EPA United States Environmental Protection Agency

EPA/600/R-20/185 | September 2020
www.epa.gov/homeland-security-research

Water Network Tool for Resilience (WNTR) User Manual, Version 0.2.3

Office of Research and Development
Homeland Security Research Program

Water Network Tool for Resilience (WNTR) User Manual

Version 0.2.3

by

Katherine Klise, David Hart, Michael Bynum, Joseph Hogge

Sandia National Laboratories

Albuquerque, New Mexico 87185

Terranna Haxton, Regan Murray, Jonathan Burkhardt

U.S. Environmental Protection Agency

Cincinnati, OH 45268

Interagency Agreement DW8992450201

Terranna Haxton

U.S. Environmental Protection Agency Project Officer

Homeland Security Research Program

Cincinnati, OH 45268

Contents

Disclaimer

The United States Environmental Protection Agency through its Office of Research and Development funded and collaborated in the research described here under an Interagency Agreement # DW8992450201 with the Department of Energy's Sandia National Laboratories. It has been subjected to the Agency's review and has been approved for publication. Note that approval does not signify that the contents necessarily reflect the views of the Agency. Mention of trade names products, or services does not convey official EPA approval, endorsement, or recommendation. The contractor role did not include establishing Agency policy.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

List of Tables

List of Figures

Abbreviations

API: Application programming interface

CSV: Comma-separated values

EPA: Environmental Protection Agency

HDF: Hierarchical Data Format

IDE: Integrated development environment

JSON: JavaScript Object Notation

SI: International System of Units

SQL: Structured Query Language

US: United States

WNTR: Water Network Tool for Resilience

Acknowledgements

WNTR is developed through a collaboration between U.S. Environmental Protection Agency, Sandia National Laboratories, and the open-source community. The U.S. Environmental Protection Agency acknowledges the additional contributors to the WNTR tool, which can be found on the WNTR GitHub site at https://github.com/USEPA/WNTR/graphs/contributors.

The U.S. Environmental Protection Agency acknowledges the technical review of the WNTR software and user manual and/or technical editing provided by the following individuals:

- Eun Jeong Cha, University of Illinois
- Sudhir Kshirsagar, Global Quality Corp
- Lu Liu, Rice University
- Lina Sela, University of Texas at Austin
- Marti Sinclair, Alion Science and Technology, for Attain
- Riccardo Taormina, Delft University of Technology

1 Overview

Drinking water systems face multiple challenges, including aging infrastructure, water quality concerns, uncertainty in supply and demand, natural disasters, environmental emergencies, and cyber and terrorist attacks. All of these have the potential to disrupt a large portion of a water system, causing damage to infrastructure and outages to customers. Increasing resilience to these types of hazards is essential to improving water security.

As one of the United States (US) sixteen critical infrastructure sectors, drinking water is a national priority. The National Infrastructure Advisory Council defined infrastructure resilience as "the ability to reduce the magnitude and/or duration of disruptive events. The effectiveness of a resilient infrastructure or enterprise depends upon its ability to anticipate, absorb, adapt to, and/or rapidly recover from a potentially disruptive event" [NIAC09].

Being able to predict how drinking water systems will perform during disruptive incidents and understanding how to best absorb, recover from, and more successfully adapt to such incidents can help enhance resilience. Simulation and analysis tools can help water utilities to explore the capacity of their systems to handle disruptive incidents and guide the planning necessary to make systems more resilient over time [USEPA14].

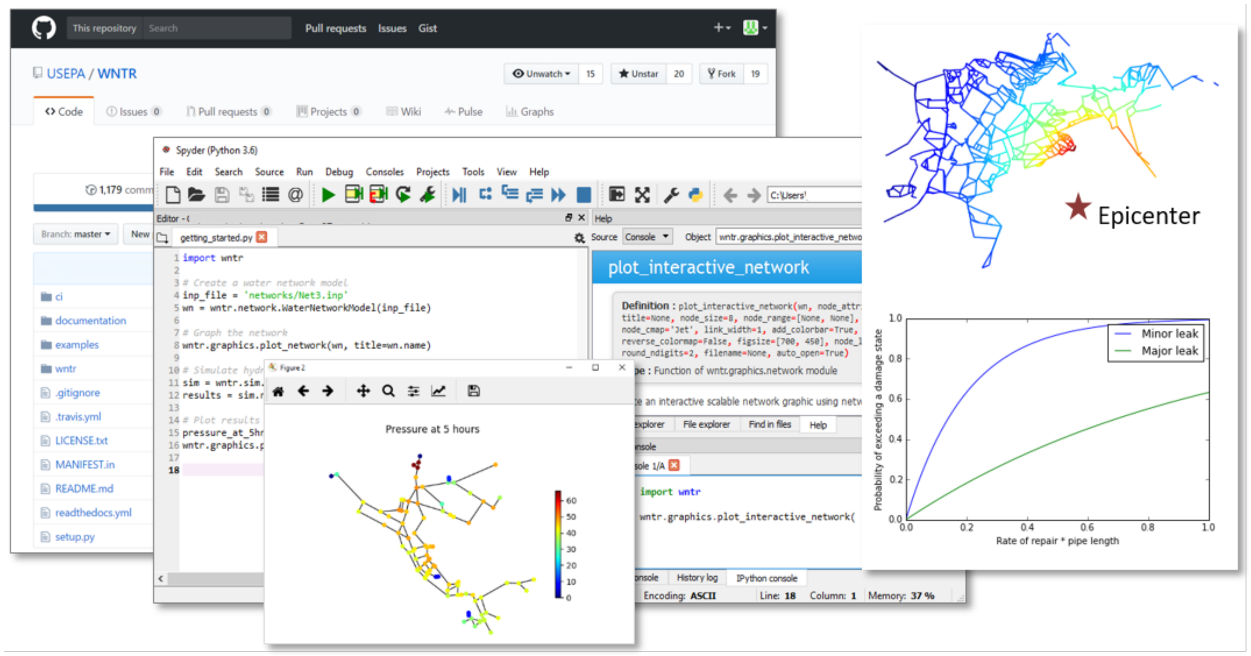
The Water Network Tool for Resilience (WNTR, pronounced *winter*) is a Python package designed to simulate and analyze resilience of water distribution networks. Here, a network refers to the collection of pipes, pumps, valves, junctions, tanks, and reservoirs that make up a water distribution system. WNTR has an application programming interface (API) that is flexible and allows for changes to the network structure and operations, along with simulation of disruptive incidents and recovery actions.

WNTR is based upon EPANET, which is a tool to simulate the movement and fate of drinking water constituents within distribution systems. Users are encouraged to be familiar with the use of EPANET and/or should have background knowledge in hydraulics and pressurized pipe network modeling before using WNTR. EPANET has a graphical user interface that might be a useful tool to facilitate the visualization of the network and the associated analysis results. Information on EPANET can be found at https://www.epa.gov/water-research/epanet. WNTR is compatible with EPANET 2.00.12 [Ross00]. In addition, users should have experience using Python, including the installation of additional Python packages. General information on Python can be found at https://www.python.org/.

WNTR can be installed through the United States Environmental Protection Agency (US EPA) GitHub organization at https://github.com/USEPA/WNTR. An integrated development environment (IDE), like Spyder, is recommended for users and developers. Figure 1 shows the GitHub webpage, Spyder IDE, and sample graphics generated by WNTR.

WNTR includes capabilities to:

- **Generate water network models** from scratch or from existing EPANET-formatted water network model input (EPANET INP) files [Ross00]
- **Modify network structure** by adding/removing components or changing component characteristics
- **Modify network operation** by changing initial conditions, component settings, supply and demand, and time-based and conditional controls
- **Add disruptive incidents** including damage to tanks, valves, and pumps, pipe leaks, power outages, contaminant injection, and environmental changes
- **Add response/repair/mitigation strategies** including leak repair, retrofitted pipes, power restoration, and backup generation
- **Simulate network hydraulics and water quality** using pressure dependent demand or demand-driven hydraulic simulation, and the ability to pause and restart simulations
- **Run probabilistic simulations** using fragility curves for component failure
- **Compute resilience** using topographic, hydraulic, water quality/security, and economic metrics
- **Analyze results and generate graphics** including state transition plots, network graphics, and network animation



```
import wntr

# Create a water network model
inp_file = 'networks/Net3.inp'
wn = wntr.network.WaterNetworkModel(inp_file)

# Graph the network
wntr.graphics.plot_network(wn, title=wn.name)

# Simulate hydr
sim = wntr.sim.
results = sim.r

# Plot results
pressure_at_5hr
wntr.graphics.p

```

```
Definition : plot_interactive_network(wn, node_attr
title=None, node_size=8, node_range=[None, None],
node_cmap='Jet', link_width=1, add_colorbar=True,
reverse_colormap=False, figsize=[700, 450], node_l
round_ndigits=2, filename=None, auto_open=True)
```

```
import wntr

wntr.graphics.plot_interactive_network(
```

Figure 1: WNTR code repository on GitHub, integrated development environment using Spyder, and sample graphics generated by WNTR.

These capabilities can be linked together in many different ways. Figure 2 illustrates four example use cases, from simple to complex.

While EPANET includes some features to model and analyze water distribution system resilience, WNTR was developed to greatly extend these capabilities. WNTR provides a flexible platform for modeling a wide range of disruptive incidents and repair strategies, and pressure dependent demand hydraulic simulation is included to model the system during low pressure conditions. Furthermore, WNTR is compatible with widely used scientific computing packages for Python, including NetworkX [HaSS08], pandas [Mcki13], NumPy [VaCV11], SciPy [VaCV11], and Matplotlib [Hunt07]. These packages allow the user to build custom analysis directly in Python, and gain access to tools that analyze the structure of complex water distribution networks, analyze time-series data from simulation results, run simulations efficiently, and create high-quality graphics and animations.

Note: EPANET refers to EPANET 2.00.12 unless otherwise noted. Future releases of WNTR will include EPANET 2.2.0.

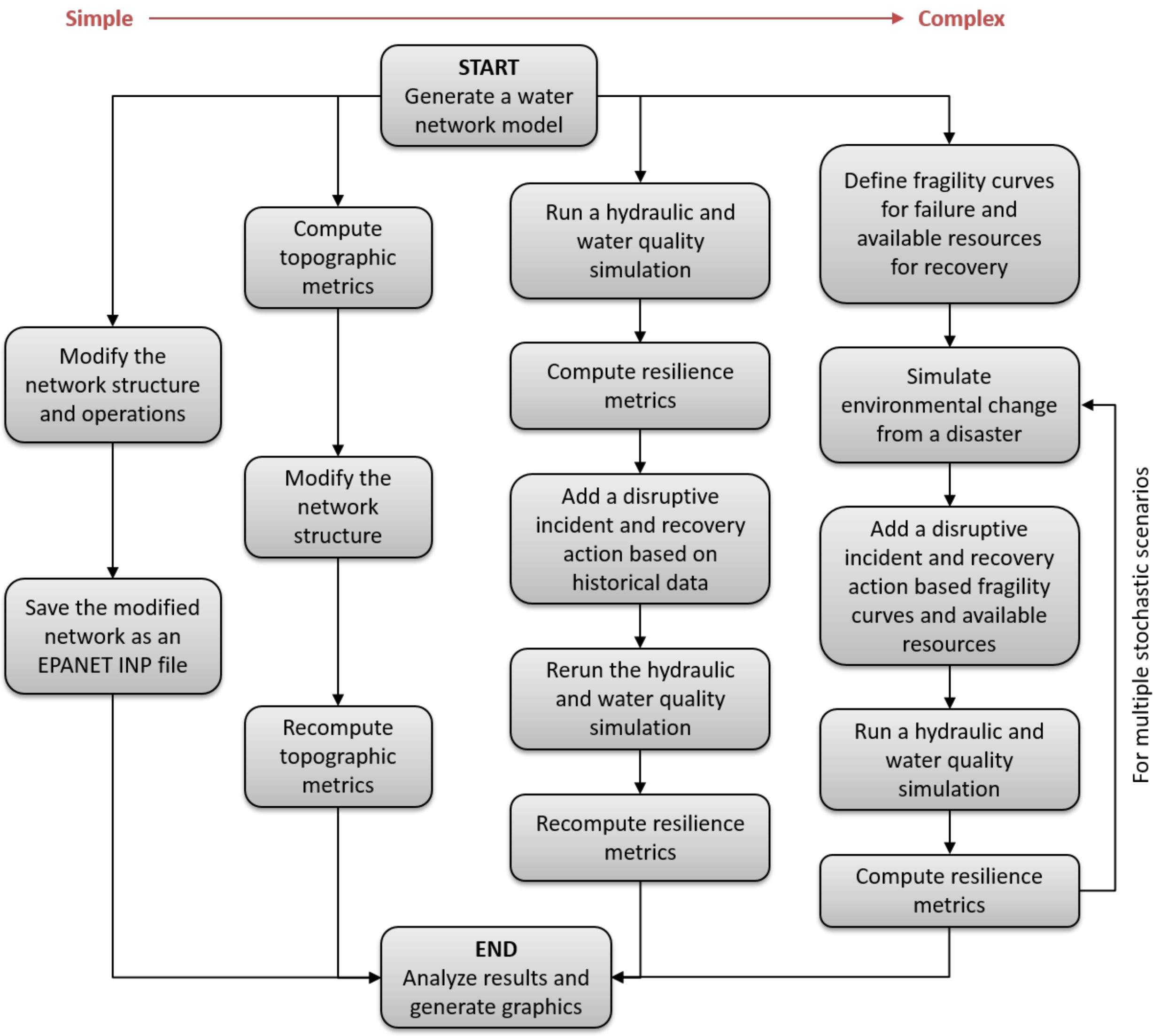

Simple
Complex
START
Generate a water network model
Modify the network structure and operations
Save the modified network as an EPANET INP file
Compute topographic metrics
Modify the network structure
Recompute topographic metrics
Run a hydraulic and water quality simulation
Compute resilience metrics
Add a disruptive incident and recovery action based on historical data
Rerun the hydraulic and water quality simulation
Recompute resilience metrics
Define fragility curves for failure and available resources for recovery
Simulate environmental change from a disaster
Add a disruptive incident and recovery action based fragility curves and available resources
Run a hydraulic and water quality simulation
Compute resilience metrics
For multiple stochastic scenarios
END
Analyze results and generate graphics

Figure 2: Flowchart illustrating four example use cases.

2 Installation

WNTR requires 64-bit Python (tested on versions 3.6 and 3.7) along with several Python package dependencies. See *Requirements* and *Optional dependencies* for more information. WNTR can be installed as a Python package as briefly described below. *Detailed instructions* are included in the following section.

Users can install the latest release of WNTR from PyPI or Anaconda using one of the following commands in a command line or PowerShell prompt.

- PyPI:

  ```
  pip install wntr
  ```

- Anaconda:

  ```
  conda install -c conda-forge wntr
  ```

Developers can install the master branch of WNTR from the GitHub repository using the following commands in a command line or PowerShell prompt:

```
git clone https://github.com/USEPA/WNTR
cd WNTR
python setup.py develop
```

Detailed instructions

Detailed installation instructions are included below.

Step 1: Setup the Python environment

Python can be installed on Windows, Linux, and Mac OS X operating systems. WNTR requires 64-bit Python (tested on versions 3.6 and 3.7) along with several Python package dependencies. Python distributions, such as Anaconda, are recommended to manage the Python environment. Anaconda can be downloaded from https://www.anaconda.com/products/individual. General information on Python can be found at https://www.python.org/.

Note:

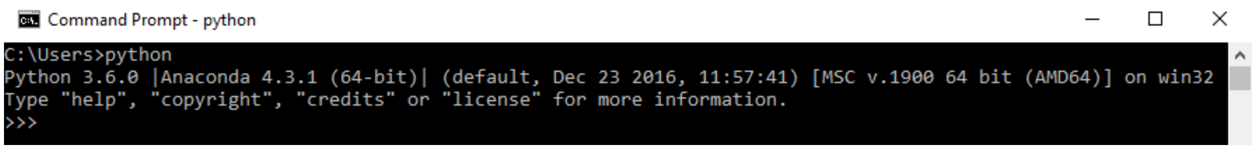
- It is recommended to install Anaconda for a single user by selecting the 'Just Me' option during installation. If a user-writeable location is selected for installation (e.g., C:\Users\username\Anaconda3), then the 'Just Me' option does not require administrator privileges.
- It is also recommended to add Anaconda to the PATH environment variable. This will facilitate access to Python from a command prompt without having to include the full path name. This can be done by either 1) selecting the 'Add Anaconda to my PATH environment variable' option during installation or 2) manually adding C:\Users\username\Anaconda3 to the environmental variables. Note that the first option is not recommended by Anaconda because it elevates the priority of Anaconda software over previously installed software. While the second option allows the user to define priority, this requires administrator privileges. If Anaconda is not added to the PATH environment variable, Python can be run by using the full path name (e.g., C:\Users\username\Anaconda3\python).

Anaconda includes the Python packages needed for WNTR, including NumPy, SciPy, NetworkX, pandas, and Matplotlib. For more information on Python package dependencies, see *Requirements*. If the

Python installation does not include these dependencies, the user will need to install them. This is most commonly done using pip or conda.

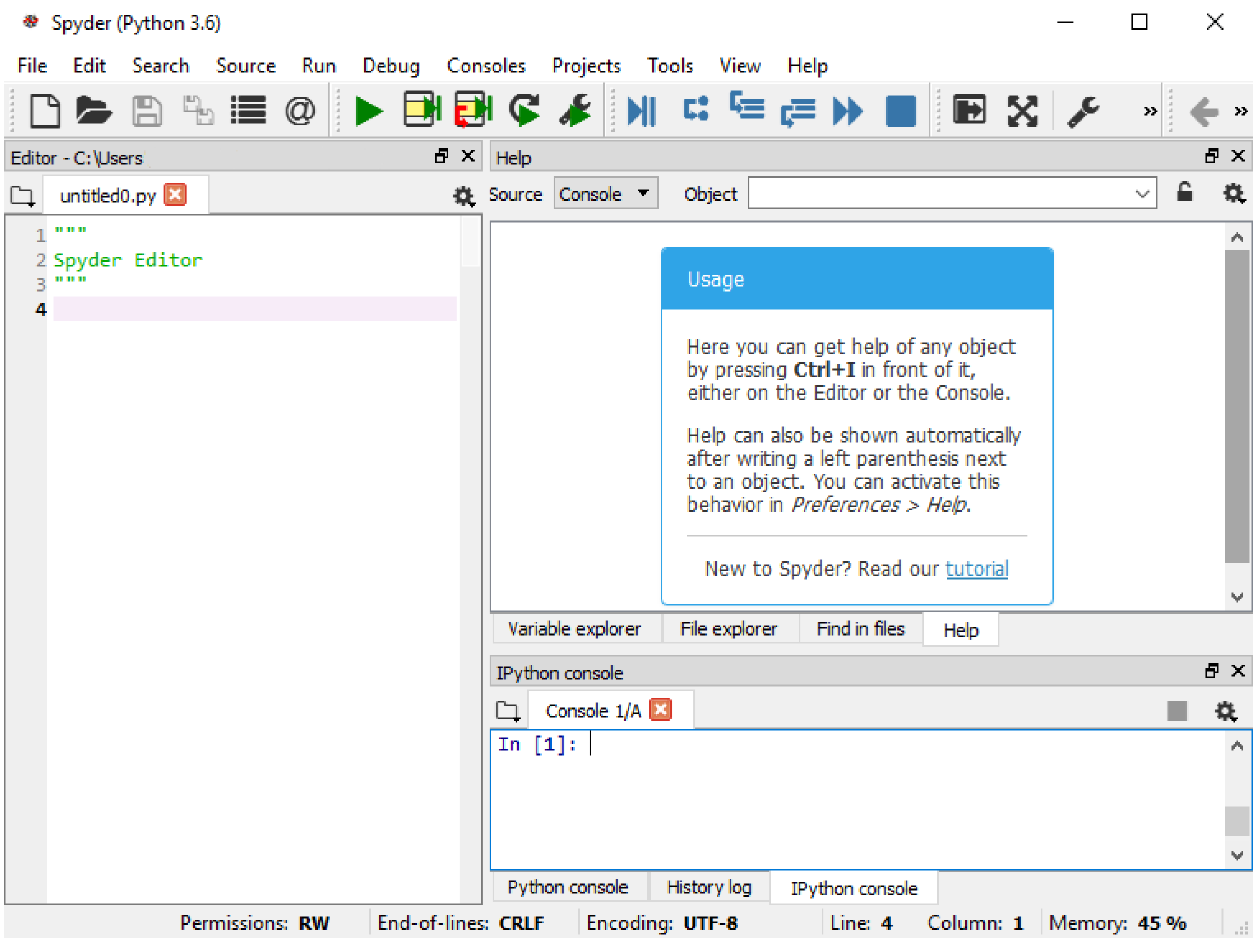
Anaconda also comes with Spyder, an IDE, that includes enhanced editing and debugging features along with a graphical user interface. Debugging options are available from the toolbar. Code documentation is displayed in the object inspection window. Pop-up information on class structure and functions are displayed in the editor and console windows.

To open a Python console, open a command prompt (cmd.exe on Windows, terminal window on Linux and Mac OS X) and run 'python', as shown in Figure 3, or open a Python console using an IDE, like Spyder, as shown in Figure 4.



```
C:\Users>python
Python 3.6.0 |Anaconda 4.3.1 (64-bit)| (default, Dec 23 2016, 11:57:41) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 3: Python console opened from a command prompt.



