Editors: Jeffrey Carver, carver@cs.ua.edu, and Damian Rouson, damian@rouson.net

OpenPNM: A Pore Network Modeling Package

Jeff Gostick and Mahmoudreza Aghighi | McGill University
James Hinebaugh | University of Toronto
Tom Tranter | University of Leeds
Michael A. Hoeh | Forschungszentrum Jülich
Harold Day and Brennan Spellacy | McGill University
Mostafa H. Sharqawy | University of Toronto and University of Guelph
Aimy Bazylak | University of Toronto
Alan Burns | University of Leeds
Werner Lehnert | Forschungszentrum Jülich and RWTH Aachen
Andreas Putz | Automotive Fuel Cell Cooperation

Pore network modeling (PNM) is a well-established and long-standing approach for simulating transport in porous materials.[1–3] PNMs represent an alternative to the more widely used continuum modeling in which a porous material is treated as a volume-averaged continuum without resolving microscale features. Continuum models are mathematically rigorous, but they have some practical limitations. First, each phenomenon being modeled requires experimentally measured constitutive relationships that describe the media's macroscopic transport properties, such as the permeability coefficient or effective diffusivity. These can be challenging to measure, especially for multiphase flow conditions. Second, treating the porous medium as a volume-averaged continuum means that discrete pore-

 1521-9615/16/$33.00 © 2016 IEEE Copublished by the IEEE CS and the AIP

scale phenomena and events aren't resolved, so only the average amount of a fluid phase in each computational node is known. Moreover, the distribution of phases within the continuum isn't often well predicted by such models, which rely on simple extensions to Darcy's law for multiphase flow, rather than more rigorous formulations.[4] Finally, it sometimes isn't appropriate to invoke the volume-averaged approximation, particularly in thin materials such as membranes and electrodes.[5]

PNMs solve these issues but sacrifice some mathematical rigor. Instead of solving *n*th-order partial differential equations (PDEs), the pore space is treated as a network of "pipes." Transport inside the network is modeled using finite difference schemes to solve 1D analytical solutions of the relevant transport equations. Despite this simplification, PNMs successfully and efficiently predict numerous aspects of multiphase transport.[6] The sizes and connectivity of the pores and throats are chosen to match the known physical structure, meaning that the interplay between the structure and flow characteristics is included. Structural properties of the porous material can be readily obtained from various imaging techniques[7,8] or computer-generated structures.[9–11] In the most basic case, pore and throat sizes are adjusted arbitrarily to allow the model to reproduce known experimental properties.[12] PNMs are naturally geared toward percolation calculations,[13] so they simulate realistic fluid invasion processes with computational ease.[14] This allows for truly pore-scale descriptions of the fluid distribution within the media, which has major impacts on almost all other transport processes. By simply setting pores and throats filled with one phase as closed to other phases, PNMs can predict constitutive relationships for experimentally inaccessible multiphase parameters.[15–17] Extensive reviews of pore network modeling and comparisons between the approaches are available in the literature.[18–21]

This article provides an overview of the OpenPNM package for pore network simulations. Currently, the only commercially viable software product is PoreXpert, a program arising from a research group at Plymouth University (formerly known as Pore-Cor[22]), and groups sometimes publish overviews of their in-house code.[23] To the best of our knowledge, no other projects are under way to produce an open source PNM framework, in stark contrast to the field of computational fluid dynamics, where a multitude of powerful commercial and open source options[24,25] exist. Generally, researchers in the PNM community develop their own code to be used internally by their research group. This requires considerable resources and investment, and existing code might not be optimized for speed, modularity, extensibility, or maintainability; it's almost never well documented for future users. These problems are all too common, which were key motivators behind OpenPNM's development. The aim is to provide the porous media community with a general, powerful, and flexible framework for tackling all manner of PNM problems from the same code base. It is hoped this will enable the sharing of code between researchers, provide a common baseline for comparing models, and allow us all to build on the work of others, thereby dramatically increasing the pace of research and discovery.

OpenPNM was designed with three overall objectives in mind: accessibility to a wide audience, generality to as many applications as possible, and extensibility to simulate any type of physical process. The first and foremost principle is to ensure that the code is accessible, in both the conceptual and physical sense. OpenPNM was coded in Python, which is a powerful, easy-to-use, and free object-oriented programming (OOP) language that's increasingly used in engineering calculations.[26,27] Great effort was expended creating detailed documentation, and the OpenPNM website (openpnm.org) has a continuously growing list of tutorials and examples. The code is hosted publicly and is completely open source and free. It's registered with the Python package index (PyPI) so it can be installed with a simple `pip install openpnm` command. OpenPNM requires no compilation of source code during installation (it's written entirely in Python, which is an interpreted scripting language). OpenPNM maintains good computational performance by relying heavily on NumPy[28] and SciPy (www.scipy.org) to perform numerical operations, both of which employ precompiled C code for speed. Another critical aspect of accessibility is to ensure that the code is well tested. At the time of this writing, 85 percent of the 7,000 lines of code in the package are tested regularly via an extensive suite of unit and integration tests. The second overall principle is generality, so the package is fully agnostic to network topology, shape, and dimension, allowing traditional cubic lattices and fully random networks to be treated identically. This is accomplished using approaches borrowed from graph theory such as adjacency and incidence matrices to store network topology and architecture. OpenPNM includes a large suite of tools for working with network topology, such as find-

ing pores connected to given pores, and for labeling pores and throats for quick access later. Finally, the package is designed to be easily customized by researchers wishing to apply their own pore-scale models, for example to calculate pore-wall surface areas in some special way. Many common pore-scale models are included with the package, but extensibility and customization were a primary consideration. Adding custom models is straightforward and requires only basic function definitions that contain blocks of procedural-style code, along with an understanding of how OpenPNM stores data.

This article doesn't provide exhaustive details about each method or function; the code itself is heavily documented in that regard, so the interested reader is directed there for more details. We highly recommend the use of an integrated development environment (IDE) that supports autocomplete and provides an object inspection pane to render context-aware help files for each method (such as Spyder). Some familiarity with Python and a basic knowledge of OOP would also be helpful, such as knowing definitions of a method and class.

Data Storage

OpenPNM stores all data, such as pore diameters, in NumPy `ndarrays`,[28] which have become the de facto standard numerical array data type in Python. They support slicing, fancy indexing, broadcasting, vectorization, and all the typically expected array operations. This approach was chosen over a more object-oriented option such as that used by NetworkX[29] because these operations are very fast when vectorized. OpenPNM also relies heavily on SciPy, which is designed to work specifically on NumPy arrays.

Pore and Throat Properties

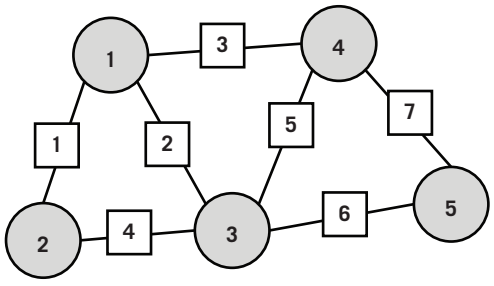
One of OpenPNM's main design considerations was to accommodate all networks of arbitrary dimensionality, connectivity, shape, and so on. To accomplish this, OpenPNM stores all pore and throat data in the most generic way possible: as lists (arrays) of either N_P or N_T length, corresponding to the number of pores and throats in the network, respectively. This means that each pore (or throat) has an index, and all properties for that pore (or throat) are stored in the array element corresponding to that index. Thus, the diameter for pore 15 is stored in the '`pore.diameter`' array in element 15, and the length of throat 32 is stored in the '`throat.length`' array at element 32. All of the property arrays are stored in a dictionary, which is similar to a structured variable or `struct` in other programming languages. This allows each property array to be accessed by its name or key, with a syntax such as `net['pore.diameter']` or `net['throat.length']`, where `net` is the name of the dictionary object.

