OpenPNM

Release

November 08, 2016

Contents

OpenPNM is an open source project aiming to provide porous media researchers with a ready-made framework for performing a wide range of pore network simulations.

Table 1: **Summary of key capabilities offered by OpenPNM**

Defines a universal means of representing any network topology	Based on a sparse representation of the adjacency matrix using principles from graph theory.
Provides a set of tools for querying, inspecting, and manipulating topology	Including finding neighboring pores, labeling specific locations, adding or removing pores and throats, joining networks, subdividing and merging pores to create multiscale models, and much more.
Able to generate various network topologies	Includes network generators for creating cubic or random networks with arbitrary connectivity.
Stores pore and throat property data in vectorized format	Allows for fast calculations even on large networks, and support for the familiar and advanced array access features such as direct indexing, slicing, Boolean masking, etc.
Includes a sophisticated mechanism for calculating the pore-scale properties	A wide assortment of pore-scale transport parameters, pore size calculations, and thermophysical property models are included, and new models can easily be created by users for their specific problem.
Ships with a suite of algorithms for performing network simulations	Includes invasion percolation, capillary drainage, mass diffusion, permeability and so on.
Supports saving, loading, importing and exporting data in numerous formats	Allows importing networks generated or extracted by other code, and exporting data for post-processing and visualization, as well as a native format for saving and loading complete simulations for future analysis.

CHAPTER 1

Documentation

1.1 User Guide

Contents:

1.1.1 Installation Instructions

Installing OpenPNM is done by running the following on your command line:

```
pip install openpnm
```

This will install OpenPNM into your Python environment. To use OpenPNM, open a Python console and type:

```
>>> import OpenPNM
```

To upgrade your OpenPNM to a newer version, use `pip install --upgrade openpnm`.

It is also possible to download the source code directly from Github and work from that. This is not recommended unless you are planning to do 'development' work on the framework. The pip install approach places the source code in the Python directory and out of harms way.

Installing the Python-SciPy Stack Environment

The above installation instructions assume that you have the Python SciPy Stack (Python including including packages like NumPy, SciPy, matplotlib) installed on your system, and the `pip` installer. If you do not, then refer to the sections below for your system. OpenPNM is designed to run with Python 3.

Windows Instructions

The simplest way to get Python and all the necessary Python packages for scientific computing in Windows is to download the WinPython package. This package also comes with Spyder, which provides an integrated development environment (IDE) that is very similar to Matlab, with an editor, command console, variable explorer and so on combined into the same window.

Once WinPython is installed, navigate to the directory where you chose to install it and open spyder.exe. From the 'tools' method, select 'open command prompt'. This will open a special version of the Window command prompt that is aware of the Python installation. Here you will use `pip install openpnm` to have OpenPNM installed on your machine.

As an alternative, you can also use the *Anaconda* distribution by Continuum Analytics for Windows (cf. below for more details).

Apple Instructions

The *Anaconda* distribution by Continuum Analytics is probably the best option for Mac users. Download the installer and follow the install instructions as stated. This will provide you with Python and all the usual scientific packages, as well as Spyder which the OpenPNM developers highly recommend.

Once Anaconda is installed, you can start the Anaconda launcher by double clicking `Launcher.app` in your `~/anaconda` directory, by which you can start a Spyder session, or also launch Spyder directly from the command line by running `spyder`. Anaconda comes with its own package management `conda`, which you can run from the command line to install specific packages or to setup virtual environments. OpenPNM is not registed with the `conda` repository so you must use `pip install openpnm`. Using `conda` you can also update your packages, either updating all by running `conda update anaconda`, which updates all of your packages, or update individual packages, e.g. updating the Python package itself by running `conda upate python`. Anaconda offers the option of running virtual environments next to each other. This allows to easily switch between different versions of Python packages, i.e. having one Python 2.7 enviroment and one Python 3.3 environment. This is possible by running `conda create -n py3k python=3.4 numpy=1.8.1 scipy=0.14 anaconda` from a command line, where you can explicitly specify which package versions should be installed, and now created a new environment called `py3k`. In order to activate this virtual environment, run `source activate py3k` from a command line, and the run `spyder`, which will give you a Spyder session using these specific packages. To run OpenPNM in this environment, you need to first activate the environment with `source activate py3k` and then run `pip install openpnm`, so that OpenPNM is installed in this specific environment.

Linux Instructions

If you are using Linux, we also recommend the *Anaconda* distribution by Continuum Analytics. Download the installer and follow the install instructions as stated. This will provide you with Python and all the usual scientific packages, as well as Spyder which the OpenPNM developers highly recommend. The process of using it is basically identical to Mac environment, so you can look at the instructions for Apple users stated above. The only difference is that the `Launcher.app` is adapted to Linux specifics, but also called `Launcher`. Your Anaconda distribution is also installed in `~/anaconda` and you can use the `conda` commands as stated above to set up environments or update packages.

Other Requirements

It is also suggested to download Paraview for visualizing the networks produced by OpenPNM.

1.1.2 Tutorials

The following 3 tutorials demonstrate the features and design of OpenPNM by example:

Tutorial 1 of 3: Getting Started with OpenPNM

This tutorial is intended to show the basic outline of how OpenPNM works, and necessarily skips many of the more useful and powerful features of the package. So if you find yourself asking "why is this step so labor intensive" it's probably because this tutorial deliberately simplifies some features to provide a more gentle introduction. The second and third tutorials of this User-Guide dive into the package more deeply, but those features are best appreciated once the basics are understood.

Topics Covered in this Tutorial

Learning Objectives

1. Introduce the main OpenPNM objects and their roles
2. Explore the way OpenPNM stores data, including network topology
3. Learn some handy tools for working with objects
4. Generate a standard cubic **Network** topology
5. Calculate geometrical properties and assign them to a **Geometry** object
6. Calculate thermophysical properties and assign to a **Phase** object
7. Define pore-scale physics and assign transport parameters to a **Physics** object
8. Run a permeability simulation using the pre-defined **Algorithm**
9. Use the package to calculate the permeability coefficient of a porous media

---

Hint: Python and Numpy Tutorials

- OpenPNM is written in Python. One of the best guides to learning Python is the excellent interactive online tutorial called Learn Python the Hard Way, which was originally a website and is now a book and video series. The Python Module of the Week website is another excellent resource with nice examples illustrating various features of Python standard library, and has also been published in a book.
- For information on using Numpy, Scipy and generally doing scientific computing in Python checkout the Scipy lecture notes. The Scipy website also offers as solid introduction to using Numpy arrays.
- The Stackoverflow website is an incredible resource for all computing related questions, including simple usage of Python, Scipy and Numpy functions.
- For users more familiar with Matlab, there is a Matlab-Numpy cheat sheet that explains how to translate familiar Matlab commands to Numpy.

Overview of Data Storage in OpenPNM

Before creating an OpenPNM simulation it is necessary to give a quick description of how data is stored in OpenPNM; after all, a significant part of OpenPNM is dedicated to data storage and handling.

Python Dictionaries or *dicts* OpenPNM employs 5 main objects which each store and manage a different type of information or data:

1. **Network**: Manages topological data such as pore spatial locations and pore-to-pore connections
2. **Geometry**: Manages geometrical properties such as pore diameter and throat length
3. **Phase**: Manages thermophysical properties such as temperature and viscosity
4. **Physics**: Manages pore-scale transport parameters such as hydraulic conductance
5. **Algorithm**: Contains algorithms that use the data from other objects to perform simulations, such as diffusion or drainage

We will encounter each of these objects in action before the end of this tutorial.

Each of the above objects is a *subclass* of the Python *dictionary* or *dict*, which is a very general storage container that allows values to be accessed by a name using syntax like:

```
>>> foo = dict()  # Create an empty dict
>>> foo['bar'] = 1  # Store an integer under the key 'bar'
>>> foo['bar']  # Retrieve the integer stored in 'bar'
1
```

A detailed tutorial on dictionaries can be found here. The *dict* does not offer much functionality aside from basic storage of arbitrary objects, and it is meant to be extended. OpenPNM extends the *dict* to have functionality specifically suited for dealing with OpenPNM data. More information about the functionality of OpenPNM's subclassed *dicts* can be found in the *Overall Design*.

***Numpy* Arrays of Pore and Throat Data** All data are stored in arrays which can accessed using standard array syntax. More details on the data storage scheme are given in *Data Storage*, but the following gives a quick overview:

1. All pore and throat properties are stored in Numpy arrays. All data will be automatically converted to a *Numpy* array if necessary.
2. The data for pore *i* (or throat *i*) can be found in element of *i* of an array. This means that pores and throat have indices which are implied by their position in arrays. When we speak of retrieving pore locations, it refers to the indices in the *Numpy* arrays.
3. Arrays that store pore data are *Np*-long, while arrays that store throat data are *Nt*-long, where *Np* is the number of pores and *Nt* is the number of throats in the network.
4. Each property is stored in it's own array, meaning that 'pore diameter' and 'throat volume' are each stored in a separate array.
5. Arrays can be any size in the other dimensions. For instance, triplets of pore coordinates (i.e. [x, y, z]) can be stored for each pore creating an *Np-by-3* array.
6. The storage of topological connections is also very nicely accomplished with this 'list-based' format, by creating an array (`'throat.conns'`) that stores which pore indices are found on either end of a throat. This leads to an *Nt-by-2* array. The implications and advantages of this storage scheme are discussed further in *Representing Topology*.

OpenPNM Objects: Combining *dicts* and *Numpy* Arrays OpenPNM objects combine the above two levels of data storage, meaning they are *dicts* that are filled with *Numpy* arrays. OpenPNM enforces several rules to help maintain data consistency:

1. When storing arrays in an OpenPNM object, their name (or *dictionary key*) must be prefixed with `'pore.'` or `'throat.'`.
2. OpenPNM uses the prefix of the *dictionary key* to infer how long the array must be.

3. The specific property that is stored in each array is indicated by the suffix such as `'pore.diameter'` or `'throat.length'`.
4. Writing scalar values to OpenPNM objects automatically results in conversion to a full length array filled with the scalar value.
5. Arrays containing *Boolean* data are treated as *labels*, which are explained later in this tutorial.

The following code snippets give examples of how all these pieces fit together using an **Empty** network as an example:

```
>>> import OpenPNM
>>> import scipy as sp
>>> net = OpenPNM.Network.Empty(Np=10, Nt=10)  # Instantiate an empty network object with 10 pores a
>>> net['pore.foo'] = sp.ones([net.Np, ])  # Assign an Np-long array of ones
>>> net['pore.bar'] = range(0, net.Np)  # Assign an Np-long array of increasing ints
>>> type(net['pore.bar'])  # The Python range iterator is converted to a proper Numpy array
<class 'numpy.ndarray'>
>>> net['pore.foo'][4] = 44.0  # Overwrite values in the array
>>> net['pore.foo'][4]  # Retrieve values from the array
44.0
>>> net['pore.foo'][2:6]  # Extract a slice of the array
array([  1.,   1.,  44.,   1.])
>>> net['pore.foo'][[2, 4, 6]]  # Extract specific locations
array([  1.,  44.,   1.])
>>> net['throat.foo'] = 2  # Assign a scalar
>>> len(net['throat.foo'])  # The scalar values is converted to an Nt-long array
10
>>> net['throat.foo'][4]  # The scalar value was placed into all locations
2
```

Generate a Cubic Network

Now that we have seen the rough outline of how OpenPNM objects store data, we can begin building a simulation. Start by importing OpenPNM and the Scipy package:

```
>>> import OpenPNM
>>> import scipy as sp
```

Next, generate a **Network** by choosing the **Cubic** class, then create an *instance* with the desired parameters:

```
>>> pn = OpenPNM.Network.Cubic(shape=[4, 3, 1], spacing=0.0001)
```

The **Network** object stored in `pn` contains pores at the correct spatial positions and connections between the pores according the cubic topology.

- The `shape` argument specifies the number of pores in the [X, Y, Z] directions of the cube. Networks in OpenPNM are always 3D dimensional, meaning that a 2D or "flat" network is still 1 layer of pores "thick" so [X, Y, Z] = [20, 10, 1], thus `pn` in this tutorial is 2D which is easier for visualization.
- The `spacing` argument controls the center-to-center distance between pores and it can be a scalar or vector (i.e. [0.0001, 0.0002, 0.0003]).

The resulting network looks like:

This image was creating using Paraview, using the instructions given here: Example in the OpenPNM-Example collection

Inspecting Object Properties OpenPNM objects have additional methods for querying their relevant properties, like the number of pores or throats, which properties have been defined, and so on:

```
>>> pn.num_pores()
12
>>> pn.Np  # Shortcut to get number of pores
12
>>> pn.num_throats()
17
>>> pn.Nt
17
>>> pn.props()
['pore.coords', 'pore.index', 'throat.conns']
```

More information about these various functions is given in *Overall Design*. It is also convenient to type `print(pn)` at the command line to view a nicely formatted table showing the current state of `pn`.

Accessing Pores and Throats One simple but important feature of OpenPNM is the ability to *label* pores and throats. When a **Cubic** network is created, several labels are automatically created: the pores on each face are labeled 'left', 'right', etc. These labels can be used as follows:

```
>>> pn.pores('left')
array([0, 3, 6, 9])
```

The ability to retrieve pore indices is handy for querying pore properties, such as retrieving the pore coordinates of all pores on the 'left' face:

```
>>> pn['pore.coords'][pn.pores('left')]
array([[  5.00000000e-05,   5.00000000e-05,   5.00000000e-05],
       [  1.50000000e-04,   5.00000000e-05,   5.00000000e-05],
       [  2.50000000e-04,   5.00000000e-05,   5.00000000e-05],
       [  3.50000000e-04,   5.00000000e-05,   5.00000000e-05]])
```

A list of all labels currently assigned to the network can be obtained with:

```
>>> pn.labels()
['pore.all', 'pore.back', 'pore.bottom', 'pore.front', 'pore.internal', 'pore.left', 'pore.right', 'p
```

The existing labels are also listed when an object is printed using `print(pn)`. Detailed use of labels is given in *Data Storage*.

Create a Geometry Object and Assign Geometric Properties to Pores and Throats

The **Network** `pn` does not contain any information about pore and throat sizes at this point. The next step is to create a **Geometry** object to manage the geometrical properties.

```
>>> geom = OpenPNM.Geometry.GenericGeometry(network=pn, pores=pn.Ps, throats=pn.Ts)
```

This statement contains three arguments:

- `network` tells the **Geometry** object which **Network** it is associated with. There can be multiple networks defined in a given session, so all objects must be associated with a single network.
- `pores` and `throats` indicate the locations in the **Network** where this **Geometry** object will apply. In this tutorial `geom` applies to *all* pores and throats, but there are many cases where different regions of the network have different geometrical properties, so OpenPNM allows multiple **Geometry** objects to be created for managing the data in each region, but this is a subject for *Tutorial 2 of 3: Digging Deeper into OpenPNM*.

Add Pore and Throat Size Information This freshly instantiated **Geometry** object (`geom`) contains no geometric properties as yet. For this tutorial we'll use the direct assignment of manually calculated values.

We'll start by assigning diameters to each pore from a random distribution, spanning 0 um to 100 um. The upper limit matches the `spacing` of the **Network** which was set to 0.0001 m (i.e. 100 um), so pore diameters exceeding 100 um might overlap with their neighbors. Using the Scipy `rand` function creates an array of random numbers between 0 and 0.0001 that is *Np*-long, meaning each pore is assigned a unique random number

```
>>> geom['pore.diameter'] = sp.rand(pn.Np)*0.0001  # Units of meters
```

We usually want the throat diameters to always be smaller than the two pores which it connects to maintain physical consistency. This requires understanding a little bit about how OpenPNM stores network topology. Consider the following:

```
>>> P12 = pn['throat.conns']  # An Nt x 2 list of pores on the end of each throat
>>> D12 = geom['pore.diameter'][P12]  # An Nt x 2 list of pore diameters
>>> Dt = sp.amin(D12, axis=1)  # An Nt x 1 list of the smaller pore from each pair
>>> geom['throat.diameter'] = Dt
```

Let's dissect the above lines.

- Firstly, `P12` is a direct copy of the **Network's** `'throat.conns'` array, which contains the indices of the pore-pair connected by each throat.
- Next, this *Nt-by-2* array is used to index into the `'pore.diameter'` array, resulting in another *Nt-by-2* array containing the diameters of the pores on each end of a throat.
- Finally, the Scipy function `amin` is used to find the minimum diameter of each pore-pair by specifying the `axis` argument as 1, and the resulting *Nt-by-1* array is assigned to `geom['throat.diameter']`.
- This trick of using `'throat.conns'` to index into a pore property array is commonly used in OpenPNM and you should have a second look at the above code to understand it fully. Refer to *Representing Topology* for a full discussion.

We must still specify the remaining geometrical properties of the pores and throats. Since we're creating a "Stick-and-Ball" geometry, the sizes are calculated from the geometrical equations for spheres and cylinders. For pore volumes, assume a sphere:

```
>>> Rp = geom['pore.diameter']/2
>>> geom['pore.volume'] = (4/3)*3.14159*(Rp)**3
```

The length of each throat is the center-to-center distance between pores, minus the radius of each of two neighboring pores.

```
>>> C2C = 0.0001  # The center-to-center distance between pores
>>> Rp12 = Rp[pn['throat.conns']]
>>> geom['throat.length'] = C2C - sp.sum(Rp12, axis=1)
```

The volume of each throat is found assuming a cylinder:

```
>>> Rt = geom['throat.diameter']/2
>>> Lt = geom['throat.length']
>>> geom['throat.volume'] = 3.14159*(Rt)**2*Lt
```

The basic geometrical properties of the network are now defined. The **Geometry** class possesses a method called `plot_histograms` that produces a plot of the most pertinent geometrical properties. The following figure doesn't look very good since the network in this example has only 12 pores, but the utility of the plot for quick inspection is apparent.

Create a Phase Object

The simulation is now topologically and geometrically defined. It has pore coordinates, pore and throat sizes and so on. In order to perform any simulations it is necessary to define a **Phase** object to manage all the thermophysical properties of the fluids in the simulation:

```
>>> water = OpenPNM.Phases.GenericPhase(network=pn)
```

- `pn` is passed as an argument because **Phases** must know to which **Network** they belong.
- Note that `pores` and `throats` are *NOT* specified; this is because **Phases** are mobile and can exist anywhere or everywhere in the domain, so providing specific locations does not make sense. Algorithms for dynamically determining actual phase distributions are discussed later.

Add Thermophysical Properties Now it is necessary to fill this **Phase** object with the desired thermophysical properties. OpenPNM includes a framework for calculating thermophysical properties from models and correlations, but this is covered in *Tutorial 2 of 3: Digging Deeper into OpenPNM*. For this tutorial, we'll use the basic approach of simply assigning static values as follows:

```
>>> water['pore.temperature'] = 298.0
>>> water['pore.viscosity'] = 0.001
```

- The above lines utilize the fact that OpenPNM converts scalars to full length arrays, essentially setting the temperature in each pore to 298.0 K.

Create a Physics Object

We are still not ready to perform any simulations. The last step is to define the desired pore-scale physics models, which dictate how the phase and geometrical properties interact to give the *transport parameters*. A classic example of this is the Hagen-Poiseuille equation for fluid flow through a throat to predict the flow rate as a function of the pressure drop. The flow rate is proportional to the geometrical size of the throat (radius and length) as well as properties of the fluid (viscosity) and thus combines geometrical and thermophysical properties:

```
>>> phys_water = OpenPNM.Physics.GenericPhysics(network=pn, phase=water, geometry=geom)
```

- As with all objects, the `Network` must be specified
- **Physics** objects combine information from a **Phase** (i.e. viscosity) and a **Geometry** (i.e. throat diameter), so each of these must be specified.
- **Physics** objects do not require the specification of which `pores` and `throats` where they apply, since this information is implied by the `geometry` argument which was already assigned to specific locations.

Specify Desired Pore-Scale Transport Parameters We need to calculate the numerical values representing our chosen pore-scale physics. To continue with the Hagen-Poiseuille example lets calculate the hydraulic conductance of each throat in the network. The throat radius and length are easily accessed as:

```
>>> R = geom['throat.diameter']/2
>>> L = geom['throat.length']
```

The viscosity of the **Phases** was only defined in the pores; however, the hydraulic conductance must be calculated for each throat. There are several options, but to keep this tutorial simple we'll create a scalar value:

```
>>> mu_w = 0.001
>>> phys_water['throat.hydraulic_conductance'] = 3.14159*R**4/(8*mu_w*L)
```

Numpy arrays support *vectorization*, so since both `L` and `R` are arrays of *Nt*-length, their multiplication in this way results in another array that is also *Nt*-long.

Create an Algorithm Object for Performing a Permeability Simulation

Finally, it is now possible to run some useful simulations. The code below estimates the permeability through the network by applying a pressure gradient across and calculating the flux. This starts by creating a **StokesFlow** algorithm, which is pre-defined in OpenPNM:

```
>>> alg = OpenPNM.Algorithms.StokesFlow(network=pn, phase=water)
```

- Like all the above objects, **Algorithms** must be assigned to a **Network** via the `network` argument.
- This algorithm is also associated with a **Phase** object, in this case `water`, which dictates which pore-scale **Physics** properties to use (recall that `phys_water` was associated with `water`).

Next the boundary conditions are applied using the `set_boundary_conditions` method on the **Algorithm** object. Let's apply a 1 atm pressure gradient between the left and right sides of the domain:

```
>>> BC1_pores = pn.pores('front')
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=202650,
...                             pores=BC1_pores)
>>> BC2_pores = pn.pores('back')
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=101325,
...                             pores=BC2_pores)
```

To actually run the algorithm use the `run` method:

```
>>> alg.run()
```

This builds the coefficient matrix from the existing values of hydraulic conductance, and inverts the matrix to solve for pressure in each pore, and stores the results within the **Algorithm's** dictionary under `'pore.pressure'`.

To determine the permeability coefficient, we must invoke Darcy's law: Q = KA/uL(Pin - Pout). Everything in this equation is known except for the volumetric flow rate Q. The **StokesFlow** algorithm possesses a `rate` method that calculates the rate of a quantity leaving a specified set of pores:

```
>>> Q = alg.rate(pores=pn.pores('top'))
>>> A = 0.0001*3*1  # Cross-sectional area for flow
>>> L = 0.0001*4  # Length of flow path
>>> del_P = 101325  # Specified pressure gradient
>>> K = Q*mu_w*L/(A*del_P)
```

The **StokesFlow** class was developed with permeability simulations in mind, so a specific method is available for determining the permeability coefficient that essentially applies to recipe from above. This method could struggle with non-uniform geometries though, so use with caution:

```
>>> K = alg.calc_eff_permeability()
```

The results (`'pore.pressure'`) are held within the `alg` object and must be explicitly returned to the `air` object by the user if they wish to use these values in a subsequent calculation. The point of this data containment is to prevent unintentional overwriting of data. Each algorithm has a method called `return_results` which places the pertinent values back onto the appropriate **Phase** object.

```
>>> alg.return_results()
```

Using Paraview for Visualization, the resulting pressure gradient across the network can be seen:

Tutorial 2 of 3: Digging Deeper into OpenPNM

This tutorial will follow the same outline as the *Tutorial 1 of 3: Getting Started with OpenPNM*, but will dig a little bit deeper at each step to reveal the important features of OpenPNM that were glossed over previously.

Topics Covered in this Tutorial

Learning Objectives

1. Explore different network topologies, and learn some handy topological query methods
2. Create a *heterogeneous* domain with different geometrical properties in different regions
3. Learn about data exchange between objects
4. Utilize pore-scale models for calculating properties of all types
5. Propagate changing geometrical and thermo-physical properties to all dependent properties
6. Calculate the permeability tensor for the stratified media
7. Use the Workspace Manager to save and load a simulation

Building a Cubic Network

As usual, start by importing the OpenPNM and Scipy packages:

```
>>> import OpenPNM
>>> import scipy as sp
```

Let's generate a cubic network again, but with a different connectivity:

```
>>> pn = OpenPNM.Network.Cubic(shape=[20, 20, 10], spacing=0.0001, connectivity=8)
```

- This **Network** has pores distributed in a cubic lattice, but connected to diagonal neighbors due to the `connectivity` being set to 8 (the default is 6 which is orthogonal neighbors). The various options are outlined in the **Cubic** class's documentation which can be viewed with the Object Inspector in Spyder.
- OpenPNM includes several other classes for generating networks including random topology based on Delaunay tessellations (**Delaunay**).
- It is also possible to import networks from external sources, such as networks extracted from tomographic images, or that networks generated by external code.