```
"""
Spyder Editor
"""
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
In [1]:
```

Permissions: **RW** End-of-lines: **CRLF** Encoding: **UTF-8** Line: **4** Column: **1** Memory: **45 %**

Figure 4: Python console using Spyder.

Step 2: Install WNTR

The installation process differs for users and developers. Installation instructions for both types are described below.

For users: Users can install WNTR using PyPI, Anaconda, or by downloading a zip file and building the source code.

Note: If WNTR is installed using PyPI or Anaconda (Options 1 or 2 below), the examples folder will not be downloaded. The examples can be downloaded by going to https://github.com/USEPA/WNTR, select the "Clone or download" button and then select "Download ZIP." Uncompress the zip file using standard software tools (e.g., unzip, WinZip) and store them in a folder.

- **Option 1**: Users can install WNTR from PyPI using pip, which is a command line software tool used to install and manage Python packages. It can be downloaded from https://pypi.python.org/pypi/pip.

  To install WNTR using pip, open a command line or PowerShell prompt and run:

  ```
  pip install wntr
  ```

  This will install the latest release of WNTR from https://pypi.python.org/pypi/wntr.

- **Option 2**: Users can install WNTR from Anaconda using conda, which is a command line software tool used to install and manage Python packages. It can be downloaded from https://www.anaconda.com/products/individual.

  To install WNTR using conda, open a command line or PowerShell prompt and run:

  ```
  conda install -c conda-forge wntr
  ```

  This will install the latest release of WNTR from https://anaconda.org/conda-forge/wntr.

- **Option 3**: Users can download a zip file that includes source files and the examples folder from the US EPA GitHub organization.

  To download the master branch, go to https://github.com/USEPA/WNTR, select the "Clone or download" button and then select "Download ZIP." This downloads a zip file called WNTR-master.zip.

  To download a specific release, go to https://github.com/USEPA/WNTR/releases and select a zip file.

  Uncompress the zip file using standard software tools (e.g., unzip, WinZip) and store them in a folder. WNTR can then be installed by running a Python script, called setup.py, that is included in the source files. To build WNTR from the source files, open a command line or PowerShell prompt from within the folder that contains the files and run:

  ```
  python setup.py install
  ```

For developers: Developers can install and build WNTR from source files using git, which is a command line software tool for version control and software development. It can be downloaded from http://git-scm.com.

To build WNTR from source files using git, open a command line or PowerShell prompt and run:

```
git clone https://github.com/USEPA/WNTR
cd WNTR
python setup.py develop
```

This will install the master branch of WNTR from https://github.com/USEPA/WNTR. More information for developers can be found in the *Software quality assurance* section.

Step 3: Test installation

To test that WNTR is installed, open a Python console and run:

```
import wntr
```

If WNTR is installed properly, Python proceeds to the next line. No other output is printed to the screen.

If WNTR is **not** installed properly, the user will see the following ImportError:

```
ImportError: No module named wntr
```

Requirements

Requirements for WNTR include 64-bit Python (tested on versions 3.6 and 3.7) along with several Python packages. Users should have experience using Python (https://www.python.org/), including the installation of additional Python packages. The following Python packages are required:

- NumPy [VaCV11]: used to support large, multi-dimensional arrays and matrices, http://www.numpy.org/
- SciPy [VaCV11]: used to support efficient routines for numerical integration, http://www.scipy.org/
- NetworkX [HaSS08]: used to create and analyze complex networks, https://networkx.github.io/
- pandas [Mcki13]: used to analyze and store time series data, http://pandas.pydata.org/
- Matplotlib [Hunt07]: used to produce graphics, http://matplotlib.org/

These packages are included in the Anaconda Python distribution.

Optional dependencies

The following Python packages are optional:

- plotly [SPHC16]: used to produce interactive scalable graphics, https://plot.ly/
- folium [Folium]: used to produce Leaflet maps, http://python-visualization.github.io/folium/
- utm [Bieni19]: used to translate node coordinates to utm and lat/long, https://pypi.org/project/utm/
- openpyxl [GaCl18]: used to read/write to Microsoft® Excel® spreadsheets, https://openpyxl.readthedocs.io
- numpydoc [VaCV11]: used to build the user manual, https://github.com/numpy/numpydoc
- nose: used to run software tests, http://nose.readthedocs.io

These packages are included in the Anaconda Python distribution.

3 Software framework and limitations

Before using WNTR, it is helpful to understand the software framework. WNTR is a Python package, which contains several subpackages, listed in Table 1. Each subpackage contains modules that contain classes, methods, and functions. The classes used to generate water network models and run simulations are described in more detail below, followed by a list of software limitations.

See the online API documentation at https://wntr.readthedocs.io for more information on the code structure.

Table 1: WNTR Subpackages

Subpackage	**Description**
`network`	Contains classes and methods to define a water network model, network controls, model options, and graph representation of the network.
`scenario`	Contains classes and methods to define disaster scenarios and fragility/survival curves.
`sim`	Contains classes and methods to run hydraulic and water quality simulations using the water network model.
`metrics`	Contains functions to compute resilience, including hydraulic, water quality, water security, and economic metrics. Methods to compute topographic metrics are included in the wntr.network.graph module.
`morph`	Contains methods to modify water network model morphology, including network skeletonization, modifying node coordinates, and splitting or breaking pipes.
`graphics`	Contains functions to generate graphics.
`epanet`	Contains EPANET 2.00.12 compatibility class and methods for WNTR.
`utils`	Contains helper functions.

Water network model

The `network` subpackage contains classes to define the water network model, network controls, and graph representation of the network. These classes are listed in Table 2. Water network models can be built from scratch or built directly from EPANET INP files. Additionally, EPANET INP files can be generated from water network models.

Table 2: Network Classes

Class	**Description**
`WaterNetworkModel`	Class to generate water network models, including methods to read and write EPANET INP files, and access/add/remove/modify network components. This class links to additional network classes that are listed below to define network components, controls, and model options.
`Junction`	Class to define junctions. Junctions are nodes where links connect. Water can enter or leave the network at a junction.
`Reservoir`	Class to define reservoirs. Reservoirs are nodes with an infinite external source or sink.
`Tank`	Class to define tanks. Tanks are nodes with storage capacity.
`Pipe`	Class to define pipes. Pipes are links that transport water.
`Pump`	Class to define pumps. Pumps are links that increase hydraulic head.
`Valve`	Class to define valves. Valves are links that regulate pressure or flow.
`Curve`	Class to define curves. Curves are data pairs representing a relationship between two quantities. Curves are used to define pump, efficiency, headloss, and volume curves.
`Source`	Class to define sources. Sources define the location and characteristics of a substance injected directly into the network.
`Demands`	Class to define multiple demands per junction. Demands are the rate of withdrawal from the network.
`Pattern`	Class to define patterns. Demands, reservoir heads, pump schedules, and water quality sources can have patterns associated with them.
`Control`	Class to define controls. Controls define a single action based on a single condition.
`Rule`	Class to define rules. Rules can define multiple actions and multiple conditions.
`WaterNetworkOptions`	Class to define model options, including the simulation duration and timestep.

Simulators

The `sim` subpackage contains classes to run hydraulic and water quality simulations using the water network model. WNTR contains two simulators: the EpanetSimulator and the WNTRSimulator. These classes are listed in Table 3.

Table 3: Simulator Classes

Class	**Description**
`EpanetSimulator`	The EpanetSimulator uses the EPANET Programmer's Toolkit [Ross00] to run demand-driven hydraulic simulations and water quality simulations. When using the EpanetSimulator, the water network model is written to an EPANET INP file which is used to run an EPANET simulation. This allows the user to run EPANET simulations, while taking advantage of additional analysis options in WNTR.
`WNTRSimulator`	The WNTRSimulator uses custom Python solvers to run demand-driven and pressure dependent demand hydraulic simulations and includes models to simulate pipe leaks. The WNTRSimulator does not perform water quality simulations, however, the hydraulic simulation results can be used with the EpanetSimulator to perform water quality simulations. See *Water quality simulation* for an example.

Note: EPANET refers to EPANET 2.00.12. Future releases of WNTR will include EPANET 2.2.0.

Limitations

Current software limitations are noted:

- Certain EPANET INP model options are not supported in WNTR, as outlined below.
- Pressure dependent demand hydraulic simulation and leak models are only available using the WNTRSimulator.
- Water quality simulations are only available using the EpanetSimulator.

WNTR reads in and writes all sections of EPANET INP files. This includes the following sections: [BACKDROP], [CONTROLS], [COORDINATES], [CURVES], [DEMANDS], [EMITTERS], [ENERGY], [JUNCTIONS], [LABELS], [MIXING], [OPTIONS], [PATTERNS], [PIPES], [PUMPS], [QUALITY], [REACTIONS], [REPORT], [RESERVOIRS], [RULES], [SOURCES], [TAGS], [TANKS], [TIMES], [TITLE], [VALVES], and [VERTICES].

However, **the following model options cannot be modified/created through the WNTR API**:

- [EMITTERS] section
- [LABELS] section
- [MIXING] section

While the EpanetSimulator uses all EPANET model options, several model options are not used by the WNTRSimulator. Of the EPANET model options that directly apply to hydraulic simulations, **the following options are not supported by the WNTRSimulator**:

- [EMITTERS] section
- D-W and C-M headloss options in the [OPTIONS] section (H-W option is used)
- Accuracy, unbalanced, and emitter exponent from the [OPTIONS] section
- Multipoint curves in the [CURVES] section (3-point curves are supported)
- Pump speed in the [PUMPS] section
- Volume curves in the [TANKS] section
- Pattern start, report start, start clocktime, and statistics in the [TIMES] section
- PBV and GPV values in the [VALVES] section

Future development of WNTR will address these limitations.

Discrepancies

Known discrepancies between the WNTRSimulator and EpanetSimulator are listed below.

- Pumps have speed settings that are adjustable by controls and/or patterns. With the EpanetSimulator, controls and patterns adjust the actual speed. With the WNTRSimulator, pumps have a 'base speed' (similar to junction demand and reservoir head), controls adjust the base speed, and speed patterns are a multiplier on the base speed. Results from the two simulators can match by scaling speed patterns and using controls appropriately.

4 Units

All data in WNTR is stored in the following SI (International System) units:

- Length = m
- Diameter = m
- Water pressure = m (this assumes a fluid density of 1000 kg/m^3)
- Elevation = m
- Mass = kg
- Time = s
- Concentration = kg/m^3
- Demand = m^3/s
- Velocity = m/s
- Acceleration = g (1 g = 9.81 m/s^2)
- Energy = J
- Power = W
- Mass injection = kg/s
- Volume = m^3

When setting up analysis in WNTR, all input values should be specified in SI units. All simulation results are also stored in SI units and can be converted to other units if desired, for instance by using the SymPy Python package [JCMG11].

EPANET unit conventions

WNTR can generate water network models from EPANET INP files using all EPANET unit conventions. When using an EPANET INP file to generate a water network model, WNTR converts model parameters to SI units using the **Units** and **Quality** options of the EPANET INP file. These options define the mass and flow units used in the file. Some units also depend on the equation used for pipe roughness headloss and on the reaction order specified.

For reference, Table 4 includes EPANET unit conventions [Ross00].

Table 4: EPANET INP File Unit Conventions

Parameter	**US customary units**	**SI-based units**
Concentration	*mass* /L where *mass* can be defined as mg or ug	*mass* /L where *mass* can be defined as mg or ug
Demand	Same as *flow*	Same as *flow*
Diameter (Pipes)	in	mm
Diameter (Tanks)	ft	m
Efficiency (Pumps)	percent	percent
Elevation	ft	m
Emitter coefficient	*flow* / sqrt(psi)	*flow* / sqrt(m)
Energy	kW-hours	kW-hours
Flow	• CFS: ft ^{3} /s • GPM: gal/min • MGD: million gal/day • IMGD: million imperial gal/day • AFD: acre-feet/day	• LPS: L/s • LPM: L/min • MLD: million L/day • CMH: m ^{3} /hr • CMD: m ^{3} /day
Friction factor	unitless	unitless
Hydraulic head	ft	m
Length	ft	m
Minor loss coefficient	unitless	unitless
Power	horsepower	kW
Pressure	psi	m
Reaction coefficient (Bulk)	1/day (1st-order)	1/day (1st-order)
Reaction coefficient (Wall)	• *mass* /ft/day (0-order) • ft/day (1st-order)	• *mass* /m/day (0-order) • m/day (1st-order)
Roughness coefficient	• 10 $^{-3}$ ft (Darcy-Weisbach) • unitless (otherwise)	• mm (Darcy-Weisbach) • unitless (otherwise)
Source mass injection rate	*mass* /min	*mass* /min
Velocity	ft/s	m/s
Volume	ft ^{3}	m ^{3}
Water age	hours	hours

5 Getting started

To start using WNTR, open a Python console and import the package:

```
import wntr
```

WNTR comes with a simple getting started example, shown below that uses EPANET Example Network 3 (Net3). This example demonstrates how to:

- Import WNTR
- Generate a water network model
- Simulate hydraulics
- Plot simulation results on the network

Note: If WNTR is installed using PyPI or Anaconda, the examples folder is not included. The examples folder can be downloaded by going to https://github.com/USEPA/WNTR, select the "Clone or download" button and then select "Download ZIP." Uncompress the zip file using standard software tools (e.g., unzip, WinZip) and store them in a folder. The following example assumes the user is running the example from the examples folder.

```
import wntr

# Create a water network model
inp_file = 'networks/Net3.inp'
wn = wntr.network.WaterNetworkModel(inp_file)

# Graph the network
wntr.graphics.plot_network(wn, title=wn.name)

# Simulate hydraulics
sim = wntr.sim.EpanetSimulator(wn)
results = sim.run_sim()

# Plot results on the network
pressure_at_5hr = results.node['pressure'].loc[5*3600, :]
wntr.graphics.plot_network(wn, node_attribute=pressure_at_5hr, node_size=30,
                        title='Pressure at 5 hours')
```

Additional examples are included throughout the WNTR documentation. The examples provided in the documentation assume that a user has experience using EPANET (https://www.epa.gov/water-research/epanet) and Python (https://www.python.org/), including the ability to install and use additional Python packages, such as those listed in *Requirements* and *Optional dependencies*.

Several EPANET INP files and example files are also included in the WNTR repository in the examples folder. Example networks range from a simple 9 node network to a 3,000 node network. Additional network models can be downloaded from the University of Kentucky Water Distribution System Research Database at https://uknowledge.uky.edu/wdsrd.

Additional examples

WNTR comes with additional examples that illustrate advanced use cases, including:

- Pipe leak, stochastic simulation example: This example runs multiple hydraulic simulations of a pipe leak scenario where the location and duration are drawn from probability distributions.
- Pipe criticality example: This example runs multiple hydraulic simulations to compute the impact that individual pipe closures have on water pressure.
- Fire flow example: This example runs hydraulic simulations with and without fire fighting flow demand.
- Sensor placement example: This example uses WNTR with Chama (https://chama.readthedocs.io) to optimize the placement of sensors that minimizes detection time.

6 Water network model

The water network model includes junctions, tanks, reservoirs, pipes, pumps, valves, patterns, curves, controls, sources, simulation options, and node coordinates. Water network models can be built from scratch or built directly from an EPANET INP file. Sections of the EPANET INP file that are not compatible with WNTR are described in *Limitations*. For more information on the water network model, see `WaterNetworkModel` in the API documentation.

Build a model from an INP file

A water network model can be created directly from an EPANET INP file. The following example builds a water network model.

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
```

Note: Unless otherwise noted, examples in the WNTR documentation use Net3.inp to build the water network model, named `wn`.

Add elements

The water network model contains methods to add junctions, tanks, reservoirs, pipes, pumps, valves, patterns, curves, sources, and controls. When an element is added to the model, it is added to the model's registry. Within the registry, junctions, tanks, and reservoirs share a namespace (e.g., those elements cannot share names) and pipes, pumps, and valves share a namespace.

For each method that adds an element to the model, there is a related object. For example, the `add_junction` method adds a `Junction` object to the model. Generally, the object is not added to the model directly.

The example below adds a junction and pipe to a water network model.

```
>>> wn.add_junction('new_junction', base_demand=10, demand_pattern='1', elevation=10,
...     coordinates=(6, 25))
>>> wn.add_pipe('new_pipe', start_node_name='new_junction', end_node_name='101',
...     length=10, diameter=0.5, roughness=100, minor_loss=0)
```

Remove elements

The water network model registry tracks when elements are used by other elements in the model. An element can only be removed if all elements that rely on it are removed or modified. For example, if a valve is used in a control, the valve cannot be removed until the control is removed or modified. Similarly, a node cannot be removed until the pipes connected to that node are removed. The following example removes a link and node from the model. If the element being removed is used by another element, an error message is printed to the screen and the element is not removed.

```
>>> wn.remove_link('new_pipe')
>>> wn.remove_node('new_junction')
```

Modify options

Water network model options are divided into the following categories: time, hydraulics, quality, solver, results, graphics, and energy. The following example returns model options, which all have default values, and then modifies the simulation duration.

```
>>> wn.options
Time options:
  duration               : 604800
  hydraulic_timestep     : 900
  quality_timestep       : 900
  rule_timestep          : 360.0
  pattern_timestep       : 3600
...
>>> wn.options.time.duration = 10*3600
```

Modify element attributes

To modify element attributes, the element object is first obtained using the `get_node` or `get_link` methods. The following example changes junction elevation, pipe diameter, and tank size.

```
>>> junction = wn.get_node('121')
>>> junction.elevation = 5
>>> pipe = wn.get_link('122')
>>> pipe.diameter = pipe.diameter*0.5
>>> tank = wn.get_node('1')
>>> tank.diameter = tank.diameter*1.1
```

The following shows how to add an additional demand to the junction 121.

```
>>> print(junction.demand_timeseries_list)
<Demands: [<TimeSeries: base=0.002626444876132, pattern='1', category='None'>]>

>>> junction.add_demand(base=1.0, pattern_name='1')
>>> print(junction.demand_timeseries_list)
<Demands: [<TimeSeries: base=0.002626444876132, pattern='1', category='None'>,
↪<TimeSeries: base=1.0, pattern='1', category='None'>]>
```

To remove the demand, use the Python `del` as with an array element.

```
>>> del junction.demand_timeseries_list[1]
>>> print(junction.demand_timeseries_list)
<Demands: [<TimeSeries: base=0.002626444876132, pattern='1', category='None'>]>
```

Modify time series

Several network attributes are stored as a time series, including junction demand, reservoir head, and pump speed. A time series contains a base value, a pattern, and a category. Time series are added to the water network model when the junction, reservoir, or pump is added. Since junctions can have multiple demands, junction demands are stored as a list of time series. The following examples modify time series.

Change reservoir supply:

```
>>> reservoir = wn.get_node('River')
>>> reservoir.head_timeseries.base_value = reservoir.head_timeseries.base_value*0.9
```

Change junction demand base value:

```
>>> junction = wn.get_node('121')
>>> junction.demand_timeseries_list[0].base_value = 0.005
```

Add a new demand time series to the junction:

```
>>> pat = wn.get_pattern('3')
>>> junction.demand_timeseries_list.append((0.001, pat))
```

Add custom element attributes

New attributes can be added to model elements simply by defining a new attribute name and value. These attributes can be used in custom analysis and graphics.

```
>>> pipe = wn.get_link('122')
>>> pipe.material = 'PVC'
```

Iterate over elements

Iterators are available for junctions, tanks, reservoirs, pipes, pumps, and valves. Each iterator returns the element's name and the element's object. The following example iterates over all pipes to modify pipe diameter.

```
>>> for pipe_name, pipe in wn.pipes():
...     pipe.diameter = pipe.diameter*0.9
```

Get element names and counts

Several methods are available to return a list of element names and the number of elements, as shown in the example below. The list of element names can be used as an iterator, especially in cases where the element object is not needed.

```
>>> node_names = wn.node_name_list
>>> num_nodes = wn.num_nodes
>>> wn.describe(level=0)
{'Nodes': 97, 'Links': 119, 'Patterns': 5, 'Curves': 2, 'Sources': 0, 'Controls': 18}
```

Query element attributes

The water network model contains methods to query node and link attributes. These methods can return attributes for all nodes or links, or for a subset using arguments that specify a node or link type (i.e., junction or pipe), or by specifying a threshold (i.e., >= 10 m). The query methods return a pandas Series with the element name and value. The following example returns node elevation, junction elevation, and junction elevations greater than 10 m (using a NumPy operator).

```
>>> import numpy as np

>>> node_elevation = wn.query_node_attribute('elevation')
>>> junction_elevation = wn.query_node_attribute('elevation',
...     node_type=wntr.network.model.Junction)
```

(continues on next page)

(continued from previous page)

```
>>> junction_elevation_10 = wn.query_node_attribute('elevation', np.greater_equal,
...     10, node_type=wntr.network.model.Junction)
```

In a similar manner, link attributes can be queried, as shown below.

```
>>> link_length = wn.query_link_attribute('length', np.less, 50)
```

Reset initial conditions

When using the same water network model to run multiple simulations using the WNTRSimulator, initial conditions need to be reset between simulations. Initial conditions include simulation time, tank head, reservoir head, pipe status, pump status, and valve status. When using the EpanetSimualtor, this step is not needed since EPANET starts at the initial conditions each time it is run.

```
>>> wn.reset_initial_values()
```

Write a model to an INP file

The water network model can be written to a file in EPANET INP format. By default, files are written in the LPS (liter per second) EPANET unit convention. The EPANET INP file will not include features not supported by EPANET (i.e., pressure dependent demand simulation options, custom element attributes).

