In order to run the Genetic Algorithm coupled to the Pore Network Model, several alterations need to be made to the OpenPNM v2.6 documentation.

The changes are shown in red, and the surface area alteration for the extracted network is shown in green.

Openpnm - Topotools:

```
def merge_pores(network, pores, geo, labels=['merged']):
    r"""
    Combines a selection of pores into a new single pore located at the
    centroid of the selected pores and connected to all of their neighbors.

    Parameters
    ----------
    network : OpenPNM Network Object

    pores : array_like
        The list of pores which are to be combined into a new single pore

    labels : string or list of strings
        The labels to apply to the new pore and new throat connections

    Notes
    -----
    (1) The method also works if a list of lists is passed, in which case
    it consecutively merges the given selections of pores.

    (2) The selection of pores should be chosen carefully, preferrable so that
    they all form a continuous cluster.  For instance, it is recommended
    to use the ``find_nearby_pores`` method to find all pores within a
    certain distance of a given pore, and these can then be merged without
    causing any abnormal connections.

    Examples
    --------
    >>> import openpnm as op
    >>> pn = op.network.Cubic(shape=[20, 20, 1])
    >>> Ps = pn.find_nearby_pores(pores=111, r=5, flatten=True)
    >>> op.topotools.merge_pores(network=pn, pores=Ps, labels=['merged'])
    >>> print(pn.Np)
    321
    >>> pn.pores('merged')
    array([320])
    >>> pn.num_throats('merged')
    32
```

```
    """
    # Assert that `pores` is list of lists
    try:
        len(pores[0])
    except (TypeError, IndexError):
        pores = [pores]

    N = len(pores)
    NBs, XYZs = [], []

    for Ps in pores:
        temp = network.find_neighbor_pores(pores=Ps, mode='union', flatten=True,
                                           include_input=False)
        NBs.append(temp)
        points = np.concatenate((temp, Ps))
        # XYZs.append(hull_centroid(network["pore.coords"][points], project, network))
        XYZs.append(hull_centroid(geo, network, pores))

    extend(network, pore_coords=XYZs, labels=labels)
    Pnew = network.Ps[-N::]

    # Possible throats between new pores: This only happens when running in
    # batch mode, i.e. multiple groups of pores are to be merged. In case
    # some of these groups share elements, possible throats between the
    # intersecting elements is not captured and must be added manually.
    pores_set = [set(items) for items in pores]
    NBs_set = [set(items) for items in NBs]
    ps1, ps2 = [], []
    from itertools import combinations
    for i, j in combinations(range(N), 2):
        if not NBs_set[i].isdisjoint(pores_set[j]):
            ps1.append([network.Ps[-N+i]])
            ps2.append([network.Ps[-N+j]])

    # Add (possible) connections between the new pores
    connect_pores(network, pores1=ps1, pores2=ps2, labels=labels)
    # Add connections between the new pores and the rest of the network
    connect_pores(network, pores2=np.split(Pnew, N), pores1=NBs, labels=labels)
    # Trim merged pores from the network
    trim(network=network, pores=np.concatenate(pores))

# def hull_centroid(points):
#     r"""
#     Computes centroid of the convex hull enclosing the given coordinates.

#     Parameters
#     ----------
#     points : Np by 3 ndarray
#         Coordinates (xyz)

#     Returns
```

```
#     -------
#     centroid : array
#         A 3 by 1 Numpy array containing coordinates of the centroid.

#     """
#     dim = [np.unique(points[:, i]).size != 1 for i in range(3)]
#     hull = ConvexHull(points[:, dim])
#     centroid = points.mean(axis=0)
#     centroid[dim] = hull.points[hull.vertices].mean(axis=0)

#     return centroid

def hull_centroid(geo, network, pores):

    # ONLY VALID WHEN MERGED IN TWO PORES ONLY!

    coords = network['pore.coords'][pores[0]]
    x_coord = (coords[0,0] + coords[1,0])/2
    y_coord = (coords[0,1] + coords[1,1])/2
    z_coord = (coords[0,2] + coords[1,2])/2
    centroid = [x_coord, y_coord, z_coord]

    return centroid
```

Openpnm – Algorithms – Reactive Transport:

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

    Parameters
    ----------
    x0 : ND-array
        Initial guess of unknown variable
    """
    self._validate_settings()
    # Check if A and b are well-defined
    self._validate_data_health()
    quantity = self.settings['quantity']
    logger.info('Running ReactiveTransport')
    x0 = np.zeros(self.Np, dtype=float) if x0 is None else x0
    self["pore.initial_guess"] = x0
    x, y = self._run_reactive(x0)
    self[quantity] = x

    return x, y

@docstr.dedent

def _run_reactive(self, x0):
    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 residual falls below ``solver_tol * norm(b)`` or when
    the maximum number of iterations is reached.