Initialize and Build Multiple *Geometry Objects*

One of the main functionalities of OpenPNM is the ability to assign drastically different geometrical properties to different regions of the domain to create heterogeneous materials, such as layered structures. To demonstrate the motivation behind this feature, this tutorial will make a material that has different geometrical properties on the top and bottom surfaces compared to the internal pores. We need to create one **Geometry** object to manage the top and bottom pores, and a second to manage the remaining internal pores:

```
>>> Ps1 = pn.pores(['top', 'bottom'])
>>> Ts1 = pn.find_neighbor_throats(pores=Ps1, mode='union')
>>> geom1 = OpenPNM.Geometry.GenericGeometry(network=pn, pores=Ps1, throats=Ts1, name='surface')
>>> Ps2 = pn.pores(['top', 'bottom'], mode='not')
>>> Ts2 = pn.find_neighbor_throats(pores=Ps2, mode='intersection')
>>> geom2 = OpenPNM.Geometry.GenericGeometry(network=pn, pores=Ps2, throats=Ts2, name='core')
```

- The above statements result in two distinct **Geometry** objects, each applying to different regions of the domain. `geom1` applies to only the pores on the top and bottom surfaces (automatically labeled 'top' and 'bottom' during the network generation step), while `geom2` applies to the pores 'not' on the top and bottom surfaces.
- The assignment of throats is more complicated and illustrates the `find_neighbor_throats` method, which is one of the more useful topological query methods on the **Network** class. In both of these calls, all throats connected to the given set of pores (`Ps1` or `Ps2`) are found; however, the `mode` argument alters which throats are returned. The terms `'union'` and `'intersection'` are used in the "set theory" sense, such that `'union'` returns *all* throats connected to the pores in the supplied list, while `'intersection'` returns the throats that are *only* connected to the supplied pores. More specifically, if pores 1 and 2 have throats [1, 2] and [2, 3] as neighbors, respectively, then the `'union'` mode returns [1, 2, 3] and the `'intersection'` mode returns [2]. A detailed description of this behavior is given in *Representing Topology*.

Naming Objects Each of the **Geometry** objects was assigned a `name` during instantiation, and this is stored in the `name` attribute:

```
>>> geom1.name  # Inspect object's name
'surface'
>>> geom1.name = 'foobar'  # Change object's name
>>> geom1.name  # Ensure new name was set
'foobar'
>>> geom1.name = 'surface'  # Replace original name
```

Naming objects in this way serves several purposes:

1. It helps users keep track of which variable points to which object (i.e. `geom1` vs. `geom2`). This is useful when interacting with the objects at the command line using `geom1.name`, which will report `'surface'`.
2. When any core object is instantiated, a *label* is created in the **Network** based on the object's name, indicating which pores and throats belong to which object. It this case, the pores assigned to `geom1` can be quickly retrieved using `pn.pores('surface')` or `pn.pores(geom1.name)`. The use of *labels* is detailed in *Data Storage*.
3. Because the *labels* are so integral to tracking which locations belong to which objects, all **Core** objects are automatically assigned a randomly generated name if none is specified during instantiation.
4. When an object is renamed, OpenPNM takes care of changing the names of the *labels* throughout the simulation. Of course, no two objects can have the same name. In fact, an object cannot be given a name if it is already in use for another *label*.

Assign Static Seed values to *Each* Geometry In *Tutorial 1 of 3: Getting Started with OpenPNM* we only assigned 'static' values to the **Geometry** object, which we calculated explicitly. In this tutorial we will use the *pore-scale models* that are provided with OpenPNM. To get started, however, we'll assign static random seed values between 0 and 1 to each pore on both **Geometry** objects, by assigning random numbers to each Geometry's `'pore.seed'` property:

```
>>> geom1['pore.seed'] = sp.rand(geom1.Np)
>>> geom2['pore.seed'] = sp.rand(geom2.Np)
```

- Each of the above lines produced an array of different length, corresponding to the number of pores assigned to each **Geometry** object. This is accomplished by the calls to `geom1.Np` and `geom2.Np`, which return the number of pores on each object.
- Every Core object in OpenPNM possesses the same set of methods for managing their data, such as counting the number of pore and throat values they represent; thus, `pn.Np` returns 1000 while `geom1.Np` and `geom2.Np` return 200 and 800 respectively.

Accessing Data Distributed Between Geometries The segmentation of the data between separate Geometry objects is essential to the management of pore-scale models, although it does create a complication: it's not easy to obtain a single array containing *all* the values of a given property for the whole network. It is technically possible to piece this data together manually since we know the locations where each **Geometry** object applies, but this is tedious so OpenPNM provides a shortcut. First, let's illustrate the manual approach using the `'pore.seed'` values we have defined:

```
>>> seeds = sp.zeros_like(pn.Ps, dtype=float)
>>> seeds[pn.pores(geom1.name)] = geom1['pore.seed']
>>> seeds[pn.pores(geom2.name)] = geom2['pore.seed']
>>> assert sp.all(seeds > 0)  # Ensure all zeros are overwritten
```

The following code illustrates the shortcut approach, which accomplishes the same result as above in a single line:

```
>>> seeds = pn['pore.seed']
```

- This shortcut works because the `pn` dictionary does not contain an array called `'pore.seed'`, so all associated **Geometry** objects are then checked for the requested array(s). If it is found, then OpenPNM essentially performs the *interleaving* of the data as demonstrated by the manual approach and returns all the values together in a single full-size array. If it is not found, then a standard *KeyError* message is received.
- This exchange of data between **Network** and **Geometry** makes sense if you consider that **Network** objects act as a sort of master object relative **Geometry** objects. **Networks** apply to *all* pores and throats in the domain, while **Geometries** apply to subsets of the domain, so if the **Network** needs some values from all pores it has direct access.

Add Pore Size Distribution Models to Each Geometry Pore-scale models are mathematical functions that are applied to each pore (or throat) in the network to produce some local property value. Each of the modules in OpenPNM (Network, Geometry, Phase and Physics) have a "library" of pre-written models located under "models" (i.e. *Geometry.models*). Below this level, the models are further categorized according to what property they calculate, and there are typical 2-3 models for each. For instance, under `Geometry.models.pore_diameter` you will see `random`, `normal` and `weibull` among others.

Pore size distribution models are assigned to each Geometry object as follows:

```
>>> geom1.models.add(propname='pore.diameter',
...                  model=OpenPNM.Geometry.models.pore_diameter.normal,
...                  scale=0.00002, loc=0.000001,
...                  seeds='pore.seed')
>>> geom2.models.add(propname='pore.diameter',
...                  model=OpenPNM.Geometry.models.pore_diameter.weibull,
...                  shape=1.2, scale=0.00004, loc=0.000001,
...                  seeds='pore.seed')
```

Pore-scale models tend to be the most complex (i.e. confusing) aspects of OpenPNM, so it's worth dwelling on the important points of the above two commands:

- Both `geom1` and `geom2` have a `models` attribute where the parameters specified in the `add` command are stored for future use if/when needed. The `models` attribute actually contains a **ModelsDict** object which is a customized dictionary for storing and managing this type of information.

- The `propname` argument specifies which property the model calculates. This means that the numerical results of the model calculation will be saved in their respective **Geometry** objects as `geom1['pore.diameter']` and `geom2['pore.diameter']`.
- Each model stores it's result under the same `propname` but these values do not conflict since each **Geometry** object presides over a unique subset of pores and throats.
- The `model` argument contains a *handle* to the desired function, which is extracted from the *models* library of the relevant *Module* (**Geometry** in this case). Each **Geometry** object has been assigned a different statistical model, *normal* and *weibull*. This ability to apply different models to different regions of the domain is reason multiple **Geometry** objects are permitted. The added complexity is well worth the added flexibility.
- The remaining arguments are those required by the chosen *model*. In the above cases, these are the parameters that define the statistical distribution. Note that the mean pore size for `geom1` will be 20 um (set by `scale`) while for `geom2` it will be 50 um, thus creating the smaller surface pores as intended. The pore-scale models are well documented regarding what arguments are required and their meaning; as usual these can be viewed with Object Inspector in Spyder.

Now that we've added pore diameter models the each **Geometry** we can visualize the network in Paraview to confirm that distinctly different pore sizes on the surface regions:

Add Additional Pore-Scale Models to Each Geometry In addition to pore diameter, there are several other geometrical properties needed to perform a permeability simulation. Let's start with throat diameter:

```
>>> geom1.models.add(propname='throat.diameter',
...                  model=OpenPNM.Geometry.models.throat_misc.neighbor,
...                  pore_prop='pore.diameter',
...                  mode='min')
>>> geom2.models.add(propname='throat.diameter',
...                  model=OpenPNM.Geometry.models.throat_misc.neighbor,
...                  pore_prop='pore.diameter',
...                  mode='min')
```

Instead of using statistical distribution functions, the above lines use the `neighbor` model which determines each throat value based on the values found `'pore_prop'` from it's neighboring pores. In this case, each throat is assigned the minimum pore diameter of it's two neighboring pores. Other options for `mode` include `'max'` and `'mean'`.

We'll also need throat length as well as the cross-sectional area of pores and throats, for calculating the hydraulic conductance model later.

```
>>> geom1.models.add(propname='throat.length',
...                  model=OpenPNM.Geometry.models.throat_length.straight)
>>> geom2.models.add(propname='throat.length',
...                  model=OpenPNM.Geometry.models.throat_length.straight)
>>> geom1.models.add(propname='throat.area',
...                  model=OpenPNM.Geometry.models.throat_area.cylinder)
>>> geom2.models.add(propname='throat.area',
...                  model=OpenPNM.Geometry.models.throat_area.cylinder)
>>> geom1.models.add(propname='pore.area',
...                  model=OpenPNM.Geometry.models.pore_area.spherical)
>>> geom2.models.add(propname='pore.area',
...                  model=OpenPNM.Geometry.models.pore_area.spherical)
```

Create a Phase Object and Assign Thermophysical Property Models

For this tutorial, we will create a generic **Phase** object for water, then assign some pore-scale models for calculating their properties. Alternatively, we could use the prewritten **Water** class included in OpenPNM, which comes complete with the necessary pore-scale models, but this would defeat the purpose of the tutorial.

```
>>> water = OpenPNM.Phases.GenericPhase(network=pn)
>>> air = OpenPNM.Phases.GenericPhase(network=pn)
```

Note that all **Phase** objects are automatically assigned standard temperature and pressure conditions when created. This can be adjusted:

```
>>> water['pore.temperature'] = 353  # K
```

A variety of pore-scale models are available for calculating **Phase** properties, generally taken from correlations in the literature. An empirical correlation specifically for the viscosity of water is available:

```
>>> water.models.add(propname='pore.viscosity',
...                  model=OpenPNM.Phases.models.viscosity.water)
```

Create Physics Objects for Each Geometry

Physics objects are where geometric information and thermophysical properties are combined to produce the pore and throat scale transport parameters. Thus we need to create one **Physics** object for *EACH* **Phase** and *EACH* **Geometry**:

```
>>> phys1 = OpenPNM.Physics.GenericPhysics(network=pn, phase=water,
...                                        geometry=geom1)
>>> phys2 = OpenPNM.Physics.GenericPhysics(network=pn, phase=water,
...                                        geometry=geom2)
```

Next add the Hagan-Poiseuille model to both:

```
>>> mod = OpenPNM.Physics.models.hydraulic_conductance.hagen_poiseuille
>>> phys1.models.add(propname='throat.hydraulic_conductance', model=mod)
>>> phys2.models.add(propname='throat.hydraulic_conductance', model=mod)
```

- The same function (`mod`) was passed as the `model` argument to both **Physics** objects. This means that both objects will calculate the hydraulic conductance using the same function. A model *must* be assigned to both objects in order for the `'throat.hydraulic_conductance'` property be defined everywhere in the domain since each **Physics** applies to a unique selection of pores and throats.
- The "pore-scale model" mechanism was specifically designed to allow for users to easily create their own custom models. Creating custom models is outlined in advanced_usage.

Accessing Data Distributed Between Multiple Physics Objects Just as **Network** objects can retrieve data from separate **Geometries** as a single array with values in the correct locations, **Phase** objects can retrieve data from **Physics** objects as follows:

```
>>> g = water['throat.hydraulic_conductance']
```

- Each **Physics** applies to the same subset for pores and throats as the **Geometries** so its values are distributed spatially, but each **Physics** is also associated with a single **Phase** object. Consequently, only a **Phase** object can to request all of the values within the domain pertaining to itself.
- In other words, a **Network** object cannot aggregate the **Physics** data because it doesn't know which **Phase** is referred to. For instance, when asking for `'throat.hydraulic_conductance'` it could refer to water or air conductivity, so it can only be requested by water or air.