Several rules have been imposed to control data integrity. First, all names must begin with either `pore` or `throat`, which serves to identify the type of information stored in the array. Second, for the sake of consistency, only arrays of length N_P or N_T are allowed in the dictionary. Any scalar values written to the dictionaries are cast into full-length vectors, effectively applying the scalar value to all network locations. This simplifies subsequent numerical steps because all arrays are of equal and known length. The drawback of this approach is that storing data isn't as memory-efficient as possible, but this isn't as important on modern computers, which typically have several Gbytes of RAM. In addition, there's no limitation on the size of other dimensions, meaning that $N_P \times 3$ or $N_T \times 2$ arrays are allowed, such as the '`pore.coords`' array, which stores each pore's `[X,Y,Z]` coordinates in an $N_P \times 3$ list.

There are also no limitations on the data type that can be stored, but OpenPNM does distinguish between Boolean arrays and all other array types. The Boolean arrays are treated as *labels*, while all other arrays are assumed to contain numerical data describing pore or throat *properties*. Labels enable easy retrieval of a list of important pores (or throats), such as all pores on the top face of the network. Several labels are added to the `Network` object during the generation steps, but users can apply their own labels as needed. For instance, you might perform a complex filtering to find all pores within a certain distance of some location that also possess a volume within some range. To avoid repeating this query, simply apply a label to the pores. Labels are applied by adding a new array to the dictionary with the label name, containing `True` values for the pore (or throat) locations where the label applies. Thus, the array `net['pore.top']` is set to `True` for every pore at the top of the `Network` object and `False` elsewhere.

Network Topology

The only topology definitions required by OpenPNM are that each throat connects exactly two pores, no more and no less, and that throats are nondirectional, meaning that transport in either direction is equal. Other general but nonessential rules are that pores can have an arbitrary number of throats (even zero, although pores with zero throats lead to singular matrices



	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	0
3	1	1	0	1	1
4	1	0	1	0	1
5	0	0	1	1	0

	1	2	3	4	5	6	7
1	1	1	1	0	0	0	0
2	1	0	0	1	0	0	0
3	0	1	0	1	1	1	0
4	0	0	1	0	1	0	1
5	0	0	0	0	0	1	1

	i	*j*
1	1	2
2	1	3
3	1	4
4	2	3
5	3	4
6	3	5
7	4	5

Figure 1. Schematic of random network architecture with pore (node) and throat (bond) numbers labeled. Also shown are the adjacency matrix (bottom left) and incidence matrix (bottom middle) representation of this network and the IJV sparse representation of the adjacency matrix (bottom right).

and other problems and thus should be avoided) and two pores should be connected by no more than one throat, unless there's some real physical reason for this, since unintentional duplicate connections impact the rate of species exchange between pores.

One of the challenges when storing networks in list-based arrays is tracking the topology of the pore and throat connections. An important property of a throat is which pores are found on either end, so that `Network` connectivity can be stored as a list of throat properties equivalently to other physical throat properties in an $N_T \times 2$ list of `[pore I, pore J]` pairs. This storage scheme happens to define an adjacency matrix in the sparse storage scheme known as IJV (or COO in the scipy.sparse module), a commonly used means of representing topology in graph theory. This is an $N_P \times N_P$ array with nonzero values, V, at locations (I, J) indicating that pores I and J are connected. It's symmetrical since the throats are bidirectional and highly sparse because a given pore only connects with a small subset of nearby pores in the network. Figure 1 shows a simple `Network` topology along with its corresponding adjacency matrix and its IJV (or COO) representation. Also shown is the incidence matrix for the same topology. Adjacency and incidence matrices are theoretically equivalent means of representing topology; both can be represented in IJV format, but each has different practical advantages.

OpenPNM `Network` objects include numerous methods for querying the topology, such as finding pores connected to a given throat (`find_connected_pores`) or finding the throats neighboring certain pores (`find_neighbor_throats`). Each of these queries is performed by inspecting the adjacency or incidence matrices. For instance, to find all pores that are direct neighbors to pore 5 requires finding which columns on row 5 of the adjacency matrix contain nonzeros. Alternatively, to find all throats that are directly connected to pore 5, it's easier to find all nonzero entries on row 5 of the incidence matrix. Both of these operations are most efficiently performed on sparse matrices stored in the list-of-lists (LIL) format, so OpenPNM stores copies of both in a private location for use in the event of such queries.

Implementation

OpenPNM has five main objects: `Network`, `Geometry`, `Physics`, `Phase`, and `Algorithm`. Each of these inherits from the `Core` class, plus has some additional methods or functionality added for its specific role.

The Core Objects

The `Core` objects in OpenPNM contain data and are used to perform calculations. Each `Core` object is a subclassed version of Python's *dict* or dictionary, with several additional methods added that are specific to handling OpenPNM's data. The main role of the `Core` class is to manage the data stored in the dictionary. This means storing and tracking label and property arrays, implementing the data integrity rules mentioned above, returning lists of pores and throats based on some combination of labels, and so forth.

Network. A `Network` object represents a fully self-contained topological entity, meaning that when two separate `Network` objects are created, they don't interact with each other. If two separate `Networks` need to exchange information, then they must be stitched together to form a single `Network`, using the provided topology manipulation tools. At minimum, a `Network` needs pore coordinates and the throat connections to define its topology.

The `GenericNetwork` class has a number of additional methods added for performing topological queries, such as finding the pores directly connected to a given pore (`find_neighbor_pores`), finding the throats that connect given pairs of pores (`find_connecting_throat`), and many others.

The `GenericNetwork` class itself isn't responsible for creating network topologies. For this, there are several subclasses of the `GenericNetwork` class, such as `Cubic`, which creates the standard lattice with specified connectivity patterns (6, 8, 26, and so on), and `Delaunay` which uses a Delaunay tessellation to connect random points in space. In addition to generating topologies, it's also possible to import networks from external sources, and several formats are supported.

Geometry. `Geometry` objects track and manage the physical properties and dimensions of pores and throats. OpenPNM was designed to allow networks to include multiple regions with differing properties for modeling multilayered, stratified, or generally heterogeneous media. In these cases, multiple `Geometry` objects can be created and assigned to different subsets of pores and throats. The difference between separate `Geometry` objects lies in the unique set of pore-scale models that were applied to calculate the geometrical properties.

Several subclassed versions of `GenericGeometry` are included in the package for convenience, including the standard `Stick_and_Ball`, as well as `Voronoi`, which is combined with the `Delaunay Network` class to model fibrous materials.[30,31] Most frequently, however, users define their own custom `Geometry` classes, which consist of an assortment of pore-scale models with suitable parameters.

Phase. `Phase` objects manage the thermophysical properties of the solids, liquids, and gases that exist in the `Network`. Because fluids can move around during the course of a simulation, via invasion percolation for instance, a `Phase` object is defined everywhere in the `Network`, and the actual presence of a phase in a given location is tracked using an *occupancy* list, a number between 0 and 1 that indicates fractional filling.

Because thermophysical properties are generally dependent on each other (viscosity is a function of temperature), consideration was made to regenerate property values as conditions change. For example, if the temperature of a `Phase` changes, all temperature-dependent properties can be recalculated by calling the `regenerate` method of the `Phase` object.

OpenPNM includes predefined `Phase` subclasses for `Air`, `Water`, and `Mercury`. Creating new fluids requires only defining a new subclass of `GenericPhase` and assigning the suitable models, either from the `Phase` models library or custom written models, with the appropriate parameters.

Physics. The combination of pore-scale geometry and thermophysical properties are what dictate the actual transport parameters in the pore network. For instance, fluid viscosity and throat diameter are both required to calculate the hydraulic conductance according to the Hagen-Poiseuille model. When a `Physics` object is created, it must be told which `Phase` and `Geometry` objects it applies to. This allows the models attached to a `Physics` object to find the necessary thermophysical and geometrical properties.

The only subclass included with the package is `Standard`, which contains an assortment of commonly used pore-scale physics models such as the Hagen-Poiseuille model for hydraulic conductance and the Washburn equation for capillary entry pressure. Creating custom pore-scale `Physics` models is one of the key ways PNMs differentiate themselves, so creating and adding custom models

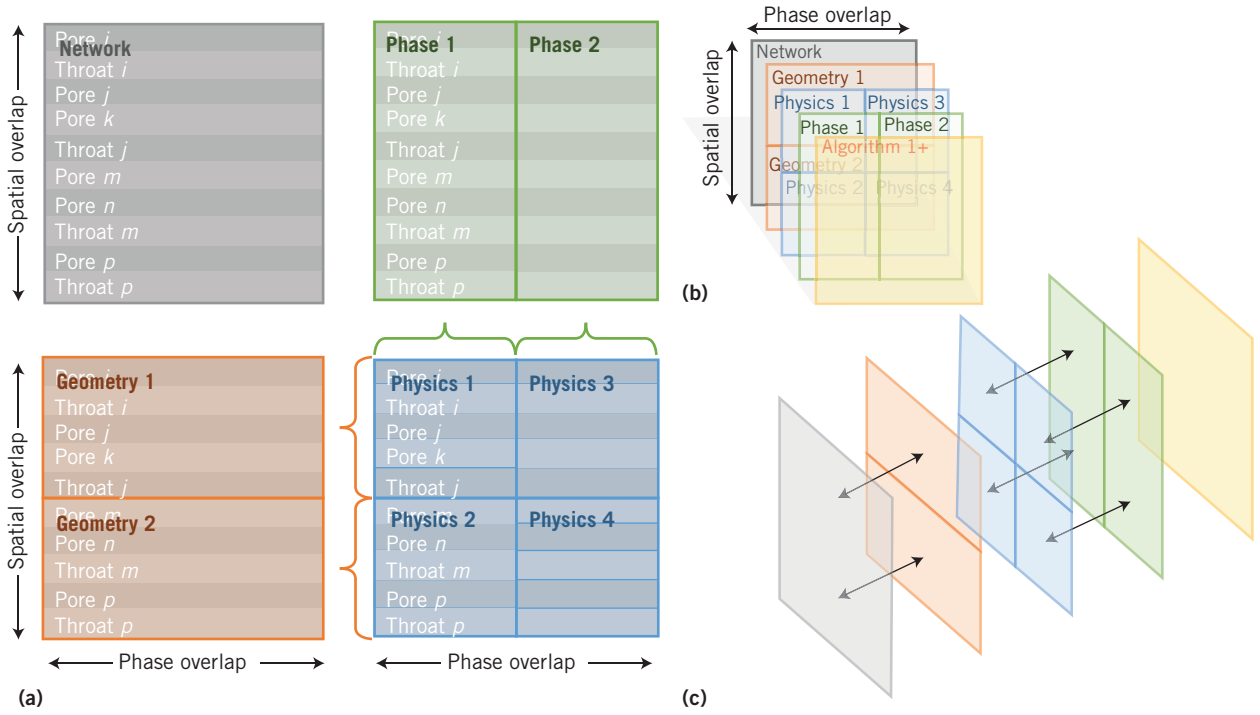


Figure 2. Schematic representation of `Core` object relationships. (a) Overlap in the vertical direction indicates that objects are associated with the same pores (or throats), while overlap in the horizontal direction indicates with which `Phases` each object is associated. `Phases` by definition don't overlap with each other but span all pores and throats, `Geometry` objects span all `Phases` but only a limited set of pores and throats, and `Physics` objects exist at the intersection of `Phase` and `Geometry` objects. (b) The objects form a conceptual stack with each object representing a layer, creating the physical interpretation of overlaps shown in (a). (c) The layers that are physically near each other in the stack can exchange data (read-only) which offers a convenient way to retrieve a complete list of data that may be dispersed across several objects (Geometry 1 and Geometry 2).

was designed to be as flexible as possible. A custom `Physics` class can be created by choosing or coding the necessary pore-scale models, then assigning them to a `GenericPhysics` object.

Algorithm. `Algorithm` objects also derive from the `Core` class, and they too store their own data, which is typically the result of some algorithm or calculation. Algorithms have numerous additional methods beyond those supplied by `Core`, and these methods can be quite complex, depending on the `Algorithm`.

The results of any calculation are stored on the `Algorithm` object to prevent overwriting or interfering with data on other objects. For instance, when a diffusion calculation determines the concentration of a species in each pore, the resulting array is stored under '`pore.mole_fraction`' on the `Algorithm`, even though mole fraction is technically a `Phase` property. This stored data can be used in subsequent calculations, but it must be explicitly transferred to the new `Algorithm` object.

Object Relationships

A given simulation consists of only one `Network` object and one or more each of `Geometry`, `Phase`, and `Physics` objects. It's useful to think about these various objects in terms of layers that stack together for a typical simulation. The `Network` contains all the pores and throats for a particular simulation and forms the base layer. One of the main features of OpenPNM is the ability to model heterogeneous materials, for instance, where different pore size distributions are to be applied to different regions. This is accomplished by defining multiple `Geometry` objects that each apply to a separate group of pores (or throats) as shown in Figure 2. Spatial overlap of `Geometry` objects is forbidden because it creates a conflict over which `Geometry` object should calculate and store information for those locations. Next, a layer of two `Phase` objects is added to the stack to allow multiphase simulations. `Phases` span all pores and throats but are immiscible and so are represented side by side. In fact, multiple Phases can exist in a given pore simultaneously. Then, a layer of four `Physics` objects are added between the

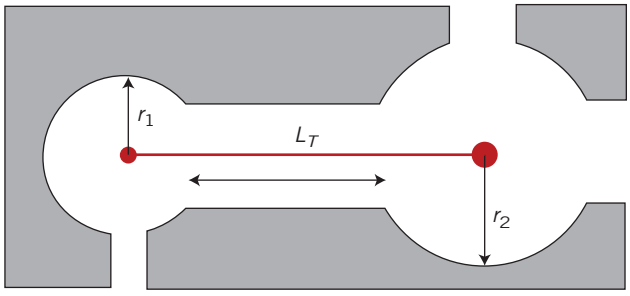


Figure 3. Schematic diagram showing the definitions and dimension of a typical pore-throat-pore conduit.

`Geometry` and `Phase` layers. `Physics` objects combine geometrical information about pores (or throats) with the properties of the fluids in that pore (or throat). Thus, `Physics` objects require information from one `Geometry` and one `Phase` object, and therefore exist at each *intersection* of `Geometry` and `Phase` objects as shown in Figure 2b. Put another way, a `Physics` object only applies to one phase because the thermophysical properties of the `Phases` are different and change the behavior of pore-scale `Physics`. Moreover, a different `Physics` is required for each `Geometry` since different geometrical properties may result in different pore-scale behavior.

Once a simulation has been set up as shown in Figure 2b, it's ready for calculations. An arbitrary number of `Algorithm` objects can be created and added as layers to the stack, with each `Algorithm` looking up the information it requires from other objects in the simulation and producing results, such as changing the occupancy of the phases in various pores due to percolation, or calculating mass fractions.

One drawback of having multiple `Geometry` and `Physics` objects for different regions of the `Network` is that a single list containing all property values for the entire network isn't readily available. OpenPNM addresses this issue by allowing `Network` and `Phase` objects (which by definition encompass all pores and throats) to retrieve and combine data from the `Geometry` and `Physics` objects, respectively. The data exchange between the various layers is indicated by the arrows in Figure 2c, conveying that associated objects are able to read data from each other. Writing data between layers isn't allowed.

Pore-Scale Models

Pore-scale models are the most important aspect of OpenPNM as they elevate the basic topological graph to the level of a pore network model by giving physical meaning to the pores and throats. Models are also the main way that users can customize OpenPNM to suit their particular scientific endeavors. Before delving into the code's relevant machinery, it's better to first discuss the meaning of *models*.

The main difference between PNMs and continuum models is how they treat the transport properties between two physical locations in the domain. For a specific example, consider viscous flow. In continuum modeling, the flow rate or pressure drop between two neighboring locations is dictated by the medium's permeability coefficient, which is typically measured in the lab on a sample of representative material. In a PNM, the two neighboring locations are treated as actual pores and connected by a throat. The flow rate between these two pores is treated as flow through a pipe, which can be described by any number of analytical solutions depending on the geometry assigned to the pipe.[32] One typical approach is to use the Hagen-Poiseuille equation for single phase flow in a cylindrical tube:

$$q = \frac{\pi R_{i-j}^4 (P_i - P_j)}{8\mu\ L_{i-j}}, \tag{1}$$

where P_i and P_j are the pressures in pores i and j, L_{i-j} and R_{i-j} are the length and radius of the throat (pipe) connecting pores i and j, and μ is the fluid viscosity. Figure 3 shows the dimensions and geometry of this pore-throat-pore conduit. If pressure loss in each half-pore is neglected for simplicity, then the total flow rate given by Equation 1 can be generalized as

$$Q = g_{i-j}(P_i - P_j), \tag{2}$$

where g_{i-j} is the conduit's hydraulic conductance.

Therefore, a hydraulic conductance model would return values of g given throat radii R and length L from the `Geometry` object and viscosity μ from the `Phase` object. Using the vectorization capabilities of NumPy, this can be done in a few lines:

```
def   hydraulic_conductance(geometry, physics):
      mu = physics.interpolate_
      data(physics['pore.viscosity'])
      L = geometry['throat.length']
      R = geometry['throat.radius']
      g = 3.14159*(R**4(/(8*mu*L)
      return g
```

A few key points are illustrated here. First, viscosity is a `Phase` property, yet it's accessed via `physics`. This utilizes the data exchange rules outlined earlier. Sec-

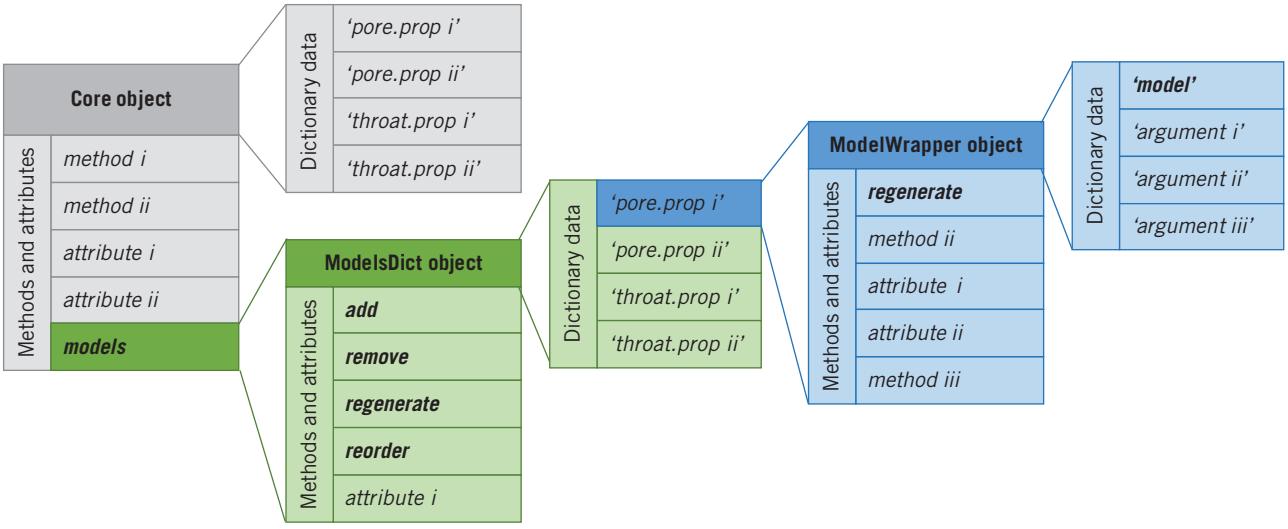


Figure 4. Schematic diagram showing how pore-scale models are associated with *Core* objects. When the regenerate method of the *ModelsDict* is called, it calls the *run* method of each *ModelWrapper* in the order in which they're stored. The values returned from the model are placed into the *Core* object's dictionary under the same property name as that of the stored model.

ond, like most Phase properties, viscosity is defined in pores so it must be interpolated to find throat values. Third, the array g is N_T long because it's the result of element-wise operations between 'throat.radius' and 'throat.length'. Finally, the pore and throat properties used weren't passed in as hardcoded numerical values; instead, the objects containing the values were passed, and the values were retrieved "on demand." This means that if the viscosity is changed on the Phase object, then rerunning the above code will automatically utilize the updated values without any effort on the user's part. This is the mechanism by which changing conditions are transmitted to all other dependent properties.

The above code snippet is a simple but representative example of a pore-scale model. It's expected that users will devise their own such models of arbitrary complexity. To utilize any custom-made pore-scale models, a user only needs to create a file in the working directory (such as "my_models.py"), populate it with his or her own function definitions, and import the file by entering `import my_models`. This will provide access to all models in the file with `my_models.hydraulic_conductance`.

Assigning Models to Core Objects

To ensure that physical properties can be recalculated as needed, it's necessary to save the pore-scale model and all parameters in memory. To accomplish this, every Core object has an attribute (that is, physics.models) that actually contains a nested dictionary called the ModelsDict as shown in Figure 4.

ModelsDict has an add method that performs the service of associating the model with the Core object. The arguments required by the add method are the name of the pore or throat property (propname) where the values generated by the model are to be stored ('throat.hydraulic_conductance'), a handle to the model to be used (my_models.hydraulic_conductance), and any parameters required by the specific model. The add method stores all the received parameters in the ModelsDict using the given propname as a key. Also shown in Figure 4 is the ModelWrapper object, which houses each model's specific arguments.

A few nuances must be considered when dealing with models. First, models are stored in ModelsDict under a specified propname; by default, the values produced when the model is run will be stored in the parent Core object dictionary under the same propname. Second, the numerical values produced by a model remain in the Core object dictionary as constant values until regenerate is called to rerun each model.

Algorithms

Numerous key algorithms are included with OpenPNM, including various percolation algorithms (Drainage and InvasionPercolation), and several linear transport models such as FickianDiffusion, StokesFlow, OhmicConduction, and FourierConduction.

Each algorithm is slightly different, but in general, Algorithms are instantiated by passing in a Network object and some additional arguments. Algorithm objects have a setup method that allows specifying

various parameters required by the algorithm. The calculation is executed by calling the `run` method. Finally, the results are stored on the `Algorithm` object but can be transferred to other `Core` objects (usually a `Phase`) for use in further calculations.

Percolation and Invasion

Performing multiphase transport calculations in the pore network is a central role of PNMs. Realistic pore-scale fluid configurations are simulated using percolation theory to determine how an invading phase will displace a defending phase. OpenPNM contains several algorithms for performing such calculations.

Drainage. In a porosimetry experiment,[33] the volume of a non-wetting fluid injected into a specimen is tracked at discrete pressure steps. The process is referred to as *drainage* and is mathematically simulated as an access-limited bond percolation problem. Access limitations are important because the invading phase can only invade throats that are accessible from the sample surface and subsequently those connected directly to the reservoir of the invading phase. Simulating porosimetry experiments is an essential part of PNMs because the results can be compared to experimental data to verify that correct pore and throat size distributions have been used when combined with other information such as the permeability and porosity of the material.

To conduct this simulation in OpenPNM, throats must first be assigned capillary entry pressures, indicating the pressure that must be applied to the invading phase for it to enter that throat. Relating throat invasion pressure to geometric throat properties is almost universally done with the Washburn equation, but other, more suitable options are available.[34] The Drainage algorithm uses the `connected_components` method included in the `csgraph` module of scipy.sparse to perform a standard graph theory clustering operation over the `Network`.

Invasion percolation. Invasion percolation differs from drainage in subtle but important ways. In drainage, all accessible throats with entry pressures lower than a certain value are simultaneously invaded (along with their neighboring pores). Invasion percolation applies a similar logic on the scale of single throats by invading only the single most easily invaded accessible throat and its neighboring pore on each step. In physical terms, this is equivalent to a quasi-static, rate-controlled injection experiment.[35]

During an invasion percolation simulation, when a pore is invaded, new throats become accessible and join the invasion front. The algorithm must choose the throat with the lowest entry pressure for the next invasion, requiring a continually maintained, dynamically changing, and sorted list of throat entry pressures.[36] A standard graph theory algorithm for this process isn't available in SciPy, so a basic algorithm was implemented in OpenPNM using Python's built-in `heapq` module, which is based on priority queues using a heap data structure. In simple terms, this means that a list of initially accessible throats is sorted into a heap, which has the property that the smallest value is always found in element 0. This allows instant access to the next throat that should be invaded. The pore attached to this throat is then invaded, and all of its throats are added to the heap, and the procedure is repeated until all pores and throats are invaded.

Resistor Network Calculations

One of the main uses of pore network models is to simulate transport phenomena through the pore space, usually in the presence of a second phase. A typical example is the diffusion of a gaseous species through a pore space that's partially filled with a liquid. To model a domain of any useful size, it's difficult to model such a scenario by discretizing the pore space as a finite-element mesh. First, it would take many nodes to model even a few pores. Second, the placement of liquid is a complex task, requiring the solution of high-order PDEs[37] or a Lattice-Boltzmann approach,[38] both of which are highly computationally intensive. An alternative to these methods is the Full Morphology approach based on image analysis,[39] which can analyze a 2D or 3D pixel image of a porous material and place phases using structuring elements. However, the user is still reliant on other methods to solve the equations of fluid flow and also limited to much smaller domains.

Pore network modeling starts by recognizing that, in many applications, an approximate model of a suitably large domain is more useful than a highly rigorous model of a limited number of pores. With this in mind, each throat represents a resistor in the network through which the species of interest must travel to reach the neighboring pore. The resistance to diffusion or flow offered by a throat constriction is a function of its geometry, as well as the fluid properties such as viscosity or diffusion coefficient. In PNMs, this resistance is described by an appropriate pore-scale physics model, such as the Hagen-Poiseuille model given in Equation 1. This and other pore-scale physics models are cast in

terms of a resistance by analogy with Ohm's law. In the case of Equation 1, pressure (P) is the driving force, $\mu R^4/8L$ represents the conductance to flow due to the geometrical properties of the throat, μ provides the resistance to flow caused by the flowing fluid's viscosity, and Q is the volumetric flow of fluid analogous to current in Ohm's law. This can be recast as

$$Q_{i-j} = \frac{\pi R_{i-j}^4}{8\mu\ L}(P_i - P_j) = g_{i-j}(X_i - X_j), \quad (3)$$

where g_{i-j} is the conductance between pores i and j, and X is the unknown to be solved for. At steady state and in the absence of any source or sink terms, the net material flow through pore i is zero, hence

$$0 = \sum_{i-j}^{n} g_{i-j}(X_i - X_j), \quad (4)$$

where pore i has n neighbors. Applying this equation to each pore in the network results in a system of linear equations in X that can be readily solved by any matrix inversion algorithm subject to given boundary conditions. Importantly, the conductance g_{i-j} can be set to very small values for pores blocked by another phase, making it trivial to incorporate the impact of multiple phases on transport processes.

Performing transport calculations in OpenPNM begins with instantiating an `Algorithm` object, then specifying the necessary boundary conditions using the `apply_boundary_conditions` method of the `Algorithm`. The `Algorithm`'s `run` method is then called to handle the processes of building the coefficient matrix, applying specified boundary conditions, and calling the matrix inversion routine. The main requirement from the user is to ensure that the conductance values for each throat are calculated correctly, meaning that the appropriate models for geometrical sizes, thermophysical properties, and pore-scale physics have been applied.

Application and Demonstration

The intense development of OpenPNM over the past few years has resulted in a concise and powerful framework that can perform significant computations in minimal lines of code. This final section will describe the steps required to calculate the effective diffusivity of a porous material as a function of liquid water saturation, a typical application of PNMs.[16] The full script is given in the sidebar.

The first step is to create a `Network` object, in this case with 3,125 pores on a cubic grid with a spacing of 100 μm between them:

```
pn = OpenPNM.Network.Cubic(shape=[25, 25,
     5], spacing=0.0001)
```

Next, a `Geometry` object must be instantiated:

```
geo = OpenPNM.Geometry.
      GenericGeometry(network=pn,
      pores=pn.pores(),
            throats=pn.throats())
```

In this step, the `Geometry` object is associated with the `Network` (`pn`) and assigned to all the pores and throats. The `GenericGeometry` class has no predefined pore-scale models, so these must be added. The script in the sidebar illustrates how to add geometrical properties such as `'pore.diameter'` and `'throat.length'` models to the `geo` object, as well as direct assignment of calculated values such as `'pore.volume'`. Some subclasses included with OpenPNM have predefined pore-scale geometry models; these can be used as templates for users to create one for their own specific materials.

Next, `Phases` are added using the predefined subclasses for `Air` and `Water`:

```
air = OpenPNM.Phases.Air(network=pn)
water = OpenPNM.Phases.Water(network=pn)
```

Only the `Network` object with which these `Phase` objects are to be associated is required as an argument.

This simulation will require first invading liquid water into the network, then diffusing gas through dry void space. The `Phases` will each require their own `Physics` objects:

```
phys_air = OpenPNM.Physics.
           GenericPhysics(network=pn,
     phase=air,
     geometry=geo)
phys_water = OpenPNM.Physics.
             GenericPhysics(network=pn,
     phase=water,
     geometry=geo)
```

`Physics` objects require the `Network`, a `Phase`, and a `Geometry` object as arguments. Both the `Physics` objects above operate on the same pores and throats (defined by the `geo` object), but each apply to a different `Phase`. Both of the above objects are instances of the `GenericPhysics` class, which has no pore-scale models. The sidebar illustrates how to add `'throat.capillary_pressure'` to the water

Sample Script

The following script shows how to set up a `Network`, define a `Geometry` by adding the necessary models, create two predefined `Phase` objects, define the necessary `Physics` objects, and finally perform sequential simulations using two `Algorithm` objects:

```
import OpenPNM
import scipy as sp
import OpenPNM.Geometry.models as gm
import OpenPNM.Physics.models as pm
pn = OpenPNM.Network.Cubic(shape=[10, 10, 10], spacing=0.0001)
geom = OpenPNM.Geometry.GenericGeometry(network=pn,
          pores=pn.pores(),
          throats=pn.throats())
geom['pore.seed'] = sp.rand(geom.num_pores())
geom.models.add(propname='pore.diameter',
    model=gm.pore_diameter.weibull,
    shape=2.77,
    loc=6.9e-7,
    scale=9.8e-6)
geom.models.add(propname='throat.diameter',
    model=gm.throat_misc.minpore,
    pore_prop='pore.diameter')
geom.models.add(propname='throat.length',
    model=gm.throat_length.straight)
geom['pore.area'] = 3.14159/4*geom['pore.diameter']**2
geom['pore.volume'] = 4/3*3.14159*(geom['pore.diameter']/2)**3
geom['throat.area'] = 3.14159/4*geom['throat.diameter']**2
geom['throat.volume'] = 3.14159/4*geom['throat.diameter']**2* geom['throat.length']
air = OpenPNM.Phases.Air(network=pn)
water = OpenPNM.Phases.Water(network=pn)
water['pore.contact_angle'] = 110.0
water['pore.surface_tension'] = 0.072
phys_air = OpenPNM.Physics.GenericPhysics(network=pn,
           phase=air,
           geometry=geom)
phys_water = OpenPNM.Physics.GenericPhysics(network=pn,
           phase=water,
           geometry=geom)
phys_air.models.add(propname='throat.diffusive_conductance',
     model=pm.diffusive_conductance.bulk_diffusion)
phys_water.models.add(propname='throat.capillary_pressure',
      model=pm.capillary_pressure.washburn)
OP = OpenPNM.Algorithms.OrdinaryPercolation(network=pn,
           invading_phase=water)
OP.run(inlets=pn.pores('bottom'))
phys_air['throat.conductance'] = phys_air['throat.diffusive_conductance']*(OP['throat.inv_Pc'] > 8000)
FD = OpenPNM.Algorithms.FickianDiffusion(network=pn,
           phase=air)
FD.set_boundary_conditions(pores=pn.pores('top'),
       bctype='Dirichlet',
       bcvalue=0.5)
FD.set_boundary_conditions(pores=pn.pores('bottom'),
       bctype='Dirichlet',
       bcvalue=0.1)
FD.run(conductance='throat.conductance')
```

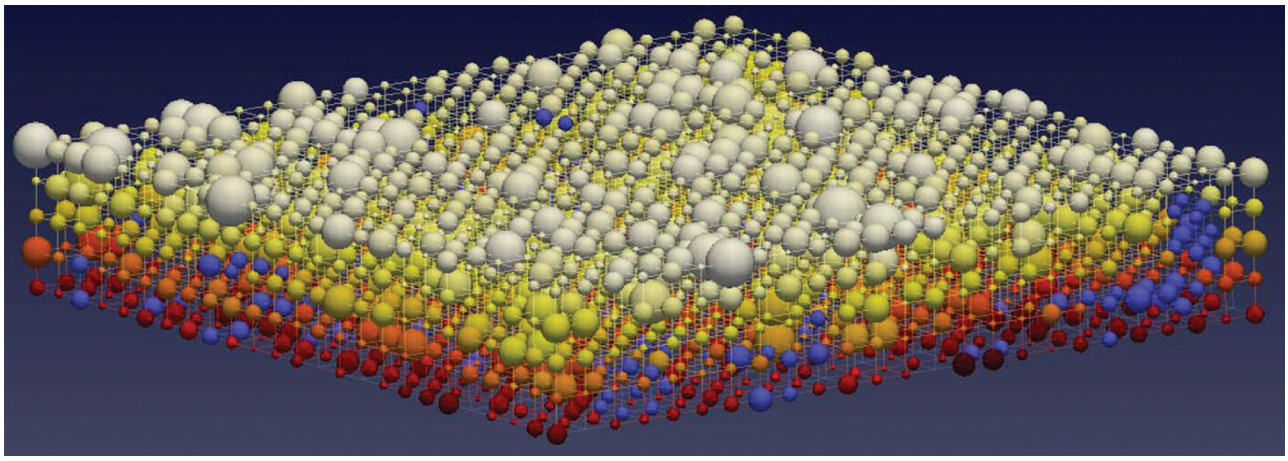


Figure 5. 3D rendering of a pore network showing water invading in from bottom (blue) and gas diffusion from top to bottom (yellow to red). The sizes of the spheres are proportional to their diameter. Throats are drawn as thin lines to enhance visualization.

phase and `'throat.diffusive_conductance'` to the gas phase.

Finally, two `Algorithm` objects are required to perform the water invasion and gas diffusion simulations. For water invasion, drainage will be used:

```
D = OpenPNM.Algorithms.Drainage(network=pn)
D.setup(invading_phase=water,
defending_phase=air)
D.set_inlets(pn.pores('bottom'))
D.run()
```

The `run` method will perform the simulation and place arrays called `'pore.inv_Pc'` and `'throat.inv_Pc'` in `D`'s dictionary. These arrays contain the pressure at which each pore and throat was invaded, thus requiring a simple Boolean comparison to find all locations invaded at some applied pressure (for example, 8,000 Pa).
Finally, gas diffusion is calculated using the provided FickianDiffusion `Algorithm` subclass:

```
FD = OpenPNM.Algorithms.
     FickianDiffusion(network=pn, phase=air)
```

The `Phase` on which this `Algorithm` operates is required as an argument, as this gives the algorithm access to the diffusive conductance values stored on `phys_air`. Mole fraction boundary conditions are set on the top and bottom of the `Network`:

```
FD.set_boundary_conditions(bctype='Dirichlet',
bcvalue=0.5, pores=pn.pores('top'))
FD.set_boundary_conditions(bctype='Dirichlet',
bcvalue=0.1, pores=pn.pores('bottom'))
```

The transport calculation is executed by calling the `run` command:

```
FD.run()
```

Figure 5 shows the resulting `Network`, water configuration (blue pores), and mass fraction distribution (colored pores). Each specific transport phenomena subclass has a method for calculating the effective value of its transport property for the entire `Network`; for FickianDiffusion, this is `calc_effective_diffusivity`, which in this case is approximately half the bulk value and 10 percent less than the dry `Network`.

Open PNM aims to provide users with an easy-to-use, computationally efficient and fully customizable framework for performing PNM calculations of all sorts. We offer this package to the porous media community in the hope that it will become a standard tool in the field, allowing researchers to share code, build on each other's work, and compare results directly. ■

Acknowledgments

OpenPNM was made possible by the support of the Automotive Fuel Cell Cooperation and the Natural Science and Engineering Research Council of Canada (NSERC) through the Collaborative Research & Development and the Discovery Grant programs. Financial support from the NSERC Canada Research Chairs Program, NSERC Collaborative Research and

Training Experience Program (CREATE) in Distributed Generation for Remote Communities (DGRC), Canadian Foundation for Innovation (CFI), and Ontario Ministry of Research and Innovation Early Researcher Award are also gratefully acknowledged. European collaborators acknowledge the support and funding of the Engineering and Physical Sciences Research Council (EPSRC).

References

1. I. Chatzis and F.A.L. Dullien, "The Modeling of Mercury Porosimetry and the Relative Permeability of Mercury in Sandstones Using Percolation Theory," *Int'l J. Chemical Eng.*, vol. 25, no. 1, 1985.
2. R.G. Larson, L.E. Scriven, and H.T. Davis, "Percolation Theory of Two Phase Flow in Porous Media," *Chemical Eng. Science*, vol. 36, no. 1, 1981, pp. 57–73.
3. M.M. Dias and A.C. Payatakes, "Network Models for Two-Phase Flow in Porous Media Part 1. Immiscible Microdisplacement of Non-Wetting Fluids," *J. Fluid Mechanics*, vol. 164, no. 3, 1986, pp. 305–336.
4. C.T. Miller et al., "Multiphase Flow and Transport Modeling in Heterogeneous Porous Media: Challenges and Approaches," *Advances in Water Resources*, vol. 21, no. 2, 1998, pp. 77–120.
5. P.A. García-Salaberri et al., "Effective Diffusivity in Partially-Saturated Carbon-Fiber Gas Diffusion Layers: Effect of Local Saturation and Application to Macroscopic Continuum Models," *J. Power Sources*, vol. 296, no. 11, 2015, pp. 440–453.
6. M.J. Blunt et al., "Pore-Scale Imaging and Modelling," *Advances in Water Resources*, vol. 51, no. 1, 2013, pp. 197–216.
7. H. Dong and M.J. Blunt, "Pore-Network Extraction from Micro-Computerized-Tomography Images," *Physical Rev. E*, vol. 80, no. 3, 2009, article no. 036307.
8. D. Silin et al., "Microtomography and Pore-Scale Modeling of Two-Phase Fluid Distribution," *Transport in Porous Media*, vol. 86, no. 2, 2010, pp. 495–515.
9. J. Hinebaugh, Z. Fishman, and A. Bazylak, "Unstructured Pore Network Modeling with Heterogeneous PEMFC GDL Porosity Distributions," *J. Electrochemical Soc.*, vol. 157, no. 11, 2010, pp. B1651–B1657.
10. S.L. Bryant, D.W. Mellor, and C.A. Cade, "Physically Representative Network Models of Transport in Porous Media," *AIChE J.*, vol. 39, no. 3, 1993, pp. 387–396.
11. R. Thiedmann et al., "Random Geometric Graphs for Modelling the Pore Space of Fibre-Based Materials," *J. Materials Science*, vol. 46, no. 24, 2011, pp. 7745–7759.
12. M.A. Ioannidis and I. Chatzis, "Network Modelling of Pore Structure and Transport Properties of Porous Media," *Chemical Eng. Science*, vol. 48, no. 5, 1993, pp. 951–972.
13. A.G. Hunt, "Basic Transport Properties in Natural Porous Media: Continuum Percolation Theory and Fractal Model," *Complexity*, vol. 10, no. 3, 2005, pp. 22–37.
14. M. Rebai and M. Prat, "Scale Effect and Two-Phase Flow in a Thin Hydrophobic Porous Layer: Application to Water Transport in Gas Diffusion Layers of Proton Exchange Membrane Fuel Cells," *J. Power Sources*, vol. 192, no. 2, 2009, pp. 534–543.
15. M.J. Blunt et al., "Detailed Physics, Predictive Capabilities and Macroscopic Consequences for Pore-Network Models of Multiphase Flow," *Advances in Water Resources*, vol. 25, nos. 8–12, 2002, pp. 1069–1089.
16. J. Gostick et al., "Pore Network Modeling of Fibrous Gas Diffusion Layers for Polymer Electrolyte Membrane Fuel Cells," *J. Power Sources*, vol. 173, no. 1, 2007, pp. 277–290.
17. P.C. Reeves and M.A. Celia, "A Functional Relationship Between Capillary Pressure, Saturation, and Interfacial Area as Revealed by a Pore-Scale Network Model," *Water Resources Research*, vol. 32, no. 8, 1996, pp. 2345–2358.
18. M. Sahimi, "Flow Phenomena in Rocks: From Continuum Models to Fractals, Percolation, Cellular Automata, and Simulated Annealing," *Rev. Modern Physics*, vol. 65, no. 4, 1993, pp. 1393–1534.
19. M.J. Blunt, "Flow in Porous Media —Pore-Network Models and Multiphase Flow," *Current Opinion in Colloid & Interface Science*, vol. 6, no. 3, 2001, pp. 197–207.
20. V. Joekar-Niasar and S.M. Hassanizadeh, "Analysis of Fundamentals of Two-Phase Flow in Porous Media Using Dynamic Pore-Network Models: A Review," *Critical Reviews in Environmental Science & Tech.*, vol. 42, no. 18, 2012, pp. 1895–1976.
21. M. Prat, "Recent Advances in Pore-Scale Models for Drying of Porous Media," *Chemical Eng. J.*, vol. 86, nos. 1–2, 2002, pp. 153–164.
22. A. Johnson et al., "An Improved Simulation of Void Structure, Water Retention and Hydraulic Conductivity in Soil with the Pore-Cor Three-Dimensional Network," *European J. Soil Science*, vol. 54, no. 3, 2003, pp. 477–490.
23. A. Raoof et al., "PoreFlow: A Complex Pore-Network Model for Simulation of Reactive Transport in Variably Saturated Porous Media,"

Computers & Geosciences, vol. 61, 2013, pp. 160–174.
24. M. Secanell et al., "OpenFCST: An Open-Source Mathematical Modelling Software for Polymer Electrolyte Fuel Cells," *ECS Trans.*, vol. 64, no. 3, 2014, pp. 655–680.
25. R. Cimrman, "SfePy: Write Your Own FE Application," ArXiv14046391 Cs, Apr. 2014; http://arxiv.org/abs/1404.6391.
26. T.E. Oliphant, "Python for Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 10–20.
27. K.J. Millman and M. Aivazis, "Python for Scientists and Engineers," *Computing in Science & Eng.*, vol. 13, no. 2, 2011, pp. 9–12.
28. S. van der Walt, S.C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Eng.*, vol. 13, no. 2, 2011, pp. 22–30.
29. A.A. Hagberg, D.A. Schult, and P.J. Swart, "Exploring Network Structure, Dynamics, and Function Using NetworkX," *Proc. 7th Python in Science Conf.*, 2008, pp. 11–15.
30. J.T. Gostick, "Random Pore Network Modeling of Fibrous PEMFC Gas Diffusion Media Using Voronoi and Delaunay Tessellations," *J. Electrochemical Soc.*, vol. 160, no. 8, 2013, pp. F731–F743.
31. K.E. Thompson, "Pore-Scale Modeling of Fluid Transport in Disordered Fibrous Materials," *AIChE J.*, vol. 48, no. 7, 2002, pp. 1369–1389.
32. V. Joekar-Niasar et al., "Network Model Investigation of Interfacial Area, Capillary Pressure and Saturation Relationships in Granular Porous Media," *Water Resources Research,* vol. 46, no. 6, 2010, p. W06526.
33. H. Giesche, "Mercury Porosimetry: A General (Practical) Overview," *Particle & Particle Systems Characterization*, vol. 23, no. 1, 2006, pp. 9–19.
34. W.B. Lindquist, "The Geometry of Primary Drainage," *J. Colloid Interface Science*, vol. 296, no. 2, 2006, pp. 655–668.
35. V. Sygouni, C.D. Tsakiroglou, and A.C. Payatakes, "Capillary Pressure Spectrometry: Toward a New Method for the Measurement of the Fractional Wettability of Porous Media," *Physics of Fluids*, vol. 18, no. 5, 2006, p. 053302.
36. A.P. Sheppard et al., "Invasion Percolation: New Algorithms and Universality Classes," *J. Physics A*, vol. 32, no. 49, 1999, p. L521.
37. A.Q. Raeini, M.J. Blunt, and B. Bijeljic, "Modelling Two-Phase Flow in Porous Media at the Pore Scale Using the Volume-of-Fluid Method," *J. Computational Physics*, vol. 231, no. 17, 2012, pp. 5653–5668.
38. C. Pan, M. Hilpert, and C.T. Miller, "Lattice-Boltzmann Simulation of Two-Phase Flow in Porous Media," *Water Resources Research*, vol. 40, no. 1, 2004, p. W01501.
39. M. Hilpert and C.T. Miller, "Pore-Morphology-Based Simulation of Drainage in Totally Wetting Porous Media," *Advances in Water Resources*, vol. 24, nos. 3–4, 2001, pp. 243–255.

Jeff Gostick, corresponding author, is an assistant professor in chemical engineering at McGill University. He received a PhD in chemical engineering from the University of Waterloo. Gostick is licensed as P.Eng., and he's a member of the American Institute of Chemical Engineers, the Canadian Society of Chemical Engineers, Interpore, and the Electrochemical Society. Contact him at jgostick@gmail.com.

Mahmoudreza Aghighi is a PhD candidate working on pore network modeling at McGill University. His research focuses on modeling transport phenomena in porous media, multiphase flows, and scientific computing. Aghighi received an M.Sc. in chemical engineering from Sharif University of Technology, Tehran. Contact him at mreza_aghighi@yahoo.com.

James Hinebaugh is a postdoctoral associate at the University of Toronto. He specializes in creating pore-scale simulation environments from high-resolution computed tomography images and is a co-founder of the OpenPNM project. Hinebaugh received a PhD in mechanical engineering from the University of Toronto. Contact him at jhinebau@gmail.com.

Tom Tranter is a PhD candidate working on modeling and measuring multiphase flow in polymer electrolyte fuel cells at the University of Leeds. His research interests include renewable energy, solar energy, and renewable heating. Contact him at t.g.tranter@gmail.com.

Michael Hoeh is a PhD candidate working on water electrolyzers at the Institute of Energy and Climate Research at Forschungszentrum Jülich. He was the first recruited member of OpenPNM from outside the initial founding group. His research interests include renewable energy, energy storage and energy systems. Contact him at michael.a.hoh@googlemail.com.

Harold Day is a graduate of McGill University in mechanical engineering, where he also received an M.Eng. in chemical engineering. Contact him at haroldday@gmail.com.

Brennan Spellacy is a graduate of at McGill University in chemical engineering. Contact him at brennanspellacy@gmail.com.

Mostafa H. Sharqawy is an assistant professor of mechanical engineering at the University of Guelph. His research centers on design, performance evaluation, and improvement of sustainable energy and water systems. Sharqawy is a licensed P.Eng in Ontario and a member of American Society of Mechanical Engineering and International Desalination Association. Contact him at melsharq@uoguelph.ca.

Aimy Bazylak is an associate professor in the Department of Mechanical and Industrial Engineering at the University of Toronto. Her research focuses on transport in porous media with an emphasis on visualization via radiography and tomography. Bazylak received a PhD in mechanical engineering from the University of Victoria. She is a licensed P.Eng and holder of a Canada Research Chair in Thermofluidics for Clean Energy. Contact her at abazylak@mie.utoronto.ca.

Alan Burns is an associate professor in the School of Chemical and Process Engineering at the University of Leeds and a senior developer of the commercial CFD code Ansys CFX. His research interests are in numerical algorithms, mathematical models, and applications of multiphase flows. Burns received a PhD in mathematical physics from the University of Durham. Contact him at a.d.burns@leeds.ac.uk.

Werner Lehnert is a professor in the Faculty of Mechanical Engineering at RWTH Aachen University in Germany and heads the high-temperature polymer-electrolyte fuel cells group at the Jülich Institute of Energy and Climate Research. Lehnert received a PhD in physical chemistry from the University of Düsseldorf. Contact him at w.lehnert@fz-juelich.de.

Andreas Putz is a senior research scientist and heads the modeling and simulation group at Automotive Fuel Cell Cooperation. His research focuses on the numerical simulation of transport, electrochemical, and degradation processes of the membrane electrode assembly for automotive fuel cell applications. Putz received a PhD in applied mathematics from the University of British Columbia. Contact him at andreas.putz@afcc-auto.com.

Selected articles and columns from IEEE Computer Society publications are also available for free at http://ComputingNow.computer.org.