    Parameters
    ----------
    x0 : ND-array
        Initial guess of unknown variable

    Returns
    -------
    x : ND-array
        Solution array.

    Notes
    -----
    The algorithm must at least complete one iteration, and hence the check for
    itr >= 1, because otherwise, _check_for_nans() never get's called in case
    there's something wrong with the data, and therefore, the user won't get
```

notified about the root cause of the algorithm divergence.

"""
```
w = self.settings['relaxation_quantity']
quantity = self.settings['quantity']
max_it = self.settings['nlin_max_iter']
# Write initial guess to algorithm obj (for _update_iterative_props to work)
self[quantity] = x = x0
# Update A and b based on self[quantity]
self._update_A_and_b()
# Just in case you got a lucky guess, i.e. x0!
if self._is_converged():
    logger.info(f'Solution converged: {self._get_residual():.4e}')
    y = True
        return x, y

for itr in range(max_it):
    # Solve, use relaxation, and update solution on algorithm obj
    self[quantity] = x = self._solve(x0=x) * w + x * (1 - w)
    self._update_A_and_b()
    # Check solution convergence
    if self._is_converged():
        logger.info(f'Solution converged: {self._get_residual():.4e}')
        y = True
        return x, y
    logger.info(f'Tolerance not met: {self._get_residual():.4e}')

if not self._is_converged():
    # raise Exception(f"Not converged after {max_it} iterations.")
    y = False
    return x, y
```

Openpnm – Algorithms – Generic Transport:

```
def _solve(self, A=None, b=None, x0=None):
    r"""
    Sends the A and b matrices to the specified solver, and solves for *x*
    given the boundary conditions, and source terms based on the present
    value of *x*.  This method does NOT iterate to solve for non-linear
    source terms or march time steps.

    Parameters
    ----------
    A : sparse matrix
        The coefficient matrix in sparse format. If not specified, then
        it uses  the ``A`` matrix attached to the object.
    b : ND-array
        The RHS matrix in any format.  If not specified, then it uses
        the ``b`` matrix attached to the object.
    x0 : ND-array
        The initial guess for the solution of Ax = b

    Notes
    -----
    The solver used here is specified in the ``settings`` attribute of the
    algorithm.

    """
    x0 = np.zeros_like(self.b) if x0 is None else x0

    # Fetch A and b from self if not given, and throw error if not found
    A = self.A if A is None else A
    b = self.b if b is None else b
    if A is None or b is None:
        raise Exception('The A matrix or the b vector not yet built.')
    A = A.tocsr()

    # Check if A and b are STILL well-defined
    self._validate_data_health()

    # Check if A is symmetric
    if self.settings['solver_type'] == 'cg':
        is_sym = op.utils.is_symmetric(self.A)
        if not is_sym:
            raise Exception('CG solver only works on symmetric matrices.')

    # Fetch additional parameters for iterative solvers
    max_it = self.settings["solver_max_iter"]
    atol = self._get_atol()
    rtol = self._get_rtol(x0=x0)

    # Fetch solver object based on settings dict.
```

```
    solver = self._get_solver()
    x = solver(A, b, atol=atol, rtol=rtol, max_it=max_it, x0=x0)

    # Check solution convergence
    # if not self._is_converged(x=x):
    #     raise Exception("Solver did not converge.")

    return x

def _get_solver(self):
    r"""
    Fetch solver object based on solver settings stored in settings dict.

    Notes
    -----
    The returned object can be called via ``obj.solve(A, b, x0[optional])``