Pore-Scale Models: The Big Picture

Having created all the necessary objects with pore-scale models, it is now time to demonstrate why the OpenPNM pore-scale model approach is so powerful. First, let's inspect the current value of hydraulic conductance in throat 1 on `phys1` and `phys2`:

```
>>> g1 = phys1['throat.hydraulic_conductance']  # Save this for later
>>> g2 = phys2['throat.hydraulic_conductance']  # Save this for later
```

Now, let's alter the **Geometry** objects by assigning new random seeds, and adjust the temperature of `water`.

```
>>> geom1['pore.seed'] = sp.rand(geom1.Np)
>>> geom2['pore.seed'] = sp.rand(geom2.Np)
>>> water['pore.temperature'] = 370  # K
```

So far we have not run the `regenerate` command on any of these objects, which means that the above changes have not yet been applied to all the dependent properties. Let's do this and examine what occurs at each step:

```
>>> geom1.models.regenerate()
>>> geom2.models.regenerate()
```

These two lines trigger the re-calculation of all the size related models on each **Geometry** object.

```
>>> water.models.regenerate()
```

This line causes the viscosity to be recalculated at the new temperature. Let's confirm that the hydraulic conductance has NOT yet changed since we have not yet regenerated the **Physics** objects' models:

```
>>> sp.all(phys1['throat.hydraulic_conductance'] == g1)  # g1 was saved above
True
>>> sp.all(phys2['throat.hydraulic_conductance'] == g2)  # g2 was saved above
True
```

Finally, if we regenerate `phys1` and `phys2` we can see that the hydraulic conductance will be updated to reflect the new sizes on the **Geometries** *and* the new temperature on the **Phase**:

```
>>> phys1.models.regenerate()
>>> phys2.models.regenerate()
>>> sp.all(phys1['throat.hydraulic_conductance'] != g1)
True
>>> sp.all(phys2['throat.hydraulic_conductance'] != g2)
True
```

Determine Permeability Tensor by Changing Inlet and Outlet Boundary Conditions

The *getting started tutorial* already demonstrated the process of performing a basic permeability simulation. In this tutorial, we'll perform the simulation in all three perpendicular dimensions to obtain the permeability tensor of our heterogeneous anisotropic material.

```
>>> alg = OpenPNM.Algorithms.StokesFlow(network=pn, phase=water)
```

Set boundary conditions for flow in the X-direction:

```
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=202650,
...                             pores=pn.pores('right'))
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=101325,
...                             pores=pn.pores('left'))
>>> alg.run()
```

The resulting pressure field can be seen using Paraview:

To determine the permeability coefficient we must find the flow rate through the network to use in Darcy's law. The **StokesFlow** class (and all analogous transport algorithms) possess a `rate` method that calculates the net transport through a given set of pores:

```
>>> Q = alg.rate(pores=pn.pores('left'))
```

To find K, we need to solve Darcy's law: Q = KA/(mu*L)(P_in - P_out). This requires knowing the viscosity and macroscopic network dimensions:

```
>>> mu = sp.mean(water['pore.viscosity'])
```

The dimensions of the network can be determined manually from the `shape` and `spacing` specified during its generation:

```
>>> L = 20 * 0.0001
>>> A = 20 * 10 * (0.0001**2)
```

The pressure drop was specified as 1 atm when setting boundary conditions, so `Kxx` can be found as:

```
>>> Kxx = Q * mu * L / (A * 101325)
```

We can either create 2 new **Algorithm** objects to perform the simulations in the other two directions, or reuse `alg` by adjusting the boundary conditions and re-running it.

```
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=202650,
...                             pores=pn.pores('front'),
...                             mode='overwrite')
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=101325,
...                             pores=pn.pores('back'),
...                             mode='merge')
>>> alg.run()
```

The first call to `set_boundary_conditions` used the `overwrite` mode, which replaces all existing boundary conditions on the `alg` object with the specified values. The second call uses the `merge` mode which adds new boundary conditions to any already present, which is the default behavior.

A new value for the flow rate must be recalculated, but all other parameters are equal to the X-direction:

```
>>> Q = alg.rate(pores=pn.pores('back'))
>>> Kyy = Q * mu * L / (A * 101325)
```

The values of `Kxx` and `Kyy` should be nearly identical since both these two directions are parallel to the small surface pores. For the Z-direction:

```
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=202650,
...                             pores=pn.pores('top'),
...                             mode='overwrite')
>>> alg.set_boundary_conditions(bctype='Dirichlet', bcvalue=101325,
...                             pores=pn.pores('bottom'))
>>> alg.run()
>>> Q = alg.rate(pores=pn.pores('bottom'))
>>> L = 10 * 0.0001
>>> A = 20 * 20 * (0.0001**2)
>>> Kzz = Q * mu * L / (A * 101325)
```

The permeability in the Z-direction is about half that in the other two directions due to the constrictions caused by the small surface pores.

Using the Workspace Manager to Save (and reload) the simulation

OpenPNM includes a **Workspace** manager that provides the type of functionality found on the *menu-bar* of a typical GUI-based application Specifically, this enables *saving* and *loading* of all active networks, or individual objects.

To use these feature it is necessary to instantiate an instance:

```
mgr = OpenPNM.Base.Workspace()
mgr.save('filename.pnm')
```

Some of the more common functions of the **Workspace** are available via short-cuts under the main package, such that `op.save` is equivalent to calling `mgr.save`.

The **Workspace** also offers a few other functions, such as `purge_object`` which removes an object from the simulation, including all traces of its labels and references on other objects. It is also possible to `clear`` the entire workspace, which is useful for *clearing the slate* when importing a new network. Of course, there is also a `load`` function to load saved *pnm* files:

```
mgr.clear()
mgr.load('filename.pnm')
```

Tutorial 3 of 3: Advanced Topics and Usage

Topics Covered in this Tutorial

Learning Outcomes

1. Use different methods to add boundary pores to a network
2. Manipulate network topology by adding and removing pores and throats
3. Explore the ModelsDict design, including copying models between objects, and changing model parameters
4. Write a custom pore-scale model and a custom Phase
5. Access and manipulate objects associated with the network
6. Combine multiple algorithms to predict relative permeability

Build and Manipulate Network Topology

For the present tutorial, we'll keep the topology simple to help keep the focus on other aspects of OpenPNM.

```
>>> import scipy as sp
>>> import OpenPNM as op
>>> pn = op.Network.Cubic(shape=[10, 10, 10], spacing=0.00006, name='net')
```

Adding Boundary Pores When performing transport simulations it is often useful to have 'boundary' pores attached to the surface(s) of the network where boundary conditions can be applied. When using the **Cubic** class, two methods are available for doing this: `add_boundaries`, which is specific for the **Cubic** class, and `add_boundary_pores`, which is a generic method that can also be used on other network types and which is inherited from **GenericNetwork**. The first method automatically adds boundaries to ALL six faces of the network and offsets them from the network by 1/2 of the value provided as the network `spacing`. The second method provides total control over which boundary pores are created and where they are positioned, but requires the user to specify to which pores the boundary pores should be attached to. Let's explore these two options:

```
>>> pn.Np
1000
>>> pn.Nt
2700
>>> pn.add_boundaries()
>>> pn.Np
1600
>>> pn.Nt
3300
```

Let's remove all these newly created boundary pores. When they are created these pores are all automatically labeled with a label such as `'top_boundary'`, so we can select all boundary pores by using the 'wildcard' feature in the `pores` look-up method to find all pores with a label containing the word `boundary`.

```
>>> Ps = pn.pores('*boundary')  # Using the * wildcard
```

We can then `trim` these pores from the network using:

```
>>> pn.trim(pores=Ps)
>>> pn.Np
1000
>>> pn.Nt
2700
```

Note that all throats connecting to the trimmed pores were automatically removed since OpenPNM does not allow 'dangling' or 'headless' throats.

Now that `pn` is back to its original size, let's explore the second approach to apply boundary pores.

```
>>> Ps = pn.pores('top')  # Select pores on top of network
>>> pn.add_boundary_pores(pores=Ps, offset=[0, 0, 0.00003],
...                       apply_label='top_boundary')
>>> Ps = pn.pores('bottom')  # Select pores on bottom of network
>>> pn.add_boundary_pores(pores=Ps, offset=[0, 0, -0.00003],
...                       apply_label='bottom_boundary')
>>> pn.Np
1200
>>> pn.Nt
2900
```

This approach requires more typing than the `add_boundaries` method, but allows for much finer control over how boundaries are created.

Adding and Removing Pores and Throats OpenPNM uses a list-based data storage scheme for all properties, including topological connections. One of the benefits of this approach is that adding and removing pores and throats from the network is essentially as simple as adding or removing rows from the data arrays. The one exception to this 'simplicity' is that the `'throat.conns'` array must be treated carefully when trimming pores, so OpenPNM provides the `extend` and `trim` functions for added and removing, respectively. To demonstrate, let's reduce the coordination number of the network to create a more random structure:

```
>>> Ts = sp.rand(pn.Nt) < 0.1  # Create a mask with ~10% of throats labeled True
>>> pn.trim(throats=Ts)  # Use mask to indicate which throats to trim
```

When the `trim` function is called, it automatically checks the health of the network afterwards, so logger messages might appear on the command line if problems were found such as isolated clusters of pores or pores with no throats. This health check is performed by calling the **Network**'s' `check_network_health` method which returns a **HealthDict** containing the results of the checks:

```
>>> a = pn.check_network_health()
>>> pn.trim(pores=a['trim_pores'])
```

The **HealthDict** contains several lists including things like duplicate throats and isolated pores, but also a suggestion of which pores to trim to return the network to a healthy state. Also, the **HealthDict** has a `health` attribute that is `False`` is any checks fail.

Define Geometry Objects

The boundary pores we've added to the network should be treated a little bit differently. Specifically, they should have no volume or length (as they are not physically representative of real pores). To do this, we create two separate **Geometry** objects, one for internal pores and one for the boundaries:

```
>>> Ps = pn.pores('*boundary', mode='not')
>>> geom = op.Geometry.Stick_and_Ball(network=pn, pores=Ps, throats=pn.Ts,
...                                   name='internal')
>>> Ps = pn.pores('*boundary')
>>> boun = op.Geometry.GenericGeometry(network=pn, pores=Ps, name='boundary')
```

The **Stick_and_Ball** class is preloaded with the pore-scale models to calculate all the necessary size information (pore diameter, throat lengths, etc). The **GenericGeometry** class used for the boundary pores is empty and requires work:

```
>>> boun['pore.diameter'] = 0
>>> boun['pore.volume'] = 0
```

These models are required for the Hagan-Poiseuille model. Most of them are straight-forward geometry calculations, except for the model used for `'throat.diameter'`. In this case the model looks into the neighbor pores, retrieves the two `'pore.diameter'` and uses the `'max'` value. Because we set the boundary pores to have 0 diameter, this will naturally find result in the throat being assigned the diameter of the internal pore.

Define Multiple Phase Objects

In order to simulate relative permeability of air through a partially water-filled network, we need to create each **Phase** object. OpenPNM includes pre-defined classes for each of these common fluids:

```
>>> air = op.Phases.Air(network=pn)
>>> water = op.Phases.Water(network=pn)
>>> water['throat.contact_angle'] = 110
>>> water['throat.surface_tension'] = 0.072
```

Aside: Creating a Custom Phase Class In many cases you will want to create your own fluid, such as an oil or brine, which may be commonly used in your research. OpenPNM cannot predict all the possible scenarios, but luckily it is easy to create a custom **Phase** class as follows:

Listing 1.1: **Example of a Subclassed Phase**

```
1 from OpenPNM.Phases import GenericPhase, models
2
3 class Oil(GenericPhase):
```

```
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.models.add(propname='pore.viscosity',
                        model=models.misc.polynomial,
                        poreprop='pore.temperature',
                        a=[1.82082e-2, 6.51E-04, -3.48E-7, 1.11E-10])
        self['pore.molecular_weight'] = 116  # g/mol
```

- Creating a **Phase** class basically involves placing a series of `self.models.add` commands within the `__init__` section of the class definition. This means that when the class is instantiated, all the models are added to *itself* (i.e. `self`).
- `**kwargs` is a Python trick that captures all arguments in a *dict* called `kwargs` and passes them to another function that may need them. In this case they are passed to the `__init__` method of **Oil**'s parent by the `super` function. Specifically, things like `name` and `network` are expected.
- The above code block also stores the molecular weight of the oil as a constant value
- Adding models and constant values in this way could just as easily be done in a run script, but the advantage of defining a class is that it can be saved in a file (i.e. 'my_custom_phases') and reused in any project:

```
from my_custom_phases import Oil
oil = Oil(network=pn)
```

Define Physics Objects for Each Geometry and Each Phase

In the previous tutorial we created two **Physics** object, one for each of the two **Geometry** objects used to handle the stratified layers. In this tutorial, the internal pores and the boundary pores each have their own **Geometry**, but there are two **Phases**, which also each require a unique **Physics**:

```
>>> phys_water_internal = op.Physics.GenericPhysics(network=pn, phase=water, geometry=geom)
>>> phys_air_internal = op.Physics.GenericPhysics(network=pn, phase=air, geometry=geom)
>>> phys_water_boundary = op.Physics.GenericPhysics(network=pn, phase=water, geometry=boun)
>>> phys_air_boundary = op.Physics.GenericPhysics(network=pn, phase=air, geometry=boun)
```

- To reiterate, *one* **Physics** object is required for each **Geometry** *AND* each **Phase**, so the number can grow to become annoying very quickly Some useful tips for easing this situation are given below.

Create a Custom Pore-Scale Physics Model Perhaps the most distinguishing feature between pore-network modeling papers is the pore-scale physics models employed. Accordingly, OpenPNM was designed to allow for easy customization in this regard, so that you can create your own models to augment or replace the ones included in the OpenPNM *models* libraries. For demonstration, let's implement the capillary pressure model proposed by Mason and Morrow in 1994. They studied the entry pressure of non-wetting fluid into a throat formed by spheres, and found that the converging-diverging geometry increased the capillary pressure required to penetrate the throat. As a simple approximation they proposed .

Pore-scale models are written as basic function definitions:

Listing 1.2: **Example of a Pore-Scale Model Definition**

```
>>> def mason_model(network, phase, physics, f=0.6667, **kwargs):
...     Dt = network['throat.diameter']
...     theta = phase['throat.contact_angle']
...     sigma = phase['throat.surface_tension']
...     Pc = -4*sigma*sp.cos(f*sp.deg2rad(theta))/Dt
...     return Pc[network.throats(physics.name)]
```

Let's examine the components of above code:

- The function receives `network`, `phase` objects as arguments. Each of these provide access to the properties necessary for the calculation: `'pore.diameter'` values are retrieved via the `network`, and the thermo-physical properties are retrieved directly from the `phase`.
- The `f` value is a scale factor that is applied to the contact angle. Mason and Morrow suggested a value of 2/3 as a decent fit to the data, but we'll make this an adjustable parameter with 2/3 as the default.
- Note the `pore.diameter` is actually a **Geometry** property, but it is retrieved via the network using the data exchange rules outlined in the second tutorial, and explained fully in *Data Storage*.
- All of the calculations are done for every throat in the network, but this pore-scale model is meant to be assigned to a single **Physics** object. As such, the last line extracts values from the `Pc` array for the location of `physics` and returns just the subset.
- The actual values of the contact angle, surface tension, and throat diameter are NOT sent in as numerical arrays, but rather as dictionary keys to the arrays. There is one very important reason for this: if arrays had been sent, then re-running the model would use the same arrays and hence not use any updated values. By having access to dictionary keys, the model actually looks up the current values in each of the arrays whenever it is run.
- It would be a better practice to include the dictionary keys as arguments, such as `'contact_angle = 'throat.contact_angle''`. This way the user could control where the contact angle could be stored on the **Phase** object.

Assuming this function is saved in a file called 'my_models.py' in the current working directory, this model can be used as:

```
from my_models import mason_model
```

Copy Models Between Physics Objects As mentioned above, the need to specify a separate **Physics** object for each **Geometry** and **Phase** can become tedious. It is possible to *copy* the pore-scale models assigned to one object onto another object. First, let's assign the models we need to `phys_water_internal`:

```
>>> phys_water_internal.models.add(propname='throat.capillary_pressure',
...                                model=mason_model)
>>> mod = op.Physics.models.hydraulic_conductance.hagen_poiseuille
>>> phys_water_internal.models.add(propname='throat.hydraulic_conductance',
...                                model=mod)
```

Now make a copy of the `models` on `phys_water_internal` and apply it all the other water **Physics** objects:

```
>>> mods = phys_water_internal.models.copy()
>>> phys_water_boundary.models = mods.copy()
```

The only 'gotcha' with this approach is that each of the **Physics** objects must be *regenerated* in order to place numerical values for all the properties into the data arrays:

```
>>> phys_water_boundary.models.regenerate()
>>> phys_air_internal.models.regenerate()
>>> phys_air_internal.models.regenerate()
```

Access Other Objects via the Network The above code used 3 lines to explicitly regenerate each **Physics** object, but an alternative and more efficient approach is possible. When every object is created, it is 'registered' with the **Network** which is a required argument in the instantiation of every other object. Any object can be looked-up by it's type using `pn.geometries`, `pn.phases`, or `pn.physics`, which return a *dict* containing *key-value* pair of `{object.name: object}`. The *dict* also has a `'keys'` method that lists the names of the stored objects:

```
>>> sorted(list(pn.geometries.keys()))  # Convert to list and sort
['boundary', 'internal']
```

One handy use of this list is that is can be iterated over to perform an action on all objects in one line. In this case running the `regenerate` method on all **Physics** objects can be accomplished with:

```
>>> temp = [item.regenerate for item in pn.physics.values()]
```

The `values` method of the *dict* class returns a list of the objects stored under each key.

Adjust Pore-Scale Model Parameters The pore-scale models are stored in a **ModelsDict** object that is itself stored under the `models` attribute of each object. This arrangement is somewhat convoluted, but it enables integrated storage of models on the object's wo which they apply. The models on an object can be inspected with `print(phys_water_internal)`, which shows a list of all the pore-scale properties that are computed by a model, and some information about the model's *regeneration* mode.

Each model in the **ModelsDict** can be individually inspected by accessing it using the dictionary key corresponding to *pore-property* that it calculates, i.e. `print(phys_water_internal)['throat.capillary_pressure']`. This shows a list of all the parameters associated with that model. It is possible to edit these parameters directly:

```
>>> phys_water_internal.models['throat.capillary_pressure']['f']  # Inspect present value
0.6667
>>> phys_water_internal.models['throat.capillary_pressure']['f'] = 0.75  # Change value
>>> phys_water_internal.models.regenerate()  # Regenerate model with new 'f' value
```

More details about the **ModelsDict** and **ModelWrapper** classes can be found in models.

Perform Multiphase Transport Simulations

Use the Built-In Drainage Algorithm to Generate an Invading Phase Configuration

```
>>> inv = op.Algorithms.Drainage(network=pn)
>>> inv.setup(invading_phase=water, defending_phase=air)
>>> inv.set_inlets(pores=pn.pores(['top', 'bottom']))
>>> inv.run()
```

- The inlet pores were set to both `'top'` and `'bottom'` using the `pn.pores` method. The algorithm applies to the entire network so the mapping of network pores to the algorithm pores is 1-to-1.
- The `run` method automatically generates a list of 25 capillary pressure points to test, but you can also specify more pores, or which specific points to tests. See the methods documentation for the details.
- Once the algorithm has been run, the resulting capillary pressure curve can be viewed with `plot_drainage_curve`. If you'd prefer a table of data for plotting in your software of choice you can use `get_drainage_data` which prints a table in the console.

Set Pores and Throats to Invaded After running, the `mip` object possesses an array containing the pressure at which each pore and throat was invaded, stored as `'pore.inv_Pc'` and `'throat.inv_Pc'`. These arrays can be used to obtain a list of which pores and throats are invaded by water, using Boolean logic:

```
>>> Pi = inv['pore.inv_Pc'] < 10000
>>> Ti = inv['throat.inv_Pc'] < 10000
```

The resulting Boolean masks can be used to manually adjust the hydraulic conductivity of pores and throats based on their phase occupancy. The following lines set the water filled throats to near-zero conductivity for air flow:

```
>>> phys_water_internal['throat.hydraulic_conductance'][~Ti] = 1e-20
```

- The logic of these statements implicitly assumes that transport between two pores is only blocked if the throat is filled with the other phase, meaning that both pores could be filled and transport is still permitted. Another option would be to set the transport to near-zero if *either* or *both* of the pores are filled as well.

- The above approach can get complicated if there are several **Geometry** objects, and it is also a bit laborious. There is a pore-scale model for this under **Physics.models.multiphase** called `conduit_conductance`. The term conduit refers to the path between two pores that includes 1/2 of each pores plus the connecting throat.

Calculate Relative Permeability of Each Phase We are now ready to calculate the relative permeability of the domain under partially flooded conditions. Instantiate an **StokesFlow** object:

```
>>> water_flow = op.Algorithms.StokesFlow(network=pn, phase=water)
>>> water_flow.set_boundary_conditions(pores=pn.pores('left'), bcvalue=200000, bctype='Dirichlet')
>>> water_flow.set_boundary_conditions(pores=pn.pores('right'), bcvalue=100000, bctype='Dirichlet')
>>> water_flow.run()
>>> Q_partial = water_flow.rate(pores=pn.pores('right'))
```

The *relative* permeability is the ratio of the water flow through the partially water saturated media versus through fully water saturated media; hence we need to find the absolute permeability of water. This can be accomplished by *regenerating* the `phys_water_internal` object, which will recalculate the `'throat.hydraulic_conductance'` values and overwrite our manually entered near-zero values from the `inv` simulation using `phys_water_internal.models.regenerate()`. We can then re-use the `water_flow` algorithm:

```
>>> water_flow.run()
>>> Q_full = water_flow.rate(pores=pn.pores('right'))
```

And finally, the relative permeability can be found from:

```
>>> K_rel = Q_partial/Q_full
```

- The ratio of the flow rates gives the normalized relative permeability since all the domain size, viscosity and pressure differential terms cancel each other.
- To generate a full relative permeability curve the above logic would be placed inside a for loop, with each loop increasing the pressure threshold used to obtain the list of invaded throats (`Ti`).
- The saturation at each capillary pressure can be found be summing the pore and throat volume of all the invaded pores and throats using `Vp = geom['pore.volume'][Pi]` and `Vt = geom['throat.volume'][Ti]`.

1.1.3 Reference

Contents:

Overall Design

The design of OpenPNM is to separate different types of properties between different objects. There are 5 types: **Network**, **Geometry**, **Phase**, **Physics**, and **Algorithms**. Each of these are described in more detail below, but their names clearly indicate what sort of data or calculations are assigned to each.

The image below outlines how each of the main objects in OpenPNM descend from the Python `dict` class. Using the Object Oriented Programming (OOP) paradigm, each of the main OpenPNM objects actually descend from a common class called **Core**. The **Core** class defines the majority of the functionality, which is then enhanced and extended by each descendent. This extra functionality is explored in more detail for each main object in the sections below.

Core

Core is a subclass of the Python Dictionary or `dict`. A `dict` is a very handy data structure that can store any piece of data by name, using the following:

```
>>> # Instantiate a dict and add some values by name
>>> foo = dict()
>>> foo['an_int'] = 1
>>> foo['a_list'] = [1, 2, 3]
>>> foo['a_string'] = 'bar'
>>> # And data can be retrieved by name
>>> foo['an_int']
1
>>> foo['a_list']
[1, 2, 3]
```

The Python `dict` class comes with a variety of methods for adding, removing, and inspecting the data stored within. The following command will generate a list of all these methods, which include things like `pop` for removing items from the dictionary, and `keys` for listing all the current dictionary entries.

```
>>> methods = [item for item in dir(foo) if not item.startswith('_')]
```

The **Core** class possess all of these methods, plus another few dozen methods that were added by OpenPNM. These additional methods also pertain to the manipulation of data, but are specific to the types of data used in OpenPNM.

1. Querying Defined Properties and Labels Returns a list of which properties or labels exist in the dictionary. These methods are basically the same as the `keys` method, but return a subset of the entries. Any arrays of Boolean type are considered labels, while all other are properties. The returned list outputs a nicely formatted table to the command line when it is printed.

`props`([element, mode])	Returns a list containing the names of all defined pore or throat properties.
`labels`([pores, throats, element, mode])	Returns the labels applied to specified pore or throat locations

2. Counting Pores and Throats Returns the number of pores or throats that the object controls. Both optionally accept a list of labels and returns the number of pores or throats possessing those labels. There is a `mode` argument which allows control over how the label query is performed. `Np` and `Nt` are short-cuts that return the total number of pores or throats.

`num_pores`([labels, mode])	Returns the number of pores of the specified labels
`num_throats`([labels, mode])	Return the number of throats of the specified labels
`Np`	A shortcut to query the total number of pores on the object'
`Nt`	A shortcut to query the total number of throats on the object'

3. Retrieving a List of Specific Pores and Throats Returns a list of pore or throat indices. Both optionally accept a list of labels and returns only a list of pores or throats possessing those labels. There is a `mode` argument which allows control over how the label query is performed. `Ps` and `Ts` are short-cuts that return ALL of the pore or throat indices.

`pores`([labels, mode])	Returns pore locations where given labels exist, according to the logic specified by the mode argument.
`throats`([labels, mode])	Returns throat locations where given labels exist, according to the logic specified by the mode argument.
`Ps`	A shortcut to get a list of all pores on the object
`Ts`	A shortcut to get a list of all throats on the object

4. Converting Between Masks and Indices These methods allow the conversion between numeric indices and Boolean masks.

`tomask`([pores, throats])	Convert a list of pore or throat indices into a boolean mask of the

Continued on next page

Table 1.4 – continued from previous page

toindices(mask)	Convert a boolean mask a list of pore or throat indices

5. Mapping Pore and Throat Indices Between Objects Each **Core** object has it's own internal numbering scheme, so these methods are for converting the pore or throat indices from one object to another. Practically speaking this usually means mapping from a **Geometry** or **Physics** object onto the **Network**, so Pnet and Tnet are short-cuts for retrieving a list of pore or throat indices on the network.

map_pores([target, pores, return_mapping])	Accepts a list of pores from the caller object and maps them onto the
map_throats([target, throats, return_mapping])	Accepts a list of throats from the caller object and maps them onto the
Pnet	A shortcut to retrieve the mapping of the current object's pores onto the network
Tnet	A shortcut to retrieve the mapping of the current object's throats onto the networ

6. Looking Up Other Objects in the Simulation When each object is instantiated it is associated with the other objects within the simulation. These methods allow for retrieval of these other objects.

OpenPNM.Base.Core.network
OpenPNM.Base.Core.geometries
OpenPNM.Base.Core.phases
OpenPNM.Base.Core.physics

7. Interpolating Between Pore and Throat Data Data is often calculated or assigned to pores or throats only. This method enables the conversion of data between these.

interpolate_data(data)	Determines a pore (or throat) property as the average of it's

8. Check the Health of all Data Arrays Checks whether any data on the object is not well formed, such as containing NaNs, or infs. This is handy be running an algorithm to ensure that all necessary properties have been defined everywhere.

check_data_health([props, element])	Check the health of pore and throat data arrays.

9. Using Pore-Scale Models The models attribute actually contains a nested dictionary which stores all the information related to the pore-scale models. This is described elsewhere in detail. add_model and regenerate are wrapper or helper methods to provide quicker access to the add and regenerate methods of the models dict.

add_model(propname, model[, regen_mode])	Add specified property estimation model to the object.
regenerate([props, mode])	This updates properties using any models on the object that were

10. Find and Set the Object's Name Contains a unique string identifier for the object. It can be specified or assigned at will, but no to objects can have the same name.

name

Network

1. Check the Health of Associated Geometry Objects Inspects that all pores and throats have been assigned to a **Geometry** object.

check_geometry_health()	Perform a check to find pores with overlapping or undefined Geometries

2. Check the Health of the Netowrk Topology Performs a suite of topological checks for ill conditioned networks (disconnected pores, duplicate throats, etc.)

check_network_health()	This method check the network topological health by checking for:

3. Manipulate Pore Topology These are topological manipulation methods that are used to add or remove pores and throats from the network. These are helper methods for the actual functions in **Network.tools**.

clone_pores(pores[, apply_label, mode])	This function as been moved to Network.tools and remains here for backw
connect_pores(pores1, pores2[, labels])	This function as been moved to Network.tools and remains here for backw
extend([pore_coords, throat_conns, labels])	This function as been moved to Network.tools and remains here for backw
stitch(donor, P_donor, P_network, method[, ...])	This function as been moved to Network.tools and remains here for backw
trim([pores, throats])	This function as been moved to Network.tools and remains here for backw

4. Query Neighborhood These methods can be used to query the neighborhood around a given set of pores.

find_neighbor_pores(pores[, mode, flatten, ...])	Returns a list of pores neighboring the given pore(s)
find_neighbor_throats(pores[, mode, flatten])	Returns a list of throats neighboring the given pore(s)
find_nearby_pores(pores, distance[, ...])	Find all pores within a given radial distance of the input pore(s) regardless o
find_connected_pores([throats, flatten])	Return a list of pores connected to the given list of throats
find_connecting_throat(P1, P2)	Return the throat number connecting pairs of pores

5. Adjacency and Incidence Matrices Returns a *Scipy Sparse* array describing the topology of the network.

create_adjacency_matrix([data, sprsfmt, ...])	Generates a weighted adjacency matrix in the desired sparse format
create_incidence_matrix([data, sprsfmt, ...])	Creates an incidence matrix filled with supplied throat values

6. Search for Clusters of Pores Finds connected clusters of pores based on a given list of Boolean values. The 2nd generation of this algorithm has more options that the original, which was kept for backwards compatibility.

find_clusters([mask])	Identify connected clusters of pores in the network.
find_clusters2([mask, t_labels])	Identify connected clusters of pores in the network.

7. Query the Domain Size These calculate the bulk dimensions of the domain.

domain_area(face)	Calculate the area of a given network face
domain_length(face_1, face_2)	Calculate the distance between two faces of the network
domain_bulk_volume()	
domain_pore_volume()	

Geometry

1. Assign Geometry to Specific Pores and Throats When instantiating a **Geometry** object it is normal to specify which pores and throats it applies to. These can be adjusted after the fact with this method.

`set_locations`([pores, throats, mode])	Assign or unassign a Geometry object to specified locations

Phase

1. Check the Health of Associated Physics Objects Inspects that all pores and throats have been assigned to a **Physics** object.

`check_physics_health`()	Perform a check to find pores which have overlapping or undefined Physics

2. Check the Health of a Mixture Phase Mixtures are not fully implemented yet, but this makes sure all mole fractions sum to 1.

`check_mixture_health`()	Query the properties of the 'virtual phases' that make up a mixture

Physics

1. Assign Physics to Specific Pores and Throats When instantiating a **Physics** object it is normal to specify which pores and throats it applies to. These can be adjusted after the fact with this method.

`set_locations`([pores, throats, mode])	Assign or unassign a Physics object to specified locations

2. Lookup the Parent Phase The `phases` method of the **Core** class gives the ability to find a list of all **Phases** in the simulation, but this method returns a handle to the specific **Phase** it's associated with.

`parent_phase`

Algorithms

Depending on the **Algorithm** in question, the additional methods can vary. Most have:

1. Specifying Setup Parameters This method is called to specify some of the optional parameters

2. Setting Boundary Conditions Used to specify the boundary conditions of the simulation. Some methods also include `set_inlets` and `set_outlets`.

Data Storage

Each OpenPNM Core object is a Python *dictionary* which is similar to a structured variable or *struct* in other languages. This allows data to be stored and accessed by name, with a syntax like `network['pore.diameter']`. Inside each *dict* are stored numerous arrays containing pore or throat data corresponding to the *key* (i.e. `'pore.diameter'` values).

All pore and throat data are stored in arrays of either *Np* or *Nt* length representing the number of pores and throats on the object, respectively. This means that each pore (or throat) has a number that is implicitly indicated by it's location in the arrays. All properties for pore *i* or throat *j* are stored in the array at the element *i* or *j*. 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. This array-based approach is ideal when using the Numpy and Scipy libraries which are designed for elementwise, vectorized programming. For instance, the volume of each throats can be found simultaneously using `T_vol = 3.1415*(network['throat.radius']**2) *`

`network['throat.length']`. `T_vol` will be an *Nt*-long array of values, assuming `'throat.length'` and `'throat.radius'` were also *Nt*-long.

Several rules have been implemented to control the integrity of the data:

1. All array names must begin with either *'pore.'* or *'throat.'* which serves to identify the type of information they contain.
2. For the sake of consistency only arrays of length *Np* or *Nt* are allowed in the dictionary. Assigning a scalar value to a dictionary results in the creation of a full length vector, either *Np* or *Nt* long, depending on the name of the array.. This effectively applies the scalar value to all locations in the network.
3. Any Boolean data will be treated as a *label* while all other numerical data is treated as a *property*. The difference between these is outlined below.

Properties (aka Data)

The physical details about pores and throats are referred to as *properties*, which includes information such as *pore volume* and *throat length*. Properties are accessed using Python dictionary syntax to access the array of choice, then Numpy array indexing to access the pore or throat locations of choice:

```
>>> import OpenPNM
>>> import scipy as sp
>>> pn = OpenPNM.Network.Cubic(shape=[3, 3, 3])
>>> pn['pore.coords'][1]
array([ 0.5,  0.5,  1.5])
```

Note that `pn['pore.coords']` retrieves the Numpy array from the dictionary, while the `[1]` retrieves the value in element 1 of the Numpy array.

Writing data is straightforward:

```
>>> pn['pore.foo'] = 1.0
>>> pn['pore.foo'][5]
1.0
>>> pn['pore.foo'][6] = 2.0
>>> pn['pore.foo'][6]
2.0
>>> pn['pore.foo'][5]
1.0
```

The above lines illustrate how a scalar value is converted to a vector (*Np*-long), and how specific pore values can be assigned. It is also possible to assign an entire array in one step:

```
>>> pn['pore.bar'] = sp.rand(27)  # pn has 27 pores (3*3*3)
```

Attempts to write an array of the wrong size will result in an error:

```
>>> pn['pore.baz'] = [2, 3, 4]
```

To quickly see a complete list *properties* on an object use the `props` method. You can specify whether only *pore* or *throat* properties should be returned, but the default is both:

```
>>> pn.props()
['pore.bar', 'pore.coords', 'pore.foo', 'pore.index', 'throat.conns']
>>> pn.props('throat')
['throat.conns']
```

You can also view a nicely formatted list of `props` with `print(pn.props())`.

Labels

Labels are a means of dynamically creating groups of pores and throats so they can be quickly accessed by the user. For instance, is helpful to know which pores are on the *'top'* surface. This label is automatically added by the *Cubic* network generator, so a list of all pores on the *'top'* can be retrieved by simply querying which pores possess the label *'top'* using the `pores` method:

```
>>> pn.pores('top')
array([ 2,  5,  8, 11, 14, 17, 20, 23, 26])
```

The only distinction between *labels* and *properties* is that *labels* are Boolean masks of True/False. Thus a `True` in element 10 of the array `'pore.top'` means that the label *'top'* has been applied to pore 10. Adding and removing existing labels to pores and throats is simply a matter of setting the element to `True` or `False`. For instance, to remove the label *'top'* from pore 2:

```
>>> pn['pore.top'][2] = False
>>> sp.where(pn['pore.top'])[0]
array([ 5,  8, 11, 14, 17, 20, 23, 26])
>>> pn['pore.top'][2] = True  # Re-apply label to pore 2
```

Creating a new label array occurs automatically if a Boolean array is stored on an object:

```
>>> pn['pore.dummy_1'] = sp.rand(27) < 0.5
```

A complication arises if you have a list of pore numbers you wish to label, such as [3, 4, 5]. You must first create the label array with all `False` values, *then* assign `True` to the desired locations:

```
>>> pn['pore.dummy_2'] = False  # Automatically assigns False to every pore
>>> pn['pore.dummy_2'][[3, 4, 5]] = True
>>> pn.pores('dummy_2')
array([3, 4, 5])
```

The *label* functionality basically works by using Scipy's `where` method to return a list of locations where the array is `True`:

```
>>> sp.where(pn['pore.dummy_2'])[0]
array([3, 4, 5])
```

The `pores` and `throats` methods offer several useful enhancements to this approach. For instance, several labels can be queried at once:

```
>>> pn.pores(['top', 'dummy_2'])
array([ 2,  3,  4,  5,  8, 11, 14, 17, 20, 23, 26])
```

And there is also a `mode` argument which can be used to apply *set theory* logic to the returned list:

```
>>> pn.pores(['top', 'dummy_2'], mode='intersection')
array([5])
```

This *set* logic basically retrieves a list of all pores with the label `'top'` and a second list of pores with the label `dummy_2`, and returns the `'intersection'` of these lists, or only pores that appear in both lists.

The `labels` method can be used to obtain a list of all defined labels. This method optionally accepts a list of *pores* or *throats* as an argument and returns only the *labels* that have been applied to the specified locations.

```
>>> pn.labels()
['pore.all', 'pore.back', 'pore.bottom', 'pore.dummy_1', 'pore.dummy_2', 'pore.front', 'pore.internal
```

This results can also be viewed with `print(pn.labels())`.

Note: The Importance of the 'all' Label

All objects are instantiated with a `'pore.all'` and `'throat.all'` label. These arrays are essential to the framework since they are used to define how long the 'pore' and 'throat' data arrays must be. In other words, the `__setitem__` method checks to make sure that any 'pore' array it receives has the same length as `'pore.all'`.

Counts and Indices

One of the most common questions about a network is "*how many pores and throats does it have?*" This can be answered easily with the `num_pores` and `num_throats` methods. Because these methods are used so often, there are also shortcuts: `Np` and `Nt`.

```
>>> pn.num_pores()
27
>>> pn.Np
27
```

It is also possible to *count* only pores that have a certain label:

```
>>> pn.num_pores('top')
9
```

These counting methods actually work by counting the number of `True` elements in the given *label* array.

Another highly used feature is to retrieve a list of pores or throats that have a certain label applied to them, which is of course is the entire purpose of the *labels* concept. To receive a list of pores on the *'top'* of the **Network**:

```
>>> pn.pores('top')
array([ 2,  5,  8, 11, 14, 17, 20, 23, 26])
```

The `pores` and `throats` methods both accept a *'mode'* argument that allows for *set-theory* logic to be applied to the query, such as returning 'unions' and 'intersections' of locations.

Often, one wants a list of *all** pore or throat indices on an object, so there are shortcut methods for this: `Ps` and `Ts`.

Data Exchange Between Objects

Representing Topology

Page Contents

Storage of Topological Connections

As the name suggests, pore network modeling borrows significantly from the fields of network and graph theory. During the development of OpenPNM, it was debated whether existing Python graph theory packages (such as graph-tool and NetworkX) should be used to store the network topology. It was decided that storage of network property data should be simply stored as Numpy ndarrays (see Numpy). In this form the data storage would be very transparent, since all engineers are used to working with arrays (i.e. vectors), and also very efficiently since this allows code vectorization. Fortuitously, around the same time as this discussion, Scipy introduced the compressed sparse graph library, which contains numerous graph theory algorithms that take Numpy arrays as arguments.

The only topology definitions required by OpenPNM are:

1. A throat connects exactly two pores, no more and no less
2. Throats are non-directional, meaning that flow in either direction is equal (note that this restriction might be worth relaxing in a future release)

Other general, but non-essential rules are:

3. Pores can have an arbitrary number of throats, including zero; however, pores with zero throats lead to singular matrices and other problems so should be avoided.
4. Two pores are connected by no more than one throat, unless there is some real physical reason for this. Unintentional duplicate connections impact the rate of mass exchange between pores.

The **GenericNetwork** class has a `check_network_health` method that scans the network for the above criteria as well as a few others and returns a **HealthDict** which lists if any problems were founds, and where.

Sparse Adjacency Matrices In OpenPNM network topology (or connectivity) is stored as an adjacency matrix. An adjacency matrix is a *Np*-by-*Np* 2D matrix. A non-zero value at location (*i*, *j*) indicates that pores *i* and *j* are connected. Describing the network in this general fashion allows OpenPNM to be agnostic to the type of network it describes. Another important feature of the adjacency matrix is that it is highly sparse and can be stored with a variety of sparse storage schemes. OpenPNM stores the adjacency matrix in the 'COO' or 'IJV' format, which essentially stores the coordinates (I,J) and values (V) of the nonzero elements in three separate lists. This approach results in a property which OpenPNM calls `'throat.conns'`; it is an *Nt*-by-2 array that gives the ID number of the two pores on either end of a given throat. The representation of an arbitrary network is shown in following figure. It has 5 pores and 7 throats, and the `'throat.conns'` array contains the (I,J,V) information to describes the adjacency matrix.

Additional Thoughts on Topology Storage

- In pore networks there is generally no difference between traversing from pore *i* to pore *j* or from pore *j* to pore *i*, so a 1 is also found at location (*j*, *i*) and the matrix is symmetrical.
- A symmetrical matrix means that determining which pores are connected directly to a given pore (say *i*) can be accomplished by finding the locations of non-zeros in row *i*.
- Since the adjacency matrix is symmetric, it is redundant to store the entire matrix when only the upper triangular part is necessary. The `'throat.conns'` array only stores the upper triangular information, and *i* is always less than *j*.
- Although this storage scheme is widely known as *IJV*, the `scipy.sparse` module calls this the Coordinate or *COO* storage scheme.
- Some tasks are best performed on other types of storages scheme, such as *CSR* or *LIL*. OpenPNM converts between these internally as necessary, but uses can generate a desired format using the `create_adjacency_matrix` method which accepts the storage type as an argument (i.e. `'csr'`, `'lil'`, etc). For a discussion of sparse storage schemes and the respective merits, see this Wikipedia article.

Performing Network Queries

Querying and inspecting the pores and throats in the **Network** is an important

Manipulating and Altering Topology

blah

Workspace Manager

OpenPNM includes a **Workspace** class that performs many of the functions found in the *menu bar* of a typical application's GUI, such as saving and loading sessions.

The **Workspace** class is a Singleton in Object Oriented Programming Jargon, meaning that only ONE instance can exist at any given time, and moreover, each time a Singleton is instantiated it returns the already existing object if one already exists. This behavior is handy since it means you can instantiate the **Workspace** at any time, from anywhere in your workflow, and you'll have access to the one and only object.

All OpenPNM objects register themselves with this single **Workspace** when they are created, so you can access any existing object via the workspace. Like so many custom classes in Python, the **Workspace** is a *dictionary* and each OpenPNM object is stored by name. For example:

```
>>> import OpenPNM as op
>>> mgr = op.Base.Workspace()
>>> mgr.clear()  # Clear workspace of any pre-existing objects
>>> mgr.keys()
dict_keys([])
>>> pn = op.Network.Cubic(shape=[5, 5, 5], name='foo')
>>> list(mgr.keys())
['foo']
>>> pn2 = op.Network.Cubic(shape=[5, 5, 5], name='bar')
>>> sorted(list(mgr.keys()))
['bar', 'foo']
>>> pn is mgr['foo']  # The object stored as 'foo' actually pn
True
```

The **Workspace** object also tracks the relationships between the OpenPNM Core objects (see class_hierarchy), and the `print` function outputs a formatted list of each simulation's structure:

```
>>> geom = op.Geometry.GenericGeometry(network=pn, name='geom_on_foo')
>>> geom2 = op.Geometry.GenericGeometry(network=pn2, name='geom_on_bar')
```

```
print(mgr)
------------------------------------------------------------
Object:         Name                 (Class)
------------------------------------------------------------
Network:        bar                  (Cubic)
++ Geometry:    geom_on_bar          (GenericGeometry)
------------------------------------------------------------
Object:         Name                 (Class)
------------------------------------------------------------
Network:        foo                  (Cubic)
++ Geometry:    geom_on_foo          (GenericGeometry)
```

A list of all the methods available on the **Workspace** object can be obtained with:

```
>>> methods = [item for item in dir(mgr) if not item.startswith('_')]
```

This list contains the usual *dictionary* methods, plus an assortment of others for specifically managing OpenPNM objects. The sections below will discuss each in more detail.

Save and Load the Entire Workspace

If for some reason you want to save ALL of your current objects, then you can use `mgr.save`. This will create a `.pnm` file in your current working directory with all objects faithfully saved (including data, labels, models, model arguments, relationships, everything.). This is accomplished by the Python Pickle library. If no name is supplied then a name will automatically be generated using the current date and time.

save(**kwargs)	This method is deprecated, use `save_workspace` instead.

Loading a simulation that was previously saved to a file is equally straightforward, using `mgr.load`. Of course, a

file name must be given. When loading a simulation from a file, any objects stored in the **Workspace** will be deleted.

`load`(**kwargs)	This method is deprecated, use `load_workspace` instead.

Note: Saving and loading entire workspaces is really only useful for simulations that take a long time to generate, such as **Delaunay** networks with **Voronoi** geometry. For fast simulations it is just as easy to save the *'.py'* script, then to recreate a whole new simulation on demand.

Warning: **Algorithm** objects are not automatically registered with the **Workspace** when they are created. This is because in some cases algorithms are instantiated inside a *for-loop* which would quickly bloat the size of the *'.pnm'* file. This may change in a future version.

Save and Load Individual Simulations

Instead of saving the entire workspace it is also possible to save individual simulations. For instance, if multiple networks have been defined but only one of them is of interest, then that **Network** along with all the **Geometry**, **Phase**, and **Physics** objects which were associated with it can be saved using `mgr.save_simulation`.

```
>>> pn1 = op.Network.Cubic(shape=[10, 10, 10])
>>> geo = op.Geometry.GenericGeometry(network=pn1, pores=pn1.Ps, throats=pn1.Ts)
>>> pn2 = op.Network.Cubic(shape=[10, 10, 10])
>>> geo2 = op.Geometry.GenericGeometry(network=pn2, pores=pn2.Ps, throats=pn2.Ts)
>>> air = op.Phases.Air(network=pn1)
>>> water = op.Phases.Water(network=pn2)
>>> mgr.save_simulation(network=pn1, filename='first_network.net')
>>> mgr.save_simulation(network=pn2, filename='second_network.net')
```

The above lines create two files in the current working directory called *'first_network.net'* and *'second_network.net'* which contain pn1 and pn2 respectively, along with all objects (ie. **Geometry** and **Phase**) associated with each.

If we now `clear` the **Workspace** object, we can reload each of these simulations:

```
>>> mgr.clear()
>>> mgr.load_simulation('first_network.net')
>>> mgr.load_simulation('second_network.net')
```

When loading multiple 'simulations' into the **Workspace** it does not remove any existing simulations (unlike loading a saved workspace *'.pnm'* file).

The `save_simulation` and `load_simulation` methods are ideal when running large batches of calculations and you want to save the numerical results for later analysis.

Warning: **Algorithm** objects are not automatically registered with the **Workspace** when they are created. This is because in some cases algorithms are instantiated inside a *for-loop* which would quickly bloat the size of the *'.net'* file. This may change in a future version.

Import and Export Data

The **Workspace** manager has methods to `import_data` and `export_data`. These are wrapper methods for the actual methods found in `Utilities.IO`. These wrapper or helper methods accept several arguments that control which type of file is imported or exported. The actual import and export is explained fully in *Importing and Exporting Data*.

Object Lookup

Each OpenPNM Core object that is created is either given or assigned a `name`. This name is used as the dictionary key when the object is saved on the **Workspace** manager, as outlined above. In addition to looking up objects by name, it is also possible to look them up by type using `networks`, `geometry`, `physics`, and `phases`. At present `algorithms` is offered but does not return any objects since **Algorithms** are not registered. Each of these methods returns a *list* of objects of the specified type. The objects in the list can be assigned to variables on the command line for easy access:

```
>>> pn = mgr.networks()[0]
```

Object Manipulation (Purging, Cloning, etc)

Importing and Exporting Data

OpenPNM has functions for importing and exporting data in various formats that are suitable for exchanging data

---

Note: Exporting data to these file formats is **NOT** the recommended way to save and load OpenPNM Simulations, as lots of key information is lost (only numerical values are exported or imported). To save and load simulations, use the methods available on the **Workspace** as described in *Workspace Manager*.

Exporting Data

OpenPNM allows for exporting the data to several formats:

CSV (Comma separated values) is the recommended format as it is widely used by almost all other software tools # MAT (Matlab file) is supported for the obvious reason that Matlab is a very popular choice for post-processing # VTK (Visualization Toolkit) is main format for exporting results to a visualization software (i.e. Paraview)

There are several ways to import and export data. All the import and export classes are stored under `OpenPNM.Utilities.IO`, but there is also `import_data` and `export_data` methods available in the top level of the project's namespace for convenience (i.e. `OpenPNM.export_data`). The **Workspace** object also possess `import_data` and `export_data` methods. All these approaches utilize the classes stored in the **Utilies.IO** module.

Comma Separated Variables CSV files were chosen as the recommended format in OpenPNM due to their simplicity and wide interoperability with virtually all other software. The list-type data storage scheme used in OpenPNM also happens to fit very well in the CSV column-based format.

Exporting data is accomplished by:

```
>>> import OpenPNM
>>>
```

Matfile

Visualization Toolkit

Importing Data

OpenPNM supports more import formats than the export formats listed above. This is to accommodate the diverse range of external packages used to produce networks. New import formats are added as the need arises.

CSV, MAT and VTK are able to import data exported by OpenPNM as well as any other similarly formatted data # Statoil is available specifically to import networks extracted using the maximal ball code of the Blunt group at ICL # NetworkX is able to import networks from the NetworkX python package

Pore Scale Models

Models are one of the most important aspects of OpenPNM, as they allow the user to specify a 'model' for calculating 'pore.volume', rather than just entering numerical values into a `geometry_object['pore.volume']` array for instance. It is mainly through customized models that users can tailor OpenPNM to a specific situation, though OpenPNM includes a variety of pre-written models. These are stored under each Module in a folder called 'models'. For instance, *'Geometry.models.pore_diameter'* contains several methods for calculating pore diameters. For an example on creating custom models see *Customizing OpenPNM*.

There are two special classes defined in OpenPNM for working with *models*: the **ModelsDict** and the **ModelWrapper**. Each of these are explained in the following sections. A schematic of their interrelation with each other and the **Core** object to which they are attached is given below:

Each **Core** object has a `models` attribute which upon instantiation of the **Core** object is filled with an empty **ModelsDict** object. The **ModelsDict** object is designed to store and interact with all *models* on the **Core** object. The **ModelsDict** is a subclass of the Python *dictionary* type, with several features added for dealing specifically with *models*. Each *model* and its associated arguments are wrapped in a **ModelWrapper** dictionary, then added to the **ModelsDict** under the specified *'propname'*. When a model is run the values it produces are automatically stored in the **Core** object's dictionary under the same specified *'propname'*.

ModelsDict: The Collection of All Models

Adding a model to an object is done as follows:

1. A handle to the desired model is retrieved, either from the included OpenPNM model libraries, or from a file containing the users custom models.
2. The model is attached to the target object using `add_model`.

This process is demonstrated by adding a random pore seed model to a **Geometry** object:

```
>>> import OpenPNM
>>> pn = OpenPNM.Network.TestNet()
>>> geom = OpenPNM.Geometry.GenericGeometry(network=pn,pores=pn.Ps,throats=pn.Ts)
>>> mod = OpenPNM.Geometry.models.pore_misc.random  # Get a handle to the desired model
>>> geom.add_model(propname='pore.seed', model=mod, seed=0)
```

The *'propname'* and *'model'* arguments are required by the `add_model` method, and all other arguments such *'seed'* are passed on the model (In this case it specifies the initialization value for the random number generator).

One can inspect all of the models stored on a given **Core** object by typing `'print(geom.models)'` at the command line:

```
------------------------------------------------------------
#     Property Name                 Regeneration Mode
------------------------------------------------------------
1     pore.seed                     normal
------------------------------------------------------------
```

By default, the `add_model` method runs the model and places the data in the Core object's dictionary under the given *'propname'*. It also wraps the handle to the model and all the given parameters into a **ModelWrapper** dictionary (described below), then saves it in the **ModelsDict** under the same *'propname'*. There are several opions for the *regeneration mode* such as *'deferred'* and *'on_demand'*. Each of these is described in the docstring for the `add_model` method.

In order to recalculate the data, the models stored in the **ModelsDict** dictionary must be rerun. This is accomplished with the `regenerate` method. This method takes an optional list of *'propnames'* that should be regenerated. Models are regenerated in the order that they were added to the object so some care must be taken to ensure that changes in property values cascade through the object correctly. The **ModelsDict** class has functions for updating the model order (`reorder`).

ModelWrapper: The Home for Each Individual Model

Each of the models that are added to the **ModelsDict** are first wrapped inside a **ModelWrapper**, which is a subclass of Python's *dictionary* type. Most users will not need to interact with the **ModelWrapper** directly, since most applications only use the `add_model` and `regenerate` methods of the **Core** object. There are a few cases where it is required, such as changing any of the initially set arguments.

When the `add_model` method of the **Core** object is called, it collects all of the arguments that it receives and stores them in a new instance of a **ModelWrapper** under the appropriate argument name (i.e. *'seed'* is stored under `ModelWrapper['seed']`). It is possible to alter any of these values directly, and these changes will be permanent. One can inspect all the arguments and their current values stored in a **ModelWrapper* by entering `'print(geom.models['pore.seed'])'` at the command line.

```
------------------------------------------------------------
OpenPNM.Geometry.models.pore_misc.random
------------------------------------------------------------
Argument Name        Value / (Default)
------------------------------------------------------------
num_range            [0, 1] / ([0, 1])
regen_mode           normal / (---)
seed                 0 / (None)
------------------------------------------------------------
```

Once creation of the **ModelWrapper** is complete, its stored in the **ModelsDict** under the specified *'propname'*. When `regenerate` is called, each of the models stored in the **ModelsDict** is run, in order. When the `run` method on the **ModelWrapper** is called, the handle to the model is retrieved from *'models'*, and it is then called with ALL other entries in the **ModelWrapper** sent as keyword arguments.

Customizing

The OpenPNM framework was designed specifically with extensibility and customization in mind. Every user will apply OpenPNM to a unique problem, and will therefore require unique pore scale models, phase properties, algorithms and so on.

There are two ways to customize OpenPNM. The first is to download the source code and *hack* it. With this approach it is possible to create your own sub-classes, add pore-scale models, define new topology generators, and to add or change OpenPNM methods. The other approach is to install OpenPNM in your Python PATH (using `pip install openpnm`) as with any other package such as Scipy, and write custom functions in a separate 'working directory' rather than in the source code. In this scenario, you can perform all of the same customizations as the first approach, with the exception of changing OpenPNM's native methods (in fact even this is possible, but that's another story). The second approach is the recommended way for several reasons. It avoids accidental changes to the framework, it allows users to keep their 'projects' compartmentalized, and it is much easier for users to contribute their work to OpenPNM project (which we highly encourage) since sections can be 'cut and pasted' or 'merged' into the framework cleanly. The second approach will be explained in detail below.

The following discussions assume that all custom files will be stored in a folder called `my_pnm`, that will be the 'working directory'.

Creating Custom Models

In the working directory, place a file called 'my_models.py' (or any name you wish). This file will be the home of all the custom models that will be created. Models that are in this file can be added to *any* object using the `add_model`

command. The *'models'* mechanism in OpenPNM was designed to be as straight-forward as possible, for users without any experience in object oriented coding. Each model is simply a 'function' definition, with no inheritance, classes or any unnecessary complications.

Let's create a model called 'surface_roughness' that calculates the surface area of a pore accounting for surface roughness, that accepts a single 'roughness parameter' and is a function of pore size. Start by writing a function in the 'my_models.py' file that simply accepts the 'roughness_parameter' and simply returns it:

```
def surface_roughness(roughness_parameter,**kwargs):
    return roughness_parameter
```

The use of the 'kwargs' argument is essential. Without diving into the details, 'kwargs' will collect all arguments that are sent to function but not used. Without this, the function will break since the `add_model` method sends many arguments 'just-in-case'.

Now, we can see this model in action by creating a script in the working directory as follows:

```
import my_models
a = my_models.surface_roughness(roughness_parameter=2.2)
print(a)
2.2
```

The next step is to have this model to calculate something useful. Assume that surface area due to roughness scales with the projected or smooth surface area as some function, which means this function will need access to the 'pore.diameter' information, which is stored on Geometry objects. Thus the relevant Geometry object must be sent as an argument:

```
def surface_roughness(geometry,roughness_parameter,**kwargs):
        P_diam = geometry['pore.diameter']
        projected_area = 4*3.14159*(P_diam/2)**2
        rough_area = projected_area**roughness_parameter
        return rough_area
```

It was noted above that the `add_model` method sent in several extra arguments 'just-in-case'. Among these arguments are the object from which the method is called. Since 'surface_roughness' will be attached to a Geometry object, this function will receive it as 'geometry' automatically. We can now update our script:

```
import OpenPNM
import scipy as sp
import my_models

#Generate a simple Cubic Network
pn = OpenPNM.Network.Cubic(shape=[3,3,3])

#Generate an 'empty' Geometry with no properties
geom = OpenPNM.Geometry.GenericGeometry(network=pn,pores=pn.pores(),throats=pn.throats())

#Assign random pores diameters between 0 and 40
geom['pore.diameter'] = sp.rand(pn.Np,)*40

#Assign model to geometry
geom.add_model(propname='pore.surface_area',
                                model=my_models.surface_roughness,
                                roughness_parameter=2.2)
```

The print-out of 'geom 'reveals that indeed the model has been added:

print(geom)

OpenPNM.Geometry.GenericGeometry: GenericGeometry_4rhgW

 # Properties Valid Values 1 pore.diameter 27 / 27 2 pore.surface_area 27 / 27 ——————————————————————— # Labels Assigned Locations ——————————————————————— 1 pore.all 27 2 throat.all 54 ———————————————————————

The same approach can be used to create models for pore-scale Physics or for calculating fluid properties that are not included with OpenPNM.

Using non-Default Property Names

In the 'surface_roughness' example above, the function assumed that pore diameter data would be found under the 'pore.diameter' dictionary key. If for some reason, there were multiple different definitions of 'pore diameter', then they might be stored as 'pore.diameter_inscribed', and 'pore.diameter_hydraulic', etc. To allow the 'surface_roughness' function to be applied to any arbitrary pore diameter, it should be rewritten as:

```
def surface_roughness(geometry, roughness_parameter, pore_diameter='pore.diameter', **kwargs):
        P_diam = geometry[pore_diameter]
        projected_area = 4*3.14159*(P_diam/2)**2
        rough_area = projected_area**roughness_parameter
        return rough_area
```

Note that *pore_diameter* is now an argument name, which defaults to 'pore.diameter'. Different *pore diameters* can be specified when calling `add_model`:

```
#Assign model to geometry
geom.add_model(propname='pore.surface_area',
                            model=my_models.surface_roughness,
                            pore_diameter = 'pore.diameter_inscribed',
                            roughness_parameter=2.2)
```

All of the models provide with OpenPNM allow for this sort of non-default argument names, and it will make your custom models more general if you follow this practice.

Creating a Customized Subclass

Another important way to customize OpenPNM is to create custom subclasses of the various objects. For instance, OpenPNM comes with a few basic Geometry subclasses that return pore-scale geometric properties representative of various materials, or common fluids. Creating a custom subclass is only slightly more complicated than writing custom models.

Let's create a Geometry subclass that is representative of Berea Sandstone. Start by creating a file in the 'working directly' (assume it's called 'my_geometries'). In this file we need define our 'class', which will inherit from OpenPNM.Geometry.GenericGeometry:

```
import OpenPNM
class BereaSandstone(OpenPNM.Geometry.GenericGeometry):
        def __init__(self,**kwargs):
                super(berea_sandstone,self).__init__(**kwargs)
```

The above is a basic template that is no different than GenericGeometry yet. The important thing to notice here is the the `__init__` of the parent class is invoked using the `super` method. This means that all arguments passed to `BereaSandstone` are bundled into *'kwargs'* and passed to **GenericGeometry**, which will run all of the tasks that are necessary for OpenPNM objects to work, such as registering this custom Geometry with the Network.

The next step is to actually customize the class. In OpenPNM, all the subclasses of Geometry, Phase and Physics are literally just a collection of 'models' with appropriate parameters to reproduce a specific material, fluid or set of physics. The BereaSandstone class then just needs a set of suitable 'models':

```
import OpenPNM
class BereaSandstone(OpenPNM.Geometry.GenericGeometry):
        def __init__(self,**kwargs):
                super(berea_sandstone,self).__init__(**kwargs)

        mod = OpenPNM.Geometry.models.pore_misc.random
        self.add_model(propname='pore.seed',
                                        model=mod)

        mod = OpenPNM.Geometry.models.pore_diameter.sphere
        self.add_model(propname='pore.diameter',
                                        model=mod,
                                        psd_name='weibull_min',
                                        psd_shape=2.5,
                                        psd_loc=4e-4,
                                        psd_scale=4e-4)
```

The first of the above two models creates a property called 'pore.seed', which is just a list of random numbers that will be used to seed the pore size distribution. The second model uses the Scipy.stats package to generate 'pore.diameter' values from the 'weibull_min' distribution using the given parameters.

Unlike Geometry, Phase and Physics objects, a Network object requires more than a collection of calls to `add_model`. The Network object must provide the 'pore.coords' and 'throat.conns' properties. The 'pore.coords' is fairly straightforward, as it's just an Np x 3 list of [x,y,z] coordinates for each pore in the Network. The 'throat.conns' list is much more difficult to produce. This list is an Nt x 2 list of pairs of connected pore, such as [P1,P2]. OpenPNM comes with two main Network classes: Cubic and Delaunay. The Cubic class connects each pore to it's immediate 6 neighbors on a cubic lattice, while the Delaunay class places pores randomly in space and determines connections via a Delaunay tessellation. There are endless possible topology generation schemes that one may wish to develop.

The approach used to subclass GenericGeometry above would also work for Networks, but there is one additional consideration. Every object must have a 'pore.all' and a 'throat.all' array so that they function properly. The Network generation must therefore, produce these two arrays as well as the 'pore.coords' and 'throat.conns' described above.

Algorithms can also be customized as described above. The GenericAlgorithm has a few additional methods that are meant to be implemented by subclasses, such as `return_results`. The intention of this method is to send the pertinent results of a calculation 'out' of the Algorithm object and to the correct object in the simulation. This step is handy, but is not actually necessary. One can of course manually transfer data from an Algorithm to a Phase, for instance with:

```
air['pore.temperature'] = thermal_simulation['pore.T']
```

CHAPTER 2

Example Usage

The following code block illustrates how to use OpenPNM to perform a mercury intrusion porosimetry simulation in just 10 lines:

```
>>> import OpenPNM as op
>>> pn = op.Network.Cubic(shape=[10, 10, 10], spacing=0.0001)
>>> geo = op.Geometry.Stick_and_Ball(network=pn, pores=pn.Ps, throats=pn.Ts)
>>> Hg = op.Phases.Mercury(network=pn)
>>> Air = op.Phases.Air(network=pn)
>>> phys = op.Physics.Standard(network=pn, phase=Hg, pores=pn.Ps, throats=pn.Ts)
>>> MIP = op.Algorithms.Drainage(network=pn)
>>> MIP.setup(invading_phase=Hg, defending_phase=Air)
>>> MIP.set_inlets(pores=pn.pores(['top', 'bottom']))
>>> MIP.run()
```

The network can be visualized in Paraview giving the following:

The drainage curve can be visualized with `MIP.plot_drainage_curve()` giving something like this:

A collection of examples has been started as a new Github repository: OpenPNM-Examples.

CHAPTER 3

Related Links

OpenPNM Homepage	http://openpnm.org
Github is used to host the code	https://www.github.com/PMEAL/OpenPNM
Github is also used as the project's issue and bug tracker	https://www.github.com/PMEAL/OpenPNM/issues
A collection of examples using OpenPNM is available in a separate repository	https://www.github.com/PMEAL/OpenPNM-Examples
Scipy is a major component of OpenPNM	http://www.scipy.org
Anaconda is the most general way to setup a numerical Python environment	https://www.continuum.io/downloads
WinPython is a slightly easier way to use numerical Python on Windows	https://github.com/winpython/winpython
Spyder is the recommended IDE when working with OpenPNM	https://github.com/spyder-ide/spyder
Paraview is suggested for visualizing OpenPNM data	http://www.paraview.org
OpenPNM is offered under an MIT License	http://opensource.org/licenses/MIT