------------------------------

pfset Process.Topology.P     1
pfset Process.Topology.Q     1
pfset Process.Topology.R     1
```

Before we really get started make a directory for our outputs and copy all of the required input files into the run directory. These files will be described in detail later as they get used.

```
#-----------------------------------------------------------------------------
# Make a directory for the simulation and copy inputs into it
#-----------------------------------------------------------------------------
exec mkdir "Outputs"
cd "./Outputs"

# ParFlow Inputs
file copy -force "../../parflow_input/LW.slopex.pfb" .
file copy -force "../../parflow_input/LW.slopey.pfb" .
file copy -force "../../parflow_input/IndicatorFile_Gleeson.50z.pfb"   .
file copy -force "../../parflow_input/press.init.pfb"  .

#CLM Inputs
file copy -force "../../clm_input/drv_clmin.dat" .
file copy -force "../../clm_input/drv_vegp.dat"  .
file copy -force "../../clm_input/drv_vegm.alluv.dat"  .

puts "Files Copied"
```

Next we set up the computational grid (*see Defining the Problem* and *Computational Grid*).

```
#-----------------------------------------------------------------------------
# Computational Grid
#-----------------------------------------------------------------------------
```

Locate the origin in the domain.

```
pfset ComputationalGrid.Lower.X    0.0
pfset ComputationalGrid.Lower.Y    0.0
pfset ComputationalGrid.Lower.Z    0.0
```

Define the size of the domain grid block. Length units, same as those on hydraulic conductivity.

```
pfset ComputationalGrid.DX     1000.0
pfset ComputationalGrid.DY     1000.0
pfset ComputationalGrid.DZ     2.0
```

Define the number of grid blocks in the domain.

```
pfset ComputationalGrid.NX      41
pfset ComputationalGrid.NY      41
pfset ComputationalGrid.NZ      50
```

This next piece is comparable to a pre-declaration of variables. These will be areas in our domain geometry. The regions themselves will be defined later. You must always have one that is the name of your entire domain. If you want subsections within your domain, you may declare these as well. Here we define two geometries one is the domain and one is for the indicator file (which will also span the entire domain).

```
#-----------------------------------------------------------------------------
# The Names of the GeomInputs
#-----------------------------------------------------------------------------
pfset GeomInput.Names                     "box_input indi_input"
```

Now you characterize the domain that you just pre-declared to be a `box` (see *Geometries*), and you also give it a name, `domain`.

```
#-----------------------------------------------------------------------------
# Domain Geometry Input
#-----------------------------------------------------------------------------
pfset GeomInput.box_input.InputType      Box
pfset GeomInput.box_input.GeomName      domain
```

Here, you set the limits in space for your entire domain. The span from `Lower.X` to `Upper.X` will be equal to the product of `ComputationalGrid.DX` times `ComputationalGrid.NX`. Same for Y and Z (i.e. the number of grid elements times size of the grid element has to equal the size of the grid in each dimension). The `Patches` key assigns names to the outside edges, because the domain is the limit of the problem in space.

```
#-----------------------------------------------------------------------------
# Domain Geometry
#-----------------------------------------------------------------------------
pfset Geom.domain.Lower.X                        0.0
pfset Geom.domain.Lower.Y                        0.0
pfset Geom.domain.Lower.Z                        0.0

pfset Geom.domain.Upper.X                        41000.0
pfset Geom.domain.Upper.Y                        41000.0
pfset Geom.domain.Upper.Z                          100.0

pfset Geom.domain.Patches            "x-lower x-upper y-lower y-upper z-lower z-upper"
```

Now we setup the indicator file. As noted above, the indicator file has integer values for every grid cell in the domain designating what geologic unit it belongs to. The GeomNames list should include a name for every unit in your indicator file. In this example we have thirteen soil units and eight geologic units. The FileName points to the indicator file that ParFlow will read. Recall that this file into the run directory at the start of the script.

```
#-----------------------------------------------------------------------------
# Indicator Geometry Input
#-----------------------------------------------------------------------------
pfset GeomInput.indi_input.InputType      IndicatorField
pfset GeomInput.indi_input.GeomNames      "s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 g1␣
↪g2 g3 g4 g5 g6 g7 g8"
pfset Geom.indi_input.FileName            "IndicatorFile_Gleeson.50z.pfb"
```

For every name in the GeomNames list we define the corresponding value in the indicator file. For example, here we are saying that our first soil unit (s1) is represented by the number "1" in the indicator file, while the first geologic unit (g1) is represented by the number "21". Note that the integers used in the indicator file do not need to be consecutive.

```
pfset GeomInput.s1.Value                 1
pfset GeomInput.s2.Value                 2
pfset GeomInput.s3.Value                 3
pfset GeomInput.s4.Value                 4
pfset GeomInput.s5.Value                 5
pfset GeomInput.s6.Value                 6
pfset GeomInput.s7.Value                 7
pfset GeomInput.s8.Value                 8
pfset GeomInput.s9.Value                 9
pfset GeomInput.s10.Value                10
pfset GeomInput.s11.Value                11
pfset GeomInput.s12.Value                12
pfset GeomInput.s13.Value                13
pfset GeomInput.g1.Value                 21
pfset GeomInput.g2.Value                 22
pfset GeomInput.g3.Value                 23
pfset GeomInput.g4.Value                 24
pfset GeomInput.g5.Value                 25
pfset GeomInput.g6.Value                 26
pfset GeomInput.g7.Value                 27
pfset GeomInput.g8.Value                 28
```

Now you add permeability data to the domain sections defined above (*Permeability*). You can reassign values simply by re-stating them – there is no need to comment out or delete the previous version – the final statement is the only one that counts. Also, note that you do not need to assign permeability values to all of the geometries names. Any geometry that is not assigned its own permeability value will take the domain value. However, every geometry listed in Porosity.GeomNames must have values assigned.

```
#-----------------------------------------------------------------------------
# Permeability (values in m/hr)
#-----------------------------------------------------------------------------
pfset Geom.Perm.Names                     "domain s1 s2 s3 s4 s5 s6 s7 s8 s9 g2 g3 g6 g8"

pfset Geom.domain.Perm.Type           Constant
pfset Geom.domain.Perm.Value          0.2

pfset Geom.s1.Perm.Type               Constant
```

(continues on next page)

(continued from previous page)

```
pfset Geom.s1.Perm.Value                0.269022595

pfset Geom.s2.Perm.Type                 Constant
pfset Geom.s2.Perm.Value                0.043630356

pfset Geom.s3.Perm.Type                 Constant
pfset Geom.s3.Perm.Value                0.015841225

pfset Geom.s4.Perm.Type                 Constant
pfset Geom.s4.Perm.Value                0.007582087

pfset Geom.s5.Perm.Type                 Constant
pfset Geom.s5.Perm.Value                0.01818816

pfset Geom.s6.Perm.Type                 Constant
pfset Geom.s6.Perm.Value                0.005009435

pfset Geom.s7.Perm.Type                 Constant
pfset Geom.s7.Perm.Value                0.005492736

pfset Geom.s8.Perm.Type                 Constant
pfset Geom.s8.Perm.Value                0.004675077

pfset Geom.s9.Perm.Type                 Constant
pfset Geom.s9.Perm.Value                0.003386794

pfset Geom.g2.Perm.Type                 Constant
pfset Geom.g2.Perm.Value                0.025

pfset Geom.g3.Perm.Type                 Constant
pfset Geom.g3.Perm.Value                0.059

pfset Geom.g6.Perm.Type                 Constant
pfset Geom.g6.Perm.Value                0.2

pfset Geom.g8.Perm.Type                Constant
pfset Geom.g8.Perm.Value               0.68
```

The following section allows you to specify the permeability tensor. In the case below, permeability is symmetric in all directions (x, y, and z) and therefore each is set to 1.0. Also note that we just specify this once for the whole domain because we want isotropic permeability everywhere. You can specify different tensors for different units by repeating these lines with different `Geom.Names`.

```
pfset Perm.TensorType                       TensorByGeom
pfset Geom.Perm.TensorByGeom.Names          "domain"
pfset Geom.domain.Perm.TensorValX           1.0d0
pfset Geom.domain.Perm.TensorValY           1.0d0
pfset Geom.domain.Perm.TensorValZ           1.0d0
```

Next we set the specific storage. Here again we specify one value for the whole domain but these lines can be easily repeated to set different values for different units.

```
#-----------------------------------------------------------------------------
# Specific Storage
#-----------------------------------------------------------------------------
pfset SpecificStorage.Type                     Constant
pfset SpecificStorage.GeomNames                "domain"
pfset Geom.domain.SpecificStorage.Value        1.0e-5
```

ParFlow has the capability to deal with a multiphase system, but we only have one (water) in this example. As we stated earlier, we set density and viscosity artificially (and later gravity) both to 1.0. Again, this is merely a trick to solve for hydraulic conductivity and pressure head. If you were to set density and viscosity to their true values, the code would calculate **k** (permeability). By using the *normalized* values instead, you effectively embed the conversion of **k** to **K** (hydraulic conductivity). So this way, we get hydraulic conductivity, which is what we want for this problem.

```
#-----------------------------------------------------------------------------
# Phases
#-----------------------------------------------------------------------------
pfset Phase.Names                              "water"

pfset Phase.water.Density.Type                 Constant
pfset Phase.water.Density.Value                1.0

pfset Phase.water.Viscosity.Type               Constant
pfset Phase.water.Viscosity.Value              1.0
```

This example does not include the ParFlow grid based transport scheme. Therefore we leave contaminants blank.

```
#-----------------------------------------------------------------------------
# Contaminants
#-----------------------------------------------------------------------------
pfset Contaminants.Names                       ""
```

As with density and viscosity, gravity is normalized here. If we used the true value (in the *[L]* and *[T]* units of hydraulic conductivity) the code would be calculating permeability. Instead, we normalize so that the code calculates hydraulic conductivity.

```
#-----------------------------------------------------------------------------
# Gravity
#-----------------------------------------------------------------------------
pfset Gravity                                  1.0
```

Next we set up the timing for our simulation.

```
#-----------------------------------------------------------------------------
# Timing (time units is set by units of permeability)
#-----------------------------------------------------------------------------
```

This specifies the base unit of time for all time values entered. All time should be expressed as multiples of this value. To keep things simple here we set it to 1. Because we expressed our permeability in units of m/hr in this example this means that our basin unit of time is 1hr.

```
pfset TimingInfo.BaseUnit                      1.0
```

This key specifies the time step number that will be associated with the first advection cycle of the transient problem. Because we are starting from scratch we set this to 0. If we were restarting a run we would set this to the last time step of your previous simulation. Refer to §3.3 *Restarting a Run* for additional instructions on restarting a run.

```
pfset TimingInfo.StartCount                 0.0
```

`StartTime` and `StopTime` specify the start and stop times for the simulation. These values should correspond with the forcing files you are using.

```
pfset TimingInfo.StartTime                  0.0
pfset TimingInfo.StopTime                   72.0
```

This key specifies the timing interval at which ParFlow time dependent outputs will be written. Here we have a base unit of 1hr so a dump interval of 24 means that we are writing daily outputs. Note that this key only controls the ParFlow output interval and not the interval that `CLM` outputs will be written out at.

```
pfset TimingInfo.DumpInterval               24.0
```

Here we set the time step value. For this example we use a constant time step of 1hr.

```
pfset TimeStep.Type                         Constant
pfset TimeStep.Value                        1.0
```

Next, we assign the porosity (*see* §6.1.12 *Porosity*). As with the permeability we assign different values for different indicator geometries. Here we assign values for all of our soil units but not for the geologic units, they will default to the domain value of 0.4. Note that every geometry listed in `Porosity.GeomNames` must have values assigned.

```
#-----------------------------------------------------------------------------
# Porosity
#-----------------------------------------------------------------------------
pfset Geom.Porosity.GeomNames               "domain s1 s2 s3 s4 s5 s6 s7 s8 s9"

pfset Geom.domain.Porosity.Type            Constant
pfset Geom.domain.Porosity.Value           0.4

pfset Geom.s1.Porosity.Type    Constant
pfset Geom.s1.Porosity.Value   0.375

pfset Geom.s2.Porosity.Type    Constant
pfset Geom.s2.Porosity.Value   0.39

pfset Geom.s3.Porosity.Type    Constant
pfset Geom.s3.Porosity.Value   0.387

pfset Geom.s4.Porosity.Type    Constant
pfset Geom.s4.Porosity.Value   0.439

pfset Geom.s5.Porosity.Type    Constant
pfset Geom.s5.Porosity.Value   0.489

pfset Geom.s6.Porosity.Type    Constant
pfset Geom.s6.Porosity.Value   0.399

pfset Geom.s7.Porosity.Type    Constant
pfset Geom.s7.Porosity.Value   0.384

pfset Geom.s8.Porosity.Type            Constant
```

(continues on next page)

(continued from previous page)

```
pfset Geom.s8.Porosity.Value          0.482

pfset Geom.s9.Porosity.Type           Constant
pfset Geom.s9.Porosity.Value          0.442
```

Having defined the geometry of our problem before and named it `domain`, we are now ready to report/upload that problem, which we do here.

```
#-----------------------------------------------------------------------------
# Domain
#-----------------------------------------------------------------------------
pfset Domain.GeomName                    "domain"
```

Mobility between phases is set to 1.0 because we only have one phase (water).

```
#----------------------------------------------------------------------------
# Mobility
#----------------------------------------------------------------------------
pfset Phase.water.Mobility.Type       Constant
pfset Phase.water.Mobility.Value      1.0
```

Again, ParFlow has more capabilities than we are using here in this example. Note that since there are no well names listed here, ParFlow assumes we have no wells. If we had pumping wells, we would have to include them here, because they would affect the head distribution throughout our domain. See *Harvey Flow Example* for an example of how to include pumping wells in this script.

```
#-----------------------------------------------------------------------------
# Wells
#-----------------------------------------------------------------------------
pfset Wells.Names                        ""
```

You can give certain periods of time names if you want. For example if you aren't running with `CLM` and you would like to have periods with rain and periods without. Here, however we have only one time cycle because `CLM` will handle the variable forcings. Therefore, we specify one time cycle and it's constant for the duration of the simulation. We accomplish this by giving it a repeat value of -1, which repeats indefinitely. The length of the cycle is the length specified below (an integer) multiplied by the base unit value we specified earlier.

```
#-----------------------------------------------------------------------------
# Time Cycles
#-----------------------------------------------------------------------------
pfset Cycle.Names                        "constant"
pfset Cycle.constant.Names               "alltime"
pfset Cycle.constant.alltime.Length       1
pfset Cycle.constant.Repeat              -1
```

Now, we assign Boundary Conditions for each face (each of the Patches in the domain defined before). Recall the previously stated Patches and associate them with the boundary conditions that follow.

```
#-----------------------------------------------------------------------------
# Boundary Conditions
#-----------------------------------------------------------------------------
pfset BCPressure.PatchNames                  [pfget Geom.domain.Patches]
```

The bottom and sides of our domain are all set to no-flow (i.e. constant flux of 0) boundaries.

```
pfset Patch.x-lower.BCPressure.Type                FluxConst
pfset Patch.x-lower.BCPressure.Cycle               "constant"
pfset Patch.x-lower.BCPressure.alltime.Value       0.0

pfset Patch.y-lower.BCPressure.Type                FluxConst
pfset Patch.y-lower.BCPressure.Cycle               "constant"
pfset Patch.y-lower.BCPressure.alltime.Value       0.0

pfset Patch.z-lower.BCPressure.Type                FluxConst
pfset Patch.z-lower.BCPressure.Cycle               "constant"
pfset Patch.z-lower.BCPressure.alltime.Value       0.0

pfset Patch.x-upper.BCPressure.Type                FluxConst
pfset Patch.x-upper.BCPressure.Cycle               "constant"
pfset Patch.x-upper.BCPressure.alltime.Value       0.0

pfset Patch.y-upper.BCPressure.Type                FluxConst
pfset Patch.y-upper.BCPressure.Cycle               "constant"
pfset Patch.y-upper.BCPressure.alltime.Value       0.0
```

The top is set to an `OverlandFLow` boundary to turn on the fully-coupled overland flow routing.

```
pfset Patch.z-upper.BCPressure.Type                OverlandFlow
pfset Patch.z-upper.BCPressure.Cycle               "constant"
pfset Patch.z-upper.BCPressure.alltime.Value       0.0
```

Next we define topographic slopes and values. These slope values were derived from a digital elevation model of the domain following the workflow outlined in *Setting Up a Real Domain*. In this example we read the slope files in from `.pfb` files that were copied into the run directory at the start of this script.

```
#-----------------------------------------------------------------------------
# Topo slopes in x-direction
#-----------------------------------------------------------------------------
pfset TopoSlopesX.Type                                "PFBFile"
pfset TopoSlopesX.GeomNames                           "domain"
pfset TopoSlopesX.FileName                            "LW.slopex.pfb"

#-----------------------------------------------------------------------------
# Topo slopes in y-direction
#-----------------------------------------------------------------------------
pfset TopoSlopesY.Type                                "PFBFile"
pfset TopoSlopesY.GeomNames                           "domain"
pfset TopoSlopesY.FileName                            "LW.slopey.pfb"
```

And now we define the Mannings n, again just one value for the whole domain in this example.

```
#-----------------------------------------------------------------------------
# Mannings coefficient
#-----------------------------------------------------------------------------
pfset Mannings.Type                                   "Constant"
pfset Mannings.GeomNames                              "domain"
pfset Mannings.Geom.domain.Value                      5.52e-6
```

Following the same approach as we did for `Porosity` we define the relative permeability inputs that will be used for Richards' equation implementation (*Richards' Equation Relative Permeabilities*). Here we use `VanGenuchten` parameters. Note that every geometry listed in `Porosity.GeomNames` must have values assigned.

```
#-----------------------------------------------------------------------------
# Relative Permeability
#-----------------------------------------------------------------------------
pfset Phase.RelPerm.Type                    VanGenuchten
pfset Phase.RelPerm.GeomNames               "domain s1 s2 s3 s4 s5 s6 s7 s8 s9 "

pfset Geom.domain.RelPerm.Alpha             3.5
pfset Geom.domain.RelPerm.N                 2.0

pfset Geom.s1.RelPerm.Alpha          3.548
pfset Geom.s1.RelPerm.N              4.162

pfset Geom.s2.RelPerm.Alpha          3.467
pfset Geom.s2.RelPerm.N              2.738

pfset Geom.s3.RelPerm.Alpha          2.692
pfset Geom.s3.RelPerm.N              2.445

pfset Geom.s4.RelPerm.Alpha          0.501
pfset Geom.s4.RelPerm.N              2.659

pfset Geom.s5.RelPerm.Alpha          0.661
pfset Geom.s5.RelPerm.N              2.659

pfset Geom.s6.RelPerm.Alpha          1.122
pfset Geom.s6.RelPerm.N              2.479

pfset Geom.s7.RelPerm.Alpha          2.089
pfset Geom.s7.RelPerm.N              2.318

pfset Geom.s8.RelPerm.Alpha          0.832
pfset Geom.s8.RelPerm.N              2.514

pfset Geom.s9.RelPerm.Alpha          1.585
pfset Geom.s9.RelPerm.N              2.413
```

Next we do the same thing for saturation (*Saturation*) again using the `VanGenuchten` parameters Note that every geometry listed in `Porosity.GeomNames` must have values assigned.

```
#-----------------------------------------------------------------------------
# Saturation
#-----------------------------------------------------------------------------
pfset Phase.Saturation.Type                 VanGenuchten
pfset Phase.Saturation.GeomNames            "domain s1 s2 s3 s4 s5 s6 s7 s8 s9 "

pfset Geom.domain.Saturation.Alpha          3.5
pfset Geom.domain.Saturation.N              2.
pfset Geom.domain.Saturation.SRes           0.2
pfset Geom.domain.Saturation.SSat           1.0

pfset Geom.s1.Saturation.Alpha        3.548
```

(continues on next page)

(continued from previous page)

```
pfset Geom.s1.Saturation.N              4.162
pfset Geom.s1.Saturation.SRes           0.000001
pfset Geom.s1.Saturation.SSat           1.0

pfset Geom.s2.Saturation.Alpha          3.467
pfset Geom.s2.Saturation.N              2.738
pfset Geom.s2.Saturation.SRes           0.000001
pfset Geom.s2.Saturation.SSat           1.0

pfset Geom.s3.Saturation.Alpha          2.692
pfset Geom.s3.Saturation.N              2.445
pfset Geom.s3.Saturation.SRes           0.000001
pfset Geom.s3.Saturation.SSat           1.0

pfset Geom.s4.Saturation.Alpha          0.501
pfset Geom.s4.Saturation.N              2.659
pfset Geom.s4.Saturation.SRes           0.000001
pfset Geom.s4.Saturation.SSat           1.0

pfset Geom.s5.Saturation.Alpha          0.661
pfset Geom.s5.Saturation.N              2.659
pfset Geom.s5.Saturation.SRes           0.000001
pfset Geom.s5.Saturation.SSat           1.0

pfset Geom.s6.Saturation.Alpha          1.122
pfset Geom.s6.Saturation.N              2.479
pfset Geom.s6.Saturation.SRes           0.000001
pfset Geom.s6.Saturation.SSat           1.0

pfset Geom.s7.Saturation.Alpha          2.089
pfset Geom.s7.Saturation.N              2.318
pfset Geom.s7.Saturation.SRes           0.000001
pfset Geom.s7.Saturation.SSat           1.0

pfset Geom.s8.Saturation.Alpha          0.832
pfset Geom.s8.Saturation.N              2.514
pfset Geom.s8.Saturation.SRes           0.000001
pfset Geom.s8.Saturation.SSat           1.0

pfset Geom.s9.Saturation.Alpha          1.585
pfset Geom.s9.Saturation.N              2.413
pfset Geom.s9.Saturation.SRes           0.000001
pfset Geom.s9.Saturation.SSat           1.0
```

Phase sources allows you to add sources other than wells and boundaries, but we do not have any so this key is constant, 0.0 over entire domain.

```
#-----------------------------------------------------------------------------
# Phase sources:
#-----------------------------------------------------------------------------
pfset PhaseSources.water.Type                           "Constant"
pfset PhaseSources.water.GeomNames                      "domain"
```

(continues on next page)

(continued from previous page)

```
pfset PhaseSources.water.Geom.domain.Value                0.0
```

In this example we are using ParFlow `CLM` so we must provide some parameters for `CLM` (*CLM Solver Parameters*). Note that `CLM` will also require some additional inputs outside of the tcl script. Refer to `/washita/clm_input/` for examples of the `CLM`, `vegm` and `driver` files. These inputs are also discussed briefly in *Setting Up a Real Domain*.

```
#-----------------------------------------------------------------
# CLM Settings:
# ------------------------------------------------------------
```

First we specify that we will be using `CLM` as the land surface model and provide the name of a directory that outputs will be written to. For this example we do not need outputs for each processor or a binary output directory. Finally we set the dump interval to 1, indicating that we will be writing outputs for every time step. Note that this does not have to match the dump interval for ParFlow outputs. Recall that earlier we set the ParFlow dump interval to 24.

```
pfset Solver.LSM                                          CLM
pfset Solver.CLM.CLMFileDir                               "clm_output/"
pfset Solver.CLM.Print1dOut                               False
pfset Solver.BinaryOutDir                                 False
pfset Solver.CLM.CLMDumpInterval                          1
```

Next we specify the details of the meteorological forcing files that `CLM` will read. First we provide the name of the files and the directory they can be found in. Next we specify that we are using `3D` forcing files meaning that we have spatially distributed forcing with multiple time steps in every file. Therefore we must also specify the number of times steps (`MetFileNT`) in every file, in this case 24. Finally, we specify the initial value for the CLM counter.

```
pfset Solver.CLM.MetFileName                              "NLDAS"
pfset Solver.CLM.MetFilePath                              "../../NLDAS/"
pfset Solver.CLM.MetForcing                               3D
pfset Solver.CLM.MetFileNT                                24
pfset Solver.CLM.IstepStart                               1
```

This last set of `CLM` parameters refers to the physical properties of the system. Refer to *CLM Solver Parameters* for details.

```
pfset Solver.CLM.EvapBeta                                 Linear
pfset Solver.CLM.VegWaterStress                           Saturation
pfset Solver.CLM.ResSat                                   0.1
pfset Solver.CLM.WiltingPoint                             0.12
pfset Solver.CLM.FieldCapacity                            0.98
pfset Solver.CLM.IrrigationType                           none
```

Next we set the initial conditions for the domain. In this example we are using a pressure `.pfb` file that was obtained by spinning up the model in the workflow outlined in *Setting Up a Real Domain*. Alternatively, the water table can be set to a constant value by changing the `ICPressure.Type`. Again, the input file that is referenced here was was copied into the run directory at the top of this script.

```
#---------------------------------------------------------
# Initial conditions: water pressure
#---------------------------------------------------------
pfset ICPressure.Type                                     PFBFile
pfset ICPressure.GeomNames                                domain
```

(continues on next page)

(continued from previous page)

```
pfset Geom.domain.ICPressure.RefPatch                  z-upper
pfset Geom.domain.ICPressure.FileName                  press.init.pfb
```

Now we specify what outputs we would like written. In this example we specify that we would like to write out `CLM` variables as well as `Pressure` and `Saturation`. However, there are many options for this and you should change these options according to what type of analysis you will be performing on your results. A complete list of print options is provided in *Code Parameters*.

```
#-----------------------------------------------------------------
# Outputs
# --------------------------------------------------------------
#Writing output (all pfb):
pfset Solver.PrintSubsurfData                        False
pfset Solver.PrintPressure                           True
pfset Solver.PrintSaturation                         True
pfset Solver.PrintMask                               True

pfset Solver.WriteCLMBinary                          False
pfset Solver.PrintCLM                                True
pfset Solver.WriteSiloSpecificStorage                False
pfset Solver.WriteSiloMannings                       False
pfset Solver.WriteSiloMask                           False
pfset Solver.WriteSiloSlopes                         False
pfset Solver.WriteSiloSubsurfData                    False
pfset Solver.WriteSiloPressure                       False
pfset Solver.WriteSiloSaturation                     False
pfset Solver.WriteSiloEvapTrans                      False
pfset Solver.WriteSiloEvapTransSum                   False
pfset Solver.WriteSiloOverlandSum                    False
pfset Solver.WriteSiloCLM                            False
```

Next we specify the solver settings for the ParFlow (*Richards' Equation Solver Parameters*). First we turn on solver Richards and the terrain following grid. We turn off variable dz.

```
#-----------------------------------------------------------------------------
# Set solver parameters
#-----------------------------------------------------------------------------
# ParFlow Solution
pfset Solver                                         Richards
pfset Solver.TerrainFollowingGrid                    True
pfset Solver.Nonlinear.VariableDz                    False
```

We then set the max solver settings and linear and nonlinear convergence tolerance settings. The linear system will be solved to a norm of 10^{-8} and the nonlinear system will be solved to less than 10^{-6}. Of note in latter key block is the EtaChoice and that we use the analytical Jacobian (*UseJacobian* = **True**). We are using the *FullJacobian* preconditioner, which is a more robust approach but is more expensive.

```
pfset Solver.MaxIter                                 25000
pfset Solver.Drop                                    1E-20
pfset Solver.AbsTol                                  1E-8
pfset Solver.MaxConvergenceFailures                  8
pfset Solver.Nonlinear.MaxIter                       80
pfset Solver.Nonlinear.ResidualTol                   1e-6
```

(continues on next page)

(continued from previous page)

```
pfset Solver.Nonlinear.EtaChoice                         EtaConstant
pfset Solver.Nonlinear.EtaValue                          0.001
pfset Solver.Nonlinear.UseJacobian                       True
pfset Solver.Nonlinear.DerivativeEpsilon                 1e-16
pfset Solver.Nonlinear.StepTol                                                 1e-
↪30
pfset Solver.Nonlinear.Globalization                     LineSearch
pfset Solver.Linear.KrylovDimension                      70
pfset Solver.Linear.MaxRestarts                           2

pfset Solver.Linear.Preconditioner                       PFMG
pfset Solver.Linear.Preconditioner.PCMatrixType     FullJacobian
```

This key is just for testing the Richards' formulation, so we are not using it.

```
#-----------------------------------------------------------------------------
# Exact solution specification for error calculations
#-----------------------------------------------------------------------------
pfset KnownSolution                                    NoKnownSolution
```

Next we distribute all the inputs as described by the keys in *PFTCL Commands*. Note the slopes are 2D files, while the rest of the ParFlow inputs are 3D so we need to alter the NZ accordingly following example 4 in *Common examples using ParFlow TCL commands (PFTCL)*.

```
#-----------------------------------------------------------------------------
# Distribute inputs
#-----------------------------------------------------------------------------
pfset ComputationalGrid.NX                41
pfset ComputationalGrid.NY                41
pfset ComputationalGrid.NZ                1
pfdist LW.slopex.pfb
pfdist LW.slopey.pfb

pfset ComputationalGrid.NX                41
pfset ComputationalGrid.NY                41
pfset ComputationalGrid.NZ                50
pfdist IndicatorFile_Gleeson.50z.pfb
pfdist press.init.pfb
```

Now we run the simulation. Note that we use a tcl variable to set the run name.

```
#-----------------------------------------------------------------------------
# Run Simulation
#-----------------------------------------------------------------------------
set runname "LW"
puts $runname
pfrun    $runname
```

All that is left is to undistribute files.

```
#-----------------------------------------------------------------------------
# Undistribute Files
```

(continues on next page)

(continued from previous page)

```
#-----------------------------------------------------------------------------
pfundist $runname
pfundist press.init.pfb
pfundist LW.slopex.pfb
pfundist LW.slopey.pfb
pfundist IndicatorFile_Gleeson.50z.pfb

puts "ParFlow run Complete"
```

CHAPTER

FOUR

MODEL EQUATIONS

In this chapter, we discuss the model equations used by ParFlow for its fully and variably saturated flow, overland flow, and multiphase flow and transport models. First, section *Steady-State, Saturated Groundwater Flow* describes steady-state, groundwater flow (specified by solver **IMPES**). Next, section *Richards' Equation* describes the Richards' equation model (specified by solver **RICHARDS**) for variably saturated flow as implemented in ParFlow. Section *Terrain Following Grid* describes the terrain following grid formulation. Next, the overland flow equations are presented in section *Overland Flow*. In section *Multi-Phase Flow Equations* we describe the multi-phase flow equations (specified by solver **IMPES**), and in section *Transport Equations* we describe the transport equations. Finally, section *Notation and Units* presents some notation and units and section *Water Balance* presents some basic water balance equations.

4.1 Steady-State, Saturated Groundwater Flow

Many groundwater problems are solved assuming steady-state, fully-saturated groundwater flow. This follows the form often written as:

$$\nabla \cdot \mathbf{q} = Q(x) \tag{4.1}$$

where Q is the spatially-variable source-sink term (to represent wells, etc) and $\mathbf{q}$ is the Darcy flux $[L^2T^{-1}]$ which is commonly written as:

$$\mathbf{q} = -\mathbf{K}\nabla H \tag{4.2}$$

where $\mathbf{K}$ is the saturated, hydraulic conductivity tensor $[LT^{-1}]$ and H $[L]$ is the head-potential. Inspection of (4.17) and (4.18) show that these equations agree with the above formulation for a single-phase ($i = 1$), fully-saturated ($S_i = S = 1$), problem where the mobility, λ_i, is set to the saturated hydraulic conductivity, $\mathbf{K}$, below. This is accomplished by setting the relative permeability and viscosity terms to unity in (4.19) as well as the gravity and density terms in (4.18). This is shown in the example in *Annotated Input Scripts*, but please note that the resulting solution is in pressure-head, h, not head potential, H, and will still contain a hydrostatic pressure gradient in the z direction.

4.2 Richards' Equation

The form of Richards' equation implemented in ParFlow is given as,

$$S(p)S_s\frac{\partial p}{\partial t} - \frac{\partial(S(p)\rho(p)\phi)}{\partial t} - \nabla\cdot(\mathbf{K}(p)\rho(p)(\nabla p - \rho(p)\vec{g})) = Q, \text{ in } \Omega, \tag{4.3}$$

where Ω is the flow domain, p is the pressure-head of water $[L]$, S is the water saturation, S_s is the specific storage coefficient $[L^{-1}]$, ϕ is the porosity of the medium, $\mathbf{K}(p)$ is the hydraulic conductivity tensor $[LT^{-1}]$, and Q is the water source/sink term $[L^3T^{-1}]$ (includes wells and surface fluxes). The hydraulic conductivity can be written as,

$$K(p) = \frac{\bar{k}k_r(p)}{\mu} \tag{4.4}$$

Boundary conditions can be stated as,

$$\begin{aligned} p = & \quad p_D, \text{ on } \Gamma^D, \\ -K(p)\nabla p \cdot \mathbf{n} = & \quad g_N, \text{ on } \Gamma^N, \end{aligned} \tag{4.5}$$

where $\Gamma^D \cup \Gamma^N = \partial\Omega$, $\Gamma^D \neq \emptyset$, and $\mathbf{n}$ is an outward pointing, unit, normal vector to Ω. This is the mixed form of Richards' equation. Note here that due to the constant (or passive) air phase pressure assumption, Richards' equation ignores the air phase except through its effects on the hydraulic conductivity, K. An initial condition,

$$p = p^0(x), \ t = 0, \tag{4.5}$$

completes the specification of the problem.

4.3 Terrain Following Grid

The terrain following grid formulation transforms the ParFlow grid to conform to topography [Max13]. This alters the form of Darcy's law to include a topographic slope component:

$$q_x = \mathbf{K}(p)\rho(p)(\frac{\partial p}{\partial x}\cos\theta_x + \sin\theta_x) \tag{4.6}$$

where $\theta_x = \arctan(S_0, x)$ and $\theta_y = \arctan(S_0, y)$ which are assumed to be the same as the **TopoSlope** keys assigned for overland flow, described below. The terrain following grid formulation can be very useful for coupled surface-subsurface flow problems where groundwater flow follows the topography. As cells are distributed near the ground surface and can be combined with the variable δZ capability, the number of cells in the problem can be reduced dramatically over the orthogonal formulation. For complete details on this formulation, the stencil used and the function evaluation developed, please see Maxwell [Max13]. NOTE: in the original formulation, θ_x and θ_y for a cell face is calculated as the average of the two adjacent cell slopes (i.e. assuming a cell centered slope calculation). The **TerrainFollowingGrid.SlopeUpwindFormulation** key provide options to use the slope of a grid cell directly (i.e. assuming face centered slope calculations) and removing the sine term from (4.6). The **Upwind** and **UpwindSine** options for this key will provide consistent results with **OverlandKinematic** and **OverlandDiffusive** boundary conditions while the **Original** option is consistent with the standard **OverlandFlow** boundary condition.

4.4 Flow Barriers

The the flow barrier multipliers allow for the reduction in flow across a cell face. This slightly alters Darcy's law to include a flow reduction in each direction, show here in x:

$$q_x = FB_x\mathbf{K}(p)\rho(p)(\frac{\partial p}{\partial x}\cos\theta_x + \sin\theta_x) \tag{4.7}$$

where FB_x, FB_y and FB_z are a dimensionless multipliers specified by the **FBx**, **FBy** and **FBz** keys. This creates behavior equivalent to the Hydraulic Flow Barrier (HFB) or *ITFC* (flow and transport parameters at interfaces) conditions in other models.

4.5 Overland Flow

As detailed in Kollet and Maxwell [KM06], ParFlow may simulate fully-coupled surface and subsurface flow via an overland flow boundary condition. While complete details of this approach are given in that paper, a brief summary of the equations solved are presented here. Shallow overland flow is now represented in ParFlow by the kinematic wave equation. In two spatial dimensions, the continuity equation can be written as:

$$\frac{\partial \psi_s}{\partial t} = \nabla \cdot (\vec{v}\psi_s) + q_r(x) \tag{4.8}$$

where $\vec{v}$ is the depth averaged velocity vector $[LT^{-1}]$; ψ_s is the surface ponding depth $[L]$ and $q_r(x)$ is the a general source/sink (e.g. rainfall) rate $[LT^{-1}]$. If diffusion terms are neglected the momentum equation can be written as:

$$S_{f,i} = S_{o,i} \tag{4.9}$$

which is commonly referred to as the kinematic wave approximation. In Equation (4.9) $S_{o,i}$ is the bed slope (gravity forcing term) $[-]$, which is equal to the friction slope $S_{f,i}$ $[L]$; i stands for the x- and y-direction. Manning's equation is used to establish a flow depth-discharge relationship:

$$v_x = -\frac{\sqrt{S_{f,x}}}{n}\psi_s^{2/3} \tag{4.10}$$

and

$$v_y = -\frac{\sqrt{S_{f,y}}}{n}\psi_s^{2/3} \tag{4.11}$$

where n $[TL^{-1/3}]$ is the Manning's coefficient. Though complete details of the coupled approach are given in Kollet and Maxwell [KM06], brief details of the approach are presented here. The coupled approach takes Equation eq:*kinematic* and adds a flux for subsurface exchanges, $q_e(x)$.

$$\frac{\partial \psi_s}{\partial t} = \nabla \cdot (\vec{v}\psi_s) + q_r(x) + q_e(x) \tag{4.12}$$

We then assign a continuity of pressure at the top cell of the boundary between the surface and subsurface systems by setting pressure-head, p in (4.3) equal to the vertically-averaged surface pressure, ψ_s as follows:

$$p = \psi_s = \psi \tag{4.13}$$

If we substitute this relationship back into Equation (4.12) as follows:

$$\frac{\partial \parallel \psi, 0 \parallel}{\partial t} = \nabla \cdot (\vec{v} \parallel \psi, 0 \parallel) + q_r(x) + q_e(x) \tag{4.14}$$

Where the $\parallel \psi, 0 \parallel$ operator chooses the greater of the two quantities, ψ and 0. We may now solve this term for the flux $q_e(x)$ which we may set equal to flux boundary condition shown in Equation (4.5). This yields the following equation, which is referred to as the overland flow boundary condition [KM06]:

$$-K(\psi)\nabla\psi \cdot \mathbf{n} = \frac{\partial \parallel \psi, 0 \parallel}{\partial t} - \nabla \cdot (\vec{v} \parallel \psi, 0 \parallel) - q_r(x) \tag{4.15}$$

This results a version of the kinematic wave equation that is only active when the pressure at the top cell of the subsurface domain has a ponded depth and is thus greater than zero. This method solves both systems, where active in the domain, over common grids in a fully-integrated, fully-mass conservative manner.

The depth-discharge relationship can also be written as

$$v_x = -\frac{S_{f,x}}{n\sqrt{S_f}}\psi_s^{2/3} \tag{4.16}$$

where $\overline{S_f}$ is the magnitude of the friction slope. This formulation for overland flow is used in the **OverlandKinematic** and **OverlandDiffusive** boundary conditions. In **OverlandKinematic** case the friction slope equals the bed slope following Equation (4.9). For the **OverlandDiffusive** case the friction slope also includes the pressure gradient. The solution for both of these options is formulated to do the upwinding internally and assumes that the user provides face centered bedslopes ($S_{o,i}$). This is different from the original formulation which assumes the user provides grid cenered bedslopes.

4.6 Multi-Phase Flow Equations

The flow equations are a set of *mass balance* and *momentum balance* (Darcy's Law) equations, given respectively by,

$$\frac{\partial}{\partial t}(\phi S_i) \; + \; \nabla \cdot \vec{V}_i \; - \; Q_i \; = \; 0, \tag{4.17}$$

$$\vec{V}_i \; + \; \lambda_i \cdot (\nabla p_i \; - \; \rho_i \vec{g}) \; = \; 0, \tag{4.18}$$

for $i = 0, \ldots, \nu - 1$ ($\nu \in \{1, 2, 3\}$), where

$$\begin{aligned} \lambda_i = & \quad \frac{\bar{k} k_{ri}}{\mu_i}, \\ \vec{g} = & \quad [0, 0, -g]^T, \end{aligned} \tag{4.19}$$

Table *5.1* defines the symbols in the above equations, and outlines the symbol dependencies and units.

Table 4.1: Notation and units for flow equations.

symbol	quantity	units
$\phi(\vec{x}, t)$	porosity	[]
$S_i(\vec{x}, t)$	saturation	[]
$\vec{V}_i(\vec{x}, t)$	Darcy velocity vector	$[LT^{-1}]$
$Q_i(\vec{x}, t)$	source/sink	$[T^{-1}]$
λ_i	mobility	$[L^3TM^{-1}]$
$p_i(\vec{x}, t)$	pressure	$[ML^{-1}T^{-2}]$
ρ_i	mass density	$[ML^{-3}]$
$\vec{g}$	gravity vector	$[LT^{-2}]$
$\bar{k}(\vec{x}, t)$	intrinsic permeability tensor	$[L^2]$
$k_{ri}(\vec{x}, t)$	relative permeability	[]
μ_i	viscosity	$[ML^{-1}T^{-1}]$
g	gravitational acceleration	$[LT^{-2}]$

Here, ϕ describes the fluid capacity of the porous medium, and S_i describes the content of phase i in the porous medium, where we have that $0 \leq \phi \leq 1$ and $0 \leq S_i \leq 1$. The coefficient $\bar{k}$ is considered a scalar here. We also assume that ρ_i and μ_i are constant. Also note that in ParFlow, we assume that the relative permeability is given as $k_{ri}(S_i)$. The Darcy velocity vector is related to the *velocity vector*, $\vec{v}_i$, by the following:

$$\vec{V}_i = \phi S_i \vec{v}_i. \tag{4.20}$$

To complete the formulation, we have the following ν *consititutive relations*

$$\sum_i S_i = 1, \tag{4.21}$$

$$p_{i0} \; = \; p_{i0}(S_0), \qquad i = 1, \ldots, \nu - 1. \tag{4.22}$$

where, $p_{ij} = p_i - p_j$ is the *capillary pressure* between phase i and phase j. We now have the 3ν equations, (4.17), (4.18), (4.21), and (4.22), in the 3ν unknowns, $S_i, \vec{V}_i$, and p_i.

For technical reasons, we want to rewrite the above equations. First, we define the *total mobility*, λ_T, and the *total velocity*, $\vec{V}_T$, by the relations

$$\lambda_T = \sum_i \lambda_i, \tag{4.23}$$

$$\vec{V}_T = \sum_i \vec{V}_i. \tag{4.24}$$

After doing a bunch of algebra, we get the following equation for p_0:

$$-\sum_i \{\nabla \cdot \lambda_i \left(\nabla(p_0 + p_{i0}) - \rho_i \vec{g}\right) + Q_i\} = 0. \tag{4.25}$$

After doing some more algebra, we get the following $\nu - 1$ equations for S_i:

$$\frac{\partial}{\partial t}(\phi S_i) + \nabla \cdot \left(\frac{\lambda_i}{\lambda_T}\vec{V}_T + \sum_{j \neq i} \frac{\lambda_i \lambda_j}{\lambda_T}(\rho_i - \rho_j)\vec{g} \right) + \sum_{j \neq i} \nabla \cdot \frac{\lambda_i \lambda_j}{\lambda_T} \nabla p_{ji} - Q_i = 0. \tag{4.26}$$

The capillary pressures p_{ji} in (4.26) are rewritten in terms of the constitutive relations in (4.22) so that we have

$$p_{ji} = p_{j0} - p_{i0}, \tag{4.27}$$

where by definition, $p_{ii} = 0$. Note that equations (4.26) are analytically the same equations as in (4.17). The reason we rewrite them in this latter form is because of the numerical scheme we are using. We now have the 3ν equations, (4.25), (4.26), (4.24), (4.18), and (4.22), in the 3ν unknowns, $S_i, \vec{V}_i$, and p_i.

4.7 Transport Equations

The transport equations in ParFlow are currently defined as follows:

$$\begin{aligned} &\left(\frac{\partial}{\partial t}(\phi c_{i,j}) + \lambda_j \, \phi c_{i,j} \right) + \nabla \cdot \left(c_{i,j} \vec{V}_i \right) \\ &= \\ -&\left(\frac{\partial}{\partial t}((1-\phi)\rho_s F_{i,j}) + \lambda_j \, (1-\phi)\rho_s F_{i,j} \right) + \sum_k^{n_I} \gamma_k^{I;i} \chi_{\Omega_k^I} \left(c_{i,j} - \bar{c}_{ij}^k \right) - \sum_k^{n_E} \gamma_k^{E;i} \chi_{\Omega_k^E} c_{i,j} \end{aligned}$$

where $i = 0, \ldots, \nu - 1$ ($\nu \in \{1, 2, 3\}$) is the number of phases, $j = 0, \ldots, n_c - 1$ is the number of contaminants, and where $c_{i,j}$ is the concentration of contaminant j in phase i. Recall also, that χ_A is the characteristic function of set A, i.e. $\chi_A(x) = 1$ if $x \in A$ and $\chi_A(x) = 0$ if $x \notin A$. Table *5.2* defines the symbols in the above equation, and outlines the symbol dependencies and units. The equation is basically a statement of mass conservation in a convective flow (no diffusion) with adsorption and degradation effects incorporated along with the addition of injection and extraction wells.

Table 4.2: Notation and units for transport equation.

symbol	quantity	units
$\phi(\vec{x})$	porosity	[]
$c_{i,j}(\vec{x},t)$	concentration fraction	[]
$\vec{V}_i(\vec{x},t)$	Darcy velocity vector	$[LT^{-1}]$
λ_j	degradation rate	$[T^{-1}]$
$\rho_s(\vec{x})$	density of the solid mass	$[ML^{-3}]]$
$F_{i,j}(\vec{x},t)$	mass concentration	$[L^3M^{-1}]$
n_I	number of injection wells	[]
$\gamma_k^{I;i}(t)$	injection rate	$[T^{-1}]$
$\Omega_k^I(\vec{x})$	injection well region	[]
$\bar{c}_{ij}^k()$	injected concentration fraction	[]
n_E	number of extraction wells	[]
$\gamma_k^{E;i}(t)$	extraction rate	$[T^{-1}]$
$\Omega_k^E(\vec{x})$	extraction well region	[]

These equations will soon have to be generalized to include a diffusion term. At the present time, as an adsorption model, we take the mass concentration term ($F_{i,j}$) to be instantaneous in time and a linear function of contaminant concentration :

$$F_{i,j} = K_{d;j} c_{i,j}, \tag{4.28}$$

where $K_{d;j}$ is the distribution coefficient of the component ($[L^3M^{-1}]$). If (4.28) is substituted into (4.28) the following equation results (which is the current model used in ParFlow) :

$$\begin{aligned} &(\phi + (1-\phi)\rho_s K_{d;j})\frac{\partial}{\partial t} c_{i,j} \; + \; \nabla \cdot \left(c_{i,j}\vec{V}_i \right) \\ &= \\ &-(\phi + (1-\phi)\rho_s K_{d;j})\lambda_j c_{i,j} \; + \; \sum_k^{n_I} \gamma_k^{I;i} \chi_{\Omega_k^I} \left(c_{i,j} - \bar{c}_{ij}^k \right) \; - \; \sum_k^{n_E} \gamma_k^{E;i} \chi_{\Omega_k^E} c_{i,j} \end{aligned}$$

4.8 Notation and Units

In this section, we discuss other common formulations of the flow and transport equations, and how they relate to the equations solved by ParFlow.

We can rewrite equation (4.18) as

$$\vec{V}_i \; + \; \bar{K}_i \cdot (\nabla h_i \; - \; \frac{\rho_i}{\gamma}\vec{g}) \; = \; 0, \tag{4.29}$$

where

$$\begin{aligned} \bar{K}_i &= \quad \gamma \lambda_i, \\ h_i &= \quad (p_i \; - \; \bar{p})/\gamma. \end{aligned} \tag{4.30}$$

Table *5.3* defines the symbols and their units.

Table 4.3: Notation and units for reformulated flow equations.

symbol	quantity	units
$\vec{V}_i$	Darcy velocity vector	$[LT^{-1}]$
$\bar{K}_i$	hydraulic conductivity tensor	$[LT^{-1}]$
h_i	pressure head	$[L]$
γ	constant scale factor	$[ML^{-2}T^{-2}]$
$\vec{g}$	gravity vector	$[LT^{-2}]$

We can then rewrite equations (4.25) and (4.26) as

$$-\sum_i \left\{ \nabla \cdot \bar{K}_i \left(\nabla(h_0 + h_{i0}) - \frac{\rho_i}{\gamma}\vec{g} \right) + Q_i \right\} = 0, \tag{4.31}$$

$$\frac{\partial}{\partial t}(\phi S_i) + \nabla \cdot \left(\frac{\bar{K}_i}{\bar{K}_T}\vec{V}_T + \sum_{j \neq i} \frac{\bar{K}_i \bar{K}_j}{\bar{K}_T} \left(\frac{\rho_i}{\gamma} - \frac{\rho_j}{\gamma} \right) \vec{g} \right) + \sum_{j \neq i} \nabla \cdot \frac{\bar{K}_i \bar{K}_j}{\bar{K}_T} \nabla h_{ji} - Q_i = 0. \tag{4.32}$$

Note that $\bar{K}_i$ is supposed to be a tensor, but we treat it as a scalar here. Also, note that by carefully defining the input to ParFlow, we can use the units of equations (4.31) and (4.32). To be more precise, let us denote ParFlow input symbols by appending the symbols in table *5.1* with (I), and let $\gamma = \rho_0 g$ (this is a typical definition). Then, we want:

$$\begin{aligned} \bar{k}(I) &= \gamma \bar{k}/\mu_0; \\ \mu_i(I) &= \mu_i/\mu_0; \\ p_i(I) &= h_i; \\ \rho_i(I) &= \rho_i/\rho_0; \\ g(I) &= 1. \end{aligned} \tag{4.33}$$

By doing this, $\bar{k}(I)$ represents hydraulic conductivity of the base phase $\bar{K}_0$ (e.g. water) under saturated conditions (i.e. $k_{r0} = 1$).

4.9 Water Balance

ParFlow can calculate a water balance for the Richards' equation, overland flow and `clm` capabilities. For a schematic of the water balance in ParFlow please see Maxwell [Max10]. This water balance is computes using `pftools` commands as described in *Manipulating Data: PFTools*. There are two water balance storage components, subsurface and surface, and two flux calculations, overland flow and evapotranspiration. The storage components have units $[L^3]$ while the fluxes may be instantaneous and have units $[L^3T^{-1}]$ or cumulative over an output interval with units $[L^3]$. Examples of water balance calculations and errors are given in the scripts `water_balance_x.tcl` and `water_balance_y.tcl`. The size of water balance errors depend on solver settings and tolerances but are typically very small, $< 10^{-10}$ [-]. The water balance takes the form:

$$\frac{\Delta[Vol_{subsurface} + Vol_{surface}]}{\Delta t} = Q_{overland} + Q_{evapotranspiration} + Q_{sourcesink} \tag{4.34}$$

where $Vol_{subsurface}$ is the subsurface storage $[L^3]$; $Vol_{surface}$ is the surface storage $[L^3]$; $Q_{overland}$ is the overland flux $[L^3T^{-1}]$; $Q_{evapotranspiration}$ is the evapotranspiration flux passed from `clm` or other LSM, etc, $[L^3T^{-1}]$; and $Q_{sourcesink}$ are any other source/sink fluxes specified in the simulation $[L^3T^{-1}]$. The surface and subsurface storage routines are calculated using the ParFlow toolset commands `pfsurfacestorage` and `pfsubsurfacestorage` respectively. Overland flow out of the domain is calculated by `pfsurfacerunoff`. Details for the use of these commands are given in *PFTCL Commands* and *Common examples using ParFlow TCL commands (PFTCL)*. $Q_{evapotranspiration}$ must be written out by ParFlow as a variable (as shown in *Code Parameters*) and only contains the external fluxes passed from a module such as `clm` or WRF. Note that these volume and flux quantities are calculated spatially over the domain and are returned as array values, just like any other quantity in ParFlow. The tools command `pfsum` will sum these arrays into a single value for the enrite domain. All other fluxes must be determined by the user.

The subsurface storage is calculated over all active cells in the domain, Ω, and contains both compressible and incompressible parts based on Equation (4.3). This is computed on a cell-by-cell basis (with the result being an array of balances over the domain) as follows:

$$Vol_{subsurface} = \sum_{\Omega} [S(\psi) S_s \psi \Delta x \Delta y \Delta z + S(\psi)(\psi)\phi \Delta x \Delta y \Delta z] \tag{4.35}$$

The surface storage is calculated over the upper surface boundary cells in the domain, Γ, as computed by the mask and contains based on Equation [eq:kinematic]. This is again computed on a cell-by-cell basis (with the result being an array of balances over the domain) as follows:

$$Vol_{surface} = \sum_{\Gamma} \psi \Delta x \Delta y \tag{4.36}$$

For the overland flow outflow from the domain, any cell at the top boundary that has a slope that points out of the domain and is ponded will remove water from the domain. This is calculated, for example in the y-direction, as the multiple of Equation [eq:manningsy] and the area:

$$Q_{overland} = vA = -\frac{\sqrt{S_{f,y}}}{n} \psi_s^{2/3} \psi \Delta x = -\frac{\sqrt{S_{f,y}}}{n} \psi_s^{5/3} \Delta x \tag{4.37}$$

CHAPTER

FIVE

PARFLOW FILES

In this chapter, we discuss the various file formats used in ParFlow. To help simplify the description of these formats, we use a pseudocode notation composed of *fields* and *control constructs*.

A field is a piece of data having one of the *field types* listed in Table *5.1* (note that field types may have one meaning in ASCII files and another meaning in binary files).

Table 5.1: Field types.

field type	ASCII	binary
`integer`	`integer`	XDR `integer`
`real`	`real`	
`string`	`string`	
`double`		IEEE 8 byte `double`
`float`		IEEE 4 byte `float`

Fields are denoted by enclosing the field name with a < on the left and a > on the right. The field name is composed of alphanumeric characters and underscores (`_`). In the defining entry of a field, the field name is also prepended by its field type and a `:`. The control constructs used in our pseudocode have the keyword names `FOR`, `IF`, and `LINE`, and the beginning and end of each of these constructs is delimited by the keywords `BEGIN` and `END`.

The `FOR` construct is used to describe repeated input format patterns. For example, consider the following file format:

```
<integer : num_coordinates>
FOR coordinate = 0 TO <num_coordinates> - 1
BEGIN
   <real : x>  <real : y>  <real : z>
END
```

The field <`num_coordinates`> is an integer specifying the number of coordinates to follow. The `FOR` construct indicates that <`num_coordinates`> entries follow, and each entry is composed of the three real fields, <`x`>, <`y`>, and <`z`>. Here is an example of a file with this format:

```
3
2.0 1.0 -3.5
1.0 1.1 -3.1
2.5 3.0 -3.7
```

The `IF` construct is actually an `IF/ELSE` construct, and is used to describe input format patterns that appear only under certain circumstances. For example, consider the following file format:

```
<integer : type>
IF (<type> = 0)
```

(continues on next page)

(continued from previous page)

```
BEGIN
   <real : x>  <real : y>  <real : z>
END
ELSE IF (<type> = 1)
BEGIN
   <integer : i>  <integer : j>  <integer : k>
END
```

The field `<type>` is an integer specifying the "type" of input to follow. The `IF` construct indicates that if `<type>` has value 0, then the three real fields, `<x>`, `<y>`, and `<z>`, follow. If `<type>` has value 1, then the three integer fields, `<i>`, `<j>`, and `<k>`, follow. Here is an example of a file with this format:

```
0
2.0 1.0 -3.5
```

The `LINE` construct indicates fields that are on the same line of a file. Since input files in ParFlow are all in "free format", it is used only to describe some output file formats. For example, consider the following file format:

```
LINE
BEGIN
   <real : x>
   <real : y>
   <real : z>
END
```

The `LINE` construct indicates that the three real fields, `<x>`, `<y>`, and `<z>` , are all on the same line. Here is an example of a file with this format:

```
2.0 1.0 -3.5
```

Comment lines may also appear in our file format pseudocode. All text following a `#` character is a comment, and is not part of the file format.

5.1 Main Input Files (.tcl, .py, .ipynb)

The main ParFlow input file can be a Python script, a TCL script, or a Jupyter Notebook. For more advanced users, the notebook or scripting environment provides a lot of flexibility and means you can very easily create programs to run ParFlow. A simple example is creating a loop to run several hundred different simulations using different seeds to the random field generators. This can be automated from within the ParFlow input file. The input structure for these files is given in the *ParFlow Input Keys* chapter.

5.2 ParFlow Binary Files (.pfb)

The `.pfb` file format is a binary file format which is used to store ParFlow grid data. It is written as BIG ENDIAN binary bit ordering [con]. The format for the file is:

```
<double : X>    <double : Y>    <double : Z>
<integer : NX>  <integer : NY>  <integer : NZ>
<double : DX>   <double : DY>   <double : DZ>
```

(continues on next page)

(continued from previous page)

```
<integer : num_subgrids>
FOR subgrid = 0 TO <num_subgrids> - 1
BEGIN
   <integer : ix>  <integer : iy>  <integer : iz>
   <integer : nx>  <integer : ny>  <integer : nz>
   <integer : rx>  <integer : ry>  <integer : rz>
   FOR k = iz TO iz + <nz> - 1
   BEGIN
      FOR j = iy TO iy + <ny> - 1
      BEGIN
         FOR i = ix TO ix + <nx> - 1
         BEGIN
            <double : data_ijk>
         END
      END
   END
END
```

5.3 ParFlow CLM Single Output Binary Files (.c.pfb)

The `.pfb` file format is a binary file format which is used to store `CLM` output data in a single file. It is written as BIG ENDIAN binary bit ordering [con]. The format for the file is:

```
<double : X>    <double : Y>    <double : Z>
<integer : NX>  <integer : NY>  <integer : NZ>
<double : DX>   <double : DY>   <double : DZ>

<integer : num_subgrids>
FOR subgrid = 0 TO <num_subgrids> - 1
BEGIN
   <integer : ix>  <integer : iy>  <integer : iz>
   <integer : nx>  <integer : ny>  <integer : nz>
   <integer : rx>  <integer : ry>  <integer : rz>
      FOR j = iy TO iy + <ny> - 1
      BEGIN
         FOR i = ix TO ix + <nx> - 1
         BEGIN
            eflx_lh_tot_ij
      eflx_lwrad_out_ij
      eflx_sh_tot_ij
      eflx_soil_grnd_ij
      qflx_evap_tot_ij
      qflx_evap_grnd_ij
      qflx_evap_soi_ij
      qflx_evap_veg_ij
      qflx_infl_ij
      swe_out_ij
      t_grnd_ij
     IF (clm_irr_type == 1)  qflx_qirr_ij
```

(continues on next page)

(continued from previous page)

```
ELSE IF (clm_irr_type == 3)  qflx_qirr_inst_ij
ELSE                         NULL
      FOR k = 1 TO clm_nz
      tsoil_ijk
      END
         END
      END
END
```

5.4 ParFlow Scattered Binary Files (.pfsb)

The .pfsb file format is a binary file format which is used to store ParFlow grid data. This format is used when the grid data is “scattered”, that is, when most of the data is 0. For data of this type, the .pfsb file format can reduce storage requirements considerably. The format for the file is:

```
<double : X>    <double : Y>    <double : Z>
<integer : NX>  <integer : NY>  <integer : NZ>
<double : DX>   <double : DY>   <double : DZ>

<integer : num_subgrids>
FOR subgrid = 0 TO <num_subgrids> - 1
BEGIN
   <integer : ix>  <integer : iy>  <integer : iz>
   <integer : nx>  <integer : ny>  <integer : nz>
   <integer : rx>  <integer : ry>  <integer : rz>
   <integer : num_nonzero_data>
   FOR k = iz TO iz + <nz> - 1
   BEGIN
      FOR j = iy TO iy + <ny> - 1
      BEGIN
         FOR i = ix TO ix + <nx> - 1
         BEGIN
            IF (<data_ijk> > tolerance)
            BEGIN
               <integer : i>  <integer : j>  <integer : k>
               <double : data_ijk>
            END
         END
      END
   END
END
```

5.5 ParFlow Solid Files (.pfsol)

The `.pfsol` file format is an ASCII file format which is used to define 3D solids. The solids are represented by closed triangulated surfaces, and surface "patches" may be associated with each solid.

Note that unlike the user input files, the solid file cannot contain comment lines.

The format for the file is:

```
<integer : file_version_number>

<integer : num_vertices>
# Vertices
FOR vertex = 0 TO <num_vertices> - 1
BEGIN
   <real : x>  <real : y>  <real : z>
END

# Solids
<integer : num_solids>
FOR solid = 0 TO <num_solids> - 1
BEGIN
   #Triangles
   <integer : num_triangles>
   FOR triangle = 0 TO <num_triangles> - 1
   BEGIN
      <integer : v0> <integer : v1> <integer : v2>
   END

   # Patches
   <integer : num_patches>
   FOR patch = 0 TO <num_patches> - 1
   BEGIN
      <integer : num_patch_triangles>
      FOR patch_triangle = 0 TO <num_patch_triangles> - 1
      BEGIN
         <integer : t>
      END
   END
END
```

The field `<file_version_number>` is used to make file format changes more manageable. The field `<num_vertices>` specifies the number of vertices to follow. The fields `<x>`, `<y>`, and `<z>` define the coordinate of a triangle vertex. The field `<num_solids>` specifies the number of solids to follow. The field `<num_triangles>` specifies the number of triangles to follow. The fields `<v0>`, `<v1>`, and `<v2>` are vertex indexes that specify the 3 vertices of a triangle. Note that the vertices for each triangle MUST be specified in an order that makes the normal vector point outward from the domain. The field `<num_patches>` specifies the number of surface patches to follow. The field `num_patch_triangles` specifies the number of triangles indices to follow (these triangles make up the surface patch). The field `<t>` is an index of a triangle on the solid `solid`.

ParFlow `.pfsol` files can be created from GMS `.sol` files using the utility `gmssol2pfsol` located in the `$PARFLOW_DIR/bin` directory. This conversion routine takes any number of GMS `.sol` files, concatenates the vertices of the solids defined in the files, throws away duplicate vertices, then prints out the `.pfsol` file. Information relating the solid index in the resulting `.pfsol` file with the GMS names and material IDs are printed to stdout.

5.6 ParFlow Well Output File (.wells)

A well output file is produced by ParFlow when wells are defined. The well output file contains information about the well data being used in the internal computations and accumulated statistics about the functioning of the wells.

The header section has the following format:

```
LINE
BEGIN
   <real : BackgroundX>
   <real : BackgroundY>
   <real : BackgroundZ>
   <integer : BackgroundNX>
   <integer : BackgroundNY>
   <integer : BackgroundNZ>
   <real : BackgroundDX>
   <real : BackgroundDY>
   <real : BackgroundDZ>
END

LINE
BEGIN
   <integer : number_of_phases>
   <integer : number_of_components>
   <integer : number_of_wells>
END

FOR well = 0 TO <number_of_wells> - 1
BEGIN
   LINE
   BEGIN
      <integer : sequence_number>
   END

   LINE
   BEGIN
      <string : well_name>
   END

   LINE
   BEGIN
      <real : well_x_lower>
      <real : well_y_lower>
      <real : well_z_lower>
      <real : well_x_upper>
      <real : well_y_upper>
      <real : well_z_upper>
      <real : well_diameter>
   END

   LINE
   BEGIN
     <integer : well_type>
```

(continues on next page)

(continued from previous page)

```
     <integer : well_action>
   END
END
```

The data section has the following format:

```
FOR time = 1 TO <number_of_time_intervals>
BEGIN
   LINE
   BEGIN
      <real : time>
   END

   FOR well = 0 TO <number_of_wells> - 1
   BEGIN
      LINE
      BEGIN
         <integer : sequence_number>
      END

      LINE
      BEGIN
         <integer : SubgridIX>
         <integer : SubgridIY>
         <integer : SubgridIZ>
         <integer : SubgridNX>
         <integer : SubgridNY>
         <integer : SubgridNZ>
         <integer : SubgridRX>
         <integer : SubgridRY>
         <integer : SubgridRZ>
      END

      FOR well = 0 TO <number_of_wells> - 1
      BEGIN
         LINE
         BEGIN
            FOR phase = 0 TO <number_of_phases> - 1
            BEGIN
               <real : phase_value>
            END
         END

         IF injection well
         BEGIN
            LINE
            BEGIN
               FOR phase = 0 TO <number_of_phases> - 1
               BEGIN
                  <real : saturation_value>
               END
            END
```

(continues on next page)

(continued from previous page)

```
      LINE
      BEGIN
         FOR phase = 0 TO <number_of_phases> - 1
         BEGIN
            FOR component = 0 TO <number_of_components> - 1
            BEGIN
               <real : component_value>
            END
         END
      END
   END

   LINE
   BEGIN
      FOR phase = 0 TO <number_of_phases> - 1
      BEGIN
         FOR component = 0 TO <number_of_components> - 1
         BEGIN
            <real : component_fraction>
         END
      END
   END

   LINE
   BEGIN
      FOR phase = 0 TO <number_of_phases> - 1
      BEGIN
         <real : phase_statistic>
      END
   END

   LINE
   BEGIN
      FOR phase = 0 TO <number_of_phases> - 1
      BEGIN
         <real : saturation_statistic>
      END
   END

   LINE
   BEGIN
      FOR phase = 0 TO <number_of_phases> - 1
      BEGIN
         FOR component = 0 TO <number_of_components> - 1
         BEGIN
            <real : component_statistic>
         END
      END
   END

   LINE
```

(continues on next page)

(continued from previous page)

```
        BEGIN
          FOR phase = 0 TO <number_of_phases> - 1
          BEGIN
            FOR component = 0 TO <number_of_components> - 1
            BEGIN
              <real : concentration_data>
            END
          END
        END
      END
    END
END
```

5.7 ParFlow Simple ASCII and Simple Binary Files (.sa and .sb)

The simple binary, `.sa`, file format is an ASCII file format which is used by `pftools` to write out ParFlow grid data. The simple binary, `.sb`, file format is exactly the same, just written as BIG ENDIAN binary bit ordering [con]. The format for the file is:

```
<integer : NX>  <integer : NY>  <integer : NZ>

    FOR k = 0 TO  <nz> - 1
    BEGIN
      FOR j = 0 TO  <ny> - 1
      BEGIN
        FOR i = 0 TO  <nx> - 1
        BEGIN
          <double : data_ijk>
        END
      END
    END
```

CHAPTER

SIX

PARFLOW INPUT KEYS

The basic idea behind ParFlow input is a simple database of keys. The database contains entries which have a key and a value associated with that key. When ParFlow runs, it queries the database you have created by key names to get the values you have specified.

The commands `pfset` in TCL or `<runname>.Key=` in Python are used to create the database entries. A simple ParFlow input script contains a long list of these commands that set key values. Note that the `<runname>` is the name a user gives to their run, and is a unique identifier to organize the key database and to anchor the files ParFlow writes.

It should be noted that the keys are "dynamic" in that many are built up from values of other keys. For example if you have two wells named *northwell* and *southwell* then you will have to set some keys which specify the parameters for each well. The keys are built up in a simple sort of heirarchy.

The following sections contain a description of all of the keys used by ParFlow. For an example of input files you can look at the `test` subdirectory of the ParFlow distribution. Looking over some examples should give you a good feel for how the file scripts are put together.

Each key entry has the form:

type **KeyName** default value Description

The "type" is one of integer, double, string, list. Integer and double are IEEE numbers. String is a text string (for example, a filename). Strings can contain spaces if you use the proper TCL syntax (i.e. using double quotes). These types are standard TCL types. Lists are strings but they indicate the names of a series of items. For example you might need to specify the names of the geometries. You would do this using space seperated names (what we are calling a list) "layer1 layer2 layer3".

The descriptions that follow are organized into functional areas. An example for each database entry is given.

Note that units used for each physical quantity specified in the input file must be consistent with units used for all other quantities. The exact units used can be any consistent set as ParFlow does not assume any specific set of units. However, it is up to the user to make sure all specifications are indeed consistent.

6.1 Input File Format Number

integer **FileVersion** no default This gives the value of the input file version number that this file fits.

```
pfset FileVersion 4            ## TCL syntax

<runname>.FileVersion = 4      ## Python syntax
```

As development of the ParFlow code continues, the input file format will vary. We have thus included an input file format number as a way of verifying that the correct format type is being used. The user can check in the `parflow/config/file_versions.h` file to verify that the format number specified in the input file matches the defined value of `PFIN_VERSION`.

6.2 Computing Topology

This section describes how processors are assigned in order to solve the domain in parallel. “P” allocates the number of processes to the grid-cells in x. “Q” allocates the number of processes to the grid-cells in y. “R” allocates the number of processes to the grid-cells in z. Please note “R” should always be 1 if you are running with Solver Richards [JW01] unless you’re running a totally saturated domain (solver IMPES).

integer **Process.Topology.P** no default This assigns the process splits in the *x* direction.

```
pfset Process.Topology.P        2    ## TCL syntax

<runname>.Process.Topology.P = 2    ## Python syntax
```

integer **Process.Topology.Q** no default This assigns the process splits in the *y* direction.

```
pfset Process.Topology.Q       1    ## TCL syntax

<runname>.Process.Topology.Q = 1    ## Python syntax
```

integer **Process.Topology.R** no default This assigns the process splits in the *z* direction.

```
pfset Process.Topology.R       1    ## TCL syntax

<runname>.Process.Topology.R = 1    ## Python syntax
```

In addition, you can assign the computing topology when you initiate your parflow script using tcl. You must include the topology allocation when using tclsh and the parflow script.

Example Usage (in TCL):

```
[from Terminal] tclsh default_single.tcl 2 1 1

[At the top of default_single.tcl you must include the following]
set NP  [lindex $argv 0]
set NQ  [lindex $argv 1]

pfset Process.Topology.P        $NP
pfset Process.Topology.Q        $NQ
pfset Process.Topology.R        1
```

6.3 Computational Grid

The computational grid is briefly described in *Defining the Problem*. The computational grid keys set the bottom left corner of the domain to a specific point in space. If using a `.pfsol` file, the bottom left corner location of the `.pfsol` file must be the points designated in the computational grid. The user can also assign the *x*, *y* and *z* location to correspond to a specific coordinate system (i.e. UTM).

double **ComputationalGrid.Lower.X** no default This assigns the lower *x* coordinate location for the computational grid.

```
pfset   ComputationalGrid.Lower.X  0.0       ## TCL syntax

<runname>.ComputationalGrid.Lower.X = 0.0    ## Python syntax
```

double **ComputationalGrid.Lower.Y** no default This assigns the lower *y* coordinate location for the computational grid.

```
pfset   ComputationalGrid.Lower.Y  0.0       ## TCL syntax

<runname>.ComputationalGrid.Lower.Y = 0.0    ## Python syntax
```

double **ComputationalGrid.Lower.Z** no default This assigns the lower *z* coordinate location for the computational grid.

```
pfset   ComputationalGrid.Lower.Z  0.0       ## TCL syntax

<runname>.ComputationalGrid.Lower.Z = 0.0    ## Python syntax
```

integer **ComputationalGrid.NX** no default This assigns the number of grid cells in the *x* direction for the computational grid.

```
 pfset  ComputationalGrid.NX  10        ## TCL syntax

<runname>.ComputationalGrid.NX = 10     ## Python syntax
```

integer **ComputationalGrid.NY** no default This assigns the number of grid cells in the *y* direction for the computational grid.

```
pfset  ComputationalGrid.NY  10        ## TCL syntax

<runname>.ComputationalGrid.NY = 10    ## Python syntax
```

integer **ComputationalGrid.NZ** no default This assigns the number of grid cells in the *z* direction for the computational grid.

```
pfset  ComputationalGrid.NZ  10        ## TCL syntax

<runname>.ComputationalGrid.NZ = 10    ## Python syntax
```

real **ComputationalGrid.DX** no default This defines the size of grid cells in the *x* direction. Units are *L* and are defined by the units of the hydraulic conductivity used in the problem.

```
pfset  ComputationalGrid.DX  10.0      ## TCL syntax

<runname>.ComputationalGrid.DX = 10.0  ## Python syntax
```

real **ComputationalGrid.DY** no default This defines the size of grid cells in the *y* direction. Units are *L* and are defined by the units of the hydraulic conductivity used in the problem.

```
pfset  ComputationalGrid.DY  10.0         ## TCL syntax

<runname>.ComputationalGrid.DY = 10.0     ## Python syntax
```

real **ComputationalGrid.DZ** no default This defines the size of grid cells in the *z* direction. Units are *L* and are defined by the units of the hydraulic conductivity used in the problem.

```
pfset  ComputationalGrid.DZ  1.0       ## TCL syntax

<runname>.ComputationalGrid.DZ = 1.0   ## Python syntax
```

Example Usage (TCL):

```
#-----------------------------------------------------------
# Computational Grid
#-----------------------------------------------------------
pfset ComputationalGrid.Lower.X      -10.0
pfset ComputationalGrid.Lower.Y     10.0
pfset ComputationalGrid.Lower.Z      1.0

pfset ComputationalGrid.NX           18
pfset ComputationalGrid.NY           18
pfset ComputationalGrid.NZ           8

pfset ComputationalGrid.DX           8.0
pfset ComputationalGrid.DY           10.0
pfset ComputationalGrid.DZ           1.0
```

Example Usage (Python):

```
#-----------------------------------------------------------
# Computational Grid
#-----------------------------------------------------------

<runname>.ComputationalGrid.Lower.X  = -10.0
<runname>.ComputationalGrid.Lower.Y = 10.0
<runname>.ComputationalGrid.Lower.Z  = 1.0

<runname>.ComputationalGrid.NX       = 18
<runname>.ComputationalGrid.NY       = 18
<runname>.ComputationalGrid.NZ       = 8

<runname>.ComputationalGrid.DX   = 8.0
<runname>.ComputationalGrid.DY       = 10.0
<runname>.ComputationalGrid.DZ       = 1.0
```

string **UseClustering** True Run a clustering algorithm to create boxes in index space for iteration. By default an octree representation is used for iteration, this may result in iterating over many nodes in the octree. The **UseClustering** key will run a clustering algorithm to build a set of boxes for iteration.

This does not always have a significant impact on performance and the clustering algorithm can be expensive to compute. For small problems and short running problems clustering is not recommended. Long running problems may or may not see a benefit. The result varies significantly based on the geometries in the problem.

The Berger-Rigoutsos algorithm is currently used for clustering.

```
pfset  UseClustering  False        ## TCL syntax

<runname>.UseClustering      = False     ## Python syntax
```

6.4 Geometries

Here we define all "geometrical" information needed by ParFlow. For example, the domain (and patches on the domain where boundary conditions are to be imposed), lithology or hydrostratigraphic units, faults, initial plume shapes, and so on, are considered geometries.

This input section is a little confusing. Two items are being specified, geometry inputs and geometries. A geometry input is a type of geometry input (for example a box or an input file). A geometry input can contain more than one geometry. A geometry input of type Box has a single geometry (the square box defined by the extants of the two points). A SolidFile input type can contain several geometries.

list **GeomInput.Names** no default This is a list of the geometry input names which define the containers for all of the geometries defined for this problem.

```
pfset GeomInput.Names    "solidinput indinput boxinput"     ## TCL syntax

<runname>.GeomInput.Names = "solidinput indinput boxinput"  ## Python syntax
```

string **GeomInput.*geom_input_name*.InputType** no default This defines the input type for the geometry input with *geom_input_name*. This key must be one of: **SolidFile, IndicatorField**, **Box**.

```
pfset GeomInput.solidinput.InputType  "SolidFile"        ## TCL syntax

<runname>.GeomInput.solidinput.InputType  = "SolidFile"  ## Python syntax
```

list **GeomInput.*geom_input_name*.GeomNames** no default This is a list of the names of the geometries defined by the geometry input. For a geometry input type of Box, the list should contain a single geometry name. For the SolidFile geometry type this should contain a list with the same number of gemetries as were defined using GMS. The order of geometries in the SolidFile should match the names. For IndicatorField types you need to specify the value in the input field which matches the name using GeomInput.*geom_input_name*.Value.

```
pfset GeomInput.solidinput.GeomNames "domain bottomlayer \
                                      middlelayer toplayer"  ## TCL syntax

<runname>.GeomInput.solidinput.GeomNames = "domain bottomlayer middlelayer toplayer"  ##␣
↪Python syntax
```

string **GeomInput.*geom_input_name*.Filename** no default For IndicatorField and SolidFile geometry inputs this key specifies the input filename which contains the field or solid information.

```
pfset GeomInput.solidinput.FileName   "ocwd.pfsol"       ## TCL syntax

<runname>.GeomInput.solidinput.FileName = "ocwd.pfsol"   ## Python syntax
```

integer **GeomInput.*geometry_input_name*.Value** no default For IndicatorField geometry inputs you need to specify the mapping between values in the input file and the geometry names. The named geometry will be defined whereever the input file is equal to the specifed value.

```
pfset GeomInput.sourceregion.Value   11      ## TCL syntax

<runname>.GeomInput.sourceregion.Value = 11  ## Python syntax
```

For box geometries you need to specify the location of the box. This is done by defining two corners of the the box.

double **Geom.*box_geom_name*.Lower.X** no default This gives the lower X real space coordinate value of the previously specified box geometry of name *box_geom_name*.

```
pfset Geom.background.Lower.X   -1.0          ## TCL syntax

<runname>.Geom.background.Lower.X = -1.0     ## Python syntax
```

double **Geom.*box_geom_name*.Lower.Y** no default This gives the lower Y real space coordinate value of the previously specified box geometry of name *box_geom_name*.

```
pfset Geom.background.Lower.Y   -1.0          ## TCL syntax

<runname>.Geom.background.Lower.Y = -1.0     ## Python syntax
```

double **Geom.*box_geom_name*.Lower.Z** no default This gives the lower Z real space coordinate value of the previously specified box geometry of name *box_geom_name*.

```
pfset Geom.background.Lower.Z   -1.0          ## TCL syntax

<runname>.Geom.background.Lower.Z = -1.0     ## Python syntax
```

double **Geom.*box_geom_name*.Upper.X** no default This gives the upper X real space coordinate value of the previously specified box geometry of name *box_geom_name*.

```
pfset Geom.background.Upper.X   151.0         ## TCL syntax

<runname>.Geom.background.Upper.X = 151.0    ## Python syntax
```

double **Geom.*box_geom_name*.Upper.Y** no default This gives the upper Y real space coordinate value of the previously specified box geometry of name *box_geom_name*.

```
pfset Geom.background.Upper.Y   171.0         ## TCL syntax

<runname>.Geom.background.Upper.Y = 171.0    ## Python syntax
```

double **Geom.*box_geom_name*.Upper.Z** no default This gives the upper Z real space coordinate value of the previously specified box geometry of name *box_geom_name*.

```
pfset Geom.background.Upper.Z   11.0          ## TCL syntax

<runname>.Geom.background.Upper.Z = 11.0     ## Python syntax
```

list **Geom.*geom_name*.Patches** no default Patches are defined on the surfaces of geometries. Currently you can only define patches on Box geometries and on the the first geometry in a SolidFile. For a Box the order is fixed (left right front back bottom top) but you can name the sides anything you want.

For SolidFiles the order is printed by the conversion routine that converts GMS to SolidFile format.

```
pfset Geom.background.Patches   "left right front back bottom top"      ## TCL syntax

<runname>.Geom.background.Patches = "left right front back bottom top"  ## Python syntax
```

Here is an example geometry input section which has three geometry inputs (TCL).

```
#-----------------------------------------------------------
# The Names of the GeomInputs
#-----------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
pfset GeomInput.Names                          "solidinput indinput boxinput"
#
# For a solid file geometry input type you need to specify the names
# of the gemetries and the filename
#

pfset GeomInput.solidinput.InputType      "SolidFile"

# The names of the geometries contained in the solid file. Order is
# important and defines the mapping. First geometry gets the first name.
pfset GeomInput.solidinput.GeomNames      "domain"
#
# Filename that contains the geometry
#

pfset GeomInput.solidinput.FileName       "ocwd.pfsol"

#
# An indicator field is a 3D field of values.
# The values within the field can be mapped
# to ParFlow geometries. Indicator fields must match the
# computation grid exactly!
#

pfset GeomInput.indinput.InputType                "IndicatorField"
pfset GeomInput.indinput.GeomNames        "sourceregion concenregion"
pfset GeomInput.indinput.FileName         "ocwd.pfb"

#
# Within the indicator.pfb file, assign the values to each GeomNames
#
pfset GeomInput.sourceregion.Value        11
pfset GeomInput.concenregion.Value        12

#
# A box is just a box defined by two points.
#

pfset GeomInput.boxinput.InputType        "Box"
pfset GeomInput.boxinput.GeomName "background"
pfset Geom.background.Lower.X             -1.0
pfset Geom.background.Lower.Y             -1.0
pfset Geom.background.Lower.Z             -1.0
pfset Geom.background.Upper.X             151.0
pfset Geom.background.Upper.Y             171.0
pfset Geom.background.Upper.Z             11.0

#
# The patch order is fixed in the .pfsol file, but you
# can call the patch name anything you
# want (i.e. left right front back bottom top)
#
```

(continues on next page)

(continued from previous page)

```
pfset Geom.domain.Patches                         "z-upper x-lower y-lower \
                                                         x-upper y-upper z-lower"
```

6.5 Timing Information

The data given in the timing section describe all the "temporal" information needed by ParFlow. The data items are used to describe time units for later sections, sequence iterations in time, indicate actual starting and stopping values and give instructions on when data is printed out.

double **TimingInfo.BaseUnit** no default This key is used to indicate the base unit of time for entering time values. All time should be expressed as a multiple of this value. This should be set to the smallest interval of time to be used in the problem. For example, a base unit of "1" means that all times will be integer valued. A base unit of "0.5" would allow integers and fractions of 0.5 to be used for time input values.

The rationale behind this restriction is to allow time to be discretized on some interval to enable integer arithmetic to be used when computing/comparing times. This avoids the problems associated with real value comparisons which can lead to events occurring at different timesteps on different architectures or compilers.

This value is also used when describing "time cycling data" in, currently, the well and boundary condition sections. The lengths of the cycles in those sections will be integer multiples of this value, therefore it needs to be the smallest divisor which produces an integral result for every "real time" cycle interval length needed.

```
pfset TimingInfo.BaseUnit      1.0     ## TCL syntax

<runname>.TimingInfo.BaseUnit = 1.0    ## Python syntax
```

integer **TimingInfo.StartCount** no default This key is used to indicate the time step number that will be associated with the first advection cycle in a transient problem. The value **-1** indicates that advection is not to be done. The value **0** indicates that advection should begin with the given initial conditions. Values greater than **0** are intended to mean "restart" from some previous "checkpoint" time-step, but this has not yet been implemented.

```
pfset TimingInfo.StartCount    0       ## TCL syntax

<runname>.TimingInfo.StartCount = 0    ## Python syntax
```

double **TimingInfo.StartTime** no default This key is used to indicate the starting time for the simulation.

```
pfset TimingInfo.StartTime     0.0     ## TCL syntax

<runname>.TimingInfo.StartTime = 0.0   ## Python syntax
```

double **TimingInfo.StopTime** no default This key is used to indicate the stopping time for the simulation.

```
pfset TimingInfo.StopTime      100.0      ## TCL syntax

<runname>.TimingInfo.StopTime = 100.0     ## Python syntax
```

double **TimingInfo.DumpInterval** no default This key is the real time interval at which time-dependent output should be written. A value of **0** will produce undefined behavior. If the value is negative, output will be dumped out every n time steps, where n is the absolute value of the integer part of the value.

```
pfset TimingInfo.DumpInterval  10.0       ## TCL syntax

<runname>.TimingInfo.DumpInterval = 10.0  ## Python syntax
```

integer **TimingInfo.DumpIntervalExecutionTimeLimit** 0 This key is used to indicate a wall clock time to halt the execution of a run. At the end of each dump interval the time remaining in the batch job is compared with the user supplied value, if remaining time is less than or equal to the supplied value the execution is halted. Typically used when running on batch systems with time limits to force a clean shutdown near the end of the batch job. Time units is seconds, a value of **0** (the default) disables the check.

Currently only supported on SLURM based systems, "–with-slurm" must be specified at configure time to enable.

```
pfset TimingInfo.DumpIntervalExecutionTimeLimit 360         ## TCL syntax

<runname>.TimingInfo.DumpIntervalExecutionTimeLimit = 360   ## Python syntax
```

For *Richards' equation cases only* input is collected for time step selection. Input for this section is given as follows:

list **TimeStep.Type** no default This key must be one of: **Constant** or **Growth**. The value **Constant** defines a constant time step. The value **Growth** defines a time step that starts as dt_0 and is defined for other steps as $dt^{new} = \gamma dt^{old}$ such that $dt^{new} \leq dt_{max}$ and $dt^{new} \geq dt_{min}$.

```
pfset TimeStep.Type      "Constant"      ## TCL syntax

<runname>.TimeStep.Type = "Constant"   ## Python syntax
```

double **TimeStep.Value** no default This key is used only if a constant time step is selected and indicates the value of the time step for all steps taken.

```
pfset TimeStep.Value      0.001     ## TCL syntax

<runanme>.TimeStep.Value = 0.001    ## Python syntax
```

double **TimeStep.InitialStep** no default This key specifies the initial time step dt_0 if the **Growth** type time step is selected.

```
pfset TimeStep.InitialStep    0.001       ## TCL syntax

<runname>.TimeStep.InitialStep = 0.001    ## Python syntax
```

double **TimeStep.GrowthFactor** no default This key specifies the growth factor γ by which a time step will be multiplied to get the new time step when the **Growth** type time step is selected.

```
pfset TimeStep.GrowthFactor      1.5      ## TCL syntax

<runname>.TimeStep.GrowthFactor = 1.5     ## Python syntax
```

double **TimeStep.MaxStep** no default This key specifies the maximum time step allowed, dt_{max}, when the **Growth** type time step is selected.

```
pfset TimeStep.MaxStep      86400      ## TCL syntax

<runname>.TimeStep.MaxStep = 86400     ## Python syntax
```

double **TimeStep.MinStep** no default This key specifies the minimum time step allowed, dt_{min}, when the **Growth** type time step is selected.

```
pfset TimeStep.MinStep      1.0e-3     ## TCL syntax

<runname>.TimeStep.MinStep = 1.0e-3    ## Python syntax
```

Here is a detailed example of how timing keys might be used in a simulation.

```
## TCL example

#-----------------------------------------------------------------------------
# Setup timing info [hr]
# 8760 hours in a year. Dumping files every 24 hours. Hourly timestep
#-----------------------------------------------------------------------------
pfset TimingInfo.BaseUnit           1.0
pfset TimingInfo.StartCount              0
pfset TimingInfo.StartTime                  0.0
pfset TimingInfo.StopTime           8760.0
pfset TimingInfo.DumpInterval     -24

## Timing constant example
pfset TimeStep.Type                            "Constant"
pfset TimeStep.Value                           1.0

## Timing growth example
pfset TimeStep.Type                            "Growth"
pfset TimeStep.InitialStep                  0.0001
pfset TimeStep.GrowthFactor              1.4
pfset TimeStep.MaxStep                      1.0
pfset TimeStep.MinStep                      0.0001


## Python Example

#-----------------------------------------------------------------------------
# Setup timing info [hr]
# 8760 hours in a year. Dumping files every 24 hours. Hourly timestep
#-----------------------------------------------------------------------------
<runname>.TimingInfo.BaseUnit = 1.0
<runname>.TimingInfo.StartCount = 0
<runname>.TimingInfo.StartTime = 0.0
<runname>.TimingInfo.StopTime = 8760.0
<runname>.TimingInfo.DumpInterval = -24

## Timing constant example
<runname>.TimeStep.Type   = "Constant"
<runname>.TimeStep.Value = 1.0

## Timing growth example
<runname>.TimeStep.Type   = "Growth"
<runname>.TimeStep.InitialStep = 0.0001
<runname>.TimeStep.GrowthFactor = 1.4
<runname>.TimeStep.MaxStep       = 1.0
<runname>.TimeStep.MinStep       = 0.0001
```

6.6 Time Cycles

The data given in the time cycle section describes how time intervals are created and named to be used for time-dependent boundary and well information needed by ParFlow. All the time cycles are synched to the **TimingInfo.BaseUnit** key described above and are *integer multipliers* of that value.

list **CycleNames** no default This key is used to specify the named time cycles to be used in a simulation. It is a list of names and each name defines a time cycle and the number of items determines the total number of time cycles specified. Each named cycle is described using a number of keys defined below.

```
pfset Cycle.Names "constant onoff"        ## TCL syntax

<runname>.Cycle.Names = "constant onoff"  ## Python syntax
```

list **Cycle.*cycle_name*.Names** no default This key is used to specify the named time intervals for each cycle. It is a list of names and each name defines a time interval when a specific boundary condition is applied and the number of items determines the total number of intervals in that time cycle.

```
pfset Cycle.onoff.Names "on off"          ## TCL syntax

<runname>.Cycle.onoff.Names = "on off"    ## Python syntax
```

integer **Cycle.*cycle_name.interval_name*.Length** no default This key is used to specify the length of a named time intervals. It is an *integer multiplier* of the value set for the **TimingInfo.BaseUnit** key described above. The total length of a given time cycle is the sum of all the intervals multiplied by the base unit.

```
pfset Cycle.onoff.on.Length       10     ## TCL syntax

<runname>.Cycle.onoff.on.Length = 10     ## Python syntax
```

integer **Cycle.*cycle_name*.Repeat** no default This key is used to specify the how many times a named time interval repeats. A positive value specifies a number of repeat cycles a value of -1 specifies that the cycle repeat for the entire simulation.

```
pfset Cycle.onoff.Repeat       -1

<runname>.Cycle.onoff.Repeat = -1
```

Here is a detailed example of how time cycles might be used in a simulation.

```
## TCL example

#-----------------------------------------------------------------------------
# Time Cycles
#-----------------------------------------------------------------------------
pfset Cycle.Names                                  "constant rainrec"
pfset Cycle.constant.Names                      "alltime"
pfset Cycle.constant.alltime.Length         8760
pfset Cycle.constant.Repeat                    -1

# Creating a rain and recession period for the rest of year
pfset Cycle.rainrec.Names                "rain rec"
pfset Cycle.rainrec.rain.Length         10
pfset Cycle.rainrec.rec.Length          8750
```

(continues on next page)

(continued from previous page)

```
pfset Cycle.rainrec.Repeat             -1

## Python example

#-----------------------------------------------------------------------------
# Time Cycles
#-----------------------------------------------------------------------------
<runname>.Cycle.Names = "constant rainrec"
<runname>.Cycle.constant.Names = "alltime"
<runname>.Cycle.constant.alltime.Length = 8760
<runname>.Cycle.constant.Repeat = -1

# Creating a rain and recession period for the rest of year
<runname>.Cycle.rainrec.Names     = "rain rec"
<runname>.Cycle.rainrec.rain.Length       = 10
<runname>.Cycle.rainrec.rec.Length = 8750
<runname>.Cycle.rainrec.Repeat = -1
```

6.7 Domain

The domain may be represented by any of the solid types in *Geometries* above that allow the definition of surface patches. These surface patches are used to define boundary conditions in *Boundary Conditions: Pressure* and *Boundary Conditions: Saturation* below. Subsequently, it is required that the union (or combination) of the defined surface patches equal the entire domain surface. NOTE: This requirement is NOT checked in the code.

string **Domain.GeomName** no default This key specifies which of the named geometries is the problem domain.

```
pfset Domain.GeomName    "domain"        ## TCL syntax

<runname>.Domain.GeomName = "domain"   ## Python syntax
```

6.8 Phases and Contaminants

list **Phase.Names** no default This specifies the names of phases to be modeled. Currently only 1 or 2 phases may be modeled.

```
pfset Phase.Names    "water"        ## TCL syntax

<runname>.Phase.Names = "water"     ## Python syntax
```

list **Contaminant.Names** no default This specifies the names of contaminants to be advected.

```
pfset Contaminants.Names   "tce"       ## TCL syntax

<runname>.Contaminants.Names = "tce"   ## Python syntax
```

6.9 Gravity, Phase Density and Phase Viscosity

double **Gravity** no default Specifies the gravity constant to be used.

```
pfset Gravity     1.0         ## TCL syntax

<runname>.Gravity = 1.0    ## Python syntax
```

string **Phase.*phase_name*.Density.Type** no default This key specifies whether density will be a constant value or if it will be given by an equation of state of the form $(rd)exp(cP)$, where P is pressure, rd is the density at atmospheric pressure, and c is the phase compressibility constant. This key must be either **Constant** or **EquationOfState**.

```
pfset Phase.water.Density.Type     "Constant"       ## TCL syntax

<runname>.Phase.water.Density.Type = "Constant"    ## Python syntax
```

double **Phase.*phase_name*.Density.Value** no default This specifies the value of density if this phase was specified to have a constant density value for the phase *phase_name*.

```
 pfset Phase.water.Density.Value   1.0        ## TCL syntax

<runname>.Phase.water.Density.Value = 1.0     ## Python syntax
```

double **Phase.*phase_name*.Density.ReferenceDensity** no default This key specifies the reference density if an equation of state density function is specified for the phase *phase_name*.

```
pfset Phase.water.Density.ReferenceDensity   1.0      ## TCL syntax

<runname>.Phase.water.Density.ReferenceDensity = 1.0  ## Python syntax
```

double **Phase.*phase_name*.Density.CompressibilityConstant** no default This key specifies the phase compressibility constant if an equation of state density function is specified for the phase *phase|-name*.

```
pfset Phase.water.Density.CompressibilityConstant   1.0        ## TCL syntax

<runname>.Phase.water.Density.CompressibilityConstant = 1.0    ## Python syntax
```

string **Phase.*phase_name*.Viscosity.Type** Constant This key specifies whether viscosity will be a constant value. Currently, the only choice for this key is **Constant**.

```
pfset Phase.water.Viscosity.Type   "Constant"         ## TCL syntax

<runname>.Phase.water.Viscosity.Type = "Constant"     ## Python syntax
```

double **Phase.*phase_name*.Viscosity.Value** no default This specifies the value of viscosity if this phase was specified to have a constant viscosity value.

```
pfset Phase.water.Viscosity.Value    1.0     ## TCL syntax

<runname>.Phase.water.Viscosity.Value = 1.0  ## Python syntax
```

6.10 Chemical Reactions

double **Contaminants.*contaminant_name*.Degradation.Value** no default This key specifies the half-life decay rate of the named contaminant, *contaminant_name*. At present only first order decay reactions are implemented and it is assumed that one contaminant cannot decay into another.

```
pfset Contaminants.tce.Degradation.Value        0.0       ## TCL syntax

<runname>.Contaminants.tce.Degradation.Value  = 0.0       ## Python syntax
```

6.11 Permeability

In this section, permeability property values are assigned to grid points within geometries (specified in *Geometries* above) using one of the methods described below. Permeabilities are assumed to be a diagonal tensor with entries given as,

$$\left(\begin{array}{ccc} k_x(\mathbf{x}) & 0 & 0 \\ 0 & k_y(\mathbf{x}) & 0 \\ 0 & 0 & k_z(\mathbf{x}) \end{array}\right) K(\mathbf{x}),$$

where $K(\mathbf{x})$ is the permeability field given below. Specification of the tensor entries (k_x, k_y and k_z) will be given at the end of this section.

The random field routines (*turning bands* and *pgs*) can use conditioning data if the user so desires. It is not necessary to use conditioning as ParFlow automatically defaults to not use conditioning data, but if conditioning is desired, the following key should be set:

string **Perm.Conditioning.FileName** "NA" This key specifies the name of the file that contains the conditioning data. The default string **NA** indicates that conditioning data is not applicable.

```
pfset Perm.Conditioning.FileName   "well_cond.txt"       ## TCL syntax

<runname>.Perm.Conditioning.FileName = "well_cond.txt"   ## Python syntax
```

The file that contains the conditioning data is a simple ascii file containing points and values. The format is:

```
nlines
x1 y1 z1 value1
x2 y2 z2 value2
.  .  .    .
.  .  .    .
.  .  .    .
xn yn zn valuen
```

The value of *nlines* is just the number of lines to follow in the file, which is equal to the number of data points.

The variables *xi,yi,zi* are the real space coordinates (in the units used for the given parflow run) of a point at which a fixed permeability value is to be assigned. The variable *valuei* is the actual permeability value that is known.

Note that the coordinates are not related to the grid in any way. Conditioning does not require that fixed values be on a grid. The PGS algorithm will map the given value to the closest grid point and that will be fixed. This is done for speed reasons. The conditioned turning bands algorithm does not do this; conditioning is done for every grid point using the given conditioning data at the location given. Mapping to grid points for that algorithm does not give any speedup, so there is no need to do it.

NOTE: The given values should be the actual measured values - adjustment in the conditioning for the lognormal distribution that is assumed is taken care of in the algorithms.

The general format for the permeability input is as follows:

list **Geom.Perm.Names** no default This key specifies all of the geometries to which a permeability field will be assigned. These geometries must cover the entire computational domain.

```
pfset GeomInput.Names   "background domain concen_region"      ## TCL syntax

<runname>.GeomInput.Names = "background domain concen_region"  ## Python syntax
```

string **Geom.geometry_name.Perm.Type** no default This key specifies which method is to be used to assign permeability data to the named geometry, *geometry_name*. It must be either **Constant**, **TurnBands**, **ParGuass**, or **PFBFile**. The **Constant** value indicates that a constant is to be assigned to all grid cells within a geometry. The **TurnBand** value indicates that Tompson's Turning Bands method is to be used to assign permeability data to all grid cells within a geometry [TAG89]. The **ParGauss** value indicates that a Parallel Gaussian Simulator method is to be used to assign permeability data to all grid cells within a geometry. The **PFBFile** value indicates that premeabilities are to be read from the "ParFlow Binary" file. Both the Turning Bands and Parallel Gaussian Simulators generate a random field with correlation lengths in the 3 spatial directions given by λ_x, λ_y, and λ_z with the geometric mean of the log normal field given by μ and the standard deviation of the normal field given by σ. In generating the field both of these methods can be made to stratify the data, that is follow the top or bottom surface. The generated field can also be made so that the data is normal or log normal, with or without bounds truncation. Turning Bands uses a line process, the number of lines used and the resolution of the process can be changed as well as the maximum normalized frequency $K_{\max}$ and the normalized frequency increment δK. The Parallel Gaussian Simulator uses a search neighborhood, the number of simulated points and the number of conditioning points can be changed.

```
pfset Geom.background.Perm.Type   "Constant"       ## TCL syntax

<runname>.Geom.background.Perm.Type = "Constant"   ## Python syntax
```

double **Geom.*geometry_name*.Perm.Value** no default This key specifies the value assigned to all points in the named geometry, *geometry_name*, if the type was set to constant.

```
pfset Geom.domain.Perm.Value   1.0        ## TCL syntax

<runname>.Geom.domain.Perm.Value = 1.0    ## Python syntax
```

double **Geom.*geometry_name*.Perm.LambdaX** no default This key specifies the x correlation length, λ_x, of the field generated for the named geometry, *geometry_name*, if either the Turning Bands or Parallel Gaussian Simulator are chosen.

```
pfset Geom.domain.Perm.LambdaX   200.0       ## TCL syntax

<runname>.Geom.domain.Perm.LambdaX = 200.0   ## Python syntax
```

double **Geom.*geometry_name*.Perm.LambdaY** no default This key specifies the y correlation length, λ_y, of the field generated for the named geometry, *geometry_name*, if either the Turning Bands or Parallel Gaussian Simulator are chosen.

```
pfset Geom.domain.Perm.LambdaY   200.0       ## TCL syntax

<runname>.Geom.domain.Perm.LambdaY = 200.0   ## Python syntax
```

double **Geom.*geometry_name*.Perm.LambdaZ** no default This key specifies the z correlation length, λ_z, of the field generated for the named geometry, *geometry_name*, if either the Turning Bands or Parallel Gaussian Simulator are chosen.

```
pfset Geom.domain.Perm.LambdaZ   10.0        ## TCL syntax

<runname>.Geom.domain.Perm.LambdaZ = 10.0    ## Python syntax
```

double **Geom.*geometry_name*.Perm.GeomMean** no default This key specifies the geometric mean, μ, of the log normal field generated for the named geometry, *geometry_name*, if either the Turning Bands or Parallel Gaussian Simulator are chosen.

```
pfset Geom.domain.Perm.GeomMean   4.56       ## TCL syntax

<runname>.Geom.domain.Perm.GeomMean = 4.56   ## Python syntax
```

double **Geom.*geometry_name*.Perm.Sigma** no default This key specifies the standard deviation, σ, of the normal field generated for the named geometry, *geometry_name*, if either the Turning Bands or Parallel Gaussian Simulator are chosen.

```
pfset Geom.domain.Perm.Sigma   2.08       ## TCL syntax

<runname>.Geom.domain.Perm.Sigma = 2.08   ## Python syntax
```

integer **Geom.*geometry_name*.Perm.Seed** 1 This key specifies the initial seed for the random number generator used to generate the field for the named geometry, *geometry_name*, if either the Turning Bands or Parallel Gaussian Simulator are chosen. This number must be positive.

```
pfset Geom.domain.Perm.Seed   1        ## TCL syntax

<runname>.Geom.domain.Perm.Seed = 1    ## Python syntax
```

integer **Geom.*geometry_name*.Perm.NumLines** 100 This key specifies the number of lines to be used in the Turning Bands algorithm for the named geometry, *geometry_name*.

```
pfset Geom.domain.Perm.NumLines   100        ## TCL syntax

<runname>.Geom.domain.Perm.NumLines = 100    ## Python syntax
```

double **Geom.*geometry_name*.Perm.RZeta** 5.0 This key specifies the resolution of the line processes, in terms of the minimum grid spacing, to be used in the Turning Bands algorithm for the named geometry, *geometry_name*. Large values imply high resolution.

```
pfset Geom.domain.Perm.RZeta   5.0        ## TCL syntax

<runname>.Geom.domain.Perm.RZeta = 5.0    ## Python syntax
```

double **Geom.*geometry_name*.Perm.KMax** 100.0 This key specifies the the maximum normalized frequency, $K_{\max}$, to be used in the Turning Bands algorithm for the named geometry, *geometry_name*.

```
pfset Geom.domain.Perm.KMax   100.0       ## TCL syntax

<runname>.Geom.domain.Perm.KMax = 100.0   ## Python syntax
```

double **Geom.*geometry_name*.Perm.DelK** 0.2 This key specifies the normalized frequency increment, δK, to be used in the Turning Bands algorithm for the named geometry, *geometry_name*.

```
pfset Geom.domain.Perm.DelK   0.2          ## TCL syntax

<runname>.Geom.domain.Perm.DelK = 0.2      ## Python syntax
```

integer **Geom.*geometry_name*.Perm.MaxNPts** no default This key sets limits on the number of simulated points in the search neighborhood to be used in the Parallel Gaussian Simulator for the named geometry, *geometry_name*.

```
pfset Geom.domain.Perm.MaxNPts   5         ## TCL syntax

<runname>.Geom.domain.Perm.MaxNPts = 5     ## Python syntax
```

integer **Geom.*geometry_name*.Perm.MaxCpts** no default This key sets limits on the number of external conditioning points in the search neighborhood to be used in the Parallel Gaussian Simulator for the named geometry, *geometry_name*.

```
pfset Geom.domain.Perm.MaxCpts   200       ## TCL syntax

<runname>.Geom.domain.Perm.MaxCpts = 200   ## Python syntax
```

string **Geom.*geometry_name*.Perm.LogNormal** "LogTruncated" The key specifies when a normal, log normal, truncated normal or truncated log normal field is to be generated by the method for the named geometry, *geometry_name*. This value must be one of **Normal**, **Log**, **NormalTruncated** or **LogTruncate** and can be used with either Turning Bands or the Parallel Gaussian Simulator.

```
pfset Geom.domain.Perm.LogNormal   "LogTruncated"        ## TCL syntax

<runname>.Geom.domain.Perm.LogNormal = "LogTruncated"    ## Python syntax
```

string **Geom.*geometry_name*.Perm.StratType** "Bottom" This key specifies the stratification of the permeability field generated by the method for the named geometry, *geometry_name*. The value must be one of **Horizontal**, **Bottom** or **Top** and can be used with either the Turning Bands or the Parallel Gaussian Simulator.

```
pfset Geom.domain.Perm.StratType  "Bottom"         ## TCL syntax

<runname>.Geom.domain.Perm.StratType = "Bottom"    ## Python syntax
```

double **Geom.*geometry_name*.Perm.LowCutoff** no default This key specifies the low cutoff value for truncating the generated field for the named geometry, *geometry_name*, when either the NormalTruncated or LogTruncated values are chosen.

```
pfset Geom.domain.Perm.LowCutoff   0.0       ## TCL syntax

<runname>.Geom.domain.Perm.LowCutoff = 0.0   ## Python syntax
```

double **Geom.*geometry_name*.Perm.HighCutoff** no default This key specifies the high cutoff value for truncating the generated field for the named geometry, *geometry_name*, when either the NormalTruncated or LogTruncated values are chosen.

```
pfset Geom.domain.Perm.HighCutoff   100.0       ## TCL syntax

<runname>.Geom.domain.Perm.HighCutoff = 100.0   ## Python syntax
```

string **Geom.*geometry_name*.Perm.FileName** no default This key specifies that permeability values for the specified geometry, *geometry_name*, are given according to a user-supplied description in the "ParFlow Binary" file whose filename is given as the value. For a description of the ParFlow Binary file format, see *ParFlow Binary Files (.pfb)*.

The ParFlow Binary file associated with the named geometry must contain a collection of permeability values corresponding in a one-to-one manner to the entire computational grid. That is to say, when the contents of the file are read into the simulator, a complete permeability description for the entire domain is supplied. Only those values associated with computational cells residing within the geometry (as it is represented on the computational grid) will be copied into data structures used during the course of a simulation. Thus, the values associated with cells outside of the geounit are irrelevant. For clarity, consider a couple of different scenarios. For example, the user may create a file for each geometry such that appropriate permeability values are given for the geometry and “garbage” values (e.g., some flag value) are given for the rest of the computational domain. In this case, a separate binary file is specified for each geometry. Alternatively, one may place all values representing the permeability field on the union of the geometries into a single binary file. Note that the permeability values must be represented in precisely the same configuration as the computational grid. Then, the same file could be specified for each geounit in the input file. Or, the computational domain could be described as a single geouint (in the ParFlow input file) in which case the permeability values would be read in only once.

```
pfset Geom.domain.Perm.FileName "domain_perm.pfb"        ## TCL syntax

<runname>.Geom.domain.Perm.FileName = "domain_perm.pfb"  ## Python syntax
```

string **Perm.TensorType** no default This key specifies whether the permeability tensor entries k_x, k_y and k_z will be specified as three constants within a set of regions covering the domain or whether the entries will be specified cell-wise by files. The choices for this key are **TensorByGeom** and **TensorByFile**.

```
pfset Perm.TensorType     "TensorByGeom"     ## TCL syntax

<runname>.Perm.TensorType = "TensorByGeom"   ## Python syntax
```

string **Geom.Perm.TensorByGeom.Names** no default This key specifies all of the geometries to which permeability tensor entries will be assigned. These geometries must cover the entire computational domain.

```
pfset Geom.Perm.TensorByGeom.Names   "background domain"       ## TCL syntax

<runname>.Geom.Perm.TensorByGeom.Names = "background domain"   ## Python syntax
```

double **Geom.*geometry_name*.Perm.TensorValX** no default This key specifies the value of k_x for the geometry given by *geometry_name*.

```
pfset Geom.domain.Perm.TensorValX   1.0          ## TCL syntax

<runname>.Geom.domain.Perm.TensorValX = 1.0      ## Python syntax
```

double **Geom.*geometry_name*.Perm.TensorValY** no default This key specifies the value of k_y for the geometry given by *geom_name*.

```
pfset Geom.domain.Perm.TensorValY   1.0          ## TCL syntax

<runname>.Geom.domain.Perm.TensorValY = 1.0      ## Python syntax
```

double **Geom.*geometry_name*.Perm.TensorValZ** no default This key specifies the value of k_z for the geometry given by *geom_name*.

```
pfset Geom.domain.Perm.TensorValZ   1.0      ## TCL syntax

<runname>.Geom.domain.Perm.TensorValZ = 1.0  ## Python syntax
```

string **Geom.*geometry_name*.Perm.TensorFileX** no default This key specifies that k_x values for the specified geometry, *geometry_name*, are given according to a user-supplied description in the "ParFlow Binary" file whose filename is given as the value. The only choice for the value of *geometry_name* is "domain".

```
pfset Geom.domain.Perm.TensorByFileX   "perm_x.pfb"         ## TCL syntax

<runname>.Geom.domain.Perm.TensorByFileX = "perm_x.pfb"    ## Python syntax
```

string **Geom.*geometry_name*.Perm.TensorFileY** no default This key specifies that k_y values for the specified geometry, *geometry_name*, are given according to a user-supplied description in the "ParFlow Binary" file whose filename is given as the value. The only choice for the value of *geometry_name* is "domain".

```
pfset Geom.domain.Perm.TensorByFileY   "perm_y.pfb"         ## TCL syntax

<runname>.Geom.domain.Perm.TensorByFileY = "perm_y.pfb"     ## Python syntax
```

string **Geom.*geometry_name*.Perm.TensorFileZ** no default This key specifies that k_z values for the specified geometry, *geometry_name*, are given according to a user-supplied description in the "ParFlow Binary" file whose filename is given as the value. The only choice for the value of *geometry_name* is "domain".

```
pfset Geom.domain.Perm.TensorByFileZ   "perm_z.pfb"         ## TCL syntax

<runname>.Geom.domain.Perm.TensorByFileZ = "perm_z.pfb"     ## Python syntax
```

6.12 Porosity

Here, porosity values are assigned within geounits (specified in *Geometries* above) using one of the methods described below.

The format for this section of input is:

list **Geom.Porosity.GeomNames** no default This key specifies all of the geometries on which a porosity will be assigned. These geometries must cover the entire computational domain.

```
pfset Geom.Porosity.GeomNames   "background"          ## TCL syntax

<runname>.Geom.Porosity.GeomNames = "background"      ## Python syntax
```

string **Geom.*geometry_name*.Porosity.Type** no default This key specifies which method is to be used to assign porosity data to the named geometry, *geometry_name*. The only choice currently available is **Constant** which indicates that a constant is to be assigned to all grid cells within a geometry.

```
pfset Geom.background.Porosity.Type   "Constant"          ## TCL syntax

<runname>.Geom.background.Porosity.Type = "Constant"     ## Python syntax
```

double **Geom.*geometry_name*.Porosity.Value** no default This key specifies the value assigned to all points in the named geometry, *geometry_name*, if the type was set to constant.

```
pfset Geom.domain.Porosity.Value   1.0       ## TCL syntax

<runname>.Geom.domain.Porosity.Value = 1.0   ## Python syntax
```

6.13 Specific Storage

Here, specific storage (S_s in Equation (4.3)) values are assigned within geounits (specified in *Geometries* above) using one of the methods described below.

The format for this section of input is:

list **Specific Storage.GeomNames** no default This key specifies all of the geometries on which a different specific storage value will be assigned. These geometries must cover the entire computational domain.

```
pfset SpecificStorage.GeomNames       "domain"     ## TCL syntax

<runname>.SpecificStorage.GeomNames = "domain"     ## Python syntax
```

string **SpecificStorage.Type** no default This key specifies which method is to be used to assign specific storage data. The only choice currently available is **Constant** which indicates that a constant is to be assigned to all grid cells within a geometry.

```
pfset SpecificStorage.Type        "Constant"       ## TCL syntax

<runname>.SpecificStorage.Type = "Constant"        ## Python syntax
```

double **Geom.*geometry_name*.SpecificStorage.Value** no default This key specifies the value assigned to all points in the named geometry, *geometry_name*, if the type was set to constant.

```
pfset Geom.domain.SpecificStorage.Value 1.0e-4        ## TCL syntax

<runname>.Geom.domain.SpecificStorage.Value = 1.0e-4  ## Python syntax
```

6.14 dZMultipliers

Here, dZ multipliers ($\delta Z * m$) values are assigned within geounits (specified in *Geometries* above) using one of the methods described below.

The format for this section of input is:

string **Solver.Nonlinear.VariableDz** False This key specifies whether dZ multipliers are to be used, the default is False. The default indicates a false or non-active variable dz and each layer thickness is 1.0 [L].

```
pfset Solver.Nonlinear.VariableDz     True      ## TCL syntax

<runnname>.Solver.Nonlinear.VariableDz = True   ## Python syntax
```

list **dzScale.GeomNames** no default This key specifies which problem domain is being applied a variable dz subsurface. These geometries must cover the entire computational domain.

```
pfset dzScale.GeomNames "domain"          ## TCL syntax

<runname>.dzScale.GeomNames = "domain"    ## Python syntax
```

string **dzScale.Type** no default This key specifies which method is to be used to assign variable vertical grid spacing. The choices currently available are **Constant** which indicates that a constant is to be assigned to all grid cells within a geometry, **nzList** which assigns all layers of a given model to a list value, and **PFBFile** which reads in values from a distributed pfb file.

```
pfset dzScale.Type        "Constant"        ## TCL syntax

<runname>.dzScale.Type = "Constant"        ## Python syntax
```

list **Specific dzScale.GeomNames** no default This key specifies all of the geometries on which a different dz scaling value will be assigned. These geometries must cover the entire computational domain.

```
pfset dzScale.GeomNames        "domain"    ## TCL syntax

<runname>.dzScale.GeomNames = "domain"    ## Python syntax
```

double **Geom.*geometry_name*.dzScale.Value** no default This key specifies the value assigned to all points in the named geometry, *geometry_name*, if the type was set to constant.

```
pfset Geom.domain.dzScale.Value 1.0          ## TCL syntax

<runname>.Geom.domain.dzScale.Value = 1.0    ## Python syntax
```

string **Geom.*geometry_name*.dzScale.FileName** no default This key specifies file to be read in for variable dz values for the given geometry, *geometry_name*, if the type was set to **PFBFile**.

```
pfset Geom.domain.dzScale.FileName       "vardz.pfb"       ## TCL syntax

<runname>.Geom.domain.dzScale.FileName = "vardz.pfb"       ## Python syntax
```

integer **dzScale.nzListNumber** no default This key indicates the number of layers with variable dz in the subsurface. This value is the same as the *ComputationalGrid.NZ* key.

```
pfset dzScale.nzListNumber  10         ## TCL syntax

<runname>.dzScale.nzListNumber = 10    ## Python syntax
```

double **Cell.*nzListNumber*.dzScale.Value** no default This key assigns the thickness of each layer defined by nzListNumber. ParFlow assigns the layers from the bottom-up (i.e. the bottom of the domain is layer 0, the top is layer NZ-1). The total domain depth (*Geom.domain.Upper.Z*) does not change with variable dz. The layer thickness is calculated by *ComputationalGrid.DZ *dZScale*. *Note that* in Python a number is not an allowed character for a variable. Thus we proceed the layer number with an underscore "_" as shown in the example below.

```
pfset Cell.0.dzScale.Value 1.0         ## TCL syntax

<runname>.Cell._0.dzScale.Value = 1.0  ## Python syntax
```

Example Usage (TCL):

```
#---------------------------------------------
# Variable dz Assignments
#--------------------------------------------
# Set VariableDz to be true
# Indicate number of layers (nzlistnumber), which is the same as nz
# (1) There is nz*dz = total depth to allocate,
# (2) Each layer's thickness is dz*dzScale, and
# (3) Assign the layer thickness from the bottom up.
# In this example nz = 5; dz = 10; total depth 40;
# Layers  Thickness [m]
```

(continues on next page)

(continued from previous page)

```
# 0                 15                       Bottom layer
# 1                 15
# 2                 5
# 3                 4.5
# 4                 0.5                      Top layer
pfset Solver.Nonlinear.VariableDz     True
pfset dzScale.GeomNames            "domain"
pfset dzScale.Type            "nzList"
pfset dzScale.nzListNumber       5
pfset Cell.0.dzScale.Value 1.5
pfset Cell.1.dzScale.Value 1.5
pfset Cell.2.dzScale.Value 0.5
pfset Cell.3.dzScale.Value 0.45
pfset Cell.4.dzScale.Value 0.05
```

Example Usage (Python):

```
#--------------------------------------------
# Variable dz Assignments
#------------------------------------------
# Set VariableDz to be true
# Indicate number of layers (nzlistnumber), which is the same as nz
# (1) There is nz*dz = total depth to allocate,
# (2) Each layer's thickness is dz*dzScale, and
# (3) Assign the layer thickness from the bottom up.
# In this example nz = 5; dz = 10; total depth 40;
# Layers  Thickness [m]
# 0                 15                       Bottom layer
# 1                 15
# 2                 5
# 3                 4.5
# 4                 0.5                      Top layer
<runname>.Solver.Nonlinear.VariableDz = True
<runname>.dzScale.GeomNames = "domain"
<runname>.dzScale.Type = "nzList"
<runname>.dzScale.nzListNumber = 5
<runname>.Cell._0.dzScale.Value = 1.5
<runname>.Cell._1.dzScale.Value = 1.5
<runname>.Cell._2.dzScale.Value = 0.5
<runname>.Cell._3.dzScale.Value = 0.45
<runname>.Cell._4.dzScale.Value = 0.05
```

6.15 Flow Barriers

Here, the values for Flow Barriers described in _FB can be input. These are only available with Solver **Richards** and can be specified in X, Y or Z directions independently using PFB files. These barriers are appied at the cell face at the location $i + 1/2$. That is a value of FB_x specified at i will be applied to the cell face at $i + 1/2$ or between cells i and $i + 1$. The same goes for FB_y $(j + 1/2)$ and FB_z $(k + 1/2)$. The flow barrier values are unitless and mulitply the flux equation as shown in (4.7).

The format for this section of input is:

string **Solver.Nonlinear.FlowBarrierX** False This key specifies whether Flow Barriers are to be used in the X direction, the default is False. The default indicates a false or FB_x value of one [-] everywhere in the domain.

```
pfset Solver.Nonlinear.FlowBarrierX       True     ## TCL syntax

<runname>.Solver.Nonlinear.FlowBarrierX = True     ## Python syntax
```

string **Solver.Nonlinear.FlowBarrierY** False This key specifies whether Flow Barriers are to be used in the Y direction, the default is False. The default indicates a false or FB_y value of one [-] everywhere in the domain.

```
pfset Solver.Nonlinear.FlowBarrierY       True      ## TCL syntax

<runname>.Solver.Nonlinear.FlowBarrierY = True      ## Python syntax
```

string **Solver.Nonlinear.FlowBarrierZ** False This key specifies whether Flow Barriers are to be used in the Z direction, the default is False. The default indicates a false or FB_z value of one [-] everywhere in the domain.

```
pfset Solver.Nonlinear.FlowBarrierZ       True     ## TCL syntax

<runname>.Solver.Nonlinear.FlowBarrierZ = True     ## Python syntax
```

string **FBx.Type** no default This key specifies which method is to be used to assign flow barriers in X. The only choice currently available is **PFBFile** which reads in values from a distributed pfb file.

```
pfset FBx.Type        "PFBFile"      ## TCL syntax

<runname>.FBx.Type = "PFBFile"      ## Python syntax
```

string **FBy.Type** no default This key specifies which method is to be used to assign flow barriers in Y. The only choice currently available is **PFBFile** which reads in values from a distributed pfb file.

```
pfset FBy.Type        "PFBFile"    ## TCL syntax

<runname>.FBy.Type = "PFBFile"    ## Python syntax
```

string **FBz.Type** no default This key specifies which method is to be used to assign flow barriers in Z. The only choice currently available is **PFBFile** which reads in values from a distributed pfb file.

```
pfset FBz.Type        "PFBFile"    ## TCL syntax

<runname>.FBz.Type = "PFBFile"    ## Python syntax
```

The Flow Barrier values may be read in from a PFB file over the entire domain. This is done as follows:

string **Geom.domain.FBx.FileName** no default This key specifies file to be read in for the X flow barrier values for the domain, if the type was set to **PFBFile**.

```
pfset Geom.domain.FBx.FileName       "Flow_Barrier_X.pfb"      ## TCL syntax

<runname>.Geom.domain.FBx.FileName = "Flow_Barrier_X.pfb"      ## Python syntax
```

string **Geom.domain.FBy.FileName** no default This key specifies file to be read in for the Y flow barrier values for the domain, if the type was set to **PFBFile**.

```
pfset Geom.domain.FBy.FileName      "Flow_Barrier_Y.pfb"     ## TCL syntax

<runname>.Geom.domain.FBy.FileName = "Flow_Barrier_Y.pfb"    ## Python syntax
```

string **Geom.domain.FBz.FileName** no default This key specifies file to be read in for the Z flow barrier values for the domain, if the type was set to **PFBFile**.

```
pfset Geom.domain.FBz.FileName  "Flow_Barrier_Z.pfb"        ## TCL syntax

<runname>.Geom.domain.FBz.FileName = "Flow_Barrier_Z.pfb"   ## Python syntax
```

6.16 Manning's Roughness Values

Here, Manning's roughness values (n in Equations (4.10) and (4.11)) are assigned to the upper boundary of the domain using one of the methods described below.

The format for this section of input is:

list **Mannings.GeomNames** no default This key specifies all of the geometries on which a different Mannings roughness value will be assigned. Mannings values may be assigned by **PFBFile** or as **Constant** by geometry. These geometries must cover the entire upper surface of the computational domain.

```
pfset Mannings.GeomNames       "domain"    ## TCL syntax

<runname>.Mannings.GeomNames = "domain"    ## Python syntax
```

string **Mannings.Type** no default This key specifies which method is to be used to assign Mannings roughness data. The choices currently available are **Constant** which indicates that a constant is to be assigned to all grid cells within a geometry and **PFBFile** which indicates that all values are read in from a distributed, grid-based ParFlow binary file.

```
pfset Mannings.Type     "Constant"     ## TCL syntax

<runname>.Mannings.Type = "Constant"   ## Python syntax
```

double **Mannings.Geom.*geometry_name*.Value** no default This key specifies the value assigned to all points in the named geometry, *geometry_name*, if the type was set to constant.

```
pfset Mannings.Geom.domain.Value 5.52e-6        ## TCL syntax

<runname>.Mannings.Geom.domain.Value = 5.52e-6  ## Python syntax
```

double **Mannings.FileName** no default This key specifies the value assigned to all points be read in from a ParFlow binary file.

```
pfset Mannings.FileName "roughness.pfb"         ## TCL syntax

<runname>.Mannings.FileName = "roughness.pfb"   ## Python syntax
```

Complete example of setting Mannings roughness n values by geometry:

```
## TCL example
pfset Mannings.Type "Constant"
pfset Mannings.GeomNames "domain"
pfset Mannings.Geom.domain.Value 5.52e-6


## Python example
<runname>.Mannings.Type = "Constant"
<runname>.Mannings.GeomNames = "domain"
<runname>.Mannings.Geom.domain.Value = 5.52e-6
```

6.17 Topographical Slopes

Here, topographical slope values ($S_{f,x}$ and $S_{f,y}$ in Equations (4.10) and (4.11)) are assigned to the upper boundary of the domain using one of the methods described below. Note that due to the negative sign in these equations $S_{f,x}$ and $S_{f,y}$ take a sign in the direction *opposite* of the direction of the slope. That is, negative slopes point "downhill" and positive slopes "uphill".

The format for this section of input is:

list **TopoSlopesX.GeomNames** no default This key specifies all of the geometries on which a different x topographic slope values will be assigned. Topographic slopes may be assigned by **PFBFile** or as **Constant** by geometry. These geometries must cover the entire upper surface of the computational domain.

```
pfset TopoSlopesX.GeomNames       "domain"      ## TCL syntax

<runname>.TopoSlopesX.GeomNames = "domain"      ## Python syntax
```

list **TopoSlopesY.GeomNames** no default This key specifies all of the geometries on which a different y topographic slope values will be assigned. Topographic slopes may be assigned by **PFBFile** or as **Constant** by geometry. These geometries must cover the entire upper surface of the computational domain.

```
pfset TopoSlopesY.GeomNames       "domain"      ## TCL syntax

<runname>.TopoSlopesY.GeomNames = "domain"      ## Python syntax
```

string **TopoSlopesX.Type** no default This key specifies which method is to be used to assign topographic slopes. The choices currently available are **Constant** which indicates that a constant is to be assigned to all grid cells within a geometry and **PFBFile** which indicates that all values are read in from a distributed, grid-based ParFlow binary file.

```
pfset TopoSlopesX.Type "Constant"         ## TCL syntax

<runname>.TopoSlopesX.Type = "Constant"   ## Python syntax
```

double **TopoSlopeX.Geom.*geometry_name*.Value** no default This key specifies the value assigned to all points in the named geometry, *geometry_name*, if the type was set to constant.

```
pfset ToposlopeX.Geom.domain.Value       0.001     ## TCL syntax

<runname>.ToposlopeX.Geom.domain.Value = 0.001     ## Python syntax
```

double **ToposlopesX.FileName** no default This key specifies the value assigned to all points be read in from a ParFlow binary file.

```
pfset TopoSlopesX.FileName       "lw.1km.slope_x.pfb"    ## TCL syntax

<runname>.TopoSlopesX.FileName = "lw.1km.slope_x.pfb"    ## Python syntax
```

double **ToposlopesY.FileName** no default This key specifies the value assigned to all points be read in from a ParFlow binary file.

```
pfset TopoSlopesY.FileName       "lw.1km.slope_y.pfb"    ## TCL syntax

<runname>.TopoSlopesY.FileName = "lw.1km.slope_y.pfb"    ## Python syntax
```

Example of setting x and y slopes by geometry:

```
pfset TopoSlopesX.Type "Constant"
pfset TopoSlopesX.GeomNames "domain"
pfset TopoSlopesX.Geom.domain.Value 0.001

pfset TopoSlopesY.Type "Constant"
pfset TopoSlopesY.GeomNames "domain"
pfset TopoSlopesY.Geom.domain.Value -0.001
```

Example of setting x and y slopes by file:

```
pfset TopoSlopesX.Type "PFBFile"
pfset TopoSlopesX.GeomNames "domain"
pfset TopoSlopesX.FileName "lw.1km.slope_x.pfb"

pfset TopoSlopesY.Type "PFBFile"
pfset TopoSlopesY.GeomNames "domain"
pfset TopoSlopesY.FileName "lw.1km.slope_y.pfb"
```

6.18 Retardation

Here, retardation values are assigned for contaminants within geounits (specified in *Geometries* above) using one of the functions described below. The format for this section of input is:

list **Geom.Retardation.GeomNames** no default This key specifies all of the geometries to which the contaminants will have a retardation function applied.

```
pfset GeomInput.Names       "background"     ## TCL syntax

<runname>.GeomInput.Names = "background"     ## Python syntax
```

string **Geom.*geometry_name*.*contaminant_name*.Retardation.Type** no default This key specifies which function is to be used to compute the retardation for the named contaminant, *contaminant_name*, in the named geometry,

geometry_name. The only choice currently available is **Linear** which indicates that a simple linear retardation function is to be used to compute the retardation.

```
pfset Geom.background.tce.Retardation.Type   "Linear"       ## TCL syntax

<runname>.Geom.background.tce.Retardation.Type = "Linear"   ## Python syntax
```

double **Geom.*geometry_name*.*contaminant_name*.Retardation.Value** no default This key specifies the distribution coefficient for the linear function used to compute the retardation of the named contaminant, *contaminant_name*, in the named geometry, *geometry_name*. The value should be scaled by the density of the material in the geometry.

```
pfset Geom.domain.Retardation.Value   0.2          ## TCL syntax

<runname>.Geom.domain.Retardation.Value = 0.2      ## Python syntax
```

6.19 Full Multiphase Mobilities

Here we define phase mobilities by specifying the relative permeability function. Input is specified differently depending on what problem is being specified. For full multi-phase problems, the following input keys are used. See the next section for the correct Richards' equation input format.

string **Phase.*phase_name*.Mobility.Type** no default This key specifies whether the mobility for *phase_name* will be a given constant or a polynomial of the form, $(S - S_0)^a$, where S is saturation, S_0 is irreducible saturation, and a is some exponent. The possibilities for this key are **Constant** and **Polynomial**.

```
pfset Phase.water.Mobility.Type   "Constant"       ## TCL syntax

<runname>.Phase.water.Mobility.Type = "Constant"   ## Python syntax
```

double **Phase.*phase_name*.Mobility.Value** no default This key specifies the constant mobility value for phase *phase_name*.

```
pfset Phase.water.Mobility.Value   1.0       ## TCL syntax

<runname>.Phase.water.Mobility.Value = 1.0   ## Python syntax
```

double **Phase.*phase_name*.Mobility.Exponent** 2.0 This key specifies the exponent used in a polynomial representation of the relative permeability. Currently, only a value of 2.0 is allowed for this key.

```
pfset Phase.water.Mobility.Exponent   2.0          ## TCL syntax

<runname>.Phase.water.Mobility.Exponent = 2.0      ## Python syntax
```

double **Phase.*phase_name*.Mobility.IrreducibleSaturation** 0.0 This key specifies the irreducible saturation used in a polynomial representation of the relative permeability. Currently, only a value of 0.0 is allowed for this key.

```
pfset Phase.water.Mobility.IrreducibleSaturation   0.0      ## TCL syntax

<runname>.Phase.water.Mobility.IrreducibleSaturation = 0.0  ## Python syntax
```

6.20 Richards' Equation Relative Permeabilities

The following keys are used to describe relative permeability input for the Richards' equation implementation. They will be ignored if a full two-phase formulation is used.

string **Phase.RelPerm.Type** no default This key specifies the type of relative permeability function that will be used on all specified geometries. Note that only one type of relative permeability may be used for the entire problem. However, parameters may be different for that type in different geometries. For instance, if the problem consists of three geometries, then **VanGenuchten** may be specified with three different sets of parameters for the three different goemetries. However, once **VanGenuchten** is specified, one geometry cannot later be specified to have **Data** as its relative permeability. The possible values for this key are **Constant, VanGenuchten, Haverkamp, Data,** and **Polynomial**.

```
pfset Phase.RelPerm.Type   "Constant"        ## TCL syntax

<runname>.Phase.RelPerm.Type = "Constant"    ## Python syntax
```

The various possible functions are defined as follows. The **Constant** specification means that the relative permeability will be constant on the specified geounit. The **VanGenuchten** specification means that the relative permeability will be given as a Van Genuchten function [VG80] with the form,

$$k_r(p) = \frac{(1 - \frac{(\alpha p)^{n-1}}{(1+(\alpha p)^n)^m})^2}{(1 + (\alpha p)^n)^{m/2}},$$

where α and n are soil parameters and $m = 1 - 1/n$, on each region. The **Haverkamp** specification means that the relative permeability will be given in the following form [HV81],

$$k_r(p) = \frac{A}{A + p^\gamma},$$

where A and γ are soil parameters, on each region. The **Data** specification is currently unsupported but will later mean that data points for the relative permeability curve will be given and ParFlow will set up the proper interpolation coefficients to get values between the given data points. The **Polynomial** specification defines a polynomial relative permeability function for each region of the form,

$$k_r(p) = \sum_{i=0}^{degree} c_i p^i.$$

list **Phase.RelPerm.GeomNames** no default This key specifies the geometries on which relative permeability will be given. The union of these geometries must cover the entire computational domain.

```
pfset Phase.RelPerm.Geonames   "domain"      ## TCL syntax

<runname>.Phase.RelPerm.Geonames = "domain"  ## Python syntax
```

double **Geom.*geom_name*.RelPerm.Value** no default This key specifies the constant relative permeability value on the specified geometry.

```
pfset Geom.domain.RelPerm.Value    0.5       ## TCL syntax

<runname>.Geom.domain.RelPerm.Value = 0.5    ## Python syntax
```

integer **Phase.RelPerm.VanGenuchten.File** 0 This key specifies whether soil parameters for the VanGenuchten function are specified in a pfb file or by region. The options are either 0 for specification by region, or 1 for specification in a file. Note that either all parameters are specified in files (each has their own input file) or none are specified by files. Parameters specified by files are: α and N.

```
pfset Phase.RelPerm.VanGenuchten.File   1       ## TCL syntax

<runname>.Phase.RelPerm.VanGenuchten.File = 1   ## Python syntax
```

string **Geom.*geom_name*.RelPerm.Alpha.Filename** no default This key specifies a pfb filename containing the alpha parameters for the VanGenuchten function cell-by-cell. The ONLY option for *geom_name* is "domain".

```
pfset Geom.domain.RelPerm.Alpha.Filename   "alphas.pfb"        ## TCL syntax

<runname>.Geom.domain.RelPerm.Alpha.Filename = "alphas.pfb"    ## Python syntax
```

string **Geom.*geom_name*.RelPerm.N.Filename** no default This key specifies a pfb filename containing the N parameters for the VanGenuchten function cell-by-cell. The ONLY option for *geom_name* is "domain".

```
pfset Geom.domain.RelPerm.N.Filename   "Ns.pfb"       ## TCL syntax

<runname>.Geom.domain.RelPerm.N.Filename = "Ns.pfb"   ## Python syntax
```

double **Geom.*geom_name*.RelPerm.Alpha** no default This key specifies the α parameter for the Van Genuchten function specified on *geom_name*.

```
pfset Geom.domain.RelPerm.Alpha  0.005          ## TCL syntax

<runname>.Geom.domain.RelPerm.Alpha = 0.005     ## Python syntax
```

double **Geom.*geom_name*.RelPerm.N** no default This key specifies the N parameter for the Van Genuchten function specified on *geom_name*.

```
pfset Geom.domain.RelPerm.N   2.0          ## TCL syntax

<runname>.Geom.domain.RelPerm.N = 2.0      ## Python syntax
```

int **Geom.*geom_name*.RelPerm.NumSamplePoints** 0 This key specifies the number of sample points for a spline base interpolation table for the Van Genuchten function specified on *geom_name*. If this number is 0 (the default) then the function is evaluated directly. Using the interpolation table is faster but is less accurate.

```
pfset Geom.domain.RelPerm.NumSamplePoints  20000         ## TCL syntax

<runname>.Geom.domain.RelPerm.NumSamplePoints = 20000    ## Python syntax
```

int **Geom.*geom_name*.RelPerm.MinPressureHead** no default This key specifies the lower value for a spline base interpolation table for the Van Genuchten function specified on *geom_name*. The upper value of the range is 0. This value is used only when the table lookup method is used (*NumSamplePoints* is greater than 0).

```
pfset Geom.domain.RelPerm.MinPressureHead -300        ## TCL syntax

<runname>.Geom.domain.RelPerm.MinPressureHead = -300  ## Python syntax
```

double **Geom.*geom_name*.RelPerm.A** no default This key specifies the A parameter for the Haverkamp relative permeability on *geom_name*.

```
pfset Geom.domain.RelPerm.A  1.0           ## TCL syntax

<runname>.Geom.domain.RelPerm.A = 1.0      ## Python syntax
```

double **Geom.*geom_name*.RelPerm.Gamma** no default This key specifies the the γ parameter for the Haverkamp relative permeability on *geom_name*.

```
pfset Geom.domain.RelPerm.Gamma  1.0         ## TCL syntax

<runname>.Geom.domain.RelPerm.Gamma = 1.0    ## Python syntax
```

integer **Geom.*geom_name*.RelPerm.Degree** no default This key specifies the degree of the polynomial for the Polynomial relative permeability given on *geom_name*.

```
pfset Geom.domain.RelPerm.Degree  1       ## TCL syntax

<runname>.Geom.domain.RelPerm.Degree = 1  ## Python syntax
```

double **Geom.*geom_name*.RelPerm.Coeff.*coeff_number*** no default This key specifies the *coeff_number*th coefficient of the Polynomial relative permeability given on *geom_name*.

```
## TCL syntax
pfset Geom.domain.RelPerm.Coeff.0  0.5
pfset Geom.domain.RelPerm.Coeff.1  1.0

## Python syntax
<runname>.Geom.domain.RelPerm.Coeff.0 = 0.5
<runname>.Geom.domain.RelPerm.Coeff.1 = 1.0
```

NOTE: For all these cases, if only one region is to be used (the domain), the background region should NOT be set as that single region. Using the background will prevent the upstream weighting from being correct near Dirichlet boundaries.

6.21 Phase Sources

The following keys are used to specify phase source terms. The units of the source term are $1/T$. So, for example, to specify a region with constant flux rate of L^3/T, one must be careful to convert this rate to the proper units by dividing by the volume of the enclosing region. For *Richards' equation* input, the source term must be given as a flux multiplied by density.

string **PhaseSources.*phase_name*.Type** no default This key specifies the type of source to use for phase *phase_name*. Possible values for this key are **Constant** and **PredefinedFunction**. **Constant** type phase sources specify a constant phase source value for a given set of regions. **PredefinedFunction** type phase sources use a preset function (choices are listed below) to specify the source. Note that the **PredefinedFunction** type can only be used to set a single source over the entire domain and not separate sources over different regions.

```
pfset PhaseSources.water.Type   "Constant"      ## TCL syntax

<runname>.PhaseSources.water.Type = "Constant"  ## Python syntax
```

list **PhaseSources.*phase_name*.GeomNames** no default This key specifies the names of the geometries on which source terms will be specified. This is used only for **Constant** type phase sources. Regions listed later "overlay" regions listed earlier.

```
pfset PhaseSources.water.GeomNames   "bottomlayer middlelayer toplayer"       ## TCL␣
↪syntax

```

(continues on next page)

(continued from previous page)

```
<runname>.PhaseSources.water.GeomNames = "bottomlayer middlelayer toplayer"    ## Python␣
↪syntax
```

double **PhaseSources.*phase_name*.Geom.*geom_name*.Value** no default This key specifies the value of a constant source term applied to phase *phase _name* on geometry *geom_name*.

```
pfset PhaseSources.water.Geom.toplayer.Value   1.0       ## TCL syntax

<runname>.PhaseSources.water.Geom.toplayer.Value = 1.0   ## Python syntax
```

string **PhaseSources.*phase_name*.PredefinedFunction** no default This key specifies which of the predefined functions will be used for the source. Possible values for this key are **X, XPlusYPlusZ, X3Y2PlusSinXYPlus1,** and **XYZTPlus1PermTensor**.

```
pfset PhaseSources.water.PredefinedFunction   "XPlusYPlusZ"       ## TCL syntax

<runname>.PhaseSources.water.PredefinedFunction = "XPlusYPlusZ"   ## Python syntax
```

The choices for this key correspond to sources as follows:

X:

$$\text{source } = 0.0$$

XPlusYPlusX:

$$\text{source } = 0.0$$

X3Y2PlusSinXYPlus1:

$$\text{source } = -(3x^2y^2+y\cos(xy))^2-(2x^3y+x\cos(xy))^2-(x^3y^2+\sin(xy)+1)(6xy^2+2x^3-(x^2+y^2)\sin(xy))$$

This function type specifies that the source applied over the entire domain is as noted above. This corresponds to $p = x^3y^2 + \sin(xy) + 1$ in the problem $-\nabla \cdot (p\nabla p) = f$.

X3Y4PlusX2PlusSinXYCosYPlus1:

$$\text{source } = -(3x^22y^4 + 2x + y\cos(xy)\cos(y))^2 - (4x^3y^3 + x\cos(xy)\cos(y) - \sin(xy)\sin(y))^2 - (x^3y^4 + x^2 + \sin(xy)\cos(y) + 1)(6xy^4 + 2 - (x^2 + y^2 + 1)\sin(xy)\cos(y) + 12x^3y^2 - 2x\cos(xy)\sin(y))$$

This function type specifies that the source applied over the entire domain is as noted above. This corresponds to $p = x^3y^4 + x^2 + \sin(xy)\cos(y) + 1$ in the problem $-\nabla \cdot (p\nabla p) = f$.

XYZTPlus1:

$$\text{source } = xyz - t^2(x^2y^2 + x^2z^2 + y^2z^2)$$

This function type specifies that the source applied over the entire domain is as noted above. This corresponds to $p = xyzt + 1$ in the problem $\frac{\partial p}{\partial t} - \nabla \cdot (p\nabla p) = f$.

XYZTPlus1PermTensor:

$$\text{source } = xyz - t^2(x^2y^23 + x^2z^22 + y^2z^2)$$

This function type specifies that the source applied over the entire domain is as noted above. This corresponds to $p = xyzt + 1$ in the problem $\frac{\partial p}{\partial t} - \nabla \cdot (Kp\nabla p) = f$, where $K = diag(1 \;\; 2 \;\; 3)$.

6.22 Capillary Pressures

Here we define capillary pressure. Note: this section needs to be defined *only* for multi-phase flow and should not be defined for single phase and Richards' equation cases. The format for this section of input is:

string **CapPressure.*phase_name*.Type** "Constant" This key specifies the capillary pressure between phase 0 and the named phase, *phase_name*. The only choice available is **Constant** which indicates that a constant capillary pressure exists between the phases.

```
pfset CapPressure.water.Type   "Constant"        ## TCL syntax

<runname>.CapPressure.water.Type = "Constant"    ## Python syntax
```

list **CapPressure.*phase_name*.GeomNames** no default This key specifies the geometries that capillary pressures will be computed for in the named phase, *phase_name*. Regions listed later "overlay" regions listed earlier. Any geometries not listed will be assigned 0.0 capillary pressure by ParFlow.

```
pfset CapPressure.water.GeomNames   "domain"       ## TCL syntax

<runname>.CapPressure.water.GeomNames = "domain"   ## Python syntax
```

double **Geom.*geometry_name*.CapPressure.*phase_name*.Value** 0.0 This key specifies the value of the capillary pressure in the named geometry, *geometry_name*, for the named phase, *phase_name*.

```
pfset Geom.domain.CapPressure.water.Value   0.0       ## TCL syntax

<runname>.Geom.domain.CapPressure.water.Value = 0.0   ## Python syntax
```

Important note: the code currently works only for capillary pressure equal zero.

6.23 Saturation

This section is *only* relevant to the Richards' equation cases. All keys relating to this section will be ignored for other cases. The following keys are used to define the saturation-pressure curve.

string **Phase.Saturation.Type** no default This key specifies the type of saturation function that will be used on all specified geometries. Note that only one type of saturation may be used for the entire problem. However, parameters may be different for that type in different geometries. For instance, if the problem consists of three geometries, then **VanGenuchten** may be specified with three different sets of parameters for the three different goemetries. However, once **VanGenuchten** is specified, one geometry cannot later be specified to have **Data** as its saturation. The possible values for this key are **Constant, VanGenuchten, Haverkamp, Data, Polynomial** and **PFBFile**.

```
pfset Phase.Saturation.Type   "Constant"        ## TCL syntax

<runname>.Phase.Saturation.Type = "Constant"    ## Python syntax
```

The various possible functions are defined as follows. The **Constant** specification means that the saturation will be constant on the specified geounit. The **VanGenuchten** specification means that the saturation will be given as a Van Genuchten function [VG80] with the form,

$$s(p) = \frac{s_{sat} - s_{res}}{(1 + (\alpha p)^n)^m} + s_{res},$$

where s_{sat} is the saturation at saturated conditions, s_{res} is the residual saturation, and α and n are soil parameters with $m = 1 - 1/n$, on each region. The **Haverkamp** specification means that the saturation will be given in the following form [HV81],

$$s(p) = \frac{\alpha(s_{sat} - s_{res})}{A + p^\gamma} + s_{res},$$

where A and γ are soil parameters, on each region. The **Data** specification is currently unsupported but will later mean that data points for the saturation curve will be given and ParFlow will set up the proper interpolation coefficients to get values between the given data points. The **Polynomial** specification defines a polynomial saturation function for each region of the form,

$$s(p) = \sum_{i=0}^{degree} c_i p^i.$$

The **PFBFile** specification means that the saturation will be taken as a spatially varying but constant in pressure function given by data in a ParFlow binary (.pfb) file.

list **Phase.Saturation.GeomNames** no default This key specifies the geometries on which saturation will be given. The union of these geometries must cover the entire computational domain.

```
pfset Phase.Saturation.Geonames   "domain"         ## TCL syntax

<runname>.Phase.Saturation.Geonames = "domain"     ## Python syntax
```

double **Geom.*geom_name*.Saturation.Value** no default This key specifies the constant saturation value on the *geom_name* region.

```
pfset Geom.domain.Saturation.Value    0.5       ## TCL syntax

<runname>.Geom.domain.Saturation.Value = 0.5    ## Python syntax
```

integer **Phase.Saturation.VanGenuchten.File** 0 This key specifies whether soil parameters for the VanGenuchten function are specified in a pfb file or by region. The options are either 0 for specification by region, or 1 for specification in a file. Note that either all parameters are specified in files (each has their own input file) or none are specified by files. Parameters specified by files are α, N, SRes, and SSat.

```
pfset Phase.Saturation.VanGenuchten.File   1       ## TCL syntax

<runname>.Phase.Saturation.VanGenuchten.File = 1   ## Python syntax
```

string **Geom.*geom_name*.Saturation.Alpha.Filename** no default This key specifies a pfb filename containing the alpha parameters for the VanGenuchten function cell-by-cell. The ONLY option for *geom_name* is "domain".

```
pfset Geom.domain.Saturation.Filename   "alphas.pfb"     ## TCL syntax

<runname.Geom.domain.Saturation.Filename = "alphas.pfb"  ## Python syntax
```

string **Geom.*geom_name*.Saturation.N.Filename** no default This key specifies a pfb filename containing the N parameters for the VanGenuchten function cell-by-cell. The ONLY option for *geom_name* is "domain".

```
pfset Geom.domain.Saturation.N.Filename   "Ns.pfb"    ## TCL syntax

pfset Geom.domain.Saturation.N.Filename = "Ns.pfb"    ## Python syntax
```

string **Geom.*geom_name*.Saturation.SRes.Filename** no default This key specifies a pfb filename containing the SRes parameters for the VanGenuchten function cell-by-cell. The ONLY option for *geom_name* is "domain".

```
pfset Geom.domain.Saturation.SRes.Filename   "SRess.pfb"          ## TCL syntax

<runname>.Geom.domain.Saturation.SRes.Filename = "SRess.pfb"      ## Python syntax
```

string **Geom.*geom_name*.Saturation.SSat.Filename** no default This key specifies a pfb filename containing the SSat parameters for the VanGenuchten function cell-by-cell. The ONLY option for *geom_name* is "domain".

```
pfset Geom.domain.Saturation.SSat.Filename   "SSats.pfb"       ## TCL syntax

<runname>.Geom.domain.Saturation.SSat.Filename = "SSats.pfb"   ## Python syntax
```

double **Geom.*geom_name*.Saturation.Alpha** no default This key specifies the α parameter for the Van Genuchten function specified on *geom_name*.

```
pfset Geom.domain.Saturation.Alpha  0.005          ## TCL syntax

<runname>.Geom.domain.Saturation.Alpha = 0.005     ## Python syntax
```

double **Geom.*geom_name*.Saturation.N** no default This key specifies the N parameter for the Van Genuchten function specified on *geom_name*.

```
pfset Geom.domain.Saturation.N   2.0          ## TCL syntax

<runname>.Geom.domain.Saturation.N = 2.0      ## Python syntax
```

Note that if both a Van Genuchten saturation and relative permeability are specified, then the soil parameters should be the same for each in order to have a consistent problem.

double **Geom.*geom_name*.Saturation.SRes** no default This key specifies the residual saturation on *geom_name*.

```
pfset Geom.domain.Saturation.SRes   0.0          ## TCL syntax

<runname>.Geom.domain.Saturation.SRes = 0.0      ## Python syntax
```

double **Geom.*geom_name*.Saturation.SSat** no default This key specifies the saturation at saturated conditions on *geom_name*.

```
pfset Geom.domain.Saturation.SSat   1.0          ## TCL syntax

<runname>.Geom.domain.Saturation.SSat = 1.0      ## Python syntax
```

double **Geom.*geom_name*.Saturation.A** no default This key specifies the A parameter for the Haverkamp saturation on *geom_name*.

```
pfset Geom.domain.Saturation.A   1.0          ## TCL syntax

<runname>.Geom.domain.Saturation.A = 1.0      ## Python syntax
```

double **Geom.*geom_name*.Saturation.Gamma** no default This key specifies the the γ parameter for the Haverkamp saturation on *geom_name*.

```
pfset Geom.domain.Saturation.Gamma   1.0          ## TCL syntax

<runname>.Geom.domain.Saturation.Gamma = 1.0      ## Python syntax
```

integer **Geom.*geom_name*.Saturation.Degree** no default This key specifies the degree of the polynomial for the Polynomial saturation given on *geom_name*.

```
pfset Geom.domain.Saturation.Degree   1      ## TCL syntax

<runname>.Geom.domain.Saturation.Degree = 1  ## Python syntax
```

double **Geom.*geom_name*.Saturation.Coeff.*coeff_number*** no default This key specifies the *coeff_number*th coefficient of the Polynomial saturation given on *geom_name*.

```
## TCL syntax
pfset Geom.domain.Saturation.Coeff.0   0.5
pfset Geom.domain.Saturation.Coeff.1   1.0

## Python syntax
<runname>.Geom.domain.Saturation.Coeff.0 = 0.5
<runname>.Geom.domain.Saturation.Coeff.1 = 1.0
```

string **Geom.*geom_name*.Saturation.FileName** no default This key specifies the name of the file containing saturation values for the domain. It is assumed that *geom_name* is "domain" for this key.

```
pfset Geom.domain.Saturation.FileName  "domain_sats.pfb"       ## TCL syntax

<runname>.Geom.domain.Saturation.FileName = "domain_sats.pfb"  ## Python syntax
```

6.24 Internal Boundary Conditions

In this section, we define internal Dirichlet boundary conditions by setting the pressure at points in the domain. The format for this section of input is:

string **InternalBC.Names** no default This key specifies the names for the internal boundary conditions. At each named point, x, y and z will specify the coordinate locations and h will specify the hydraulic head value of the condition. This real location is "snapped" to the nearest gridpoint in ParFlow.

NOTE: Currently, ParFlow assumes that internal boundary conditions and pressure wells are separated by at least one cell from any external boundary. The user should be careful of this when defining the input file and grid.

```
pfset InternalBC.Names   "fixedvalue"        ## TCL syntax

<runname>.InternalBC.Names = "fixedvalue"    ## Python syntax
```

double **InternalBC.*internal_bc_name*.X** no default This key specifies the x-coordinate, x, of the named, *internal_bc_name*, condition.

```
pfset InternalBC.fixedheadvalue.X   40.0        ## TCL syntax

<runname>.InternalBC.fixedheadvalue.X = 40.0    ## Python syntax
```

double **InternalBC.*internal_bc_name*.Y** no default This key specifies the y-coordinate, y, of the named, *internal_bc_name*, condition.

```
pfset InternalBC.fixedheadvalue.Y   65.2        ## TCL syntax

<runname>.InternalBC.fixedheadvalue.Y = 65.2    ## Python syntax
```

double **InternalBC.*internal_bc_name*.Z** no default This key specifies the z-coordinate, z, of the named, *internal_bc_name*, condition.

```
pfset InternalBC.fixedheadvalue.Z   12.1        ## TCL syntax

<runname>.InternalBC.fixedheadvalue.Z = 12.1    ## Python syntax
```

double **InternalBC.*internal_bc_name*.Value** no default This key specifies the value of the named, *internal_bc_name*, condition.

```
pfset InternalBC.fixedheadvalue.Value   100.0         ## TCL syntax

<runname>.InternalBC.fixedheadvalue.Value = 100.0     ## Python syntax
```

6.25 Boundary Conditions: Pressure

Here we define the pressure boundary conditions. The Dirichlet conditions below are hydrostatic conditions, and it is assumed that at each phase interface the pressure is constant. *It is also assumed here that all phases are distributed within the domain at all times such that the lighter phases are vertically higher than the heavier phases.*

Boundary condition input is associated with domain patches (see *Domain*). Note that different patches may have different types of boundary conditions on them.

list **BCPressure.PatchNames** no default This key specifies the names of patches on which pressure boundary conditions will be specified. Note that these must all be patches on the external boundary of the domain and these patches must "cover" that external boundary.

```
pfset BCPressure.PatchNames    "left right front back top bottom"
```

string **Patch.*patch_name*.BCPressure.Type** no default This key specifies the type of boundary condition data given for patch *patch_name*. Possible values for this key are **DirEquilRefPatch, DirEquilPLinear, FluxConst, FluxVolumetric, PressureFile, FluxFile, OverlandFow, OverlandFlowPFB, SeepageFace, OverlandKinematic, OverlandDiffusive** and **ExactSolution**. The choice **DirEquilRefPatch** specifies that the pressure on the specified patch will be in hydrostatic equilibrium with a constant reference pressure given on a reference patch. The choice **DirEquilPLinear** specifies that the pressure on the specified patch will be in hydrostatic equilibrium with pressure given along a piecewise line at elevation $z = 0$. The choice **FluxConst** defines a constant normal flux boundary condition through the domain patch. This flux must be specified in units of $[L]/[T]$. For *Richards' equation*, fluxes must be specified as a mass flux and given as the above flux multiplied by the density. Thus, this choice of input type for a Richards' equation problem has units of $([L]/[T])([M]/[L]^3)$. The choice **FluxVolumetric** defines a volumetric flux boundary condition through the domain patch. The units should be consistent with all other user input for the problem. For *Richards' equation* fluxes must be specified as a mass flux and given as the above flux multiplied by the density. The choice **PressureFile** defines a hydraulic head boundary condition that is read from a properly distributed .pfb file. Only the values needed for the patch are used. The choice **FluxFile** defines a flux boundary condition that is read form a properly distributed .pfb file defined on a grid consistent with the pressure field grid. Only the values needed for the patch are used. The choices **OverlandFlow** and **OverlandFlowPFB** both turn on fully-coupled overland flow routing as described in Kollet and Maxwell [KM06] and in *Overland Flow*. The key **OverlandFlow** corresponds to a **Value** key with a positive or negative value, to indicate uniform fluxes (such as rainfall or evapotranspiration) over the entire domain while the key **OverlandFlowPFB** allows a file to contain grid-based, spatially-variable fluxes. The **OverlandKinematic** and **OverlandDiffusive** both turn on an kinematic and diffusive wave overland flow routing boundary that solve Maning's equation in *Overland Flow* and do the upwinding internally (i.e. assuming that the user provides cell face slopes, as opposed to the traditional cell centered slopes). The key **SeepageFace** simulates a boundary that allows flow to exit but keeps the surface pressure at zero. The choice **ExactSolution** specifies that an exact known solution is to be applied as a Dirichlet boundary condition on the respective patch. Note that this does not change according to any cycle. Instead, time dependence is handled by evaluating at the time the boundary condition value is desired. The solution is specified by using a predefined function (choices are described below). NOTE: These last six types of boundary condition input is for *Richards' equation cases only!*

```
pfset Patch.top.BCPressure.Type  DirEquilRefPatch
```

string **Patch.*patch_name*.BCPressure.Cycle** no default This key specifies the time cycle to which boundary condition data for patch *patch_name* corresponds.

```
pfset Patch.top.BCPressure.Cycle   Constant
```

string **Patch.*patch_name*.BCPressure.RefGeom** no default This key specifies the name of the solid on which the reference patch for the **DirEquilRefPatch** boundary condition data is given. Care should be taken to make sure the correct solid is specified in cases of layered domains.

```
pfset Patch.top.BCPressure.RefGeom   "domain"
```

string **Patch.*patch_name*.BCPressure.RefPatch** no default This key specifies the reference patch on which the **DirEquilRefPatch** boundary condition data is given. This patch must be on the reference solid specified by the Patch.*patch_name*.BCPressure.RefGeom key.

```
pfset Patch.top.BCPressure.RefPatch    "bottom"
```

double **Patch.*patch_name*.BCPressure.*interval_name*.Value** no default This key specifies the reference pressure value for the **DirEquilRefPatch** boundary condition or the constant flux value for the **FluxConst** boundary condition, or the constant volumetric flux for the **FluxVolumetric** boundary condition.

```
pfset Patch.top.BCPressure.alltime.Value  -14.0
```

double **Patch.*patch_name*.BCPressure.*interval_name*.*phase_name*.IntValue** no default Note that the reference conditions for types **DirEquilPLinear** and **DirEquilRefPatch** boundary conditions are for phase 0 *only*. This key specifies the constant pressure value along the interface with phase *phase_name* for cases with two phases present.

```
pfset Patch.top.BCPressure.alltime.water.IntValue   -13.0
```

double **Patch.*patch_name*.BCPressure.*interval_name*.XLower** no default This key specifies the lower x coordinate of a line in the xy-plane.

```
pfset Patch.top.BCPressure.alltime.XLower  0.0
```

double **Patch.*patch_name*.BCPressure.*interval_name*.YLower** no default This key specifies the lower y coordinate of a line in the xy-plane.

```
pfset Patch.top.BCPressure.alltime.YLower  0.0
```

double **Patch.*patch_name*.BCPressure.*interval_name*.XUpper** no default This key specifies the upper x coordinate of a line in the xy-plane.

```
pfset Patch.top.BCPressure.alltime.XUpper  1.0
```

double **Patch.*patch_name*.BCPressure.*interval_name*.YUpper** no default This key specifies the upper y coordinate of a line in the xy-plane.

```
pfset Patch.top.BCPressure.alltime.YUpper  1.0
```

integer **Patch.*patch_name*.BCPressure.*interval_name*.NumPoints** no default This key specifies the number of points on which pressure data is given along the line used in the type **DirEquilPLinear** boundary conditions.

```
pfset Patch.top.BCPressure.alltime.NumPoints   2
```

double **Patch.*patch_name*.BCPressure.*interval_name*.*point_number*.Location** no default This key specifies a number between 0 and 1 which represents the location of a point on the line on which data is given for type **DirEquilPLinear** boundary conditions. Here 0 corresponds to the lower end of the line, and 1 corresponds to the upper end.

```
pfset Patch.top.BCPressure.alltime.0.Location   0.0
```

double **Patch.*patch_name*.BCPressure.*interval_name*.*point_number*.Value** no default This key specifies the pressure value for phase 0 at point number *point_number* and $z = 0$ for type **DirEquilPLinear** boundary conditions. All pressure values on the patch are determined by first projecting the boundary condition coordinate onto the line, then linearly interpolating between the neighboring point pressure values on the line.

```
pfset Patch.top.BCPressure.alltime.0.Value   14.0
```

string **Patch.*patch_name*.BCPressure.*interval_name*.FileName** no default This key specifies the name of a properly distributed `.pfb` file that contains boundary data to be read for types PressureFile and FluxFile. For flux data, the data must be defined over a grid consistent with the pressure field. In both cases, only the values needed for the patch will be used. The rest of the data is ignored.

```
pfset Patch.top.BCPressure.alltime.FileName   "ocwd_bc.pfb"
```

string **Patch.*patch_name*.BCPressure.*interval_name*.PredefinedFunction** no default This key specifies the predefined function that will be used to specify Dirichlet boundary conditions on patch *patch_name*. Note that this does not change according to any cycle. Instead, time dependence is handled by evaluating at the time the boundary condition value is desired. Choices for this key include **X, XPlusYPlusZ, X3Y2PlusSinXYPlus1, X3Y4PlusX2PlusSinXYCosYPlus1, XYZTPlus1** and **XYZTPlus1PermTensor**.

```
pfset Patch.top.BCPressure.alltime.PredefinedFunction  "XPlusYPlusZ"
```

The choices for this key correspond to pressures as follows.

X:
$p = x$

XPlusYPlusZ:
$p = x + y + z$

X3Y2PlusSinXYPlus1:
$p = x^3y^2 + \sin(xy) + 1$

X3Y4PlusX2PlusSinXYCosYPlus1:
$p = x^3y^4 + x^2 + \sin(xy)\cos y + 1$

XYZTPlus1:
$p = xyzt + 1$

XYZTPlus1PermTensor:
$p = xyzt + 1$

Example Script:

```
#-----------------------------------------------------------
# Initial conditions: water pressure [m]
```

(continues on next page)

(continued from previous page)

```
#-----------------------------------------------------------
# Using a patch is great when you are not using a box domain
# If using a box domain HydroStaticDepth is fine
# If your RefPatch is z-lower (bottom of domain), the pressure is positive.
# If your RefPatch is z-upper (top of domain), the pressure is negative.
### Set water table to be at the bottom of the domain, the top layer is initially dry
pfset ICPressure.Type                                   "HydroStaticPatch"
pfset ICPressure.GeomNames                         "domain"
pfset Geom.domain.ICPressure.Value           2.2

pfset Geom.domain.ICPressure.RefGeom      "domain"
pfset Geom.domain.ICPressure.RefPatch     z-lower

### Using a .pfb to initialize
pfset ICPressure.Type                  "PFBFile"
pfset ICPressure.GeomNames                         "domain"
pfset Geom.domain.ICPressure.FileName     "press.00090.pfb"

pfset Geom.domain.ICPressure.RefGeom      "domain"
pfset Geom.domain.ICPressure.RefPatch     "z-upper"
```

6.26 Boundary Conditions: Saturation

Note: this section needs to be defined *only* for multi-phase flow and should *not* be defined for the single phase and Richards' equation cases.

Here we define the boundary conditions for the saturations. Boundary condition input is associated with domain patches (see *Domain*). Note that different patches may have different types of boundary conditions on them.

list **BCSaturation.PatchNames** no default This key specifies the names of patches on which saturation boundary conditions will be specified. Note that these must all be patches on the external boundary of the domain and these patches must "cover" that external boundary.

```
pfset BCSaturation.PatchNames    "left right front back top bottom"
```

string **Patch.*patch_name*.BCSaturation.*phase_name*.Type** no default This key specifies the type of boundary condition data given for the given phase, *phase_name*, on the given patch *patch_name*. Possible values for this key are **DirConstant**, **ConstantWTHeight** and **PLinearWTHeight**. The choice **DirConstant** specifies that the saturation is constant on the whole patch. The choice **ConstantWTHeight** specifies a constant height of the water-table on the whole patch. The choice **PLinearWTHeight** specifies that the height of the water-table on the patch will be given by a piecewise linear function.

Note: the types **ConstantWTHeight** and **PLinearWTHeight** assume we are running a 2-phase problem where phase 0 is the water phase.

```
pfset Patch.left.BCSaturation.water.Type  "ConstantWTHeight"
```

double **Patch.*patch_name*.BCSaturation.*phase_name*.Value** no default This key specifies either the constant saturation value if **DirConstant** is selected or the constant water-table height if **ConstantWTHeight** is selected.

```
pfset Patch.top.BCSaturation.air.Value 1.0
```

double **Patch.*patch_name*.BCSaturation.*phase_name*.XLower** no default This key specifies the lower x coordinate of a line in the xy-plane if type **PLinearWTHeight** boundary conditions are specified.

```
pfset Patch.left.BCSaturation.water.XLower -10.0
```

double **Patch.*patch_name*.BCSaturation.*phase_name*.YLower** no default This key specifies the lower y coordinate of a line in the xy-plane if type **PLinearWTHeight** boundary conditions are specified.

```
pfset Patch.left.BCSaturation.water.YLower 5.0
```

double **Patch.*patch_name*.BCSaturation.*phase_name*.XUpper** no default This key specifies the upper x coordinate of a line in the xy-plane if type **PLinearWTHeight** boundary conditions are specified.

```
pfset Patch.left.BCSaturation.water.XUpper  125.0
```

double **Patch.*patch_name*.BCSaturation.*phase_name*.YUpper** no default This key specifies the upper y coordinate of a line in the xy-plane if type **PLinearWTHeight** boundary conditions are specified.

```
pfset Patch.left.BCSaturation.water.YUpper  82.0
```

integer **Patch.*patch_name*.BCPressure.*phase_name*.NumPoints** no default This key specifies the number of points on which saturation data is given along the line used for type **DirEquilPLinear** boundary conditions.

```
pfset Patch.left.BCPressure.water.NumPoints 2
```

double **Patch.*patch_name*.BCPressure.*phase_name*.*point_number*.Location** no default This key specifies a number between 0 and 1 which represents the location of a point on the line for which data is given in type **DirEquilPLinear** boundary conditions. The line is parameterized so that 0 corresponds to the lower end of the line, and 1 corresponds to the upper end.

```
pfset Patch.left.BCPressure.water.0.Location 0.333
```

double **Patch.*patch_name*.BCPressure.*phase_name*.*point_number*.Value** no default This key specifies the water-table height for the given point if type **DirEquilPLinear** boundary conditions are selected. All saturation values on the patch are determined by first projecting the water-table height value onto the line, then linearly interpolating between the neighboring water-table height values onto the line.

```
pfset Patch.left.BCPressure.water.0.Value  4.5
```

6.27 Initial Conditions: Phase Saturations

Note: this section needs to be defined *only* for multi-phase flow and should *not* be defined for single phase and Richards' equation cases.

Here we define initial phase saturation conditions. The format for this section of input is:

string **ICSaturation.*phase_name*.Type** no default This key specifies the type of initial condition that will be applied to different geometries for given phase, *phase_name*. The only key currently available is **Constant**. The choice **Constant** will apply constants values within geometries for the phase.

```
ICSaturation.water.Type Constant
```

string **ICSaturation.*phase_name*.GeomNames** no default This key specifies the geometries on which an initial condition will be given if the type is set to **Constant**.

Note that geometries listed later "overlay" geometries listed earlier.

```
ICSaturation.water.GeomNames "domain"
```

double **Geom.*geom_input_name*.ICSaturation.*phase_name*.Value** no default This key specifies the initial condition value assigned to all points in the named geometry, *geom_input_name*, if the type was set to **Constant**.

```
Geom.domain.ICSaturation.water.Value 1.0
```

6.28 Initial Conditions: Pressure

The keys in this section are used to specify pressure initial conditions for Richards' equation cases *only*. These keys will be ignored if any other case is run.

string **ICPressure.Type** no default This key specifies the type of initial condition given. The choices for this key are **Constant, HydroStaticDepth, HydroStaticPatch** and **PFBFile**. The choice **Constant** specifies that the initial pressure will be constant over the regions given. The choice **HydroStaticDepth** specifies that the initial pressure within a region will be in hydrostatic equilibrium with a given pressure specified at a given depth. The choice **HydroStaticPatch** specifies that the initial pressure within a region will be in hydrostatic equilibrium with a given pressure on a specified patch. Note that all regions must have the same type of initial data - different regions cannot have different types of initial data. However, the parameters for the type may be different. The **PFBFile** specification means that the initial pressure will be taken as a spatially varying function given by data in a ParFlow binary (.pfb) file.

```
pfset ICPressure.Type   "Constant"
```

list **ICPressure.GeomNames** no default This key specifies the geometry names on which the initial pressure data will be given. These geometries must comprise the entire domain. Note that conditions for regions that overlap other regions will have unpredictable results. The regions given must be disjoint.

```
pfset ICPressure.GeomNames   "toplayer middlelayer bottomlayer"
```

double **Geom.*geom_name*.ICPressure.Value** no default This key specifies the initial pressure value for type **Constant** initial pressures and the reference pressure value for types **HydroStaticDepth** and **HydroStaticPatch**.

```
pfset Geom.toplayer.ICPressure.Value  -734.0
```

double **Geom.*geom_name*.ICPressure.RefElevation** no default This key specifies the reference elevation on which the reference pressure is given for type **HydroStaticDepth** initial pressures.

```
pfset Geom.toplayer.ICPressure.RefElevation  0.0
```

double **Geom.*geom_name*.ICPressure.RefGeom** no default This key specifies the geometry on which the reference patch resides for type **HydroStaticPatch** initial pressures.

```
pfset Geom.toplayer.ICPressure.RefGeom   "bottomlayer"
```

double **Geom.*geom_name*.ICPressure.RefPatch** no default This key specifies the patch on which the reference pressure is given for type **HydorStaticPatch** initial pressures.

```
pfset Geom.toplayer.ICPressure.RefPatch   "bottom"
```

string **Geom.*geom_name*.ICPressure.FileName** no default This key specifies the name of the file containing pressure values for the domain. It is assumed that *geom_name* is "domain" for this key.

```
pfset Geom.domain.ICPressure.FileName  "ic_pressure.pfb"
```

6.29 Initial Conditions: Phase Concentrations

Here we define initial concentration conditions for contaminants. The format for this section of input is:

string **PhaseConcen.*phase_name*.*contaminant_name*.Type** no default This key specifies the type of initial condition that will be applied to different geometries for given phase, *phase_name*, and the given contaminant, *contaminant_name*. The choices for this key are **Constant** or **PFBFile**. The choice **Constant** will apply constants values to different geometries. The choice **PFBFile** will read values from a "ParFlow Binary" file (see *ParFlow Binary Files (.pfb)*).

```
PhaseConcen.water.tce.Type "Constant"
```

string **PhaseConcen.*phase_name*.GeomNames** no default This key specifies the geometries on which an initial condition will be given, if the type was set to **Constant**.

Note that geometries listed later "overlay" geometries listed earlier.

```
PhaseConcen.water.GeomNames "ic_concen_region"
```

double **PhaseConcen.*phase_name*.*contaminant_name*.*geom_input_name*.Value** no default This key specifies the initial condition value assigned to all points in the named geometry, *geom_input_name*, if the type was set to **Constant**.

```
PhaseConcen.water.tce.ic_concen_region.Value 0.001
```

string **PhaseConcen.*phase_name*.*contaminant_name*.FileName** no default This key specifies the name of the "ParFlow Binary" file which contains the initial condition values if the type was set to **PFBFile**.

```
PhaseConcen.water.tce.FileName "initial_concen_tce.pfb"
```

6.30 Known Exact Solution

For *Richards equation cases only* we allow specification of an exact solution to be used for testing the code. Only types that have been coded and predefined are allowed. Note that if this is speccified as something other than no known solution, corresponding boundary conditions and phase sources should also be specified.

string **KnownSolution** no default This specifies the predefined function that will be used as the known solution. Possible choices for this key are **NoKnownSolution, Constant, X, XPlusYPlusZ, X3Y2PlusSinXYPlus1, X3Y4PlusX2PlusSinXYCosYPlus1, XYZTPlus1** and **XYZTPlus1PermTensor**.

```
pfset KnownSolution  "XPlusYPlusZ"
```

Choices for this key correspond to solutions as follows.

NoKnownSolution:
No solution is known for this problem.

Constant:
$p = \text{constant}$

X:
$p = x$

XPlusYPlusZ:
$p = x + y + z$

X3Y2PlusSinXYPlus1:
$p = x^3y^2 + sin(xy) + 1$

X3Y4PlusX2PlusSinXYCosYPlus1:
$p = x^3y^4 + x^2 + \sin(xy)\cos y + 1$

XYZTPlus1:
$p = xyzt + 1$

XYZTPlus1PermTensor:
$p = xyzt + 1$

double **KnownSolution.Value** no default This key specifies the constant value of the known solution for type **Constant** known solutions.

```
pfset KnownSolution.Value  1.0
```

Only for known solution test cases will information on the L^2-norm of the pressure error be printed.

6.31 Wells

Here we define wells for the model. The format for this section of input is:

string **Wells.Names** no default This key specifies the names of the wells for which input data will be given.

```
Wells.Names "test_well inj_well ext_well"
```

string **Wells.*well_name*.InputType** no default This key specifies the type of well to be defined for the given well, *well_name*. This key can be either **Vertical** or **Recirc**. The value **Vertical** indicates that this is a single segmented well whose action will be specified by the user. The value **Recirc** indicates that this is a dual segmented, recirculating, well with one segment being an extraction well and another being an injection well. The extraction well filters out a specified fraction of each contaminant and recirculates the remainder to the injection well where the diluted fluid is injected back in. The phase saturations at the extraction well are passed without modification to the injection well.

Note with the recirculating well, several input options are not needed as the extraction well will provide these values to the injection well.

```
Wells.test_well.InputType "Vertical"
```

string **Wells.*well_name*.Action** no default This key specifies the pumping action of the well. This key can be either **Injection** or **Extraction**. A value of **Injection** indicates that this is an injection well. A value of **Extraction** indicates that this is an extraction well.

```
Wells.test_well.Action "Injection"
```

double **Wells.*well_name*.Type** no default This key specfies the mechanism by which the well works (how ParFlow works with the well data) if the input type key is set to **Vectical**. This key can be either **Pressure** or **Flux**. A value of **Pressure** indicates that the data provided for the well is in terms of hydrostatic pressure and ParFlow will ensure that the computed pressure field satisfies this condition in the computational cells which define the well. A value of **Flux** indicates that the data provided is in terms of volumetric flux rates and ParFlow will ensure that the flux field satisfies this condition in the computational cells which define the well.

```
Wells.test_well.Type "Flux"
```

string **Wells.*well_name*.ExtractionType** no default This key specfies the mechanism by which the extraction well works (how ParFlow works with the well data) if the input type key is set to **Recirc**. This key can be either **Pressure** or **Flux**. A value of **Pressure** indicates that the data provided for the well is in terms of hydrostatic pressure and ParFlow will ensure that the computed pressure field satisfies this condition in the computational cells which define the well. A value of **Flux** indicates that the data provided is in terms of volumetric flux rates and ParFlow will ensure that the flux field satisfies this condition in the computational cells which define the well.

```
Wells.ext_well.ExtractionType "Pressure"
```

string **Wells.*well_name*.InjectionType** no default This key specfies the mechanism by which the injection well works (how ParFlow works with the well data) if the input type key is set to **Recirc**. This key can be either **Pressure** or **Flux**. A value of **Pressure** indicates that the data provided for the well is in terms of hydrostatic pressure and ParFlow will ensure that the computed pressure field satisfies this condition in the computational cells which define the well. A value of **Flux** indicates that the data provided is in terms of volumetric flux rates and ParFlow will ensure that the flux field satisfies this condition in the computational cells which define the well.

```
Wells.inj_well.InjectionType "Flux"
```

double **Wells.*well_name*.X** no default This key specifies the x location of the vectical well if the input type is set to **Vectical** or of both the extraction and injection wells if the input type is set to **Recirc**.

```
Wells.test_well.X 20.0
```

double **Wells.*well_name*.Y** no default This key specifies the y location of the vectical well if the input type is set to **Vectical** or of both the extraction and injection wells if the input type is set to **Recirc**.

```
Wells.test_well.Y 36.5
```

double **Wells.*well_name*.ZUpper** no default This key specifies the z location of the upper extent of a vectical well if the input type is set to **Vectical**.

```
Wells.test_well.ZUpper 8.0
```

double **Wells.*well_name*.ExtractionZUpper** no default This key specifies the z location of the upper extent of a extraction well if the input type is set to **Recirc**.

```
Wells.ext_well.ExtractionZUpper 3.0
```

double **Wells.*well_name*.InjectionZUpper** no default This key specifies the z location of the upper extent of a injection well if the input type is set to **Recirc**.

```
Wells.inj_well.InjectionZUpper 6.0
```

double **Wells.*well_name*.ZLower** no default This key specifies the z location of the lower extent of a vectical well if the input type is set to **Vectical**.

```
Wells.test_well.ZLower 2.0
```

double **Wells.*well_name*.ExtractionZLower** no default This key specifies the z location of the lower extent of a extraction well if the input type is set to **Recirc**.

```
Wells.ext_well.ExtractionZLower 1.0
```

double **Wells.*well_name*.InjectionZLower** no default This key specifies the z location of the lower extent of a injection well if the input type is set to **Recirc**.

```
Wells.inj_well.InjectionZLower 4.0
```

string **Wells.*well_name*.Method** no default This key specifies a method by which pressure or flux for a vertical well will be weighted before assignment to computational cells. This key can only be **Standard** if the type key is set to **Pressure**; or this key can be either **Standard**, **Weighted** or **Patterned** if the type key is set to **Flux**. A value of **Standard** indicates that the pressure or flux data will be used as is. A value of **Weighted** indicates that the flux data is to be weighted by the cells permeability divided by the sum of all cell permeabilities which define the well. The value of **Patterned** is not implemented.

```
Wells.test_well.Method "Weighted"
```

string **Wells.*well_name*.ExtractionMethod** no default This key specifies a method by which pressure or flux for an extraction well will be weighted before assignment to computational cells. This key can only be **Standard** if the type key is set to **Pressure**; or this key can be either **Standard**, **Weighted** or **Patterned** if the type key is set to **Flux**. A value of **Standard** indicates that the pressure or flux data will be used as is. A value of **Weighted** indicates that the flux data is to be weighted by the cells permeability divided by the sum of all cell permeabilities which define the well. The value of **Patterned** is not implemented.

```
Wells.ext_well.ExtractionMethod "Standard"
```

string **Wells.*well_name*.InjectionMethod** no default This key specifies a method by which pressure or flux for an injection well will be weighted before assignment to computational cells. This key can only be **Standard** if the type key is set to **Pressure**; or this key can be either **Standard**, **Weighted** or **Patterned** if the type key is set to **Flux**. A value of **Standard** indicates that the pressure or flux data will be used as is. A value of **Weighted** indicates that the flux data is to be weighted by the cells permeability divided by the sum of all cell permeabilities which define the well. The value of **Patterned** is not implemented.

```
Wells.inj_well.InjectionMethod "Standard"
```

string **Wells.*well_name*.Cycle** no default This key specifies the time cycles to which data for the well *well_name* corresponds.

```
Wells.test_well.Cycle "all_time"
```

double **Wells.*well_name*.*interval_name*.Pressure.Value** no default This key specifies the hydrostatic pressure value for a vectical well if the type key is set to **Pressure**.

Note This value gives the pressure of the primary phase (water) at $z = 0$. The other phase pressures (if any) are computed from the physical relationships that exist between the phases.

```
Wells.test_well.all_time.Pressure.Value 6.0
```

double **Wells.*well_name*.*interval_name*.Extraction.Pressure.Value** no default This key specifies the hydrostatic pressure value for an extraction well if the extraction type key is set to **Pressure**.

Note This value gives the pressure of the primary phase (water) at $z = 0$. The other phase pressures (if any) are computed from the physical relationships that exist between the phases.

```
Wells.ext_well.all_time.Extraction.Pressure.Value 4.5
```

double **Wells.*well_name*.*interval_name*.Injection.Pressure.Value** no default This key specifies the hydrostatic pressure value for an injection well if the injection type key is set to **Pressure**.

Note This value gives the pressure of the primary phase (water) at $z = 0$. The other phase pressures (if any) are computed from the physical relationships that exist between the phases.

```
Wells.inj_well.all_time.Injection.Pressure.Value 10.2
```

double **Wells.*well_name*.*interval_name*.Flux.*phase_name*.Value** no default This key specifies the volumetric flux for a vectical well if the type key is set to **Flux**.

Note only a positive number should be entered, ParFlow assignes the correct sign based on the chosen action for the well.

```
Wells.test_well.all_time.Flux.water.Value 250.0
```

double **Wells.*well_name*.*interval_name*.Extraction.Flux.*phase_name*.Value** no default This key specifies the volumetric flux for an extraction well if the extraction type key is set to **Flux**.

Note only a positive number should be entered, ParFlow assigns the correct sign based on the chosen action for the well.

```
Wells.ext_well.all_time.Extraction.Flux.water.Value 125.0
```

double **Wells.*well_name*.*interval_name*.Injection.Flux.*phase_name*.Value** no default This key specifies the volumetric flux for an injection well if the injection type key is set to **Flux**.

Note only a positive number should be entered, ParFlow assigns the correct sign based on the chosen action for the well.

```
Wells.inj_well.all_time.Injection.Flux.water.Value 80.0
```

double **Wells.*well_name*.*interval_name*.Saturation.*phase_name*.Value** no default This key specifies the saturation value of a vertical well.

```
Wells.test_well.all_time.Saturation.water.Value 1.0
```

double **Wells.*well_name*.*interval_name*.Concentration.*phase_name*.*contaminant_name*.Value** no default This key specifies the contaminant value of a vertical well.

```
Wells.test_well.all_time.Concentration.water.tce.Value 0.0005
```

double **Wells.*well_name*.*interval_name*.Injection.Concentration.*phase_name*.*contaminant_name*.Fraction** no default This key specifies the fraction of the extracted contaminant which gets resupplied to the injection well.

```
Wells.inj_well.all_time.Injection.Concentration.water.tce.Fraction 0.01
```

Multiple wells assigned to one grid location can occur in several instances. The current actions taken by the code are as follows:

- If multiple pressure wells are assigned to one grid cell, the code retains only the last set of overlapping well values entered.
- If multiple flux wells are assigned to one grid cell, the code sums the contributions of all overlapping wells to get one effective well flux.
- If multiple pressure and flux wells are assigned to one grid cell, the code retains the last set of overlapping hydrostatic pressure values entered and sums all the overlapping flux well values to get an effective pressure/flux well value.

6.32 Code Parameters

In addition to input keys related to the physics capabilities and modeling specifics there are some key values used by various algorithms and general control flags for ParFlow. These are described next :

string **Solver.Linear** PCG This key specifies the linear solver used for solver **IMPES**. Choices for this key are **MGSemi, PPCG, PCG** and **CGHS**. The choice **MGSemi** is an algebraic mulitgrid linear solver (not a preconditioned conjugate gradient) which may be less robust than **PCG** as described in Ashby and Falgout [AF96]. The choice **PPCG** is a preconditioned conjugate gradient solver. The choice **PCG** is a conjugate gradient solver with a multigrid preconditioner. The choice **CGHS** is a conjugate gradient solver.

```
pfset Solver.Linear   "MGSemi"         ## TCL syntax

<runname>.Solver.Linear = "MGSemi"     ## Python syntax
```

integer **Solver.SadvectOrder** 2 This key controls the order of the explicit method used in advancing the saturations. This value can be either 1 for a standard upwind first order or 2 for a second order Godunov method.

```
pfset Solver.SadvectOrder 1          ## TCL syntax

<runname>.Solver.SadvectOrder = 1    ## Python syntax
```

integer **Solver.AdvectOrder** 2 This key controls the order of the explicit method used in advancing the concentrations. This value can be either 1 for a standard upwind first order or 2 for a second order Godunov method.

```
pfset Solver.AdvectOrder 2           ## TCL syntax

<runname>.Solver.AdvectOrder = 2     ## Python syntax
```

double **Solver.CFL** 0.7 This key gives the value of the weight put on the computed CFL limit before computing a global timestep value. Values greater than 1 are not suggested and in fact because this is an approximation, values slightly less than 1 can also produce instabilities.

```
pfset Solver.CFL 0.7          ## TCL syntax

<runname>.Solver.CFL = 0.7    ## Python syntax
```

integer **Solver.MaxIter** 1000000 This key gives the maximum number of iterations that will be allowed for time-stepping. This is to prevent a run-away simulation.

```
pfset Solver.MaxIter 100         ## TCL syntax

<runname>.Solver.MaxIter = 100   ## Python syntax
```

double **Solver.RelTol** 1.0 This value gives the relative tolerance for the linear solve algorithm.

```
pfset Solver.RelTol 1.0          ## TCL syntax

<runname>.Solver.RelTol = 1.0    ## Python syntax
```

double **Solver.AbsTol** 1E-9 This value gives the absolute tolerance for the linear solve algorithm.

```
pfset Solver.AbsTol 1E-8         ## TCL syntax

<runname>.Solver.AbsTol = 1E-8   ## Python syntax
```

double **Solver.Drop** 1E-8 This key gives a clipping value for data written to PFSB files. Data values greater than the negative of this value and less than the value itself are treated as zero and not written to PFSB files.

```
pfset Solver.Drop 1E-6           ## TCL syntax

<runname>.Solver.Drop = 1E-6     ## Python syntax
```

double **Solver.OverlandDiffusive.Epsilon** 1E-5 This key provides a minimum value for the $\bar{S_f}$ used in the **OverlandDiffusive** boundary condition.

```
pfset Solver.OverlandDiffusive.Epsilon 1E-7           ## TCL syntax

<runname>.Solver.OverlandDiffusive.Epsilon = 1E-7     ## Python syntax
```

double **Solver.OverlandKinematic.Epsilon** 1E-5 This key provides a minimum value for the $\bar{S_f}$ used in the **OverlandKinematic** boundary condition.

```
pfset Solver.OverlandKinematic.Epsilon 1E-7           ## TCL syntax

<runname>.Solver.OverlandKinematic.Epsilon = 1E-7     ## Python syntax
```

string **Solver.PrintSubsurf** True This key is used to turn on printing of the subsurface data, Permeability and Porosity. The data is printed after it is generated and before the main time stepping loop - only once during the run. The data is written as a PFB file.

```
pfset Solver.PrintSubsurf False          ## TCL syntax

<runname>.Solver.PrintSubsurf = False    ## Python syntax
```

string **Solver.PrintPressure** True This key is used to turn on printing of the pressure data. The printing of the data is controlled by values in the timing information section. The data is written as a PFB file.

```
pfset Solver.PrintPressure False         ## TCL syntax

<runname>.Solver.PrintPressure = False   ## Python syntax
```

string **Solver.PrintVelocities** False This key is used to turn on printing of the x, y, and z velocity (Darcy flux) data. The printing of the data is controlled by values in the timing information section. The x, y, and z data are written to separate PFB files. The dimensions of these files are slightly different than most PF data, with the dimension of interest representing interfaces, and the other two dimensions representing cells. E.g. the x-velocity PFB has dimensions [NX+1, NY, NZ]. This key produces files in the format of `<runname>.out.phase<x||y||z>.00.0000.pfb` when using ParFlow's saturated solver and `<runname>.out.vel<x||y||z>.00000.pfb` when using the Richards equation solver.

```
pfset Solver.PrintVelocities True         ## TCL syntax

<runname>.Solver.PrintVelocities = True   ## Python syntax
```

string **Solver.PrintSaturation** True This key is used to turn on printing of the saturation data. The printing of the data is controlled by values in the timing information section. The data is written as a PFB file.

```
pfset Solver.PrintSaturation False        ## TCL syntax

<runname>.Solver.PrintSaturation = False  ## Python syntax
```

string **Solver.PrintConcentration** True This key is used to turn on printing of the concentration data. The printing of the data is controlled by values in the timing information section. The data is written as a PFSB file.

```
pfset Solver.PrintConcentration False           ## TCL syntax

<runname>.Solver.PrintConcentration = False     ## Python syntax
```

string **Solver.PrintWells** True This key is used to turn on collection and printing of the well data. The data is collected at intervals given by values in the timing information section. Printing occurs at the end of the run when all collected data is written.

```
pfset Solver.PrintWells False          ## TCL syntax

<runname>.Solver.PrintWells = False    ## Python syntax
```

string **Solver.PrintLSMSink** False This key is used to turn on printing of the flux array passed from `CLM` to ParFlow. Printing occurs at each **DumpInterval** time.

```
pfset Solver.PrintLSMSink True            ## TCL syntax

<runname>.Solver.PrintLSMSink = True      ## Python syntax
```

string **Solver.WriteSiloSubsurfData** False This key is used to specify printing of the subsurface data, Permeability and Porosity in silo binary file format. The data is printed after it is generated and before the main time stepping loop - only once during the run. This data may be read in by VisIT and other visualization packages.

```
pfset Solver.WriteSiloSubsurfData True          ## TCL syntax

<runname>.Solver.WriteSiloSubsurfData = True    ## Python syntax
```

string **Solver.WriteSiloPressure** False This key is used to specify printing of the saturation data in silo binary format. The printing of the data is controlled by values in the timing information section. This data may be read in by VisIT and other visualization packages.

```
pfset Solver.WriteSiloPressure True          ## TCL syntax

<runname>.Solver.WriteSiloPressure = True    ## Python syntax
```

string **Solver.WriteSiloSaturation** False This key is used to specify printing of the saturation data using silo binary format. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloSaturation True        ## TCL syntax

<runname>.Solver.WriteSiloSaturation = True  ## Python syntax
```

string **Solver.WriteSiloConcentration** False This key is used to specify printing of the concentration data in silo binary format. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloConcentration True           ## TCL syntax

<runname>.Solver.WriteSiloConcentration = True     ## Python syntax
```

string **Solver.WriteSiloVelocities** False This key is used to specify printing of the x, y and z velocity data in silo binary format. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloVelocities True           ## TCL syntax

<runname>.Solver.WriteSiloVelocities = True     ## Python syntax
```

string **Solver.WriteSiloSlopes** False This key is used to specify printing of the x and y slope data using silo binary format. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloSlopes  True        ## TCL syntax

<runname>.Solver.WriteSiloSlopes = True   ## Python syntax
```

string **Solver.WriteSiloMannings** False This key is used to specify printing of the Manning's roughness data in silo binary format. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloMannings True          ## TCL syntax

<runname>.Solver.WriteSiloMannings = True    ## Python syntax
```

string **Solver.WriteSiloSpecificStorage** False This key is used to specify printing of the specific storage data in silo binary format. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloSpecificStorage True         ## TCL syntax

<runname>.Solver.WriteSiloSpecificStorage = True   ## Python syntax
```

string **Solver.WriteSiloMask** False This key is used to specify printing of the mask data using silo binary format. The mask contains values equal to one for active cells and zero for inactive cells. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloMask  True          ## TCL syntax

<runname>.Solver.WriteSiloMask = True     ## Python syntax
```

string **Solver.WriteSiloEvapTrans** False This key is used to specify printing of the evaporation and rainfall flux data using silo binary format. This data comes from either `clm` or from external calls to ParFlow such as WRF. This data is in units of $[L^3 T^{-1}]$. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloEvapTrans  True        ## TCL syntax

<runname>.Solver.WriteSiloEvapTrans = True   ## Python syntax
```

string **Solver.WriteSiloEvapTransSum** False This key is used to specify printing of the evaporation and rainfall flux data using silo binary format as a running, cumulative amount. This data comes from either `clm` or from external calls to ParFlow such as WRF. This data is in units of $[L^3]$. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloEvapTransSum  True          ## TCL syntax

<runname>.Solver.WriteSiloEvapTransSum = True     ## Python syntax
```

string **Solver.WriteSiloOverlandSum** False This key is used to specify calculation and printing of the total overland outflow from the domain using silo binary format as a running cumulative amount. This is integrated along all domain boundaries and is calculated any location that slopes at the edge of the domain point outward. This data is in units of $[L^3]$. The printing of the data is controlled by values in the timing information section.

```
pfset Solver.WriteSiloOverlandSum  True             ## TCL syntax

<runname>.Solver.WriteSiloOverlandSum = True        ## Python syntax
```

string **Solver.TerrainFollowingGrid** False This key specifies that a terrain-following coordinate transform is used for solver Richards. This key sets x and y subsurface slopes to be the same as the Topographic slopes (a value of False sets these subsurface slopes to zero). These slopes are used in the Darcy fluxes to add a density, gravity -dependent term. This key will not change the output files (that is the output is still orthogonal) or the geometries (they will still follow the computational grid)– these two things are both to do items. This key only changes solver Richards, not solver Impes.

```
pfset Solver.TerrainFollowingGrid  True          ## TCL syntax

<runname>.Solver.TerrainFollowingGrid = True     ## Python syntax
```

string **Solver.TerrainFollowingGrid.SlopeUpwindFormulation** Original This key specifies optional modifications to the terrain following grid formulation described in *Terrain Following Grid*. Choices for this key are **Original, Upwind, UpwindSine**. **Original** is the original TFG formulation documented in [Max13]. The **Original** option calculates the θ_x and θ_y for a cell face as the average of the two adjacent cell slopes (i.e. assuming a cell centered slope calculation). The **Upwind** option uses the the θ_x and θ_y of a cell directly without averaging (i.e. assuming a face centered slope calculation). The **UpwindSine** is the same as the **Upwind** option but it also removes the Sine term from the TFG Darcy Formulation (in *Terrain Following Grid*). Note the **UpwindSine** option is for experimental purposes only and should not be used in standard simulations. Also note that the choice of **upwind** or **Original** formulation should consistent with the choice of overland flow boundary condition if overland flow is being used. The **upwind** and **UpwindSine** are consistent with **OverlandDiffusive** and **OverlandKinematic** while **Original** is consistent with **OverlandFow**

```
pfset Solver.TerrainFollowingGrid.SlopeUpwindFormulation   "Upwind"        ## TCL syntax

<runname>.Solver.TerrainFollowingGrid.SlopeUpwindFormulation = "Upwind"    ## Python␣
↪syntax
```

6.33 SILO Options

The following keys are used to control how SILO writes data. SILO allows writing to PDB and HDF5 file formats. SILO also allows data compression to be used, which can save signicant amounts of disk space for some problems.

string **SILO.Filetype** PDB This key is used to specify the SILO filetype. Allowed values are PDB and HDF5. Note that you must have configured SILO with HDF5 in order to use that option.

```
pfset SILO.Filetype  "PDB"       ## TCL syntax

<runname>.SILO.Filetype = "PDB"  ## Python syntax
```

string **SILO.CompressionOptions** This key is used to specify the SILO compression options. See the SILO manual for the DB_SetCompression command for information on available options. NOTE: the options avaialable are highly dependent on the configure options when building SILO.

```
pfset SILO.CompressionOptions  "METHOD=GZIP"          ## TCL syntax

<runname>.SILO.CompressionOptions = "METHOD=GZIP"     ## Python syntax
```

6.34 Richards' Equation Solver Parameters

The following keys are used to specify various parameters used by the linear and nonlinear solvers in the Richards' equation implementation. For information about these solvers, see Woodward [Woo98] and Ashby and Falgout [AF96].

double **Solver.Nonlinear.ResidualTol** 1e-7 This key specifies the tolerance that measures how much the relative reduction in the nonlinear residual should be before nonlinear iterations stop. The magnitude of the residual is measured with the l^1 (max) norm.

```
pfset Solver.Nonlinear.ResidualTol   1e-4          ## TCL syntax

<runname>.Solver.Nonlinear.ResidualTol = 1e-4      ## Python syntax
```

double **Solver.Nonlinear.StepTol** 1e-7 This key specifies the tolerance that measures how small the difference between two consecutive nonlinear steps can be before nonlinear iterations stop.

```
pfset Solver.Nonlinear.StepTol   1e-4         ## TCL syntax

<runname>.Solver.Nonlinear.StepTol = 1e-4     ## Python syntax
```

integer **Solver.Nonlinear.MaxIter** 15 This key specifies the maximum number of nonlinear iterations allowed before iterations stop with a convergence failure.

```
pfset Solver.Nonlinear.MaxIter   50       ## TCL syntax

<runname>.Solver.Nonlinear.MaxIter = 50   ## Python syntax
```

integer **Solver.Linear.KrylovDimension** 10 This key specifies the maximum number of vectors to be used in setting up the Krylov subspace in the GMRES iterative solver. These vectors are of problem size and it should be noted that large increases in this parameter can limit problem sizes. However, increasing this parameter can sometimes help nonlinear solver convergence.

```
pfset Solver.Linear.KrylovDimension   15        ## TCL syntax

<runname>.Solver.Linear.KrylovDimension = 15    ## Python syntax
```

integer **Solver.Linear.MaxRestarts** 0 This key specifies the number of restarts allowed to the GMRES solver. Restarts start the development of the Krylov subspace over using the current iterate as the initial iterate for the next pass.

```
pfset Solver.Linear.MaxRestarts   2       ## TCL syntax

<runname>.Solver.Linear.MaxRestarts = 2   ## Python syntax
```

integer **Solver.MaxConvergenceFailures** 3 This key gives the maximum number of convergence failures allowed. Each convergence failure cuts the timestep in half and the solver tries to advance the solution with the reduced timestep.

The default value is 3.

Note that setting this value to a value greater than 9 may result in errors in how time cycles are calculated. Time is discretized in terms of the base time unit and if the solver begins to take very small timesteps $smaller than base time unit 1000$ the values based on time cycles will be change at slightly incorrect times. If the problem is failing converge so poorly that a large number of restarts are required, consider setting the timestep to a smaller value.

```
pfset Solver.MaxConvergenceFailures 4           ## TCL syntax

<runname>.Solver.MaxConvergenceFailures = 4     ## Python syntax
```

string **Solver.Nonlinear.PrintFlag** HighVerbosity This key specifies the amount of informational data that is printed to the `*.out.kinsol.log` file. Choices for this key are **NoVerbosity**, **LowVerbosity**, **NormalVerbosity** and **HighVerbosity**. The choice **NoVerbosity** prints no statistics about the nonlinear convergence process. The choice **LowVerbosity** outputs the nonlinear iteration count, the scaled norm of the nonlinear function, and the number of function calls. The choice **NormalVerbosity** prints the same as for **LowVerbosity** and also the global strategy statistics. The choice **HighVerbosity** prints the same as for **NormalVerbosity** with the addition of further Krylov iteration statistics.

```
pfset Solver.Nonlinear.PrintFlag   "NormalVerbosity"        ## TCL syntax

<runname>.Solver.Nonlinear.PrintFlag = "NormalVerbosity"    ## Python syntax
```

string **Solver.Nonlinear.EtaChoice** Walker2 This key specifies how the linear system tolerance will be selected. The linear system is solved until a relative residual reduction of η is achieved. Linear residual norms are measured in the l^2 norm. Choices for this key include **EtaConstant, Walker1** and **Walker2**. If the choice **EtaConstant** is specified, then η will be taken as constant. The choices **Walker1** and **Walker2** specify choices for η developed by Eisenstat and Walker [EW96]. The choice **Walker1** specifies that η will be given by $|\|F(u^k)\| - \|F(u^{k-1}) + J(u^{k-1}) * p\||/\|F(u^{k-1})\|$. The choice **Walker2** specifies that η will be given by $\gamma\|F(u^k)\|/\|F(u^{k-1})\|^{\alpha}$. For both of the last two choices, η is never allowed to be less than 1e-4.

```
pfset Solver.Nonlinear.EtaChoice   "EtaConstant"          ## TCL syntax

<runname>.Solver.Nonlinear.EtaChoice = "EtaConstant"      ## Python syntax
```

double **Solver.Nonlinear.EtaValue** 1e-4 This key specifies the constant value of η for the EtaChoice key **EtaConstant**.

```
pfset Solver.Nonlinear.EtaValue   1e-7          ## TCL syntax

<runname>.Solver.Nonlinear.EtaValue = 1e-7      ## Python syntax
```

double **Solver.Nonlinear.EtaAlpha** 2.0 This key specifies the value of α for the case of EtaChoice being **Walker2**.

```
pfset Solver.Nonlinear.EtaAlpha   1.0         ## TCL syntax

<runname>.Solver.Nonlinear.EtaAlpha = 1.0     ## Python syntax
```

double **Solver.Nonlinear.EtaGamma** 0.9 This key specifies the value of γ for the case of EtaChoice being **Walker2**.

```
pfset Solver.Nonlinear.EtaGamma   0.7         ## TCL syntax

<runname>.Solver.Nonlinear.EtaGamma = 0.7     ## Python syntax
```

string **Solver.Nonlinear.UseJacobian** False This key specifies whether the Jacobian will be used in matrix-vector products or whether a matrix-free version of the code will run. Choices for this key are **False** and **True**. Using the Jacobian will most likely decrease the number of nonlinear iterations but require more memory to run.

```
pfset Solver.Nonlinear.UseJacobian   True           ## TCL syntax

<runname>.Solver.Nonlinear.UseJacobian = True       ## Python syntax
```

double **Solver.Nonlinear.DerivativeEpsilon** 1e-7 This key specifies the value of ϵ used in approximating the action of the Jacobian on a vector with approximate directional derivatives of the nonlinear function. This parameter is only used when the UseJacobian key is **False**.

```
pfset Solver.Nonlinear.DerivativeEpsilon   1e-8       ## TCL syntax

<runname>.Solver.Nonlinear.DerivativeEpsilon = 1e-8   ## Python syntax
```

string **Solver.Nonlinear.Globalization** LineSearch This key specifies the type of global strategy to use. Possible choices for this key are **InexactNewton** and **LineSearch**. The choice **InexactNewton** specifies no global strategy, and the choice **LineSearch** specifies that a line search strategy should be used where the nonlinear step can be lengthened or decreased to satisfy certain criteria.

```
pfset Solver.Nonlinear.Globalization   "LineSearch"         ## TCL syntax

<runname>.Solver.Nonlinear.Globalization = "LineSearch"     ## Python syntax
```

string **Solver.Linear.Preconditioner** MGSemi This key specifies which preconditioner to use. Currently, the three choices are **NoPC, MGSemi, PFMG, PFMGOctree** and **SMG**. The choice **NoPC** specifies that no preconditioner should be used. The choice **MGSemi** specifies a semi-coarsening multigrid algorithm which uses a point relaxation method. The choice **SMG** specifies a semi-coarsening multigrid algorithm which uses plane relaxations. This method is more robust than **MGSemi**, but generally requires more memory and compute time. The choice **PFMGOctree** can be more efficient for problems with large numbers of inactive cells.

```
pfset Solver.Linear.Preconditioner   "MGSemi"          ## TCL syntax

<runname>.Solver.Linear.Preconditioner = "MGSemi"      ## Python syntax
```

string **Solver.Linear.Preconditioner.SymmetricMat** Symmetric This key specifies whether the preconditioning matrix is symmetric. Choices for this key are **Symmetric** and **Nonsymmetric**. The choice **Symmetric** specifies that the symmetric part of the Jacobian will be used as the preconditioning matrix. The choice **Nonsymmetric** specifies that the full Jacobian will be used as the preconditioning matrix. NOTE: ONLY **Symmetric** CAN BE USED IF MGSemi IS THE SPECIFIED PRECONDITIONER!

```
pfset Solver.Linear.Preconditioner.SymmetricMat     "Symmetric"      ## TCL syntax

<runname>.Solver.Linear.Preconditioner.SymmetricMat = "Symmetric"    ## Python syntax
```

integer **Solver.Linear.Preconditioner.*precond_method*.MaxIter** 1 This key specifies the maximum number of iterations to take in solving the preconditioner system with *precond_method* solver.

```
pfset Solver.Linear.Preconditioner.SMG.MaxIter    2          ## TCL syntax

<runname>.Solver.Linear.Preconditioner.SMG.MaxIter = 2       ## Python syntax
```

integer **Solver.Linear.Preconditioner.SMG.NumPreRelax** 1 This key specifies the number of relaxations to take before coarsening in the specified preconditioner method. Note that this key is only relevant to the SMG multigrid preconditioner.

```
pfset Solver.Linear.Preconditioner.SMG.NumPreRelax    2        ## TCL syntax

<runname>.Solver.Linear.Preconditioner.SMG.NumPreRelax = 2     ## Python syntax
```

integer **Solver.Linear.Preconditioner.SMG.NumPostRelax** 1 This key specifies the number of relaxations to take after coarsening in the specified preconditioner method. Note that this key is only relevant to the SMG multigrid preconditioner.

```
pfset Solver.Linear.Preconditioner.SMG.NumPostRelax    0       ## TCL syntax

<runname>.Solver.Linear.Preconditioner.SMG.NumPostRelax = 0    ## Python syntax
```

string **Solver.Linear.Preconditioner.PFMG.RAPType** NonGalerkin For the PFMG solver, this key specifies the *Hypre* RAP type. Valid values are **Galerkin** or **NonGalerkin**

```
pfset Solver.Linear.Preconditioner.PFMG.RAPType    "Galerkin"     ## TCL syntax

<runname>.Solver.Linear.Preconditioner.PFMG.RAPType = "Galerkin"  ## Python syntax
```

logical **Solver.EvapTransFile** False This key specifies specifies that the Flux terms for Richards' equation are read in from a `.pfb` file. This file has [T^-1] units. Note this key is for a steady-state flux and should not be used in conjunction with the transient key below.

```
pfset Solver.EvapTransFile    True        ## TCL syntax

<runname>.Solver.EvapTransFile = True     ## Python syntax
```

logical **Solver.EvapTransFileTransient** False This key specifies specifies that the Flux terms for Richards' equation are read in from a series of flux `.pfb` file. Each file has $[T^-1]$ units. Note this key should not be used with the key above, only one of these keys should be set to `True` at a time, not both.

```
pfset Solver.EvapTransFileTransient    True        ## TCL syntax

<runname>.Solver.EvapTransFileTransient = True     ## Python syntax
```

string **Solver.EvapTrans.FileName** no default This key specifies specifies filename for the distributed `.pfb` file that contains the flux values for Richards' equation. This file has $[T^-1]$ units. For the steady-state option (*Solver.EvapTransFile*=**True*) this key should be the complete filename. For the transient option (*Solver.EvapTransFileTransient*=**True*) then the filename is a header and ParFlow will load one file per timestep, with the form `filename.00000.pfb`.

```
pfset Solver.EvapTrans.FileName   "evap.trans.test.pfb"        ## TCL syntax

<runname>.Solver.EvapTrans.FileName = "evap.trans.test.pfb"    ## Python syntax
```

string **Solver.LSM** none This key specifies whether a land surface model, such as `CLM`, will be called each solver timestep. Choices for this key include **none** and **CLM**. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.LSM "CLM"           ## TCL syntax

<runname>.Solver.LSM = "CLM"     ## Python syntax
```

6.35 Spinup Options

These keys allow for *reduced or dampened physics* during model spinup or initialization. They are **only** intended for these initialization periods, **not** for regular runtime.

integer **OverlandFlowSpinUp** 0 This key specifies that a *simplified* form of the overland flow boundary condition (Equation (4.15)) be used in place of the full equation. This formulation *removes lateral flow* and drives and ponded water pressures to zero. While this can be helpful in spinning up the subsurface, this is no longer coupled subsurface-surface flow. If set to zero (the default) this key behaves normally.

```
pfset OverlandFlowSpinUp   1         ## TCL syntax
<runname>.OverlandFlowSpinUp = 1     ## Python syntax
```

double **OverlandFlowSpinUpDampP1** 0.0 This key sets P_1 and provides exponential dampening to the pressure relationship in the overland flow equation by adding the following term: $P_2 * exp(\psi * P_2)$

```
pfset OverlandSpinupDampP1  10.0       ## TCL syntax
<runname>.OverlandSpinupDampP1 = 10.0  ## Python syntax
```

double **OverlandFlowSpinUpDampP2** 0.0 This key sets P_2 and provides exponential dampening to the pressure relationship in the overland flow equation adding the following term: $P_2 * exp(\psi * P_2)$

```
pfset OverlandSpinupDampP2  0.1        ## TCL syntax
<runname>.OverlandSpinupDampP2 = 0.1   ## Python syntax
```

6.36 CLM Solver Parameters

string **Solver.CLM.Print1dOut** False This key specifies whether the CLM one dimensional (averaged over each processor) output file is written or not. Choices for this key include True and False. Note that CLM must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.Print1dOut   False       ## TCL syntax
<runname>.Solver.CLM.Print1dOut = False   ## Python syntax
```

integer **Solver.CLM.IstepStart** 1 This key specifies the value of the counter, *istep* in CLM. This key primarily determines the start of the output counter for CLM. It is used to restart a run by setting the key to the ending step of the previous run plus one. Note that CLM must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.IstepStart     8761      ## TCL syntax
<runname>.Solver.CLM.IstepStart = 8761    ## Python syntax
```

String **Solver.CLM.MetForcing** no default This key specifies defines whether 1D (uniform over the domain), 2D (spatially distributed) or 3D (spatially distributed with multiple timesteps per .pfb forcing file) forcing data is used. Choices for this key are **1D**, **2D** and **3D**. This key has no default so the user *must* set it to 1D, 2D or 3D. Failure to set this key will cause CLM to still be run but with unpredictable values causing CLM to eventually crash. 1D meteorological forcing files are text files with single columns for each variable and each timestep per row, while 2D forcing files are distributed ParFlow binary files, one for each variable and timestep. File names are specified in the **Solver.CLM.MetFileName** variable below. Note that CLM must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.MetForcing   "2D"        ## TCL syntax
<runname>.Solver.CLM.MetForcing = "2D"    ## Python syntax
```

String **Solver.CLM.MetFileName** no default This key specifies defines the file name for 1D, 2D or 3D forcing data. 1D meteorological forcing files are text files with single columns for each variable and each timestep per row, while 2D and 3D forcing files are distributed ParFlow binary files, one for each variable and timestep (2D) or one for each variable and *multiple* timesteps (3D). Behavior of this key is different for 1D and 2D and 3D cases, as specified by the **Solver.CLM.MetForcing** key above. For 1D cases, it is the *FULL FILE NAME*. Note that in this configuration, this forcing file is **not** distributed, the user does not provide copies such as `narr.1hr.txt.0`, `narr.1hr.txt.1` for each processor. ParFlow only needs the single original file (*e.g.* `narr.1hr.txt`). For 2D cases, this key is the BASE FILE NAME for the 2D forcing files, currently set to NLDAS, with individual files determined as follows `NLDAS.<variable>.<time step>.pfb`. Where the `<variable>` is the forcing variable and `<timestep>` is the integer file counter corresponding to istep above. Forcing is needed for following variables:

DSWR:
Downward Visible or Short-Wave radiation $[W/m^2]$.

DLWR:
Downward Infa-Red or Long-Wave radiation $[W/m^2]$

APCP:
Precipitation rate $[mm/s]$

Temp:
Air temperature $[K]$

UGRD:
West-to-East or U-component of wind $[m/s]$

VGRD:
South-to-North or V-component of wind $[m/s]$

Press:
Atmospheric Pressure $[pa]$

SPFH:
Water-vapor specific humidity $[kg/kg]$

Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.MetFileName     "narr.1hr.txt"    ## TCL syntax
<runname>.Solver.CLM.MetFileName = "narr.1hr.txt"  ## Python syntax
```

String **Solver.CLM.MetFilePath** no default This key specifies defines the location of 1D, 2D or 3D forcing data. For 1D cases, this is the path to a single forcing file (*e.g.* `narr.1hr.txt`). For 2D and 3D cases, this is the path to the directory containing all forcing files. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.MetFilePath "path/to/met/forcing/data/"          ## TCL syntax
<runname>.Solver.CLM.MetFilePath = "path/to/met/forcing/data/"    ## Python syntax
```

integer **Solver.CLM.MetFileNT** no default This key specifies the number of timesteps per file for 3D forcing data.

```
pfset Solver.CLM.MetFileNT        24          ## TCL syntax
<runname>.Solver.CLM.MetFileNT = 24    ## Python syntax
```

string **Solver.CLM.ForceVegetation** False This key specifies whether vegetation should be forced in `CLM`. Currently this option only works for 1D and 3D forcings, as specified by the key `Solver.CLM.MetForcing`. Choices for this key include **True** and **False**. Forced vegetation variables are :

LAI:
Leaf Area Index $[-]$

SAI:
Stem Area Index $[-]$

Z0M:
Aerodynamic roughness length $[m]$

DISPLA:
Displacement height $[m]$

In the case of 1D meteorological forcings, `CLM` requires four files for vegetation time series and one vegetation map. The four files should be named respectively `lai.dat`, `sai.dat`, `z0m.dat`, `displa.dat`. They are ASCII files and contain 18 time-series columns (one per IGBP vegetation class, and each timestep per row). The vegetation map should be a properly distributed 2D ParFlow binary file (`.pfb`) which contains vegetation indices (from 1 to 18). The vegetation map filename is `veg_map.pfb`. ParFlow uses the vegetation map to pass to `CLM` a 2D map for each vegetation variable at each time step. In the case of 3D meteorological forcings, ParFlow expects four distincts properly distributed ParFlow binary file (`.pfb`), the third dimension being the timesteps. The files should be named `LAI.pfb`, `SAI.pfb`, `Z0M.pfb`, `DISPLA.pfb`. No vegetation map is needed in this case.

```
pfset Solver.CLM.ForceVegetation  True       ## TCL syntax
<runname>.Solver.CLM.ForceVegetation = True  ## Python syntax
```

string **Solver.WriteSiloCLM** False This key specifies whether the `CLM` writes two dimensional binary output files to a silo binary format. This data may be read in by VisIT and other visualization packages. Note that `CLM` and silo must be compiled and linked at runtime for this option to be active. These files are all written according to the standard format used for all ParFlow variables, using the *runname*, and *istep*. Variables are either two-dimensional or over the number of `CLM` layers (default of ten).

```
pfset Solver.WriteSiloCLM True          ## TCL syntax
<runname>.Solver.WriteSiloCLM = True    ## Python syntax
```

The output variables are:

`eflx_lh_tot` for latent heat flux total $[W/m^2]$ using the silo variable *LatentHeat*;

`eflx_lwrad_out` for outgoing long-wave radiation $[W/m^2]$ using the silo variable *LongWave*;

`eflx_sh_tot` for sensible heat flux total $[W/m^2]$ using the silo variable *SensibleHeat*;

`eflx_soil_grnd` for ground heat flux $[W/m^2]$ using the silo variable *GroundHeat*;

`qflx_evap_tot` for total evaporation $[mm/s]$ using the silo variable *EvaporationTotal*;

`qflx_evap_grnd` for ground evaporation without condensation $[mm/s]$ using the silo variable *EvaporationGround-NoSublimation*;

`qflx_evap_soi` for soil evaporation $[mm/s]$ using the silo variable *EvaporationGround*;

`qflx_evap_veg` for vegetation evaporation $[mm/s]$ using the silo variable *EvaporationCanopy*;

`qflx_tran_veg` for vegetation transpiration $[mm/s]$ using the silo variable *Transpiration*;

`qflx_infl` for soil infiltration $[mm/s]$ using the silo variable *Infiltration*;

`swe_out` for snow water equivalent $[mm]$ using the silo variable *SWE*;

`t_grnd` for ground surface temperature $[K]$ using the silo variable *TemperatureGround*; and

`t_soil` for soil temperature over all layers $[K]$ using the silo variable *TemperatureSoil*.

string **Solver.PrintCLM** False This key specifies whether the `CLM` writes two dimensional binary output files to a `PFB` binary format. Note that `CLM` must be compiled and linked at runtime for this option to be active. These files are all written according to the standard format used for all ParFlow variables, using the *runname*, and *istep*. Variables are either two-dimensional or over the number of `CLM` layers (default of ten).

```
pfset Solver.PrintCLM True          ## TCL syntax
<runname>.Solver.PrintCLM = True    ## Python syntax
```

The output variables are:

`eflx_lh_tot` for latent heat flux total $[W/m^2]$ using the silo variable *LatentHeat*;

`eflx_lwrad_out` for outgoing long-wave radiation $[W/m^2]$ using the silo variable *LongWave*;

`eflx_sh_tot` for sensible heat flux total $[W/m^2]$ using the silo variable *SensibleHeat*;

`eflx_soil_grnd` for ground heat flux $[W/m^2]$ using the silo variable *GroundHeat*;

`qflx_evap_tot` for total evaporation $[mm/s]$ using the silo variable *EvaporationTotal*;

`qflx_evap_grnd` for ground evaporation without sublimation $[mm/s]$ using the silo variable *EvaporationGroundNo-Sublimation*;

`qflx_evap_soi` for soil evaporation $[mm/s]$ using the silo variable *EvaporationGround*;

`qflx_evap_veg` for vegetation evaporation $[mm/s]$ using the silo variable *EvaporationCanopy*;

`qflx_tran_veg` for vegetation transpiration $[mm/s]$ using the silo variable *Transpiration*;

`qflx_infl` for soil infiltration $[mm/s]$ using the silo variable *Infiltration*;

`swe_out` for snow water equivalent $[mm]$ using the silo variable *SWE*;

`t_grnd` for ground surface temperature $[K]$ using the silo variable *TemperatureGround*; and

`t_soil` for soil temperature over all layers $[K]$ using the silo variable *TemperatureSoil*.

string **Solver.WriteCLMBinary** True This key specifies whether the `CLM` writes two dimensional binary output files in a generic binary format. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.WriteCLMBinary False          ## TCL syntax
<runname>.Solver.WriteCLMBinary = False    ## Python syntax
```

string **Solver.CLM.BinaryOutDir** True This key specifies whether the `CLM` writes each set of two dimensional binary output files to a corresponding directory. These directories my be created before ParFlow is run (using the tcl script, for example). Choices for this key include **True** and **False**. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.BinaryOutDir True         ## TCL syntax
<runname>.Solver.CLM.BinaryOutDir = True   ## Python syntax
```

These directories are:

`/qflx_top_soil` for soil flux;

`/qflx_infl` for infiltration;

`/qflx_evap_grnd` for ground evaporation;

`/eflx_soil_grnd` for ground heat flux;

`/qflx_evap_veg` for vegetation evaporation;

`/eflx_sh_tot` for sensible heat flux;

`/eflx_lh_tot` for latent heat flux;

`/qflx_evap_tot` for total evaporation;

`/t_grnd` for ground surface temperature;

`/qflx_evap_soi` for soil evaporation;

`/qflx_tran_veg` for vegetation transpiration;

`/eflx_lwrad_out` for outgoing long-wave radiation;

`/swe_out` for snow water equivalent; and

`/diag_out` for diagnostics.

string **Solver.CLM.CLMFileDir** no default This key specifies what directory all output from the `CLM` is written to. This key may be set to `"./"` or `""` to write output to the ParFlow run directory. This directory must be created before ParFlow is run. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.CLMFileDir "CLM_Output/"           ## TCL syntax
<runname>.Solver.CLM.CLMFileDir = "CLM_Output/"     ## Python syntax
```

integer **Solver.CLM.CLMDumpInterval** 1 This key specifies how often output from the `CLM` is written. This key is in integer multipliers of the `CLM` timestep. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.CLMDumpInterval 2           ## TCL syntax
<runname>.Solver.CLM.CLMDumpInterval = 2     ## Python syntax
```

string **Solver.CLM.EvapBeta** Linear This key specifies the form of the bare soil evaporation β parameter in `CLM`. The valid types for this key are **None**, **Linear**, **Cosine**.

None:
No beta formulation, $\beta = 1$.

Linear:
$\beta = \frac{\phi S - \phi S_{res}}{\phi - \phi S_{res}}$

Cosine:
$\beta = \frac{1}{2}(1 - \cos(\frac{(\phi - \phi S_{res})}{(\phi S - \phi S_{res})}\pi)$

Note that S_{res} is specified by the key `Solver.CLM.ResSat` below, that β is limited between zero and one and also that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.EvapBeta "Linear"           ## TCL syntax
<runname>.Solver.CLM.EvapBeta = "Linear"     ## Python syntax
```

double **Solver.CLM.ResSat** 0.1 This key specifies the residual saturation for the β function in `CLM` specified above. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.ResSat  0.15          ## TCL syntax
<runname>.Solver.CLM.ResSat = 0.15     ## Python syntax
```

string **Solver.CLM.VegWaterStress** Saturation This key specifies the form of the plant water stress function β_t parameter in `CLM`. The valid types for this key are **None**, **Saturation**, **Pressure**.

None:
No transpiration water stress formulation, $\beta_t = 1$.

Saturation:
$\beta_t = \frac{\phi S - \phi S_{wp}}{\phi S_{fc} - \phi S_{wp}}$

Pressure:
$\beta_t = \frac{P - P_{wp}}{P_{fc} - P_{wp}}$

Note that the wilting point, S_{wp} or p_{wp}, is specified by the key `Solver.CLM.WiltingPoint` below, that the field capacity, S_{fc} or p_{fc}, is specified by the key `Solver.CLM.FieldCapacity` below, that β_t is limited between zero and one and also that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.VegWaterStress  "Pressure"        ## TCL syntax
<runname>.Solver.CLM.VegWaterStress = "Pressure"   ## Python syntax
```

double **Solver.CLM.WiltingPoint** 0.1 This key specifies the wilting point for the β_t function in `CLM` specified above. Note that the units for this function are pressure $[m]$ for a **Pressure** formulation and saturation $[-]$ for a **Saturation** formulation. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.WiltingPoint  0.15       ## TCL syntax
<runname>.Solver.CLM.WiltingPoint = 0.15  ## Python syntax
```

double **Solver.CLM.FieldCapacity** 1.0 This key specifies the field capacity for the β_t function in `CLM` specified above. Note that the units for this function are pressure $[m]$ for a **Pressure** formulation and saturation $[-]$ for a **Saturation** formulation. Note that `CLM` must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.FieldCapacity  0.95        ## TCL syntax
<runname>.Solver.CLM.FieldCapacity = 0.95   ## Python syntax
```

string **Solver.CLM.IrrigationTypes** none This key specifies the form of the irrigation in `CLM`. The valid types for this key are **none**, **Spray**, **Drip**, **Instant**.

```
pfset Solver.CLM.IrrigationTypes "Drip"      ## TCL syntax
<runname>.Solver.CLM.IrrigationTypes "Drip"  ## Python syntax
```

string **Solver.CLM.IrrigationCycle** Constant This key specifies the cycle of the irrigation in `CLM`. The valid types for this key are **Constant**, **Deficit**. Note only **Constant** is currently implemented. Constant cycle applies irrigation each day from IrrigationStartTime to IrrigationStopTime in GMT.

```
pfset Solver.CLM.IrrigationCycle "Constant"        ## TCL syntax
<runname>.Solver.CLM.IrrigationCycle = "Constant"  ## Python syntax
```

double **Solver.CLM.IrrigationRate** no default This key specifies the rate of the irrigation in `CLM` in $[mm/s]$.

```
pfset Solver.CLM.IrrigationRate 10.         ## TCL syntax
<runname>.Solver.CLM.IrrigationRate = 10.   ## Python syntax
```

double **Solver.CLM.IrrigationStartTime** no default This key specifies the start time of the irrigation in `CLM` GMT.

```
pfset Solver.CLM.IrrigationStartTime 0800          ## TCL syntax
<runname>.Solver.CLM.IrrigationStartTime = 0800    ## Python syntax
```

double **Solver.CLM.IrrigationStopTime** no default This key specifies the stop time of the irrigation in `CLM` GMT.

```
pfset Solver.CLM.IrrigationStopTime 1200        ## TCL syntax
<runname>.Solver.CLM.IrrigationStopTime = 1200  ## Python syntax
```

double **Solver.CLM.IrrigationThreshold** 0.5 This key specifies the threshold value for the irrigation in `CLM`.

```
pfset Solver.CLM.IrrigationThreshold 0.2          ## TCL syntax
<runname>.Solver.CLM.IrrigationThreshold = 0.2    ## Python syntax
```

integer **Solver.CLM.ReuseCount** 1 How many times to reuse a `CLM` atmospheric forcing file input. For example timestep=1, reuse =1 is normal behavior but reuse=2 and timestep=0.5 subdivides the time step using the same `CLM` input for both halves instead of needing two files. This is particually useful for large, distributed runs when the user wants to run ParFlow at a smaller timestep than the `CLM` forcing. Forcing files will be re-used and total fluxes adjusted accordingly without needing duplicate files.

```
pfset Solver.CLM.ReuseCount      5     ## TCL syntax
<runname>.Solver.CLM.ReuseCount = 5    ## Python syntax
```

string **Solver.CLM.WriteLogs** True When **False**, this disables writing of the CLM output log files for each processor. For example, in the clm.tcl test case, if this flag is added **False**, `washita.output.txt.p` and `washita.para.out.dat.p` (were *p* is the processor #) are not created, assuming *washita* is the run name.

```
pfset Solver.CLM.WriteLogs    False       ## TCL syntax
<runname>.Solver.CLM.WriteLogs = False    ## Python syntax
```

string **Solver.CLM.WriteLastRST** False Controls whether CLM restart files are sequentially written or whether a single file *restart file name*.00000.*p* is overwritten each time the restart file is output, where *p* is the processor number. If "True" only one file is written/overwritten and if "False" outputs are written more frequently. Compatible with DailyRST and ReuseCount; for the latter, outputs are written every n steps where n is the value of ReuseCount.

```
pfset Solver.CLM.WriteLastRST   True      ## TCL syntax
<runname>.Solver.CLM.WriteLastRST = True  ## Python syntax
```

string **Solver.CLM.DailyRST** True Controls whether CLM writes daily restart files (default) or at every time step when set to False; outputs are numbered according to the istep from ParFlow. If **ReuseCount=n**, with n greater than 1, the output will be written every n steps (i.e. it still writes hourly restart files if your time step is 0.5 or 0.25, etc…). Fully compatible with **WriteLastRST=False** so that each daily output is overwritten to time 00000 in *restart file name*.00000.p where *p* is the processor number.

```
pfset Solver.CLM.DailyRST    False     ## TCL syntax
<runname>.Solver.CLM.DailyRST = False  ## Python syntax
```

string **Solver.CLM.SingleFile** False Controls whether ParFlow writes all `CLM` output variables as a single file per time step. When "True", this combines the output of all the CLM output variables into a special multi-layer PFB with the file extension ".C.pfb". The first 13 layers correspond to the 2-D CLM outputs and the remaining layers are the soil temperatures in each layer. For example, a model with 4 soil layers will create a SingleFile CLM output with 17 layers at each time step. The file pseudo code is given below in *ParFlow CLM Single Output Binary Files (.c.pfb)* and the variables and units are as specified in the multiple `PFB` and `SILO` formats as above.

```
pfset Solver.CLM.SingleFile   True        ## TCL syntax
<runname>.Solver.CLM.SingleFile = True    ## Python syntax
```

integer **Solver.CLM.RootZoneNZ** 10 This key sets the number of soil layers the ParFlow expects from `CLM`. It will allocate and format all the arrays for passing variables to and from `CLM` accordingly. This value now sets the CLM number as well so recompilation is not required anymore. Most likely the key `Solver.CLM.SoiLayer`, described below, will also need to be changed.

```
pfset Solver.CLM.RootZoneNZ      4     ## TCL syntax
<runname>.Solver.CLM.RootZoneNZ = 4    ## Python syntax
```

integer **Solver.CLM.SoiLayer** 7 This key sets the soil layer, and thus the soil depth, that `CLM` uses for the seasonal temperature adjustment for all leaf and stem area indices.

```
pfset Solver.CLM.SoiLayer      4    ## TCL syntax
<runname>.Solver.CLM.SoiLayer = 4   ## Python syntax
```

string **Solver.CLM.UseSlopeAspect** False This key specifies whether or not allows for the inclusion of slopes when determining solar zenith angles. Note that must be compiled and linked at runtime for this option to be active.

```
pfset Solver.CLM.UseSlopeAspect True         ## TCL syntax
<runname>.Solver.CLM.UseSlopeAspect = True   ## Python syntax
```

6.37 ParFlow NetCDF4 Parallel I/O

NetCDF4 parallel I/O is being implemented in ParFlow. As of now only output capability is implemented. Input functionality will be added in later version. Currently user has option of printing 3-D time varying pressure or saturation or both in a single NetCDF file containing multiple time steps. User should configure ParFlow(pfsimulatior part) `--with-netcdf` option and link the appropriate NetCDF4 library. Naming convention of output files is analogues to binary file names. Following options are available for NetCDF4 output along with various performance tuning options. User is advised to explore NetCDF4 chunking and ROMIO hints option for better I/O performance.

HDF5 Library version 1.8.16 or higher is required for NetCDF4 parallel I/O

integer **NetCDF.NumStepsPerFile** This key sets number of time steps user wishes to output in a NetCDF4 file. Once the time step count increases beyond this number, a new file is automatically created.

```
pfset NetCDF.NumStepsPerFile    5      ## TCL syntax
<runname>.NetCDF.NumStepsPerFile = 5   ## Python syntax
```

string **NetCDF.WritePressure** False This key sets pressure variable to be written in NetCDF4 file.

```
pfset NetCDF.WritePressure    True     ## TCL syntax
<runanme>.NetCDF.WritePressure = True  ## Python syntax
```

string **NetCDF.WriteSaturation** False This key sets saturation variable to be written in NetCDF4 file.

```
pfset NetCDF.WriteSaturation    True      ## TCL syntax
<runname>.NetCDF.WriteSaturation = True   ## Python syntax
```

string **NetCDF.WriteMannings** False This key sets Mannings coefficients to be written in NetCDF4 file.

```
pfset NetCDF.WriteMannings            True    ## TCL syntax
<runname>.NetCDF.WriteMannings = True  ## Python syntax
```

string **NetCDF.WriteSubsurface** False This key sets subsurface data (permeabilities, porosity, specific storage) to be written in NetCDF4 file.

```
pfset NetCDF.WriteSubsurface          True    ## TCL syntax
<runname>.NetCDF.WriteSubsurface  = True   ## Python syntax
```

string **NetCDF.WriteSlopes** False This key sets x and y slopes to be written in NetCDF4 file.

```
pfset NetCDF.WriteSlopes      True    ## TCL syntax
<runname>.NetCDF.WriteSlopes = True    ## Python syntax
```

string **NetCDF.WriteMask** False This key sets mask to be written in NetCDF4 file.

```
pfset NetCDF.WriteMask True         ## TCL syntax
<runname>.NetCDF.WriteMask        = True   ## Python syntax
```

string **NetCDF.WriteDZMultiplier** False This key sets DZ multipliers to be written in NetCDF4 file.

```
pfset NetCDF.WriteDZMultiplier True          ## TCL syntax
<runname>.NetCDF.WriteDZMultiplier = True    ## Python syntax
```

string **NetCDF.WriteEvapTrans** False This key sets Evaptrans to be written in NetCDF4 file.

```
pfset NetCDF.WriteEvapTrans True          ## TCL syntax
<runname>.NetCDF.WriteEvapTrans = True    ## Python syntax
```

string **NetCDF.WriteEvapTransSum** False This key sets Evaptrans sum to be written in NetCDF4 file.

```
pfset NetCDF.WriteEvapTransSum True          ## TCL syntax
<runname>.NetCDF.WriteEvapTransSum = True    ## Python syntax
```

string **NetCDF.WriteOverlandSum** False This key sets overland sum to be written in NetCDF4 file.

```
pfset NetCDF.WriteOverlandSum     True        ## TCL syntax
<runname>.NetCDF.WriteOverlandSum = True   ## Python syntax
```

string **NetCDF.WriteOverlandBCFlux** False This key sets overland bc flux to be written in NetCDF4 file.

```
pfset NetCDF.WriteOverlandBCFlux  True        ## TCL syntax
<runname>.NetCDF.WriteOverlandBCFlux = True  ## Python syntax
```

6.38 NetCDF4 Chunking

Chunking may have significant impact on I/O. If this key is not set, default chunking scheme will be used by NetCDF library. Chunks are hypercube(hyperslab) of any dimension. When chunking is used, chunks are written in single write operation which can reduce access times. For more information on chunking, refer to NetCDF4 user guide.

string **NetCDF.Chunking** False This key sets chunking for each time varying 3-D variable in NetCDF4 file.

```
pfset NetCDF.Chunking    True       ## TCL syntax
<runname>.NetCDF.Chunking = True    ## Python syntax
```

Following keys are used only when **NetCDF.Chunking** is set to true. These keys are used to set chunk sizes in x, y and z direction. A typical size of chunk in each direction should be equal to number of grid points in each direction for each processor. e.g. If we are using a grid of 400(x)X400(y)X30(z) with 2-D domain decomposition of 8X8, then each core has 50(x)X50(y)X30(z) grid points. These values can be used to set chunk sizes each direction. For unequal distribution, chunk sizes should as large as largest value of grid points on the processor. e.g. If one processor has grid distribution of 40(x)X40(y)X30(z) and another has 50(x)X50(y)X30(z), the later values should be used to set chunk sizes in each direction.

integer **NetCDF.ChunkX** None This key sets chunking size in x-direction.

```
pfset NetCDF.ChunkX    50     ## TCL syntax
<runname>.NetCDF.ChunkX = 50  ## Python syntax
```

integer **NetCDF.ChunkY** None This key sets chunking size in y-direction.

```
pfset NetCDF.ChunkY    50     ## TCL syntax
<runname>.NetCDF.ChunkY = 50  ## Python syntax
```

integer **NetCDF.ChunkZ** None This key sets chunking size in z-direction.

```
pfset NetCDF.ChunkZ    30        ## TCL syntax
<runname>.NetCDF.ChunkZ = 30     ## Python syntax
```

6.39 NetCDF4 Compression

integer **NetCDF.Compression** False This key enables in-transit deflate compression for all NetCDF variables using zlib. To use this feature, NetCDF4 v4.7.4 must be available, which supports the necessary parallel zlib compression. The compression quality can be influenced by the chunk sizes and the overall data distribution. Compressed variables in NetCDF files can be opened in serial mode also within older versions of NetCDF4.

```
pfset NetCDF.Compression True          ## TCL syntax
<runname>.NetCDF.Compression = True    ## Python syntax
```

integer **NetCDF.CompressionLevel** 1 This key sets the deflate compression level (if **NetCDF.Compression** is enabled), which influence the overall compression quality. zlib supports values between 0 (no compression), 1 (fastest compression) - 9 (slowest compression, smallest files).

```
pfset NetCDF.CompressionLevel 1           ## TCL syntax
<runname>.NetCDF.CompressionLevel = 1     ## Python syntax
```

6.40 ROMIO Hints

ROMIO is a poratable MPI-IO implementation developed at Argonne National Laboratory, USA. Currently it is released as a part of MPICH. ROMIO sets hints to optimize I/O operations for MPI-IO layer through MPI_Info object. This object is passed on to NetCDF4 while creating a file. ROMIO hints are set in a text file in “key” and “value” pair. *For correct settings contact your HPC site administrator*. As in chunking, ROMIO hints can have significant performance impact on I/O.

string **NetCDF.ROMIOhints** None This key sets ROMIO hints file to be passed on to NetCDF4 interface. If this key is set, the file must be present and readable in experiment directory.

```
pfset NetCDF.ROMIOhints "romio.hints"         ## TCL syntax
<runname>.NetCDF.ROMIOhints = "romio.hints"   ## Python syntax
```

An example ROMIO hints file looks as follows.

```
romio_ds_write disable
romio_ds_read disable
romio_cb_write enable
romio_cb_read enable
cb_buffer_size 33554432
```

6.41 Node Level Collective I/O

A node level collective strategy has been implemented for I/O. One process on each compute node gathers the data, indices and counts from the participating processes on same compute node. All the root processes from each compute node open a parallel NetCDF4 file and write the data. e.g. If ParFlow is running on 3 compute nodes where each node consists of 24 processors(cores); only 3 I/O streams to filesystem would be opened by each root processor each compute node. This strategy could be particularly useful when ParFlow is running on large number of processors and every processor participating in I/O may create a bottleneck. **Node level collective I/O is currently implemented for 2-D domain decomposition and variables Pressure and Saturation only. All the other ParFlow NetCDF output Tcl flags should be set to false(default value). CLM output is independently handled and not affected by this key. Moreover on speciality architectures, this may not be a portable feature. Users are advised to test this feature on their machine before putting into production.**

string **NetCDF.NodeLevelIO** False This key sets flag for node level collective I/O.

```
pfset NetCDF.NodeLevelIO   True        ## TCL syntax
<runname>.NetCDF.NodeLevelIO = True    ## Python syntax
```

6.42 NetCDF4 Initial Conditions: Pressure

Analogues to ParFlow binary files, NetCDF4 based option can be used to set the initial conditions for pressure to be read from an "nc" file containing single time step of pressure. The name of the variable in "nc" file should be "pressure". A sample NetCDF header of an initial condition file looks as follows. The names of the dimensions are not important. The order of dimensions is important e.g. *(time, lev, lat, lon) or (time,z, y, x)*

```
netcdf initial_condition {
dimensions:
  x = 200 ;
  y = 200 ;
  z = 40 ;
  time = UNLIMITED ; // (1 currently)
variables:
  double time(time) ;
  double pressure(time, z, y, x) ;
}
```

Node level collective I/O is currently not implemented for setting initial conditions.

string **ICPressure.Type** no default This key sets flag for initial conditions to be read from a NetCDF file.

```
pfset ICPressure.Type   "NCFile"        ## TCL syntax
pfset Geom.domain.ICPressure.FileName "initial_condition.nc" ## TCL syntax

<runname>.ICPressure.Type = "NCFile"    ## Python syntax
<runname>.Geom.domain.ICPressure.FileName = "initial_condition.nc" ## Python syntax
```

6.43 NetCDF4 Slopes

NetCDF4 based option can be used slopes to be read from an "nc" file containing single time step of slope values. The name of the variable in "nc" file should be "slopex" and "slopey" A sample NetCDF header of slope file looks as follows. The names of the dimensions are not important. The order of dimensions is important e.g. *(time, lat, lon) or (time, y, x)*

```
netcdf slopex {
dimensions:
  time = UNLIMITED ; // (1 currently)
  lon = 41 ;
  lat = 41 ;
variables:
  double time(time) ;
  double slopex(time, lat, lon) ;
}
netcdf slopey {
dimensions:
  time = UNLIMITED ; // (1 currently)
  lon = 41 ;
  lat = 41 ;
variables:
  double time(time) ;
  double slopey(time, lat, lon) ;
}
```

The two NetCDF files can be merged into one single file and can be used with tcl flags. The variable names should be exactly as mentioned above. Please refer to "slopes.nc" under Little Washita test case. **Node level collective I/O is currently not implemented for setting initial conditions.**

string **TopoSlopesX.Type** no default This key sets flag for slopes in x direction to be read from a NetCDF file.

```
pfset TopoSlopesX.Type "NCFile"              ## TCL syntax
pfset TopoSlopesX.FileName "slopex.nc"       ## TCL syntax

<runname>.TopoSlopesX.Type = "NCFile"        ## Python syntax
<runname>.TopoSlopesX.FileName = "slopex.nc" ## Python syntax
```

string **TopoSlopesY.Type** no default This key sets flag for slopes in y direction to be read from a NetCDF file.

```
pfset TopoSlopesY.Type "NCFile"              ## TCL syntax
pfset TopoSlopesy.FileName "slopey.nc"       ## TCL syntax

<runname>.TopoSlopesY.Type = "NCFile"        ## Python syntax
<runname>.TopoSlopesy.FileName = "slopey.nc" ## Python syntax
```

6.44 NetCDF4 Transient EvapTrans Forcing

Following keys can be used for NetCDF4 based transient evaptrans forcing. The file should contain forcing for all time steps. For a given time step, if the forcing is null, zero values could be filled for the given time step in the ".nc" file. The format of the sample file looks as follows. The names of the dimensions are not important. The order of dimensions is important e.g. *(time, lev, lat, lon) or (time,z, y, x)*

```
netcdf evap_trans {
dimensions:
  time = UNLIMITED ; // (1000 currently)
  x = 72 ;
  y = 72 ;
  z = 3 ;
variables:
  double evaptrans(time, z, y, x) ;
}
```

Node level collective I/O is currently not implemented for transient evaptrans forcing.

string **NetCDF.EvapTransFileTransient** False This key sets flag for transient evaptrans forcing to be read from a NetCDF file.

```
pfset NetCDF.EvapTransFileTransient True           ## TCL syntax
<runname>.NetCDF.EvapTransFileTransient = True     ## Python syntax
```

string **NetCDF.EvapTrans.FileName** no default This key sets the name of the NetCDF transient evaptrans forcing file.

```
pfset NetCDF.EvapTrans.FileName "evap_trans.nc"          ## TCL syntax
<runname>.NetCDF.EvapTrans.FileName = "evap_trans.nc"    ## Python syntax
```

6.45 NetCDF4 CLM Output

Similar to ParFlow binary and silo, following keys can be used to write output CLM variables in a single NetCDF file containing multiple time steps.

integer **NetCDF.CLMNumStepsPerFile** None This key sets number of time steps to be written to a single NetCDF file.

```
pfset NetCDF.CLMNumStepsPerFile 24           ## TCL syntax
<runname>.NetCDF.CLMNumStepsPerFile = 24     ## Python syntax
```

string **NetCDF.WriteCLM** False This key sets CLM variables to be written in a NetCDF file.

```
pfset NetCDF.WriteCLM True           ## TCL syntax
<runname>.NetCDF.WriteCLM = True     ## Python syntax
```

The output variables are:

`eflx_lh_tot` for latent heat flux total $[W/m^2]$ using the silo variable *LatentHeat*;

`eflx_lwrad_out` for outgoing long-wave radiation $[W/m^2]$ using the silo variable *LongWave*;

`eflx_sh_tot` for sensible heat flux total $[W/m^2]$ using the silo variable *SensibleHeat*;

`eflx_soil_grnd` for ground heat flux $[W/m^2]$ using the silo variable *GroundHeat*;

`qflx_evap_tot` for total evaporation $[mm/s]$ using the silo variable *EvaporationTotal*;

`qflx_evap_grnd` for ground evaporation without condensation $[mm/s]$ using the silo variable *EvaporationGroundNoSublimation*;

`qflx_evap_soi` for soil evaporation $[mm/s]$ using the silo variable *EvaporationGround*;

`qflx_evap_veg` for vegetation evaporation $[mm/s]$ using the silo variable *EvaporationCanopy*;

`qflx_tran_veg` for vegetation transpiration $[mm/s]$ using the silo variable *Transpiration*;

`qflx_infl` for soil infiltration $[mm/s]$ using the silo variable *Infiltration*;

`swe_out` for snow water equivalent $[mm]$ using the silo variable *SWE*;

`t_grnd` for ground surface temperature $[K]$ using the silo variable *TemperatureGround*; and

`t_soil` for soil temperature over all layers $[K]$ using the silo variable *TemperatureSoil*.

6.46 NetCDF4 CLM Input/Forcing

NetCDF based meteorological forcing can be used with following TCL keys. It is built similar to 2D forcing case for CLM with parflow binary files. All the required forcing variables must be present in one single NetCDF file spanning entire length of simulation. If the simulation ends before number of time steps in NetCDF forcing file, next cycle of simulation can be restarted with same forcing file provided it covers the time span of this cycle.

e.g. If the NetCDF forcing file contains 100 time steps and simulation CLM-ParFlow simulation runs for 10 cycles containing 10 time steps in each cycle, the same forcing file can be reused. The user has to set correct value for the key `Solver.CLM.IstepStart`

The format of input file looks as follows. The variable names should match exactly as follows. The names of the dimensions are not important. The order of dimensions is important e.g. *(time, lev, lat, lon) or (time,z, y, x)*

```
netcdf metForcing {
dimensions:
  lon = 41 ;
  lat = 41 ;
  time = UNLIMITED ; // (72 currently)
variables:
  double time(time) ;
  double APCP(time, lat, lon) ;
  double DLWR(time, lat, lon) ;
  double DSWR(time, lat, lon) ;
  double Press(time, lat, lon) ;
  double SPFH(time, lat, lon) ;
  double Temp(time, lat, lon) ;
  double UGRD(time, lat, lon) ;
  double VGRD(time, lat, lon) ;
```

Note: While using NetCDF based CLM forcing, `Solver.CLM.MetFileNT` **should be set to its default value of 1**

string **Solver.CLM.MetForcing** no default This key sets meteorological forcing to be read from NetCDF file.

```
pfset Solver.CLM.MetForcing "NC"          ## TCL syntax
<runname>.Solver.CLM.MetForcing = "NC"    ## Python syntax
```

Set the name of the input/forcing file as follows.

```
pfset Solver.CLM.MetFileName "metForcing.nc"          ## TCL syntax
<runname>.Solver.CLM.MetFileName = "metForcing.nc"    ## Python syntax
```

This file should be present in experiment directory. User may create soft links in experiment directory in case where data can not be moved.

6.47 NetCDF Testing Little Washita Test Case

The basic NetCDF functionality of output (pressure and saturation) and initial conditions (pressure) can be tested with following tcl script. CLM input/output functionality can also be tested with this case.

```
parflow/test/washita/tcl_scripts/LW_NetCDF_Test.tcl
```

This test case will be initialized with following initial condition file, slopes and meteorological forcing.

```
parflow/test/washita/parflow_input/press.init.nc
parflow/test/washita/parflow_input/slopes.nc
parflow/test/washita/clm_input/metForcing.nc
```

CHAPTER

SEVEN

PYTHON

The following sections provide a brief introduction to usage of the Python PFTools package, from installation to execution of a Python ParFlow run script.

7.1 PFTools

Welcome to Python PFTools. This is a Python package that creates a user-friendly Python interface for ParFlow. This package allows users to run ParFlow directly from a Python script, leveraging the power and accessibility of Python. More documentation for this package can be found here: https://pypi.org/project/pftools/

7.1.1 Installation

`pftools` can be installed with the following command:

```
pip install pftools[all]
```

The `[all]` argument will download the dependencies necessary for running ParFlow and fully employing the other tools within this package. `[all]` encompasses the subsets of dependencies, including:

- `[pfsol]`: installs the `imageio` package for handling image processing to assist some workflows to build ParFlow solid (.pfsol) files.
- `[io]`: installs the `numpy`, `xarray`, and `dask` packages for handling reading and storing of ParFlow binary (.pfb) data.
- `[fastio]`: installs the `numba` package for translating certain I/O operations into fast machine code.

If you would like to set up a virtual environment to install `pftools`, execute the following commands:

```
python3 -m venv py-env
source py-env/bin/activate
pip install pftools[all]
```

7.1.2 Execution

Command: `python3 /path/to/script/run_script.py [args]`

Usage:

```
run_script.py [-h] [--parflow-directory PARFLOW_DIRECTORY] [--parflow-version PARFLOW_
↪VERSION]
[--working-directory WORKING_DIRECTORY] [--skip-validation] [--dry-run] [--show-line-
↪error]
[--exit-on-error] [--write-yaml] [--validation-verbose] [-p P] [-q Q] [-r R]
```

Parflow run arguments:

- Optional arguments:

```
-h, --help            show help message and exit
```

- Parflow settings:

```
--parflow-directory PARFLOW_DIRECTORY
                Path to use for PARFLOW_DIR

--parflow-version PARFLOW_VERSION
                Override detected Parflow version
```

- Execution settings:

```
--working-directory WORKING_DIRECTORY
                Path to execution working directory

--skip-validation     Disable validation pass

--dry-run             Prevent execution
```

- Error handling settings:

```
--show-line-error     Show line error

--exit-on-error       Exit at error
```

- Additional output:

```
--write-yaml          Enable config to be written as YAML file

--validation-verbose    Prints validation results for all key/value pairs
```

- Parallel execution:

```
-p P
     P allocates the number of processes to the grid-cells in x (overrides Process.
↪Topology.P)
-q Q
     Q allocates the number of processes to the grid-cells in y (overrides Process.
```

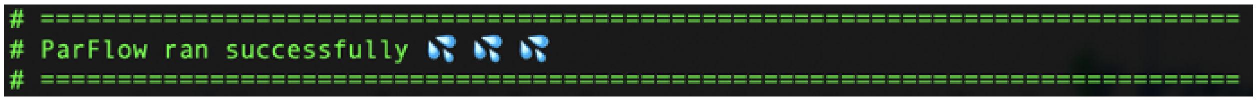
(continues on next page)

(continued from previous page)

```
↪Topology.Q)
-r R
     R allocates the number of processes to the grid-cells in z (overrides Process.
↪Topology.R)
```

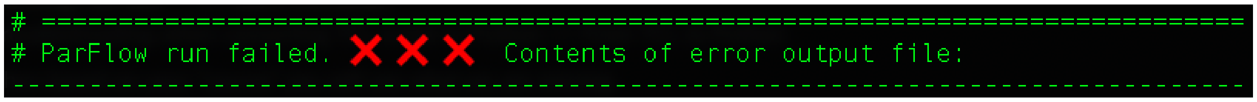
Output:

When executing ParFlow via the Python script using `run()`, you will get the following message if the ParFlow run succeeds:



```
# ==============================================================================
# ParFlow ran successfully 💦 💦 💦
# ==============================================================================
```

Or if it fails:



```
# ==============================================================================
# ParFlow run failed. ❌ ❌ ❌  Contents of error output file:
--------------------------------------------------------------------------------
```

This will be followed by the contents of the *runname.out.txt* file.

7.2 Run script

7.2.1 Anatomy

At the top of all the Python test scripts (located in parflow/test/python/) are something similar to the following lines:

```python
from parflow import Run
test_run = Run("test_run", __file__)
```

These lines import the Run class from the `parflow` module and create a new Run object called `test_run` (or whatever you want the run name to be). All the key/value pairs are set on this object.

7.2.2 Setting keys

The basic way to set keys in a Python script is assigning a variable to the Run object that you initialize at the beginning of your model, like shown:

```python
from parflow import Run
test_run = Run("test_run", __file__)

# Now that the Run object is initialized, you can set keys:
test_run.Process.Topology.P = 1
test_run.Process.Topology.Q = 1
test_run.Process.Topology.R = 1
```

Pretty simple! You can also create your user-defined keys, as shown below. Note that the following excerpts of code assume that you have already instantiated the Run object `test_run`.

```
test_run.GeomInput.Names = 'domain_input background_input'

# Defining the InputType and GeomName of the 'domain_input' that you already defined:
test_run.GeomInput.domain_input.InputType = 'Box'
test_run.GeomInput.domain_input.GeomName = 'domain'
```

Python PFTools requires that you define the user-defined input names (e.g., `GeomInput.Names`, `Cycle.Names`, `Phase.Names`) *before* you use them as part of a key name.

7.2.3 Valid key names

As a general rule, each "token" within a key name (e.g. `GeomInput` or `domain_input` in the prior example) must be a valid Python variable name. Information about valid Python variable names is here. This means that you can't use hyphens in your user-defined variables or use integers as tokens. However, if you absolutely *must* use non-Pythonic key names, there is a way. You can specify the token in brackets without the preceding decimal, as in the following example:

```
test_run.Patch['x-lower'].BCPressure.Type = 'FluxConst'
```

Specifying integer tokens (e.g., setting `Cell.0.dzScale.Value`) can be done in multiple ways. The preferred method is to use the token's "prefix", which is a character (alphanumeric or "_") that will always prefix that token. Right now, the prefixes for all the integer tokens is an underscore ("_"). However, as shown in the following example, these integer tokens can be set in multiple ways:

```
prefix.dzScale.nzListNumber = 6

# Here are four different ways to set integer values as part of a key name:
# 1) no bracket, no quotes, underscore
prefix.Cell._3.dzScale.Value = 1.000

# 2) bracket, quotes, underscore, no preceding decimal
prefix.Cell['_0'].dzScale.Value = 1.0

# 3) bracket, quotes, no underscore, no preceding decimal
prefix.Cell['1'].dzScale.Value = 1.00

# 4) bracket, no quotes, no underscore, no preceding decimal
prefix.Cell[2].dzScale.Value = 1.000
```

These will all write the key in the ParFlow database file in the correct format.

7.2.4 Setting keys and values with `pfset()`

The `pfset()` method does more than just allow you to set an individual key. You can set groups of keys at a time using the `hierarchical_map`, `flat_map`, or `yaml_content` arguments in the `pfset` method, as shown in the test file `$PARFLOW_SOURCE/test/python/new_features/pfset_test/pfset_test.py`:

```
#-----------------------------------------------------------------------------
# pfset: hierarchical_map
#-----------------------------------------------------------------------------

```

(continues on next page)

(continued from previous page)

```
pfset_test.pfset(hierarchical_map={
    'SpecificStorage': {
        'Type': 'Constant',
        'GeomNames': 'domain',
 }
})

constOne = {'Type': 'Constant', 'Value': 1.0}

pfset_test.Phase.water.Density.pfset(hierarchical_map=constOne)
pfset_test.Phase.water.Viscosity.pfset(flat_map=constOne)

#-----------------------------------------------------------------------------
# pfset: flat_map
#-----------------------------------------------------------------------------

pfset_test.pfset(flat_map={
    'Phase.Saturation.Type': 'VanGenuchten',
    'Phase.Saturation.GeomNames': 'domain',
})

pfset_test.Phase.pfset(flat_map={
    'RelPerm.Type': 'VanGenuchten',
    'RelPerm.GeomNames': 'domain',
})

#---------------------------------------------------------
# pfset: yaml_content
#---------------------------------------------------------

pfset_test.Geom.source_region.pfset(yaml_content='''
Lower:
    X: 65.56
    Y: 79.34
    Z: 4.5
Upper:
    X: 74.44
    Y: 89.99
    Z: 5.5
''')

pfset_test.Geom.concen_region.pfset(yaml_content='''
Lower:
    X: 60.0
    Y: 80.0
    Z: 4.0
Upper:
    X: 80.0
    Y: 100.0
    Z: 6.0
''')
```

Or, if you have a yaml file, you can use the `yaml_file` argument to read in a yaml file to set the keys:

```
#-----------------------------------------------------------
# pfset: yaml_file
#-----------------------------------------------------------

pfset_test.pfset(yaml_file='./BasicSettings.yaml')
pfset_test.pfset(yaml_file='./ComputationalGrid.yaml')
pfset_test.Geom.pfset(yaml_file='./GeomChildren.yaml')
```

This can make your run scripts more compact and readable.

7.2.5 Setting keys that aren't in the library with `pfset()`

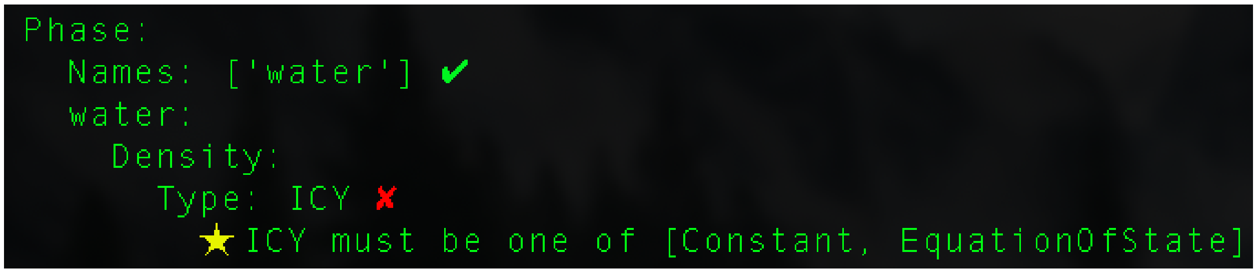
If you want to set a key in the Python script that's not already in the library, you have two options: 1) add the key to the library (see the documentation on "Contributing keys") or 2) using the `pfset(key, value)` method. `pfset(key, value)` allows the user to set a key (or token) `name` at any level with any `value`. Here are some examples from the test file `$PARFLOW_SOURCE/test/python/new_features/pfset_test/pfset_test.py`:

```
# Sets A.New.Key.Test = 'SomeSuperContent'
pfset_test.pfset(key='A.New.Key.Test', value='SomeSuperContent')

# Sets Process.Topology.Random.Path = 5
pfset_test.pfset(key='Process.Topology.Random.Path', value=5)

# Sets Process.Topology.Random.PathFromTopology = 6
pfset_test.Process.Topology.pfset(key='Random.PathFromTopology', value=6)

# Sets Process.Topology.P = 2
pfset_test.pfset(key='Process.Topology.P', value=2)

# Sets Process.Topology.Q = 2
pfset_test.Process.pfset(key='Topology.Q', value=3)

# Sets Process.Topology.R = 2
pfset_test.Process.Topology.pfset(key='R', value=4)

# Sets Process.Topology.Seb = 2
pfset_test.Process.Topology.pfset(key='Seb', value=5)
```

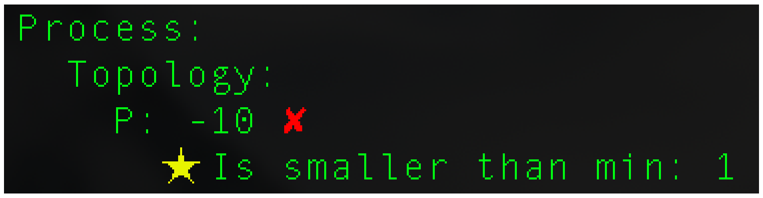
As you can see from the many examples here, you can use `pfset(key, value)` at any level of token within your key, and even set keys that already exist.

7.2.6 Key validation

An objective of the Python PFTools is to improve the error messages before and during a ParFlow run. The first step of this is validation. If you call the `validate()` method on your `Run` object with incorrect values set to a key, you will get a range of error messages like the following:

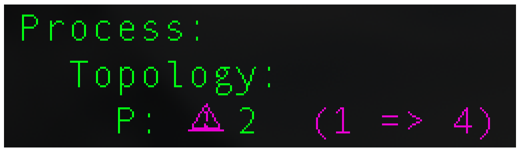


```
Phase:
  Names: ['water'] ✔
  water:
    Density:
      Type: ICY ✘
        ★ICY must be one of [Constant, EquationOfState]
```



```
Process:
  Topology:
    P: -10 ✘
      ★Is smaller than min: 1
```

You will also get a warning if you set a key multiple times, as shown:



```
Process:
  Topology:
    P: ⚠ 2  (1 => 4)
```

Here, `Process.Topology.P` was set three times: first to 1, then to 4, and finally to 2. Note: if you use the `pfset()` method to define a new key name, it will not throw an error in the validation.

7.2.7 Methods

Other methods that can be called on a Run object are shown below:

```python
from parflow import Run

# Instantiate a Run object
test_run = Run("test_run", __file__)

# Distribute a ParFlow binary file associated with a run
# P, Q, and R optional arguments override Process.Topology values
test_run.dist('test_slopes.pfb')

# Validate the values set to the keys of the Run object
test_run.validate()

# Write out key/value pairs to a file
test_run.write(file_format='pfidb')
test_run.write(file_format='yaml')
test_run.write(file_format='json')

# Write pfidb file and run ParFlow in the same directory as the script, skipping␣
↪validation
test_run.run(skip_validation=True)

# Clone the run into a new Run object
cloned_run = test_run.clone('cloned_run')
```

7.2.8 Full API

1. `runobj.validate(indent=1, verbose=False, enable_print=True)` - validates the values set to each key. Validation checks for:

   - Data type (int, float, string)
   - Appropriate range of value (e.g. saturation can't be less than zero!)
   - File availability
   - Duplicate values
   - Necessary module(s) installed
   - Key exists in working version of ParFlow

   The three optional arguments deal with printing the validation messages. `indent=1` is the tab length for each level of the hierarchy. The number of spaces that each level is indented is two times `indent` (so default is two spaces). `verbose=False`, if set to `True`, will print all key/value pairs in the run. Otherwise, `validate` will only print the key/value pairs with errors and their respective error messages. The runtime argument `--validation-verbose` is equivalent to setting `verbose=True`. `enable_print=True` defaults to printing all the validation messages. If set to `False`, no validation messages will be printed.

2. `runobj.write(file_name=None, file_format='pfidb')` - this will write the set of key/value pairs associated with the `runobj` in a specified format. The default `file_name` is the name of the `Run` object, and the default format is the ParFlow databse format. Other supported formats include *.yaml*, *.yml*, and *.json*.

3. `runobj.write_subsurface_table(file_name=None)` - this will write out a table with the subsurface properties assigned to each subsurface unit. If a file name is not specified, it will default to a *.csv* file using the name you set to your `Run` object at the top of the script, e.g., *default_richards_subsurface.csv*. More information is in the subsurface property tutorial.

4. `runobj.run(working_directory=None, skip_validation=False)` - this calls the `write()` method to write the set of key/value pairs to a ParFlow binary file. It also calls the `validate()` method if `skip_validation=False`. If `skip_validation=True`, it will skip the validation. This is equivalent to the `--skip-validation` runtime argument. Finally, the method will attempt to execute ParFlow. If `working_directory` is not given, `run()` defaults to writing all files in the directory of the Python script. The `working_directory` argument is equivalent to the `--working-directory` runtime argument.

5. `runobj.dist(pfb_file)` - distributes a given ParFlow binary file using the `parflowio` library with the given `Process.Topology.[P/Q/R]` values. The topology that the `dist()` method uses can be overwritten as in the above example. This will be covered in more detail in Tutorial 4.

6. `runobj.clone(name)` - clones the object `runobj` to a new object `name`. This makes it easy to develop ensembles of runs without having to reset all the keys and values.

7.2.9 Example

The default_richards.py test is a straightforward example of a run script. If you'd like to explore the other methods, copy this test to a local directory, and replace the last line (`drich.run`) with the following lines:

```
drich_2 = drich.clone('drich_2')
drich_2.Patch.left.BCPressure.alltime.Value = 6.0
drich_2.validate()
drich_2.write(file_format=‘yaml’)
drich_2.run(skip_validation=True)
```

Execute the script and look at what prints and which output files are created - explore to your heart's content!

7.3 Contributing keys

7.3.1 YAML definitions

The files in this directory are split up into groups to limit their length. Each ParFlow key comprises one or more tokens, separated by periods. In the YAML files, tokens are set up in a tabbed hierarchical structure, where each token is nested within the preceding token. Tokens are either static (starting with a capital letter) or dynamic (denoted by `.{dynamic_name}`, e.g. `geom_name` in `Geom.geom_name.Lower.X`). Leaf tokens are the tokens where the value is stored (e.g. `R` in `Process.Topology.R`). All other tokens are referred to as intermediate tokens.

Each token has one or more annotations associated with it, which fall into one of three categories, which are described below:

1. Generator annotations

The generator uses these annotations to generate the Python library and documentation

`__class__`

This is for adding dynamically defined tokens. The generator uses the *__class__* name to reference the location of a dynamically defined token. The *__class__* names usually end in *Item* to denote a dynamic token, e.g. *CycleItem* for the *.{cycle_name}* token in the key *Cycle.cycle_name.Names*.

`__from__`

This includes the source path of the dynamically defined token referenced in `__class__`. For example,

```
BCPressure:

    .{interval_name}:
        __class__: BCPressureIntervalItem
        __from__: /Cycle/{CycleItem}/Names
```

Here, `BCPressureIntervalItem` is the dynamically defined `.{interval_name}` token. The values of the `Cycle.cycle_name.Names` key generate these `.{interval_name}` tokens. The path to the `Cycle.cycle_name.Names` key is `/Cycle/{CycleItem}/Names`.

`__rst__`

This contains details to support the documentation. Arguments for this include:

- `name:` {string}: This argument will change the name of the key as it appears in the documentation.
- `skip:` {no arguments}: This will cause the key to not print in the documentation. This does not affect nested tokens.
- `warning:` {string}: This argument will add a special warning message to the documentation. This should be used for special cases, such as when a key must be set differently in Python as opposed to a TCL script.

__prefix__

This handles the tokens for key names with integers as tokens (e.g. `Cell.0.dzScale.Value`). Since Python does not recognize integers as a valid variable name, the user must specify a prefix to the integer. This can be any alphabetical character (upper or lower case) or an underscore. The specified prefix must be used to set the token within the key. For example, the prefix for `Cell.0.dzScale.Value` is an underscore, so you must define the key as `Cell._0.dzScale.Value`.

2. Key annotations

These annotations apply to the key itself, assisting documentation

help, __doc__

This contains the documentation for the key. `help` is used for leaf tokens, and `__doc__` is used for intermediate tokens.

__value__

This annotation applies to intermediate tokens that contain a value, but are not a leaf token (e.g. `Solver`). This will be treated as if it were a leaf token, including the value annotations that apply to the intermediate token.

3. Value annotations

These annotations apply to the value set to the key.

domains

This defines the domains that constrain the value of the key. The domains must include one or more of the following:

- `AddedInVersion`: This is for keys that have been added in a known version of ParFlow. This takes a string argument of the ParFlow version when the key was added, e.g. `'3.6.0'`.
- `AnyString`: This is for keys that must be a string. There are no arguments for this domain.
- `BoolDomain`: This is for keys that must be `True` or `False`. There are no arguments for this domain.
- `DeprecatedInVersion`: This is for keys that have been or will be deprecated in a known version of ParFlow. This takes a string argument of the ParFlow version when the key has been or will be deprecated, e.g., `'3.6.0'`.
- `DoubleValue`: This is for keys that must be a double. It takes up to two arguments: `min_value` for the minimum value, and `max_value` for the maximum value. Keys with a DoubleValue domain can also be integers.
- `EnumDomain`: This is for values that must be one of an enumerated list. This takes one argument, `enum_list`, that includes the list of acceptable values for the key. To accommodate instances where new options are added in new versions of ParFlow, `enum_list` can take an argument of a ParFlow version, which would include the list of acceptable values beginning with the specified version. See the `EnumDomain` for the `Patch.{patch_name}.BCPressure.Type` (*bconditions.yaml*) for an example.
- `IntValue`: This is for keys that must be an integer. It takes up to two arguments: `min_value` for the minimum value, and `max_value` for the maximum value.
- `MandatoryValue`: This is for keys that must be set for a ParFlow run. `MandatoryValue` does not take any arguments.

- `RemovedInVersion`: This is for keys that have been or will be removed in a known version of ParFlow. This takes a string argument of the ParFlow version when the key has been or will be removed, e.g. `'3.6.0'`.
- `RequiresModule`: This is for keys that must have a particular module installed or compiled to be a valid key (e.g., `Solver.CLM....`). This takes an argument of the required module in all caps, e.g., `RequiresModule: NETCDF`.
- **`ValidFile`: This is for keys which reference file names to make sure that the file exists. It can take two arguments: `working_directory`, for which you can specify the absolute path of the directory where your file is stored, `path_prefix_source`, for which you can specify the path to a key that defines the path**
  to the file (e.g. `Solver.CLM.MetFile`). If no arguments are provided, it will check your current working directory for the file.

`handlers`

This will transform inputs or help generate dynamically defined tokens within other keys based on the provided value for the key. Each argument is an updater that specifies where and how the value is used to create other tokens. An example from phase.yaml is below:

```
Phase:
    Names:
        handlers:
            PhaseUpdater:
                type: ChildrenHandler
                class_name: PhaseNameItem
                location: .
```

`PhaseUpdater` is the name of the handler. The arguments for the handler include `type`, `class_name`, and `location`. The most common option for `type` is `ChildrenHandler`. `class_name` corresponds to the `__class__` annotation of the dynamic token. In this example, `PhaseNameItem` is the `__class__` of the dynamic token `.{phase_name}`. `location` is the location of the token referenced in `class_name`. In this example, the Names token in `Phase.Names` is on the same level as the `.{phase_name}` in `Phase.phase_name`. This can also be an absolute path. See `handlers.py` for more on the other handlers.

`ignore`

Skip field exportation but allow to set other keys from it in a more convinient manner using some handler.

```
Solver:
    CLM:
        Input:
            Timing:
                StartDate:
                  help: >
                    [Type: string] Helper property that will set StartYear/StartMonth/
↪StartDay
                  ignore: _not_exported_
                  handlers:
                    FieldsUpdater:
                      type: SplitHandler
                      separator: /
                      convert: int
                      fields:
```

(continues on next page)

(continued from previous page)

```
                                - StartYear
                                - StartMonth
                                - StartDay
```

7.3.2 Steps to add a new key

1. Select the yaml file that most closely matches the key that you want to add. If your key is a token nested within an existing key, be sure to find which yaml file includes the parent token(s). For example, if you wanted to add the key `Solver.Linear.NewKey`, you would add it within the file *solver.yaml*.

2. Open the yaml file and navigate to the level within the hierarchy where you want to put your key. The structure of the yaml files is designed to be easy to follow, so it should be easy to find the level where you'd like to add your key. The indentation of these files is two spaces. Using our `Solver.Linear.NewKey` example, `Solver` is at the far left, `Linear` is two spaces (one tab) in, and you would add `NewKey` two more spaces in (two tabs). We suggest copying and pasting an existing key from the same level to make sure it's correct.

3. Fill in the details of your key. Again, this format is designed to be readable, so please refer to examples in the yaml files to guide you. The details you can include are listed in the section above.

4. Regenerate the Python keys using `make GeneratePythonKeys`.

You should see a longer message indicating an update that lists the overlapping classes, including the line `Defined ## fields were found`.

5. Test your new key. If you have an input script with the new key, you can run that to check whether it's working.

CHAPTER

EIGHT

MANIPULATING DATA: PFTOOLS

8.1 Introduction to the ParFlow TCL commands (PFTCL)

Several tools for manipulating data are provided in PFTCL command set. Tools can be accessed directly from the TCL shell or within a ParFlow input script. In both cases you must first load the ParFlow package into the TCL shell as follows:

```
#
# To Import the ParFlow TCL package
#
lappend auto_path $env(PARFLOW_DIR)/bin
package require parflow
namespace import Parflow::*
```

In addition to these methods xpftools provides GUI access to most of these features. However the simplest approach is generally to include the tools commands within a tcl script. The following section lists all of the available ParFlow TCL commands along with detailed instructions for their use. *PFTCL Commands* provides several examples of pre and post processing using the tools. In addition, a list of tools can be obtained by typing `pfhelp` into a TCL shell after importing ParFlow. Typing ¿`pfhelp`¿ followed by a command name will display a detailed description of the command in question.

8.2 PFTCL Commands

The tables that follow *8.1*, *8.2* and *8.3* provide a list of ParFlow commands with short descriptions grouped according to their function. The last two columns in this table indicate what examples from *Common examples using ParFlow TCL commands (PFTCL)*, if any, the command is used in and whether the command is compatible with a terrain following grid domain formulation.

Table 8.1: List of PFTools commands by function.

Name	Short Description	Examples	Co
pfhelp	Get help for PF Tools		X
Mathematical Operations			
pfcellsum	datasetx + datasety		X
pfcelldiff	datasetx - datasety		X
pfcellmult	datasetx * datasety		X
pfcelldiv	datasetx / datasety		X
pfcellsumconst	dataset + constant		X
p fcelldiffconst	dataset - constant		X

co

Table 8.1 – continued from previous page

Name	Short Description	Examples	Co
p fcellmultconst	dataset * constant		X
pfcelldivconst	dataset / constant		X
pfsum	Sum dataset	7, 9	X
pfdiffelt	Element difference		X
pfprintdiff	Print difference		X
pfmdiff	Calculate area where the difference between two datasets is less than a threshold		X
pfprintmdiff	Print the locations with differences greater than a minimum threshold		X
pfsavediff	Save the difference between two datasets		X
pfaxpy	y=alpha*x+y		X
pfgetstats	Calculate dataset statistics (min, max, mean, var, stdev)		X
pfprintstats	Print formatted statistics		X
pfstats	Calculate and print dataset statistics (min, max, mean, var, stdev)		X
Calculate physical parameters			
pfbfcvel	Calculate block face centered velocity		
pfcvel	Calculate Darcy velocity		
pfvvel	Calculate Darcy velocity at cell vertices		
pfvmag	Calculate velocity magnitude given components		
pfflux	Calculate Darcy flux		
pfhhead	Calculate hydraulic head	2	
pfphead	Calculate pressure head from hydraulic head		
pfsattrans	calculate saturated transmissivity		X
pfupstreamarea	Calculate upstream area		X
pfeff ectiverecharge	Calculate effective recharge		X
pfw atertabledepth	Calculate water table from saturation		X
pfhydrostatic	Calculate hydrostatic pressure field		
pfsub surfacestorage	Calculate total sub-surface storage	7	X
pfgwstorage	Calculate saturated subsurface storage		X
p fsurfacerunoff	Calculate total surface runoff	9	X
pf surfacestorage	Calculate total surface storage	8	X

Table 8.2: List of PFTools commands by function (cont.).

Name	Short Description	Examples	Compatible with TFG?
DEM Operations			
pfslopex	Calculate slopes in the x-direction	5	X
pfslopey	Calculate slope in the y-direction	5	X
pfchildD8	Calculate D8 child		X
pfsegmentD8	Calculate D8 segment lengths		X
pfslopeD8	Calculate D8 slopes		X
pfslopexD4	Calculate D4 slopes in the x-direction		X
pfslopeyD4	Calculate D4 slopes in the y-direction		X
pffillflats	Fill DEM flats	5	X
pfmovingavgdem	Fill dem sinks with moving average		X
pfpitfilldem	Fill sinks in the dem using iterative pitfilling routine	5	X
pfflintslawfit	Calculate Flint's Law parameters		X
pfflintslaw	Smooth DEM using Flints Law		X
pffl intslawbybasin	Smooth DEM using Flints Law by basin		X
Topmodel functions			
pftopodeficit	Calculate TOPMODEL water deficit		X

continues on next page

Table 8.2 – continued from previous page

Name	Short Description	Examples	Compatible with TFG?
pftopoindex	Calculate topographic index		X
pftopowt	Calculate watertable based on topographic index		X
pftoporecharge	Calculate effective recharge		X
Domain Operations			
p fcomputedomain	Compute domain mask	3	X
pfcomputetop	Compute domain top	3, 6, 8, 9	X
pfextracttop	Extract domain top	6	X
p fcomputebottom	Compute domain bottom	3	X
pfsetgrid	Set grid	5	X
pfgridtype	Set grid type		X
pfgetgrid	Return grid information		X
pfgetelt	Extract element from domain	10	X
pfe xtract2Ddomain	Build 2D domain		X
pfenlargebox	Compute expanded dataset		X
pfgetsubbox	Return subset of data		X
pfprintdomain	Print domain	3	X
pfbuilddomain	Build a subgrid array from a ParFlow database		X
Dataset operations			
pflistdata	Return dataset names and labels		X
pfgetlist	Return dataset descriptions		X
pfprintlist	Print list of datasets and their labels		X
pfnewlabel	Change dataset label		X
pfnewdata	Create new dataset		X
pfprintgrid	Print grid		X
pfnewgrid	Set grid for new dataset		X
pfdelete	Delete dataset		X
pfreload	Reload dataset		X
pfreloadall	Reload all current datasets		X
pfprintdata	Print all elements of a dataset		X
pfprintelt	Print a single element		X

Table 8.3: List of PFTools commands by function (cont.).

Name	Short Description	Examples	Compatible with TFG?
File Operations			
pfload	Load file	All	X
pfloadsds	Load Scientific Data Set from HDF file		X
pfdist	Distribute files based on processor topology	4	X
pfdistondomain	Distribute files based on domain		X
pfundist	Undistribute files		X
pfsave	Save dataset	1,2,5,6	X
pfsavesds	Save dataset in an HDF format		X
pfvtksave	Save dataset in VTK format using DEM	X	X
pfwritedb	Write the settings for a PF run to a database		X
Solid file operations			
pfpatchysolid	Build a solid file between two complex surfaces and assign user-defined patches around the edges		X
pfs olidfmt-convert	Converts back and forth between ascii and binary formats for solid files		X

Detailed descriptions of every command are included below in alphabetical order. Note that the required inputs are listed following each command. Commands that perform operations on data sets will require an identifier for each data set it takes as input. Inputs listed in square brackets are optional and do not need to be provided.

```
pfaxpy alpha x y
```

This command computes y = alpha*x+y where alpha is a scalar and x and y are identifiers representing data sets. No data set identifier is returned upon successful completion since data set y is overwritten.

```
pfbfcvel conductivity phead
```

This command computes the block face centered flow velocity at every grid cell. Conductivity and pressure head data sets are given as arguments. The output includes x, y, and z velocity components that are appended to the Tcl result.

```
pfbuilddomain database
```

This command builds a subgrid array given a ParFlow database that contains the domain parameters and the processor topology.

```
pfcelldiff datasetx datasety mask
```

This command computes cell-wise differences of two datasets (diff=datasetx-datasety). This is the difference at each individual cell, not over the domain. Datasets must have the same dimensions.

```
pfcelldiffconst dataset constant mask
```

This command subtracts a constant value from each (active) cell of dataset (dif=dataset - constant).

```
pfcelldiv datasetx datasety mask
```

This command computes the cell-wise quotient of datasetx and datasety (div = datasetx/datasety). This is the quotient at each individual cell. Datasets must have the same dimensions.

```
pfcelldivconst dataset constant mask
```

This command divides each (active) cell of dataset by a constant (div=dataset/constant).

```
pfcellmult datasetx datasety mask
```

This command computes the cell-wise product of datasetx and datasety (mult = datasetx * datasety). This is the product at each individual cell. Datasets must have the same dimensions.

```
pfcellmultconst dataset constant mask
```

This command multiplies each (active) cell of dataset by a constant (mult=dataset * constant).

```
pfcellsum datasetp datasetq mask
```

This command computes the cellwise sum of two datasets (i.e., the sum at each individual cell, not the sum over the domain). Datasets must have the same dimensions.

```
pfcellsumconst dataset constant mask
```

This command adds the value of constant to each (active) cell of dataset.

```
pfchildD8 dem
```

This command computes the unique D8 child for all cells. Child[i,j] is the elevation of the cell to which [i,j] drains (i.e. the elevation of [i,j]'s child). If [i,j] is a local minima the child elevation set the elevation of [i,j].

```
pfcomputebottom mask
```

This command computes the bottom of the domain based on the mask of active and inactive zones. The identifier of the data set created by this operation is returned upon successful completion.

```
pfcomputedomain top bottom
```

This command computes a domain based on the top and bottom data sets. The domain built will have a single subgrid per processor that covers the active data as defined by the top and botttom. This domain will more closely follow the topology of the terrain than the default single computation domain.

A typical usage pattern for this is to start with a mask file (zeros in inactive cells and non-zero in active cells), create the top and bottom from the mask, compute the domain and then write out the domain. Refer to example number 3 in the following section.

```
pfcomputetop mask
```

This command computes the top of the domain based on the mask of active and inactive zones. This is the land-surface in `clm` or overland flow simulations. The identifier of the data set created by this operation is returned upon successful completion.

```
pfcvel conductivity phead
```

This command computes the Darcy velocity in cells for the conductivity data set represented by the identifier 'conductivity' and the pressure head data set represented by the identifier 'phead'. (note: This "cell" is not the same as the grid cells; its corners are defined by the grid vertices.) The identifier of the data set created by this operation is returned upon successful completion.

```
pfdelete dataset
```

This command deletes the data set represented by the identifier 'dataset'. This command can be useful when working with multiple datasets / time series, such as those created when many timesteps of a file are loaded and processed. Deleting these datasets in between reads can help with tcl memory management.

```
pfdiffelt datasetp datasetq i j k digits [zero]
```

This command returns the difference of two corresponding coordinates from 'datasetp' and 'datasetq' if the number of digits in agreement (significant digits) differs by more than 'digits' significant digits and the difference is greater than the absolute zero given by 'zero'.

```
pfdist [options] filename
```

Distribute the file onto the virtual file system. This utility must be used to create files which ParFlow can use as input. ParFlow uses a virtual file system which allows each node of the parallel machine to read from the input file independently. The utility does the inverse of the pfundist command. If you are using a ParFlow binary file for input you should do a pfdist just before you do the pfrun. This command requires that the processor topology and computational grid be set in the input file so that it knows how to distribute the data. Note that the old syntax for pfdist required the NZ key be set to 1 to indicate a two dimensional file but this can now be specified manually when pfdist is called by using the optional argument -nz followed by the number of layers in the file to be distributed, then the filename. If the -nz argument is absent the NZ key is used by default for the processor topology.

For example,

```
pfdist -nz 1 slopex.pfb
```

```
pfdistondomain filename domain
```

Distribute the file onto the virtual file system based on the domain provided rather than the processor topology as used by pfdist. This is used by the SAMRAI version of which allows for a more complicated computation domain specification with different sized subgrids on each processor and allows for more than one subgrid per processor. Frequently this will be used with a domain created by the pfcomputedomain command.

```
pfeffectiverecharge precip et slopex slopey dem
```

This command computes the effective recharge at every grid cell based on total precipitation minus evapotranspiration (P-ET)in the upstream area. Effective recharge is consistent with TOPMODEL definition, NOT local P-ET. Inputs are total annual (or average annual) precipitation (precip) at each point, total annual (or average annual) evapotranspiration (ET) at each point, slope in the x direction, slope in the y direction and elevation.

```
pfenlargebox dataset sx sy sz
```

This command returns a new dataset which is enlarged to be of the new size indicated by sx, sy and sz. Expansion is done first in the z plane, then y plane and x plane.

```
pfextract2Ddomain domain
```

This command builds a 2D domain based off a 3D domain. This can be used for a pfdistondomain command for Parflow 2D data (such as slopes and soil indices).

```
pfextracttop top data
```

This command computes the top of the domain based on the top of the domain and another dataset. The identifier of the data set created by this operation is returned upon successful completion.

```
pffillflats dem
```

This command finds the flat regions in the DEM and eliminates them by bilinearly interpolating elevations across flat region.

```
pfflintslaw dem c p
```

This command smooths the digital elevation model dem according to Flints Law, with Flints Law parameters specified by c and p, respectively. Flints Law relates the slope magnitude at a given cell to its upstream contributing area: S = c*A**p. In this routine, elevations at local minima retain the same value as in the original dem. Elevations at all other cells are computed by applying Flints Law recursively up each drainage path, starting at its terminus (a local minimum) until a drainage divide is reached. Elevations are computed as:

dem[i,j] = dem[child] + c*(A[i,j]**p)*ds[i,j]

where child is the D8 child of [i,j] (i.e., the cell to which [i,j] drains according to the D8 method); ds[i,j] is the segment length between the [i,j] and its child; A[i,j] is the upstream contributing area of [i,j]; and c and p are constants.

```
pfflintslawbybasin dem c0 p0 maxiter
```

This command smooths the digital elevation model (dem) using the same approach as “pfflints law”. However here the c and p parameters are fit for each basin separately. The Flint’s Law parameters are calculated for the provided digital elevation model dem using the iterative Levenberg-Marquardt method of non-linear least squares minimization, as in “pfflintslawfit”. The user must provide initial estimates of c0 and p0; results are not sensitive to these initial values. The user must also specify the maximum number of iterations as maxiter.

```
pfflintslawfit dem c0 p0 maxiter
```

This command fits Flint’s Law parameters c and p for the provided digital elevation model dem using the iterative Levenberg-Marquardt method of non-linear least squares minimization. The user must provide initial estimates of c0 and p0; results are not sensitive to these initial values. The user must also specify the maximum number of iterations as maxiter. Final values of c and p are printed to the screen, and a dataset containing smoothed elevation values is returned. Smoothed elevations are identical to running pfflintslaw for the final values of c and p. Note that dem must be a ParFlow dataset and must have the correct grid information – dx, dy, nx, and ny are used in parameter estimation and Flint’s Law calculations. If gridded elevation values are read in from a text file (e.g., using pfload’s simple ascii format), grid information must be specified using the pfsetgrid command.

```
pfflux conductivity hhead
```

This command computes the net Darcy flux at vertices for the conductivity data set ‘conductivity’ and the hydraulic head data set given by ‘hhead’. An identifier representing the flux computed will be returned upon successful completion.

```
pfgetelt dataset i j k
```

This command returns the value at element (i,j,k) in data set ‘dataset’. The i, j, and k above must range from 0 to (nx - 1), 0 to (ny - 1), and 0 to (nz - 1) respectively. The values nx, ny, and nz are the number of grid points along the x, y, and z axes respectively. The string ‘dataset’ is an identifier representing the data set whose element is to be retrieved.

```
pfgetgrid dataset
```

This command returns a description of the grid which serves as the domain of data set ‘dataset’. The format of the description is given below.

-
  ```
  (nx, ny, nz)
  ```

  The number of coordinates in each direction.

-
  ```
  (x, y, z)
  ```

  The origin of the grid.

-
  ```
  (dx, dy, dz)
  ```

  The distance between each coordinate in each direction.

The above information is returned in the following Tcl list format: nx ny nz x y z dx dy dz

```
pfgetlist dataset
```

This command returns the name and description of a dataset if an argument is provided. If no argument is given, then all of the data set names followed by their descriptions is returned to the TCL interpreter. If an argument (dataset) is given, it should be the it should be the name of a loaded dataset.

```
pfgetstats dataset
```

This command calculates the following statistics for the data set represented by the identifier *dataset*: minimum, maximum, mean, sum, variance, and standard deviation.

```
pfgetsubbox dataset il jl kl iu ju ku
```

This command computes a new dataset with the subbox starting at il, jl, kl and going to iu, ju, ku.

```
pfgridtype gridtype
```

This command sets the grid type to either cell centered if 'gridtype' is set to 'cell' or vetex centered if 'gridtype' is set to 'vertex'. If no new value for 'gridtype' is given, then the current value of 'gridtype' is returned. The value of 'gridtype' will be returned upon successful completion of this command.

```
pfgwstorage mask porosity pressure saturation specific_storage
```

This command computes the sub-surface water storage (compressible and incompressible components) based on mask, porosity, saturation, storativity and pressure fields, similar to pfsubsurfacestorage, but only for the saturated cells.

```
pfhelp [command]
```

This command returns a list of pftools commands. If a command is provided it gives a detailed description of the command and the necessary inputs.

```
pfhhead phead
```

This command computes the hydraulic head from the pressure head represented by the identifier 'phead'. An identifier for the hydraulic head computed is returned upon successful completion.

```
pfhydrostatic wtdepth top mask
```

Compute hydrostatic pressure field from water table depth

```
pflistdata dataset
```

This command returns a list of pairs if no argument is given. The first item in each pair will be an identifier representing the data set and the second item will be that data set's label. If a data set's identifier is given as an argument, then just that data set's name and label will be returned.

```
pfload [file format] filename
```

Loads a dataset into memory so it can be manipulated using the other utilities. A file format may preceed the filename in order to indicate the file's format. If no file type option is given, then the extension of the filename is used to determine the default file type. An identifier used to represent the data set will be returned upon successful completion.

File type options include:

-
  ```
  pfb
  ```

  ParFlow binary format. Default file type for files with a '.pfb' extension.

-
  ```
  pfsb
  ```

  ParFlow scattered binary format. Default file type for files with a '.pfsb' extension.

-
  ```
  sa
  ```

  ParFlow simple ASCII format. Default file type for files with a '.sa' extension.

-
  ```
  sb
  ```

  ParFlow simple binary format. Default file type for files with a '.sb' extension.

-
  ```
  silo
  ```

  Silo binary format. Default file type for files with a '.silo' extension.

-
  ```
  rsa
  ```

  ParFlow real scattered ASCII format. Default file type for files with a '.rsa' extension

```
pfloadsds filename dsnum
```

This command is used to load Scientific Data Sets from HDF files. The SDS number 'dsnum' will be used to find the SDS you wish to load from the HDF file 'filename'. The data set loaded into memory will be assigned an identifier which will be used to refer to the data set until it is deleted. This identifier will be returned upon successful completion of the command.

```
pfmdiff datasetp datasetq digits [zero]
```

If 'digits' is greater than or equal to zero, then this command computes the grid point at which the number of digits in agreement (significant digits) is fewest and differs by more than 'digits' significant digits. If 'digits' is less than zero, then the point at which the number of digits in agreement (significant digits) is minimum is computed. Finally, the maximum absolute difference is computed. The above information is returned in a Tcl list of the following form: mi mj mk sd adiff

Given the search criteria, (mi, mj, mk) is the coordinate where the minimum number of significant digits 'sd' was found and 'adiff' is the maximum absolute difference.

```
pfmovingaveragedem dem wsize maxiter
```

This command fills sinks in the digital elevation model dem by a standard iterative moving-average routine. Sinks are identified as cells with zero slope in both x- and y-directions, or as local minima in elevation (i.e., all adjacent cells have higher elevations). At each iteration, a moving average is taken over a window of width wsize around each remaining sink; sinks are thus filled by averaging over neighboring cells. The procedure continues iteratively until all sinks are filled or the number of iterations reaches maxiter. For most applications, sinks should be filled prior to computing slopes (i.e., prior to executing pfslopex and pfslopey).

```
pfnewdata {nx ny nz} {x y z} {dx dy dz} label
```

This command creates a new data set whose dimension is described by the lists nx ny nz, x y z, and dx dy dz. The first list, describes the dimensions, the second indicates the origin, and the third gives the length intervals between each coordinate along each axis. The 'label' argument will be the label of the data set that gets created. This new data set that is created will have all of its data points set to zero automatically. An identifier for the new data set will be returned upon successful completion.

```
pfnewgrid {nx ny nz} {x y z} {dx dy dz} label
```

Create a new data set whose grid is described by passing three lists and a label as arguments. The first list will be the number of coordinates in the x, y, and z directions. The second list will describe the origin. The third contains the intervals between coordinates along each axis. The identifier of the data set created by this operation is returned upon successful completion.

```
pfnewlabel dataset newlabel
```

This command changes the label of the data set 'dataset' to 'newlabel'.

```
pfphead hhead
```

This command computes the pressure head from the hydraulic head represented by the identifier 'hhead'. An identifier for the pressure head is returned upon successful completion.

```
pfpatchysolid -top topdata -bot botdata -msk emaskdata [optional args]
```

Creates a solid file with complex upper and lower surfaces from a top surface elevation dataset (topdata), a bottom elevation dataset (botdata), and an enhanced mask dataset (emaskdata) all of which must be passed as handles to 2-d datasets that share a common size and origin. The solid is built as the volume between the top and bottom surfaces using the mask to deactivate other regions. The "enhanced mask" used here is a gridded dataset containing integers where all active cells have values of one but inactive cells may be given a positive integer value that identifies a patch along the model edge or the values may be zero. Any mask cell with value 0 is omitted from the active domain and *is not* written to a patch. If an active cell is adjacent to a non-zero mask cell, the face between the active and inactive cell is assigned to the patch with the integer value of the adjacent inactive cell. Bottom and Top patches are always written for every active cell and the West, East, South, and North edges are written automatically anytime active cells touch the edges of the input dataset(s). Up to 30 user defined patches can be specified using arbitrary integer values that are *greater than* 1. Note that the -msk flag may be omitted and doing so will make every cell active.

The -top and -bot flags, and -msk if it is used, MUST each be followed by the handle for the relevant dataset. Optional argument flag-name pairs include:

- -pfsol <file name>.pfsol (or -pfsolb <file name>.pfsolb)
- -vtk <file name>.vtk
- -sub

where <file name> is replaced by the desired text string. The -pfsolb option creates a compact binary solid file; pfsolb cannot currently be read directly by ParFlow but it can be converted with *pfsolidfmtconvert* and full support is under development. If -pfsol (or -pfsolb) is not specified the default name "SolidFile.pfsol" will be used. If -vtk is omitted, no vtk file will be created. The vtk attributes will contain mean patch elevations and patch IDs from the enhanced mask. Edge patch IDs are shown as negative values in the vtk. The patchysolid tool also outputs the list of the patch names in the order they are written, which can be directly copied into a ParFlow TCL script for the list of patch names. The -sub option writes separate patches for each face (left,right,front,back), which are indicated in the output patch write order list.

Assuming $Msk, $Top, and $Bot are valid dataset handles from pfload, two valid examples are:

```
pfpatchysolid -msk $Msk -top $Top -bot $Bot -pfsol "MySolid.pfsol" -vtk "MySolid.vtk"
pfpatchysolid -bot $Bot -top $Top -vtk "MySolid.vtk" -sub
```

Note that all flag-name pairs may be specified in any order for this tool as long as the required argument immediately follows the flag. To use with a terrain following grid, you will need to subtract the surface elevations from the top and bottom datasets (this makes the top flat) then add back in the total thickness of your grid, which can be done using “pfcelldiff” and “pfcellsumconst”.

```
pfpitfilldem dem dpit maxiter
```

This command fills sinks in the digital elevation model dem by a standard iterative pit-filling routine. Sinks are identified as cells with zero slope in both x- and y-directions, or as local minima in elevation (i.e., all adjacent neighbors have higher elevations). At each iteration, the value dpit is added to all remaining sinks. The procedure continues iteratively until all sinks are filled or the number of iterations reaches maxiter. For most applications, sinks should be filled prior to computing slopes (i.e., prior to executing pfslopex and pfslopey).

```
pfprintdata dataset
```

This command executes ‘pfgetgrid’ and ‘pfgetelt’ in order to display all the elements in the data set represented by the identifier ‘dataset’.

```
pfprintdiff datasetp datasetq digits [zero]
```

This command executes ‘pfdiffelt’ and ‘pfmdiff’ to print differences to standard output. The differences are printed one per line along with the coordinates where they occur. The last two lines displayed will show the point at which there is a minimum number of significant digits in the difference as well as the maximum absolute difference.

```
pfprintdomain domain
```

This command creates a set of TCL commands that setup a domain as specified by the provided domain input which can be then be written to a file for inclusion in a Parflow input script. Note that this kind of domain is only supported by the SAMRAI version of Parflow.

```
pfprintelt i j k dataset
```

This command prints a single element from the provided dataset given an i, j, k location.

```
pfprintgrid dataset
```

This command executes pfgetgrid and formats its output before printing it on the screen. The triples (nx, ny, nz), (x, y, z), and (dx, dy, dz) are all printed on seperate lines along with labels describing each.

```
pfprintlist [dataset]
```

This command executes pflistdata and formats the output of that command. The formatted output is then printed on the screen. The output consists of a list of data sets and their labels one per line if no argument was given or just one data set if an identifier was given.

```
pfprintmdiff datasetp datasetq digits [zero]
```

This command executes ‘pfmdiff’ and formats that command’s output before displaying it on the screen. Given the search criteria, a line displaying the point at which the difference has the least number of significant digits will be displayed. Another line displaying the maximum absolute difference will also be displayed.

```
printstats dataset
```

This command executes ‘pfstats’ and formats that command’s output before printing it on the screen. Each of the values mentioned in the description of ‘pfstats’ will be displayed along with a label.

```
pfreload dataset
```

This argument reloads a dataset. Only one arguments is required, the name of the dataset to reload.

```
pfreloadall
```

This command reloads all of the current datasets.

```
pfsattrans mask perm
```

Compute saturated transmissivity for all [i,j] as the sum of the permeability[i,j,k]*dz within a column [i,j]. Currently this routine uses dz from the input permeability so the dz in permeability must be correct. Also, it is assumed that dz is constant, so this command is not compatible with variable dz.

```
pfsave dataset -filetype filename
```

This command is used to save the data set given by the identifier ‘dataset’ to a file ‘filename’ of type ‘filetype’ in one of the ParFlow formats below.

File type options include:

- pfb ParFlow binary format.
- sa ParFlow simple ASCII format.
- sb ParFlow simple binary format.
- silo Silo binary format.
- vis Vizamrai binary format.

```
pfsavediff datasetp datasetq digits [zero] -file filename
```

This command saves to a file the differences between the values of the data sets represented by ‘datasetp’ and ‘datasetq’ to file ‘filename’. The data points whose values differ in more than ‘digits’ significant digits and whose differences are greater than ‘zero’ will be saved. Also, given the above criteria, the minimum number of digits in agreement (significant digits) will be saved.

If ‘digits’ is less than zero, then only the minimum number of significant digits and the coordinate where the minimum was computed will be saved.

In each of the above cases, the maximum absolute difference given the criteria will also be saved.

```
pfsavesds dataset -filetype filename
```

This command is used to save the data set represented by the identifier ‘dataset’ to the file ‘filename’ in the format given by ‘filetype’.

The possible HDF formats are:

- -float32
- -float64
- -int8

- -uint8
- -int16
- -uint16
- -int32
- -uint32

```
pfsegmentD8 dem
```

This command computes the distance between the cell centers of every parent cell [i,j] and its child cell. Child cells are determined using the eight-point pour method (commonly referred to as the D8 method) based on the digital elevation model dem. If [i,j] is a local minima the segment length is set to zero.

```
pfsetgrid {nx ny nz} {x0 y0 z0} {dx dy dz} dataset
```

This command replaces the grid information of dataset with the values provided.

```
pfslopeD8 dem
```

This command computes slopes according to the eight-point pour method (commonly referred to as the D8 method) based on the digital elevation model dem. Slopes are computed as the maximum downward gradient between a given cell and it's lowest neighbor (adjacent or diagonal). Local minima are set to zero; where local minima occur on the edge of the domain, the 1st order upwind slope is used (i.e., the cell is assumed to drain out of the domain). Note that dem must be a ParFlow dataset and must have the correct grid information – dx and dy both used in slope calculations. If gridded elevation values are read in from a text file (e.g., using pfload's simple ascii format), grid information must be specified using the pfsetgrid command. It should be noted that ParFlow uses slopex and slopey (NOT D8 slopes!) in runoff calculations.

```
pfslopex dem
```

This command computes slopes in the x-direction using 1st order upwind finite differences based on the digital elevation model dem. Slopes at local maxima (in x-direction) are calculated as the maximum downward gradient to an adjacent neighbor. Slopes at local minima (in x-direction) do not drain in the x-direction and are therefore set to zero. Note that dem must be a ParFlow dataset and must have the correct grid information – dx in particular is used in slope calculations. If gridded elevation values are read from a text file (e.g., using pfload's simple ascii format), grid inforamtion must be specified using the pfsetgrid command.

```
pfslopexD4 dem
```

This command computes the slope in the x-direction for all [i,j] using a four point (D4) method. The slope is set to the maximum downward slope to the lowest adjacent neighbor. If [i,j] is a local minima the slope is set to zero (i.e. no drainage).

```
pfslopey dem
```

This command computes slopes in the y-direction using 1st order upwind finite differences based on the digital elevation model dem. Slopes at local maxima (in y-direction) are calculated as the maximum downward gradient to an adjacent neighbor. Slopes at local minima (in y-direction) do not drain in the y-direction and are therefore set to zero. Note that dem must be a ParFlow dataset and must have the correct grid information - dy in particular is used in slope calculations. If gridded elevation values are read in from a text file (e.g., using pfload's simple ascii format), grid information must be specified using the pfsetgrid command.

```
pfslopeyD4 dem
```

This command computes the slope in the y-direction for all [i,j] using a four point (D4) method. The slope is set to the maximum downward slope to the lowest adjacent neighbor. If [i,j] is a local minima the slope is set to zero (i.e. no drainage).

```
pfsolidfmtconvert filename1 filename2
```

This command converts solid files back and forth between the ascii .pfsol format and the binary .pfsolb format. The tool automatically detects the conversion mode based on the extensions of the input file names. The *filename1* is the name of source file and *filename2* is the target output file to be created or overwritten. Support to directly use a binary solid (.pfsolb) is under development but this allows a significant reduction in file sizes.

For example, to convert from ascii to binary, then back to ascii:

```
pfsolidfmtconvert "MySolid.pfsol" "MySolid.pfsolb"
pfsolidfmtconvert "MySolid.pfsolb" "NewSolid.pfsol"
```

```
pfstats dataset
```

This command prints various statistics for the data set represented by the identifier 'dataset'. The minimum, maximum, mean, sum, variance, and standard deviation are all computed. The above values are returned in a list of the following form: min max mean sum variance (standard deviation)

```
pfsubsurfacestorage mask porosity pressure saturation specific_storage
```

This command computes the sub-surface water storage (compressible and incompressible components) based on mask, porosity, saturation, storativity and pressure fields. The equations used to calculate this quantity are given in *Water Balance*. The identifier of the data set created by this operation is returned upon successful completion.

```
pfsum dataset
```

This command computes the sum over the domain of the dataset.

```
pfsurfacerunoff top slope_x slope_y  mannings pressure
```

This command computes the surface water runoff (out of the domain) based on a computed top, pressure field, slopes and mannings roughness values. This is integrated along all domain boundaries and is calculated at any location that slopes at the edge of the domain point outward. This data is in units of $[L^3T^{-1}]$ and the equations used to calculate this quantity are given in *Water Balance*. The identifier of the data set created by this operation is returned upon successful completion.

```
pfsurfacestorage top pressure
```

This command computes the surface water storage (ponded water on top of the domain) based on a computed top and pressure field. The equations used to calculate this quantity are given in *Water Balance*. The identifier of the data set created by this operation is returned upon successful completion.

```
pftopodeficit profile m trans dem slopex slopey recharge ssat sres porosity mask
```

Compute water deficit for all [i,j] based on TOPMODEL/topographic index. For more details on methods and assumptions refer to toposlopes.c in pftools.

```
pftopoindex dem sx sy
```

Compute topographic index for all [i,j]. Here topographic index is defined as the total upstream area divided by the contour length, divided by the local slope. For more details on methods and assumptions refer to toposlopes.c in pftools.

```
pftoporecharge riverfile nriver  trans dem sx sy
```

Compute effective recharge at all [i,j] over upstream area based on topmodel assumptions and given list of river points. Notes: See detailed notes in toposlopes.c regarding assumptions, methods, etc. Input Notes: nriver is an integer (number of river points) river is an array of integers [nriver][2] (list of river indices, ordered from outlet to headwaters) is a Databox of saturated transmissivity dem is a Databox of elevations at each cell sx is a Databox of slopes (x-dir) – lets you use processed slopes! sy is a Databox of slopes (y-dir) – lets you use processed slopes!

```
pftopowt deficit porosity ssat sres mask top wtdepth
```

Compute water depth from column water deficit for all [i,j] based on TOPMODEL/topographic index.

```
pfundist filename, pfundist runname
```

The command undistributes a ParFlow output file. ParFlow uses a distributed file system where each node can write to its own file. The pfundist command takes all of these individual files and collapses them into a single file.

The arguments can be a runname or a filename. If a runname is given then all of the output files associated with that run are undistributed.

Normally this is done after every pfrun command.

```
pfupstreamarea slope_x slope_y
```

This command computes the upstream area contributing to surface runoff at each cell based on the x and y slope values provided in datasets `slope_x` and `slope_y`, respectively. Contributing area is computed recursively for each cell; areas are not weighted by slope direction. Areas are returned as the number of upstream (contributing) cells; to compute actual area, simply multiply by the cell area (dx*dy).

```
pfvmag datasetx datasety datasetz
```

This command computes the velocity magnitude when given three velocity components. The three parameters are identifiers which represent the x, y, and z components respectively. The identifier of the data set created by this operation is returned upon successful completion.

```
pfvtksave dataset filetype filename [options]
```

This command loads PFB or SILO output, reads a DEM from a file and generates a 3D VTK output field from that ParFlow output.

The options: Any combination of these can be used and they can be specified in any order as long as the required elements immediately follow each option.

-var specifies what the variable written to the dataset will be called. This is followed by a text string, like “Pressure” or “Saturation” to define the name of the data that will be written to the VTK. If this isn’t specified, you’ll get a property written to the file creatively called “Variable”. This option is ignored if you are using -clmvtk since all its variables are predefined.

-dem specifies that a DEM is to be used. The argument following -dem MUST be the handle of the dataset containing the elevations. If it cannot be found, the tool ignores it and reverts to non-dem mode. If the nx and ny dimensions of the grids don’t match, the tool will error out. This option shifts the layers so that the top of the domain coincides with the land surface defined by the DEM. Regardless of the actual number of layers in the DEM file, the tool only uses the elevations in the top layer of this dataset, meaning a 1-layer PFB can be used.

-flt tells the tool to write the data as type float instead of double. Since the VTKs are really only used for visualization, this reduces the file size and speeds up plotting.

-tfg causes the tool to override the specified dz in the dataset PFB and uses a user specified list of layer thicknesses instead. This is designed for terrain following grids and can only be used in conjunction with a DEM. The argument following the flag is a text string containing the number of layers and the dz list of actual layer thicknesses (not dz multipliers) for each layer from the bottom up such as: -tfg “5 200.0 1.0 0.7 0.2 0.1” Note that the quotation marks around the list are necessary.

Example:

```
file copy -force CLM_dem.cpfb CLM_dem.pfb

set CLMdat [pfload -pfb clm.out.clm_output.00005.C.pfb]
set Pdat [pfload -pfb clm.out.press.00005.pfb]
set Perm [pfload -pfb clm.out.perm_x.pfb]
set DEMdat [pfload -pfb CLM_dem.pfb]

set dzlist "10 6.0 5.0 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5"

pfvtksave $Pdat -vtk "CLM.out.Press.00005a.vtk" -var "Press"
pfvtksave $Pdat -vtk "CLM.out.Press.00005b.vtk" -var "Press" -flt
pfvtksave $Pdat -vtk "CLM.out.Press.00005c.vtk" -var "Press" -dem $DEMdat
pfvtksave $Pdat -vtk "CLM.out.Press.00005d.vtk" -var "Press" -dem $DEMdat -flt
pfvtksave $Pdat -vtk "CLM.out.Press.00005e.vtk" -var "Press" -dem $DEMdat -flt -tfg
↪$dzlist
pfvtksave $Perm -vtk "CLM.out.Perm.00005.vtk" -var "Perm" -flt -dem $DEMdat -tfg $dzlist

pfvtksave $CLMdat -clmvtk "CLM.out.CLM.00005.vtk" -flt
pfvtksave $CLMdat -clmvtk "CLM.out.CLM.00005.vtk" -flt -dem $DEMdat

pfvtksave $DEMdat -vtk "CLM.out.Elev.00000.vtk" -flt -var "Elevation" -dem $DEMdat
```

```
pfvvel conductivity phead
```

This command computes the Darcy velocity in cells for the conductivity data set represented by the identifier ‘conductivity’ and the pressure head data set represented by the identifier ‘phead’. The identifier of the data set created by this operation is returned upon successful completion.

```
pfwatertabledepth top saturation
```

This command computes the water table depth (distance from top to first cell with saturation = 1). The identifier of the data set created by this operation is returned upon successful completion.

```
pfwritedb runname
```

This command writes the settings of parflow run to a pfidb database that can be used to run the model at a later time. In general this command is used in lieu of the pfrun command.

8.3 Common examples using ParFlow TCL commands (PFTCL)

This section contains some brief examples of how to use the pftools commands (along with standard *TCL* commands) to postprocess data.

Load a file as one format and write as another format.

```
set press [pfload harvey_flow.out.press.pfb]
pfsave $press -sa harvey_flow.out.sa

#####################################################################
# Also note that PFTCL automatically assigns
#identifiers to each data set it stores. In this
# example we load the pressure file and assign
#it the identifier press. However if you
#read in a file called foo.pfb into a TCL shell
#with assigning your own identifier, you get
#the following:

#parflow> pfload foo.pfb
#dataset0

# In this example, the first line is typed in by the
#user and the second line is printed out
#by PFTCL. It indicates that the data read
#from file foo.pfb is associated with the
#identifier dataset0.
```

Load pressure-head output from a file, convert to head-potential and write out as a new file.

```
set press [pfload harvey_flow.out.press.pfb]
set head [pfhhead $press]
pfsave $head -pfb harvey_flow.head.pfb
```

Build a SAMARI compatible domain decomposition based off of a mask file

```
#---------------------------------------------------------
# This example script takes 3 command line arguments
# for P,Q,R and then builds a SAMRAI compatible
# domain decomposition based off of a mask file.
#---------------------------------------------------------

# Processor Topology
set P [lindex $argv 0]
set Q [lindex $argv 1]
set R [lindex $argv 2]
pfset Process.Topology.P $P
pfset Process.Topology.Q $Q
pfset Process.Topology.R $R

# Computational Grid
pfset ComputationalGrid.Lower.X -10.0
pfset ComputationalGrid.Lower.Y 10.0
pfset ComputationalGrid.Lower.Z 1.0
```

(continues on next page)

(continued from previous page)

```
pfset ComputationalGrid.DX 8.8888888888888893
pfset ComputationalGrid.DY 10.666666666666666
pfset ComputationalGrid.DZ 1.0

pfset ComputationalGrid.NX 10
pfset ComputationalGrid.NY 10
pfset ComputationalGrid.NZ 8

# Calculate top and bottom and build domain
set mask [pfload samrai.out.mask.pfb]
set top [pfcomputetop $mask]
set bottom [pfcomputebottom $mask]

set domain [pfcomputedomain $top $bottom]
set out [pfprintdomain $domain]
set grid\_file [open samrai_grid.tcl w]

puts $grid_file $out
close $grid_file

#-----------------------------------------------------------
# The resulting TCL file samrai_grid.tcl may be read into
# a Parflow input file using ¿¿source samrai_grid.tcl¿¿.
#-----------------------------------------------------------
```

Distributing input files before running [dist example]

```
#----------------------------------------------------------
# A common problem for new ParFlow users is to
# distribute slope files using
# the 3-D computational grid that is
# set at the begging of a run script.
# This results in errors because slope
# files are 2-D.
# To avoid this problem the computational
# grid should be reset before and after
# distributing slope files. As follows:
#-----------------------------------------------------------

#First set NZ to 1 and distribute the 2D slope files
pfset ComputationalGrid.NX                40
pfset ComputationalGrid.NY                40
pfset ComputationalGrid.NZ                1
pfdist slopex.pfb
pfdist slopey.pfb

#Reset NZ to the correct value and distribute any 3D inputs
pfset ComputationalGrid.NX                40
pfset ComputationalGrid.NY                40
pfset ComputationalGrid.NZ                50
pfdist IndicatorFile.pfb
```

Calculate slopes from an elevation file

```
#Read in DEM
set dem [pfload -sa dem.txt]
pfsetgrid {209 268 1} {0.0 0.0 0.0} {100 100 1.0} $dem

# Fill flat areas (if any)
set flatfill [pffillflats $dem]

# Fill pits (if any)
set  pitfill [pfpitfilldem $flatfill 0.01 10000]

# Calculate Slopes
set  slope_x [pfslopex $pitfill]
set  slope_y [pfslopey $pitfill]

# Write to output...
pfsave $flatfill -silo klam.flatfill.silo
pfsave $pitfill  -silo klam.pitfill.silo
pfsave $slope_x  -pfb  klam.slope_x.pfb
pfsave $slope_y  -pfb  klam.slope_y.pfb
```

Calculate and output the *subsurface storage* in the domain at a point in time.

```
set saturation [pfload runname.out.satur.00001.silo]
set pressure [pfload runname.out.press.00001.silo]
set specific_storage [pfload runname.out.specific_storage.silo]
set porosity [pfload runname.out.porosity.silo]
set mask [pfload runname.out.mask.silo]

set subsurface_storage [pfsubsurfacestorage $mask $porosity \
$pressure $saturation $specific_storage]
set total_subsurface_storage [pfsum $subsurface_storage]
puts [format "Subsurface storage\t\t\t\t : %.16e" $total_subsurface_storage]
```

Calculate and output the *surface storage* in the domain at a point in time.

```
set pressure [pfload runname.out.press.00001.silo]
set mask [pfload runname.out.mask.silo]
set top [pfcomputetop $mask]
set surface_storage [pfsurfacestorage $top $pressure]
set total_surface_storage [pfsum $surface_storage]
puts [format "Surface storage\t\t\t\t : %.16e" $total_surface_storage]
```

Calculate and output the runoff out of the *entire domain* over a timestep.

```
set pressure [pfload runname.out.press.00001.silo]
set slope_x [pfload runname.out.slope_x.silo]
set slope_y [pfload runname.out.slope_y.silo]
set mannings [pfload runname.out.mannings.silo]
set mask [pfload runname.out.mask.silo]
set top [pfcomputetop $mask]

set surface_runoff [pfsurfacerunoff $top $slope_x $slope_y $mannings $pressure]
```

(continues on next page)

(continued from previous page)

```
set total_surface_runoff [expr [pfsum $surface_runoff] * [pfget TimeStep.Value]]
puts [format "Surface runoff from pftools\t\t\t : %.16e" $total_surface_runoff]
```

Calculate overland flow at a point using *Manning's* equation

```
#Set the location
set Xloc 2
set Yloc 2
set Zloc 50  #This should be a z location on the surface of your domain

#Set the grid dimension and Mannings roughness coefficient
set dx  1000.0
set n   0.000005

#Get the slope at the point
set slopex   [pfload runname.out.slope_x.pfb]
set slopey   [pfload runname.out.slope_y.pfb]
set sx1 [pfgetelt $slopex $Xloc $Yloc 0]
set sy1 [pfgetelt $slopey $Xloc $Yloc 0]
set S [expr ($sx**2+$sy**2)**0.5]

#Get the pressure at the point
set press [pfload runname.out.press.00001.pfb]
set P [pfgetelt $press $Xloc $Yloc $Zloc]

#If the pressure is less than zero set to zero
if {$P < 0} { set P 0 }
set QT [expr ($dx/$n)*($S**0.5)*($P**(5./3.))]
puts $QT
```

CHAPTER

NINE

TUTORIALS

The following sections include tutorials for using various functionalities contained within Python PFTools.

9.1 From TCL to Python

Welcome to the tutorial for the Python pftools. You will need the following to fully follow this tutorial:

- Python >= 3.6
- ParFlow installed and running, with the correct `$PARFLOW_DIR` environment variable established (You can check this by running `echo $PARFLOW_DIR` in your terminal)

The commands in the tutorial assume that you are running a bash shell in Linux or MacOS.

9.1.1 Virtual environment setup

In this first tutorial, we will set up a virtual environment with pftools and its dependencies before importing a TCL file, converting it to Python, and running ParFlow.

First, let's set an environment variable for the newly cloned repo:

```
export PARFLOW_SOURCE=/path/to/new/parflow/
```

Now, set up a virtual environment and install pftools:

```
python3 -m venv tutorial-env
source tutorial-env/bin/activate
pip install pftools[all]
```

Test your pftools installation:

```
python3 $PARFLOW_SOURCE/test/python/base_3d/default_richards/default_richards.py
```

The run should execute successfully, printing the message `ParFlow ran successfully`.

9.1.2 From TCL to Python file

Great, now you have a working ParFlow interface! Next, create a new directory and import a TCL file (example here drawn from the ParFlow TCL tests):

```
mkdir -p pftools_tutorial/tcl_to_py
cd pftools_tutorial/tcl_to_py
cp $PARFLOW_SOURCE/test/default_richards.tcl .
```

You can use our `tcl2py` tool to convert the TCL script to a Python script using the following command:

```
python3 -m parflow.cli.tcl2py -i default_richards.tcl
```

The converter gets you most of the way there, but there are a few things you'll have to change by hand. Open and edit the new `.py` file that you have generated and change the lines that need to be changed. If you are following this example, you will need to edit the `Process.Topology` values, the `GeomInput.Names` values, and comment out the two `Solver.Linear.Preconditioner.MGSemi` keys, as shown here:

```
default_richards.Process.Topology.P = 1
default_richards.Process.Topology.Q = 1
default_richards.Process.Topology.R = 1
...

default_richards.GeomInput.Names = 'domain_input background_input source_region_input \
         concen_region_input'
...

# default_richards.Solver.Linear.Preconditioner.MGSemi.MaxIter = 1
# default_richards.Solver.Linear.Preconditioner.MGSemi.MaxLevels = 100
```

Once you have edited your Python script, you can run it like you would any other Python script:

```
python3 default_richards.py
```

Voilà! You have now successfully converted your first ParFlow TCL script to Python. In the next tutorial, we'll get more advanced to leverage the many other features in the Python PFTools. Onward!

9.1.3 Troubleshooting when converting TCL script to Python

Although the tutorial above (hopefully) went without a hitch, you may not always be so lucky. For those instances, Python PFTools has a tool that allows you to sort two *.pfidb* files to determine any discrepancies between two files. This is especially useful when comparing an existing TCL script's generated file to its Python-generated equivalent. First, you must sort each of the *.pfidb* files, using the following command:

```
python3 -m parflow.cli.pfdist_sort -i /path/to/file.pfidb -o /tmp/sorted.pfidb
```

`/path/to/file.pfidb` is the path to the existing (input, denoted by the `-i`) *.pfidb* file, and `/tmp/sorted.pfidb` is the file path where you want the sorted output (denoted by the `-o`) *.pfidb* file to be written.

Once you have the newly sorted files, you can compare them using one of many methods of file comparison, such as `diff`:

```
diff /path/to/from_tcl_sorted.pfidb /path/to/from_py_sorted.pfidb
```

You'll likely see some subtle format differences between the TCL- and Python-generated files (decimal printing, etc.). Most of these do not affect the execution of ParFlow.

9.2 Filesystem

9.2.1 Introduction

Files in ParFlow are like sand at the beach: everywhere. Python's native modules offer plenty of methods to handle files and directories, but they can be inconvenient when dealing with a ParFlow runs. Fortunately, Python PFTools has some helpful functions to deal with the ParFlow run working directory.

For example, let's pretend you want to automatically create a sub-directory for your run while copying some data files into it at run time based on where your run script lives. You can simply do the following to achieve that while using environment variable interpolation to dynamically adapt your run script at runtime without having to continuously edit your script:

```
from parflow.tools.fs import mkdir, cp

mkdir('input-data')
cp('$PF_SRC/test/input/*.pfb', './input-data/')
```

The working directory used to resolve your relative path gets automatically set when you initialize your run instance by doing `test_run = Run("demo", __file__)`. This means that you should only use the `fs` methods after that initialization line.

The `parflow.tools.fs` module offers the following set of methods which all allow usage of environment variables and relative paths within your run script:

```
from parflow import Run
from parflow.tools.fs import get_absolute_path, exists, chdir
from parflow.tools.fs import mkdir, cp, rm
from parflow.tools.fs import get_text_file_content
# Initialize Run object and set working directory
test_run = Run("demo", __file__)

# Initialize Run object
test_run = Run("demo", __file__)

# Create directory in your current run script directory
mkdir('input')
mkdir('tmp')

# Copy if file missing
if not exists('data.pfb'):
    # Use environment variable to resolve location of PF_DATA
    cp('$PF_DATA/data.pfb')

# Read data using Python tools
full_path = get_absolute_path('data.csv')
with open(full_path) as file:
    pass

# Or use python working directory
```

(continues on next page)

(continued from previous page)

```
chdir('.')
with open('data.csv') as file:
    pass

# Or use the text file content helper
txt = get_text_file_content('data.csv')

# Clean behind yourself
rm('tmp')
```

9.2.2 Full API

1. `get_absolute_path(file_path)`: Returns the absolute file path of the relative file location argument `file_path`.
2. `exists(file_path)`: Returns `True` or `False` as to whether the file at `file_path` exists.
3. `mkdir(dir_name)`: Makes a new directory `dir_name`. This works recursively, so it will also create intermediate directories if they do not exist.
4. `chdir(directory_path)`: Changes the working directory to `directory_path`.
5. `cp(source, target_path='.')`: Copies the file specified in the `source` argument to the location and/or file name specified in the `target_path` argument.
6. `rm(path)`: Removes the file or directory located at `path`.
7. `get_text_file_content(file_path)`: Reads a text file located at `file_path` and returns its content.

9.2.3 Example

If you want more examples on how to leverage those helper functions, you can look at $PARFLOW_SOURCE/test/python/clm/clm/clm.py

The syntax and usage is more compact than the `os` and `shutil` methods commonly used in Python. If you don't provide an absolute path to the file name, these functions will use `get_absolute_path` to find the absolute path based on your working directory, which defaults to the directory where your Python script lives.

9.3 PFB

9.3.1 Introduction

ParFlow Binary (PFB) files are an integral part of ParFlow, and we need an easy way to handle them. Fortunately, we have a several functions within `pftools` that help us with this. The `parflow.tools.io` module allows the user to work with numpy arrays, which are easy to visualize and manipulate in Python. We'll walk through some examples working with PFB files in Python to see just how powerful this is.

9.3.2 Distributing

Let's say you have mastered the conversion of a TCL script to Python, and you have a few PFB files that you need to distribute to convert your workflow to Python. Here, you can use the `dist()` method on your `Run` object that you created, as mentioned in the first tutorial:

```
LWvdz.dist('lw.1km.slope_x.10x.pfb', 'P'=2, 'Q'=2)
```

This will distribute the PFB file with the distribution assigned to the `Process.Topology` keys on the `Run` object (`LWvdz` in this example). However, this can be overwritten for a particular file, as shown above.

9.3.3 Creating PFB from Python

Let's copy another test Python script into our tutorial directory:

```
mkdir -p ~/path/pftools_tutorial/pfb_test
cd ~/path/pftools_tutorial/pfb_test
cp $PARFLOW_SOURCE/test/python/base/richards_FBx/richards_FBx.py .
```

This test is a use case where an internal flow boundary is defined as a numpy array, written to a PFB file, and distributed for use in the run. Open the file, and add the following modules at the top:

```
from parflow import Run
from parflow.tools.fs import get_absolute_path
from parflow.tools.io import write_pfb, read_pfb
import numpy as np
```

We have already covered the first two in prior tutorials. The third line imports the `read_pfb` and `write_pfb` functions from the `parflow.tools.io` module, and the fourth line imports the `numpy` module. We convert a numpy array to a PFB file with the `write_pfb()` function:

```
# Create numpy array
FBx_data = np.ones((20, 20, 20))

# Reduction of 1E-3
FBx_data[:, :, 9] = 0.001

# Write flow boundary file as PFB with write_pfb() function
write_pfb(get_absolute_path('Flow_Barrier_X.pfb'), FBx_data)
```

This creates a 3D numpy array that covers the entire domain and changes the values in the array where X = 10 to 0.001. Note that the numpy array translation to a PFB file reads the dimensions as (Z, Y, X). `write_pfb(get_absolute_path('Flow_Barrier_X.pfb'), FBx_data)` writes the data from the `FBx_data` numpy array to a file called `'Flow_Barrier_X.pfb'`, which will be located in the current working directory, and distributes it.

Now, try running the file. It should execute successfully. Check out the files you now have in your directory - among the other output files is the *'Flow_Barrier_X.pfb'* that you created! If you have a PFB reader tool (such as ParaView), you can see what the file looks like: a 20 x 20 x 20 unit cube with a low-conductivity slice through the middle. Nice!

9.3.4 Loading PFB from Python

Now that we understand how to write a PFB file, how about reading one? This can be useful to do inside a Python script so you can visualize or manipulate existing data. Visualizing output data within the same script as a run can be very helpful!

Let's say you want to visualize some of your output data from the model you just ran, `richards_FBx.py`. In the script, add the following lines to the bottom:

```
FBx_press_out_data = read_pfb(get_absolute_path('richards_FBx.out.press.00010.pfb'))

print(f'Dimensions of output file: {FBx_press_out_data.shape}')
print(FBx_press_out_data)
```

The first line reads the PFB file of the output pressure field at time step = 10 and converts the data to a numpy array. The `print` statements print the dimensions of the array and the data from the file. Save and run this script again to see the printed output. If you're savvy with `matplotlib` or other visualization packages in Python, feel free to visualize to your heart's content!

9.3.5 Full API

1. **`read_pfb(file: str, keys: dict=None, mode: str='full', z_first: bool=True)`**
   Write a single pfb file. The data must be a 3D numpy array with `float64` values. The number of subgrids in the saved file will be `p` * `q` * `r`. This is regardless of the number of subgrids in the PFB file loaded by the ParflowBinaryReader into the numpy array. Therefore, loading a file with ParflowBinaryReader and saving it with this method may restructure the file into a different number of subgrids if you change these values.

   If dist is True then also write a file with the .dist extension added to the file_name. The `.dist` file will contain one line per subgrid with the offset of the subgrid in the `.pfb` file.

   param `file`
   The name of the file to write the array to.

   param `array`
   The array to write.

   param `p`
   Number of subgrids in the x direction.

   param `q`
   Number of subgrids in the y direction.

   param `r`
   Number of subgrids in the z direction.

   param `x`
   The length of the x-axis

   param `y`
   The length of the y-axis

   param `z`
   The length of the z-axis

   param `dx`
   The spacing between cells in the x direction

param dy
The spacing between cells in the y direction

param dz
The spacing between cells in the z direction

param `z_first`
Whether the z-axis should be first or last.

param `dist`
Whether to write the distfile in addition to the pfb.

param kwargs
Extra keyword arguments, primarily to eat unnecessary args by passing in a dictionary with `**dict`.

2. **`write_pfb(file, array, p=1, q=1, r=1, x=0.0, y=0.0, z=0.0, dx=1.0, dy=1.0, dz=1.0, z_first=True, dist=True, **kwargs)`**

   Write a single pfb file. The data must be a 3D numpy array with `float64` values. The number of subgrids in the saved file will be `p * q * r`. This is regardless of the number of subgrids in the PFB file loaded by the ParflowBinaryReader into the numpy array. Therefore, loading a file with ParflowBinaryReader and saving it with this method may restructure the file into a different number of subgrids if you change these values.

   If dist is True then also write a file with the `.dist` extension added to the file_name. The .dist file will contain one line per subgrid with the offset of the subgrid in the `.pfb` file.

   param `file`
   The name of the file to write the array to.

   param array
   The array to write.

   param p
   Number of subgrids in the x direction.

   param q
   Number of subgrids in the y direction.

   param r
   Number of subgrids in the z direction.

   param x
   The length of the x-axis

   param y
   The length of the y-axis

   param z
   The length of the z-axis

   param dx
   The spacing between cells in the x direction

   param dy
   The spacing between cells in the y direction

   param dz
   The spacing between cells in the z direction

   param `z_first`
   Whether the z-axis should be first or last.

param `dist`
Whether to write the distfile in addition to the pfb.

param `kwargs`
Extra keyword arguments, primarily to eat unnecessary args by passing in a dictionary with `**dict`.

3. **`write_dist(file, sg_offs)`**
   Write a distfile.

   param `file`
   The path of the file to be written.

   param `sg_offs`
   The subgrid offsets.

4. **`read_pfb_sequence(file_seq: Iterable[str], keys=None, z_first: bool=True, z_is: str='z')`**
   An efficient wrapper to read a sequence of pfb files. This approach is faster than looping over the `read_pfb` function because it caches the subgrid information from the first pfb file and then uses that to initialize all other readers.

   param `file_seq`
   An iterable sequence of file names to be read.

   param `keys`
   A set of keys for indexing subarrays of the full pfb. Optional. This is mainly a trick for interfacing with xarray, but the format of the keys is:

   ```
   {'x': {'start': start_x, 'stop': end_x},
   'y': {'start': start_y, 'stop': end_y},
   'z': {'start': start_z, 'stop': end_z}}
   ```

   param `z_first`
   Whether the z dimension should be first. If true returned arrays have dimensions `('z', 'y', 'x')` else `('x', 'y', 'z')`

   param `z_is`
   A descriptor of what the z axis represents. Can be one of `'z'`, `'time'`, `'variable'`. Default is `'z'`.

   return
   An `ndarray` containing the data from the files.

9.4 Solid Files

Generating solid (.pfsol) files for a ParFlow run can be somewhat of a pain. PFTools has a few features that can help with this process.

9.4.1 Example

To see the how Python can help generate solid files, navigate to *$PARFLOW_SOURCE/test/python/pfsol/simple-mask/* and open the Python script *simple-mask.py*. Here, you'll see the following lines at the top of the script:

```
from parflow import Run
from parflow.tools.fs import get_absolute_path
from parflow.tools.io import load_patch_matrix_from_sa_file, load_patch_matrix_from_asc_
↪file, load_patch_matrix_from_image_file
from parflow.tools.builders import SolidFileBuilder
```

By now, you should be familiar with the first two modules and functions. The `load_patch_matrix...` functions handle different file types to generate solid files. The `SolidFileBuilder` class imported from the `parflow.tools.builders` module handles the matrices of patches, converting them to ASCII files and passing those to the `pfmask-to-pfsol` converter in ParFlow. This way, the user doesn't have to deal with the more complicated steps.

Lines 52 through 55 show examples of how the `patch_matrix` functions are used for different types of files:

```
sabino_mask = load_patch_matrix_from_sa_file(get_absolute_path('Sabino_Mask.sa'))
# sabino_mask = load_patch_matrix_from_asc_file(get_absolute_path('Sabino_Mask.asc'))
# sabino_mask = load_patch_matrix_from_image_file(get_absolute_path('Sabino_Mask.png'))
# sabino_mask = load_patch_matrix_from_image_file(get_absolute_path('Sabino_Mask.tiff'))
```

Note that only one is used at a time, but all four will work. These functions return a matrix, which is assigned to `sabino_mask`. The input files are located in the same directory as the example, so feel free to reference them. Several of these functions have extra optional arguments, which are described in the full API below.

Next, we'll show some examples of the `SolidFileBuilder` class to demonstrate the arguments and methods that can be called on the object:

```
# Example of using unique ids for each surface [top/bottom/side]
SolidFileBuilder(top=1, bottom=2, side=3) \ # Initializing the SolidFileBuilder
    .mask(sabino_mask) \                      # Setting the 2D mask
    .write('sabino_domain.pfsol', cellsize=90) \  # Write pfsol file
    .for_key(sabino.GeomInput.domaininput)  # Setting keys to "sabino" Run object that␣
↪relate to the solid file

# Example using an id mask for the top patches
SolidFileBuilder(bottom=2, side=3) \ # Initializing the SolidFileBuilder
    .mask(sabino_mask) \               # Setting the 2D mask
    .top_ids(id_array) \                  # Using a 2D numpy array to provide patch ids
    .write('sabino_domain.pfsol', cellsize=90) # Write pfsol file

# Example using the same matrix to write multiple solid files
SolidFileBuilder(top=1, bottom=2, side=3) \
    .mask(sabino_mask) \                      # Setting the 2D mask
    .write('sabino_domain.pfsol', cellsize=90) \  # Write first pfsol file
    .mask(sabino_mask_2) \                      # Setting another 2D mask
    .side_ids(id_array) \              # Using a 2D numpy array to provide new patch ids␣
↪(possibly to change boundary conditions)
    .write('sabino_domain_2.pfsol', cellsize=90)   # Write second pfsol file
```

9.4.2 Full API: IO tools (from `parflow.tools.io`)

1. `load_patch_matrix_from_pfb_file(file_name, layer=None)` - reads in a 2D or 3D ParFlow binary (PFB) file `file_name` and converts it to a patch matrix. If it is a 3D PFB file, the user can specify a vertical layer of the file to convert to the matrix. If no layer is specified, the function will use the top layer of the file.
2. `load_patch_matrix_from_image_file(file_name, color_to_patch=None, fall_back_id=0)` - reads in an image file `file_name` and converts it to a patch matrix. The `color_to_patch` argument is a dictionary with hexidecimal colors as keys with their corresponding ID numbers as values. See *$PARFLOW_SOURCE/test/python/pfsol/image-as-mask/image-as-mask.py* for an example. If `color_to_patch` is not provided, it will default to assume that everything in white is not part of the mask and everything else is part of the mask. The `fall_back_id` is the ID number for colors that are found in the image but are not specified in the `color_to_patch` dictionary. Its default value is zero.
3. `load_patch_matrix_from_asc_file(file_name)` - reads in an ASCII file `file_name` and converts it to a patch matrix.
4. `load_patch_matrix_from_sa_file(file_name)` - reads in a simple ASCII file `file_name` and converts it to a patch matrix.
5. `write_patch_matrix_as_asc(matrix, file_name, xllcorner=0.0, yllcorner=0.0, cellsize=1.0, NODATA_value=0)` - writes an ASCII file to `file_name` from the patch matrix `matrix`. The arguments `xllcorner`, `yllcorner`, `cellsize`, and `NODATA_value` are necessary for the header of the ASCII file.
6. `write_patch_matrix_as_sa(matrix, file_name)` - writes a simple ASCII file to `file_name` from the patch matrix `matrix`.

9.4.3 Full API: SolidFileBuilder

1. `SolidFileBuilder(top=1, bottom=2, side=3)` - initializes a SolidFileBuilder object with default values for the top, bottom, and sides of a domain, respectively.
2. `mask(mask_array)` - applies the matrix array `mask_array` to the SolidFileBuilder object.
3. `write(self, name, xllcorner=0, yllcorner=0, cellsize=0, vtk=False, extra=None, generate_asc_files=False)` - writes the `SolidFileBuilder` object data to the *.pfsol* file `name`. The arguments `xllcorner`, `yllcorner`, and `cellsize=0` help define the size of the solid file domain. If `vtk` is set to `True`, it will write a VTK file `name.vtk` that you can view in ParaView or another VTK viewer to check that the solid file is correct. If there are any extra arguments you want to pass to the `pfmask-to-pfsol` converter in Parflow, specify them using the `extra` parameter, as a list of strings. When `generate_asc_files` is set to `True`, this method generates .asc files for top/bottom/sides, with filenames `<name>_top.asc`, `<name>_bottom.asc`, `<name>_front.asc`, `<name>_back.asc`, `<name>_left.asc`, `<name>_right.asc`, and calls `pfmask-to-pfsol` with individual `mask-*` flags with these files.
4. `for_key(self, geomItem)` - sets two keys on the `Run` object passed in as the `geomItem` argument: 1) `geomItem.InputType = 'SolidFile'` 2) `geomItem.FileName = 'name.pfsol'`. `'name.pfsol'` is implicitly referenced from the `name` argument of the `write` method.
5. `top(patch_id)` - sets the ID number of the top of the solid file domain to the integer `patch_id`. This will override the `top` argument passed to the `SolidFileBuilder` object.
6. `bottom(patch_id)` - sets the ID number of the bottom of the solid file domain to the integer `patch_id`. This will override the `bottom` argument passed to the `SolidFileBuilder` object.
7. `side(patch_id)` - sets the ID number of the side of the solid file domain to the integer `patch_id`. This will override the `side` argument passed to the `SolidFileBuilder` object.

8. `top_ids(top_patch_ids)` - sets the ID numbers of the top of the solid file domain to the values in the numpy array `top_patch_ids`.
9. `bottom_ids(bottom_patch_ids)` - sets the ID numbers of the bottom of the solid file domain to the values in the numpy array `bottom_patch_ids`.
10. `side_ids(side_patch_ids)` - sets the ID numbers of the side of the solid file domain to the values in the numpy array `side_patch_ids`.

9.4.4 More examples

Other example scripts showing how to use the `SolidFileBuilder` can be found in *$PARFLOW_SOURCE/test/python/pfsol/*. If you have an idea for a new feature or improvement to the functionality, please let us know, or better yet, become a contributor!

9.5 Tables for Subsurface Parameters

9.5.1 Introduction

ParFlow domains with complex geology often involve many lines in the input script, which lengthens the script and makes it more cumbersome to navigate. Python PFTools makes it easy to do the following:

- Load in a table of your subsurface properties
- Export a table of the subsurface properties
- Load a database of common soil and geologic properties to set up your domain

9.5.2 Usage of `SubsurfacePropertiesBuilder`

First, we'll show some usage examples of loading tables of parameters within a ParFlow Python script:

```
from parflow import Run
from parflow.tools.builders import SubsurfacePropertiesBuilder

table_test = Run("table_test", __file__)

table_test.GeomInput.Names = 'box_input indi_input'
table_test.GeomInput.indi_input.InputType = 'IndicatorField'
table_test.GeomInput.indi_input.GeomNames = 's1 s2 s3 s4 g1 g2 g3 g4'

# First example: table as in-line text
soil_properties = '''
# ----------------------------------------------------------------
# Sample header
# ----------------------------------------------------------------
key     Perm    Porosity    RelPermAlpha  RelPermN  SatAlpha  SatN    SatSRes     SatSSat

s1      0.26    0.375       3.548         4.162     3.548     4.162   0.000001    1
s2      0.04    -           3.467         2.738     -         2.738   0.000001    1
s3      0.01    0.387       2.692         2.445     2.692     2.445   0.000001    1
s4      0.01    0.439       -             2.659     0.501     2.659   0.000001    1
```

(continues on next page)

(continued from previous page)

```
'''

# Creating object and assigning the subsurface properties
SubsurfacePropertiesBuilder(table_test) \
  .load_txt_content(soil_properties) \          # loading the in-line text
  .load_csv_file('geologic_properties_123.csv') \   # loading external csv file
  .assign('g3', 'g4') \                         # assigns properties of unit 'g3' to␣
↪unit 'g4'
  .apply() \                                    # setting keys based on loaded␣
↪properties
  .print_as_table()                             # printing table of loaded properties
```

At the top of the script, we import the SubsurfacePropertiesBuilder class from parflow.tools.builders. Then, after specifying the GeomInput.indi_input.GeomNames, we include a table for the soil properties as an in-line text variable. Note that the GeomInput.indi_input.GeomNames are in the first column (key), and the parameter names are across the top. These parameter names don't have to match the key names exactly, but they have to be similar. We will explain this later.

We load the soil_properties text by calling the load_txt_content method on the SubsurfacePropertiesBuilder object. To load the geologic properties for the geometric units *g1*, *g2*, and *g3*, we call load_csv_file to load an external csv file. That now leaves one unit, *g4*, that needs properties. We use the assign method to assign properties to unit *g4* from the properties of unit *g3*. Now that all the geometric units have properties, we call apply to set the appropriate keys. The print_as_table method prints out the subsurface properties for each unit. Executing this example will result in a table that looks something like this:

```
key  Perm  Porosity  RelPermAlpha  RelPermN  SatAlpha  SatN   SRes   SSat
s1   0.26  0.375     3.548         4.162     3.548     4.162  1e-06  1.0
s2   0.04  -         3.467         2.738     -         2.738  1e-06  1.0
s3   0.01  0.387     2.692         2.445     2.692     2.445  1e-06  1.0
s4   0.01  0.439     -             2.659     0.501     2.659  1e-06  1.0
g1   0.26  0.375     3.548         4.162     3.548     4.162  1e-06  1.0
g2   0.04  -         3.467         2.738     -         2.738  1e-06  1.0
g3   0.01  0.387     2.692         2.445     2.692     2.445  1e-06  1.0
g4   0.01  0.387     2.692         2.445     2.692     2.445  1e-06  1.0
```

9.5.3 Table formatting for importing

Let's have another look at the in-line table from the usage example above:

```
soil_properties = '''
# ----------------------------------------------------------------
# Sample header
# ----------------------------------------------------------------
key     Perm    Porosity    RelPermAlpha  RelPermN  SatAlpha  SatN     SatSRes     SatSSat

s1      0.26    0.375       3.548         4.162     3.548     4.162    0.000001    1
s2      0.04    -           3.467         2.738     -         2.738    0.000001    1
s3      0.01    0.387       2.692         2.445     2.692     2.445    0.000001    1
s4      0.01    0.439       -             2.659     0.501     2.659    0.000001    1
'''
```

These tables can be formatted in a number of different ways. Here are several considerations:

- Any blank rows or rows beginning with # are ignored in processing.
- Delimiters can be either commas or spaces.
- Table orientation does not matter (i.e., whether the field names are across the first row or down the first column). The only requirement is for that the top-left entry be `key` or one of its aliases.
- The table does not have to be completely filled. As shown here, blank property values must be designated by a hyphen.
- To properly process the table and map to the correct keys, the field names (including `key`) must be one of several possible aliases. The aliases are listed in this yaml file that is included in the Python PFTools. These aliases include the exact end of the key name (e.g., `Perm.Value` as opposed to the alias `Perm`), so when in doubt, you can use the exact name.

9.5.4 Default database loading

We have added several databases of commonly used parameters for different soil and geologic units to provide some helpful guidance. To load these database, you can simply call the `load_default_properties` method on the `SubsurfacePropertiesBuilder` object. The available databases in the Python PFTools package can be found in the "subsurface_*.txt" files here. You can load any of the databases into your `SubsurfacePropertiesBuilder` object by passing in the `database` argument, which is the latter part of the database file name (e.g. "subsurface_conus_1.txt" can be loaded by calling `load_default_properties('conus_1')`). The default database is from Maxwell and Condon (2016). Note that the parameters in the databases are all in the default ParFlow units of meters and hours.

Below is an example of how to use the default database importer on the `Run` object `db_test`:

```
# setting GeomNames
db_test.GeomInput.Names = 'box_input indi_input'
db_test.GeomInput.box_input.InputType = 'Box'
db_test.GeomInput.box_input.GeomName = 'domain'
db_test.GeomInput.indi_input.InputType = 'IndicatorField'
db_test.GeomInput.indi_input.GeomNames = 's1 s2 g2'

# setting dictionary for mapping from database properties (i.e. the keys of map_dict)
# to the different subsurface units (i.e. the values of map_dict)
map_dict = {
  'bedrock_1': ['domain', 'g2'],
  'sand': 's1',
  'loamy_sand': 's2'
}

SubsurfacePropertiesBuilder(db_test)\
  .load_default_properties() \
  .assign(mapping=map_dict) \
  .apply() \
  .print_as_table()
```

The dictionary `map_dict` maps the database properties to the subsurface units in your `Run` object. Note that the properties from the database unit `bedrock_1` are applied to both `domain` and `g2`. If you are assigning a database unit to multiple `GeomNames`, these must be input as a list, as shown. This will print the following:

```
key     Perm    Porosity  SRes   RelPermAlpha  RelPermN
domain  0.005   0.33      0.001  1.0           3.0
g2      0.005   0.33      0.001  1.0           3.0
s1      0.269   0.38      0.14   3.55          4.16
s2      0.0436  0.39      1.26   3.47          2.74
```

9.5.5 Full API for `SubsurfacePropertiesBuilder`

1. `SubsurfacePropertiesBuilder(run=None)`: Instantiates a `SubsurfacePropertiesBuilder` object. If the optional `Run` object `run` is given, it will use the subsurface units in `run` for later application. `run` must be provided as an argument either here or when calling the `apply()` method (see below).
2. `load_csv_file(tableFile, encoding='utf-8-sig')`: Loads a comma-separated (csv) file to your `SubsurfacePropertiesBuilder` object. The default text encoding format is `utf-8-sig`, which should translate files generated from Microsoft Excel.
3. `load_txt_file(tableFile, encoding='utf-8-sig')`: Loads a text file to your `SubsurfacePropertiesBuilder` object. The default text encoding format is `utf-8-sig`.
4. `load_txt_content(txt_content)`: Loads in-line text to your `SubsurfacePropertiesBuilder` object.
5. `load_default_properties(database='conus_1')`: Loads one of several databases of subsurface properties. The default, `conus_1`, is from Maxwell and Condon (2016).
6. `assign(old=None, new=None, mapping=None)`: Assigns properties to the `new` subsurface unit using the properties from the `old` subsurface unit. Alternatively, a dictionary (`mapping`) can be passed in as an argument, which should have the keys as the `old` units, and the values as the `new` units. If an `old` unit will apply to multiple `new` units, the `new` units need to be passed in as a list.
7. `apply(run=None, name_registration=True)`: Applies the loaded subsurface properties to the subsurface units in the `Run` object `run`. If `run` is not provided here, the user must provide the `run` argument when instantiating the `SubsurfacePropertiesBuilder``object. If ``name_registration` is set to `True`, it will add the subsurface unit names (e.g., *s1*, *s2* from the example above) to the list of unit names for each property (e.g., setting `Geom.Perm.Names = 's1 s2 s3 s4'`), and set the `addon` keys not associated with a specific unit (e.g., `Phase.RelPerm.Type`).
8. `print()`: Prints out the subsurface parameters for all subsurface units in a hierarchical format.
9. `print_as_table(props_in_header=True, column_separator=' ')`: Prints out the subsurface parameters for all subsurface units in a table format. `props_in_header` will print the table with the property names as column headings if set to `True`, or as row headings if set to `False`.

9.5.6 Exporting subsurface properties

It is often useful to have a table of the subsurface properties assigned to various subsurface units during a run. As mentioned in the run script API, you can write out a table of the subsurface properties by calling the `write_subsurface_table` method on your `Run` object.

For example, try adding the following line just above the `run()` method call in the `default_richards.py` Python test, with the name of the output file passed in as the `file_name` argument:

```
drich.write_subsurface_table(file_name='def_richards_subsurf.txt')
```

If you do not provide `file_name`, the default file will be a *.csv* file with the name of your run and *subsurface*. In this case, the default file would be *default_richards_subsurface.csv*. Execute the Python script, and you should see the output file *def_richards_subsurf.txt* containing the following:

```
key         Perm  Porosity  SpecStorage  RelPermAlpha  RelPermN  SatAlpha  SatN  SRes ␣
↪SSat
domain      -     -         0.0001       0.005         2.0       0.005     2.0   0.2   0.
↪99
background  4.0   1.0       -            -             -         -         -     -     -
```

See that it only prints out the properties that are explicitly assigned to each of the subsurface units `domain` and `background`.

9.5.7 Examples

Full examples of the `SubsurfacePropertiesBuilder` can be found in the *new_features* subdirectory of the ParFlow Python tests.

- *default_db*: Loading the default database and mapping the database units to subsurface units in the current run.
- *tables_LW*: Showing multiple ways to load tables to replace the subsurface property definition keys in the Little Washita test script.

9.6 Domain definition helpers

9.6.1 Introduction

One of ParFlow's strengths is its customizability; you can practically define any type of hydrologic problem with it. One of the downsides of that, however, is that setting all the keys can be cumbersome, especially when starting a run from scratch. With the new `DomainBuilder`, Python-PFTools helps condense the setting of keys for many common problem definitions.

9.6.2 Usage of `DomainBuilder`

First, we'll show some usage examples of loading tables of parameters within a ParFlow Python script:

```python
from parflow import Run
from parflow.tools.builders import DomainBuilder

LW_Test = Run("LW_Test", __file__)

# ---------------------------------------------------------------------------

bounds = [
    0.0, 41000.0,
    0.0, 41000.0,
    0.0, 100.0
]

domain_patches = 'x_lower x_upper y_lower y_upper z_lower z_upper'
zero_flux_patches = 'x_lower x_upper y_lower y_upper z_lower'
```

(continues on next page)

(continued from previous page)

```
DomainBuilder(LW_Test) \
    .no_wells() \
    .no_contaminants() \
    .water('domain') \
    .variably_saturated() \
    .box_domain('box_input', 'domain', bounds, domain_patches) \
    .homogeneous_subsurface('domain', specific_storage=1.0e-5, isotropic=True) \
    .zero_flux(zero_flux_patches, 'constant', 'alltime') \
    .slopes_mannings('domain', slope_x='LW.slopex.pfb', slope_y='LW.slopey.pfb',␣
↪mannings=5.52e-6) \
    .ic_pressure('domain', patch='z_upper', pressure='press.init.pfb')
```

In this example, the 10 lines associated with the instantiation of the `DomainBuilder` class generate about 70 keys! As is possible with any other key setting, you can always overwrite the keys as necessary; the `DomainBuilder` is designed to help you get started. Once you instantitate the `DomainBuilder` object on a `Run` object, each method will set various keys with the given arguments, which are described below.

9.6.3 Full API

1. `DomainBuilder(run, name='domain')`: Instantiates the `DomainBuilder` object on the `Run` object `run`. The `name` argument is used to define this key:

```
run.Domain.GeomName = name
```

The following examples of the method usage assume that the name of the `Run` object is `run`. All arguments for methods that are passed in as tokens in keys are denoted by `{argument}`.

2. `no_wells()`: Sets the key `run.Wells.Names = ''`
3. `no_contaminants()`: Sets the key `run.Contaminants.Names = ''`
4. `water(self, geom_name=None)`: Sets the following keys:

```
run.Gravity = 1.0
run.Phase.Names = 'water'
run.Phase.water.Density.Type = 'Constant'
run.Phase.water.Density.Value = 1.0
run.Phase.water.Viscosity.Type = 'Constant'
run.Phase.water.Viscosity.Value = 1.0
run.Phase.water.Mobility.Type = 'Constant'
run.Phase.water.Mobility.Value = 1.0
run.PhaseSources.water.Type = 'Constant'

# if geom_name is provided, it will set these keys:
run.PhaseSources.water.GeomNames = geom_name
run.PhaseSources.water.Geom.{geom_name}.Value = 0.0
```

5. `variably_saturated()`: Sets the following keys:

```
run.Solver = 'Richards'
run.Solver.Nonlinear.MaxIter = 10
run.Solver.Nonlinear.ResidualTol = 1e-5
```

(continues on next page)

(continued from previous page)

```
run.Solver.Nonlinear.EtaChoice = 'EtaConstant'
run.Solver.Nonlinear.EtaValue = 1e-5
run.Solver.Nonlinear.UseJacobian = True
run.Solver.Nonlinear.DerivativeEpsilon = 1e-2
run.Solver.Linear.Preconditioner = 'PFMG'
```

6. `fully_saturated()`: Sets the following keys:

```
run.Solver = 'Impes'
```

7. `homogeneous_subsurface(domain_name, perm=None, porosity=None, specific_storage=None, rel_perm=None, saturation=None, isotropic=False)`: Sets the following keys:

```
# if perm is a value, it will set these keys:
# appending domain_name to the list of Geom.Perm.Names
run.Geom.Perm.Names = domain_name
run.Geom.{domain_name}.Perm.Type = 'Constant'
run.Geom.{domain_name}.Perm.Value = perm
# if perm is a file name, it will set these keys:
run.Geom.{domain_name}.Perm.FileName = perm
# if the file name is a PFB file:
run.Geom.{domain_name}.Perm.Type = 'PFBFile'
# if the file name is a NetCDF file:
run.Geom.{domain_name}.Perm.Type = 'NCFile'

# if porosity is a value, it will set these keys:
# appending domain_name to the list of Geom.Porosity.Names
run.Geom.Porosity.GeomNames = domain_name
run.Geom.{domain_name}.Porosity.Type = 'Constant'
run.Geom.{domain_name}.Porosity.Value = porosity
# if porosity is a file name, it will set these keys:
run.Geom.{domain_name}.Porosity.FileName = porosity
# if the file name is a PFB file:
run.Geom.{domain_name}.Porosity.Type = 'PFBFile'
# if the file name is a NetCDF file:
run.Geom.{domain_name}.Porosity.Type = 'NCFile'

# if specific_storage is provided, it will set these keys:
# appending domain_name to the list of SpecificStorage.GeomNames
run.SpecificStorage.GeomNames = domain_name
run.SpecificStorage.Type = 'Constant'
run.Geom.{domain_name}.SpecificStorage.Value = specific_storage

# if rel_perm is provided, it must be a dictionary with the following key/value pairs:
# {'Type': 'VanGenuchten', 'Alpha': 3.5, 'N': 2.0}
# using this dictionary, it will set the following keys:
# appending domain_name to the list of Phase.RelPerm.GeomNames
run.Phase.RelPerm.GeomNames = domain_name
# if Type = VanGenuchten, it will set the following keys:
self.run.Geom.{domain_name}.RelPerm.Alpha = rel_perm['Alpha']
self.run.Geom.{domain_name}.RelPerm.N = rel_perm['N']
```

(continues on next page)

(continued from previous page)

```
# if saturation is provided, it must be a dictionary with the following key/value pairs:
# {'Type': 'VanGenuchten', 'Alpha': 3.5, 'N': 2.0, 'SRes': 0.1, 'SSat': 1.0}
# Alpha and N are optional, and can default to the value of the corresponding properties␣
↪in rel_perm
# using this dictionary, it will set the following keys:
# appending domain_name to the list of Phase.Saturation.GeomNames
run.Phase.Saturation.GeomNames = domain_name
# if Type = VanGenuchten, it will set the following keys:
run.Geom.{domain_name}.Saturation.Alpha = saturation['Alpha']
run.Geom.{domain_name}.Saturation.N = saturation['N']
run.Geom.{domain_name}.Saturation.SRes = saturation['SRes']
run.Geom.{domain_name}.Saturation.SSat = saturation['SSat']

# if isotropic is True, it will set these keys:
run.Perm.TensorType = 'TensorByGeom'
# appending domain_name to the list of Geom.Perm.TensorByGeom.Names
run.Geom.Perm.TensorByGeom.Names = domain_name
run.Geom.{domain_name}.Perm.TensorValX = 1.0
run.Geom.{domain_name}.Perm.TensorValY = 1.0
run.Geom.{domain_name}.Perm.TensorValZ = 1.0
```

8. `box_domain(box_input, domain_geom_name, bounds=None, patches=None)`: Sets the following keys:

```
# append box_input to the GeomInput.Names
run.GeomInput.Names = box_input
run.GeomInput.{box_input}.InputType = 'Box'
run.GeomInput.{box_input}.GeomName = domain_geom_name

# if bounds is not provided, it will default to using the ComputationalGrid keys to␣
↪define the boundaries:
run.Geom.{domain_geom_name}.Lower.X = 0.0
run.Geom.{domain_geom_name}.Lower.Y = 0.0
run.Geom.{domain_geom_name}.Lower.Z = 0.0
run.Geom.{domain_geom_name}.Upper.X = run.ComputationalGrid.DX * run.ComputationalGrid.NX
run.Geom.{domain_geom_name}.Upper.Y = run.ComputationalGrid.DY * run.ComputationalGrid.NY
run.Geom.{domain_geom_name}.Upper.Z = run.ComputationalGrid.DZ * run.ComputationalGrid.NZ

# bounds should be provided as a list of coordinates in this order:
# [lower_x, upper_x, lower_y, upper_y, lower_z, upper_z]
run.Geom.{domain_geom_name}.Lower.X = bounds[0]
run.Geom.{domain_geom_name}.Upper.X = bounds[1]
run.Geom.{domain_geom_name}.Lower.Y = bounds[2]
run.Geom.{domain_geom_name}.Upper.Y = bounds[3]
run.Geom.{domain_geom_name}.Lower.Z = bounds[4]
run.Geom.{domain_geom_name}.Upper.Z = bounds[5]

# if patches is provided as a single string of the box domain patches (e.g., 'left right .
↪..'), it will set this key:
run.Geom.{domain_geom_name}.Patches = patches
```

9. `slopes_mannings(self, domain_geom_name, slope_x=None, slope_y=None, mannings=None)`: Sets the following keys:

```
# if slope_x is provided, it will set these keys:
# appending domain_name to the list of TopoSlopesX.GeomNames
run.TopoSlopesX.GeomNames = domain_geom_name
# if slope_x is a number, it will set these keys:
run.TopoSlopesX.Type = 'Constant'
run.TopoSlopesX.Geom.{domain_geom_name}.Value = slope_x
# if slope_x is a file name, it will set these keys:
run.TopoSlopesX.FileName = slope_x
# if the file name is a PFB file:
run.TopoSlopesX.Type = 'PFBFile'
# if the file name is a NetCDF file:
run.TopoSlopesX.Type = 'NCFile'

# if slope_y is provided, it will set these keys:
# appending domain_name to the list of TopoSlopesY.GeomNames
run.TopoSlopesY.GeomNames = domain_geom_name
# if slope_y is a number, it will set these keys:
run.TopoSlopesY.Type = 'Constant'
run.TopoSlopesY.Geom.{domain_geom_name}.Value = slope_y
# if slope_y is a file name, it will set these keys:
run.TopoSlopesY.FileName = slope_y
# if the file name is a PFB file:
run.TopoSlopesY.Type = 'PFBFile'
# if the file name is a NetCDF file:
run.TopoSlopesY.Type = 'NCFile'

# if mannings is provided, it will set these keys:
# appending domain_name to the list of Mannings.GeomNames
run.Mannings.GeomNames = domain_geom_name
# if mannings is a number, it will set these keys:
run.Mannings.Type = 'Constant'
run.Mannings.Geom.{domain_geom_name}.Value = mannings
# if mannings is a file name, it will set these keys:
run.Mannings.FileName = mannings
# if the file name is a PFB file:
run.Mannings.Type = 'PFBFile'
# if the file name is a NetCDF file:
run.Mannings.Type = 'NCFile'
```

10. `zero_flux(self, patches, cycle_name, interval_name)`: Sets the following keys:

```
run.BCPressure.PatchNames += [patch]
run.Patch[patch].BCPressure.Type = 'FluxConst'
run.Patch[patch].BCPressure.Cycle = cycle_name
run.Patch[patch].BCPressure[interval_name].Value = 0.0
```

11. `ic_pressure(self, domain_geom_name, patch, pressure)`: Sets the following keys:

```
run.ICPressure.GeomNames = domain_geom_name
run.Geom.{domain_geom_name}.ICPressure.RefPatch = patch

# if pressure is a PFB file, it will set the following keys:
run.ICPressure.Type = 'PFBFile'
```

(continues on next page)

(continued from previous page)

```
run.Geom.domain.ICPressure.FileName = pressure
```

12. `clm(met_file_name, top_patch, cycle_name, interval_name)`: Sets the following keys:

```
# ensure time step is hourly
run.TimeStep.Type = 'Constant'
run.TimeStep.Value = 1.0
# ensure OverlandFlow is the top boundary condition
run.Patch.{top_patch}.BCPressure.Type = 'OverlandFlow'
run.Patch.{top_patch}.BCPressure.Cycle = cycle_name
run.Patch.{top_patch}.BCPressure.{interval_name}.Value = 0.0
# set CLM keys
run.Solver.LSM = 'CLM'
run.Solver.CLM.CLMFileDir = "."
run.Solver.PrintCLM = True
run.Solver.CLM.Print1dOut = False
run.Solver.BinaryOutDir = False
run.Solver.CLM.DailyRST = True
run.Solver.CLM.SingleFile = True
run.Solver.CLM.CLMDumpInterval = 24
run.Solver.CLM.WriteLogs = False
run.Solver.CLM.WriteLastRST = True
run.Solver.CLM.MetForcing = '1D'
run.Solver.CLM.MetFileName = met_file_name
run.Solver.CLM.MetFilePath = "."
run.Solver.CLM.MetFileNT = 24
run.Solver.CLM.IstepStart = 1.0
run.Solver.CLM.EvapBeta = 'Linear'
run.Solver.CLM.VegWaterStress = 'Saturation'
run.Solver.CLM.ResSat = 0.1
run.Solver.CLM.WiltingPoint = 0.12
run.Solver.CLM.FieldCapacity = 0.98
run.Solver.CLM.IrrigationType = 'none'
```

13. `well(name, type, x, y, z_upper, z_lower, cycle_name, interval_name, action='Extraction', saturation=1.0, phase='water', hydrostatic_pressure=None, value=None)`: Sets the following keys:

```
# append name to Wells.Names
run.Wells.Names += [name]

run.Wells.{name}.InputType = 'Vertical'
run.Wells.{name}.Action = action
run.Wells.{name}.Type = type
run.Wells.{name}.X = x
run.Wells.{name}.Y = y
run.Wells.{name}.ZUpper = z_upper
run.Wells.{name}.ZLower = z_lower
run.Wells.{name}.Method = 'Standard'
run.Wells.{name}.Cycle = cycle_name
run.Wells.{name}.{interval_name}.Saturation.{phase}.Value = saturation

```

(continues on next page)

(continued from previous page)

```
# if type is set to 'Pressure', set Pressure.Value
run.Wells.{name}.{interval_name}.Pressure.Value = hydrostatic_pressure

# For extraction wells (run.Wells.{name}.Action = 'Extraction'), set these keys:
# if type is set to 'Pressure' and value is provided, set Extraction.Pressure.Value
run.Wells.{name}.{interval_name}.Extraction.Pressure.Value = value
# if type is set to 'Flux' and value is provided, set Extraction.Flux.{phase}.Value
run.Wells.{name}.{interval_name}.Extraction.Flux.{phase}.Value = value

# For injection wells (run.Wells.{name}.Action = 'Injection'), set these keys:
# if type is set to 'Pressure' and value is provided, set Injection.Pressure.Value
run.Wells.{name}.{interval_name}.Injection.Pressure.Value = value
# if type is set to 'Flux' and value is provided, set Injection.Flux.{phase}.Value
run.Wells.{name}.{interval_name}.Injection.Flux.{phase}.Value = value
```

14. `spinup_timing(self, initial_step, dump_interval)`: Sets the following keys:

```
run.TimingInfo.BaseUnit = 1
run.TimingInfo.StartCount = 0
run.TimingInfo.StartTime = 0.0
run.TimingInfo.StopTime = 10000000
run.TimingInfo.DumpInterval = dump_interval
run.TimeStep.Type = 'Growth'
run.TimeStep.InitialStep = initial_step
run.TimeStep.GrowthFactor = 1.1
run.TimeStep.MaxStep = 1000000
run.TimeStep.MinStep = 0.1
```

9.7 Data Accessor

9.7.1 Introduction

The `DataAccessor` class is a helper class for extracting numpy arrays from a given ParFlow run.

9.7.2 Usage of `DataAccessor`

First, we'll show some examples of using the DataAccessor class within a ParFlow Python script:

```
from parflow import Run

# Create a Run object from a .pfidb file
run = Run.from_definition('/path/to/pfidb/file')

# Get the DataAccessor object corresponding to the Run object
data = run.data_accessor

# Iterate through the timesteps of the DataAccessor object
# i goes from 0 to n_timesteps - 1
for i in data.times:
```

(continues on next page)

(continued from previous page)

```
#----------------------------- Evapotranspiration -------------------------------

# nz-by-ny-by-nx array of ET values (bottom to top layer)
print(data.et)

#------------------------------- Overland Flow ----------------------------------

# ny-by-nx array of overland flow values - 'OverlandKinematic' flow method
print(data.overland_flow_grid())

# ny-by-nx array of overland flow values - 'OverlandFlow' flow method
print(data.overland_flow_grid(flow_method='OverlandFlow'))

# Total outflow for the domain (scalar value) - 'OverlandKinematic' flow method
print(data.overland_flow())

# Total outflow for the domain (scalar value) - 'OverlandFlow' flow method
print(data.overland_flow(flow_method='OverlandFlow'))

#-------------------------- Subsurface/Surface Storage --------------------------

# nz-by-ny-by-nx array of subsurface storage values (bottom to top layer)
print(data.subsurface_storage)

# ny-by-nx array of surface storage values
print(data.surface_storage)

#----------------------------- Water Table Depth --------------------------------

# ny-by-nx array of water table depth values
print(data.wtd)

data.time += 1
```

9.7.3 Full API

1. **`run.data_accessor`**

   Accesses the `DataAccessor` object on the current `Run` object `run`.

   return
   An instance of the `DataAccessor` class.

2. **`data.time`**

   Get current timestep set on the current `DataAccessor` object `data`. This timestep will be used to determine the file index for accessing timeseries data.

   return
   An integer value representing the current timestep.

3. **`data.times`**

   return
   An array of timesteps on the current `DataAccessor` object `data`, from 0 to n_timesteps - 1.

4. **`data.forcing_time`**

   Get current forcing timestep set on the current `DataAccessor` object `data`. This timestep will be used to determine the file index for accessing forcing timeseries data.

   return
   An integer value representing the current forcing timestep.

5. **`data.shape`**

   return
   Tuple containing `(ComputationalGrid.NZ, ComputationalGrid.NY, ComputationalGrid.NX)` set on the current `DataAccessor` object `data`.

6. **`data.dx`**

   return
   Value of `ComputationalGrid.DX` on the current `DataAccessor` object `data`.

7. **`data.dy`**

   return
   Value of `ComputationalGrid.DY` on the current `DataAccessor` object `data`.

8. **`data.dz`**

   return
   Array of size `ComputationalGrid.NZ` containing either `dz_scale` values or the value of `ComputationalGrid.DZ` set on the current `DataAccessor` object `data`.

9. **`data.mannings`**

   return
   An `ndarray` containing mannings roughness coefficient data for the current `DataAccessor` object `data`.

10. **`data.mask`**

    return
    An `ndarray` containing the mask of your domain for the current `DataAccessor` object `data`.

11. **`data.slope_x`**

    return
    An `ndarray` containing the `x` topographic slope values for the current `DataAccessor` object `data`.

12. **`data.slope_y`**

    return
    An `ndarray` containing the `y` topographic slope values for the current `DataAccessor` object `data`.

13. **`data.elevation`**

    return
    An `ndarray` containing the elevation topographic slope values for the current `DataAccessor` object `data`.

14. **`data.computed_porosity`**

    return
    An `ndarray` containing computed porosity values on the current `DataAccessor` object `data`.

15. **`data.computed_permeability_x`**

return
An `ndarray` containing computed permeability `x` values on the current `DataAccessor` object `data`.

16. **`data.computed_permeability_y`**

    return
    An `ndarray` containing computed permeability `y` values on the current `DataAccessor` object `data`.

17. **`data.computed_permeability_z`**

    return
    An `ndarray` containing computed permeability `z` values on the current `DataAccessor` object `data`.

18. **`data.pressure_initial_condition`**

    return
    An `ndarray` containing initial condition pressure values on the current `DataAccessor` object `data`.

19. **`data.pressure_boundary_conditions`**

    return
    A dictionary containing `key=value` pairs of the form `{patch_name}__{cycle_name} = value` for pressure boundary conditions on the current `DataAccessor` object `data`.

20. **`data.pressure`**

    return
    An `ndarray` containing pressure values for the current timestep set on the `DataAccessor` object `data`.

21. **`data.saturation`**

    return
    An `ndarray` containing saturation values for the current timestep set on the `DataAccessor` object `data`.

22. **`data.specific_storage`**

    return
    An `ndarray` containing specific storage values for the current `DataAccessor` object `data`.

23. **`data.et`**

    return
    An `nz` by `ny` by `nx` `ndarray` of evapotranspiration values (units L^3/T), spanning all layers (bottom to top) for the current `DataAccessor` object `data`.

24. **`data.overland_flow(flow_method='OverlandKinematic', epsilon=1e-5)`**

    param `flow_method`
    Either `'OverlandFlow'` or `'OverlandKinematic'`. `'OverlandKinematic'` by default.

    param `epsilon`
    Minimum slope magnitude for solver. Only applicable if `flow_method='OverlandKinematic'`. This is set using the `Solver.OverlandKinematic.Epsilon` key in Parflow.

    return
    An `ny` by `nx` `ndarray` of overland flow values for the current `DataAccessor` object `data`.

25. **`data.overland_flow_grid(flow_method='OverlandKinematic', epsilon=1e-5)`**

    param `flow_method`
    Either `'OverlandFlow'` or `'OverlandKinematic'`. `'OverlandKinematic'` by default.

    param `epsilon`
    Minimum slope magnitude for solver. Only applicable if `kinematic=True`. This is set using the `Solver.OverlandKinematic.Epsilon` key in Parflow.

    return
    An `ny` by `nx` ndarray of overland flow values for the current `DataAccessor` object `data`.

26. **`data.subsurface_storage`**

    return
    An `nz` by `ny` by `nx` `ndarray` of subsurface storage values, spanning all layers (bottom to top), for the current `DataAccessor` object `data`.

27. **`data.surface_storage`**

    return
    An `ny` by `nx` `ndarray` of surface storage values for the current `DataAccessor` object `data`.

28. **`data.wtd`**

    return
    An `ny` by `nx` `ndarray` of water table depth values (measured from the top) for the current `DataAccessor` object `data`.

29. **`data.clm_output(field, layer=-1)`**

    param field
    CLM field, one of: `'eflx_lh_tot'`, `'eflx_lwrad_out'`, `'eflx_sh_tot'`, `'eflx_soil_grnd'`, `'qflx_evap_tot'`, `'qflx_evap_grnd'`, `'qflx_evap_soi'`, `'qflx_evap_veg'`, `'qflx_tran_veg'`, `'qflx_infl'`, `'swe_out'`, `'t_grnd'`, `'qflx_qirr'`, `'t_soil'`

    param layer
    Layer of data

    return
    An `ndarray` of CLM output for the given `field` and `layer` on the current `DataAccessor` object `data`.

30. **`data.clm_output_variables`**

    return
    Tuple containing names of all CLM output variables: `('eflx_lh_tot', 'eflx_lwrad_out', 'eflx_sh_tot', 'eflx_soil_grnd', 'qflx_evap_tot', 'qflx_evap_grnd', 'qflx_evap_soi', 'qflx_evap_veg', 'qflx_tran_veg', 'qflx_infl', 'swe_out', 't_grnd', 'qflx_qirr', 't_soil')`

31. **`data.clm_output_diagnostics`**

    return
    String filepath to CLM output diagnostics file for the current `DataAccessor` object `data`.

32. **`data.clm_output_eflx_lh_tot`**

    return
    An `ndarray` containing CLM `eflx_lh_tot` data for the current `DataAccessor` object `data`.

33. **`data.clm_output_eflx_lwrad_out`**

return
An ndarray containing CLM eflx_lwrad_out data for the current DataAccessor object data.

34. **data.clm_output_eflx_sh_tot**

    return
    An ndarray containing CLM eflx_sh_tot data for the current DataAccessor object data.

35. **data.clm_output_eflx_soil_grnd**

    return
    An ndarray containing CLM eflx_soil_grnd data for the current DataAccessor object data.

36. **data.clm_output_qflx_evap_grnd**

    return
    An ndarray containing CLM qflx_evap_grnd data for the current DataAccessor object data.

37. **data.clm_output_qflx_evap_soi**

    return
    An ndarray containing CLM qflx_evap_soi data for the current DataAccessor object data.

38. **data.lm_output_qflx_evap_tot**

    return
    An ndarray containing CLM qflx_evap_tot data for the current DataAccessor object data.

39. **data.clm_output_qflx_evap_veg**

    return
    An ndarray containing CLM qflx_evap_veg data for the current DataAccessor object data.

40. **data.clm_output_qflx_infl**

    return
    An ndarray containing CLM qflx_infl data for the current DataAccessor object data.

41. **data.clm_output_qflx_top_soil**

    return
    An ndarray containing CLM qflx_top_soil data for the current DataAccessor object data.

42. **data.clm_output_qflx_tran_veg**

    return
    An ndarray containing CLM qflx_tran_veg data for the current DataAccessor object data.

43. **data.clm_output_swe_out**

    return
    An ndarray containing CLM swe_out data for the current DataAccessor object data.

44. **data.clm_output_t_grnd**

return
An `ndarray` containing CLM `t_grnd` data for the current `DataAccessor` object `data`.

45. **`data.clm_forcing(name)`**

    param name
    Forcing type you're interested in

    return
    An `ndarray` containing CLM forcing data for the given `name` and forcing timestep set on the current `DataAccessor` object `data`.

46. **`data.clm_forcing_dswr`**

    return
    An `ndarray` containing CLM forcing data for Downward Visible or Short-Wave radiation [W/m2] for the forcing timestep set on the current `DataAccessor` object `data`.

47. **`data.clm_forcing_dlwr`**

    return
    An `ndarray` containing CLM forcing data for Downward Infa-Red or Long-Wave radiation [W/m2] for the forcing timestep set on the current `DataAccessor` object `data`.

48. **`data.clm_forcing_apcp`**

    return
    An `ndarray` containing CLM forcing data for Precipitation rate [mm/s] for the forcing timestep set on the current `DataAccessor` object `data`.

49. **`data.clm_forcing_temp`**

    return
    An `ndarray` containing CLM forcing data for Air temperature [K] for the forcing timestep set on the current `DataAccessor` object `data`.

50. **`data.clm_forcing_ugrd`**

    return
    An `ndarray` containing CLM forcing data for West-to-East or U-component of wind [m/s] for the forcing timestep set on the current `DataAccessor` object `data`.

51. **`data.clm_forcing_vgrd`**

    return
    An `ndarray` containing CLM forcing data for South-to-North or V-component of wind [m/s] for the forcing timestep set on the current `DataAccessor` object `data`.

52. **`data.clm_forcing_press`**

    return
    An `ndarray` containing CLM forcing data for Atmospheric Pressure [pa] for the forcing timestep set on the current `DataAccessor` object `data`.

53. **`data.clm_forcing_spfh`**

    return
    An `ndarray` containing CLM forcing data for Water-vapor specific humidity [kg/kg] for the forcing timestep set on the current `DataAccessor` object `data`.

54. **`data.clm_map_land_fraction(name)`**

    param name
    Type of land frac data you're interested in

return
Data corresponding to `Solver.CLM.Vegetation.Map.LandFrac[name]` key set on the ParFlow run for the current `DataAccessor` object `data`.

55. **`data.clm_map_latitude`**

    return
    Data corresponding to `Solver.CLM.Vegetation.Map.Latitude` key set on the ParFlow run for the current `DataAccessor` object `data`.

56. **`data.clm_map_longitude`**

    return
    Data corresponding to `Solver.CLM.Vegetation.Map.Longitude` key set on the ParFlow run for the current `DataAccessor` object `data`.

57. **`data.clm_map_sand`**

    return
    Data corresponding to `Solver.CLM.Vegetation.Map.Sand` key set on the ParFlow run for the current `DataAccessor` object `data`.

58. **`data.clm_map_clay`**

    return
    Data corresponding to `Solver.CLM.Vegetation.Map.Clay` key set on the ParFlow run for the current `DataAccessor` object `data`.

59. **`data.clm_map_color`**

    return
    Data corresponding to `Solver.CLM.Vegetation.Map.Color` key set on the ParFlow run for the current `DataAccessor` object `data`.

9.8 Hydrology Module

9.8.1 Introduction

The Python PFTools Hydrology module provides standalone functions for common hydrologic calculations.

9.8.2 Usage of `Hydrology`

First, we'll show some examples of using the Hydrology class within a ParFlow Python script:

```
import numpy as np
from parflow import Run
from parflow.tools.hydrology import calculate_surface_storage, calculate_subsurface_
↪storage, \
    calculate_water_table_depth, calculate_evapotranspiration, calculate_overland_flow_
↪grid

# Create a Run object from the .pfidb file
run = Run.from_definition('/path/to/pfidb/file')

# Get the DataAccessor object corresponding to the Run object
```

(continues on next page)

(continued from previous page)

```
data = run.data_accessor

# ----------------------------------------------
# Get relevant information from the DataAccessor
# ----------------------------------------------

# Resolution
dx = data.dx
dy = data.dy
# Thickness of each layer, bottom to top
dz = data.dz

# Extent
nx = data.shape[2]
ny = data.shape[1]
nz = data.shape[0]

# ------------------------------------------
# Time-invariant values
# ------------------------------------------

porosity = data.computed_porosity
specific_storage = data.specific_storage
mask = data.mask
et = data.et                           # shape (nz, ny, nx) - units 1/T.
slopex = data.slope_x                  # shape (ny, nx)
slopey = data.slope_y                  # shape (ny, nx)
mannings = data.mannings               # scalar value

# ------------------------------------------
# Time-variant values
# ------------------------------------------

# no. of time steps
nt = len(data.times)

# ------------------------------------------
# Initialization
# ------------------------------------------

# Arrays for total values (across all layers), with time as the first axis
subsurface_storage = np.zeros(nt)
surface_storage = np.zeros(nt)
wtd = np.zeros((nt, ny, nx))
et = np.zeros(nt)
overland_flow = np.zeros((nt, ny, nx))

# ------------------------------------------
# Loop through time steps
# i goes from 0 to n_timesteps - 1
# ------------------------------------------
for i in data.times:
```

(continues on next page)

(continued from previous page)

```
    pressure = data.pressure
    saturation = data.saturation

    # Total subsurface storage for this time step is the summation of substorage surface␣
↪across all x/y/z slices
    subsurface_storage[i, ...] = np.sum(
        calculate_subsurface_storage(porosity, pressure, saturation, specific_storage,␣
↪dx, dy, dz, mask=mask),
        axis=(0, 1, 2)
    )

    # Total surface storage for this time step is the summation of substorage surface␣
↪across all x/y slices
    surface_storage[i, ...] = np.sum(
        calculate_surface_storage(pressure, dx, dy, mask=mask),
        axis=(0, 1)
    )

    wtd[i, ...] = calculate_water_table_depth(pressure, saturation, dz)

    if et is not None:
        # Total ET for this time step is the summation of ET values across all x/y/z␣
↪slices
        et[i, ...] = np.sum(
            calculate_evapotranspiration(et_flux_values, dx, dy, dz, mask=mask),
            axis=(0, 1, 2)
        )

    overland_flow[i, ...] = calculate_overland_flow_grid(pressure, slopex, slopey,␣
↪mannings, dx, dy, mask=mask)

    data.time += 1
```

9.8.3 Full API

1. **`calculate_water_table_depth(pressure, saturation, dz)`**
   Calculate water table depth from the land surface.

   param pressure
   An `nz` by `ny` by `nx` `ndarray` of pressure values (bottom layer to top layer)

   param saturation
   An `nz` by `ny` by `nx` `ndarray` ndarray of saturation values (bottom layer to top layer)

   param dz
   An `ndarray` of shape `(nz,)` of thickness values (bottom layer to top layer)

   return
   A `ny` by `nx` `ndarray` of water table depth values (measured from the top)

2. **`calculate_subsurface_storage(porosity, pressure, saturation, specific_storage, dx, dy, dz, mask=None)`**
   Calculate gridded subsurface storage across several layers. For each layer in the subsurface, storage consists of two parts:

1) Incompressible subsurface storage (`porosity` * `saturation` * depth of this layer) * `dx` * `dy`
2) Compressible subsurface storage (`pressure` * `saturation` * `specific storage` * depth of this layer) * `dx` * `dy`

param porosity
An `nz` by `ny` by `nx` `ndarray` of porosity values (bottom layer to top layer)

param pressure
An `nz` by `ny` by `nx` `ndarray` of pressure values (bottom layer to top layer)

param saturation
An `nz` by `ny` by `nx` `ndarray` of saturation values (bottom layer to top layer)

param specific_storage
An `nz` by `ny` by `nx` `ndarray` of specific storage values (bottom layer to top layer)

param dx
Length of a grid element in the `x` direction

param dy
Length of a grid element in the `y` direction

param dz
Thickness of a grid element in the `z` direction (bottom layer to top layer)

param mask
An `nz` by `ny` by `nx` `ndarray` of mask values (bottom layer to top layer). If `None`, assumed to be an `nz` by `ny` by `nx` `ndarray` of 1s.

return
An `nz` by `ny` by `nx` `ndarray` of subsurface storage values, spanning all layers (bottom to top)

3. **`calculate_surface_storage(pressure, dx, dy, mask=None)`**
   Calculate gridded surface storage on the top layer. Surface storage is given by: Pressure at the top layer * `dx` * `dy` (for pressure values > 0)

param pressure
An `nz` by `ny` by `nx` `ndarray` of pressure values (bottom layer to top layer)

param dx
Length of a grid element in the x direction

param dy
Length of a grid element in the y direction

param mask
An `nz` by `ny` by `nx` `ndarray` of mask values (bottom layer to top layer). If `None`, assumed to be an `nz` by `ny` by `nx` `ndarray` of 1s.

return
An `ny` by `nx` `ndarray` of surface storage values

4. **`calculate_evapotranspiration(et, dx, dy, dz, mask=None)`**
   Calculate gridded evapotranspiration across several layers.

param et
An `nz` by `ny` by `nx` `ndarray` of evapotranspiration flux values with units 1/T (bottom layer to top layer)

param dx
Length of a grid element in the `x` direction

param dy
Length of a grid element in the `y` direction

param dz
Thickness of a grid element in the `z` direction (bottom layer to top layer)

param mask
An `nz` by `ny` by `nx` `ndarray` of mask values (bottom layer to top layer). If `None`, assumed to be an `nz` by `ny` by `nx` `ndarray` of 1s.

return
An `nz` by `ny` by `nx` `ndarray` of evapotranspiration values (units L^3/T), spanning all layers (bottom to top)

5. **`calculate_overland_fluxes(pressure, slopex, slopey, mannings, dx, dy, flow_method='OverlandKinematic', epsilon=1e-5, mask=None)`**
   Calculate overland fluxes across grid faces.

param pressure
An `nz` by `ny` by `nx` `ndarray` of pressure values (bottom layer to top layer)

param slopex
`ny` by `nx`

param slopey
`ny` by `nx`

param mannings
a scalar value, or a `ny` by `nx` `ndarray`

param dx
Length of a grid element in the `x` direction

param dy
Length of a grid element in the `y` direction

param flow_method
Either 'OverlandFlow' or 'OverlandKinematic'. 'OverlandKinematic' by default.

param epsilon
Minimum slope magnitude for solver. Only applicable if `flow_method='OverlandKinematic'`. This is set using the `Solver.OverlandKinematic.Epsilon` key in Parflow.

param mask
An `nz` by `ny` by `nx` `ndarray` of mask values (bottom layer to top layer). If `None`, assumed to be an `nz` by `ny` by `nx` `ndarray` of 1s.

return
A 2-tuple:

(`qeast`: A `ny` by `(nx+1)` `ndarray` of overland flux values,

`qnorth`: A `(ny+1)` by `nx` `ndarray` of overland flux values)

```
Numpy array origin is at the top left.
The cardinal direction along axis 0 (rows) is North (going down!!).
The cardinal direction along axis 1 (columns) is East (going right).
qnorth ``(ny+1,nx)`` and qeast ``(ny,nx+1)`` values are to be interpreted as␣
↪follows.

```

(continues on next page)

(continued from previous page)

```
+------------------------------------> (East)
|
|                           qnorth_i,j (outflow if negative)
|                                  +-----+------+
|                                  |     |      |
|                                  |     |      |
|  qeast_i,j (outflow if negative) |-->  v      |---> qeast_i,j+1 (outflow if␣
↪positive)
|                                  |            |
|                                  | Cell  i,j  |
|                                  +-----+------+
|                                        |
|                                        |
|                                        v
|                           qnorth_i+1,j (outflow if positive)
v
(North)
```

6. **`calculate_overland_flow_grid(pressure, slopex, slopey, mannings, dx, dy, flow_method='OverlandKinematic', epsilon=1e-5, mask=None)`**
   Calculate overland outflow per grid cell of a domain.

   param `pressure`
   An `nz` by `ny` by `nx` `ndarray` of pressure values (bottom layer to top layer)

   param `slopex`
   `ny` by `nx`

   param `slopey`
   `ny` by `nx`

   param `mannings`
   a scalar value, or a `ny` by `nx` `ndarray`

   param `dx`
   Length of a grid element in the `x` direction

   param `dy`
   Length of a grid element in the `y` direction

   param `flow_method`
   Either 'OverlandFlow' or 'OverlandKinematic'. 'OverlandKinematic' by default.

   param `epsilon`
   Minimum slope magnitude for solver. Only applicable if `kinematic=True`. This is set using the `Solver.OverlandKinematic.Epsilon` key in Parflow.

   param `mask`
   An `nz` by `ny` by `nx` `ndarray` of mask values (bottom layer to top layer). If `None`, assumed to be an `nz` by `ny` by `nx` `ndarray` of 1s.

   return
   An `ny` by `nx` `ndarray` of overland flow values

7. `calculate_overland_flow(pressure, slopex, slopey, mannings, dx, dy, flow_method='OverlandKinematic', epsilon=1e-5, mask=None)`

   param `pressure`
   An `nz` by `ny` by `nx` `ndarray` of pressure values (bottom layer to top layer)

param `slopex`
`ny` by `nx`

param `slopey`
`ny` by `nx`

param `mannings`
a scalar value, or a `ny` by `nx` `ndarray`

param `dx`
Length of a grid element in the `x` direction

param `dy`
Length of a grid element in the `y` direction

param `flow_method`
Either 'OverlandFlow' or 'OverlandKinematic'. 'OverlandKinematic' by default.

param `epsilon`
Minimum slope magnitude for solver. Only applicable if `flow_method='OverlandKinematic'`. This is set using the `Solver.OverlandKinematic.Epsilon` key in Parflow.

param `mask`
An `nz` by `ny` by `nx` `ndarray` of mask values (bottom layer to top layer). If None, assumed to be an `nz` by `ny` by `nx` `ndarray` of 1s.

return
A `ny` by `nx` `ndarray` of overland flow values

CHAPTER

TEN

BIBLIOGRAPHY

BIBLIOGRAPHY

[AFBBP+02] Tompson AFB., C. J. Bruton, G. A. Pawloski, D. K. Smith, W. L. Bourcier, D. E. Shumaker, A. B. Kersting, S. F. Carle, and R. M. Maxwell. On the evaluation of groundwater contamination from underground nuclear tests. *Environmental Geology*, 42(2-3):235–247, 2002.

[AF96] S. F. Ashby and R. D. Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124:145–159, 1996.

[AM11] A. Atchley and R. M. Maxwell. Influences of subsurface heterogeneity and vegetation cover on soil moisture, surface temperature, and evapotranspiration at hillslope scales. *Hydrogeology Journal*, 19(2):289–305, 2011. doi:10.1007/s10040-010-0690-1.

[AMNS13a] A. L. Atchley, R. M. Maxwell, and A. K. Navarre-Sitchler. Human health risk assessment of co 2 leakage into overlying aquifers using a stochastic, geochemical reactive transport approach. *Environmental Science and Technology*, 47:5954–5962, 2013. doi:10.1021/es400316c.

[AMNS13b] A. L. Atchley, R. M. Maxwell, and A. K. Navarre-Sitchler. Using streamlines to simulate stochastic reactive transport in heterogeneous aquifers: kinetic metal release and transport in co2 impacted drinking water aquifers. *Advances in Water Resources*, 52:93–106, 2013. doi:10.1016/j.advwatres.2012.09.005.

[CM13] L. E. Condon and R. M. Maxwell. Implementation of a linear optimization water allocation algorithm into a fully integrated physical hydrology model. *Advances in Water Resources*, 60:135–147, 2013. doi:10.1016/j.advwatres.2013.07.012.

[CM14a] L. E. Condon and R. M. Maxwell. Feedbacks between managed irrigation and water availability: diagnosing temporal and spatial patterns using an integrated hydrologic model. *Water Resources Research*, 50:2600–2616, 2014. doi:10.1002/2013WR014868.

[CM14b] L. E. Condon and R. M. Maxwell. Groundwater-fed irrigation impacts spatially distributed temporal scaling behavior of the natural system: a spatio-temporal framework for understanding water management impacts. *Environmental Research Letters*, 9:1–9, 2014. doi:10.1088/1748-9326/9/3/034009.

[CMG13] L. E. Condon, R. M. Maxwell, and S. Gangopadhyay. The impact of subsurface conceptualization on land energy fluxes. *Advances in Water Resources*, 60:188–203, 2013. doi:10.1016/j.advwatres.2013.08.001.

[con] Wikipedia contributors. Endianness — wikipedia, the free encyclopedia. URL: http://en.wikipedia.org/wiki/Endianness.

[CWM14] Z. Cui, C. Welty, and R. M. Maxwell. Modeling nitrogen transport and transformation in aquifers using a particle-tracking approach. *Computers and Geosciences*, 70:1–14, 2014. doi:10.1016/j.cageo.2014.05.005.

[DZD+03] Y. Dai, X. Zeng, R. E. Dickinson, I. Baker, G. B. Bonan, M. G. Bosilovich, A. S. Denning, P. A. Dirmeyer, G. Niu P. R., K. W. Oleson, C. A. Schlosser, and Z. L. Yang. The common land model. *The Bulletin of the American Meteorological Society*, 84(8):1013–1023, 2003.

[DMC10] M. H. Daniels, R. M. Maxwell, and F. K. Chow. An algorithm for flow direction enforcement using subgrid-scale stream location data. *Journal of Hydrologic Engineering*, 16:677–683, 2010. doi:10.1061/(ASCE)HE.1943-5584.0000340.

[dBRM09] F. P. J. de Barros, Y. Rubin, and R. M. Maxwell. The concept of comparative information yield curves and their application to risk-based site characterization. *Water Resources Research*, 2009. doi:10.1029/2008WR007324.

[EW96] S. C. Eisenstat and H. F. Walker. Choosing the forcing terms in an inexact newton method. *SIAM J. Sci. Comput*, 17(1):16–32, 1996.

[FM10] I. M. Ferguson and R. M. Maxwell. Role of groundwater in watershed response and land surface feedbacks under climate change. *Water Resources Research*, 2010. doi:10.1029/2009WR008616.

[FWP95] P. A. Forsyth, Y. S. Wu, and K. Pruess. Robust numerical methods for saturated-unsaturated flow with dry initial conditions. *Advances in Water Resources*, 17:25–38, 1995.

[FFKM09] S. Frei, J. H. Fleckenstein, S. J. Kollet, and R. M. Maxwell. Patterns and dynamics of river-aquifer exchange with variably-saturated flow using a fully-coupled model. *Journal of Hydrology*, 375(3–4):383–393, 2009. doi:10.1016/j.jhydrol.2009.06.038.

[HV81] R. Haverkamp and M. Vauclin. A comparative study of three forms of the Richard equation used for predicting one-dimensional infiltration in unsaturated soil. *Soil Sci. Soc. of Am. J*, 45:13–20, 1981.

[JW01] J. E. Jones and C. S. Woodward. Newton-krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems. *Advances in Water Resources*, 24:763–774, 2001.

[KMW+13] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, and others. Multiphysics simulations: challenges and opportunities. *International Journal of High Performance Computing Applications*, 27:4–83, 2013. doi:10.1177/1094342012468181.

[KRM11] J. B. Kollat, P. M. Reed, and R. M. Maxwell. Many-objective groundwater monitoring network design using bias-aware ensemble kalman filtering, evolutionary optimization, and visual analytics. *Water Resources Research*, 2011. doi:10.1029/2010WR009194.

[KCSchauttemeyer+09] S. J. Kollet, I. Cvijanovic, D. Schaüttemeyer, R. M. Maxwell, A. F. Moene, and P. Bayer. The influence of rain sensible heat, subsurface heat convection and the lower temperature boundary condition on the energy balance at the land surface. *Vadose Zone Journal*, 2009. doi:10.2136/vzj2009.0005.

[KM06] S. J. Kollet and R. M. Maxwell. Integrated surface-groundwater flow modeling: a free-surface overland flow boundary condition in a parallel groundwater flow model. *Advances in Water Resources*, 29:945–958, 2006.

[KM08a] S. J. Kollet and R. M. Maxwell. Capturing the influence of groundwater dynamics on land surface processes using an integrated, distributed watershed model. *Water Resources Research*, 2008.

[KM08b] S. J. Kollet and R. M. Maxwell. Demonstrating fractal scaling of baseflow residence time distributions using a fully-coupled groundwater and land surface model. *Geophysical Research Letters*, 2008.

[KMW+10] S. J. Kollet, R. M. Maxwell, C. S. Woodward, S. G. Smith, J. Vanderborght, H. Vereecken, and C. Simmer. Proof-of-concept of regional scale hydrologic simulations at hydrologic resolution utilizing massively parallel computer resources. *Water Resources Research*, 2010. doi:10.1029/2009WR008730.

[Max10] R. M. Maxwell. Infiltration in arid environments: spatial patterns between subsurface heterogeneity and water-energy balances. *Vadose Zone Journal*, 9:970–983, 2010. doi:10.2136/vzj2010.0014.

[Max13] R. M. Maxwell. A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling. *Advances in Water Resources*, 53:109–117, 2013. doi:10.1016/j.advwatres.2012.10.001.

[MCT00] R. M. Maxwell, S. F Carle, and A. F. B. Tompson. Risk-based management of contaminated groundwater: the role of geologic heterogeneity, exposure and cancer risk in determining the performance of aquifer remediation, in. *Proceedings of Computational Methods in Water Resources XII*, pages 533–539, 2000.

[MCT08] R. M. Maxwell, S. F. Carle, and A. F. B. Tompson. Contamination, risk, and heterogeneity: on the effectiveness of aquifer remediation. *Environmental Geology*, 54:1771–1786, 2008.

[MCK07] R. M. Maxwell, F. K. Chow, and S. J. Kollet. The groundwater-land-surface-atmosphere connection: soil moisture effects on the atmospheric boundary layer in fully-coupled simulations. *Advances in Water Resources*, 30(12):2447–2466, 2007.

[MK08a] R. M. Maxwell and S. J. Kollet. Interdependence of groundwater dynamics and land-energy feedbacks under climate change. *Nature Geoscience*, 1(10):665–669, 2008.

[MK08b] R. M. Maxwell and S. J. Kollet. Quantifying the effects of three-dimensional subsurface heterogeneity on hortonian runoff processes using a coupled numerical, stochastic approach. *Advances in Water Resources*, 31(5):807–817, 2008.

[MLM+11] R. M. Maxwell, J. K. Lundquist, J. D. Mirocha, S. G. Smith, C. S. Woodward, and A. F. B. Tompson. Development of a coupled groundwater-atmospheric model. *Monthly Weather Review*, 2011. doi:10.1175/2010MWR3392.

[MM05] R. M. Maxwell and N. L. Miller. Development of a coupled land surface and groundwater model. *Journal of Hydrometeorology*, 6(3):233–247, 2005.

[MTK09] R. M. Maxwell, A. F. B. Tompson, and S. J. Kollet. A serendipitous, long-term infiltration experiment: water and tritium circulation beneath the cambric trench at the nevada test site. *Journal of Contaminant Hydrology*, 108(1-2):12–28, 2009. doi:10.1016/j.jconhyd.2009.05.002.

[MWH07] R. M. Maxwell, C. Welty, and R. W. Harvey. Revisiting the cape cod bacteria injection experiment using a stochastic modeling approach. *Environmental Science and Technology*, 41(15):5548–5558, 2007.

[MWT03] R.M. Maxwell, C. Welty, and A.F.B. Tompson. Streamline-based simulation of virus transport resulting from long term artificial recharge in a heterogeneous aquifer. *Advances in Water Resources*, 22(3):203–221, 2003.

[MM11] S. B. Meyerhoff and R. M. Maxwell. Quantifying the effects of subsurface heterogeneity on hillslope runoff using a stochastic approach. *Hydrogeology Journal*, 19:1515–1530, 2011. doi:10.1007/s10040-011-0753-y.

[MMGW14] S. B. Meyerhoff, R. M. Maxwell, W. D. Graham, and J. L. Williams. Improved hydrograph prediction through subsurface characterization: conditional stochastic hillslope simulations. *Hydrogeology Journal*, 2014. doi:10.1007/s10040-014-1112-6.

[MMR+14] S. B. Meyerhoff, R. M. Maxwell, A. Revil, J. B. Martin, M. Karaoulis, and W. D. Graham. Characterization of groundwater and surface water mixing in a semiconfined karst aquifer using time-lapse electrical resistivity tomography. *Water Resources Research*, 50:2566–2585, 2014. doi:10.1002/2013WR013991.

[MMF+13] K. M. Mikkelson, R. M. Maxwell, I. Ferguson, J. D. Stednick, J. E. McCray, and J. O. Sharp. Mountain pine beetle infestation impacts: modeling water and energy budgets at the hill-slope scale. *Ecohydrology*, 2013. doi:10.1002/eco.278.

[RMC10] J. Rihani, R. M. Maxwell, and F. K. Chow. Coupling groundwater and land-surface processes: idealized simulations to identify effects of terrain and subsurface heterogeneity on land surface energy fluxes. *Water Resources Research*, 2010. doi:10.1029/2010WR009111.

[SSM+14] P. Shrestha, M. Sulis, M. Masbou, S. Kollet, and C. Simmer. A scale-consistent terrestrial systems modeling platform based on cosmo, clm and parflow. *Monthly Weather Review*, 2014. doi:10.1175/MWR-D-14-00029.1.

[SM12a] E. R. Siirila and R. M. Maxwell. A new perspective on human health risk assessment: development of a time dependent methodology and the effect of varying exposure durations. *Science of The Total Environment*, 431:221–232, 2012. doi:10.1016/j.scitotenv.2012.05.030.

[SM12b] E. R. Siirila and R. M. Maxwell. Evaluating effective reaction rates of kinetically driven solutes in large-scale, statistically anisotropic media: human health risk implications. *Water Resources Research*, 48:1–23, 2012. doi:10.1029/2011WR011516.

[SNSMM12] E. R. Siirila, A. K. Navarre-Sitchler, R. M. Maxwell, and J. E. McCray. A quantitative methodology to assess the risks to human health from co2 leakage into groundwater. *Advances in Water Resources*, 36:146–164, 2012. doi:10.1016/j.advwatres.2010.11.005.

[SMP+10] M. Sulis, S. Meyerhoff, C. Paniconi, R. M. Maxwell, M. Putti, and S. J. Kollet. A comparison of two physics-based numerical models for simulating surface water-groundwater interactions. *Advances in Water Resources*, 33(4):456–467, 2010. doi:10.1016/j.advwatres.2010.01.010.

[TAG89] A. F. B. Tompson, R. Ababou, and L. W. Gelhar. Implementation of of the three-dimensional turning bands random field generator. *Water Resources Research*, 25(10):2227–2243, 1989.

[TBP99] A. F. B. Tompson, C. J. Bruton, and G. A. Pawloski. Evaluation of the hydrologic source term from underground nuclear tests in frenchman flat at the nevada test site: the cambric test. *Lawrence Livermore National Laboratory, Livermore, CA (UCRL-ID-132300), 360pp*, 1999.

[TCRM99] A. F. B. Tompson, S. F. Carle, N. D. Rosenberg, and R. M. Maxwell. Analysis of groundwater migration from artificial recharge in a large urban aquifer: a simulation perspective. *Water Resources Research*, 35(10):2981–2998, 1999.

[TFS+98] A. F. B. Tompson, R. D. Falgout, S. G. Smith, W. J. Bosl, and S. F. Ashby. Analysis of subsurface contaminant migration and remediation using high performance computing. *Advances in Water Resources*, 22(3):203–221, 1998.

[TMC+05] A. F. B. Tompson, R. M. Maxwell, S. F. Carle, M. Zavarin, G. A. Pawloski, and D. E. Shumaker. Evaluation of the non-transient hydrologic source term from the cambric underground nuclear test in frenchman flat, nevada test site. Technical Report, Lawrence Livermore National Laboratory, Livermore, CA, UCRL-TR-217191, 2005.

[VG80] M.Th. Van Genuchten. A closed form equation for predicting the hydraulic conductivity of unsaturated soils. *Soil Sci. Soc. Am. J*, 44(5):892–898, 1980.

[WM11] J. L. Williams and R. M. Maxwell. Propagating subsurface uncertainty to the atmosphere using fully coupled stochastic simulations. *Journal of Hydrometeorology*, 12:690–701, 2011. doi:10.1175/2011JHM1363.1.

[WMM13] J. L. Williams, R. M. Maxwell, and L. D. Monache. Development and verification of a new wind speed forecasting system using an ensemble kalman filter data assimilation technique in a fully coupled hydrologic and atmospheric model. *Journal of Advances in Modeling Earth Systems*, 5:785–800, 2013. doi:10.1002/jame.20051.

[Woo98] C. S. Woodward. A newton-krylov-multigrid solver for variably saturated flow problems. In *Proceedings of the XIIth International Conference on Computational Methods in Water Resources*. June 1998.

[WGM02] C. S. Woodward, K. E. Grant, and R. M. Maxwell. Applications of sensitivity analysis to uncertainty quantification for variably saturated flow. In *Proceedings of the XIVth International Conference on Computational Methods in Water Resources, Amsterdam, The Netherlands*. June 2002.