    """
    # SciPy
    if self.settings['solver_family'] == 'scipy':
        def solver(A, b, atol=None, rtol=None, max_it=None, x0=None):
            r"""
            Wrapper method for scipy sparse linear solvers.
            """
            ls = getattr(scipy.sparse.linalg, self.settings['solver_type'])
            if self.settings["solver_type"] == "spsolve":
                x = ls(A=A, b=b)
            else:
                tol = self.settings["solver_tol"]
                x, _ = ls(A=A, b=b, atol=atol, tol=tol, maxiter=max_it, x0=x0)
            return x
    # PETSc
    elif self.settings['solver_family'] == 'petsc':
        def solver(A, b, atol=None, rtol=None, max_it=None, x0=None):
            r"""
            Wrapper method for PETSc sparse linear solvers.
            """
            from openpnm.utils.petsc import PETScSparseLinearSolver as SLS
            temp = {"type": self.settings["solver_type"],
                    "preconditioner": self.settings["solver_preconditioner"]}
            ls = SLS(A=A, b=b, settings=temp)
            x = ls.solve(x0=x0, atol=atol, rtol=rtol, max_it=max_it)
            return x
    # PyAMG
    elif self.settings['solver_family'] == 'pyamg':
        def solver(A, b, rtol=None, max_it=None, x0=None, **kwargs):
            r"""
            Wrapper method for PyAMG sparse linear solvers.
            """
            import pyamg
            ml = pyamg.smoothed_aggregation_solver(A)
```

```
            x = ml.solve(b=b, x0=x0, tol=rtol, maxiter=max_it, accel="bicgstab")
            return x
        # PyPardiso
        elif self.settings['solver_family'] == 'pypardiso':
            try:
                import pypardiso
            except ModuleNotFoundError:
                if self.Np <= 8000:
                    # logger.critical("Pardiso not found, reverting to much "
                    #                 + "slower spsolve.  Install pardiso with: "
                    #                 + "conda install -c conda-forge pardiso4py")
                    self.settings['solver_family'] = 'scipy'
                    return self._get_solver()
                else:
                    raise Exception("Pardiso not found. Install it with: "
                                    + "conda install -c conda-forge pardiso4py")

            def solver(A, b, **kwargs):
                r"""
                Wrapper method for PyPardiso sparse linear solver.
                """
                x = pypardiso.spsolve(A=A, b=b)
                return x
        else:
            raise Exception(f"{self.settings['solver_family']} not available.")

        return solver

    def _is_converged(self, x=None):
        r"""
        Check if solution has converged based on the following criterion:
            res <= max(norm(b) * tol, atol)
        """
        res = self._get_residual(x=x)
        # Verify that residual is finite (i.e. not inf/nan)
        if not np.isfinite(res):
            # logger.error(f'Solution diverged: {res:.4e}')
            # raise Exception(f"Solution diverged, undefined residual: {res:.4e}")
            flag_converged = False
        # Check convergence
        tol = self.settings["solver_tol"]
        res_tol = norm(self.b) * tol
        flag_converged = True if res <= res_tol else False
        return flag_converged
```

Openpnm – Models – Geometry:

```
r"""
This submodule contains pore-scale models that calculate geometrical
properties. These models are to be added to a Geometry object.
"""

from . import pore_size
from . import pore_seed
from . import pore_volume
from . import pore_surface_area
from . import pore_cross_sectional_area
from . import throat_endpoints
from . import throat_cross_sectional_area
from . import throat_seed
from . import throat_size
from . import throat_length
from . import throat_perimeter
from . import throat_surface_area
from . import throat_volume
from . import throat_capillary_shape_factor
from . import throat_centroid
from . import throat_vector

# Up for deprecation
pore_area = pore_cross_sectional_area
throat_area = throat_cross_sectional_area


def pore_label(target):
    for j in range(len(target)):
        target = j

    return target
```

Openpnm – Models – Geometry – Pore Surface Area:

```
def sphere(
    target,
    pore_diameter='pore.diameter',
    throat_cross_sectional_area='throat.cross_sectional_area',
):
    r"""
    Calculates internal surface area of pore bodies assuming they are
    spherical then subtracts the area of the neighboring throats in a
    crude way, by simply considering the throat cross-sectional area, thus
    not accounting for the actual curvature of the intersection.

    Parameters
    ----------
    target : GenericGeometry
        The Geometry object for which these values are being calculated.
        This controls the length of the calculated array, and also
        provides access to other necessary thermofluid properties.
    pore_diameter : str
        The dictionary key to the pore diameter array.
    throat_cross_sectional_area : str
        The dictionary key to the throat cross sectional area array.
        Throat areas are needed since their insection with the pore are
        removed from the computation.

    Returns
    -------
    value : NumPy ndarray
        Array containing pore surface area values.

    """
    network = target.project.network
    R = target[pore_diameter] / 2
    Asurf = 4 * _np.pi * R**2
    Tn = network.find_neighbor_throats(pores=target.Ps, flatten=False)
    Tsurf = _np.array([network[throat_cross_sectional_area][Ts].sum() for Ts in Tn])
    value = Asurf # - Tsurf
    return value
```