Note: The EPANET referred to here is EPANET 2.00.12, which does not include the pressure dependent algorithm in EPANET 2.2.0.

```
>>> wn.write_inpfile('filename.inp')
```

Build a model from scratch

A water network model can also be created from scratch by adding elements to an empty model. Elements must be added before they are used in a simulation. For example, demand patterns are added to the model before they are used within a junction. The section below includes additional information on adding elements to a water network model.

```
>>> wn = wntr.network.WaterNetworkModel()
>>> wn.add_pattern('pat1', [1])
>>> wn.add_pattern('pat2', [1,2,3,4,5,6,7,8,9,10])
>>> wn.add_junction('node1', base_demand=0.01, demand_pattern='pat1', elevation=100,
...     coordinates=(1,2))
>>> wn.add_junction('node2', base_demand=0.02, demand_pattern='pat2', elevation=50,
...     coordinates=(1,3))
>>> wn.add_pipe('pipe1', 'node1', 'node2', length=304.8, diameter=0.3048,
...    roughness=100, minor_loss=0.0, status='OPEN')
>>> wn.add_reservoir('res', base_head=125, head_pattern='pat1', coordinates=(0,2))
>>> wn.add_pipe('pipe2', 'node1', 'res', length=100, diameter=0.3048, roughness=100,
...     minor_loss=0.0, status='OPEN')
>>> nodes, edges = wntr.graphics.plot_network(wn)
```

7 Water network controls

One of the key features of water network models is the ability to control pipes, pumps, and valves using simple and complex conditions. EPANET uses "controls" and "rules" to define conditions [Ross00]. WNTR replicates EPANET functionality, and includes additional options, as described below. The EPANET user manual provides more information on simple controls and rule-based controls (controls and rules, respectively in WNTR) [Ross00].

Controls are defined using an "IF condition; THEN action" format. Controls use a single action (i.e., closing/opening a link or changing the setting) based on a single condition (i.e., time based or tank level based). If a time based or tank level condition is not exactly matched at a simulation timestep, controls make use of partial timesteps to match the condition before the control is deployed. Controls in WNTR emulate EPANET simple controls.

Rules are more complex; rules are defined using an "IF condition; THEN action1; ELSE action2" format, where the ELSE block is optional. Rules can use multiple conditions and multiple actions in each of the logical blocks. Rules can also be prioritized to set the order of operation. If rules with conflicting actions should occur at the same time, the rule with the highest priority will override all others. Rules operate on a rule timestep specified by the user, which can be different from the simulation timestep. Rules in WNTR emulate EPANET rule-based controls.

When generating a water network model from an EPANET INP file, WNTR generates controls and rules based on input from the [CONTROLS] and [RULES] sections. These controls and rules are then used when simulating hydraulics with either the EpanetSimulator or the WNTRSimulator. Controls and rules can also be defined directly in WNTR using the API described below. WNTR includes additional options to define controls and rules that can be used by the WNTRSimulator.

The basic steps to define a control or rule are:

1. Define the action(s) (i.e., define the action that should occur, such as closing/opening a link)
2. Define condition(s) (i.e., define what should cause the action to occur, such as a tank level)
3. Define the control or rule using the action(s) and condition(s) (i.e., combine the defined action and condition)
4. Add the control or rule to the water network model

These steps are defined below.

See the online API documentation for more information on controls.

Actions

Control and rule actions tell the simulator what to do when a condition becomes "true." Actions are created using the `ControlAction` class. An action is defined by a target link, the attribute to change, and the value to change it to. The following example creates an action that opens pipe 330, in which a status of 1 means open:

```
>>> import wntr
>>> import wntr.network.controls as controls

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> pipe = wn.get_link('330')
>>> act1 = controls.ControlAction(pipe, 'status', 1)
>>> print(act1)
set Pipe('330').status to Open
```

Conditions

Conditions define when an action should occur. The condition classes are listed in Table 5.

Table 5: Condition Classes

Condition class	Description
`TimeOfDayCondition`	Time-of-day or "clocktime" based condition statement
`SimTimeCondition`	Condition based on time since start of the simulation
`ValueCondition`	Compare a network element attribute to a set value
`RelativeCondition`	Compare attributes of two different objects (e.g., levels from tanks 1 and 2)
`OrCondition`	Combine two WNTR conditions with an OR
`AndCondition`	Combine two WNTR conditions with an AND

All of the above conditions are valid EpanetSimulator conditions except `RelativeCondition`. The EpanetSimulator is also limited to always repeat conditions that are defined with `TimeOfDayCondition` and not repeat conditions that are defined with in `SimTimeCondition`. The WNTRSimulator can handle repeat or not repeat options for both of these conditions.

Controls

A control is created in WNTR with the `Control` class, which takes an instance of any of the above conditions, and an action that should occur when the condition is true.

Controls are also assigned a priority. If controls with conflicting actions should occur at the same time, the control with the highest priority will override all others. The priority argument should be an element of the `ControlPriority` class. The default priority is medium (3).

In the following example, a conditional control is defined that opens pipe 330 if the level of tank 1 goes above 46.0248 m (151.0 ft). The target is the tank and the attribute is the tank's level. To specify that the condition should be true when the level is greater than the threshold, the operation is set to > and the threshold is set to 46.0248. The action *act1* from above is used in the control.

```
>>> tank = wn.get_node('1')
>>> cond1 = controls.ValueCondition(tank, 'level', '>', 46.0248)
>>> print(cond1)
Tank('1').level > 46.0248

>>> ctrl1 = controls.Control(cond1, act1, name='control1')
>>> print(ctrl1)
Control control1 := if Tank('1').level > 46.0248 then set Pipe('330').status to Open␣
↪with priority 3
```

In the following example, a time-based control is defined that opens pump 10 at hour 121. A new action is defined that opens the pump. The SimTimeCondition parameter can be specified as decimal hours or as a string in `[dd-]hh:mm[:ss]` format. When printed, the output is converted to seconds.

```
>>> pump = wn.get_link('10')
>>> act2 = controls.ControlAction(pump, 'status', 1)
>>> cond2 = controls.SimTimeCondition(wn, '=', '121:00:00')
>>> print(cond2)
sim_time = 435600 sec

>>> ctrl2 = controls.Control(cond2, act2, name='control2')
```

(continues on next page)

(continued from previous page)

```
>>> print(ctrl2)
Control control2 := if sim_time = 435600 sec then set HeadPump('10').status to Open␣
↪with priority 3
```

Rules

A rule is created in WNTR with the `Rule` class, which takes any of the above conditions, a list of actions that should occur when the condition is true, and an optional list of actions that should occur when the condition is false.

Like controls, rules are also assigned a priority. If rules with conflicting actions should occur at the same time, the rule with the highest priority will override all others. The priority argument should be an element of the `ControlPriority` class. The default priority is medium (3). Priority can only be assigned when the rule is created.

The following examples illustrate the creation of rules, using conditions and actions similar to those defined above.

```
>>> cond2 = controls.SimTimeCondition(wn, controls.Comparison.ge, '121:00:00')

>>> rule1 = controls.Rule(cond1, [act1], name='rule1')
>>> print(rule1)
Rule rule1 := if Tank('1').level > 46.0248 then set Pipe('330').status to Open with␣
↪priority 3

>>> pri5 = controls.ControlPriority.high
>>> rule2 = controls.Rule(cond2, [act2], name='rule2', priority=pri5)
>>> print(rule2)
Rule rule2 := if sim_time >= 435600 sec then set HeadPump('10').status to Open with␣
↪priority 5
```

Since rules operate on a different timestep than controls, these rules might behave differently than the equivalent controls defined above. Controls (or simple controls in EPANET) operate on the hydraulic timestep while Rules (or rule-based controls in EPANET) operate at a smaller timestep. By default, the rule timestep is 1/10th of the hydraulic timestep. It is important to remember that significant differences might occur when timesteps are smaller; this applies not only to rule timesteps, but also to hydraulic or quality timesteps.

More complex rules can be written using one of the Boolean logic condition classes. The following example creates a new rule that will open pipe 330 if both conditions are true, and otherwise it will open pump 10.

```
>>> cond3 = controls.AndCondition(cond1, cond2)
>>> print(cond3)
( Tank('1').level > 46.0248 && sim_time >= 435600 sec )

>>> rule3 = controls.Rule(cond3, [act1], [act2], priority=3, name='complex_rule')
>>> print(rule3)
Rule complex_rule := if ( Tank('1').level > 46.0248 && sim_time >= 435600 sec ) then␣
↪set Pipe('330').status to Open else set HeadPump('10').status to Open with priority␣
↪3
```

Actions can also be combined, as shown in the following example.

```
>>> cond4 = controls.OrCondition(cond1, cond2)
>>> rule4 = controls.Rule(cond4, [act1, act2], name='rule4')
>>> print(rule4)
Rule rule4 := if ( Tank('1').level > 46.0248 || sim_time >= 435600 sec ) then set␣
↪Pipe('330').status to Open and set HeadPump('10').status to Open with priority 3
```

The flexibility of rules provides an extremely powerful tool for defining complex network operations.

Adding controls/rules to a network

Once a control or rule is created, it can be added to the network. This is accomplished using the `add_control` method of the water network model object. The control or rule should be named so that it can be retrieved and modified if desired.

```
>>> wn.add_control('NewTimeControl', ctrl2)
>>> wn.get_control('NewTimeControl')
<Control: 'control2', <SimTimeCondition: model, 'Is', '5-01:00:00', False, 0>, [
↪<ControlAction: 10, status, Open>], [], priority=3>
```

8 NetworkX graph

WNTR can generate a NetworkX data object that stores network connectivity as a graph. The ability to easily integrate NetworkX with WNTR facilitates the use of numerous standard graph algorithms, including algorithms that describe network structure.

A **graph** is a collection of nodes that are connected by links. For water networks, nodes represent junctions, tanks, and reservoirs while links represent pipes, pumps, and valves. The NetworkX graph can be used to analyze network structure.

The type of NetworkX graph generated by WNTR is a directed multigraph. A **directed multigraph** is a graph with direction associated with links and the graph can have multiple links with the same start and end node. A simple example is shown in Figure 5. For water networks, the link direction is from the start node to the end node. The link direction is used as a reference to track flow direction in the network. For example, positive flow indicates that the flow direction is from the start node to the end node while negative flow indicates that the flow direction is from the end node to the start node. Multiple links with the same start and end node can be used to represent redundant pipes or backup pumps.

A NetworkX graph generated from a water network model stores the start and end node of each link, node coordinates, and node and link types (i.e., tank, reservoir, valve). NetworkX includes numerous methods to analyze the structure of complex networks. For more information on NetworkX, see https://networkx.github.io/.

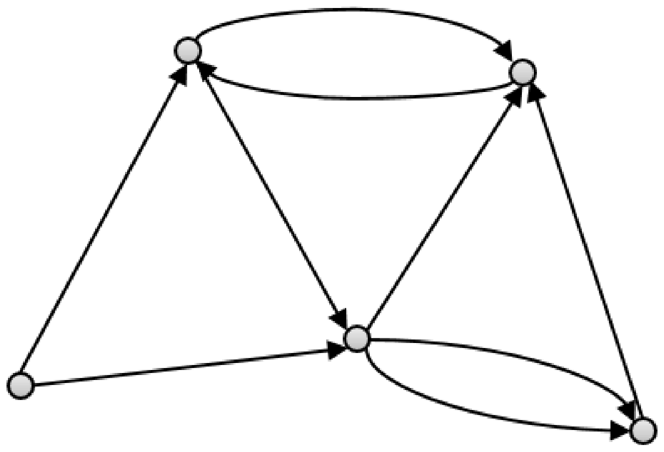


Figure 5: Example directed multigraph.

A NetworkX directed multigraph can an be obtained from a WaterNetworkModel using the following function:

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> G = wn.get_graph() # directed multigraph
```

The graph is stored as a nested dictionary. The nodes and links can be accessed using the graph's *node* and *adj* attribute (*adj* is used to get adjacent nodes and links).

```
>>> node_name = '123'
>>> G.nodes[node_name]
>>> G.adj[node_name]
```

The graph can be used to access NetworkX methods, for example:

```
>>> import networkx as nx

>>> node_degree = G.degree()
>>> closeness_centrality = nx.closeness_centrality(G)
>>> nodes, edges = wntr.graphics.plot_network(wn, node_attribute=closeness_centrality)
```

See *Topographic metrics* for more information.

Additional network types

Some methods in NetworkX require that networks are undirected, connected, weighted, or have only one edge between nodes.

An **undirected graph** is a graph with no direction associated with links. The following NetworkX method can be used to convert a directed graph to an undirected graph:

```
>>> uG = G.to_undirected() # undirected multigraph
```

A **connected graph** is a graph where a path exists between every node in the network (i.e., no node is disconnected). The following NetworkX method can be used to check if a graph is connected:

```
>>> nx.is_connected(uG)
True
```

A **weighted graph** is a graph in which each node and/or link is given a weight. The WNTR method `get_graph` can be used to weight the graph by node and/or link attributes. In the following example, the graph is weighted by length. This graph can then be used to compute path lengths:

```
>>> length = wn.query_link_attribute('length')
>>> wG = wn.get_graph(wn, link_weight=length) # weighted directed multigraph
```

A **simple graph** is a graph with one edge between nodes. The following NetworkX method can be used to convert a multigraph to a simple graph:

```
>>> sG = nx.Graph(G) # directed simple graph
```

9 Data layers

Data layers contain data which are not part of the water network model or graph, but can be used in analysis. Currently, WNTR includes a data format for valve layers; additional data layers can be added in the future.

Valve layer

While valves are typically included in the water network model, the user can also define a valve layer to be used in additional analysis. A valve layer can be used to group links and nodes into segments based on the location of isolation valves. In a valve layer, the location of each valve is defined using the link the valve is installed on and the node the valve protects. This information is stored in a pandas DataFrame, which is indexed by valve number with columns named ‘link’ and ‘node.’ For example, the following valve layer defines a valve on Pipe 1 that protects Junction A (Figure 6).

```
>>> print(valve_layer)
     link        node
0  Pipe 1  Junction A
```

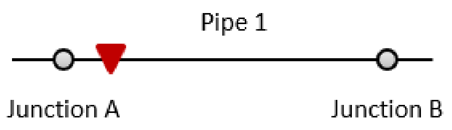


Figure 6: Example valve placement.

WNTR includes a method to generate valve layers based on **random** or **strategic** placement. The following example generates a **random** valve placement with 40 valves, “head()” is used to return the first 5 rows. The valve layer can be included in water network graphics (Figure 7).

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> random_valve_layer = wntr.network.generate_valve_layer(wn, 'random', 40)
>>> print(random_valve_layer.head())
  link node
0  317  273
1  221  161
2  283  239
3  295  249
4  303  257
>>> nodes, edges = wntr.graphics.plot_network(wn, node_size=7,
...     valve_layer=random_valve_layer)
```

The **strategic** placement specifies the number of pipes (n) from each node that do NOT contain a valve. In this case, n is generally 0, 1, or 2 (i.e., N, N-1, or N-2 valve placement) [WaWC06] [LWFZ17]. For example, if three pipes connect to a node and n = 2, then two of those pipes will not contain a valve and one pipe will contain a valve. The following example generates a strategic N-2 valve placement. The valve layer can be included in water network graphics (Figure 8).

```
>>> strategic_valve_layer = wntr.network.generate_valve_layer(wn, 'strategic', 2)
>>> nodes, edges = wntr.graphics.plot_network(wn, node_size=7,
...     valve_layer=strategic_valve_layer)
```

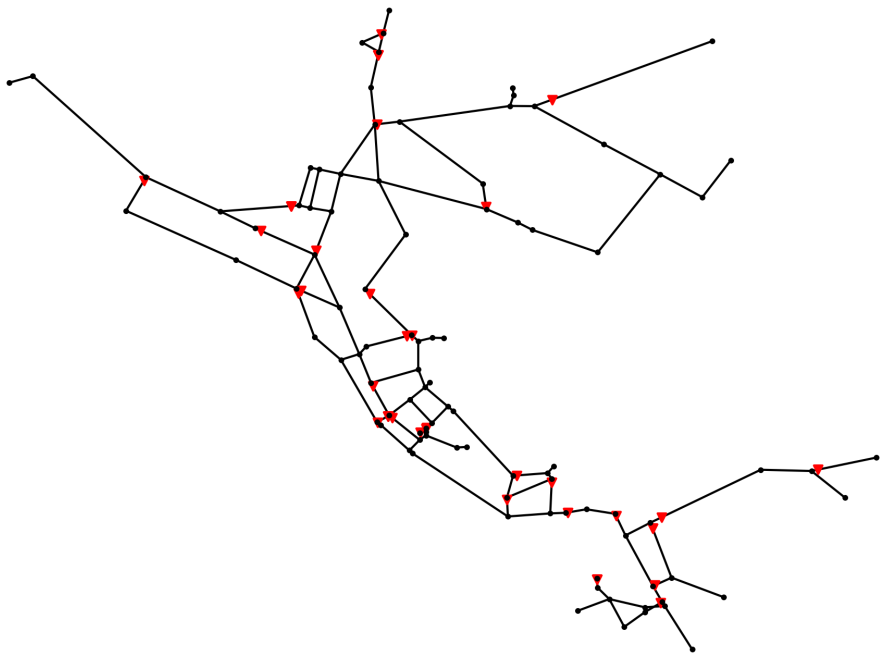


Figure 7: Valve layer using random placement.

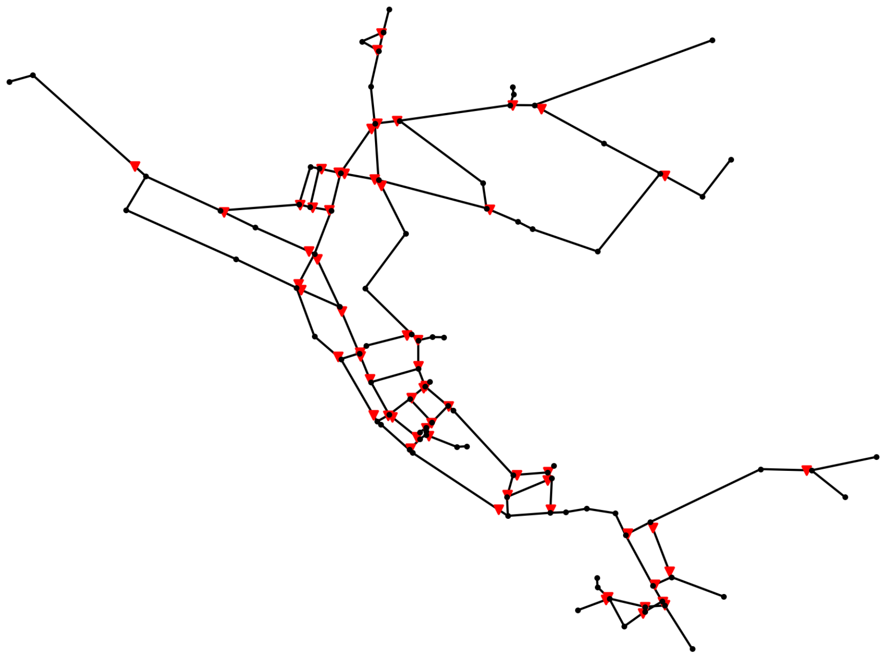


Figure 8: Valve layer using strategic N-2 placement.

10 Hydraulic simulation

WNTR contains two simulators: the EpanetSimulator and the WNTRSimulator. See *Software framework and limitations* for more information on features and limitations of these simulators.

The EpanetSimulator can be used to run demand-driven hydraulic simulations using the EPANET Programmer's Toolkit. The simulator can also be used to run water quality simulations, as described in *Water quality simulation*. A hydraulic simulation using the EpanetSimulator is run using the following code:

Note: EPANET refers to EPANET 2.00.12. Future releases of WNTR will include EPANET 2.2.0.

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results = sim.run_sim()
```

The WNTRSimulator is a hydraulic simulation engine based on the same equations as EPANET. The WNTRSimulator does not include equations to run water quality simulations. The WNTRSimulator includes the option to simulate leaks, and run hydraulic simulations in either demand-driven or pressure dependent demand mode ('DD' or 'PDD'). A hydraulic simulation using the WNTRSimulator is run using the following code:

```
>>> sim = wntr.sim.WNTRSimulator(wn, mode='DD')
>>> results = sim.run_sim()
```

More information on the simulators can be found in the API documentation, under `EpanetSimulator` and `WNTRSimulator`. The simulators use different solvers for the system of hydraulic equations; as such, small differences in the results are expected.

Options

Hydraulic simulation options are defined in the `WaterNetworkOptions` class. These options include duration, hydraulic timestep, rule timestep, pattern timestep, pattern start, default pattern, report timestep, report start, start clocktime, headloss, trials, accuracy, unbalanced, demand multiplier, and emitter exponent. All options are used with the EpanetSimulator. Options that are not used with the WNTRSimulator are described in *Limitations*.

The following example returns model options, which all have default values.

```
>>> wn.options
Time options:
  duration             : 604800
  hydraulic_timestep   : 900
  quality_timestep     : 900
  rule_timestep        : 360.0
  pattern_timestep     : 3600
  ...
```

Mass balance at nodes

Both simulators use the mass balance equations from EPANET [Ross00]:

$$\sum_{p \in P_n} q_{p,n} - D_n^{act} = 0 \qquad \forall n \in N$$

where P_n is the set of pipes connected to node n, $q_{p,n}$ is the flow rate of water into node n from pipe p (m^3/s), D_n^{act} is the actual demand out of node n (m^3/s), and N is the set of all nodes. If water is flowing out of node n and into pipe p, then $q_{p,n}$ is negative. Otherwise, it is positive.

Headloss in pipes

Both simulators use the Hazen-Williams headloss formula from EPANET [Ross00]:

$$H_{n_j} - H_{n_i} = h_L = 10.667 C^{-1.852} d^{-4.871} L q^{1.852}$$

where h_L is the headloss in the pipe (m), C is the Hazen-Williams roughness coefficient (unitless), d is the pipe diameter (m), L is the pipe length (m), q is the flow rate of water in the pipe (m^3/s), H_{n_j} is the head at the starting node (m), and H_{n_i} is the head at the ending node (m).

The flow rate in a pipe is positive if water is flowing from the starting node to the ending node and negative if water is flowing from the ending node to the starting node.

The WNTRSimulator solves for pressures and flows throughout the network as a set of linear equations. However, the Hazen-Williams headloss formula is not valid for negative flow rates. Therefore, the WNTRSimulator uses a reformulation of this constraint.

For $q < 0$:

$$h_L = -10.667 C^{-1.852} d^{-4.871} L |q|^{1.852}$$

For $q \geq 0$:

$$h_L = 10.667 C^{-1.852} d^{-4.871} L |q|^{1.852}$$

These equations are symmetric across the origin and valid for any q. Thus, this equation can be used for flow in either direction. However, the derivative with respect to q at $q = 0$ is 0. In certain scenarios, this can cause the Jacobian matrix of the set of hydraulic equations to become singular (when $q = 0$). To overcome this limitation, the WNTRSimulator splits the domain of q into segments to create a piecewise smooth function.

Demand-driven simulation

In a demand-driven simulation, the pressure in the system depends on the node demands. The mass balance and headloss equations described above are solved assuming that node demands are known and satisfied. This assumption is reasonable under normal operating conditions and for use in network design. Both simulators can run hydraulics using demand-driven simulation.

Pressure dependent demand simulation

In situations that lead to low pressure conditions (i.e., fire fighting, power outages, pipe leaks), consumers do not always receive their requested demand and a pressure dependent demand simulation is recommended. In a pressure dependent demand simulation, the delivered demand depends on the pressure. The mass balance and headloss equations described above are solved by simultaneously determining demand along with the network pressures and flow rates.

The WNTRSimulator can run hydraulics using a pressure dependent demand simulation according to the following pressure-demand relationship [WaSM88]:

$$d = \begin{cases} 0 & p \leq P_0 \\ D_f(\frac{p-P_0}{P_f-P_0})^{\frac{1}{2}} & P_0 \leq p \leq P_f \\ D^f & p \geq P_f \end{cases}$$

where d is the actual demand (m^3/s), D_f is the desired demand (m^3/s), p is the pressure (Pa), P_f is the nominal pressure (Pa) - this is the pressure above which the consumer should receive the desired demand, and P_0 is the minimum pressure (Pa) - this is the pressure below which the consumer cannot receive any water. The set of nonlinear equations comprising the hydraulic model and the pressure-demand relationship is solved directly using a Newton-Raphson algorithm.

Figure 9 illustrates the pressure-demand relationship using both the demand-driven and pressure dependent demand simulations. In the example, D_f is 0.0025 m^3/s (39.6 GPM), P_f is 30 psi (21.097 m), and P_0 is 5 psi (3.516 m). Using the demand-driven simulation, the demand is equal to D_f regardless of pressure. Using the pressure dependent demand simulation, the demand starts to decrease when the pressure is below P_f and goes to 0 when pressure is below P_0.

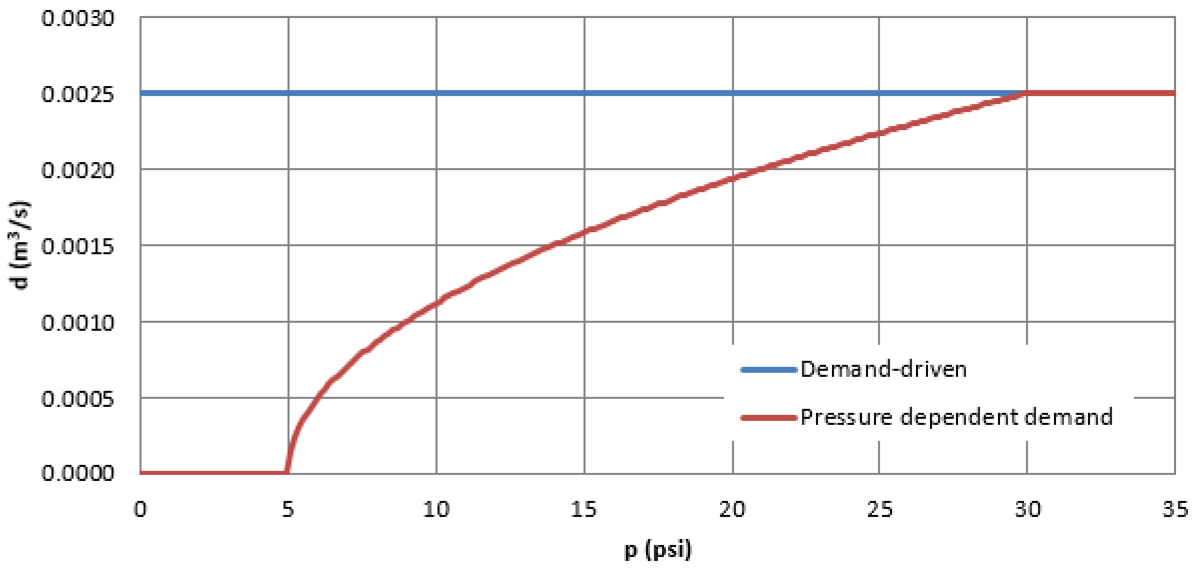


Figure 9: Relationship between pressure (p) and demand (d) using both the demand-driven and pressure dependent demand simulations.

The following example sets nominal and minimum pressure for each junction. Note that nominal and minimum pressure can vary throughout the network.

```
>>> for name, node in wn.junctions():
...     node.nominal_pressure = 21.097 # 30 psi = 21.097 psi
...     node.minimum_pressure = 3.516 # 5 psi = 3.516 psi
```

Leak model

The WNTRSimulator includes the ability to add leaks to the network using a leak model. As such, emitter coefficients defined in the water network model options are not used by the WNTRSimulator. Users interested in using the EpanetSimulator to model leaks can still do so by defining emitter coefficients. Note, that the emitter coefficient cannot be modified using the WNTR API, and can only be modified within the EPANET INP file.

When using the WNTRSimulator, leaks are modeled with a general form of the equation proposed by Crowl and Louvar [CrLo02] where the mass flow rate of fluid through the hole is expressed as:

$$d_{leak} = C_d A p^{\alpha} \sqrt{\frac{2}{\rho}}$$

where d_{leak} is the leak demand (m^3/s), C_d is the discharge coefficient (unitless), A is the area of the hole (m^2), p is the gauge pressure inside the pipe (Pa), α is the discharge coefficient, and ρ is the density of the fluid. The default discharge coefficient is 0.75 (assuming turbulent flow) [Lamb01], but the user can specify other values if needed. The value of α is set to 0.5 (assuming large leaks out of steel pipes) [Lamb01]. Leaks can be added to junctions and tanks. A pipe break is modeled using a leak area large enough to drain the pipe. WNTR includes methods to add leaks to any location along a pipe by splitting the pipe into two sections and adding a node.

Figure 10 illustrates leak demand. In the example, the diameter of the leak is set to 0.5 cm, 1.0 cm, and 1.5 cm.

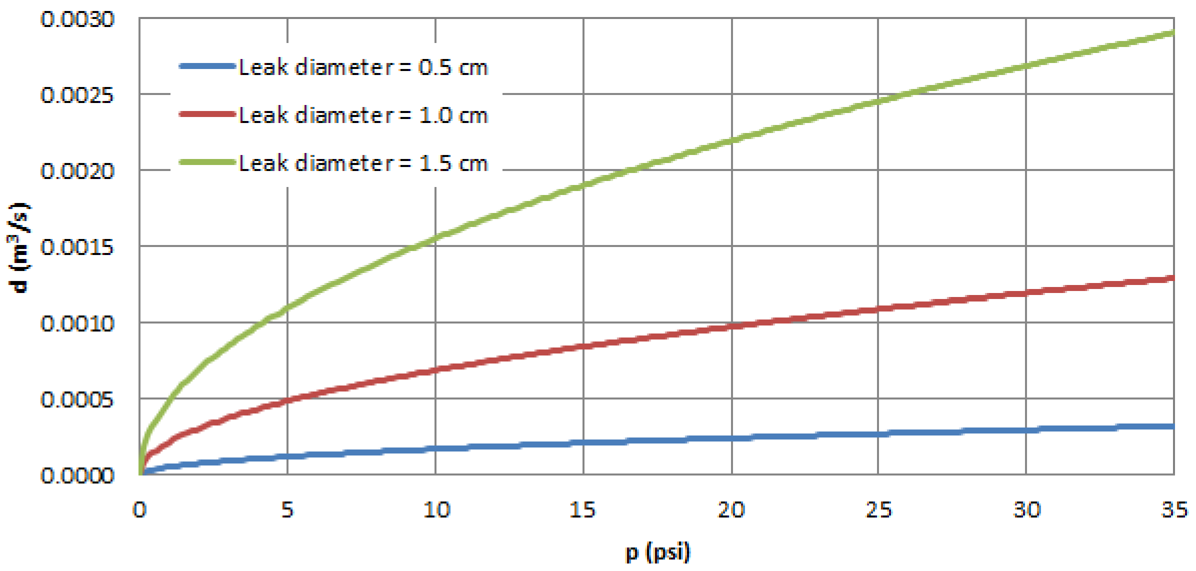


Figure 10: Relationship between leak demand (d) and pressure (p).

The following example adds a leak to the water network model.

```
>>> node = wn.get_node('123')
>>> node.add_leak(wn, area=0.05, start_time=2*3600, end_time=12*3600)
```

Pause and restart

The WNTRSimulator includes the ability to

- Reset initial values and re-simulate using the same water network model. Initial values include simulation time, tank head, reservoir head, pipe status, pump status, and valve status.
- Pause a hydraulic simulation, change network operations, and then restart the simulation
- Save the water network model and results to files and reload for future analysis

These features are helpful when evaluating various response action plans or when simulating long periods of time where the time resolution might vary.

The following example runs a hydraulic simulation for 10 hours and then restarts the simulation for another 14 hours. The results from the first 10 hours and last 14 hours can be combined for analysis or analyzed separately. Furthermore, network operations can be modified between simulations.

```
>>> wn.options.time.duration = 10*3600
>>> sim = wntr.sim.WNTRSimulator(wn)
>>> first_10_hours_results = sim.run_sim()
>>> wn.options.time.duration = 24*3600
>>> sim = wntr.sim.WNTRSimulator(wn)
>>> last_14_hours_results = sim.run_sim()
```

To restart the simulation from time zero, the user has several options.

1. Use the existing water network model and reset initial conditions. Initial conditions include simulation time, tank head, reservoir head, pipe status, pump status, and valve status. This option is useful when only initial conditions have changed between simulations.

   ```
   >>> wn.reset_initial_values()
   ```

2. Save the water network model to a file and reload that file each time a simulation is run. A pickle file is generally used for this purpose. A pickle file is a binary file used to serialize and de-serialize a Python object. More information on the use of pickle files can be found at https://docs.python.org/3/library/pickle.html. This option is useful when the water network model contains custom controls that would not be reset using the option 1, or when the user wants to change operations between simulations.

   The following example saves the water network model to a file before using it in a simulation.

   ```
   >>> import pickle

   >>> f=open('wn.pickle','wb')
   >>> pickle.dump(wn,f)
   >>> f.close()
   >>> sim = wntr.sim.WNTRSimulator(wn)
   >>> results = sim.run_sim()
   ```

   The next example reload the water network model from the file before the next simulation.

   ```
   >>> f=open('wn.pickle','rb')
   >>> wn = pickle.load(f)
   >>> f.close()
   >>> sim = wntr.sim.WNTRSimulator(wn)
   >>> results = sim.run_sim()
   ```

If these options do not cover user specific needs, then the water network model would need to be recreated between simulations or reset manually by changing individual attributes to the desired values. Note that when using the EpanetSimulator, the model is reset each time it is used in a simulation.

Advanced: Customized models with WNTR's AML

WNTR has a custom algebraic modeling language (AML) that is used for WNTR's hydraulic model (used in the `WNTRSimulator`). This AML is primarily used for efficient evaluation of constraint residuals and derivatives. WNTR's AML drastically simplifies the implementation, maintenance, modification, and customization of hydraulic models. The AML allows defining variables and constraints in a natural way. For example, suppose the user wants to solve the following system of nonlinear equations.

$$y - x^2 = 0$$
$$y - x - 1 = 0$$

To create this model using WNTR's AML, the following can be used:

```
>>> from wntr.sim import aml

>>> m = aml.Model()
>>> m.x = aml.Var(1.0)
>>> m.y = aml.Var(1.0)
>>> m.c1 = aml.Constraint(m.y - m.x**2)
>>> m.c2 = aml.Constraint(m.y - m.x - 1)
```

Before evaluating the constraint residuals or the Jacobian, `set_structure()` must be called:

```
>>> m.set_structure()
>>> m.evaluate_residuals()
array([ 0., -1.])
>>> m.evaluate_jacobian()
<2x2 sparse matrix of type '<class 'numpy.float64'>'
     with 4 stored elements in Compressed Sparse Row format>
>>> m.evaluate_jacobian().toarray()
array([[-2.,  1.],
    [-1.,  1.]])
```

The methods `evaluate_residuals()` and `evaluate_jacobian()` return a NumPy array and a SciPy sparse CSR matrix, respectively. Variable values can also be loaded with a NumPy array. For example, a Newton step (without a line search) would look something like

```
>>> from scipy.sparse.linalg import spsolve

>>> x = m.get_x()
>>> d = spsolve(m.evaluate_jacobian(), -m.evaluate_residuals())
>>> x += d
>>> m.load_var_values_from_x(x)
>>> m.evaluate_residuals()
array([-1., 0.])
```

WNTR includes an implementation of Newton's Method with a line search which can solve one of these models.

```
>>> from wntr.sim.solvers import NewtonSolver

>>> opt = NewtonSolver()
>>> res = opt.solve(m)
>>> m.x.value
1.618033988749989
>>> m.y.value
2.618033988749989
```

11 Water quality simulation

Water quality simulations can only be run using the EpanetSimulator. As listed in the *Software framework and limitations* section, this means that the hydraulic simulation must use demand-driven simulation. The WNTRSimulator can be used to compute demands under pressure dependent demand conditions and those demands can be used in the EpanetSimulator (see *Using PDD* below).

Note: The hydraulic simulation limitation is due to WNTR currently using EPANET 2.00.12, and not the currently released EPANET 2.2.0 with the pressure dependent algorithm.

After defining water quality options and sources (described in the *Options* and *Sources* sections below), a hydraulic and water quality simulation using the EpanetSimulator is run using the following code:

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results = sim.run_sim()
```

The results include a quality value for each node (see *Simulation results* for more details).

Options

Water quality simulation options are defined in the `WaterNetworkOptions` class. Three types of water quality analysis are supported. These options include water age, tracer, and chemical concentration.

- **Water age**: A water quality simulation can be used to compute water age at every node. To compute water age, set the 'quality' option as follows:

  ```
  >>> wn.options.quality.mode = 'AGE'
  ```

- **Tracer**: A water quality simulation can be used to compute the percent of flow originating from a specific location. The results include tracer percent values at each node. For example, to track a tracer from node '111,' set the 'quality' and 'tracer_node' options as follows:

  ```
  >>> wn.options.quality.mode = 'TRACE'
  >>> wn.options.quality.trace_node = '111'
  ```

- **Chemical concentration**: A water quality simulation can be used to compute chemical concentrations given a set of source injections. The results include chemical concentration values at each node. To compute chemical concentrations, set the 'quality' options as follows:

  ```
  >>> wn.options.quality.mode = 'CHEMICAL'
  ```

  The initial concentration is set using the *initial_quality* parameter on each node. This parameter can also be set using the [QUALITY] section of the INP file. The user can also define sources (described in the *Sources* section below).

- To skip the water quality simulation, set the 'quality' options as follows:

  ```
  >>> wn.options.quality.mode = 'NONE'
  ```

Additional water quality options include viscosity, diffusivity, specific gravity, tolerance, bulk reaction order, wall reaction order, tank reaction order, bulk reaction coefficient, wall reaction coefficient, limiting potential, and roughness correlation. These parameters are defined in the `WaterNetworkOptions` API documentation.

When creating a water network model from an EPANET INP file, water quality options are populated from the [OPTIONS] and [REACTIONS] sections of the EPANET INP file. All of these options can be modified in WNTR and then written to an EPANET INP file.

Sources

Sources are required for CHEMICAL water quality analysis. Sources can still be defined, but *will not* be used if AGE, TRACE, or NONE water quality analysis is selected. Sources are added to the water network model using the `add_source` method. Sources include the following information:

- **Source name**: A unique source name used to reference the source in the water network model.
- **Node name**: The injection node.
- **Source type**: Options include 'CONCEN,' 'MASS,' 'FLOWPACED,' or 'SETPOINT.'
  - CONCEN source represents injection of a specific concentration.
  - MASS source represents a booster source with a fixed mass flow rate.
  - FLOWPACED source represents a booster source with a fixed concentration at the inflow of the node.
  - SETPOINT source represents a booster source with a fixed concentration at the outflow of the node.
- **Strength**: Baseline source strength (in mass/time for MASS and mass/volume for CONCEN, FLOWPACED, and SETPOINT).
- **Pattern**: The pattern name associated with the injection.

For example, the following code can be used to add a source, and associated pattern, to the water network model:

```
>>> source_pattern = wntr.network.elements.Pattern.binary_pattern('SourcePattern',
...       start_time=2*3600, end_time=15*3600, duration=wn.options.time.duration,
...       step_size=wn.options.time.pattern_timestep)
>>> wn.add_pattern('SourcePattern', source_pattern)
>>> wn.add_source('Source', '121', 'SETPOINT', 1000, 'SourcePattern')
```

In the above example, the pattern is given a value of 1 between 2 and 15 hours, and 0 otherwise. The method `remove_source` can be used to remove sources from the water network model.

In the example below, the strength of the source is changed from 1000 to 1500.

```
>>> source = wn.get_source('Source')
>>> print(source)
<Source: 'Source', '121', 'SETPOINT', 1000, SourcePattern>

>>> source.strength_timeseries.base_value = 1500
>>> print(source)
<Source: 'Source', '121', 'SETPOINT', 1500, SourcePattern>
```

When creating a water network model from an EPANET INP file, the sources that are defined in the [SOURCES] section are added to the water network model. These sources are given the name 'INP#' where # is an integer related to the number of sources in the INP file.

Using PDD

As noted in the *Software framework and limitations* section, a pressure dependent demand hydraulic simulation is only available using the WNTRSimulator and water quality simulations are only available using the EpanetSimulator. The following example illustrates how to use pressure dependent demands in a water quality simulation. A hydraulic simulation is first run using the WNTRSimulator in PDD mode. The resulting demands are used to reset demands in the WaterNetworkModel and hydraulics and water quality are run using the EpanetSimulator.

```
>>> sim = wntr.sim.WNTRSimulator(wn, 'PDD')
>>> results = sim.run_sim()

>>> wn.assign_demand(results.node['demand'].loc[:,wn.junction_name_list], 'PDD')

>>> sim = wntr.sim.EpanetSimulator(wn)
>>> wn.options.quality.mode = 'TRACE'
>>> wn.options.quality.trace_node = '111'
>>> results_withPDD = sim.run_sim()
```

12 Simulation results

Simulation results are stored in a results object which contains:

- Timestamp when the results were created
- Network name
- Node results
- Link results

As shown in the *Hydraulic simulation* and *Water quality simulation* sections, simulations results can be generated using the EpanetSimulator as follows (similar methods are used to generate results using the WNTRSimulator):

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results = sim.run_sim()
```

The node and link results are dictionaries of pandas DataFrames. Each dictionary is a key:value pair, where the key is a result attribute (e.g., node demand, link flowrate) and the value is a DataFrame. DataFrames are indexed by timestep (in seconds from the start of the simulation) with columns that are labeled using node or link names. The use of pandas facilitates a comprehensive set of time series analysis options that can be used to evaluate results. For more information on pandas, see http://pandas.pydata.org/.

Conceptually, DataFrames can be visualized as blocks of data with 2 axis, as shown in Figure 11.

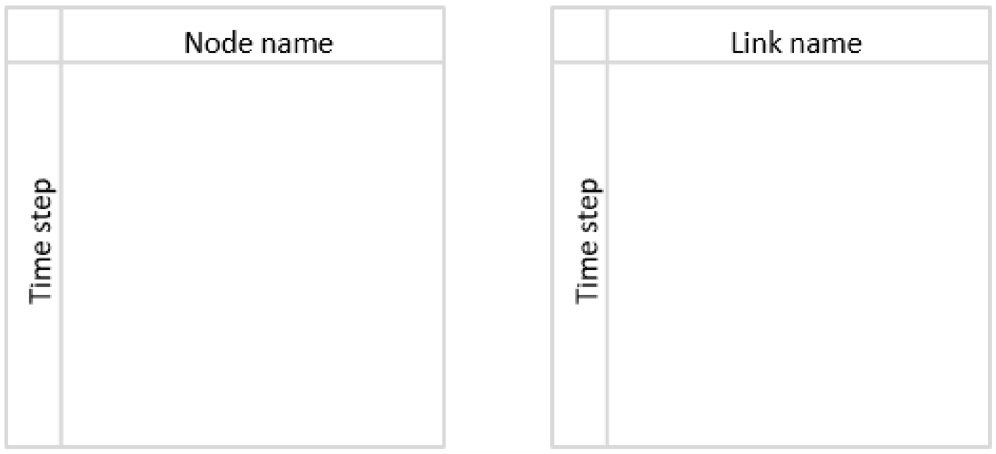


Figure 11: Conceptual representation of DataFrames used to store simulation results.

Node results include DataFrames for each of the following attributes:

- Demand
- Leak demand (only when the WNTRSimulator is used)
- Head
- Pressure
- Quality (only when the EpanetSimulator is used. Water age, tracer percent, or chemical concentration is stored, depending on the mode of water quality analysis)

For example, node results generated with the EpanetSimulator have the following keys:

```
>>> node_keys = results.node.keys()
>>> print(node_keys)
dict_keys(['demand', 'head', 'pressure', 'quality'])
```

Link results include DataFrames for each of the following attributes:

- Velocity
- Flowrate
- Status (0 indicates closed, 1 indicates open)
- Headloss (only when the EpanetSimulator is used)
- Setting (only when the EpanetSimulator is used)
- Friction factor (only when the EpanetSimulator is used)
- Reaction rate (only when the EpanetSimulator is used)
- Link quality (only when the EpanetSimulator is used)

The link results that are only accessible from the EpanetSimulator could be included in the WNTRSimulator in a future release. For example, link results generated with the EpanetSimulator have the following keys:

```
>>> link_keys = results.link.keys()
>>> print(link_keys)
dict_keys(['flowrate', 'frictionfact', 'headloss', 'linkquality', 'rxnrate', 'setting
↪', 'status', 'velocity'])
```

To access node pressure over all nodes and times:

```
>>> pressure = results.node['pressure']
```

DataFrames can be sliced to extract specific information. For example, to access the pressure at node '123' over all times (the ":" notation returns all variables along the specified axis, "head()" returns the first 5 rows, values displayed to 2 decimal places):

```
>>> pressure_at_node123 = pressure.loc[:,'123']
>>> print(pressure_at_node123.head())
0       47.08
900     47.13
1800    47.18
2700    47.23
3600    47.94
Name: 123, dtype: float32
```

To access the pressure at time 3600 over all nodes (values displayed to 2 decimal places):

```
>>> pressure_at_1hr = pressure.loc[3600,:]
>>> print(pressure_at_1hr.head())
name
10    28.25
15    28.89
20     9.10
35    41.52
40     4.18
Name: 3600, dtype: float32
```

Data can be plotted as a time series, as shown in Figure 12:

```
>>> ax = pressure_at_node123.plot()
>>> text = ax.set_xlabel("Time (s)")
>>> text = ax.set_ylabel("Pressure (m)")
```

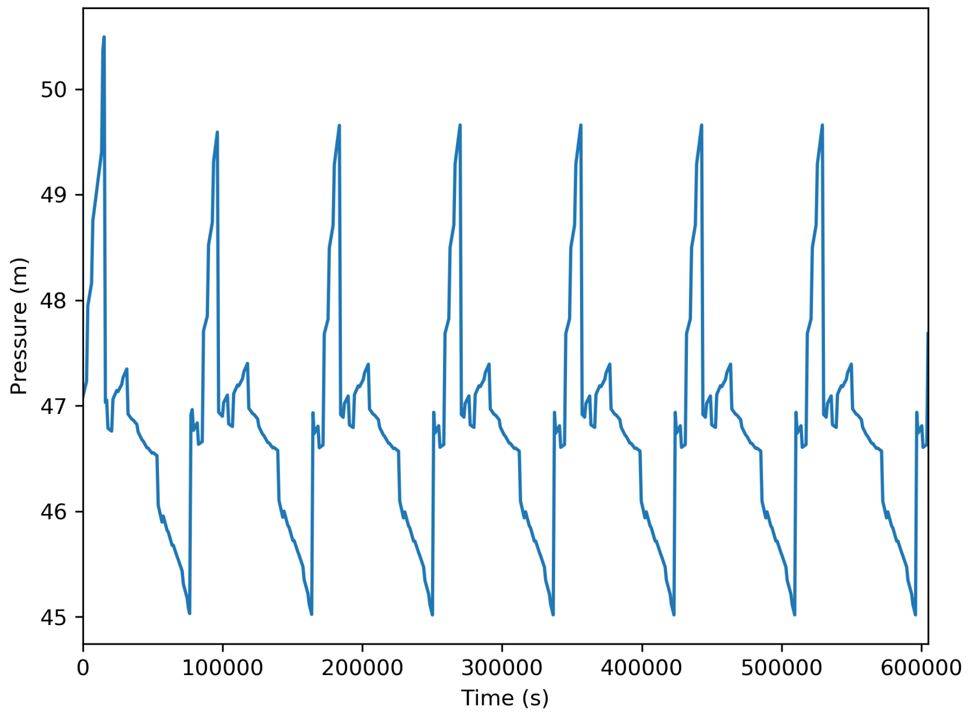


Figure 12: Time series graphic showing pressure at a node.

Data can also be plotted on the water network model, as shown in Figure 13. Note that the `plot_network` function returns matplotlib objects for the network nodes and edges, which can be further customized by the user. In this figure, the node pressure at 1 hr is plotted on the network. Link attributes can be plotted in a similar manner.

```
>>> nodes, edges = wntr.graphics.plot_network(wn, node_attribute=pressure_at_1hr,
...     node_range=[30,55], node_colorbar_label='Pressure (m)')
```

Network and time series graphics can be customized to add titles, legends, axis labels, and/or subplots.

Pandas includes methods to write DataFrames to the following file formats:

- Excel
- Comma-separated values (CSV)
- Hierarchical Data Format (HDF)
- JavaScript Object Notation (JSON)
- Structured Query Language (SQL)

For example, DataFrames can be saved to Excel files using:

```
>>> pressure.to_excel('pressure.xlsx')
```

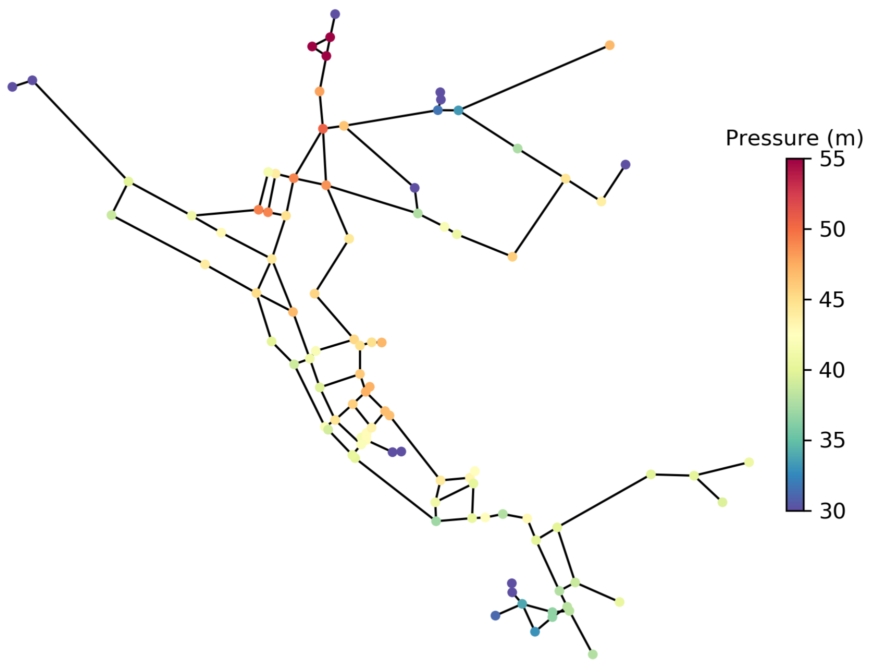


Figure 13: Network graphic showing pressure at 1 hour.

Note: The Pandas method `to_excel` requires the Python package **openpyxl** [GaCl18], which is an optional dependency of WNTR.

13 Disaster scenarios

Drinking water utilities might be interested in examining many different disaster scenarios. They could be acute incidents like power outages and earthquakes or they could be long term issues like persistent pipe leaks, population fluctuation, and changes to supply and demand. The following section describes disaster scenarios that can be modeled in WNTR.

Earthquake

Earthquakes can be some of the most sudden and impactful disasters that a water system experiences. An earthquake can cause lasting damage to the system that could take weeks, if not months, to fully repair. Earthquakes can cause damage to pipes, tanks, pumps, and other infrastructure. Additionally, earthquakes can cause power outages and fires.

WNTR includes methods to add leaks to pipes and tanks, shut off power to pumps, and change demands for fire conditions, as described in the sections below. The `Earthquake` class includes methods to compute peak ground acceleration, peak ground velocity, and repair rate based on the earthquake location and magnitude. Alternatively, external earthquake models or databases (e.g., ShakeMap [WWQP06]) can be used to compute earthquake properties and those properties can be loaded into Python for analysis in WNTR.

When simulating the effects of an earthquake, fragility curves are commonly used to define the probability that a component is damaged with respect to peak ground acceleration, peak ground velocity, or repair rate. The American Lifelines Alliance report [ALA01] includes seismic fragility curves for water system components. See *Stochastic simulation* for more information on fragility curves.

Since properties like peak ground acceleration, peak ground velocity, and repair rate are a function of the distance to the epicenter, node coordinates in the water network model must be in units of meters. Since some network models use other units for node coordinates, WNTR includes methods to change coordinate scale, as shown in the following example.

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> wn = wntr.morph.scale_node_coordinates(wn, 1000)
```

The following example computes peak ground acceleration, peak ground velocity, and repair rate for each pipe.

```
>>> epicenter = (32000,15000) # x,y location
>>> magnitude = 6.5 # Richter scale
>>> depth = 10000 # m, shallow depth
>>> earthquake = wntr.scenario.Earthquake(epicenter, magnitude, depth)
>>> distance = earthquake.distance_to_epicenter(wn, element_type=wntr.network.Pipe)
>>> pga = earthquake.pga_attenuation_model(distance)
>>> pgv = earthquake.pgv_attenuation_model(distance)
>>> repair_rate = earthquake.repair_rate_model(pgv)
```

The earthquake properties can be plotted on the network using the following example. The resulting map is shown in Figure 14.

```
>>> nodes, edges = wntr.graphics.plot_network(wn, link_attribute=pga, node_size=4,
...     link_width=2, link_colorbar_label='PGA (g)')
```

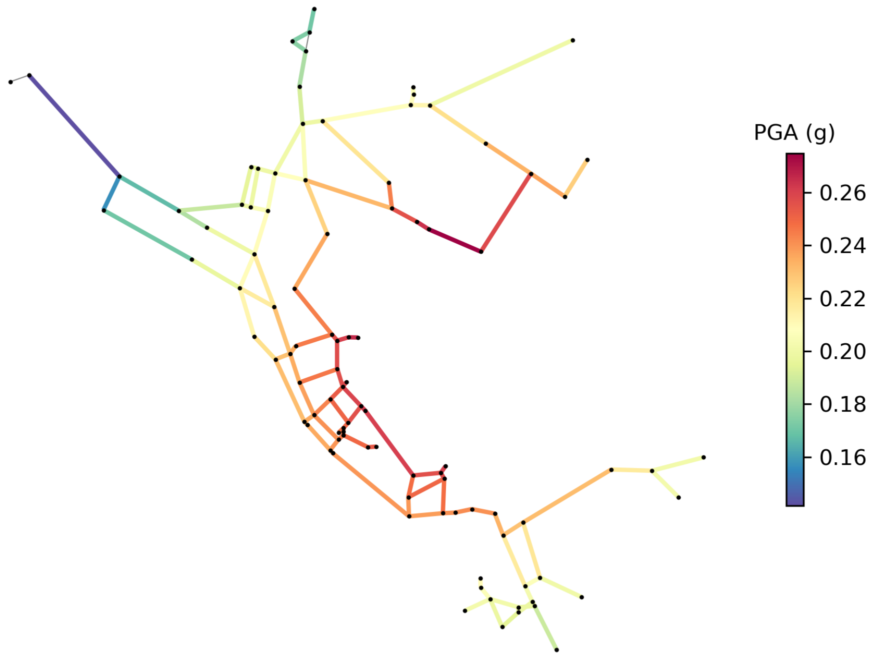


Figure 14: Peak ground acceleration.

Pipe breaks or leaks

Pipes are susceptible to leaks. Leaks can be caused by aging infrastructure, the freeze/thaw process, increased demand, or pressure changes. This type of damage is especially common in older cities where distribution systems were constructed from outdated materials like cast iron and even wood.

WNTR includes methods to add leaks to junctions and tanks (see *Leak model* for more details). Leaks can be added to a pipe by splitting the pipe and adding a junction. The following example adds a leak to a specific pipe.

```
>>> wn = wntr.morph.split_pipe(wn, '123', '123_B', '123_leak_node')
>>> leak_node = wn.get_node('123_leak_node')
>>> leak_node.add_leak(wn, area=0.05, start_time=2*3600, end_time=12*3600)
```

The method `add_leak` adds time controls to a junction, which includes the start and stop time for the leak.

Power outage

Power outages can be small and brief, or they can also span over several days and affect whole regions as seen in the 2003 Northeast Blackout. While the Northeast Blackout was an extreme case, a 2012 Lawrence Berkeley National Laboratory study [ELLT12] showed the frequency and duration of power outages are increasing domestically by a rate of two percent annually. In water distribution systems, a power outage can cause pump stations to shut down and result in reduced water pressure. This can lead to shortages in some areas of the system. Typically, no lasting damage in the system is associated with power outages.

WNTR can be used to simulate power outages by changing the pump status from ON to OFF and defining the duration of the outage. The following example adds a 5 hour power outage to a specific pump.

```
>>> pump = wn.get_link('335')
>>> pump.add_outage(wn, 5*3600, 10*3600)
```

The method `add_outage` adds time controls to a pump to start and stop a power outage. When simulating power outages, consider placing check bypasses around pumps and check valves next to reservoirs.

Fires

WNTR can be used to simulate damage caused to system components due to fire and/or to simulate water usage due to fighting fires. To fight fires, additional water is drawn from the system. Fire codes vary by state. Minimum required fire flow and duration are generally based on the building's area and purpose. While small residential fires might require 1500 gallons/minute for 2 hours, large commercial spaces might require 8000 gallons/minute for 4 hours [ICC12]. This additional demand can have a large impact on water pressure in the system.

WNTR can be used to simulate firefighting conditions in the system. WNTR simulates firefighting conditions by specifying the demand, time, and duration of firefighting. Pressure dependent demand simulation is recommended in cases where firefighting might impact expected demand. The following example adds fire flow conditions at a specific node.

```
>>> fire_flow_demand = 0.252 # 4000 gal/min = 0.252 m3/s
>>> fire_start = 10*3600
>>> fire_end = 14*3600
>>> node = wn.get_node('197')
>>> node.add_fire_fighting_demand(wn, fire_flow_demand, fire_start, fire_end)
```

Environmental change

Environmental change is a long term problem for water distribution systems. Changes in the environment could lead to reduced water availability, damage from weather incidents, or even damage from subsidence. For example, severe drought in California has forced lawmakers to reduce the state's water usage by 25 percent. Environmental change also leads to sea level rise which can inundate distribution systems. This is especially prevalent in cities built on unstable soils like New Orleans and Washington, DC, which are experiencing land subsidence.

WNTR can be used to simulate the effects of environmental change on the water distribution system by changing supply and demand, adding disruptive conditions (i.e., power outages, pipe leaks) caused by severe weather, or by adding pipe leaks caused by subsidence. Power outages and pipe leaks are described above. Changes to supply and demand can be simple (i.e., changing all nodes by a certain percent), or complex (i.e., using external data or correlated statistical methods). The following example changes supply and demand in the model.

```
>>> for res_name, res in wn.reservoirs():
...     res.head_timeseries.base_value = res.head_timeseries.base_value*0.9
>>> for junc_name, junc in wn.junctions():
...     for demand in junc.demand_timeseries_list:
...         demand.base_value = demand.base_value*1.15
```

Contamination

Water distribution systems are vulnerable to contamination by a variety of chemical, microbial, or radiological substances. During disasters, contamination can enter the system through reservoirs, tanks, and at other access points within the distribution system. Long term environmental change can lead to degradation of water sources. Contamination can be difficult to detect and is very expensive to clean up. Recent incidents, including the Elk River chemical spill and Flint lead contamination, highlight the need to minimize human health and economic impacts.

WNTR simulates contamination incidents by introducing contaminants into the distribution system and allowing them to propagate through the system. The section on *Water quality simulation* includes steps to define and simulate contamination incidents.

Future versions of WNTR will be able to simulate changes in source water quality due to contamination incidents.

Other disaster scenarios

Drinking water systems are also susceptible to other natural disasters including floods, droughts, hurricanes, tornadoes, extreme winter storms, and wind events. WNTR can be used to simulate these events by combining the disaster models already described above. For example, tornadoes might cause power outages, pipe breaks, other damage to infrastructure, and fires. Floods might cause power outages, changes to source water (because of treatment failures), and pipe breaks.

14 Resilience metrics

Resilience of water distribution systems refers to the design, maintenance, and operations of that system. All these aspects must work together to limit the effects of disasters and enable rapid return to normal delivery of safe water to customers. Numerous resilience metrics have been suggested [USEPA14]. These metrics generally fall into five categories: topographic, hydraulic, water quality, water security, and economic [USEPA14]. When quantifying resilience, it is important to understand which metric best defines resilience for a particular scenario. WNTR includes many metrics to help users compare resilience using different methods.

The following sections outline metrics that can be computed using WNTR, including:

- Topographic metrics (Table 6)
- Hydraulic metrics (Table 7)
- Water quality metrics (Table 8)
- Water security metrics (Table 9)
- Economic metrics (Table 10)

While some metrics define resilience as a single system-wide quantity, other metrics define quantities that are a function of time, space, or both. For this reason, state transition plots [BaRR13] and network graphics are useful ways to visualize resilience and compare metrics, as shown in Figure 15. In the state transition plot, the x-axis represents time (before, during, and after a disruptive incident). The y-axis represents performance. This can be any time varying resilience metric that responds to the disruptive state. State transition plots are often generated to show time varying performance of the system, but they can also represent the time varying performance of individual components, like tanks or pipes. Network graphics are useful to visualize resilience metrics that vary with respect to location. For metrics that vary with respect to time and space, network animation can be used to illustrate resilience.

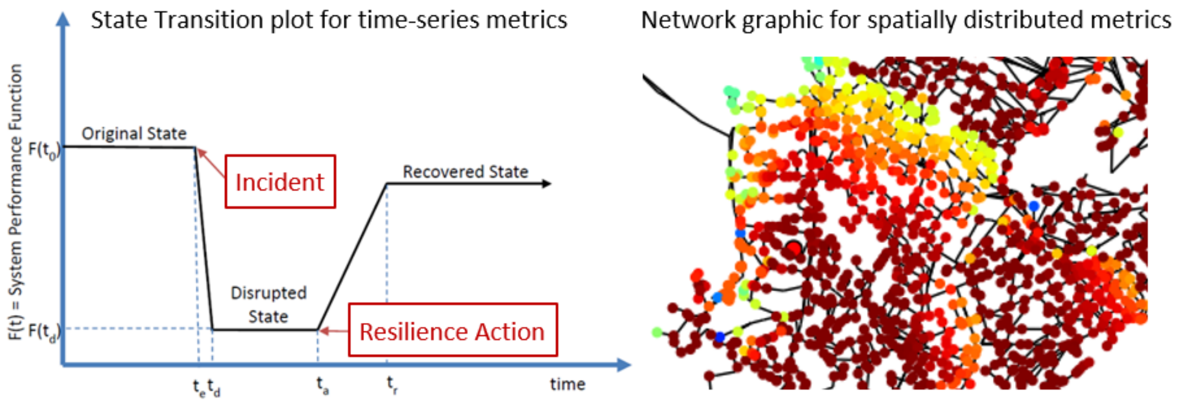

State Transition plot for time-series metrics
F(t) = System Performance Function
F(t_0)
F(t_d)
Original State
Incident
Recovered State
Disrupted State
Resilience Action
t_e t_d
t_a
t_r
time
Network graphic for spatially distributed metrics

Figure 15: Example state transition plot (left) and network graphic (right) used to visualize resilience.

Topographic metrics

Topographic metrics, based on graph theory, can be used to assess the connectivity of water distribution networks. These metrics rely on the physical layout of the network components and can be used to understand how the underlying structure and connectivity constrains resilience. For example, a regular lattice, where each node has the same number of edges (except at the border), is considered to be the most reliable graph structure. On the other hand, a random lattice has nodes and edges that are placed according to a random process. A real world water distribution system probably lies somewhere in between a regular lattice and a random lattice in terms of structure and reliability.

Commonly used topographic metrics are listed in Table 6. Many of these metrics can be computed using NetworkX directly (see https://networkx.github.io/ for more information). WNTR includes additional topographic metrics to help compute resilience.

Table 6: Topographic Resilience Metrics

Metric	Description
Node degree and terminal nodes	Node degree is the number of links adjacent to a node. Node degree is a measure of the number of branches in a network. A node with degree 0 is not connected to the network. Terminal nodes have degree 1. A node connected to every node (including itself) has a degree equal to the number of nodes in the network. The average node degree is a system wide metric used to describe the number of connected links in a network.
Link density	Link density is the ratio between the total number of links and the maximum number of links in the network. If links are allowed to connect a node to itself, then the maximum number of links is n^2, where n is the number of nodes. Otherwise, the maximum number of nodes is $n(n-1)$. Link density is 0 for a graph without edges and 1 for a dense graph. The density of multigraphs can be higher than 1.
Eccentricity and diameter	Eccentricity is the maximum number of links between a node and all other nodes in the graph. Eccentricity is a value between 0 and the number of links in the network. Diameter is the maximum eccentricity in the network. Eccentricity and diameter can only be computed using undirected, connected networks.
Betweenness centrality	Betweenness centrality is the fraction of shortest paths that pass through each node. Betweenness coefficient is a value between 0 and 1. Central point dominance is the average difference in betweenness centrality of the most central point (having the maximum betweenness centrality) and all other nodes.
Closeness centrality	Closeness centrality is the inverse of the sum of shortest path from one node to all other nodes.
Articulation points	A node is considered an articulation point if the removal of that node (along with all its incident edges) increases the number of connected components of a network. Density of articulation points is the ratio of the number of articulation points and the total number of nodes. Density of articulation points is a value between 0 and 1.
Bridges	A link is considered a bridge if the removal of that link increases the number of connected components in the network. The ratio of the number of bridges and the total number of links in the network is the bridge density. Bridge density is a value between 0 and 1.
Simple paths	A simple path is a path between two nodes that does not repeat any nodes. Paths can be time dependent, if related to flow direction.
Shortest path lengths	Shortest path lengths is the minimum number of links between a source node and all other nodes in the network. Shortest path length is a value between 0 and the number of links in the network. The average shortest path length is a system wide metric used to describe the number of links between a node and all other nodes.
Valve segmentation	Valve segmentation groups links and nodes into segments based on the location of isolation valves. Valve segmentation returns a segment number for each node and link, along with the number of nodes and links in each segment.

To compute topographic metrics, a NetworkX MultiDiGraph is first extracted from a WaterNetworkModel. Note that some metrics require an undirected graph or a graph with a single edge between two nodes.

```
>>> import networkx as nx
>>> import wntr
```

(continues on next page)

(continued from previous page)

```
>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> G = wn.get_graph() # directed multigraph
>>> uG = G.to_undirected() # undirected multigraph
>>> sG = nx.Graph(uG) # undirected simple graph (single edge between two nodes)
```

The following examples compute topographic metrics. Note that many of these metrics use NetworkX directly, while others use metrics included in WNTR.

- Node degree and terminal nodes

  ```
  >>> node_degree = G.degree()
  >>> terminal_nodes = wntr.metrics.terminal_nodes(G)
  ```

- Link density

  ```
  >>> link_density = nx.density(G)
  ```

- Diameter and eccentricity

  ```
  >>> diameter = nx.diameter(uG)
  >>> eccentricity = nx.eccentricity(uG)
  ```

- Betweenness centrality and central point dominance

  ```
  >>> betweenness_centrality = nx.betweenness_centrality(sG)
  >>> central_point_dominance = wntr.metrics.central_point_dominance(G)
  ```

- Closeness centrality

  ```
  >>> closeness_centrality = nx.closeness_centrality(G)
  ```

- Articulation points and bridges

  ```
  >>> articulation_points = list(nx.articulation_points(uG))
  >>> bridges = wntr.metrics.bridges(G)
  ```

- Shortest path lengths between all nodes and average shortest path length

  ```
  >>> shortest_path_length = nx.shortest_path_length(uG)
  >>> ave_shortest_path_length = nx.average_shortest_path_length(uG)
  ```

- Paths between two nodes in a weighted graph, where the graph is weighted by flow direction from a hydraulic simulation

  ```
  >>> sim = wntr.sim.EpanetSimulator(wn)
  >>> results = sim.run_sim()

  >>> flowrate = results.link['flowrate']
  >>> G = wn.get_graph(link_weight=flowrate)
  >>> all_paths = nx.all_simple_paths(G, '119', '193')
  ```

- Valve segmentation, where each valve is defined by a node and link pair (see *Valve layer*)

  ```
  >>> valve_layer = wntr.network.generate_valve_layer(wn, 'random', 40)
  >>> node_segments, link_segments, segment_size = wntr.metrics.valve_segments(G,
  ...     valve_layer)
  ```

Hydraulic metrics

Hydraulic metrics are based on flow, demand, and/or pressure. With the exception of expected demand and average expected demand, the calculation of these metrics requires simulation of network hydraulics that reflect how the system operates under normal or abnormal conditions. Hydraulic metrics included in WNTR are listed in Table 7.

Table 7: Hydraulic Resilience Metrics

Metric	Description
Pressure	To determine the number of node-time pairs above or below a specified pressure threshold, use the `query` method on results.node['pressure'].
Demand	To determine the number of node-time pairs above or below a specified demand threshold, use the `query` method on results.node['demand']. This method can be used to compute the fraction of delivered demand, from [OsKS02].
Water service availability	Water service availability is the ratio of delivered demand to the expected demand. This metric can be computed as a function of time or space using the `water_service_availability` method. This method can be used to compute the fraction of delivered volume, from [OsKS02].
Todini index	The Todini index [Todi00] is related to the capability of a system to overcome failures while still meeting demands and pressures at the nodes. The Todini index defines resilience at a specific time as a measure of surplus power at each node and measures relative energy redundancy. The Todini index can be computed using the `todini_index` method.
Entropy	Entropy [AwGB90] is a measure of uncertainty in a random variable. In a water distribution network model, the random variable is flow in the pipes and entropy can be used to measure alternate flow paths when a network component fails. A network that carries maximum entropy flow is considered reliable with multiple alternate paths. Connectivity will change at each timestep, depending on the flow direction. The `get_graph` method can be used to generate a weighted graph. Entropy can be computed using the `entropy` method.
Expected demand	Expected demand is computed at each node and timestep based on node demand, demand pattern, and demand multiplier [USEPA15]. The metric can be computed using the `expected_demand` method. This method does not require running a hydraulic simulation.
Average expected demand	Average expected demand per day is computed at each node based on node demand, demand pattern, and demand multiplier [USEPA15]. The metric can be computed using the `average_expected_demand` method. This method does not require running a hydraulic simulation.
Population impacted	Population that is impacted by a specific quantity can be computed using the `population_impacted` method. For example, this method can be used to compute the population impacted by pressure below a specified threshold. Population per node is computed using the method `population`, which divides the average expected demand by the average volume of water consumed per capita per day. The default value for average volume of water consumed per capita per day is 200 gallons/day and can be modified by the user.

The following examples compute hydraulic metrics, including:

- Nodes and times when pressure exceeds a threshold, using results from a hydraulic simulation

```
>>> import numpy as np

>>> sim = wntr.sim.WNTRSimulator(wn, mode='PDD')
>>> results = sim.run_sim()
```

(continues on next page)

(continued from previous page)

```
>>> pressure = results.node['pressure']
>>> threshold = 21.09 # 30 psi
>>> pressure_above_threshold = wntr.metrics.query(pressure, np.greater,
...     threshold)
```

- Water service availability (Note that for Net3, the simulated demands are never less than the expected demand, and water service availability is always 1 (for junctions that have positive demand) or NaN (for junctions that have demand equal to 0).

```
>>> expected_demand = wntr.metrics.expected_demand(wn)
>>> demand = results.node['demand']
>>> wsa = wntr.metrics.water_service_availability(expected_demand, demand)
```

- Todini index

```
>>> head = results.node['head']
>>> pump_flowrate = results.link['flowrate'].loc[:,wn.pump_name_list]
>>> todini = wntr.metrics.todini_index(head, pressure, demand, pump_flowrate, wn,
...     threshold)
```

- Entropy

```
>>> flowrate = results.link['flowrate'].loc[12*3600,:]
>>> G = wn.get_graph(link_weight=flowrate)
>>> entropy, system_entropy = wntr.metrics.entropy(G)
```

Water quality metrics

Water quality metrics are based on the concentration or water age. The calculation of these metrics require a water quality simulation. Water quality metrics included in WNTR are listed in Table 8.

Table 8: Water Quality Resilience Metrics

Metric	Description
Water age	To determine the number of node-time pairs above or below a specified water age threshold, use the `query` method on results.node['quality'] after a simulation using AGE. Water age can also be computed using the average age from the last 48 hours of the simulation results.
Concentration	To determine the number of node-time pairs above or below a specified concentration threshold, use the `query` method on results.node['quality'] after a simulation using CHEM or TRACE. This method can be used to compute the fraction of delivered quality, from [OsKS02].
Population impacted	As stated above, population that is impacted by a specific quantity can be computed using the `population_impacted` method. This can be applied to water quality metrics.

The following examples compute water quality metrics, including:

- Water age using the last 48 hours of a water quality simulation

```
>>> wn.options.quality.mode = 'AGE'
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results = sim.run_sim()
```

(continues on next page)

(continued from previous page)

```
>>> age = results.node['quality']
>>> age_last_48h = age.loc[age.index[-1]-48*3600:age.index[-1]]
>>> average_age = age_last_48h.mean()/3600 # convert to hours
```

- Population that is impacted by water age greater than 24 hours

```
>>> pop = wntr.metrics.population(wn)
>>> threshold = 24
>>> pop_impacted = wntr.metrics.population_impacted(pop, average_age, np.greater,
...     threshold)
```

- Nodes that exceed a chemical concentration using a water quality simulation

```
>>> wn.options.quality.mode = 'CHEMICAL'
>>> source_pattern = wntr.network.elements.Pattern.binary_pattern('SourcePattern',
...     step_size=3600, start_time=2*3600, end_time=15*3600, duration=7*24*3600)
>>> wn.add_pattern('SourcePattern', source_pattern)
>>> wn.add_source('Source1', '121', 'SETPOINT', 1000, 'SourcePattern')
>>> wn.add_source('Source2', '123', 'SETPOINT', 1000, 'SourcePattern')
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results = sim.run_sim()

>>> chem = results.node['quality']
>>> threshold = 750
>>> mask = wntr.metrics.query(chem, np.greater, threshold)
>>> chem_above_regulation = mask.any(axis=0) # True/False for each node
```

Water security metrics

Water security metrics quantify potential consequences of contamination scenarios. These metrics are documented in [USEPA15]. Water security metrics included in WNTR are listed in Table 9.

Table 9: Water Security Resilience Metrics

Metric	Description
Mass consumed	Mass consumed is the mass of a contaminant that exits the network via node demand at each node-time pair [USEPA15]. The metric can be computed using the `mass_contaminant_consumed` method.
Volume consumed	Volume consumed is the volume of a contaminant that exits the network via node demand at each node-time pair [USEPA15]. The metric can be computed using the `volume_contaminant_consumed` method.
Extent of contamination	Extent of contamination is the length of contaminated pipe at each node-time pair [USEPA15]. The metric can be computed using the `extent_contaminant` method.
Population impacted	As stated above, population that is impacted by a specific quantity can be computed using the `population_impacted` method. This can be applied to water security metrics.

The following examples use the results from the chemical water quality simulation (from above) to compute water security metrics, including:

- Mass consumed

```
>>> demand = results.node['demand'].loc[:,wn.junction_name_list]
>>> quality = results.node['quality'].loc[:,wn.junction_name_list]
>>> MC = wntr.metrics.mass_contaminant_consumed(demand, quality)
```

- Volume consumed

```
>>> detection_limit = 750
>>> VC = wntr.metrics.volume_contaminant_consumed(demand, quality,
...     detection_limit)
```

- Extent of contamination

```
>>> quality = results.node['quality'] # quality at all nodes
>>> flowrate = results.link['flowrate'].loc[:,wn.pipe_name_list]
>>> EC = wntr.metrics.extent_contaminant(quality, flowrate, wn, detection_limit)
```

- Population impacted by mass consumed over a specified threshold.

```
>>> pop = wntr.metrics.population(wn)
>>> threshold = 80000
>>> pop_impacted = wntr.metrics.population_impacted(pop, MC, np.greater,
...     threshold)
```

Economic metrics

Economic metrics include network cost and greenhouse gas emissions. Economic metrics included in WNTR are listed in Table 10.

Table 10: Economic Resilience Metrics

Metric	Description
Network cost	Network cost is the annual maintenance and operations cost of tanks, pipes, valves, and pumps based on the equations from the Battle of Water Networks II [SOKZ12]. Default values can be included in the calculation. Network cost can be computed using the `annual_network_cost` method.
Greenhouse gas emissions	Greenhouse gas emissions is the annual emissions associated with pipes based on equations from the Battle of Water Networks II [SOKZ12]. Default values can be included in the calculation. Greenhouse gas emissions can be computed using the `annual_ghg_emissions` method.
Pump operating energy and cost	The energy and cost required to operate a pump can be computed using the `pump_energy` and `pump_cost` methods. These use the flowrates and pressures from simulation results to compute pump energy and cost.

The following examples compute economic metrics, including:

- Network cost

```
>>> network_cost = wntr.metrics.annual_network_cost(wn)
```

- Greenhouse gas emission

```
>>> network_ghg = wntr.metrics.annual_ghg_emissions(wn)
```

- Pump energy and pump cost using results from a hydraulic simulation

```
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results = sim.run_sim()

>>> pump_flowrate = results.link['flowrate'].loc[:,wn.pump_name_list]
>>> head = results.node['head']
>>> pump_energy = wntr.metrics.pump_energy(pump_flowrate, head, wn)
>>> pump_cost = wntr.metrics.pump_cost(pump_flowrate, head, wn)
```

15 Stochastic simulation

Stochastic simulations can be used to evaluate an ensemble of hydraulic and/or water quality scenarios. For disaster scenarios, the location, duration, and severity of different types of incidents can be drawn from distributions and included in the simulation. Distributions can be a function of component properties (i.e., age, material) or based on engineering standards. The Python packages NumPy and SciPy include statistical distributions and random selection methods that can be used for stochastic simulations.

For example, the following code can be used to select N unique pipes based on the failure probability of each pipe.

```
>>> import numpy as np

>>> pipe_names = ['pipe1', 'pipe2', 'pipe3', 'pipe4']
>>> failure_probability = [0.10, 0.20, 0.30, 0.40]
>>> N = 2
>>> selected_pipes = np.random.choice(pipe_names, N, replace=False,
...     p=failure_probability)
```

A stochastic simulation example provided with WNTR runs multiple realizations of a pipe leak scenario where the location and duration are drawn from probability distributions.

16 Fragility curves

Fragility curves are commonly used in disaster models to define the probability of exceeding a given damage state as a function of environmental change. Fragility curves are closely related to survival curves, which are used to define the probability of component failure due to age. For example, to estimate earthquake damage, fragility curves are defined as a function of peak ground acceleration, peak ground velocity, or repair rate. The American Lifelines Alliance report [ALA01] includes seismic fragility curves for water network components. Fragility curves can also be defined as a function of flood stage, wind speed, and temperature for other types of disasters.

Fragility curves can have multiple damage states. Each state should correspond to specific changes to the network model that represent damage, for example, a major or minor leak. Each state is defined with a name (i.e., 'Major,' 'Minor'), priority (i.e., 1, 2, where increasing (higher) numbers equal higher priority), and distribution (using the SciPy Python package). The distribution can be defined for all elements using the keyword 'Default,' or can be defined for individual components. Each fragility curve includes a 'No damage' state with priority 0 (lowest priority).

The following example defines a fragility curve with two damage states: Minor damage and Major damage.

```
>>> from scipy.stats import lognorm
>>> import wntr

>>> FC = wntr.scenario.FragilityCurve()
>>> FC.add_state('Minor', 1, {'Default': lognorm(0.5,scale=0.2)})
>>> FC.add_state('Major', 2, {'Default': lognorm(0.5,scale=0.5)})
>>> ax = wntr.graphics.plot_fragility_curve(FC, xlabel='Peak Ground Acceleration (g)')
```

Figure 16 illustrates the fragility curve as a function of peak ground acceleration. For example, if the peak ground acceleration is 0.3 at a specific pipe, the probability of exceeding a Major damage state is 0.16 and the probability of exceeding the Minor damage state is 0.80.

To use the fragility curve to assign damage to pipes, a random number is drawn between 0 and 1 and the associated probability of failure and damage state can be obtained. The example below uses the fragility curve to select a damage state for each pipe based on peak ground acceleration. After obtaining the damage state for the pipe, the network can be changed to reflect the associated damage. For example, if the pipe has Major damage, a large leak might be defined at that location.

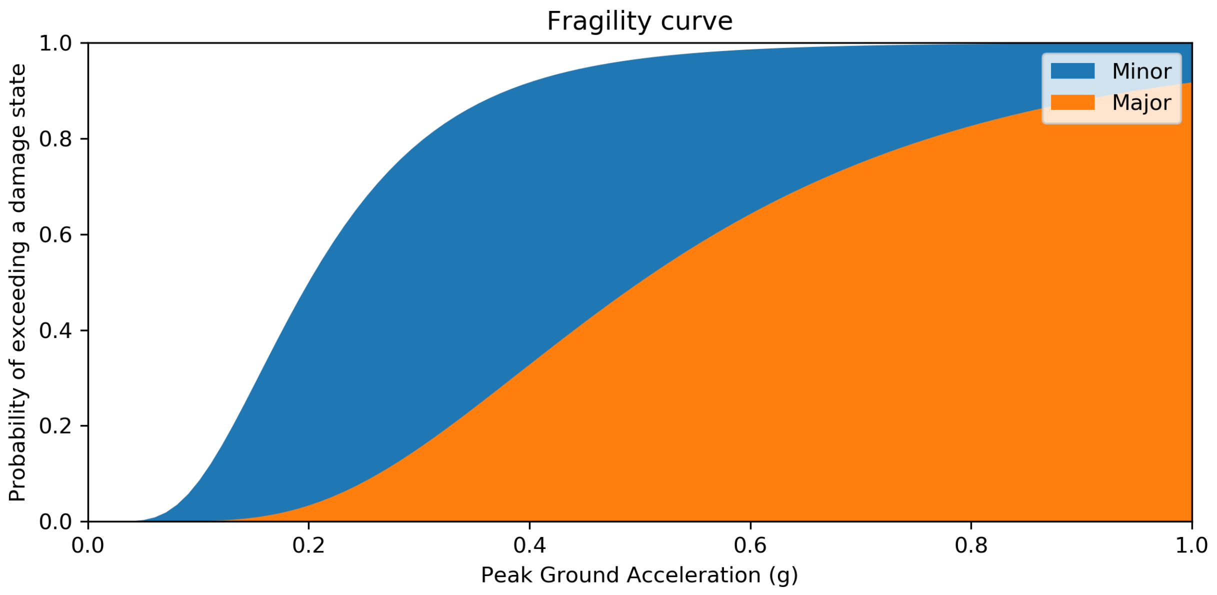


Figure 16: Example fragility curve.

```
>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> wn = wntr.morph.scale_node_coordinates(wn, 1000)
>>> epicenter = (32000,15000) # x,y location
>>> magnitude = 6.5 # Richter scale
>>> depth = 10000 # m, shallow depth
>>> earthquake = wntr.scenario.Earthquake(epicenter, magnitude, depth)
>>> distance = earthquake.distance_to_epicenter(wn, element_type=wntr.network.Pipe)
>>> pga = earthquake.pga_attenuation_model(distance)

>>> failure_probability = FC.cdf_probability(pga)
>>> damage_state = FC.sample_damage_state(failure_probability)
```

To plot the damage state on the network, the state (i.e., Major) can be converted to a number using the priority map, as shown below (Figure 17).

```
>>> priority_map = FC.get_priority_map()
>>> damage_value = damage_state.map(priority_map)
>>> custom_cmp = wntr.graphics.custom_colormap(3, ['grey', 'royalblue', 'darkorange'])
>>> nodes, edges = wntr.graphics.plot_network(wn, link_attribute=damage_value,
...     node_size=0, link_width=2, link_cmap=custom_cmp,
...     title='Damage state: 0=None, 1=Minor, 2=Major')
```

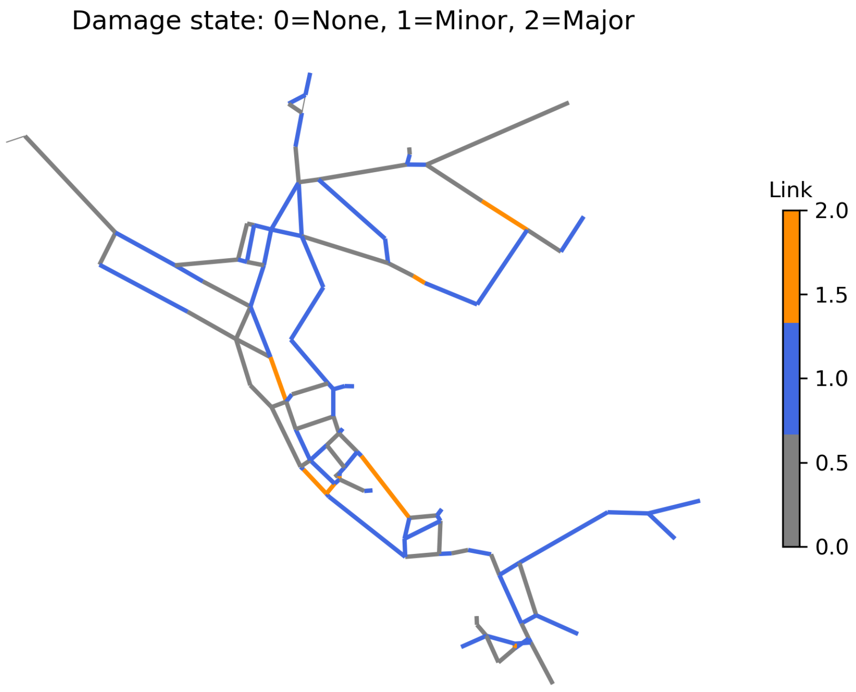


Figure 17: Damage state, selected from the fragility curve.

17 Network morphology

The water network model morphology can be modified in several ways using WNTR, including network skeletonization, modifying node coordinates, and splitting or breaking pipes.

Network skeletonization

The goal of network skeletonization is to reduce the size of a water network model with minimal impact on system behavior. Network skeletonization in WNTR follows the procedure outlined in [WCSG03]. The skeletonization process retains all tanks, reservoirs, valves, and pumps, along with all junctions and pipes that are associated with controls. Junction demands and demand patterns are retained in the skeletonized model, as described below. Merged pipes are assigned equivalent properties for diameter, length, and roughness to approximate the updated system behavior. Pipes that fall below a user defined pipe diameter threshold are candidates for removal based on three operations, including:

1. **Branch trimming**: Dead-end pipes that are below the pipe diameter threshold are removed from the model (Figure 18). The demand and demand pattern assigned to the dead-end junction is moved to the junction that is retained in the model. Dead-end pipes that are connected to tanks or reservoirs are not removed from the model.

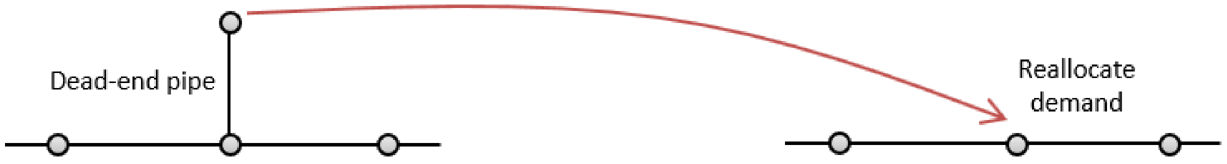


Figure 18: Branch trimming.

2. **Series pipe merge**: Pipes in series are merged if both pipes are below the pipe diameter threshold (Figure 19). The demand and demand pattern assigned to the connecting junction is moved to the nearest junction that is retained in the model. The merged pipe is assigned the following equivalent properties:

$$D_m = max\left(D_1, D_2\right)$$

$$L_m = L_1 + L_2$$

$$C_m = \left(\frac{L_m}{{D_m}^{4.87}}\right)^{0.54} \left(\frac{L_1}{{D_1}^{4.87}{C_1}^{1.85}} + \frac{L_2}{{D_2}^{4.87}{C_2}^{1.85}}\right)^{-0.54}$$

where D_m is the diameter of the merged pipe, D_1 and D_2 are the diameters of the original pipes, L_m is the length of the merged pipe, L_1 and L_2 are the lengths of the original pipes, C_m is the Hazen-Williams roughness coefficient of the merged pipe, and C_1 and C_2 are the Hazen-Williams roughness coefficients of the original pipes. Minor loss and pipe status of the merged pipe are set equal to the minor loss and pipe status for the pipe selected for maximum diameter. Note, if the original pipes have the same diameter, D_m is based on the pipe name that comes first in alphabetical order.

3. **Parallel pipe merge**: Pipes in parallel are merged if both pipes are below the pipe diameter threshold (Figure 20). This operation does not reduce the number of junctions in the system. The merged pipe is assigned the following equivalent properties:

$$D_m = max\left(D_1, D_2\right)$$

$$L_m = \text{Length of the pipe selected for max diameter}$$

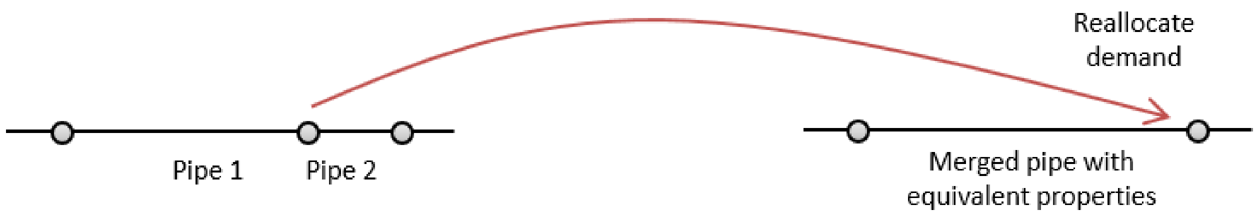


Figure 19: Series pipe merge.

$$C_m = \left(\frac{L_m^{0.54}}{{D_m}^{2.63}}\right)\left(\frac{C_1 {D_1}^{2.63}}{{L_1}^{0.54}} + \frac{C_2 {D_2}^{2.63}}{{L_2}^{0.54}}\right)$$

where D_m is the diameter of the merged pipe, D_1 and D_2 are the diameters of the original pipes, L_m is the length of the merged pipe, L_1 and L_2 are the lengths of the original pipes, C_m is the Hazen-Williams roughness coefficient of the merged pipe, and C_1 and C_2 are the Hazen-Williams roughness coefficients of the original pipes. Minor loss and pipe status of the merged pipe are set equal to the minor loss and pipe status for the pipe selected for max diameter. Note, if the original pipes have the same diameter, D_m is based on the pipe name that comes first in alphabetical order.

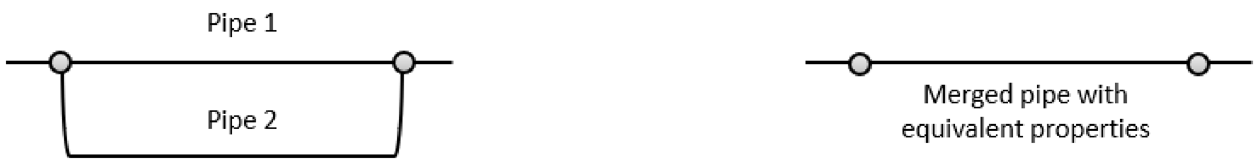


Figure 20: Parallel pipe merge.

The `skeletonize` function is used to perform network skeletonization. The iterative algorithm first loops over all candidate pipes (pipes below the pipe diameter threshold) and removes branch pipes. Then the algorithm loops over all candidate pipes and merges pipes in series. Finally, the algorithm loops over all candidate pipes and merges pipes in parallel. This initial set of operations can generate new branch pipes, pipes in series, and pipes in parallel. This cycle repeats until the network can no longer be reduced. The user can specify if branch trimming, series pipe merge, and/or parallel pipe merge should be included in the skeletonization operations. The user can also specify a maximum number of cycles to include in the process.

See the online API documentation for more information on skeletonization.

Results from network skeletonization include the skeletonized water network model and (optionally) a “skeletonization map” that maps original network nodes to merged nodes that are represented in the skeletonized network. The skeletonization map is a dictionary where the keys are original network nodes and the values are a list of nodes in the network that were merged as a result of skeletonization operations. For example, if ‘Junction 1’ was merged into ‘Junction 2’ and ‘Junction 3’ remained unchanged as part of network skeletonization, then the skeletonization map would contain the following information:

```
{
'Junction 1': [],
'Junction 2': ['Junction 1', 'Junction 2'],
'Junction 3': ['Junction 3']
}
```

This map indicates that the skeletonized network does not contain ‘Junction 1’, ‘Junction 2’ in the skeletonized network is the merged product of the original ‘Junction 1’ and ‘Junction 2,’ and ‘Junction 3’ was not changed. ‘Junction 2’ in the skeletonized network will therefore contain demand and demand patterns from the original ‘Junction 1’ and ‘Junction 2.’

The following example performs network skeletonization on Net6 and compares system pressure using the original and skeletonized networks. The example starts by creating a water network model for Net6, listing the number of network components (e.g., 3356 nodes, 3892 links), and then skeletonizing it using a using a pipe diameter threshold of 12 inches. The skeletonization procedure reduces the number of nodes in the network from approximately 3000 to approximately 1000 (Figure 21).

```
>>> import matplotlib.pylab as plt
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net6.inp')
>>> wn.describe()
{'Nodes': 3356, 'Links': 3892, 'Patterns': 3, 'Curves': 60, 'Sources': 0, 'Controls':␣
↪124}

>>> skel_wn = wntr.morph.skeletonize(wn, 12*0.0254)
>>> skel_wn.describe()
{'Nodes': 1154, 'Links': 1610, 'Patterns': 3, 'Curves': 60, 'Sources': 0, 'Controls':␣
↪124}

>>> fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))
>>> nodes, edges = wntr.graphics.plot_network(wn, node_size=10, title='Original',
...     ax=ax1)
>>> nodes, edges = wntr.graphics.plot_network(skel_wn, node_size=10,
...     title='Skeletonized', ax=ax2)
```

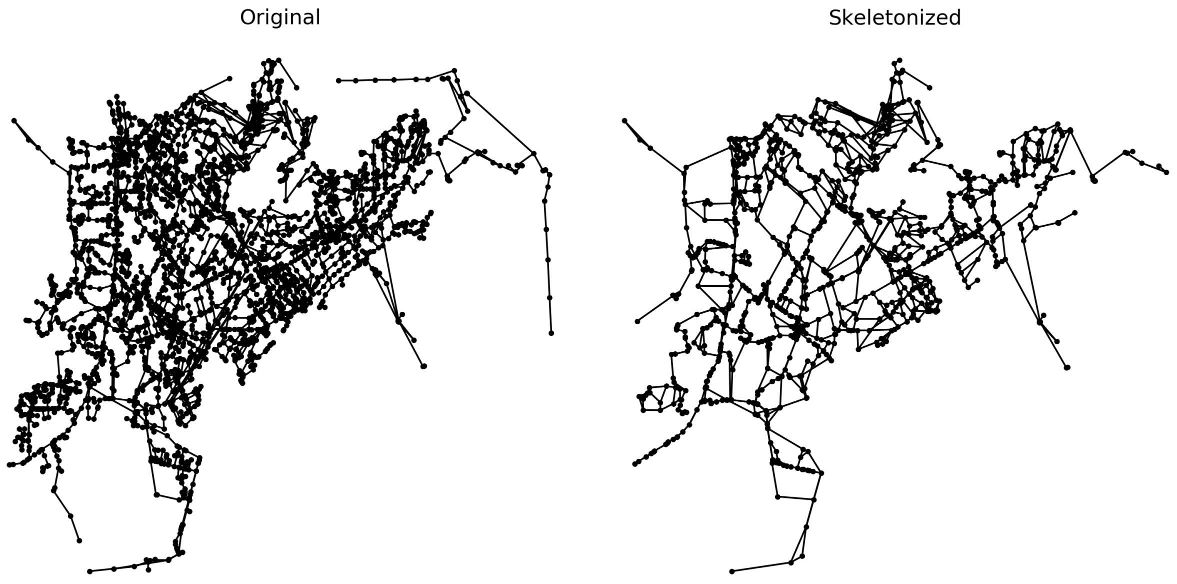


Figure 21: Original and skeletonized Net6.

Hydraulic are then simulated using the original and skeletonized networks.

```
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results_original = sim.run_sim()

>>> sim = wntr.sim.EpanetSimulator(skel_wn)
>>> results_skel = sim.run_sim()
```

The simulation results are used to compute the pressure difference between the original and skeletonized networks.

The pressure difference is computed at nodes that exist in the skeletonized network.

```
>>> skel_junctions = skel_wn.junction_name_list
>>> pressure_orig = results_original.node['pressure'].loc[:,skel_junctions]
>>> pressure_skel = results_skel.node['pressure'].loc[:,skel_junctions]
>>> pressure_diff = (abs(pressure_orig - pressure_skel)/pressure_orig)*100
>>> pressure_diff.index = pressure_diff.index/3600 # convert time to hours
```

The 25th, 50th (median) and 75th percentiles in pressure difference can then be extracted.

```
>>> m25 = pressure_diff.quantile(0.25, axis=1)
>>> m50 = pressure_diff.quantile(0.50, axis=1)
>>> m75 = pressure_diff.quantile(0.75, axis=1)
```

Figure 22 shows the median (dark blue line) and the 25th to 75th percentile (shaded region) for node pressure throughout the network over a 4 day simulation. Pressure differences are very small in this example.

```
>>> fig = plt.figure()
>>> ax = m50.plot()
>>> poly = ax.fill_between(m25.index, m25, m75, color='b', alpha=0.2)
>>> text = ax.set_xlabel('Time (hr)')
>>> text = ax.set_ylabel('Percent change in pressure (%)')
```

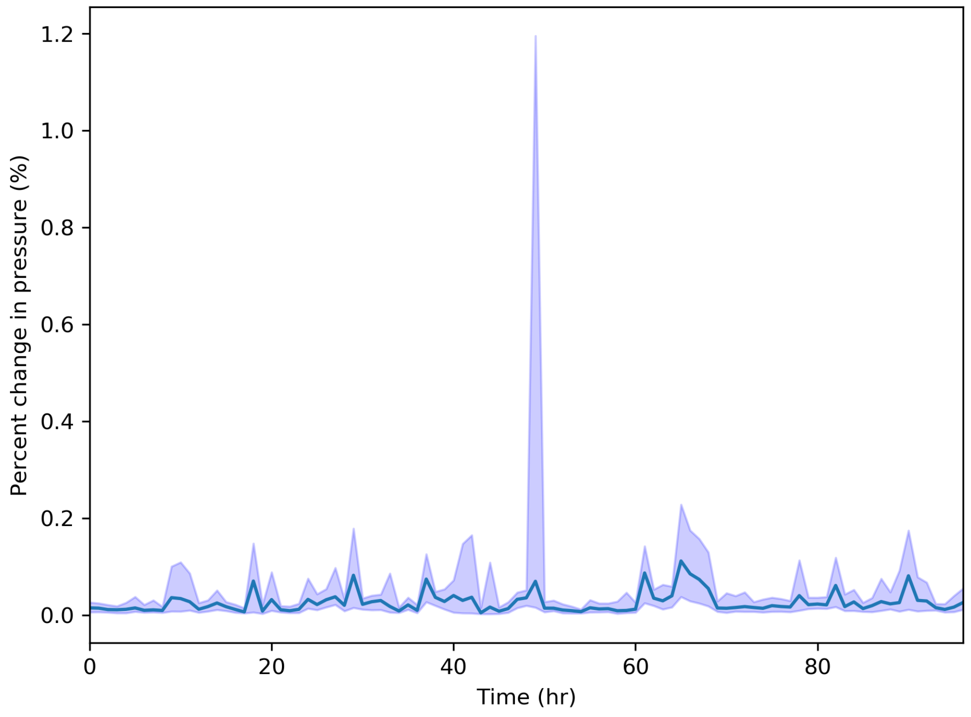


Figure 22: Pressure differences between the original and skeletonized Net6.

Modify node coordinates

WNTR includes several options to modify node coordinates, denoted as (x, y) below, including:

- **Scale coordinates**: Multiply coordinates by a scale factor (in meters) using the function `scale_node_coordinates`.

$$(x, y) = (x * scale, y * scale)$$

- **Translate coordinates**: Shift coordinates by an offset (in meters) in the x and y direction using the function `translate_node_coordinates`.

$$(x, y) = (x + offset_x, y + offset_y)$$

- **Rotate coordinates**: Rotate coordinates counterclockwise by θ degrees using the function `rotate_node_coordinates`.

$$(x, y) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \cdot (x, y)$$

- **Convert coordinates between UTM and longitude/latitude**: Convert coordinates from UTM to longitude/latitude or visa-versa using the functions `convert_node_coordinates_UTM_to_longlat` and `convert_node_coordinates_longlat_to_UTM`.
- **Convert coordinates to UTM or longitude/latitude**: Convert coordinates from arbitrary distance units directly into UTM or longitude/latitude using the functions `convert_node_coordinates_to_UTM` and `convert_node_coordinates_to_longlat`. The user supplies the names of two nodes in their network along with their UTM or longitude/latitude coordinates. Ideally, these nodes span a decent range of the network (for example, the nodes could be in the upper right and lower left).

Note: Functions that convert coordinates to UTM and longitude/latitude require the Python package **utm** [Bieni19], which is an optional dependency of WNTR.

The following example returns a copy of the water network model with node coordinates scaled by 100 m.

```
>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> wn_scaled_coord = wntr.morph.scale_node_coordinates(wn, 100)
```

The next example converts node coordinates to longitude/latitude. The longitude and latitude coordinates of two locations (e.g., nodes, tanks) on the map must be provided to convert the other node coordinates to longitude/latitude.

```
>>> longlat_map = {'Lake':(-106.6587, 35.0623), '219': (-106.5248, 35.1918)}
>>> wn_longlat = wntr.morph.convert_node_coordinates_to_longlat(wn, longlat_map)
```

Split or break pipes

WNTR includes the functions `split_pipe` and `break_pipe` to split or break a pipe.

For a pipe split, the original pipe is split into two pipes by adding a new junction and new pipe to the model. For a pipe break, the original pipe is broken into two disconnected pipes by adding two new junctions and a new pipe to the model.

Note: With a pipe break, flow is no longer possible from one side of the break to the other. This is more likely to introduce non-convergable hydraulics than a pipe split with a leak added.

The updated model retains the original length of the pipe section (Figure 23). The split or break occurs at a user specified distance between the original start and end nodes of the pipe (in that direction). The new pipe can be added to either end of the original pipe.

- The new junction has a base demand of 0 and the default demand pattern. The elevation and coordinates of the new junction are based on a linear interpolation between the end points of the original pipe.
- The new pipe has the same diameter, roughness, minor loss, and base status of the original pipe.
- Check valves are not added to the new pipe. Since the new pipe can be connected at either the start or the end of the original pipe, the user can control if the split occurs before or after a check valve.
- No controls are added to the new pipe; the original pipe keeps any controls.

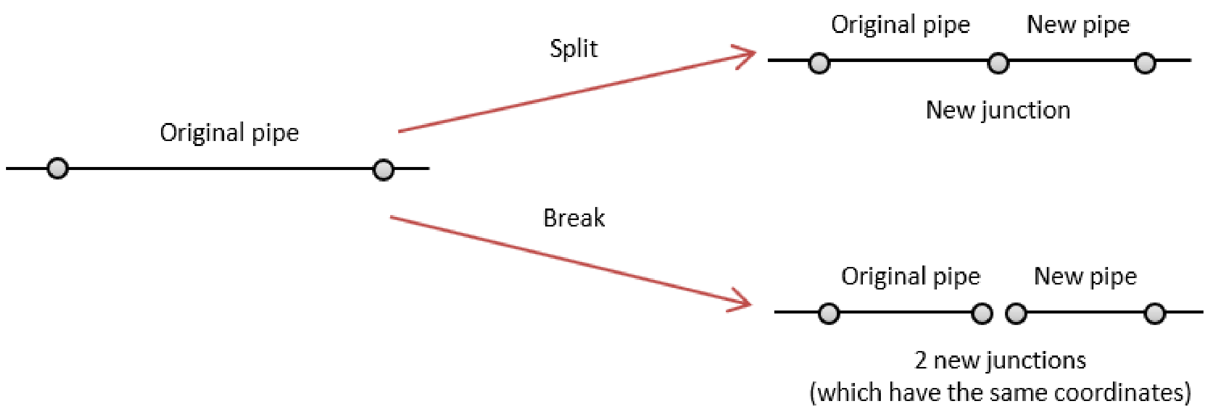


Figure 23: Pipe split and pipe break.

The following example splits pipe '123' in Net3 into pipes '123' and '123_B.' The new junction is named '123_node.' The new node is then used to add a leak to the model.

```
>>> wn = wntr.morph.split_pipe(wn, '123', '123_B', '123_node')
>>> leak_node = wn.get_node('123_node')
>>> leak_node.add_leak(wn, area=0.05, start_time=2*3600, end_time=12*3600)
```

18 Graphics

WNTR includes several functions to plot water network models and to plot fragility and pump curves.

Networks

Basic network graphics can be generated using the function `plot_network`. A wide range of options can be supplied, including node attributes, node size, node range, link attributes, link width, and link range.

Node and link attributes can be specified using the following options:

- Name of the attribute (i.e., 'elevation' for nodes or 'length' for links), this calls `query_node_attribute` or `query_link_attribute` method on the water network model and returns a pandas Series with node/link names and associated values
- Pandas Series with node/link names and associated values, this option is useful to show simulation results (i.e., `results.node['pressure'].loc[5*3600, :]`) and metrics (i.e., `wntr.metrics.population(wn)`)
- Dictionary with node/link names and associated values (similar to pandas Series)
- List of node/link names (i.e., `['123', '199']`), this highlights the node or link in red

The following example plots the network along with node elevation (Figure 24). Note that the `plot_network` function returns matplotlib objects for the network nodes and edges, which can be further customized by the user.

```
>>> import wntr

>>> wn = wntr.network.WaterNetworkModel('networks/Net3.inp')
>>> nodes, edges = wntr.graphics.plot_network(wn, node_attribute='elevation',
...     node_colorbar_label='Elevation (m)')
```

Interactive plotly networks

Interactive plotly network graphics can be generated using the function `plot_interactive_network`. This function produces an HTML file that the user can pan, zoom, and hover-over network elements. As with basic network graphics, a wide range of plotting options can be supplied. However, link attributes currently cannot be displayed on the graphic.

Note: This function requires the Python package **plotly** [SPHC16], which is an optional dependency of WNTR.

The following example plots the network along with node population (Figure 25).

```
>>> pop = wntr.metrics.population(wn)
>>> wntr.graphics.plot_interactive_network(wn, node_attribute=pop,
...     node_range=[0,500], filename='population.html', auto_open=False)
```

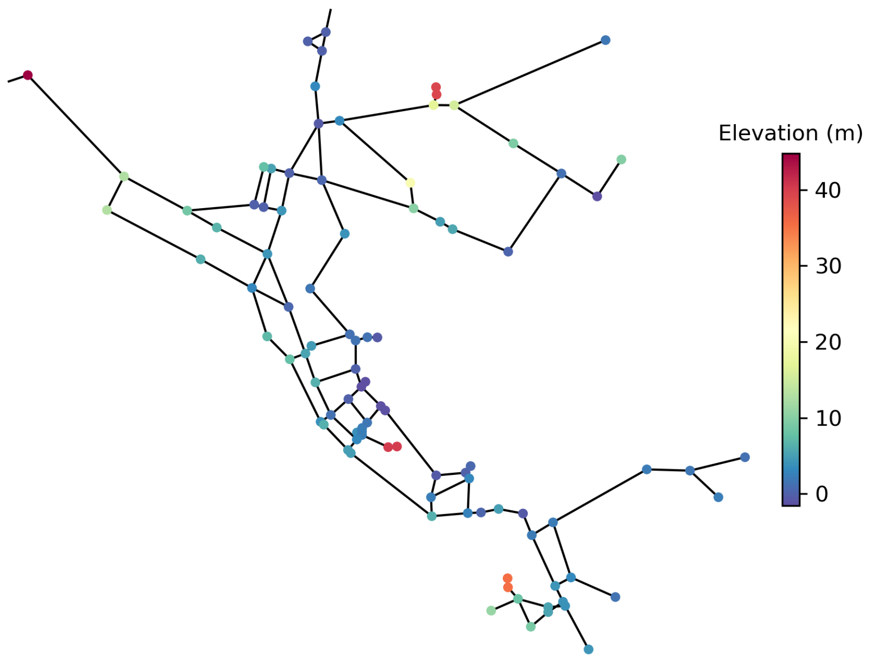


Figure 24: Basic network graphic.

Interactive Leaflet networks

Interactive Leaflet network graphics can be generated using the function `plot_leaflet_network`. This function produces an HTML file that overlays the network model onto a Leaflet map. Leaflet is an open-source JavaScript library for mobile-friendly interactive maps. More information on Leaflet is provided at https://leafletjs.com/. The network model should have coordinates in longitude/latitude. See *Modify node coordinates* for more information on converting node coordinates. As with basic network graphics, a wide range of plotting options can be supplied.

Note: This function requires the Python package **folium** [Folium], which is an optional dependency of WNTR.

The following example using EPANET Example Network 3 (Net3) converts node coordinates to longitude/latitude and plots the network along with pipe length over the city of Albuquerque (for demonstration purposes only) (Figure 26). The longitude and latitude for two locations are needed to plot the network. For the EPANET Example Network 3, these locations are the reservoir 'Lake' and node '219'. This example requires the Python package **utm** [Bieni19] to convert the node coordinates.

```
>>> longlat_map = {'Lake':(-106.6851, 35.1344), '219': (-106.5073, 35.0713)}
>>> wn2 = wntr.morph.convert_node_coordinates_to_longlat(wn, longlat_map)
>>> length = wn2.query_link_attribute('length')
>>> wntr.graphics.plot_leaflet_network(wn2, link_attribute=length, link_width=3,
...                                    link_range=[0,1000], filename='length.html')
```

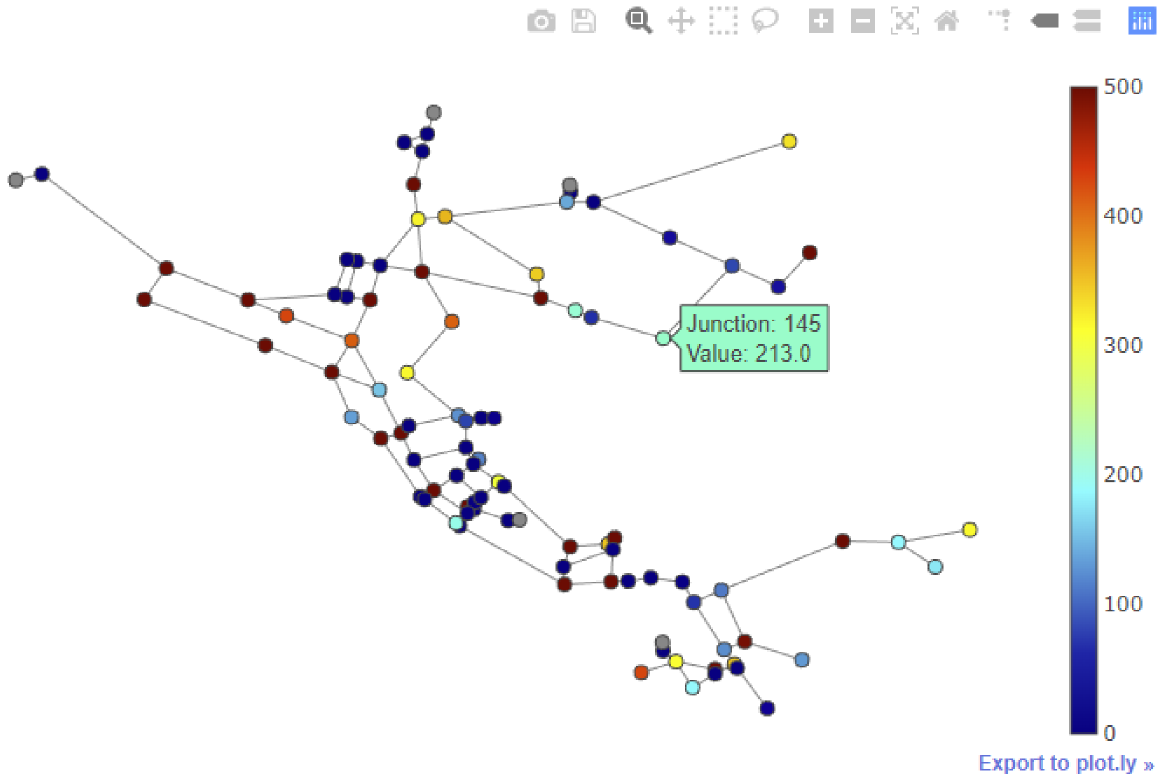


Figure 25: Interactive network graphic with the legend showing the node population.

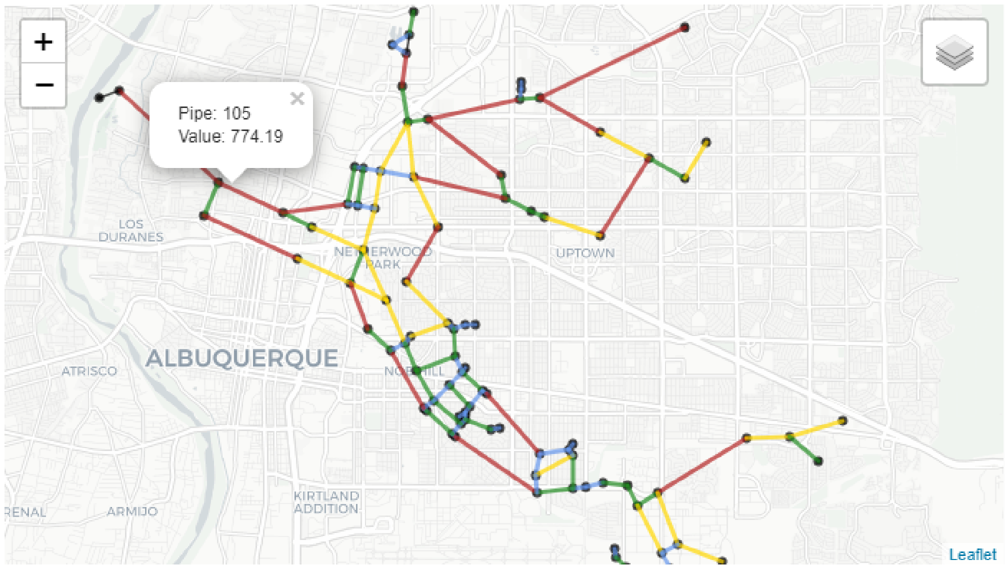


Figure 26: Interactive Leaflet network graphic.

Network animation

Network animation can be generated using the function `network_animation`. Node and link attributes can be specified using pandas DataFrames, where the index is time and columns are the node or link name.

The following example creates a network animation of water age over time.

```
>>> wn.options.quality.mode = 'AGE'
>>> sim = wntr.sim.EpanetSimulator(wn)
>>> results = sim.run_sim()
>>> water_age = results.node['quality']/3600 # convert seconds to hours
>>> anim = wntr.graphics.network_animation(wn, node_attribute=water_age, node_
↪range=[0,24])
```

Time series

Time series graphics can be generated using options available in Matplotlib and pandas.

The following example plots simulation results from above, showing pressure at a single node over time (Figure 27).

```
>>> pressure_at_node123 = results.node['pressure'].loc[:,'123']
>>> ax = pressure_at_node123.plot()
>>> text = ax.set_xlabel("Time (s)")
>>> text = ax.set_ylabel("Pressure (m)")
```

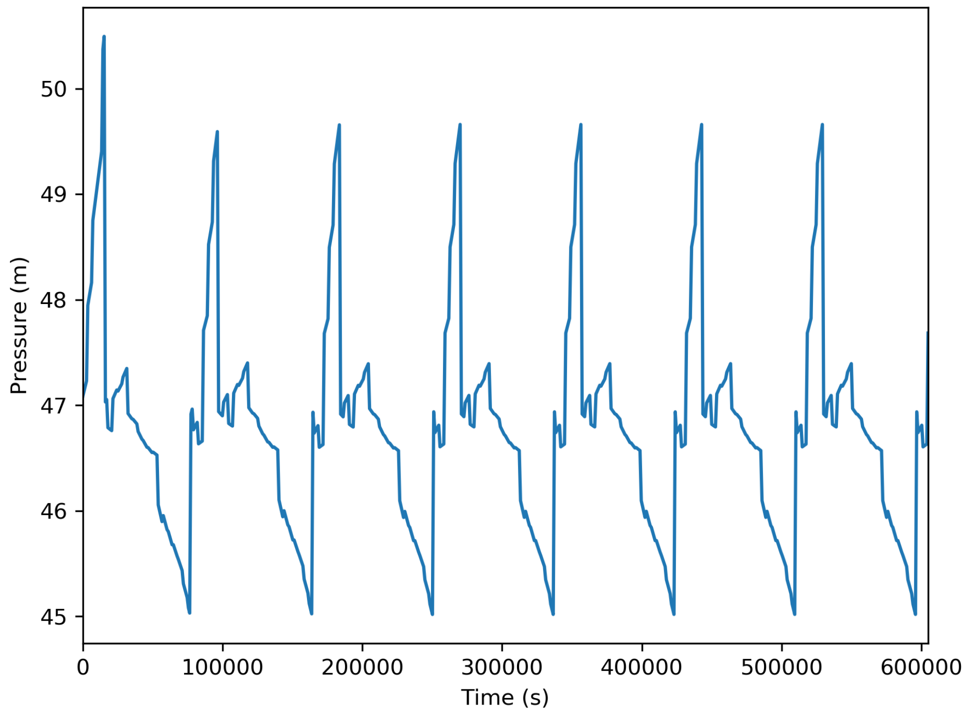


Figure 27: Time series graphic.

Interactive time series

Interactive time series graphics are useful when visualizing large datasets. Basic time series graphics can be converted to interactive time series graphics using the `plotly.express` module.

Note: This functionality requires the Python package **plotly** [SPHC16], which is an optional dependency of WNTR.

The following example uses simulation results from above, and converts the graphic to an interactive graphic (Figure 28).

```
>>> import plotly.express as px

>>> tankH = results.node['pressure'].loc[:,wn.tank_name_list]
>>> tankH = tankH * 3.28084 # Convert tank height to ft
>>> tankH.index /= 3600 # convert time to hours
>>> fig = px.line(tankH)
>>> fig = fig.update_layout(xaxis_title='Time (hr)', yaxis_title='Head (ft)',
...                     template='simple_white', width=650, height=400)
>>> fig.write_html('tank_head.html')
```

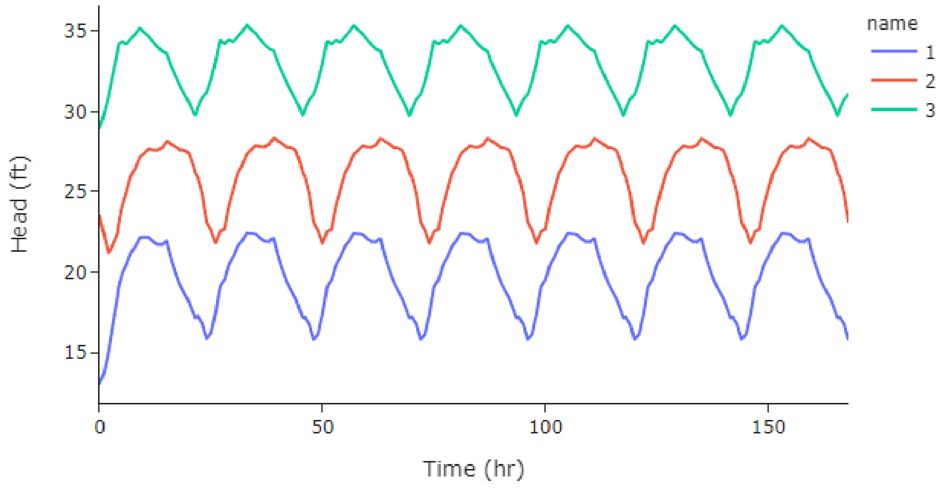


Figure 28: Interactive time series graphic with the tank heights for Tank 1 (blue), Tank 2 (orange), and Tank 3 (green).

Fragility curves

Fragility curves can be plotted using the function `plot_fragility_curve`.

The following example plots a fragility curve with two states (Figure 29).

```
>>> from scipy.stats import lognorm

>>> FC = wntr.scenario.FragilityCurve()
>>> FC.add_state('Minor', 1, {'Default': lognorm(0.5,scale=0.3)})
>>> FC.add_state('Major', 2, {'Default': lognorm(0.5,scale=0.7)})
>>> ax = wntr.graphics.plot_fragility_curve(FC, xlabel='Peak Ground Acceleration (g)')
```

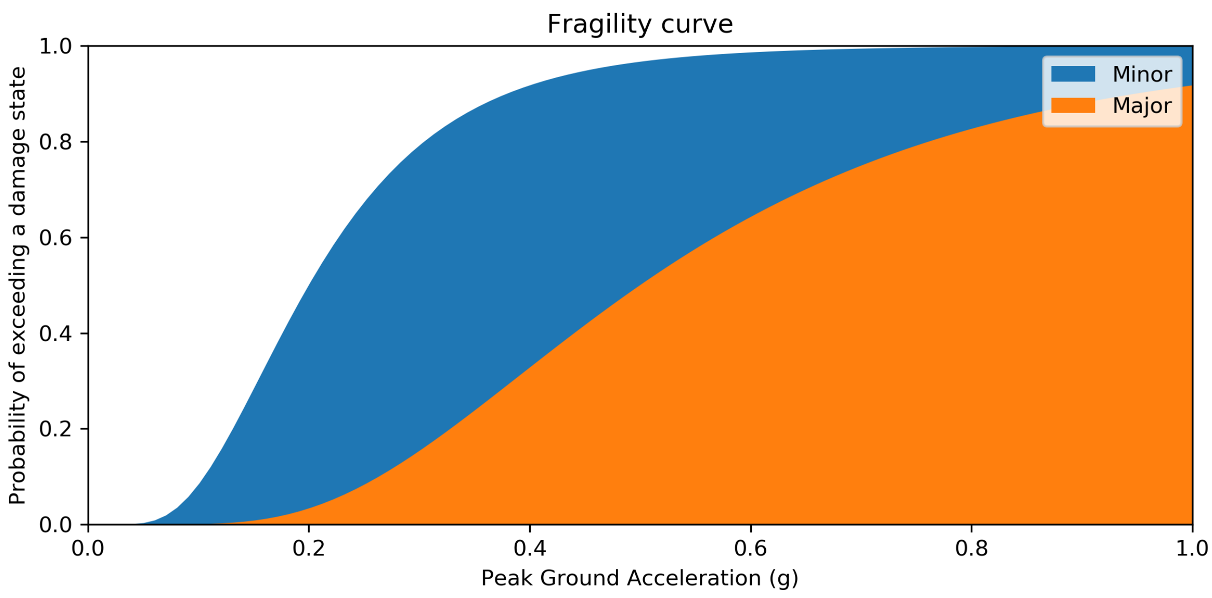


Figure 29: Fragility curve graphic.

Pump curves

Pump curves can be plotted using the function `plot_pump_curve`. By default, a 2nd order polynomial is included in the graphic.

The following example plots a pump curve (Figure 30).

```
>>> pump = wn.get_link('10')
>>> ax = wntr.graphics.plot_pump_curve(pump)
```

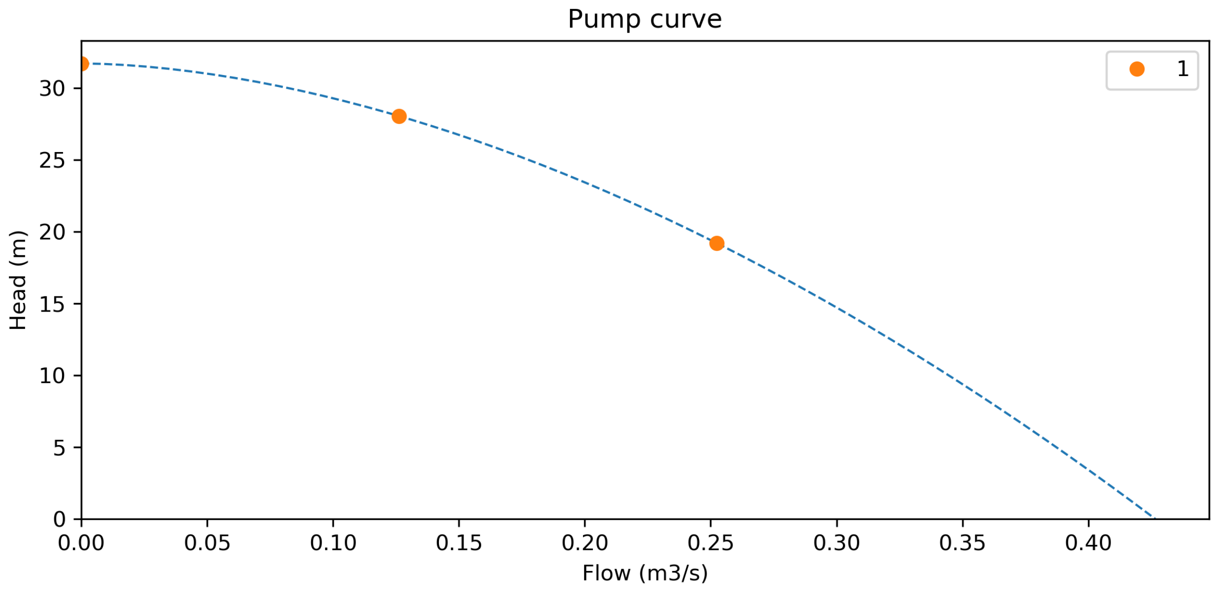


Figure 30: Pump curve graphic.

19 Copyright and license

The WNTR Python package is copyright through Sandia National Laboratories. The software is distributed under the Revised BSD License. WNTR also leverages a variety of third-party software packages, which have separate licensing policies.

Copyright

```
Copyright 2019 National Technology & Engineering Solutions of Sandia,
LLC (NTESS). Under the terms of Contract DE-NA0003525 with NTESS, the U.S.
Government retains certain rights in this software.
```

Revised BSD license

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.
* Neither the name of Sandia National Laboratories, nor the names of
  its contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

20 Software quality assurance

The following section includes information about the WNTR software repository, software tests, documentation, examples, bug reports, feature requests, and ways to contribute.

GitHub repository

WNTR is maintained in a version controlled repository. WNTR is hosted on US EPA GitHub organization at https://github.com/USEPA/WNTR.

Software tests

WNTR includes continuous integration software tests that are run using GitHub Actions. Travis CI and AppVeyor are used by the core development team as secondary testing services. The tests are run each time changes are made to the repository. The tests cover a wide range of unit and integration tests designed to ensure that the code is performing as expected. New tests are developed each time new functionality is added to the code. Testing status (passed/failed) and code coverage statistics are posted on the README section at https://github.com/USEPA/WNTR.

Tests can also be run locally using the Python package nose. For more information on nose, see http://nose.readthedocs.io/. The nose package comes with a command line software tool called nosetests. Tests can be run in the WNTR directory using the following command in a command line/PowerShell prompt:

```
nosetests -v --with-coverage --cover-package=wntr wntr
```

In addition to the publicly available software tests run using GitHub Actions, WNTR is also tested on private servers using several large water utility network models.

Documentation

WNTR includes a user manual that is built using the Read the Docs service. The user manual is automatically rebuilt each time changes are made to the code. The documentation is publicly available at http://wntr.readthedocs.io/. The user manual includes an overview, installation instructions, simple examples, and information on the code structure and functions. WNTR includes documentation on the API for all public functions, methods, and classes. New content is marked *Draft*.

Examples

WNTR includes examples to help new users get started. These examples are intended to demonstrate high level features and use cases for WNTR. The examples are tested to ensure they stay current with the software project.

Bug reports and feature requests

Bug reports and feature requests can be submitted to https://github.com/USEPA/WNTR/issues. The core development team will prioritize and assign bug reports and feature requests to team members.

Contributing

Software developers, within the core development team and external collaborators, are expected to follow standard practices to document and test new code. Software developers interested in contributing to the project are encouraged to create a *Fork* of the project and submit a *Pull Request* using GitHub. Pull requests will be reviewed by the core development team.

Pull requests must meet the following minimum requirements to be included in WNTR:

- Code is expected to be documented using Read the Docs.
- Code is expected have sufficient tests. *Sufficient* is judged by the strength of the test and code coverage. An 80% code coverage is recommended.
- Large files (> 1Mb) will not be committed to the repository without prior approval.
- Network model files will not be duplicated in the repository. Network files are stored in examples/network and wntr/tests/networks_for_testing only.

Development team

WNTR was developed as part of a collaboration between the United States Environmental Protection Agency Office of Research and Development, Sandia National Laboratories, and Purdue University. See https://github.com/USEPA/WNTR/graphs/contributors for a full list of contributors.

21 References

[ALA01] American Lifelines Alliance. (2001). Seismic Fragility Formulations for Water Systems, Part 1 and 2. Report for the American Lifelines Alliance, ASCE (Ed.) Reston, VA: American Society of Civil Engineers. April 2001.

[AwGB90] Awumah, K., Goulter, I., and Bhatt, S.K. (1990). Assessment of reliability in water distribution networks using entropy based measures. Stochastic Hydrology and Hydraulics, 4(4), 309-320.

[BaRR13] Barker, K., Ramirez-Marquez, J.E., and Rocco, C.M. (2013). Resilience-based network component importance measures. Reliability Engineering and System Safety, 117, 89-97.

[Bieni19] Bieniek, T., van Andel, B., and Bø, T.I. (2019). