In order to run the Pore Network Model, several alterations need to be made to the OpenPNM v3.0 documentation, shown in red. You can also copy-paste the altered functions from the “Altered OpenPNM functions” folder.

Openpnm – Algorithms – Reactive Transport:

```
@docstr.get_sections(base='ReactiveTransportSettings', sections=['Parameters'])
@docstr.dedent
class ReactiveTransportSettings:
    r"""

    Parameters
    ----------
    %(TransportSettings.parameters)s
    sources : list
        List of source terms that have been added
    relaxation_factor : float (default = 1.0)
        A relaxation factor to control under-relaxation for the quantity
        solving for. Factor approaching 0 leads to improved stability but
        slower simulation. Factor approaching 1 gives fast simulation but
        may be unstable.
    newton_maxiter : int
        Maximum number of iterations allowed for the nonlinear solver to
        converge.
    f_rtol : float
        Relative tolerance for the solution residual
    x_rtol : float
        Relative tolerance for the solution vector

    """
    relaxation_factor = .8
    newton_maxiter = 50
    f_rtol = 1e-6
    x_rtol = 1e-6

    def _run_special(self, solver, x0, verbose=None):
        r"""
        Repeatedly updates ``A``, ``b``, and the solution guess within
        according to the applied source term then calls ``_solve`` to
        solve the resulting system of linear equations.

        Stops when the max-norm of the residual drops by at least
        ``f_rtol``:

            ``norm(R_n) < norm(R_0) * f_rtol``
        AND
```

```
    ``norm(dx) < norm(x) * x_rtol``

where R_i is the residual at ith iteration, x is the solution at
current iteration, and dx is the change in the solution between two
consecutive iterations. ``f_rtol`` and ``x_rtol`` are defined in
the algorithm's settings under: ``alg.settings['f_rtol']``, and
``alg.settings['x_rtol']``, respectively.

Parameters
----------
x0 : ndarray
    Initial guess of the unknown variable

"""
w = self.settings['relaxation_factor']
maxiter = self.settings['newton_maxiter']
f_rtol = self.settings['f_rtol']
x_rtol = self.settings['x_rtol']
xold = self.x
dx = self.x - xold
condition = TerminationCondition(f_rtol=f_rtol, x_rtol=x_rtol)

tqdm_settings = {
    "total": 100,
    "desc": f"{self.name} : Newton iterations",
    "disable": not verbose,
    "file": sys.stdout,
    "leave": False
}

with tqdm(**tqdm_settings) as pbar:
    for i in range(maxiter):
        self.soln.num_iter = i + 1
        res = self._get_residual()
        progress = self._get_progress(res)
        pbar.update(progress - pbar.n)
        is_converged = bool(condition.check(f=res, x=xold, dx=dx))
        if is_converged:
            pbar.update(100 - pbar.n)
            self.soln.is_converged = is_converged
            logger.info(f'Solution converged, residual norm: {norm(res):.4e}')
            return
        super()._run_special(solver=solver, x0=xold, w=w)
        dx = self.x - xold
        xold = self.x
        logger.info(f'Iteration #{i:<4d} | Residual norm: {norm(res):.4e}')

self.soln.is_converged = False
# logger.warning(f"{self.name} didn't converge after {maxiter} iterations")
```

Openpnm – Algorithms – Transport:

```
import logging
import numpy as np
import scipy.sparse.csgraph as spgr
from openpnm.topotools import is_fully_connected
from openpnm.algorithms import Algorithm
from openpnm.utils import Docorator, TypedSet, Workspace
from openpnm.utils import check_data_health
from openpnm import solvers
from ._solution import SteadyStateSolution, SolutionContainer
import Custom_functions_pressure_fitting as cf_pres_fit

  def run(self, solver=None, x0=None, verbose=False):
      r"""
      Builds the A and b matrices, and calls the solver specified in the
      ``settings`` attribute.

      This method stores the solution in the algorithm's ``soln``
      attribute as a ``SolutionContainer`` object. The solution itself
      is stored in the ``x`` attribute of the algorithm as a NumPy array.

      Parameters
      ----------
      x0 : ndarray
          Initial guess of unknown variable

      Returns
      -------
      None

      """
      logger.info('Running Transport')
      if solver is None:
          solver = getattr(solvers, ws.settings.default_solver)()
      # Perform pre-solve validations
      self._validate_settings()
      self._validate_topology_health()
      self._validate_linear_system()
      # Write x0 to algorithm (needed by _update_iterative_props)
      self.x = x0 = np.zeros_like(self.b) if x0 is None else x0.copy()
      self["pore.initial_guess"] = x0
      self._validate_x0()
      # Initialize the solution object
      self.soln = SolutionContainer()
      self.soln[self.settings['quantity']] = SteadyStateSolution(x0)
      self.soln.is_converged = False

      # Extract fitting parameters and update the hydraulic resistance
      net = self.project['net']                    # Obtain network
```

```
        n_fit = net['pore.Fitting_parameter_n'][0]        # Contraction curvature factor
        m_fit = net['pore.Fitting_parameter_m'][0]        # Expansion curvature factor
        Gamma_fit = net['pore.Fitting_parameter_Gamma'][0] # Flow pattern constant (1 = flat velocity
profile, 2 = Parabolic velocity profile)
        Init = net['pore.parameter_Init'][0]              # Hydraulic conductance initialization indicator
        # If Init == 0 > Do not update the hydraulic conductance (i.e. Initial guess of flow rate)
        # If Init == 1 > Update the hydraulic conductance (i.e. iterative scheme)
        if Init == 1:   # Reset self._pure_A to re-assign A instead of copy old values
            self._pure_A = None # See @_build_A(self)
            Hydraulic_conductance_network_new = cf_pres_fit.Total_hydraulic_conductance_inv(net,
                                                                      n_fit,
                                                                      m_fit,
                                                                      Gamma_fit)
            # Assign new hydraulic conductance
            gvals = self.settings['conductance']
            phase = self.project[self.settings.phase]
            phase[gvals] = Hydraulic_conductance_network_new

        # Build A and b, then solve the system of equations
        self._update_A_and_b()
        self._run_special(solver=solver, x0=x0, verbose=verbose)

    def _run_special(self, solver, x0, w=0.5, verbose=None):

        # Make sure A,b are STILL well-defined
        self._validate_topology_health()
        self._validate_linear_system()
        # Solve and apply under-relaxation
        x_new, exit_code = solver.solve(A=self.A, b=self.b, x0=x0)
        self.x = w * x_new + (1 - w) * self.x

        # Update SteadyStateSolution object on algorithm (placed from bottom to here: update pressure field
to recompute hydraulic conductance)
        self.soln[self.settings['quantity']][:] = self.x
        phase = self.project[self.settings.phase]
        phase['pore.pressure'] = self.x.copy()

        # Extract fitting parameters and update the hydraulic resistance
        net = self.project['net']                          # Obtain network
        n_fit = net['pore.Fitting_parameter_n'][0]        # Contraction curvature factor
        m_fit = net['pore.Fitting_parameter_m'][0]        # Expansion curvature factor
        Gamma_fit = net['pore.Fitting_parameter_Gamma'][0] # Flow pattern constant (1 = flat velocity
profile, 2 = Parabolic velocity profile)
        Init = net['pore.parameter_Init'][0]              # Hydraulic conductance initialization indicator
        # If Init == 0 > Do not update the hydraulic conductance (i.e. Initial guess of flow rate)
        # If Init == 1 > Update the hydraulic conductance (i.e. iterative scheme)
        if Init == 1:   # Reset self._pure_A to re-assign A instead of copy old values
            self._pure_A = None # See @_build_A(self)
            Hydraulic_conductance_network_new = cf_pres_fit.Total_hydraulic_conductance_inv(net,
                                                                      n_fit,
                                                                      m_fit,
```

```
                                                    Gamma_fit)
    # Assign new hydraulic conductance
    gvals = self.settings['conductance']
    # phase = self.project[self.settings.phase]
    phase[gvals] = Hydraulic_conductance_network_new

# Update A and b using the recent solution otherwise, for iterative
# algorithms, residual will be incorrectly calculated ~0, since A & b
# are outdated
self._update_A_and_b()
self.soln.is_converged = not bool(exit_code)
```