SANDIA REPORT
SAND2021-7191
Printed June 2021

Sandia National Laboratories

Multicontinuum Flow Models for Assessing Two-Phase Flow in Containment Science

Kristopher L. Kuhlman and Jason E. Heath

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods/

ABSTRACT

We present a new pre-processor tool written in Python that creates multicontinuum meshes for PFLOTRAN to simulate two-phase flow and transport in both the fracture and matrix continua. We discuss the multicontinuum modeling approach to simulate potentially mobile water and gas in the fractured volcanic tuffs at Aqueduct Mesa, at the Nevada National Security Site.

ACKNOWLEDGEMENTS

This research was funded by the National Nuclear Security Administration, Defense Nuclear Nonproliferation Research and Development (NNSA DNN R&D). The authors acknowledge important interdisciplinary collaboration with scientists and engineers from LANL, LLNL, MSTS, PNNL, and SNL. The authors thank Rick Jayne for technically reviewing the report.

CONTENTS

LIST OF FIGURES

ACRONYMS AND DEFINITIONS

Abbreviation	Definition
CASH	Campbell-Shashkov hydrocode
CTRW	continuous time random walk
FEHM	Finite Element Heat and Mass Transfer (fehm.lanl.gov)
GDKM	generalized dual permeability model
GFM	Geologic Framework Model
HDF5	Hierarchical data format, version 5
LANL	Los Alamos National Laboratory
MINC	Multiple Interacting Continua
NNSS	Nevada National Security Site
PFLOTRAN	Parallel Flow and Transport (www.pflotran.org)
STOMP	Subsurface Transport of Multiple Phases (www.pnnl.gov/projects/stomp)
TOUGH2	Transport of Unsaturated Groundwater and Heat (tough.lbl.gov)
UNE	underground nuclear explosion
UNPWT	Upper non- to partially welded tuff (GFM unit)
UWT	Upper welded tuff (GFM unit)
UZNT	Upper zeolitic nonwelded tuff (GFM unit)
VWT	Vitric nonwelded tuff (GFM unit)

1. INTRODUCTION

This report introduces a newly developed tool for creating generalized multicontinuum (e.g., double-porosity, triple porosity, or double-permeability) PFLOTRAN meshes to simulate two-phase miscible flow of water and gas with energy (i.e., heat conduction and convection, including phase changes). The multicontinuum approach allows for multiphase flow through domains whose properties can be assigned to represent the heterogeneous matrix of host rock lithologies, pre-existing natural fractures, induced fractures from some dynamic process, and potentially other features such as zones with loss of porosity. At Aqueduct Mesa, on the Nevada National Security Site (NNSS), it is well-known that a range of gases and liquids permeate the rocks and flow occurs through both the fractured and intact (i.e., matrix) components of the rock (Heath et al., 2021). Numerical modeling of gas and radionuclide signatures in the complex geology of Aqueduct Mesa requires a flexible multicontinuum approach to account for possibly simultaneous flow of water and gas within natural and induced fractures that connect to the heterogeneous host rock lithologies.

The multicontinuum meshing tool is a Python script designed to work with PFLOTRAN (Hammond et al., 2014), but it has been constructed to be fairly general and could be modified in a future version to generate unstructured meshes compatible with FEHM, TOUGH, or other Voronoi-based simulators.

See our previous work for a detailed summary of relevant hydrogeology of fractured volcanic rocks (i.e., mostly volcanic tuff) at Aqueduct Mesa, including a literature review of fractured rock properties (Kuhlman et al., 2020; Heath et al., 2021). In this report we emphasize the multicontinuum conceptual model and discuss the model implementation details as opposed to focusing on the geology and hydrogeology of Aqueduct Mesa.

1.1. Fractured Tuff Conceptual Models

For two-phase fluid flow in fractured rocks, there are several viable conceptual models (NRC, 2001). Altman et al. (1996) presents a clear graphical summary of characterization approaches for multi-phase fluid flow in volcanic tuffs at Yucca Mountain (Figure 1), which is approximately 50 km south-southeast from Aqueduct Mesa. Yucca Mountain rocks are similar to those at Aqueduct Mesa. Significant effort was expended to measure, simulate, and understand two-phase flow in volcanic tuffs at Yucca Mountain (e.g., Doughty, 1999; Rechard et al. 2014) and analogue studies were performed in other volcanic tuff sites as part of the Yucca Mountain Project, including the Bishop Tuff in California (Evans & Bradbury, 2004) and the Apache Leap Site in Arizona (Neuman et al., 2001).

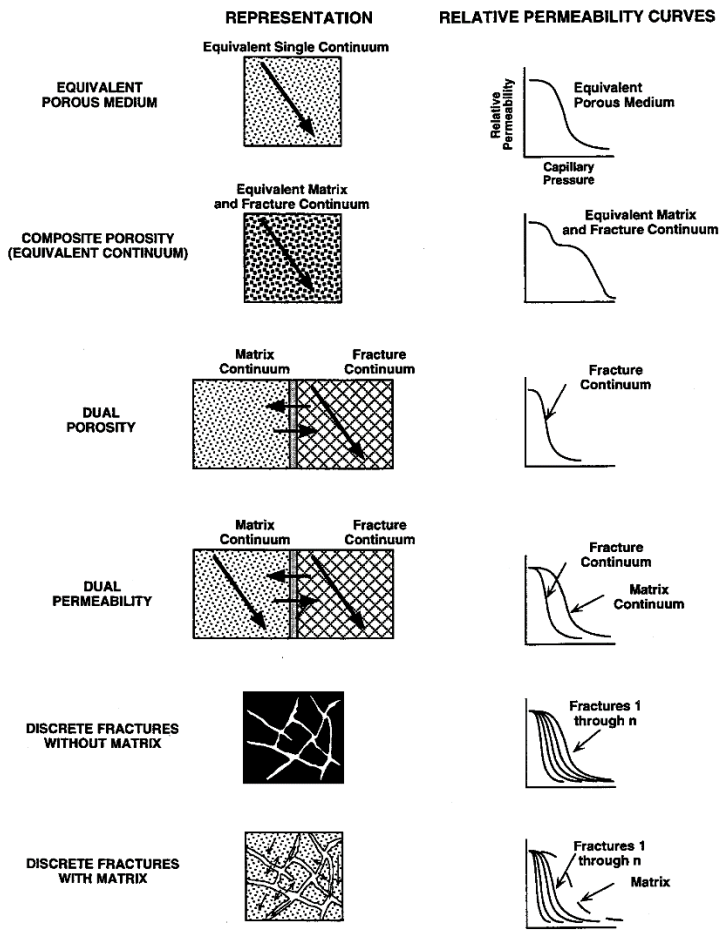

Figure 1. Hierarchy of conceptual models for multi-phase flow in fractured tuff at Yucca Mountain (Altman et al., 1996). Models range in complexity from simplest and easiest to parameterize (top) to most physically realistic and most demanding to parameterize (bottom).

1.1.1. Undisturbed Tuff Conceptual Models

This subsection on "undisturbed" conceptual models focuses on the matrix and natural fractures of tuff lithologies and not on induced fractures from dynamic processes such as underground nuclear explosions (UNEs). A single-porosity equivalent porous medium conceptual model is the simplest approach (top of Figure 1). A single-porosity model requires the fewest parameters and is the grossest approximation to fracture/matrix flow (i.e., only valid in the limit as flow in one of the continua can be ignored). At the bottom, the porous discrete fracture network, with each fracture represented within a permeable matrix, is considered the most physically realistic. The most complex approach is considered the most difficult to parameterize and simulate. Approaches of intermediate complexity include multi-continuum approaches like dual porosity or dual permeability (e.g., Warren and Root, 1963; Kazemi, 1969; Kuhlman et al., 2015; the terms dual porosity and dual permeability are further explained in Section 1.2), featured in the middle of the figure.

Despite Figure 1 being 25 years old, it still essentially summarizes the range conceptual models for two-phase flow in fractured rocks, with one possible modern extension/addition for discrete fault networks in an equivalent porous medium. Recent work has pursued a hybrid approach, where a small number of discrete fractures or faults are embedded into an equivalent porous medium, using

either single or multiple porosity (White et al., 2020). This embedded approach is not conceptually different from the options already in Figure 1, but represents the logical progression in the search for an optimal balance of computational efficiency (i.e., fewer unknowns to solve for in a flow domain), fewer parameters to estimate or specify (i.e., not including every fracture in the system), and physical realism (i.e., representing mapped faults or fractures that may be fast pathways). Reeves et al. (2014) used a similar "fracture continuum" method for simulating multiphase fluid flow at Rainier Mesa, which is adjacent to Aqueduct Mesa and which contains more significant faulting. Reeves et al.'s fracture continuum approach includes discontinuous, discrete fault networks that are incorporated into a dual-permeability formulation. Discrete mapped features could also be added to the current multicontinuum framework presented here.

The Aqueduct Mesa conceptual model presented in Heath et al. (2021) essentially falls in the middle of this range of conceptual models. In this approach, multiple continua are considered, and both natural and induced fractures are treated, with possibly multiple types of matrix (damaged and intact), considering two-phase flow properties for each continuum. Thus, it is an expansion of the dual permeability conceptual model in the middle of Figure 1 to multiple porosity types that each can have mobile water and gas. Similar to our approach, Wu et al. (2004) and Liu et al. (2003) present triple-porosity continuum approaches for two-phase flow in fractured volcanic tuff. These two similar approaches consider fractures, matrix and either "small fractures" (Wu et al., 2004) or vugs/macropores (Liu et al., 2003) as the third continuum. Damage to the matrix could change the properties of the matrix and produce new fractures, which may be expected to have a very different distribution and properties than natural fractures (although few data are available to parameterize the induced fractures). The multi-porosity approach could be extended to include any other porosities deemed important (e.g., "small fractures" or vugs), but is presented here in its simplest form, we plan to start with simple approaches and increase complexity of the model as justified by laboratory or field observations.

1.1.2. Damaged Tuff Conceptual Models

Representation of rock deformation from UNEs in numerical models in terms of multiphase fluid flow properties is needed for predicting gas and radionuclide migration to earth's surface. Such damage may create new fractures and hence increase permeability, or it may crush the rock matrix and decrease porosity and permeability. Dynamic rock deformation is not included in work from Yucca Mountain but is obviously important to numerical modeling of Rainier and Aqueduct Mesas. Jordan et al. (2015) presents a workflow that relates a rock damage model to hydrologic parameters that are then incorporated into a gas transport model to predict gas arrival times to earth's surface under barometric pumping. The rock deformation model is a hydrocode named CASH (CAmpbell-SHashkov), which can model dynamic events including shocks, and, in the case of modeling for tuff and granite, includes compressive damage around an explosive charge and tensile cracking. For more details on the rock damage modeling, please see the methods and supplemental material of Jordan et al. (2015). They used FEHM for gas transport, which includes a Generalized Dual permeability (K) Model (GDKM) for flow and transport between matrix-fracture, fracture-fracture, and matrix-matrix materials in the simulation domain.

Of key interest here is mapping between rock damage and fluid flow properties to be used by the flow and transport simulator. Jordon et al. (2015) uses a nearest-neighbor (non-averaging) approach with overlapping Voronoi regions on a separate grid used to track mechanical damage and the FEHM computational mesh, with the greatest overlapping area being where the damage is assigned

to the FEHM mesh. A fracture aperture parameter is mapped to the damage—cutoffs are assigned to determine if a value of damage maps to a fracture permeability using the cubic law or an equivalent continuum model permeability that represents matrix and/or fracture permeability combined with the matrix that is below the damage-grid scale. This approach does not simply volume-average permeabilities as that process would lead to vastly underpredicted fluid flow. Volumetrically, fractures may be a small part of the medium, but they can be orders of magnitude more permeable and contribute to fast flow.

1.2. Review of Processes in Multiple Continua

The idea of a single rock being conceptualized as being constructed of multiple overlapping continua started with a conceptual model of Barenblatt & Zheltov (1960), but the first widely used analytical solution for the problem of flow to a pumping well in the petroleum engineering literature is associated with Warren & Root (1963). It is common (though not universal) to refer to fracture flow (high permeability with little storage capacity) and matrix diffusion (no permeability and high storage capacity) as "double porosity" (although Figure 1 refers to this as "dual porosity"). A more general flow system with variable permeability and connectivity in both the fractures and the matrix is commonly called "double permeability" (Figure 1 refers to it as "dual permeability").

Warren & Root's (1963) solution was a "diffusion of excess fluid pressure" solution for flow in a single-phase reservoir (i.e., slightly compressible fluids like oil or water). The authors developed their solution to match observed recovery data from production wells with multiple slopes on log-time scale – an early slope associated with fast fracture flow and a late-time slope associated with slower matrix diffusion. The Warren & Root (1963) solution simplifies flow from the matrix to the fracture to be proportional to a difference in pressure between two reservoirs, rather than considering spatially variable pressure within the matrix. A large number of analytical and numerical solutions extended the Warren & Root solution for different boundary conditions (Gringarten, 1982), additional porosities (Clossman, 1975), fracture-matrix skin (Moench, 1984), transient flow in the matrix with pressure gradients (Kazemi, 1969), and potentially infinite distributions of porosities (Kuhlman et al., 2015). Zimmerman et al. (1993) presented a hybrid analytical/numerical method, where flow in the fractures is treated numerically and diffusive flow in the matrix is predicted using an analytical solution similar to Warren & Root. This historical hybrid solution is mentioned because it is typical of the quest to find a balance between computational economy and geometric/physical realism.

Diffusion of solute into a set of overlapping continua was also treated first in the petroleum engineering literature by Coats & Smith (1964). The authors developed their model to explain anomalous "long tail" tracer breakthrough observations in numerous single-phase flow problems. The tracer slowly diffuses into the matrix at early times when the concentration within the fracture is high, and later slowly diffuses out of the matrix when the concentration in the fracture has flushed out and is low again. In this case, diffusion of solute into an impermeable matrix was considered a "capacitance" term in the problem, drawing an analogy with electrical circuits. This process is often referred to as "multirate mass transport" or "mobile-immobile transport," since the matrix only has diffusion (immobile) and the fracture is mainly advection (mobile). Analogous to the flow problem, many extensions of the initial approach have been derived, including sorbing mass transfer with multiple sites (van Genuchten & Wierenga, 1976) and an extension to solute transport between a distribution of fractures and a potentially infinite distribution of porosities (Haggerty & Gorelick, 1995).

Outside of hydrology and petroleum engineering, the use of multiple overlapping continua has seen wide application to unrelated problems. In solid state physics, a similar theory of multiple trapping states was developed to predict small-scale charge transport in disordered materials (Noolandi, 1977; Scher et al., 1991). In mathematics, the diffusion in disordered, fractal, or comb-shaped media also can be approximated using similar solutions (Havlin & Ben-Avarham, 1987).

These diffusion-only approaches (i.e., not double permeability) can be generalized to a system where a diffusion-limited reservoir (i.e., the matrix) exists that can be simplified down to behaving like a source or sink from the view of the adjacent fast-flowing fractures. The diffusion-limited behavior results in a simplistic representation in terms of convolution of "shape factors," which are simple products in Laplace-transform space. Any of these diffusive systems can also be shown to be equivalent with the continuous time-domain random walk (CTRW) statistical physics model (Montroll & Weiss, 1965), which is a random walk (i.e., Brownian motion) with an additional random wait for a random length of time. From the point of view of solute transport in a fracture, the matrix can be thought of as a mechanism that slows down changes in time (reduces quick rises in concentration and delays quick declines in concentration). All these approaches can also be related to a generalized diffusion equation with fractional time derivatives (Oldham & Spanier, 1970; Schumer et al. 2003), which represents deviations of fracture/matrix flow from the ideal fracture-only behavior as fracture flow governed by non-integer-order derivatives.

This short historical review provides some context for the wide range of methods and applications used to treat "multiple overlapping continua" systems and illustrates this is a problem of real-world interest beyond the application of the current two-phase flow system on Aqueduct Mesa. Multicontinuum solutions are widely used, sometimes in applications where they are not physically motivated. Some researchers consider multi-rate mass transport and CTRW solutions in heterogeneous (but otherwise single porosity) domains to be a type of proxy or "fitting" model (Fiori et al., 2015). It is true that the methods may be applied in some areas where they are not justified, but this also shows the methods are a widely used tool to solve flow in complex systems. Rather than being limited to the simplistic diffusion-limited types of problems, the multicontinuum approach we present here is physically motivated and can include non-linear two-phase flow and multiple permeabilities as well as diffusive porosities. These applications cannot be treated using simple diffusion-only type analytical solutions, and in our application are physically motivated by the presence of heterogeneous fractured and unfractured rock.

We present here a system that allows for multiple-continuum advection and diffusion of fluid pressure, two miscible and competing phases (with the capillary pressure relationships between phases specified individually for each continuum), with advection and diffusion of gases and energy through the fractured system. This system is a synthesis and natural extension of the many solutions in the literature summarized in this section. After introducing the mechanics of how the meshes are created, we present a small portion of the wide range of possible behaviors that could be expected when two-phase advection and diffusive transport are included. Numerical models are being developed to deepen our understanding of the system to better make predictions in a new situation, not just to match observations made in the laboratory or field.

2. MESH DEVELOPMENT

2.1. Background

Numerical simulations require construction of a solution mesh. Numerical models based on the finite-element method (e.g., COMSOL, Sierra Mechanics) typically build meshes out of triangles (2D) or tetrahedra (3D). Numerical models based on the finite-volume or integrated finite-differences approaches (e.g., TOUGH2, FEHM, PFLOTRAN) usually use Voronoi meshes (in 2D or 3D). A uniform mesh of boxes (i.e., rectangles in 2D, hexahedra in 3D) is the simplest form of Voronoi mesh, but more complex Voronoi meshes can be constructed for arbitrary geometry indirectly from more common tetrahedra meshes or directly using algorithms like VoroCrust (Abdelkader et al., 2020).

For Voronoi elements, each element includes all the area or volume closest to the center of the element than any other element center. The line connecting the centers of adjacent elements is then by construction perpendicular to the element face. One way of constructing Voronoi elements is to first construct a Delaunay tetrahedral mesh, and then the Voronoi mesh is the "dual" of the tetrahedral mesh (i.e., corners of tetrahedra becomes centers of Voronoi elements, and the faces of Voronoi elements are perpendicular to the edges of the tetrahedra, located halfway between each node; Figure 2). Triangles and tetrahedra are commonly used because they always have the same number of sides, and they naturally define a convex hull around any group of points. Voronoi elements can have as few as 4 sides in 2D, but often have many more, and require additional specification of the convex hull (since they are specified by their centers, rather than their corners). PFLOTRAN uses Voronoi meshes, and the multicontinuum tool currently works with hexahedra meshes, but future versions of the tool will be able to apply multicontinuum methods to arbitrary Voronoi meshes.

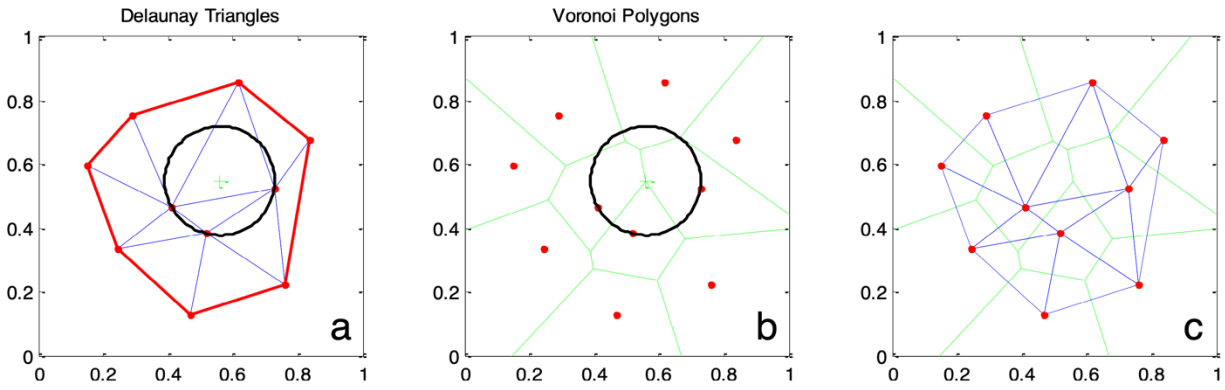

Figure 2. Delaunay triangulation and Voronoi polygons for 10 random points in 2D.

PFLOTRAN documentation is available in website form from the main PFLOTRAN website (http://pflotran.org). Currently the documentation for the development version is at https://doc-dev.pflotran.org, in the user's guide. Keywords in the following sections are specified in all caps, but this is only a convention (not required) in PFLOTRAN input files.

PFLOTRAN is based on the finite-volume method (requiring Voronoi elements for mass balance), and it has three major ways of specifying solution meshes. At the simplest end, "structured" meshes are made of hexahedra specified using three vectors of Δx, Δy, and Δz, for cartesian, cylindrical, or spherical geometry. The cartesian product of these three vectors and the geometry type gives a

simple grid of boxes. In the middle (complexity-wise), an "implicit unstructured" mesh is specified by listing the corners of each element. From these corners, PFLOTRAN geometrically computes the element volumes, connectivity between elements (i.e., determining which elements touch one another), and interfacial areas. The most complex and most flexible way to specify a mesh in PFLOTRAN is the "explicit unstructured" mesh. In this case, the user specifies the coordinates of element centers, element volumes, and the connectivity between elements (including each connection's location and perpendicular area). Simple meshes can be represented equivalently using any three of these methods, with the structured mesh approach resulting in the simplest PFLOTRAN input files. More complex geometries, including multicontinuum meshes, can only be represented using an explicit unstructured mesh, since multiple elements can be located at the same physical location. Implicit unstructured meshes cannot be used for multicontinuum problems, since proximity is the mechanism for specifying connectivity in these meshes, and the connectivity of physically coincident elements cannot be inferred from geometry alone.

The simplest structured meshing approach also has the simplest methods for specifying regions (points, areas, or volumes) that are used to assign material properties (element volumes), boundary conditions (areas or element faces), sources and sinks (element volumes), or observation locations (points at centers of elements). Figure 3 illustrates how boxes (specified via a list of 6 COORDINATES: $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, $z_{\min}$, $z_{\max}$ or a BLOCK of 6 indices $i_{\min}$, $i_{\max}$, $j_{\min}$, $j_{\max}$, $k_{\min}$, $k_{\max}$) can be used to indicate regions. Surfaces (i.e., faces) can be specified using simple word/compass directions (i.e., north, south, east, west, top, bottom), like CARTESIAN_BOUNDARY.

```
GRID
  TYPE STRUCTURED
  NXYZ 5 4 7
  BOUNDS
    0.0   2.0  0.0 # x,y,z min
    100. 50.0 25.0 # x,y,z max
  END
END
```

```
REGION pumping_well
  COORDINATES
    20.0 12.0 3.57 # x,y,z min
    40.0 24.0 7.14 # x,y,z max
  END
END
```

```
REGION downstream
  CARTESIAN_BOUNDARY EAST
END
```

```
REGION source
  BLOCK 6 6  1 1  1 1 # (min,max) i,j,k
END
```

Figure 3. Examples of PFLOTRAN structured mesh grid specifications and region specifications.

Fewer of these convenience keywords exist in PFLOTRAN when using the explicit unstructured mesh input approach. Region volumes are specified using lists of integer element ID numbers, and boundary conditions are specified using lists of connections (i.e., the connection between element centers and element faces where boundary conditions are applied). The flexibility and generality of the explicit unstructured approach means it is typically only used with a front-end or pre-processor (e.g., dfnWorks: Hyman et al., 2015; https://dfnworks.lanl.gov), since manually constructing all but the smallest explicit unstructured meshes is tedious and error prone.

PFLOTRAN has double-porosity capability, called SECONDARY_CONTINUUM, the theory of which is documented at https://doc-dev.pflotran.org/theory_guide/multiple_continuum.html. This existing secondary continuum approach can only simulate matrix diffusion of solute or temperature (no advection of water or gas, the intrinsic permeability is effectively assumed zero)—this approach is only for diffusion-limited matrix processes. These diffusion-only capabilities in the secondary

continuum approach in PFLTORAN are implemented independently of the main flow and transport modes in PFLOTRAN. Any new heat or solute transport features needed in the existing double-porosity approach (e.g., sorption or radioactive decay in solute diffusion) must be implemented anew and verified against the similar capability in the main part of PFLOTRAN. The primary advantage of this limited (i.e., diffusion only) double-porosity approach is execution speed; the approach takes advantage of the diffusion-limited assumption and makes significant simplifications to speed up solution.

2.2. Meshing to Represent Multiple Continua

There are a range of different approaches for conceptualizing, and subsequently implementing, multiphase flow through fractured rocks (Figure 1). Fractures and intact rock (i.e., porous matrix) both play important roles in flow through volcanic rocks at Aqueduct Mesa. Some fractures are natural and formed early (e.g., due to rapid cooling of the volcanic rocks immediately after deposition), some fractures may be natural and in response to regional tectonic stresses (i.e., both micro- and macro-fractures), and others are recent and due to human activities, including chemical and nuclear explosions. Recent fractures include damage associated with excavations (e.g., drifts, boreholes, and cavities) or explosions (e.g., shock damage or chimney collapse). Note that natural fractures within the P-Tunnel Complex have little displacement (< 1.5 m) as opposed to Rainier Mesa with faults/fractures with tens of meters of displacement (Drellack et al., 2011; Prothro, 2018).

We have developed a set of Python scripts to create explicit unstructured meshes for PFLOTRAN that represent the 1D, 2D, or 3D domains consisting of an arbitrary number of overlapping continua. Figure 4 illustrates a typical set of possibilities for a 1D physical domain with (a) a single matrix continuum, (b) a spatially connected fracture continuum linked to an un-connected matrix continuum (i.e., dual porosity), and two (c) or three (d) spatially connected continua (i.e., dual permeability).

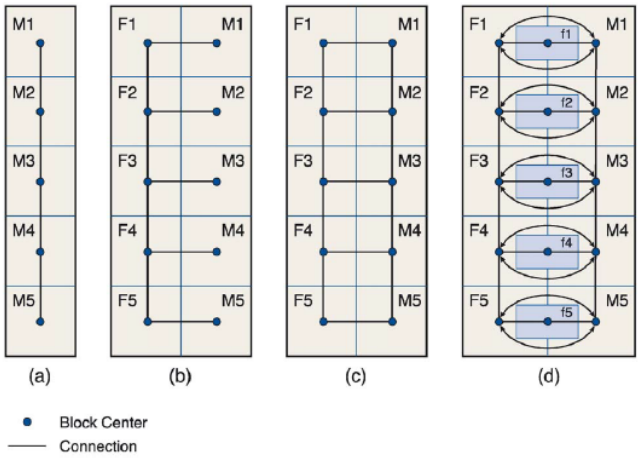

Figure 4. Example of mesh construction for multiple continuum approaches, M=matrix, F=major fractures, f=minor fractures (Wu et al., 2004). (a) is a single-porosity domain, (b) is a connected fracture domain with diffusion-limited dead-end transport into the matrix, (c) represents spatial connections in both the fracture and matrix, and (d) includes a third domain of minor fractures, which are not connected in space.

2.3. PFLOTRAN Multicontinuum Mesh Generation

The approach described here, inspired by the MINC (Multiple Interacting Continua) mesh pre-processor for TOUGH (Pruess & Narasimhan, 1982; Pruess, 1992), implements meshes for multi-continuum problems via the "explicit unstructured" mesh input specification approach in PFLOTRAN. The working version of this capability is tentatively called "PINC," starting off as a MINC implementation for PFLOTRAN. This approach can utilize any flow, reaction, transport or geophysical process already implemented in the "regular" part of PFLOTRAN across secondary, tertiary, or higher continua. Spatial connectivity can be implemented in the secondary continua (e.g., double permeability).

To implement this MINC-inspired mesh pre-processor for PFLOTRAN in a general way, we re-implement the "structured" meshing algorithm in PFLOTRAN as a Python script, and then we added in multicontinuum capabilities. This external approach was chosen, rather than modify the PFLOTRAN source code directly, because the PFLOTRAN source is significantly more complex because of its usage of MPI-based parallel mesh construction and PETSc for parallel solution (https://www.mcs.anl.gov/petsc). Although we implemented the PINC preprocessor outside the PFLOTRAN source code, we fixed several bugs or limitations in the spherical and explicit unstructured mesh capabilities of PFLOTRAN during the development of the tool. These fixes benefit all PFLOTRAN users, not just those using our multicontinuum approach.

The work to implement this proceeded in several steps.

1. The PINC script reads a PFLOTRAN input file and parses the GRID and REGION blocks (e.g., Figure 3). The script parses and error-checks these inputs.
2. The PINC script then builds an unstructured mesh that is equivalent to the specified structured mesh. The script also converts REGIONs specified on the explicit mesh to equivalent REGIONs for the unstructured mesh (i.e., for boundary conditions, material properties, and observation locations). As a check, the two models (initial structured mesh and explicit unstructured mesh) should be run to ensure the meshes produce equivalent results.
3. Next the script can read in variable elevations for layers in the mesh, if desired. The z-elevations of the unstructured mesh are modified, and a new explicit unstructured mesh is generated.
4. Lastly, the multicontinuum input file is read and the secondary continua are meshed and an explicit unstructured mesh that cumulatively has the effects of adjusted elevations and multiple continua (and the associated regions) is generated.

The variable steps for layer elevations (step 3) and multicontinuum (step 4) are optional.

2.3.1. Unstructured Mesh Generation

Unstructured mesh generation involved two requirements. First, we must generate the files needed by PFLOTRAN to run the simulation. Second, we must generate the files needed by visualization software to inspect/plot the output without significant user intervention. For structured and implicit unstructured meshes, PFLOTRAN writes both of these, but for explicit unstructured meshes, PFLOTRAN does not need or write the information required for plotting, so it must be handled separately.

Most of the information needed to build a solution mesh is read from the GRID block of the input file. For a structured mesh, the primary options are CARTESIAN, CYLINDRICAL, and SPHERICAL. For this application, we will only be using CARTESIAN (the default), but applications using the other types of mesh are useful for certain problems with symmetric geometries. During early stages of implementing the PINC tool, the authors fixed two errors in the spherical mesh implementation in PFLOTRAN (bug 1, bug 2). The explicit unstructured mesh is specified via a text file with a "uge" filename extension. This file contains two required sections CELLS and CONNECTIONS. The relationship between them is illustrated for a trivial 3-element 1D domain with two connections.

```
GRID
  TYPE STRUCTURED
  NXYZ 3 1 1
  DXYZ
    3@1.0 # dx
    1.0   # dy
    1.0   # dz
  END
END
```

```
CELLS 3
1 5.000E-01 5.000E-01 5.000E-01 1.00000000000E+00
2 1.500E+00 5.000E-01 5.000E-01 1.00000000000E+00
3 2.500E+00 5.000E-01 5.000E-01 1.00000000000E+00
CONNECTIONS 2
2 1 1.000E+00 5.000E-01 5.000E-01 1.00000000000E+00
3 2 2.000E+00 5.000E-01 5.000E-01 1.00000000000E+00
```

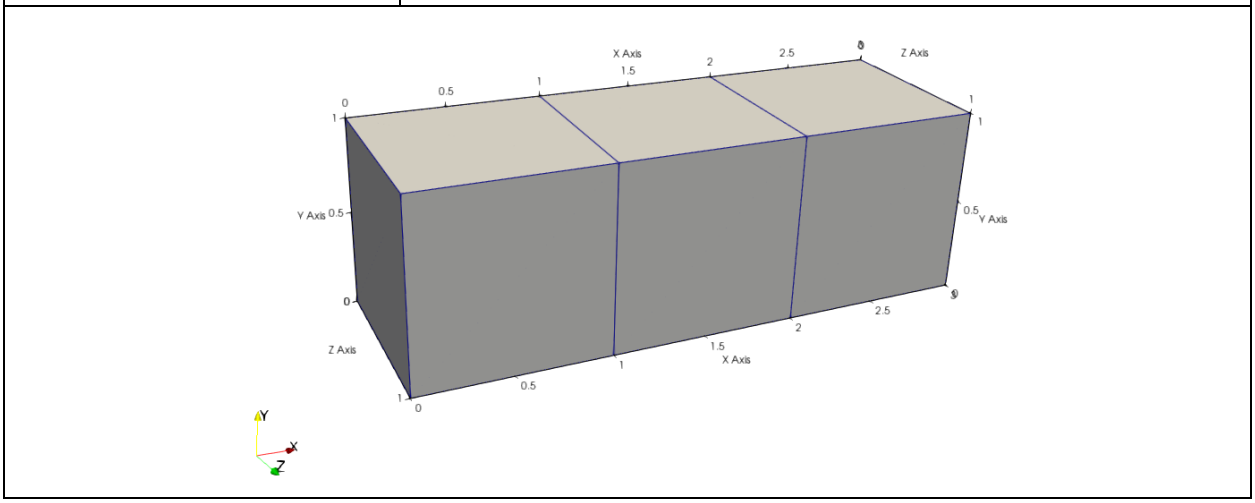

Figure 5. Example 3-element 1D domain; structured (top L) and unstructured (top R), domain visualized via ParaView in bottom image.

In the 3-element unstructured mesh example (Figure 5), each line of the CELLS section (after the header indicating the number of rows) has the following space-delimited information:

1. element identification number (ID, starting at 1),
2. element center x, y, z coordinate (m), and
3. element volume (m^3).

The CONNECTIONS section (after the header indicating the number of rows) lists the following information for each connection:

1. upstream element ID (referencing IDs listed in CELLS section),
2. downstream element ID,
3. connection midpoint x, y, z coordinate (m), and

4. connection area (m^2).

Consistent with the Voronoi-based element structure needed by PFLOTRAN, only the element center coordinates and the connections between elements are specified in the explicit unstructured grid specification. Even though it is not used by PFLOTRAN to simulate the problem, the coordinates of element corners and the mapping from corners to elements is needed to visualize the mesh or model output using typical tools like ParaView (http://www.paraview.org) or VisIt (https://wci.llnl.gov/simulation/computer-codes/visit).

The multicontinuum mesh generation script writes both the text PFLOTRAN input file (.uge) needed to run the mesh, but it also writes a hierarchical data format version 5 (hdf5) domain file that has the corners and mapping from corner nodes to elements, needed by visualization tools like ParaView. Hdf5 is a parallel-capable binary file format for writing inputs and outputs for PFLOTRAN (https://www.hdfgroup.org/solutions/hdf5/); we utilize a Python wrapper to the hdf5 library. The domain file includes a Domain group, with the several datasets. The datasets specify the mapping between vertices and cells (Cells), the coordinates of the element corners (Vertices), and the coordinates of the centers of the cells (XC, YC and ZC), see Figure 3.

To make the PFLOTRAN output from explicit unstructured mesh load automatically in ParaView, the following settings must be ensured in the PFLOTRAN input file:

1. In the GRID block, the DOMAIN_FILENAME option must point to the .h5 file generated by the PINC script;
2. In the OUTPUT block, the SNAPSHOT_FILE output should be set to FORMAT HDF5; and
3. In the OUTPUT block EXPLICIT_GRID_PRIMAL_GRID_TYPE CELL_CENTERED should be specified.

Previously, saving cell-centered flow velocities as output in hdf5 SNAPSHOT_FILE output format was not possible when using unstructured grids (only text VTK format for velocities was implemented). In January 2021 the authors added this feature to PFLOTRAN, which now exists in the mainline version of PFLOTRAN and benefits models created with the PINC tool and all users of explicit unstructured meshes (e.g., including users of dfnWorks, which also is a Python-based PFLOTRAN explicit unstructured mesh pre-processor). In February 2021 the authors also fixed a bug in the handling of filenames for the DOMAIN_FILENAME step, mentioned in the first bullet above, which now allows more typical filename lengths.

2.3.2. Unstructured Mesh Region Generation

Regions are used in PFLOTRAN to select a sub-part of the mesh for multiple purposes, including assigning material properties to elements, specifying boundary conditions, or indicating locations in the mesh where output observations should be saved. Specification of regions in PFLOTRAN structured meshes is straightforward, and the multicontinuum mesh generator creates equivalent explicit unstructured regions from structured ones.

Regions can be broadly divided into volume- or area-based. Volume regions are associated with one or more elements (specified by their centers) and can be used with a coupler to assign:

- material properties (i.e., MATERIAL_PROPERTY, tied to a region through a STRATA coupler);
- initial conditions (i.e., FLOW_CONDITION, tied to a region through an INITIAL_CONDITION coupler);

- volume-distributed sources and sinks of water, energy, or gas (i.e., FLOW_CONDITION, tied to a region through a SOURCE_SINK coupler); or
- observation points (OBSERVATION_FILE in the OUTPUT block, tied to a region through an OBSERVATION coupler) where output should be written through time.

Area regions are associated with faces of elements and are used through BOUNDARY_CONDITION couplers between regions and FLOW_CONDITIONS on the external faces of a model domain. The difference between area and volume regions is the presence of an optional FACE argument [WEST, EAST, SOUTH, NORTH, BOTTOM, TOP] in a COORDINATE or BLOCK REGION block. Leaving out the FACE makes these regions volumes. The CARTESIAN_BOUNDARY region can only be associated with a FACE.

In explicit unstructured meshes, volume regions are specified in text files (with a .txt extension) as lists of element IDs and area regions are specified in text files (with a .ex extension) with lists of connections. Connections are similar to how specified in the .uge file (Figure 5), but boundary connections don't have both an upstream and downstream element, so they have one fewer index.

The regions in an explicit unstructured mesh are specified with a "FILE filename.txt" line in a REGION block, which the PINC script will write for each region read in, to create an equivalent mesh to the initial structured mesh.

The multicontinuum model generator script creates regions files in the expected text file format, and additionally writes the same information into an .h5 file, specifying cell IDs and face IDs for each region.

2.3.3. Layer Elevation Specification

Once the mesh is created with the PINC Python script, it is stored in memory as a Python dictionary data structure. If desired, layer elevations in CARTESIAN meshes can be read in, which then modify the z-coordinates of the elements to create a model domain with deformed layers. The slope or curvature of any layers should be minimized (i.e., the deviation from flat layers should be small), since the orthogonality and mass-conservation of the mesh is changed. This can be useful in some situations, but often it is better to simply make a finer regular mesh to resolve changes in layers in space.

After outputting an explicit unstructured mesh that corresponds exactly to the structured mesh, the mesh is modified in place and exported again (with a different name). The output of the two meshes could be run by the user and compared, to quantify the impact deformation of layers has on the solution. The script generates the mesh for comparison, but the user best knows what quantities or locations to compare.

2.3.4. Multicontinuum Specification

The (possibly modified) explicit unstructured mesh is now modified by adding extra continua to the initial mesh. The specifications are given in a new PINC input block. This block is read from a separate file, since it is not valid PFLOTRAN input.

Similar to the MINC approach, a multicontinuum mesh is built from an initially specified mesh that is assumed to be the base "fracture" continuum (i.e., it is connected in space). Secondary (and

tertiary and higher) domains are connected physically to each element of the initial continuum, within the specified region.

```
PINC all
  TYPE DOUBLE_POROSITY
  VOLUME_FRACTIONS 0.05
  DOMAIN_LENGTHS 1.0 0.25
  NUMBER_ELEMENTS 1 10
  GEOMETRIES FRACTURE SLAB
  MESH
    FRACTURE
    DX 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
  END
  DIRECTIONS F +Y
END
```

```
PINC all
  TYPE DOUBLE_PERMEABILITY
  VOLUME_FRACTIONS 0.05
  DOMAIN_LENGTHS 1.0 0.25
  NUMBER_ELEMENTS 1 3
  GEOMETRIES FRACTURE SLAB
  MESH
    FRACTURE
    DX 0.1 0.2 0.3
  END
  DIRECTIONS F -Y
END
```

```
PINC north_half
  TYPE N_POROSITY 4
  VOLUME_FRACTIONS 0.05 0.4 0.2
  DOMAIN_LENGTHS 1.0 0.25 0.35 0.1
  NUMBER_ELEMENTS 1 3 4 5
  GEOMETRIES FRACTURE NESTED_CUBES NESTED_SPHERES SLAB
  MESH
    FRACTURE
    UNIFORM
    UNIFORM
    DX 0.1 0.2 0.3 0.4 0.5
  END
  DIRECTIONS F -Y +Y +Z
  SPATIAL_CONNECTIVITY 1 1 0 0
END
```

Figure 6. Three example multicontinuum specifications

The keywords follow a PFLOTRAN-like input format (indentation and capitalization of keywords is conventional but optional; unique names are case-sensitive). Each keyword is described below. The specifications in angled brackets "<>" indicate whether and how many integers (e.g., "2"), floating point numbers (e.g., "2.0"), or keywords (e.g., "DOUBLE_POROSITY") are expected.

PINC <region name>

This keyword opens a multicontinuum section. The case-sensitive region name indicates region the multicontinuum is associated with. Region can correspond to the entire domain, or a subset of the domain. Multiple PINC sections with overlapping regions may be used. The first fracture continuum is common between all regions. The block continues to an associated END keyword.

TYPE <keyword>

This defines the number of continua (N) and specifies connectivity. Valid keywords are:

- DOUBLE_POROSITY
- DOUBLE_PERMEABILITY
- TRIPLE_POROSITY
- TRIPLE_PERMEABILITY
- N_POROSITY <int>
- N_PERMEABILITY <int>

DOUBLE_{POROSITY,PERMEABILITY} implies N=2,
TRIPLE_{POROSITY,PERMEABILITY} implies N=3, and
N_{POROSITY,PERMEABILITY} allows specifying an arbitrary N.

The distinction between POROSITY and PERMEABILTY is related to the spatial connectivity of the continua. For example, DOUBLE_POROSITY is only connected spatially in the first (fracture) continuum (e.g., Figure 4b); the second continuum is dead-end diffusion into a matrix. DOUBLE_PERMEABILITY would have spatial connectivity in each continuum (e.g., Figure 4c).

Specifying the SPATIAL_CONNECTIVITY input over-rides any connectivity or lack of connectivity implied by these keywords (the value of N from TYPE is still used).

VOLUME_FRACTIONS <N-1 floats>

This defines the volume fraction for all but one of the continua. The constraint that the volume fractions must sum to 1 automatically specifies the last one.

DOMAIN_LENGTHS <N floats>

This defines the "length" of the mesh in the primary matrix direction. It is not used for the primary domain (i.e., the fracture domain) or any spatially connected continuum, but a valid float must be specified. For SLAB geometry, the length and cross-sectional areas are specified independently. For NESTED_CUBE and NESTED_SPHERE, the DOMAIN_LENGTH is equivalent to ½ the fracture spacing (i.e., length is the "radius" of the sphere or cube).

NUMBER_ELEMENTS <N integers>

This specifies the number of elements at each spatial location in each continuum (base continuum is not used, but a valid integer must be specified).

GEOMETRIES <N keywords>

This specifies the type of geometry used to compute the element volumes and connectivities in the secondary continua mesh. Valid keywords are:

- FRACTURE
- SLAB
- NESTED_CUBES
- NESTED_SPHERES

FRACTURE is a placeholder for the primary domain. SLAB is a cartesian geometry, where the cross-sectional area to flow is constant in each element in the matrix. NESTED_{CUBE,SPHERE} include geometric assumptions for the cross-sectional area to flow and the element volume variability. The beginning of the continuum is at the outside of the cube or sphere, and flow travels to the center.

DIRECTIONS <N keywords> (optional)

This optional parameter specifies the direction of the mesh ($+Y$ is default). Choices are X, Y, Z, $+X$, $-X$, $+Y$, $-Y$, $+Z$ and $-Z$. This mostly impacts visualization output (it is difficult to visualize multidimensional meshes with multiple secondary continua, but sometimes this can help), but it could have an impact on solutions that include the effects of gravity (e.g., two-phase flow or density-dependent flow), if $+Z$ or $-Z$ is chosen and gravity is active (if gravity is associated with a different direction, this could likewise have an effect). Gravity can be changed in the GRID block with the "GRAVITY 0.0 0.0 -9.8068" option (see online PFLOTRAN documentation).

MESH block <N rows>

This block has one row for each continuum. Each row (until an END row) starts with one of the following keywords:

FRACTURE

This keyword by itself indicates the fracture connectivity will be used.

UNIFORM

This keyword by itself indicates the elements will be uniformly subdivided (dx = DOMAIN_LENGTHS / NUMBER_ELEMENTS).

DX

This keyword is followed by NUMBER_ELEMENTS <floats> that specify the grid spacing along this continuum. Using this approach, uniform spacing could be specified, or spacing could grow geometrically (a common approach). This spacing should be computed externally and the values pasted here.

Since this input file is parsed by a Python script, there is effectively no limit to the length of the line. In PFLOTRAN files there is a maximum line length of 512 characters, and long lines must be broken into shorter lines.

SPATIAL_CONNECTIVITY <N 0/1 integers> (optional)

This optional block can be used to over-ride the connectivity implied by the TYPE block (TYPE block must still be used to specify N). Valid values are 1 (True) and 0 (False), indicating spatial connectivity between matrix locations. Connectivity of the first continuum is always 1.

TYPE	SPATIAL_CONNECTIVITY
DOUBLE_POROSITY	1 0
DOUBLE_PERMEABILITY	1 1
TRIPLE_POROSITY	1 0 0
TRIPLE_PERMEABILITY	1 1 1

SLAB_AREAS <N floats> (optional)

This optional block can be used to specify the cross-sectional area for the SLAB geometry. Otherwise, the cross-sectional area is taken from the cross-sectional area of the parent cell. Values specified for continua that are not SLAB are not used (but a valid float must be specified).

3. MULTICONTINUUM MODEL BENCHMARK

We compare the more general multicontinuum approach to the existing SECONDARY_CONTINUUM approach in PFLOTRAN (Figure 7), for the special case of water flow and solute transport. Water flow only occurs in the 1D fracture, while solute transport happens in both the fracture and in the matrix. The fracture includes solute advection and diffusion, while only solute diffusion occurs in the matrix. The 1D fracture and matrix is initially set to a uniform tracer concentration of 0.01 M, then fresh water (1.0E-8 M tracer) is flushed into the system in the fracture by specifying a liquid pressure gradient across the fracture (2.5 mbar). The domain is 10 m long, discretized into 100 elements.

The concentration is plotted at the two ends and in the middle of the domain for three cases:

- A 1D fracture with no matrix;
- Fracture and matrix, computed via SECONDARY_CONTINUUM ("SC"); and
- Fracture and matrix, computed via the multicontinuum ("MC") method presented here.

Figure 8 shows two plots illustrating these cases, showing the two fracture/matrix approaches produce nearly identical results. The only observable differences are in first element of the matrix at early time (see red ellipse), when gradients are the steepest. The SECONDARY_CONTINUUM approach only converges for this test case (taken from the PFLOTRAN regression test suite) if the solver tolerance is relaxed. For the SECONDARY_CONTINUUM approach the fracture and matrix use different solution methods. For the multicontinuum approach described here, the entire mesh is solved implicitly and simultaneously and default solver settings work well, and it is likely these results are somewhat more accurate at early time near the matrix/fracture boundary where concentration gradients are steepest.

```
PINC all
  TYPE DOUBLE_POROSITY
  VOLUME_FRACTIONS 0.4167D+0
  DOMAIN_LENGTHS 1.0 0.1
  NUMBER_ELEMENTS 1 50
  GEOMETRIES FRACTURE SLAB
  MESH
    FRACTURE
    UNIFORM
  END
  DIRECTIONS F -Y
  SLAB_AREAS 0.0 1.0D-6
END
```

```
SECONDARY_CONTINUUM
  TYPE SLAB
  LENGTH 0.1
  AREA 1.0E-6
  NUM_CELLS 50
  EPSILON 0.4167
  DIFFUSION_COEFFICIENT 1.0E-9
  POROSITY 0.4464
END
```

Figure 7. PINC (left) and SECONDARY_CONTINUUM (right) inputs for test case.

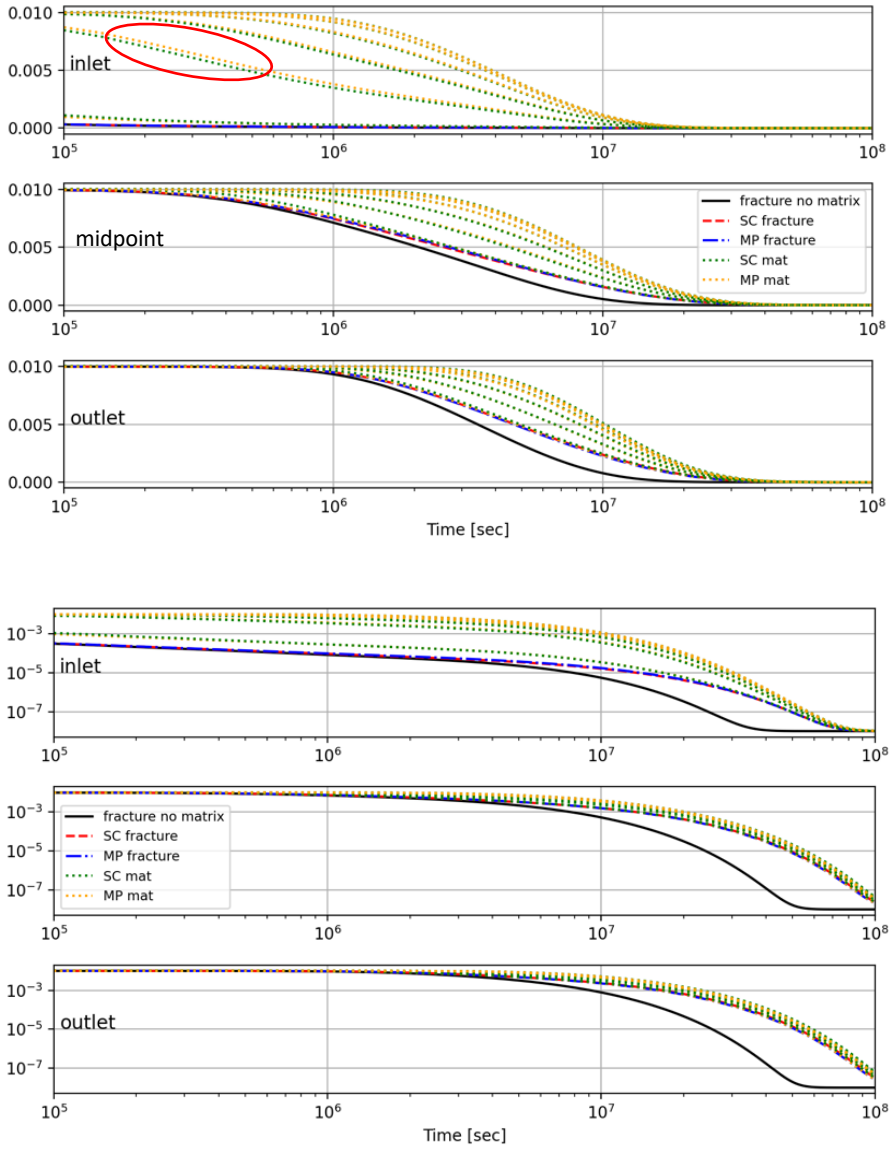

Figure 8. Comparison at 3 observation locations between fracture no matrix, SECONDARY_CONTINUUM fracture/matrix (SC), and multiporosity (i.e., multicontinuum) fracture/matrix (MP). Log-linear (top) and log-log (bottom) scales. Only matrix elements 1, 10, 20, 30, 40 & 50 are plotted.

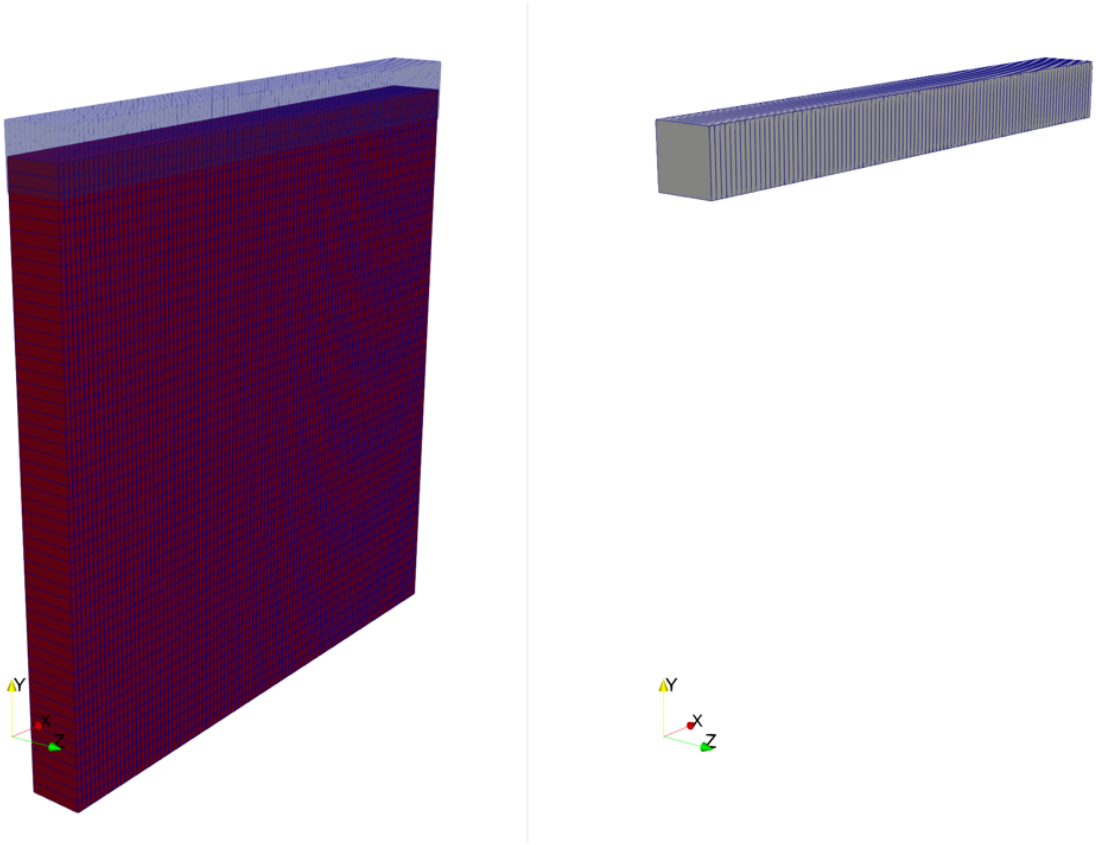

Figure 9. ParaView illustration of model domains in test problem. 1D fracture domain (Right), PINC domain (Left) shows both 1D fracture domain (transparent blue) and array of 1D matrix domains (red).

Figure 9 illustrates the domain used in the test case. The fracture-only domain is shown as the 1D domain on the right, while the combination fracture/matrix domain is shown on the left. Output from the SECONDARY_CONTINUUM approach does not include the matrix in the mesh, so the spatial output is shaped similar to the fracture-only domain in Figure 9. In the PINC domain, the fracture part of the grid is plotted semi-transparent, since the first two matrix elements are located within the fracture volume. Each 1D matrix continuum at each spatial location is connected to the center of the fracture element and not connected to the adjacent 1D matrix continua associated with adjacent elements. The mesh plotted in Figure 9 is topologically like a "comb" and not like a plate.

As specified in the PINC input (Figure 7), the matrix domain extends in the $-Y$ direction away from the fracture. Changing the DIRECTIONS input will change the direction the matrix mesh extends. In a problem with two matrix continua, the two directions could be the same, but for visualization purposes it would be better if they were different. Visualization of matrix continua in 2D or 3D domains is difficult and likely no choice of direction will be optimal.

4. AQUEDUCT MESA CONCEPTUAL MODEL

The porous volcanic tuff at Aqueduct Mesa contains natural fractures and is variably water saturated. The fractures are mostly air-filled, and the matrix is mostly water saturated. Briefly, the layers from the Geologic Framework Model (GFM) of Prothero (2018) are summarized stratigraphically from the top of the mesa down as:

- Upper nonwelded to partially welded tuff (UNPWT)
- Upper welded tuff (UWT)
- Vitric nonwelded tuff (VNT)
- Upper zeolitic nonwelded tuff (UZNT)

The observed matrix saturation data from vertical borehole UE12p#4 (Lupo & Klauber, 1987; Torres, 1988) are illustrated in the left part of Figure 10 (dots), with one of several possible simplifications of the observations with possible unit-averaged values (dashed line). Fracture properties are adopted from Eaton and Bixler (1987).

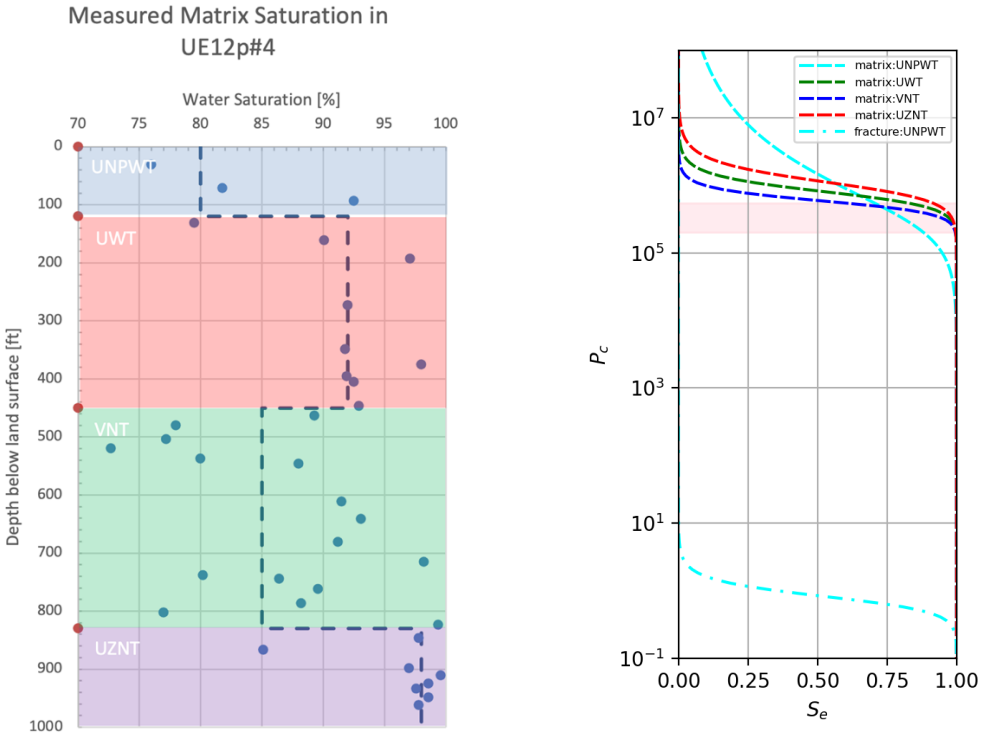

Figure 10. (Left) Saturation data from UE12P#4 (points), showing GFM layers (colored boxes) and one interpretation of uniform saturation (dashed line). (Right) Plot of van Genuchten moisture retention curves used in PFLOTRAN model corresponding to these units (horizontal pink stripe shows observed capillary pressure range inferred from waxed samples).

Based on the interpreted capillary pressure across several very different samples of tuff, analyzed by both MICP and effective diffusion coefficient, a relatively narrow range of capillary pressure of 0.2 to 0.55 MPa was inferred (Table S1 of Heath et al., 2021). The representativeness of these wax-preserved samples from the 1980s is not known, as the samples may have dried out or been otherwise altered during more than 30 years of storage.

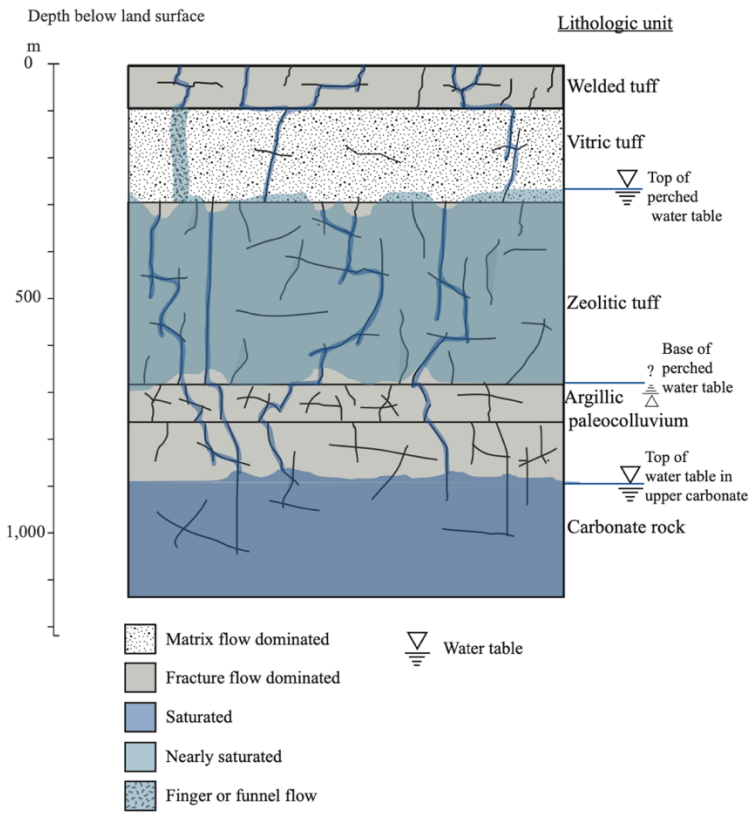

Figure 11. Conceptual flow model for Rainier Mesa (Ebel & Nimmo, 2009).

Figure 11 shows a conceptualization of water saturation among the volcanic tuff units at Rainier Mesa. Although Rainier Mesa receives more rainfall than Aqueduct Mesa and is considered to have more tectonic fractures, in general the distribution of wetter and dryer units prevails across them both. The zeolitic units have higher water content (compare Figure 11 and the left half of Figure 10).

4.1. Model Initialization

The simplest approach to gas transport would be to assume the observed pre-testing moisture contents in fractures and matrix at Aqueduct Mesa (Figure 10) is immobile and the system is at steady state. Flow through the remaining gas-filled pores of the vadose zone is then essentially single-phase gas transport. Fractures are mostly dry, the matrix is mostly wet, and the different lithologic layers have different matrix saturations (but there is significant variability within layers).

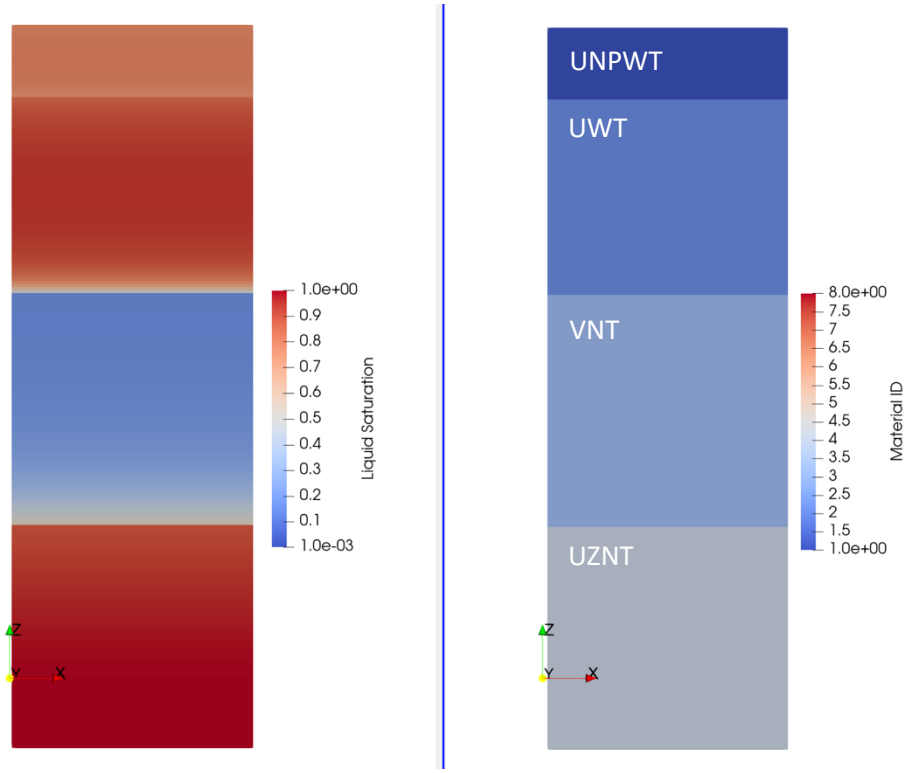

Figure 12. PFLOTRAN-predicted initial saturation (left) across matrix domains (layers shown on right). Fracture continuum is dry and not plotted. Vertical coss-section is 366 m tall, with the land surface at the top.

Initializing a two-phase flow model requires specifying several related parameters (liquid and gas saturation, liquid or capillary pressure, and temperature) to values that are in physical equilibrium with one another. The simplest modeling approach (immobile water) is to assign an observed matrix liquid saturation and set the residual water content at the same level as the observed water content. Essentially, this forces the system to be at steady state, preventing water from redistributing.

As part of this investigation, we wish to better understand the hydrologic system, by striving to re-create the observed water content by changing the steady-state processes influencing the distribution of water in the mesa. Determining what impacts the distribution of water in the vadose zone provides information on what may perturb it. Models that assume all water is immobile from first principles cannot be used to assess whether the water may move or not (Heath et al., 2021).

Figure 12 illustrates the results of simulating a one-dimensional column of a multicontinuum with physically realistic saturations and approximately uniform initial capillary pressure (i.e., see capillary pressure curves in the right half of Figure 10). Running the system to steady-state results in variation of the saturation, especially at changes in material properties. Some of this variability may also be observed in the profile of liquid saturation seen in borehole UE12p#4 (left half of Figure 10). Water content in the lower-permeability matrix interacts with flow through the gas-permeable fractures.

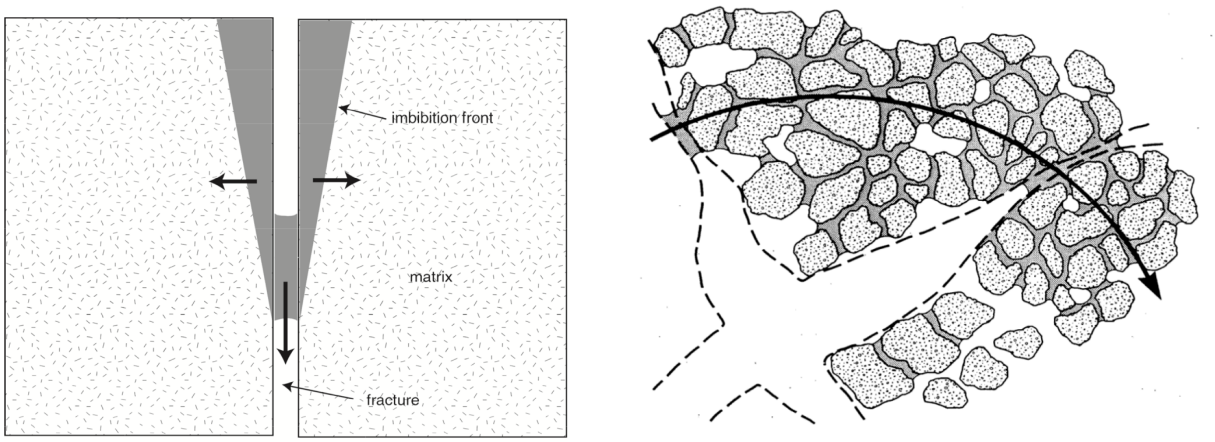

Figure 13. Illustrations of water and gas transport in fractured rock (NRC, 2001).

Especially since we are interested in making predictions about the system after a large man-made perturbation has occurred it makes sense to test our assumptions regarding the unconditional stability of liquid water in the subsurface. Increases in temperature will reduce surface tension and could liberate previously held water from the matrix. Sudden increases in gas pressure or stress, due to an explosion, could liberate water from the matrix. Water released from the matrix could then flow into fractures and even small amounts of water can block rapid gas transport through fractures (Figure 13).

5. SUMMARY AND NEXT STEPS

This work has primarily been focused on the development of a more general meshing tool for creating multicontinuum meshes allowing simulation of two-phase flow in fractured rocks, and we discuss applicability to the fractured tuff at Aqueduct Mesa.

As part of this work in the next fiscal year, we will continue to gather field and laboratory data relevant to understanding two-phase flow, developing modeling tools to predict two-phase flow, and strive to build physical intuition and understanding that can be used at new future sites where we don't have as much data.

We will continue to benchmark and test models using the multicontinuum modeling framework to ensure it is working and validated against a number of published results. We will develop numerical models of physically realistic flow through simplified geometries, initial conditions, and boundary conditions, building complexity to understand the processes that impact two-phase flow at the Aqueduct Mesa, Rainier Mesa, and eventually for application to other sites.

REFERENCES

Abdelkader, A., C.L. Bajaj, M.S. Ebeida, A.H. Mahmoud, S.A. Mitchell, J.D. Owens & A.A. Rushdi. VoroCrust: Voronoi meshing without clipping. *ACM Transactions on Graphics 39*(3): 1-16.

Barenblatt, G.I., I.P. Zheltov, & I.N. Kochina, 1960. Basic concepts in the theory of seepage of homogeneous liquids in fissured rocks [strata]. *Journal of Applied Mathematics and Mechanics*, *24*(5), 1286-1303.

Clossman, P.J., 1975. An aquifer model for fissured reservoirs. *Society of Petroleum Engineers Journal*, *15*(05), 385-398.

Coats, K.H. & B.D. Smith, 1964. Dead-end pore volume and dispersion in porous media. *Society of Petroleum Engineers Journal*, *4*(1), 73-84.

Eaton, RR, Bixler, NE. 1986. *Analysis of a Multiphase, Porous-Flow Imbibition Experiment in Fractured Volcanic Tuff*. SAND86-1679C, Sandia National Laboratories, Albuquerque, NM.

Evans, J.P. & K.K. Bradbury, 2004. Faulting and fracturing of nonwelded Bishop Tuff, eastern California: deformation mechanisms in very porous materials in the vadose zone. *Vadose Zone Journal, 3*:602-623.

Fiori, A., A. Zarlenga, H. Gotovac, I. Jancovic, E. Volpi, V. Cvetkovic & G. Dagan, 2015. Advective transport in heterogeneous aquifers: are proxy models predictive? *Water Resources Research, 51*:9577-9594.

Gringarten, A.C., 1982. *Flow-Test Evaluation of Fractured Reservoirs*. Geological Society of America Special Paper 189.

Haggerty, R., & S.M. Gorelick, 1995. Multiple-rate mass transfer for modeling diffusion and surface reactions in media with pore-scale heterogeneity. *Water Resources Research*, *31*(10), 2383-2400.

Hammond, G.E., P.C. Lichtner & R.T. Mills, 2014. Evaluating the performance of parallel subsurface simulators: An illustrative example with PFLOTRAN. *Water Resources Research*, *50*(1), 208-228.

Havlin, S. & D. Ben-Avraham, 1987. Diffusion in disordered media, *Advances in Physics, 36*(6):695-798.

Heath, J.E., K.L. Kuhlman, S.T. Broome, J.E. Wilson & B. Malama, 2021. Heterogeneous multiphase flow properties of volcanic rocks and implications for noble gas transport from underground nuclear explosions, *Vadose Zone Journal* 2021;e20123. https://doi.org/10.1002/vzj2.20123

Hyman, J.D., S. Karra, N. Makedonska, C.W. Gable, S.L. Painter & H.S. Viswanathan. dfnWorks: A discrete fracture network framework for modeling subsurface flow and transport. *Computers & Geosciences, 84*(2015):10-19.

Jordan, A.B., P.H. Stauffer, E.E. Knight, E. Rougier & D.N. Anderson, 2015. Radionuclide gas transport through nuclear explosion-generated fracture networks, *Scientific Reports, 5*:18383.

Kazemi, H., 1969. Pressure transient analysis of naturally fractured reservoirs with uniform fracture distribution. *Society of Petroleum Engineers Journal*, *9*(04), 451-462.

Kuhlman, K.L., B. Malama & J.E. Heath, 2015. Multiporosity flow in fractured low-permeability rocks. *Water Resources Research*, *51*(2), 848-860.

Lupo, J, Klauber, W. 1987. *Physical and Mechanical Characterization of Tuff from UE12P#4*. TR 87-94, Terra Tek Research, Salt Lake City, Utah, 134 p.

Moench, A.F., 1984. Double-porosity models for a fissured groundwater reservoir with fracture skin. *Water Resources Research*, *20*(7), 831-846.

Montroll, E.W. & G.H. Weiss, 1965. Random walks on lattices II. *Journal of Mathematical Physics, 6*(2):167-181.

Neuman, S.P., W.A. Illman, V.V. Vessilinov, D.L. Thompson, G. Chen & A. Guzman, 2001. "Lessons from the field studies at the Apache Leap Research Site in Arizona", p 295-334, in National Research Council, *Conceptual Models of Flow and Transport in the Fractured Vadose Zone*, Washington DC: The National Academies Press.

NRC (National Research Council), 2001. *Conceptual Models of Flow and Transport in the Fractured Vadose Zone*. Washington DC: National Academies Press.

Noolandi, J., 1977. Equivalence of multiple-trapping model and time-dependent random walk. *Physical Review B, 16*(10):4474-4479.

Oldham, K.B. & J. Spanier, 1970. The replacement of Fick's law by a formulation involving semidifferentiation. *Electroanalytical Chemistry and Interfacial Electrochemistry, 26*:331-341.

Prothro, L. 2018. *Geologic Framework Model for the Underground Nuclear Explosions Signatures Experiment P-Tunnel Testbed, Aqueduct Mesa, Nevada National Security Site*. DOE/NV/03624-0312, 60 p.

Pruess, K., 1992. *Brief Guide to the MINC-Method for Modeling Flow and Transport in Fractured Media.* LBL-32195, Berkeley, CA: Lawrence Berkeley National Laboratory.

Pruess, K. & T.N. Narasimhan, 1982. A practical method for modeling fluid and heat flow in fractured porous media, *Society of Petroleum Engineers Journal, 25*(1):14-26.

Reeves, D.M., R. Parashar, K. Pohlmann, C. Russell, & J. Chapman. Development of calibration of dual-permeability flow models with discontinuous fault networks. *Vadose Zone Journal*, 13(8).

Scher, H., M.F. Shlesinger & J.T. Bendler, 1991. Time-scale invariance in transport and relaxation, *Physics Today, 44*:26-34.

Schumer, R., D.A. Benson, M.M. Meerschaert & B. Baeumer, 2003. Fractal mobile/immobile solute transport. *Water Resources Research, 39*(10):1296.

Torres, G. 1988. *Characterization of Tuff from Vertical Drill Hole UE12p.#4 with Emphasis on Material from 813.6-1777.8 ft*. TR 89-36, Terra Tek Research, Salt Lake City, Utah, 105 p.

Van Genuchten, M.Th. & P.J. Wierenga, 1976. Mass transfer studies in sorbing porous media I. Analytical solutions. *Soil Science Society of America Journal, 40*(4):473-480.

Warren, J.E., & P.J. Root, 1963. The behavior of naturally fractured reservoirs. *Society of Petroleum Engineers Journal*, *3*(03), 245-255.

White, M.D., P. Fu & EGS Collab Team, 2020. Application of an Embedded Fracture and Borehole Modeling Approach to the Understanding of EGS Collab Experiment 1, *Proceedings of the 45th Workshop on Geothermal Reservoir Engineering, February 10-12, 2020*, SGP-TR-216.

Zimmerman, R.W., G. Chen, T. Hadgu & G.S. Bodvarsson, 1993. A numerical dual-porosity model with semianalytical treatment of fracture/matrix flow. *Water Resources Research, 29*(7):2127-2137.

APPENDIX A. SOURCE LISTING

```
import numpy as np
from itertools import product
import sys
from h5py import File

VERBOSE = 4  # higher number -> more screen output. 0=quiet
MULTIPOROSITY = True
READ_IN_ELEVATIONS = True

# only used for variable Z elevations
VARIABLE_Z_CONNECTION_AREA = "average"  # "average" or "minimum"
VARIABLE_Z_VERTICES = "piecewise"  # "piecewise" or "continuous"

eps = 1.0e-8  # things within this distance (m) are at same point
most = 0.95

df = 8  # decimal places to print in distances
af = 16  # decimal places to print in areas
vf = af  # decimal places to print in volumes

dv = ("x", "y", "z")

if len(sys.argv) < 2:
    print(
        f"""Specify PFLOTRAN input on the command line ("prefix.in").
This script reads the GRID & REGION cards from input and writes:
  1) an equivalent explicit unstructured mesh ("prefix.uge"),
  2) equivalent regions for unstructured mesh (as "prefix-region-POINT-name.txt",
      "prefix-region-SURFACE-name.ex", "prefix-region-VOLUME-name.txt" also the
      equivalent data in hdf5 format "prefix-regions.h5"), and
  3) a new PFLOTRAN input file referencing explicit GRID/REGION blocks ("prefix-explicit.in").

GRID card is required, REGION cards are optional
NB: "skip" ... "noskip" comments and EXTERNAL_FILE (in REGION or GRID blocks) are not handled yet.
Run like this: "python {sys.argv[0]} input.in"

 file types
 ------------------------------------------
 *.in  : ASCII free-form PFLOTRAN input file
 *.uge : ASCII explicit unstructured mesh (CELL, CONNECTIONS, ELEMENTS & VERTICES blocks)
 *.h5  : binary hdf5 file (domain file for paraview, BC/region info in h5 format)
 *.ex  : ASCII file listing CONNECTIONs for BOUNDARY_CONDITION associated with faces
 *.txt : ASCII file listing node ids for MATERIAL_PROPERTYs, INITIAL_CONDITIONs, and SOURCE_SINKs
"""
    )
    sys.exit(1)
else:
    fn = sys.argv[1]


def ffloat(s):
    # gracefully read a fortran floating point, possibly with "D" exponential
    # or a _ placeholder

    try:
        v = float(s)
    except ValueError:
        if s == "_":
            s = "NAN"
        v = float(s.upper().replace("D", "E"))
    return v


def find_point_idx_up(point, cumdx, name):
    # point = tuple of x, y, z coordinates for point
    # cdx = tuple of numpy arrays of cumulative dx, dy, dz (with origin added as 0th term)
    # name = string name of region
    # find what element the point is in (or on the boundary of)
    # convention is to assign a point to the upper-index cell in each direction

    idx = [-999, -999, -999]
    for i, (coord, d, cd) in enumerate(zip(point, dv, cumdx)):
        N = cd.shape[0] - 2
        if coord < cd[0]:
            idx[i] = 0
            if VERBOSE > 1:
                print(
```

```
                    f"IDX-UP: {d}-coord of region '{name}' below origin "
                    f"({coord} < {cd[0]}), outside mesh -> idx=0"
                )
        elif coord > cd[-1]:
            idx[i] = N
            if VERBOSE > 1:
                print(
                    f"IDX-UP: {d}-coord of region '{name}' beyond mesh extent "
                    f"({coord} > {cd[-1]}), outside mesh -> idx={N}"
                )
        elif abs(coord - cd[-1]) < eps:
            idx[i] = N - 1
            if VERBOSE > 1:
                print(
                    f"IDX-UP: {d}-coord of region '{name}' on upper edge of domain "
                    f"({coord} == {cd[-1]}), in idx={N-1}"
                )
        else:
            for j, cv in enumerate(cd[1:]):
                if coord < cv:
                    idx[i] = j
                    if VERBOSE > 1:
                        print(
                            f"IDX-UP: {d}-coord of region '{name}' in cell {j} "
                            f"({cd[j]} <= {coord} <= {cd[j+1]})"
                        )
                    break
    return tuple(idx)


def find_point_idx_down(point, cumdx, name):
    idx = [-999, -999, -999]
    for i, (coord, d, cd) in enumerate(zip(point, dv, cumdx)):
        N = cd.shape[0] - 2
        if coord < cd[0] - eps:
            idx[i] = 0
            if VERBOSE > 1:
                print(
                    f"IDX-DOWN: {d}-coord of region '{name}' below origin "
                    f"({coord} < {cd[0]}), outside mesh -> idx=0"
                )
        elif coord > cd[-1] + eps:
            idx[i] = N
            if VERBOSE > 1:
                print(
                    f"IDX-DOWN: {d}-coord of region '{name}' beyond mesh extent "
                    f"({coord} > {cd[-1]}), outside mesh -> idx={N}"
                )
        else:
            for j, cv in enumerate(cd[1:]):
                if coord <= cv + eps:
                    idx[i] = j
                    if VERBOSE > 1:
                        print(
                            f"IDX-DOWN: {d}-coord of region '{name}' in cell {j} "
                            f"({cd[j]} <= {coord} <= {cd[j+1]})"
                        )
                    break
    return tuple(idx)


def face_str_to_idx(s, name):
    if s == "WEST":
        idx = 1
    elif s == "EAST":
        idx = 2
    elif s == "SOUTH":
        idx = 3
    elif s == "NORTH":
        idx = 4
    elif s == "BOTTOM":
        idx = 5
    elif s == "TOP":
        idx = 6
    else:
        print(f"ERROR: invalid region '{name}' FACE: '{s}'")
        sys.exit("input inconsistency")
```

```
155     return idx
156
157
158 # open filename specified on command-line
159 fh = open(fn, "r")
160 base_fn = fn.replace(".in", "")
161
162 if VERBOSE > 0:
163     print(f"\n--------- begin processing GRID/REGION blocks from {fn} ----------")
164
165
166 def clean_lines(lines):
167     # read lines, eliminate comments
168
169     for j in range(len(lines)):
170         # strip off leading/trailing whitespace
171         lines[j] = lines[j].strip()
172
173         # remove single-line comments
174         comment_char = -1
175         for i in range(len(lines[j])):
176             ch = lines[j][i]
177             if ch in ["!", "#", ":"]:
178                 comment_char = i
179                 break
180
181         if comment_char >= 0:
182             lines[j] = lines[j][:comment_char]
183     return lines
184
185
186 def parse_lines(lines):
187     # parse input into lists of tokens
188
189     rows = []
190     for line in lines:
191         line = line.strip()
192         fields = line.split()
193         if VERBOSE > 5:
194             print(fields)
195         row = []
196         for f in fields:
197             if f == "/":
198                 row.append("END")
199             else:
200                 row.append(f)
201         if len(row) > 0:
202             row[0].upper()  # don't change case of name
203         # TODO handle skip ... noskip here?
204         # need to handle the pair of skip/noskip
205         # being on same or different lines
206
207         # TODO handle "EXTERNAL_FILE" construct here?
208         # this just means parsing that file and inserting
209         # it into the list in place of this row
210         if len(row) > 0:
211             # don't add empty rows
212             rows.append(row)
213     return rows
214
215
216 # %%%%%%%%%%%%%%%%%%%
217
218 if VERBOSE > 1:
219     print("\ninitial input")
220
221 rows = parse_lines(clean_lines(fh.readlines()))
222 fh.close()
223
224 # %%%%%%%%%%%%%%%%%%%
225 # split into "GRID" and "REGION" chunks
226 level_0_blocks = []
227 GRID_IDX = -1
228 REGION_IDX = -1
229
230 # these are other places where "REGION name" will be called
231 OTHER_BLOCKS = [
```

```
    "BOUNDARY_CONDITION",
    "INITIAL_CONDITION",
    "STRATA",
    "OBSERVATION",
]
OBI = -1 * np.ones((len(OTHER_BLOCKS),), dtype=np.int32)
num_regions = 0

for j, row in enumerate(rows):
    if row[0] in OTHER_BLOCKS:
        for i, OB in enumerate(OTHER_BLOCKS):
            if row[0] == OB and OBI[i] == -1:
                # REGION may appear in this block
                OBI[i] = 0

    elif row[0] == "END" and np.any(OBI < 0):
        # assume there are no nested BC/STRATA/OBS blocks
        for i, OB in enumerate(OTHER_BLOCKS):
            if OBI[i] == 0:
                # this is the end block
                OBI[i] = -1

    if row[0] == "GRID" and GRID_IDX == -1:
        level_0_blocks.append(["GRID", j])
        GRID_IDX = 0
    elif row[0] == "END" and GRID_IDX == 0:
        GRID_IDX = 1
        # this is the closing of the required level-1 BOUNDS or DXYZ block
    elif row[0] == "END" and GRID_IDX == 1:
        level_0_blocks[-1].append(j + 1)
        GRID_IDX = -1
    elif row[0] == "REGION" and REGION_IDX == -1 and np.all(OBI < 0):
        # check this region token isn't inside a coupler block
        level_0_blocks.append(["REGION", row[1], j])
        REGION_IDX = 0
    elif row[0] == "COORDINATES" and REGION_IDX == 0:
        REGION_IDX = 1
    elif row[0] == "END" and REGION_IDX == 1:
        # end of possible COORDINATES block
        REGION_IDX = 0
    elif row[0] == "END" and REGION_IDX == 0:
        level_0_blocks[-1].append(j + 1)
        REGION_IDX = -1
        num_regions += 1

# rows that should be echoed in final output file
context = np.ones((len(rows),), dtype=np.int32)

# only allowed a single GRID block
for j, block in enumerate(level_0_blocks):
    if block[0] == "GRID":
        grid_row_idx = block[1:3]
        context[block[1] : block[2]] = -1  # mark where GRID block was
    else:
        context[block[2] : block[3]] = 0

if VERBOSE > 0:
    print("\nLevel-0 blocks in input file")
    print(f"GRID block found at rows {grid_row_idx[0]}-{grid_row_idx[1]}")
    print(f"found {num_regions} REGION blocks")
    for block in level_0_blocks:
        if block[0] == "REGION":
            print(block)

# %%%%%%%%%%%%%%%%%%%
# handle line continuation characters
# make a copy so possibly deleting a row doesn't mess other things up
grid_rows = rows[grid_row_idx[0] : grid_row_idx[1]]

nl = len(grid_rows)
for j in range(nl):
    # check starting second from end, moving forward
    idx = nl - 1 - j
    if grid_rows[idx][-1] == "\\":
        del grid_rows[idx][-1]  # delete continuation char
        grid_rows[idx].extend(grid_rows[idx + 1])  # merge lines in place
        del grid_rows[idx + 1]  # delete repeated line
```

```
309
310 # %%%%%%%%%%%%%%%%%%
311 # handle multiple values using "@" nomenclature in DXYZ block
312 for j in range(len(grid_rows)):
313     new_row = []
314     for i in range(len(grid_rows[j])):
315         if "@" in grid_rows[j][i]:
316             mult, dx = grid_rows[j][i].split("@")
317             new_row.extend([dx] * int(mult))
318         else:
319             new_row.append(grid_rows[j][i])
320
321     grid_rows[j] = new_row  # replace old row with new
322
323 if VERBOSE > 1:
324     # dump for debugging
325     print("\nprocessed grid input")
326     for row in grid_rows:
327         print(row)
328
329 # ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
330 #                   primary grid options
331 # ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
332
333 p = {}  # dict to store options read
334
335 # TYPE: STRUCTURED {[CARTESIAN],CYLINDRICAL,SPHERICAL},
336 #       UNSTRUCTURED <filename>,
337 #       UNSTRUCTURED_EXPLICIT <filename>
338 # only handle structured for now
339
340 # NXYZ <int int int>
341
342 # ORIGIN <float float float>
343 # coordinates of grid origin, taken as lower corner of "bounds" if specified
344
345 # optional GRID sub-cards to echo unmodified
346 # does not include "2ND_ORDER_BOUNDARY_CONDITION", "DOMAIN_FILENAME", "INVERT_Z"
347
348 # INVERT_Z doesn't actually do anything (despite it being in the official documentation,
349 # and not throwing an error when used, it just reads the keyword and silently does nothing)
350 echo = [
351     "GRAVITY",
352     "PERM_TENSOR_TO_SCALAR_MODEL",
353     "MAX_CELLS_SHARING_A_VERTEX",
354     "STENCIL_WIDTH",
355     "STENCIL_TYPE",
356     "UPWIND_FRACTION_METHOD",
357 ]
358
359 # arguments with following values on one line
360 for row in grid_rows:
361     if "TYPE" in row:
362         p["TYPE"] = [f.upper() for f in row[1:]]
363         if p["TYPE"][0] in ["UNSTRUCTURED", "UNSTRUCTURED_EXPLICIT"]:
364             # eventually start from implicit unstructured mesh. not yet.
365             print("python conversion script only works with STRUCTURED mesh for now")
366             sys.exit("script fail")
367     elif "NXYZ" in row:
368         p["NXYZ"] = tuple(int(x) for x in row[1:])
369     elif "ORIGIN" in row:
370         p["ORIGIN"] = [ffloat(x) for x in row[1:]]
371
372 if len(p["TYPE"]) == 1:
373     p["TYPE"].append("CARTESIAN")  # default structured
374 else:
375     if not p["TYPE"][1] in ["CYLINDRICAL", "SPHERICAL", "CARTESIAN"]:
376         print(f"ERROR: unknown structured mesh type: {p['TYPE'][1]}")
377         sys.exit("input inconsistency")
378
379 p["TYPE"] = tuple(p["TYPE"])
380
381 # arguments specified in a multi-line block with an END
382 for j in range(len(grid_rows)):
383     if "BOUNDS" in grid_rows[j]:
384         p["BOUNDS"] = [[ffloat(x) for x in r] for r in grid_rows[j + 1 : j + 3]]
385     elif "DXYZ" in grid_rows[j]:
```

```
        p["DXYZ"] = [[ffloat(x) for x in r] for r in grid_rows[j + 1 : j + 4]]

# things to be echoed in modified output
p["ECHO"] = []
for row in grid_rows:
    if row[0] in echo:
        p["ECHO"].append(row)

# %%%%%%%%%%%%%%%%%%%
# allow a single number to be "spread" across a dimension
if "DXYZ" in p:
    for j in range(3):
        lx = len(p["DXYZ"][j])
        nx = p["NXYZ"][j]
        if lx == nx:
            if VERBOSE > 0:
                print(f"dimension {j+1} is correct (NXYZ vs. DXYZ)")
        elif lx < nx and lx == 1:
            p["DXYZ"][j] = [p["DXYZ"][j][0]] * nx
            if VERBOSE > 0:
                print(f"extending single DXYZ value {nx} times")
        else:
            print(f"ERROR in dimension {j+1}: NXYZ={nx}, len(DXYZ)={lx}")
            sys.exit("input inconsistency")

# %%%%%%%%%%%%%%%%%%%
if "BOUNDS" in p and "DXYZ" in p:
    print("ERROR: cannot specify both BOUNDS and DXYZ")
    sys.exit("input inconsistency")

# convert BOUNDS to DXYZ and ORIGIN to reduce later options
if "BOUNDS" in p:
    Lx = p["BOUNDS"][1][0] - p["BOUNDS"][0][0]
    Ly = p["BOUNDS"][1][1] - p["BOUNDS"][0][1]
    Lz = p["BOUNDS"][1][2] - p["BOUNDS"][0][2]

    if "ORIGIN" in p:
        x0, y0, z0 = p["ORIGIN"]
        xb0, yb0, zb0 = p["BOUNDS"][0]
        if ((x0 - xb0) ** 2 + (y0 - yb0) ** 2 + (z0 - zb0) ** 2) > eps:
            print("ERROR: cannot specify differing BOUNDS and ORIGIN")
            sys.exit("input inconsistency")
    p["ORIGIN"] = tuple(p["BOUNDS"][0])

    dx = Lx / float(p["NXYZ"][0])
    dy = Ly / float(p["NXYZ"][1])
    dz = Lz / float(p["NXYZ"][2])

    p["DXYZ"] = [[dx] * p["NXYZ"][0], [dy] * p["NXYZ"][1], [dz] * p["NXYZ"][2]]

# %%%%%%%%%%%%%%%%%%%
# check cylindrical and spherical grid conventions/limitations
if "CYLINDRICAL" in p["TYPE"]:
    if not (p["NXYZ"][1] == 1):
        print(f"ERROR: in cylindrical mesh, NXYZ={p['NXYZ']}, NY must be 1")
        sys.exit("input inconsistency")
    else:
        if VERBOSE > 1:
            print("cylindrical mesh has correct Y dimension")

    if abs(p["DXYZ"][1][0] - 1.0) > eps:
        print(
            f"\nWARNING: value of single cylindrical "
            f"mesh DY ({p['DXYZ'][1][0]}) reset to unity\n"
        )
        p["DXYZ"][1][0] = 1.0

if "SPHERICAL" in p["TYPE"]:
    if (not (p["NXYZ"][1] == 1)) or (not (p["NXYZ"][2] == 1)):
        print(f"ERROR in spherical mesh: NXYZ={p['NXYZ']}, while NY & NZ must be 1")
        sys.exit("input inconsistency")
    else:
        if VERBOSE > 1:
            print("spherical mesh has correct Y & Z dimensions")

    if abs(p["DXYZ"][1][0] - 1.0) > eps or abs(p["DXYZ"][2][0] - 1.0) > eps:
        print(
```

```
            f"\nWARNING: value of single spherical mesh DY "
            f"({p['DXYZ'][1][0]}) & DZ ({p['DXYZ'][2][0]}) reset to unity\n"
        )
        p["DXYZ"][1][0] = 1.0
        p["DXYZ"][2][0] = 1.0

p["DXYZ"] = tuple(p["DXYZ"])

# %%%%%%%%%%%%%%%%%%%
# add ORIGIN and BOUNDS for domains that didn't specify it
if not "ORIGIN" in p:
    p["ORIGIN"] = (0.0, 0.0, 0.0)

if not "BOUNDS" in p:
    p["BOUNDS"] = [p["ORIGIN"]]

    p["BOUNDS"].append(tuple(v0 + sum(d) for v0, d in zip(p["ORIGIN"], p["DXYZ"])))

if VERBOSE > 1:
    # dump for debugging
    print("\nParsed input")
    for k in p.keys():
        print(k, p[k])

# ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
#                  primary region options
# ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))

# regions can be points, areas, or volumes
# each handled differently

# region must use one of the following 4 possibilities

# option 1
# COORDINATE <float float float>
# single point in space for OBSERVATION locations

# option 2
# COORDINATES
#   x_min y_min z_min
#   x_max y_max z_max
# END
# volume for assigning MATERIALs or SOURCE_SINKs

# option 3
# BLOCK istart iend jstart jend kstart kend
# i,j,k bounded volume for assigning MATERIALs or SOURCE_SINKs

# option 4
# CARTESIAN_BOUNDARY [WEST, EAST, SOUTH, NORTH, BOTTOM, TOP]
# area for applying BOUNDARY_CONDITIONs

# if region with COORDINATES or BLOCK has
# FACE [WEST, EAST, SOUTH, NORTH, BOTTOM, TOP]
# one side of volume is area for BOUNDARY_CONDITION

r = {}

for block in level_0_blocks:
    if block[0] == "REGION":

        bn = block[1]
        reg_row_idx = block[2:4]
        reg_rows = rows[reg_row_idx[0] : reg_row_idx[1]]
        nl = len(reg_rows)

        # each region a dictionary, keyed on its case-sensitive name
        r[bn] = {}

        # arguments with following values on one line
        for row in reg_rows:
            if "COORDINATE" in row:
                r[bn]["COORDINATE"] = tuple(ffloat(x) for x in row[1:])
            elif "BLOCK" in row:
                r[bn]["BLOCK"] = tuple(
                    int(x) - 1 for x in row[1:]
                )  # convert to 0-based
```

```
            elif "FACE" in row:
                r[bn]["FACE"] = row[1].upper()
            elif "CARTESIAN_BOUNDARY" in row:
                r[bn]["CARTESIAN_BOUNDARY"] = row[1].upper()
            elif "FILE" in row:
                # Specifies a file (e.g. HDF5) from which cell ids and
                # face directions (for structured: 1=west, 2=east,
                # 3=south, 4=north, 5=bottom, 6=top) can be read.
                r[bn]["FILE_LIST"] = row[1:]
                print(
                    f"python conversion script can't handle regfion FILE inputs '{bn}'"
                )
                sys.exit("script fail")

        # arguments in a multi-line block with an END
        for j, row in enumerate(reg_rows):
            if "COORDINATES" in row:
                r[bn]["COORDS"] = tuple(
                    tuple(ffloat(x) for x in r) for r in reg_rows[j + 1 : j + 3]
                )

for bn in r.keys():
    if "COORDINATE" in r[bn]:
        r[bn]["DIM"] = "POINT"
    elif "CARTESIAN_BOUNDARY" in r[bn] or "FACE" in r[bn]:
        r[bn]["DIM"] = "SURFACE"
    elif "BLOCK" in r[bn] or "COORDS" in r[bn]:
        r[bn]["DIM"] = "VOLUME"
    else:
        print(f"REGION of unknown type? {bn}")
        print(r[bn])
        sys.exit("script fail")

gb0 = p["BOUNDS"][0]  # grid bounds
gb1 = p["BOUNDS"][1]

for bn in r.keys():
    if "COORDS" in r[bn]:
        rb0 = r[bn]["COORDS"][0]  # region bounds
        rb1 = r[bn]["COORDS"][1]
        rdx = rb1[0] - rb0[0]
        rdy = rb1[1] - rb0[1]
        rdz = rb1[2] - rb0[2]
        for j, rr in enumerate([rdx, rdy, rdz]):
            if rr < 0.0:
                print(f"ERROR: negative-width region '{bn}' in {dv[j]} direction")
                sys.exit("input inconsistency")
        for j in range(3):
            if rb1[j] < gb0[j] or rb0[j] > gb1[j]:
                print(
                    f"ERROR: region '{bn}' doesn't overlap "
                    f"grid in {dv[j]} direction"
                )
                sys.exit("input inconsistency")

for bn in r.keys():
    if p["TYPE"][1] in ["CYLINDRICAL", "SPHERICAL"]:
        f = ""
        if "CARTESIAN_BOUNDARY" in r[bn]:
            f = r[bn]["CARTESIAN_BOUNDARY"][0]
        elif "FACE" in r[bn]:
            f = r[bn]["FACE"]
        if f == "NORTH" or f == "SOUTH":
            print(
                f"ERROR in {p['TYPE']} grid, illegal "
                f"NORTH or SOUTH (Y) faces in region '{bn}'"
            )
            sys.exit("input inconsistency")
        if "SPHERICAL" in p["TYPE"]:
            if f == "TOP" or f == "BOTTOM":
                print(
                    f"ERROR in SPHERICAL grid, illegal "
                    f"TOP or BOTTOM (Z) faces in region '{bn}'"
                )
                sys.exit("input inconsistency")

if VERBOSE > 1:
```

```
    print("\nParsed region blocks")
    for j, bn in enumerate(r.keys()):
        print(f"{j} {bn}", r[bn])

# ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
#       compute explicit mesh stuff
# ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
m = {}
d = p["DXYZ"]
d2 = [[val / 2.0 for val in vec] for vec in p["DXYZ"]]  # dx/2.0

# compute x,y,z coordinates of cell centers
# ******************************

m["vXYZc"] = [[], [], []]
for i in range(3):
    m["vXYZc"][i].append(p["ORIGIN"][i] + d2[i][0])
    for j in range(1, p["NXYZ"][i]):
        m["vXYZc"][i].append(m["vXYZc"][i][j - 1] + d2[i][j] + d2[i][j - 1])
    m["vXYZc"][i] = tuple(m["vXYZc"][i])

if VERBOSE > 2:
    print("\nCell centers, each dimension")
    for i in range(3):
        print(i, m["vXYZc"][i])

# Cartesian product of dxdydz vectors (sets ordering of cells)
m["XYZc"] = []
m["IDX"] = []

m["dXYZc"] = {}  # save in dict for reverse lookup

ii = 0
for i, x in enumerate(m["vXYZc"][0]):
    for j, y in enumerate(m["vXYZc"][1]):
        for k, z in enumerate(m["vXYZc"][2]):
            # z changes fastest, x changes slowest
            # same as product(dx,dy,dz)
            m["XYZc"].append((x, y, z, d[0][i], d[1][j], d[2][k]))
            m["IDX"].append((ii, i, j, k))
            # (x, y, z) tuple as key
            m["dXYZc"][(i, j, k)] = (ii, x, y, z, d[0][i], d[1][j], d[2][k])
            ii += 1

if VERBOSE > 2:
    print("\nCell coordinates / extents (x,y,z, dx,dy,dz)")
    print(m["XYZc"])

    print("\nCell indices (idx, i,j,k)")
    print(m["IDX"])

# compute cell volumes
# ****************************
m["VOL"] = []

if "CARTESIAN" in p["TYPE"]:
    for dx, dy, dz in product(d[0], d[1], d[2]):
        # box
        m["VOL"].append(dx * dy * dz)

elif "CYLINDRICAL" in p["TYPE"]:
    for i, (dx, dy, dz) in enumerate(product(d[0], d[1], d[2])):
        # annular ring
        # form that doesn't involve subtraction or squaring x coordinate
        m["VOL"].append(np.pi * 2.0 * dx * m["XYZc"][i][0] * dz)

elif "SPHERICAL" in p["TYPE"]:
    for i, (dx, dy, dz) in enumerate(product(d[0], d[1], d[2])):
        # spherical shell
        # form that doesn't involve subtraction or cubing x coordinate
        m["VOL"].append(np.pi * dx / 3.0 * (dx ** 2 + 12.0 * m["XYZc"][i][0] ** 2))

if VERBOSE > 2:
    print("\nCell volumes")
    print(m["VOL"])

# compute connectivity
```

```
# ****************************
m["CON"] = []

for i in range(p["NXYZ"][0]):  # x
    for j in range(p["NXYZ"][1]):  # y
        for k in range(p["NXYZ"][2]):  # z

            # down = this element
            v_dn = m["dXYZc"][(i, j, k)]

            if k < p["NXYZ"][2] - 1:
                # Z direction
                v_up = m["dXYZc"][(i, j, k + 1)]
                up_idx = v_up[0]
                dn_idx = v_dn[0]
                x, y, z = v_dn[1:4]
                dx, dy, dz = v_dn[4:7]
                xf = x
                yf = y
                zf = z + dz / 2.0

                if "CARTESIAN" in p["TYPE"]:
                    A = dx * dy
                elif "CYLINDRICAL" in p["TYPE"]:
                    A = np.pi * 2.0 * dx * x  # annular area

                # no spherical
                m["CON"].append((up_idx, dn_idx, xf, yf, zf, A, 0))

            if j < p["NXYZ"][1] - 1:
                # Y direction
                v_up = m["dXYZc"][(i, j + 1, k)]
                up_idx = v_up[0]
                dn_idx = v_dn[0]
                x, y, z = v_dn[1:4]
                dx, dy, dz = v_dn[4:7]
                xf = x
                yf = y + dy / 2.0
                zf = z

                # only cartesian
                A = dx * dz
                m["CON"].append((up_idx, dn_idx, xf, yf, zf, A, 0))

            if i < p["NXYZ"][0] - 1:
                # X direction
                v_up = m["dXYZc"][(i + 1, j, k)]
                up_idx = v_up[0]
                dn_idx = v_dn[0]
                x, y, z = v_dn[1:4]
                dx, dy, dz = v_dn[4:7]
                xf = x + dx / 2.0
                yf = y
                zf = z

                if "CARTESIAN" in p["TYPE"]:
                    A = dy * dz
                elif "CYLINDRICAL" in p["TYPE"]:
                    A = dz * 2.0 * xf * np.pi  # dz * circ
                elif "SPHERICAL" in p["TYPE"]:
                    A = 4.0 * np.pi * xf ** 2  # spherical area

                m["CON"].append((up_idx, dn_idx, xf, yf, zf, A, 0))

if VERBOSE > 2:
    print("\nCell connections")
    print(m["CON"])

# compute vertices of hexahedral blocks
# ****************************
m["Vidx"] = {}
m["Vxyz"] = {}

nx, ny, nz = p["NXYZ"]
nx1, ny1, nz1 = [v + 1 for v in p["NXYZ"]]
nz1ny1 = nz1 * ny1

```

```
ijk = 0  # zero-based index

for i, x in enumerate(m["vXYZc"][0]):
    for j, y in enumerate(m["vXYZc"][1]):
        for k, z in enumerate(m["vXYZc"][2]):

            v_here = m["dXYZc"][(i, j, k)]
            dx2, dy2, dz2 = [v / 2.0 for v in v_here[4:7]]
            indicies = []

            # xmin, ymin
            idx = k + nz1 * j + nz1ny1 * i
            indicies.extend([idx, idx + 1])  # [0,1]
            m["Vxyz"][idx] = (x - dx2, y - dy2, z - dz2)
            if k == nz - 1:
                m["Vxyz"][idx + 1] = (x - dx2, y - dy2, z + dz2)

            # xmin, ymax
            idx = k + nz1 * (j + 1) + nz1ny1 * i
            indicies.extend([idx, idx + 1])  # [2,3]
            if j == ny - 1:
                m["Vxyz"][idx] = (x - dx2, y + dy2, z - dz2)
                if k == nz - 1:
                    m["Vxyz"][idx + 1] = (x - dx2, y + dy2, z + dz2)

            # xmax, ymin
            idx = k + nz1 * j + nz1ny1 * (i + 1)
            indicies.extend([idx, idx + 1])  # [4,5]
            if i == nx - 1:
                m["Vxyz"][idx] = (x + dx2, y - dy2, z - dz2)
                if k == nz - 1:
                    m["Vxyz"][idx + 1] = (x + dx2, y - dy2, z + dz2)

            # xmax, ymax
            idx = k + nz1 * (j + 1) + nz1ny1 * (i + 1)
            indicies.extend([idx, idx + 1])  # [6,7]
            if (i == nx - 1) and (j == ny - 1):
                m["Vxyz"][idx] = (x + dx2, y + dy2, z - dz2)
                if k == nz - 1:
                    m["Vxyz"][idx + 1] = (x + dx2, y + dy2, z + dz2)
            m["Vidx"][ijk] = tuple(indicies)
            ijk += 1

if VERBOSE > 3:
    print("\nZero-based Indices of Vertices")
    print(m["Vidx"])

    print("\nCoordinates of Vertices (x,y,z)")
    print(m["Vxyz"])

# ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
#       write explicit mesh files
# ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
def write_uge_file(BASE_FN, M, order="old"):

    EXFN = f"{BASE_FN}.uge"
    fh = open(EXFN, "w")

    DOMFN = f"{BASE_FN}-domain.h5"
    dom = File(DOMFN, "w")  # hdf5

    # CELL block
    ncells = len(M["VOL"])
    w = int(np.ceil(np.log10(float(ncells + 1))))  # width for formatting ints

    fh.write(f"CELLS {ncells}\n")
    for cidx in range(ncells):
        x, y, z = M["XYZc"][cidx][:3]
        # cell ID; x,y,z of cell center; cell volume
        fh.write(
            f"{cidx+1:{w}} {x:.{df}E} {y:.{df}E} {z:.{df}E} {M['VOL'][cidx]:.{vf}E}\n"
        )

    # CONNECTIONS block
    nconn = len(M["CON"])
    fh.write(f"CONNECTIONS {nconn}\n")
    for cidx in range(nconn):
```

```
848         iup, idn, x, y, z, A, _ = M["CON"][cidx]
849         # upstr cell ID; downst cell ID; x,y,z of connection center; area perpendicular to connection
850         fh.write(
851             f"{iup+1:{w}} {idn+1:{w}} {x:.{df}E} {y:.{df}E} {z:.{df}E} {A:.{af}E}\n"
852         )
853
854     # ELEMENTS block (PFLOTRAN doesn't use uge section, paraview uses h5 data)
855     nverts = len(M["Vxyz"])
856     w = int(np.ceil(np.log10(float(nverts + 1))))
857     # H=hexahedral, vert ID for 8 corners of hex
858     fstr = f"H %{w}i %{w}i %{w}i %{w}i %{w}i %{w}i %{w}i %{w}i\n"
859     h5_cells = []
860     h5_cents = []
861
862     fh.write(f"ELEMENTS {ncells}\n")
863     for eidx in range(ncells):
864         ii = [i + 1 for i in M["Vidx"][eidx]]  # convert to 1-based index
865         # Exodus II numbering scheme (fig 4 on p. 10 of SAND92-2137)
866         if order == "old":
867             this_element = (ii[0], ii[4], ii[6], ii[2], ii[1], ii[5], ii[7], ii[3])
868         else:
869             this_element = tuple(ii)
870         fh.write(fstr % this_element)
871
872         ee = [9]  # "hexahedral (8+1)"
873         # paraview-read hdf5 arrays are 0-based
874         ee.extend([e - 1 for e in this_element])
875         h5_cells.extend(ee)
876
877         h5_cents.append(M["XYZc"][eidx])
878
879     hg_cells = np.array(h5_cells, dtype=np.int32)
880     dom.create_dataset("Domain/Cells", data=h5_cells)
881     del h5_cells
882
883     # float32 is the default type in xdmf, but pflotran writes 64-bit
884     h5_cents = np.array(h5_cents, dtype=np.float64)
885     dom.create_dataset("Domain/XC", data=h5_cents[:, 0])
886     dom.create_dataset("Domain/YC", data=h5_cents[:, 1])
887     dom.create_dataset("Domain/ZC", data=h5_cents[:, 2])
888     del h5_cents
889
890     h5_verts = []
891
892     # VERTICES block (PFLOTRAN doesn't use uge section, paraview uses h5 data)
893     fh.write(f"VERTICES {nverts}\n")
894     for vidx in range(nverts):
895         x, y, z = M["Vxyz"][vidx]
896         # x,y,z coordinates of vertices
897         fh.write(f"{x:.{df}E} {y:.{df}E} {z:.{df}E}\n")
898
899         h5_verts.append((x, y, z))
900
901     fh.close()
902     h5_verts = np.array(h5_verts, dtype=np.float64)
903     dom.create_dataset("Domain/Vertices", data=h5_verts)
904     del h5_verts
905     dom.close()
906
907     return (EXFN, DOMFN)
908
909
910 # call previous
911 efn, dfn = write_uge_file(base_fn, m)
912
913 # ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
914 #       write explicit region/boundary files
915 # ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
916
917 # vectors of coordinates for cell edges, inserting origin as 0th element
918 cumDX = tuple(np.cumsum(np.array([v0] + v)) for v, v0 in zip(p["DXYZ"], p["ORIGIN"]))
919
920 for bn in r.keys():
921
922     if r[bn]["DIM"] == "POINT":
923
924         # only way to specify a point is via coordinates
```

```
925         # write the id of cell containing point (may fall on boundary)
926         idx = find_point_idx_up(r[bn]["COORDINATE"], cumDX, bn)
927 
928         cell_id = m["dXYZc"][idx][0]
929         if VERBOSE > 1:
930             print(f"cell ijk: {idx}, cell id: {cell_id}")
931         txt_fn = f"{base_fn}-region-POINT-{bn}.txt"
932         r[bn]["FILENAME"] = txt_fn
933         fh = open(txt_fn, "w")
934         ii = cell_id + 1
935         fh.write(f"{ii}\n")
936         r[bn]["REG_CELL_IDS"] = np.array([ii,], dtype=np.int32)
937         fh.close()
938 
939     elif r[bn]["DIM"] == "SURFACE":
940         # write connections associated with boundary conditions
941         # i_min, j_min, k_min
942         # i_max, j_max, k_max
943         idx = []
944 
945         # specify COORDINATES and FACE
946         if "COORDS" in r[bn]:
947             idx.append(find_point_idx_up(r[bn]["COORDS"][0], cumDX, bn))
948             idx.append(find_point_idx_down(r[bn]["COORDS"][1], cumDX, bn))
949             face_idx = face_str_to_idx(r[bn]["FACE"], bn)
950 
951         # specify BLOCK and FACE
952         if "BLOCK" in r[bn]:
953             idx.append((r[bn]["BLOCK"][0], r[bn]["BLOCK"][2], r[bn]["BLOCK"][4]))
954             idx.append((r[bn]["BLOCK"][1], r[bn]["BLOCK"][3], r[bn]["BLOCK"][5]))
955             face_idx = face_str_to_idx(r[bn]["FACE"], bn)
956 
957         # specify CARTESIAN_BOUNDARY
958         if "CARTESIAN_BOUNDARY" in r[bn]:
959             face_idx = face_str_to_idx(r[bn]["CARTESIAN_BOUNDARY"], bn)
960 
961             nXYZ = [v - 1 for v in p["NXYZ"]]
962 
963             if face_idx == 1:
964                 # WEST (x-min, full range of y and z)
965                 idx.append([0, 0, 0])
966                 idx.append([0, nXYZ[1], nXYZ[2]])
967             elif face_idx == 2:
968                 # EAST (x-max, full range of y and z)
969                 idx.append([nXYZ[0], 0, 0])
970                 idx.append([nXYZ[0], nXYZ[1], nXYZ[2]])
971             elif face_idx == 3:
972                 # SOUTH (y-min, full range of x and z)
973                 idx.append([0, 0, 0])
974                 idx.append([nXYZ[0], 0, nXYZ[2]])
975             elif face_idx == 4:
976                 # NORTH (y-max, full range of x and z)
977                 idx.append([0, nXYZ[1], 0])
978                 idx.append([nXYZ[0], nXYZ[1], nXYZ[2]])
979             elif face_idx == 5:
980                 # BOTTOM (z-min, full range of x and y)
981                 idx.append([0, 0, 0])
982                 idx.append([nXYZ[0], nXYZ[1], 0])
983             elif face_idx == 6:
984                 # TOP (z-max, full range of x and y)
985                 idx.append([0, 0, nXYZ[2]])
986                 idx.append([nXYZ[0], nXYZ[1], nXYZ[2]])
987 
988         # also have option to save this info in hdf5 format
989         txt_fn = f"{base_fn}-region-SURFACE-{bn}.ex"
990         r[bn]["FILENAME"] = txt_fn
991         fh = open(txt_fn, "w")
992         nDIR = [(i1 - i0) + 1 for i0, i1 in zip(idx[0], idx[1])]
993         nCONN = nDIR[0] * nDIR[1] * nDIR[2]
994         cell_ids = []
995         face_ids = []
996         fh.write(f"CONNECTIONS {nCONN}\n")
997         if VERBOSE > 1:
998             print("CONN i range:", list(range(idx[0][0], idx[1][0] + 1)))
999             print("CONN j range:", list(range(idx[0][1], idx[1][1] + 1)))
1000            print("CONN k range:", list(range(idx[0][2], idx[1][2] + 1)))
1001
```

```
        for i in range(idx[0][0], idx[1][0] + 1):
            for j in range(idx[0][1], idx[1][1] + 1):
                for k in range(idx[0][2], idx[1][2] + 1):
                    v = m["dXYZc"][(i, j, k)]
                    ii = v[0] + 1
                    x, y, z = v[1:4]
                    dx, dy, dz = v[4:7]
                    sign = 1.0

                    if face_idx in [1, 3, 5]:
                        sign = -1.0

                    if face_idx in [1, 2]:  # X-direction connection
                        X = x + sign * dx / 2.0
                        Y = y
                        Z = z
                        if "CARTESIAN" in p["TYPE"]:
                            A = dy * dz
                        elif "CYLINDRICAL" in p["TYPE"]:
                            # dz * circumference
                            A = dz * 2.0 * X * np.pi
                        elif "SPHERICAL" in p["TYPE"]:
                            # spherical area
                            A = 4.0 * np.pi * X ** 2

                    elif face_idx in [3, 4]:  # Y-direction connection
                        X = x
                        Y = y + sign * dy / 2.0
                        Z = z

                        # only ever cartesian
                        A = dx * dz

                    else:  # face_idx in [5, 6] # Z-direction connection
                        X = x
                        Y = y
                        Z = z + sign * dz / 2.0

                        if "CARTESIAN" in p["TYPE"]:
                            A = dx * dy
                        elif "CYLINDRICAL" in p["TYPE"]:
                            # annular area
                            A = np.pi * 2.0 * X * dx

                    fh.write(f"{ii} {X:.{df}E} {Y:.{df}E} {Z:.{df}E} {A:.{af}E}\n")
                    cell_ids.append(ii)
                    face_ids.append(face_idx)  # same for every element

        r[bn]["REG_FACE_IDS"] = np.array(face_ids, dtype=np.int32)
        r[bn]["REG_CELL_IDS"] = np.array(cell_ids, dtype=np.int32)

    elif r[bn]["DIM"] == "VOLUME":
        idx = []

        # specify COORDINATES (no FACE)
        if "COORDS" in r[bn]:
            idx.append(find_point_idx_up(r[bn]["COORDS"][0], cumDX, bn))
            idx.append(find_point_idx_down(r[bn]["COORDS"][1], cumDX, bn))

        # specify BLOCK (no FACE)
        elif "BLOCK" in r[bn]:
            idx.append((r[bn]["BLOCK"][0], r[bn]["BLOCK"][2], r[bn]["BLOCK"][4]))
            idx.append((r[bn]["BLOCK"][1], r[bn]["BLOCK"][3], r[bn]["BLOCK"][5]))

        txt_fn = f"{base_fn}-region-VOLUME-{bn}.txt"
        r[bn]["FILENAME"] = txt_fn
        fh = open(txt_fn, "w")
        cell_ids = []
        if VERBOSE > 1:
            print("VOL i range:", list(range(idx[0][0], idx[1][0] + 1)))
            print("VOL j range:", list(range(idx[0][1], idx[1][1] + 1)))
            print("VOL k range:", list(range(idx[0][2], idx[1][2] + 1)))

        for i in range(idx[0][0], idx[1][0] + 1):
            for j in range(idx[0][1], idx[1][1] + 1):
                for k in range(idx[0][2], idx[1][2] + 1):
                    v = m["dXYZc"][(i, j, k)]
```

```
                    ii = v[0] + 1
                    fh.write(f"{ii}\n")
                    # PFLOTRAN-read hdf5 arrays are 1-based
                    cell_ids.append(ii)

        r[bn]["REG_CELL_IDS"] = np.array(cell_ids, dtype=np.int32)

h5fn = f"{base_fn}-regions.h5"
h5 = File(h5fn, "w")
# PFLOTRAN-read hdf5 arrays are 1-based
global_cell_ids = np.arange(len(m["VOL"]), dtype=np.int32) + 1
h5.create_dataset("Materials/Cell Ids", data=global_cell_ids)

for bn in r.keys():
    h5.create_dataset(f"Regions/{bn}/Cell Ids", data=r[bn]["REG_CELL_IDS"])
    if r[bn]["DIM"] == "SURFACE":
        h5.create_dataset(f"Regions/{bn}/Face Ids", data=r[bn]["REG_FACE_IDS"])

h5.close()

# ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
#       write replacement input file blocks
# ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
outfn = f"{base_fn}-explicit.in"
outfh = open(outfn, "w")
PRINTED = False

# put the GRID and REGION cards where the GRID card used to be
# TODO the position of each component should be remembered and replaced with new version

outfh.write("# comments and leading/trailing whitespace have been stripped\n")
outfh.write("# and '/' tokens converted to END\n")
for j in range(len(rows)):
    if context[j] == 1:
        outfh.write(" ".join(rows[j]) + "\n")
    elif context[j] == -1 and not PRINTED:
        echostring = ""
        for item in p["ECHO"]:
            echostring += "\n" + " ".join([" "] + item)

        # grid block is required
        outfh.write(
            f"""GRID
  TYPE UNSTRUCTURED_EXPLICIT {efn}
  DOMAIN_FILENAME {dfn} {echostring}
END
"""
        )

        for bn in r.keys():
            outfh.write(
                f"""REGION {bn}
  FILE {r[bn]["FILENAME"]}
END
"""
            )
        PRINTED = True

outfh.close()

if VERBOSE > 0:
    print(f"--------- finished processing GRID/REGION inputs from {fn} ---------")

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if READ_IN_ELEVATIONS:
    # element tops/bottoms assumed piecewise flat (i.e., stair-stepped)
    # values read here are elevations of center of top/bottom faces
    # everything in this section updates the mesh properties in-place
    # any multi-continuum stuff should then just work with new variable-z data

    try:
        zfn = f"{base_fn}-zelev.h5"
        zfh = File(zfn, "r")
    except OSError:
        READ_IN_ELEVATIONS = False
        if VERBOSE > 0:
```

```
            print(f"WARNING: elevation file name '{zfn}' not found, skipping ...")

    if READ_IN_ELEVATIONS and ("CARTESIAN" in p["TYPE"]):
        # there will 2D arrays of elevations values for N+1 arrays,
        # top of each layer (1-N), and bottom of lowest layer (00)

        Ztopbot = []
        nlays = len(zfh.keys()) - 1
        Ztopbot.append(zfh["elev-00"][:, :])

        for lay in range(nz):
            Ztopbot.append(zfh[f"elev-{lay + 1:02}"][:, :])

        if VERBOSE > 1:
            print(
                f"opened '{zfn}' reading {len(Ztopbot)} "
                f"arrays, each of shape {Ztopbot[0].shape}"
            )

        for j, zarray in enumerate(Ztopbot):
            if zarray.shape != p["NXYZ"][0:2]:
                print(
                    f"ERROR: incorrect shape of array read from '{zfn}' for level {j} "
                    f"in elev-{j:02}: {zarray.shape}. should be {p['NXYZ'][0:2]}"
                )
                sys.exit(1)

        # modify mesh and write a z-variable version
        # 0) only makes sense for CARTESIAN (not CYLINDRICAL or SPHERICAL)
        # 1) z-component of element centers
        # 2) dz values
        # 3) element volumes
        # 4) z-component of connection centers (not guaranteed orthogonal anymore)
        # 5) z-component of element vertices
        # connectivity, xy coordinates/extents remain the same
        # assume regions computed apply to initial mesh don't change for now...

        # compute new dz & z (matricies, rather than lists)
        DZ = np.empty(p["NXYZ"], dtype=np.float64)
        Z = np.empty(p["NXYZ"], dtype=np.float64)  # element center
        for lay in range(nz):
            DZ[:, :, lay] = Ztopbot[lay + 1] - Ztopbot[lay]
            Z[:, :, lay] = (Ztopbot[lay + 1] + Ztopbot[lay]) / 2.0

        # re-compute locations and volumes
        ii = 0
        for i in range(p["NXYZ"][0]):
            for j in range(p["NXYZ"][1]):
                for k in range(p["NXYZ"][2]):
                    (x, y, z, dx, dy, dz) = m["XYZc"][ii]  # old value
                    m["XYZc"][ii] = (x, y, Z[i, j, k], dx, dy, DZ[i, j, k])
                    # m["IDX"] unchanged
                    m["dXYZc"][(i, j, k)] = (ii, x, y, Z[i, j, k], dx, dy, DZ[i, j, k])
                    m["VOL"][ii] = dx * dy * DZ[i, j, k]
                    ii += 1

        if VERBOSE > 2:
            print("\nUpdated (z-variable) Cell coordinates / extents (x,y,z,dx,dy,dz)")
            print(m["XYZc"])

            print("\nUpdated (z-variable) Cell Volumes")
            print(m["VOL"])

        valid_var_z_opts = ["average", "minimum"]
        # TODO: should create another option to compute "overlap."
        # Adjacent elements might have same dz, but different origins
        # this method
        if not VARIABLE_Z_CONNECTION_AREA in valid_var_z_opts:
            print(
                f"invalid value for VARIABLE_Z_CONNECTION_AREA "
                f"{valid_var_z_opts}: '{VARIABLE_Z_CONNECTION_AREA}'"
            )
            sys.exit(1)

        # re-compute connection locations and areas
        ii = 0
        for i in range(p["NXYZ"][0]):  # x
```

```
            for j in range(p["NXYZ"][1]):  # y
                for k in range(p["NXYZ"][2]):  # z

                    # down = this element
                    v_dn = m["dXYZc"][(i, j, k)]

                    if k < p["NXYZ"][2] - 1:
                        # Z direction
                        v_up = m["dXYZc"][(i, j, k + 1)]
                        up_idx = v_up[0]
                        dn_idx = v_dn[0]
                        xdn, ydn, zdn = v_dn[1:4]
                        xup, yup, zup = v_up[1:4]
                        dx, dy, dzdn = v_dn[4:7]
                        dzup = v_up[6]
                        xf = xdn
                        yf = ydn
                        zf = zdn + dzdn / 2.0
                        A = dx * dy  # no projection in z-direction
                        m["CON"][ii] = (up_idx, dn_idx, xf, yf, zf, A, 0)
                        ii += 1

                    if j < p["NXYZ"][1] - 1:
                        # Y direction
                        v_up = m["dXYZc"][(i, j + 1, k)]
                        up_idx = v_up[0]
                        dn_idx = v_dn[0]
                        xdn, ydn, zdn = v_dn[1:4]
                        xup, yup, zup = v_up[1:4]
                        dxup, dyup, dzup = v_up[4:7]
                        dxdn, dydn, dzdn = v_dn[4:7]
                        dzup = v_up[6]
                        xf = xdn
                        yf = ydn + dydn / 2.0
                        zf = (zup + zdn) / 2.0  # avg z-elev
                        if VARIABLE_Z_CONNECTION_AREA == "minimum":
                            dz = min(dzup, dzdn)
                        else:
                            dz = (dzup + dzdn) / 2.0
                        # project area onto normal of line connecting centers
                        Ly = dyup / 2.0 + dxup / 2.0
                        A = dxdn * dz * (Ly / np.sqrt((zup - zdn) ** 2 + Ly ** 2))
                        m["CON"][ii] = (up_idx, dn_idx, xf, yf, zf, A, 0)
                        ii += 1

                    if i < p["NXYZ"][0] - 1:
                        # X direction
                        v_up = m["dXYZc"][(i + 1, j, k)]
                        up_idx = v_up[0]
                        dn_idx = v_dn[0]
                        xdn, ydn, zdn = v_dn[1:4]
                        xup, yup, zup = v_up[1:4]
                        dxup, dyup, dzup = v_up[4:7]
                        dxdn, dydn, dzdn = v_dn[4:7]
                        dzup = v_up[6]
                        xf = xdn + dxdn / 2.0
                        yf = ydn
                        zf = (zup + zdn) / 2.0  # avg z-elev
                        if VARIABLE_Z_CONNECTION_AREA == "minimum":
                            dz = min(dzup, dzdn)
                        else:
                            dz = (dzup + dzdn) / 2.0
                        Lx = dxup / 2.0 + dxdn / 2.0
                        A = dydn * dz * (Lx / np.sqrt((zup - zdn) ** 2 + Lx ** 2))
                        m["CON"][ii] = (up_idx, dn_idx, xf, yf, zf, A, 0)
                        ii += 1

        if VERBOSE > 2:
            print("\nUpdated (z-variable) Cell connections")
            print(m["CON"])

        valid_var_z_verts = ["piecewise", "continuous"]
        if not VARIABLE_Z_VERTICES in valid_var_z_verts:
            print(
                f"invalid value for VARIABLE_Z_VERTICES "
                f"{valid_var_z_verts}: '{VARIABLE_Z_VERTICES}'"
            )
```

```
            sys.exit(1)

        if VARIABLE_Z_VERTICES == "continuuous":
            # this approach "re-uses" vertices between adjacent elements
            # only update z-coordinate of element vertices
            for i in range(p["NXYZ"][0]):
                for j in range(p["NXYZ"][1]):
                    for k in range(p["NXYZ"][2]):

                        v_here = m["dXYZc"][(i, j, k)]
                        zdn = Z[i, j, k] - DZ[i, j, k] / 2.0
                        zup = Z[i, j, k] + DZ[i, j, k] / 2.0

                        # xmin, ymin
                        idx = k + nz1 * j + nz1ny1 * i
                        xx, yy, _ = m["Vxyz"][idx]
                        m["Vxyz"][idx] = (xx, yy, zdn)
                        if k == nz - 1:
                            m["Vxyz"][idx + 1] = (xx, yy, zup)

                        # xmin, ymax
                        idx = k + nz1 * (j + 1) + nz1ny1 * i
                        xx, yy, _ = m["Vxyz"][idx]
                        if j == ny - 1:
                            m["Vxyz"][idx] = (xx, yy, zdn)
                            if k == nz - 1:
                                m["Vxyz"][idx + 1] = (xx, yy, zup)

                        # xmax, ymin
                        idx = k + nz1 * j + nz1ny1 * (i + 1)
                        xx, yy, _ = m["Vxyz"][idx]
                        if i == nx - 1:
                            m["Vxyz"][idx] = (xx, yy, zdn)
                            if k == nz - 1:
                                m["Vxyz"][idx + 1] = (xx, yy, zup)

                        # xmax, ymax
                        idx = k + nz1 * (j + 1) + nz1ny1 * (i + 1)
                        xx, yy, _ = m["Vxyz"][idx]
                        if (i == nx - 1) and (j == ny - 1):
                            m["Vxyz"][idx] = (xx, yy, zdn)
                            if k == nz - 1:
                                m["Vxyz"][idx + 1] = (xx, yy, zup)

            zefn, zdfn = write_uge_file(f"{base_fn}-varz", m)
        else:
            # piecewise approach uses 8 new verticies for each element
            # completely replaces the Vxyz and Vidx arrays
            m["Vidx"] = {}
            m["Vxyz"] = {}

            ijk = 0  # element index
            vidx = 0  # vertex index

            for i, x in enumerate(m["vXYZc"][0]):
                for j, y in enumerate(m["vXYZc"][1]):
                    for k in range(p["NXYZ"][2]):

                        zdn = Z[i, j, k] - DZ[i, j, k] / 2.0
                        zup = Z[i, j, k] + DZ[i, j, k] / 2.0
                        v_here = m["dXYZc"][(i, j, k)]
                        dx2, dy2, dz2 = [v / 2.0 for v in v_here[4:7]]

                        m["Vxyz"][vidx] = (x - dx2, y - dy2, zdn)
                        m["Vxyz"][vidx + 1] = (x + dx2, y - dy2, zdn)
                        m["Vxyz"][vidx + 2] = (x + dx2, y + dy2, zdn)
                        m["Vxyz"][vidx + 3] = (x - dx2, y + dy2, zdn)

                        m["Vxyz"][vidx + 4] = (x - dx2, y - dy2, zup)
                        m["Vxyz"][vidx + 5] = (x + dx2, y - dy2, zup)
                        m["Vxyz"][vidx + 6] = (x + dx2, y + dy2, zup)
                        m["Vxyz"][vidx + 7] = (x - dx2, y + dy2, zup)

                        m["Vidx"][ijk] = tuple(vidx + i for i in range(8))
                        vidx += 8
                        ijk += 1
```

```
            zefn, zdfn = write_uge_file(f"{base_fn}-varz", m, order="fixed")

        if VERBOSE > 3:
            print("\nUpdated (z-variable) zero-based Indices of Vertices")
            print(m["Vidx"])

            print("\nUpdated (z-variable) Vertices (x,y,z)")
            print(m["Vxyz"])


# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if MULTIPOROSITY:
    # read in file specifying multiporosity-specific stuff
    mfn = "multiporosity.in"
    mfh = open(mfn, "r")

    if VERBOSE > 0:
        print(f"--------- processing multiporosity inputs from {mfn} ---------")

    mrows = parse_lines(clean_lines(mfh.readlines()))
    mfh.close()

    if VERBOSE > 2:
        print(f"processed rows of '{mfn}' read in")
        for mr in mrows:
            print(mr)

    Q = {}
    mr = "INVALID"
    for j, row in enumerate(mrows):
        # %%%%%%%%%%%%%%%%%%%%%%%%%
        # process keywords and arguments

        if "PINC" in row:
            # always first, so "mr" should be available below
            mr = row[1]  # region name
            Q[mr] = {}

        elif "TYPE" in row:
            mt = row[1].upper()
            if mt == "DOUBLE_POROSITY" or mt == "DOUBLE_PERMEABILITY":
                Q[mr]["N"] = 2
            elif mt == "TRIPLE_POROSITY" or mt == "TRIPLE_PERMEABILITY":
                Q[mr]["N"] = 3
            elif mt == "N_POROSITY" or mt == "N_PERMEABILITY":
                Q[mr]["N"] = int(row[2])
            else:
                print(f"unknown multicontinuum type '{mt}' in region '{mr}'")
                sys.exit(1)
            Q[mr]["TYPE"] = mt

        elif "VOLUME_FRACTIONS" in row:
            # <float> volume fraction for N-1 porosities which sum to 1
            Q[mr]["VF"] = [ffloat(x) for x in row[1:]]

        elif "DOMAIN_LENGTHS" in row:
            # length of subdomain; <float> for each of N porosities
            Q[mr]["LENS"] = [ffloat(x) for x in row[1:]]

        elif "NUMBER_ELEMENTS" in row:
            # N_elements in subdomain <int> for each of N porosities (frac = 1)
            Q[mr]["NUM_EL"] = [int(x) for x in row[1:]]

        elif "GEOMETRIES" in row:
            # geometry relating dx to (dA,dV), one word per subdomain
            gv = [x.upper() for x in row[1:]]
            Q[mr]["GEOM"] = gv

            for g in gv:
                if not g in ["FRACTURE", "SLAB", "NESTED_CUBES", "NESTED_SPHERES"]:
                    print(f"unknown type of GEOMETRIES '{g}' in region '{mr}'")
                    sys.exit(1)

        elif "DIRECTIONS" in row:
            # (e.g., X, Y, Z, +X, -X) assume +Y for if not specified (no gravity)
            # improve/modify visualization of output in paraview
```

```
            # impacts results w/ anisotropic permeability or thermal conductivity
            Q[mr]["DIRS"] = [x.upper() for x in row[1:]]

        elif "MESH" in row:
            # mesh for each subdomain (on subsequent rows, with END block)
            # order of rows in mesh block correspond to the order of the continuua
            # UNIFORM: (don't need to specify anything else,
            #           computed from NUMBER_ELEMENTS and DOMAIN_LENGTHS)
            # GEOMETRIC x0 factor (initial spacing, growth factor)
            # DX values (arbitrary mesh spacing values)

            Q[mr]["MESH"] = []

            for i in range(1, len(mrows) - j):
                jj = j + i
                mt = mrows[jj][0].upper()

                if mt == "END":
                    break
                if mt == "UNIFORM":
                    Q[mr]["MESH"].append(["UNIFORM"])
                # elif mt == "GEOMETRIC":
                #    Q[mr]["MESH"].append(
                #        ["GEOMETRIC"] + [ffloat(x) for x in mrows[jj][1:]]
                #    )
                elif mt == "FRACTURE":
                    # copy of initial mesh (in this region)
                    Q[mr]["MESH"].append(["FRACTURE"])
                elif mt == "DX":
                    # TODO add ability to specify DX similar to DXYZ notation
                    # (i.e., 17@2.5 and "\" end-of line continuuation characters)
                    # for now just read in a single row of DX values
                    Q[mr]["MESH"].append(["DX"] + [ffloat(x) for x in mrows[jj][1:]])
                else:
                    print(f"unknown MESH type '{mt}' in region '{mr}'")
                    sys.exit(1)

        elif "SPATIAL_CONNECTIVITY" in row:
            # dual permeability
            # is there connectivity between spatial locations within each subdomain?
            # 1 = True connectivity (i.e., typical fracture),
            # 0 = False connectivity (i.e., typical matrix)
            Q[mr]["CONN"] = [bool(int(x)) for x in row[1:]]

        elif "FAR_FIELD_CONNECTION" in row:
            # FAR_FIELD_CONNECTION
            # for subdomains with SPATIAL_CONNECTIVITY = 1 aren't used
            # 0 is no-flow (default)
            # 1-N means it connects to other domain (a fracture?)
            Q[mr]["FAR_FIELD"] = [int(x) for x in row[1:]]

        elif "SLAB_AREAS" in row:
            # SLAB_AREAS is optional (only makes sense for slab)
            Q[mr]["SLAB_AREAS"] = [ffloat(x) for x in row[1:]]

        elif "CONSTRAIN_TOTAL_VOL" in row:
            # logical (default is False, add for True)
            Q[mr]["CTV"] = True

    if VERBOSE > 0:
        print(f"processed {len(Q)} PINC blocks")
        for mr in Q.keys():
            print(mr)

    if VERBOSE > 2:
        print("dump of initially processes multicontinuum input")
        for mr in Q.keys():
            print(mr, Q[mr])

    # perform checks on inputs, compute intermediates
    # %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    for mr in Q.keys():
        # check the correct number/type of things were specified for number of subdomains
        # can't do this as it is being read because we don't know the order of the cards
        # N could be specified last

        if "N" in Q[mr]:
```

```
            N = Q[mr]["N"]
        else:
            print(f"specify TYPE of multicontinuum problem in region '{mr}'")
            sys.exit(1)

        if "VF" in Q[mr]:
            if len(Q[mr]["VF"]) == (N - 1):
                for vfv in Q[mr]["VF"]:
                    if vfv <= 0.0:
                        print(
                            f"0: specify exactly {N-1} (N-1) <float> positive VOLUME_FRACTIONs"
                            f" ({Q[mr]['VF']}) in region '{mr}'"
                        )
                Q[mr]["VF"].append(1.0 - sum(Q[mr]["VF"]))
            else:
                print(
                    f"1: specify exactly {N-1} (N-1) <float> VOLUME_FRACTIONs "
                    f"({Q[mr]['VF']}) in region '{mr}'"
                )
                sys.exit(1)
        else:
            print(f"2: specify N-1 <float> VOLUME_FRACTIONs in region '{mr}'")
            sys.exit(1)

        if "LENS" in Q[mr]:
            if len(Q[mr]["LENS"]) == N:
                for l in Q[mr]["LENS"]:
                    if l <= 0.0:
                        print(
                            f"0: specify exactly N positive <float> DOMAIN_LENGTHS "
                            f"({Q[mr]['LENS']}) in region '{mr}'"
                        )
                        sys.exit(1)
            else:
                print(
                    f"1: specify exactly N <float> DOMAIN_LENGTHS "
                    f"({Q[mr]['LENS']}) in region '{mr}'"
                )
                sys.exit(1)
        else:
            print(f"2: specify N <float> DOMAIN_LENGTHS in region '{mr}'")
            sys.exit(1)

        if "NUM_EL" in Q[mr]:
            if len(Q[mr]["NUM_EL"]) == N:
                for ne in Q[mr]["NUM_EL"]:
                    if ne < 1:
                        print(
                            f"0: specify exactly N positive <int> NUMBER_ELEMENTS "
                            f"({Q[mr]['NUM_EL']}) in region '{mr}'"
                        )
                        sys.exit(1)
            else:
                print(
                    f"1: specify exactly N <int> NUMBER_ELEMENTS "
                    f"({Q[mr]['NUM_EL']}) in region '{mr}'"
                )
                sys.exit(1)
        else:
            print(f"2: specify N <int> NUMBER_ELEMENTS in region '{mr}'")
            sys.exit(1)

        if "GEOM" in Q[mr]:
            if len(Q[mr]["GEOM"]) == N:
                pass
            else:
                print(
                    f"0: specify exactly N GEOMETRY types ({Q[mr]['GEOM']}) in region '{mr}'"
                )
                sys.exit(1)
        else:
            print(f"1: specify N GEOMETRY types in region '{mr}' ")
            sys.exit(1)

        if "DIRS" in Q[mr]:
            if len(Q[mr]["DIRS"]) == N:
                pass
```

```
            else:
                print(
                    f"0: specify exactly N DIRECTIONS ({Q[mr]['GEOM']}) in region '{mr}'"
                )
                sys.exit(1)
        else:
            # default behaivor (optional card)
            Q[mr]["DIRS"] = ["+Y"] * N

        if "MESH" in Q[mr]:
            if len(Q[mr]["MESH"]) == N:
                for cont, mm in enumerate(Q[mr]["MESH"]):
                    if mm[0] == "GEOMETRIC":
                        if len(mm) == 3:
                            if mm[1] <= 0.0 or mm[2] <= 0.0:
                                print(
                                    f"X0 {mm[1]} and GROWTH_FACTOR {mm[2]} in "
                                    f"MESH:GEOMETRIC:{cont} must be >=0 for region '{mr}'"
                                )
                                sys.exit(1)
                        else:
                            print(
                                f"specify two <float> parameters (X0,GROWTH_FACTOR) for "
                                f"MESH:GEOMETRIC:{cont} in region '{mr}'"
                            )
                            sys.exit(1)
                    elif mm[0] == "DX":
                        NE = Q[mr]["NUM_EL"][cont]
                        if (len(mm[1:]) == NE) or (len(mm[1:]) > NE):
                            for i in range(NE):
                                if mm[1 + i] <= 0.0:
                                    print(
                                        f"all DX values in MESH:DX:{cont} ({mm}) "
                                        f"must be positive in region '{mr}'"
                                    )
                                    sys.exit(1)
                            if len(mm[1:]) > NE:
                                ldx = len(mm[1:])
                                print(
                                    f"WARNING ignoring DX values beyond NE ({mm}) in region ({mr})"
                                )
                                for _ in range(ldx - NE):
                                    mm.pop()  # delete items from end of list
                        else:
                            print(
                                f"specify NUMBER_ELEMENT ({NE}) <float> dx values ({mm}) for "
                                f"MESH:GEOMETRIC:{cont} in region '{mr}'"
                            )
                            sys.exit(1)
            else:
                print(
                    f"specify MESH block with exactly N rows ({Q[mr]['MESH']}) "
                    f"in region '{mr}' [UNIFORM,GEOMETRIC,DX]"
                )
                sys.exit(1)
        else:
            print(
                f"specify MESH block with a row for each "
                f"continuum in region '{mr}' [UNIFORM,GEOMETRIC,DX]"
            )
            sys.exit(1)

        if "CONN" in Q[mr]:
            if len(Q[mr]["CONN"]) == N:
                pass
            else:
                print(
                    f"0: specify exactly N [0,1] <bool> SPATIAL_CONNECTIVITY "
                    f"values ({Q[mr]['CONN']}) in region '{mr}'"
                    " or use the TYPE shortcuts DOUBLE_POROSITY, DOUBLE_PERMEABILITY, etc."
                )
                sys.exit(1)
        else:
            if "TYPE" in Q[mr]:
                typ = Q[mr]["TYPE"]
                if typ == "DOUBLE_POROSITY":
                    Q[mr]["CONN"] = [1, 0]
```

```
                elif typ == "DOUBLE_PERMEABILITY":
                    Q[mr]["CONN"] = [1, 1]
                elif typ == "TRIPLE_POROSITY":
                    Q[mr]["CONN"] = [1, 0, 0]
                elif typ == "TRIPLE_PERMEABILITY":
                    Q[mr]["CONN"] = [1, 1, 1]
                elif typ == "N_POROSITY":
                    Q[mr]["CONN"] = [1]
                    Q[mr]["CONN"].extend([0] * Q[mr]["N"])
                elif typ == "N_PERMEABILITY":
                    Q[mr]["CONN"] = [1]
                    Q[mr]["CONN"].extend([1] * Q[mr]["N"])
                else:
                    print(
                        f"1: specify N [0,1] <bool> SPATIAL_CONNECTIVITY values in region '{mr}'"
                        " or use the TYPE shortcuts DOUBLE_POROSITY, DOUBLE_PERMEABILITY, etc."
                    )
                    sys.exit(1)
            else:
                print(
                    f"2: specify N [0,1] <bool> SPATIAL_CONNECTIVITY values in region '{mr}'"
                    " or use the TYPE shortcuts DOUBLE_POROSITY, DOUBLE_PERMEABILITY, etc."
                )
                sys.exit(1)

        if "FAR_FIELD" in Q[mr]:
            if len(Q[mr]["FAR_FIELD"]) == N:
                pass
            else:
                print(
                    f"0: specify exactly N <int> FAR_FIELD_CONNECTION values "
                    f"({Q[mr]['FAR_FIELD']}) in region '{mr}' (or leave out/optional)"
                )
                sys.exit(1)
        else:
            pass  # optional block doesn't do anything yet
            # print(f"1: specify N <int> FAR_FIELD_CONNECTION values in region '{mr}'")
            # sys.exit(1)

        if "SLAB_AREAS" in Q[mr]:
            if len(Q[mr]["SLAB_AREAS"]) == N:
                pass
            else:
                print(
                    f"0: specify exactly N <float> SLAB_AREAS values (only makes sense for slab geometry)
"
                    f"(Q[mr]['SLAB_AREAS']) in region {mr} (or leave out/optional)"
                )
                sys.exit(1)
        else:
            pass  # optional

        if "CTV" not in Q[mr]:
            Q[mr]["CTV"] = False

    if VERBOSE > 2:
        print("validated multiporosity inputs")
        print("dump of further processes and checked multicontinuum input")
        for mr in Q.keys():
            print(mr, Q[mr])

    def direction_to_vec(DIR):
        # returns a unit vector representing
        # X,Y,Z,+X,-X,+Y,-Y,+Z,-Z (no sign assumed positive)
        vec = np.array([0.0, 0.0, 0.0], dtype=np.float64)
        DIR = DIR.upper()

        if DIR[0] == "F":
            # i.e., no new "matrix" contiuum
            return vec

        if DIR[-1] == "X":
            vec[0] = 1.0
        elif DIR[-1] == "Y":
            vec[1] = 1.0
        elif DIR[-1] == "Z":
            vec[2] = 1.0
```

```
1771            else:
1772                print("ERROR: invalid last character in dir (X,Y,Z): {DIR}")
1773                sys.exit(1)
1774
1775            if len(DIR) == 1:
1776                return vec
1777            elif DIR[0] == "-":
1778                vec *= -1.0
1779            return vec
1780
1781        def build_1d_mesh(dx, xyz, DIR, CONT_VOL, GEOM, AREAS):
1782            # build matrix (i.e., not connected in space) mesh
1783            vec = direction_to_vec(DIR)
1784            XYZ = np.array(xyz[:3], dtype=np.float64)  # center of root cell
1785            dxdydz = np.array(xyz[3:], dtype=np.float64)  # extents of root cell
1786
1787            dx = np.array(dx)
1788            L = dx.sum()
1789            NE = dx.shape[0]
1790
1791            nx, ny, nz = (1, 1, 1)
1792            if abs(abs(vec[0]) - 1.0) < eps:
1793                MDIR = 0
1794                nx = NE
1795            elif abs(abs(vec[1]) - 1.0) < eps:
1796                MDIR = 1
1797                ny = NE
1798            else:
1799                MDIR = 2
1800                nz = NE
1801
1802            # compute cell coordinates
1803            el_cent = np.empty((NE, 3), dtype=np.float64)
1804            # in local coordinates
1805            for i in range(NE):
1806                el_cent[i, :] = vec[None, :] * (dx[:i].sum() + dx[i] / 2.0)
1807
1808            # compute element volumes & connection areas
1809            el_vol = np.empty((NE,), dtype=np.float64)
1810            c_areas = np.empty((NE,), dtype=np.float64)
1811            c_cent = np.empty((NE, 3), dtype=np.float64)
1812
1813            if GEOM == "SLAB":
1814                # constant area and volum
1815                if AREAS is not None:
1816                    c_areas[:] = AREAS  # allow to optionally override area calc for slab
1817                else:
1818                    c_areas[:] = CONT_VOL / L
1819                el_vol[:] = c_areas[:] * dx[:]
1820            elif GEOM == "NESTED_CUBES":
1821                # first element on outside / last element at center
1822                r = L - el_cent[:, MDIR]  # distance from center
1823                c_areas[:] = 4.0 * (r[:] + dx[:] / 2.0) ** 2
1824                el_vol[:] = dx[:] ** 3 / 2.0 + 6.0 * dx[:] * r[:] ** 2
1825            elif GEOM == "NESTED_SPHERES":
1826                r = L - el_cent[:, MDIR]
1827                c_areas[:] = 4.0 * np.pi * (r[:] + dx[:] / 2.0) ** 2
1828                el_vol[:] = np.pi * dx[:] / 3.0 * (dx[:] ** 2 + 12.0 * r[:] ** 2)
1829            else:
1830                print("ERROR: invalid GEOM in build_1d_mesh(): {GEOM}")
1831                sys.exit(1)
1832
1833            # shift physical domain to begin at _center_ of root element
1834            el_cent[:, :] += XYZ[None, :]
1835            c_cent[:, :] = el_cent[:, :] - dx[:, None] * vec[None, :] / 2.0
1836
1837            # vertices for elements (for paraview) shouldn't overlap
1838            # maybe 95% of the size, so they don't overlap (small gap)
1839
1840            # the volume of the elements computed from the verticies will not
1841            # be the same as that used by PFLOTRAN, but that was already the case
1842            # with SPHERICAL or CYLINDRICAL meshes (dy = 1.0)
1843
1844            #   XY planes         XZ planes         YZ planes
1845            #  z_lo    z_hi      y_lo    y_hi      x_lo    x_hi
1846            # 3----2  7----6   4----5  7----6   7----4  6----5
1847            # |    |  |    |   |    |  |    |   |    |  |    |
```

```
        # 0----1  4----5   0----1  3----2   3----0  2----1

        el_indx = np.empty((NE, 8), dtype=np.int32)  # hexahedra
        ids = {}
        verts = {}
        nx1, ny1, nz1 = [v + 1 for v in (nx, ny, nz)]
        nz1ny1 = nz1 * ny1

        for i in range(nx):
            for j in range(ny):
                for k in range(nz):
                    x = el_cent[i, 0]
                    y = el_cent[j, 1]
                    z = el_cent[k, 2]
                    ijk = max(i, j, k)  # 0-based

                    dx2, dy2, dz2 = (
                        dx[ijk] * vec[:] - dxdydz[:] * (1.0 - abs(vec[:])) * most
                    ) / 2.0

                    ids[ijk] = []

                    # xmin, ymin
                    idx = k + nz1 * j + nz1ny1 * i
                    ids[ijk].extend([idx, idx + 1])  # [0,1]
                    verts[idx] = (x - dx2, y - dy2, z - dz2)
                    if k == nz - 1:
                        verts[idx + 1] = (x - dx2, y - dy2, z + dz2)

                    # xmin, ymax
                    idx = k + nz1 * (j + 1) + nz1ny1 * i
                    ids[ijk].extend([idx, idx + 1])  # [2,3]
                    if j == ny - 1:
                        verts[idx] = (x - dx2, y + dy2, z - dz2)
                        if k == nz - 1:
                            verts[idx + 1] = (x - dx2, y + dy2, z + dz2)

                    # xmax, ymin
                    idx = k + nz1 * j + nz1ny1 * (i + 1)
                    ids[ijk].extend([idx, idx + 1])  # [4,5]
                    if i == nx - 1:
                        verts[idx] = (x + dx2, y - dy2, z - dz2)
                        if k == nz - 1:
                            verts[idx + 1] = (x + dx2, y - dy2, z + dz2)

                    # xmax, ymax
                    idx = k + nz1 * (j + 1) + nz1ny1 * (i + 1)
                    ids[ijk].extend([idx, idx + 1])  # [6,7]
                    if (i == nx - 1) and (j == ny - 1):
                        verts[idx] = (x + dx2, y + dy2, z - dz2)
                        if k == nz - 1:
                            verts[idx + 1] = (x + dx2, y + dy2, z + dz2)

        return (el_cent, el_vol, c_areas, dx, c_cent, ids, verts)

    # %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    # compute continuua mesh parameters
    CM = {}

    # These regions multicontinuum is applied should be disjoint?
    # Not sure what to do if they overlap...
    # connectivity will probably work, but need to think about
    # how to compute total volumes.

    for mr in Q.keys():
        if mr in r:
            if r[mr]["DIM"] == "VOLUME":
                pass
            else:
                print(
                    f"PINC '{mr}' mulst be associated "
                    f"with volumetric REGION: {r[mr]['DIM']}"
                )
                sys.exit(1)
        else:
            print(f"PINC '{mr}' has no corresponding REGION, out of: {r.keys()}")
            sys.exit(1)
```

```
    # modify m dictionary with mesh parameters in-place.
    m["INIT_VOL"] = m["VOL"]

    # volumes of elements in initial mesh then
    # bound total volume of sum of all elements total
    for mr in Q.keys():
        CM[mr] = {}
        dl = Q[mr]["LENS"]
        ne = Q[mr]["NUM_EL"]
        ge = Q[mr]["GEOM"]
        dr = Q[mr]["DIRS"]
        if "SLAB_AREAS" in Q[mr]:
            areas = Q[mr]["SLAB_AREAS"]
        else:
            areas = [None] * len(dl)

        # each cell in the inital domain has
        # several continuua attached to it
        # these are 1-based (hdf5 array for PFLOTRAN)
        cell_ids = r[mr]["REG_CELL_IDS"]
        # make a copy and reset to zero-based
        initial_reg_cell_ids = [i - 1 for i in cell_ids[:]]

        # empty list of lists with global cell ids for each continuum
        # to allow definition of regions
        region_ids = [[] for reg in range(Q[mr]["N"])]

        # create a dictionary for each element in physical domain,
        # which will have another continuum mesh attached to it
        # cid is now zero-based
        for cid in [i - 1 for i in cell_ids]:
            CM[mr][cid] = {}

            # all cells fit in initial root physical volume
            CM[mr][cid]["TOTAL_VOL"] = m["VOL"][cid]

            xyz = m["XYZc"][cid]  # center (x,y,z) & extents (dx,dy,dz)

            # cycle through continuua
            for ctm in range(Q[mr]["N"]):
                CM[mr][cid][ctm] = {}

                # total volume for this continuum
                # based on volume fraction and volume of element it is rooted in
                # volume of all elements in this continuum must add up to this\
                if "SLAB_AREAS" in Q[mr] and ge[ctm] == "SLAB":
                    # compute volume from specified L and area
                    # this only possible for slab geometry
                    cvol = dl[ctm] * areas[ctm]
                else:
                    cvol = CM[mr][cid]["TOTAL_VOL"] * Q[mr]["VF"][ctm]

                CM[mr][cid][ctm]["CONTINUUM_VOL"] = cvol
                CM[mr][cid][ctm]["VF"] = Q[mr]["VF"][ctm]

                mt = Q[mr]["MESH"][ctm]
                if mt[0] == "UNIFORM":
                    # uniform grid spacing
                    # SLAB, L specified by user

                    if ge[ctm] == "NESTED_SPHERES":
                        # domain size is constrained total volume
                        # L specified in input file not used
                        # L is radius of outermost nested sphere
                        L = (0.75 * cvol / np.pi) ** (1.0 / 3.0)
                    elif ge[ctm] == "NESTED_CUBES":
                        # L specified in input file not used
                        # L is half edge of outermost nested cube
                        L = (0.125 * cvol) ** (1.0 / 3.0)
                    else:
                        # for slab use L specified in input file
                        L = dl[ctm]

                    dx = np.ones((ne[ctm],)) * L / float(ne[ctm])

                    R = build_1d_mesh(dx, xyz, dr[ctm], cvol, ge[ctm], areas[ctm])
```

```
                    ec, ev, ca, dx, cc, id_dict, vert_dict = R
                    CM[mr][cid][ctm].update(
                        {
                            "GEOM": ge[ctm],
                            "MESH": mt[0],
                            "NE": ne[ctm],
                            "L": L,
                            "eXYZ": ec,
                            "DX": dx,
                            "eVOL": ev,
                            "DIR": dr[ctm],
                            "cAREA": ca,
                            "cXYZ": cc,
                            "idDict": id_dict,
                            "vertDict": vert_dict,
                        }
                    )

                elif mt[0] == "DX":
                    # all dx specified by user
                    R = build_1d_mesh(mt[1:], xyz, dr[ctm], cvol, ge[ctm], areas[ctm])
                    ec, ev, ca, dx, cc, id_dict, vert_dict = R
                    CM[mr][cid][ctm].update(
                        {
                            "GEOM": ge[ctm],
                            "MESH": mt[0],
                            "DX": dx,
                            "NE": len(dx),
                            "VOL": ev,
                            "eXYZ": ec,
                            "DIR": dr[ctm],
                            "cAREA": ca,
                            "cXYZ": cc,
                            "idDict": id_dict,
                            "vertDict": vert_dict,
                        }
                    )

                elif mt[0] == "FRACTURE":
                    # if this is the "first" continuum, re-use the continuum
                    # provided as the seed. If this is a later fracture continuum,
                    # we need to make a copy of the first continuum to re-use
                    # (only in the region of interest)

                    if Q[mr]["CTV"]:
                        # CONSTRAIN_TOTAL_VOLUME
                        # reduce volume of primary continuum
                        # so all the continua together equal initial volume
                        m["VOL"][cid] *= Q[mr]["VF"][ctm]
                    CM[mr][cid][ctm]["MESH"] = "FRACTURE"
                    CM[mr][cid][ctm]["XYZ"] = xyz[:3]
                    CM[mr][cid][ctm]["dXdYdZ"] = xyz[3:]
                    CM[mr][cid][ctm]["NE"] = 1

                CM[mr][cid][ctm]["GIDS"] = []

                if not (mt[0] == "FRACTURE"):
                    # save local matrix continum into global data structures
                    mat_ids = np.arange(ne[ctm])
                    NN = len(m["VOL"])  # 0 to NN-1

                    for LMID, EC, EV, CA, DX, CC in zip(mat_ids, ec, ev, ca, dx, cc):
                        GID = LMID + NN
                        region_ids[ctm].append(GID)
                        CM[mr][cid][ctm]["GIDS"].append(GID)

                        if VERBOSE > 5:
                            print(
                                f"rooted={cid}, c={ctm}, global={GID}, total={NN}, local={LMID}"
                            )

                        m["VOL"].append(EV)  # element volume
                        # i,j,k don't make sense here, save as -1
                        m["IDX"].append((GID, -1, -1, -1))
                        # 1D in x, 1 in other dirs?
                        m["XYZc"].append((*EC, DX, 1.0, 1.0))
```

```
                        if LMID == 0:
                            # connection back to intial domain
                            idx_down = cid
                        else:
                            idx_down = GID - 1

                        # connections
                        m["CON"].append((GID, idx_down, *CC, CA, ctm))

                    # material for ELEMENTS and VERTICES (.uge and .h5)
                    nverts = len(m["Vxyz"])
                    id_keys = sorted(id_dict.keys())

                    for LMID in id_keys:
                        GID = LMID + NN
                        # map onto global numbering
                        m["Vidx"][GID] = [i + nverts for i in id_dict[LMID]]

                    for VID in vert_dict.keys():
                        GID = VID + nverts
                        # vertices not numbered
                        m["Vxyz"][GID] = vert_dict[VID]
                else:
                    # FRACTURE
                    # add a region list for fractures too
                    region_ids[ctm].append(cid)

        # if dual-permeability, need to add connections between matrix nodes
        for ctm in range(1, Q[mr]["N"]):
            # 0th porosity is assumed to have connectivity (already accounted for)
            if Q[mr]["CONN"][ctm]:
                # cycle through initial cell ids (before adding in other porosities)
                for cid in initial_reg_cell_ids:
                    # apply similar connectivity from physical elements
                    # to matrix elements that have "double permeability" turned on

                    XYZ_root = m["XYZc"][cid][0:3]

                    neighbors = {
                        x[1]: x
                        for x in m["CON"]
                        if x[6] == 0 and x[0] == cid and x[1] in initial_reg_cell_ids
                    }

                    for el in range(CM[mr][cid][ctm]["NE"]):
                        GMID_here = CM[mr][cid][ctm]["GIDS"][el]
                        for nb in neighbors.keys():
                            GMID_next = CM[mr][nb][ctm]["GIDS"][el]
                            XYZ_here = m["XYZc"][GMID_here][0:3]
                            XYZ_next = m["XYZc"][GMID_next][0:3]
                            # put center of connection at location of
                            # root connection, with same offset as matrix
                            offset = tuple(
                                mv - rv for (mv, rv) in zip(XYZ_here, XYZ_root)
                            )
                            rf_XYZ = neighbors[nb][2:5]
                            face_A = neighbors[nb][5]  # same area as in root connection
                            dp_face_XYZ = tuple(
                                rf + off for (rf, off) in zip(rf_XYZ, offset)
                            )
                            m["CON"].append(
                                (GMID_here, GMID_next, *dp_face_XYZ, face_A, ctm)
                            )

        cumulative_ids = []
        # iterate over regions used in PINC block
        for ctm, reg_ids in enumerate(region_ids):
            txt_fh = open(f"{base_fn}-region-mc-PINC-{ctm}-{mr}.txt", "w")
            # list of element IDs associated with a multicontinuum region
            for rid in reg_ids:
                RID = rid + 1  # make them plfotran-convention 1-based
                txt_fh.write(f"{RID}\n")
                cumulative_ids.append(RID)
            txt_fh.close()
        # write a text file that includes every continuum that is associated with the
        # region in the initial domain
        txt_fh = open(f"{base_fn}-region-mc-ALLCONTINUA-{mr}.txt", "w")
```

```
        txt_fh.write("\n".join([f"{idx}" for idx in cumulative_ids]) + "\n")
        txt_fh.close()

        # write list of IDs for other regions too, which fall inside PINC block
        # useful for MATERIAL_PROPERTIES, SOURCE_SINK, or OBSERVATIONs.
        for bn in r.keys():
            if not bn == mr:
                # doesn't make sense for areas, which are connections
                # flux BC would need to be handled differently
                if r[bn]["DIM"] == "POINT" or r[bn]["DIM"] == "VOLUME":
                    cumulative_ids = []
                    ctm_fhs = []
                    for Z, idx in enumerate(r[bn]["REG_CELL_IDS"]):
                        IDX = idx - 1
                        # only look in "base" porosity [0], where region was initially defined
                        if IDX in region_ids[0]:
                            for ctm in range(Q[mr]["N"]):
                                if Z == 0:
                                    # first time through open up filehandles
                                    ctm_fhs.append(
                                        open(
                                            f"{base_fn}-region-mc-{ctm}-{bn}-OTHER-{mr}.txt",
                                            "w",
                                        )
                                    )

                                if ctm == 0:
                                    RIDs = [idx]
                                else:
                                    RIDs = [i + 1 for i in CM[mr][IDX][ctm]["GIDS"]]
                                ctm_fhs[ctm].write("\n".join([f"{i}" for i in RIDs]) + "\n")
                                cumulative_ids.extend(RIDs)
                    for fh in ctm_fhs:
                        fh.close()  # at end close filehandles
                    txt_fh = open(
                        f"{base_fn}-region-mc-ALLCONTINUA-{bn}-OTHER-{mr}.txt", "w"
                    )
                    txt_fh.write("\n".join([f"{idx}" for idx in cumulative_ids]) + "\n")
                    txt_fh.close()

if VERBOSE > 3:
    print("CM debug")
    for cm in CM.keys():
        print(f"PINC REGION: ******{cm}*****")
        for k in CM[cm].keys():
            print(f"=root cell id {k}")
            for c in CM[cm][k].keys():
                if c == "TOTAL_VOL":
                    print(f"==root element total volume: {CM[cm][k][c]}")
                else:
                    print(f"---continuum {c}")
                    for i in CM[cm][k][c].keys():
                        print(f"    {i} :: {CM[cm][k][c][i]}")


# re-use mesh writing function, now with all multicontinuum elements
# (from all the possible regions with PINC blocks
mefn, mdfn = write_uge_file(f"{base_fn}-mc", m)
```

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	01977	sanddocs@sandia.gov

This page left blank

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.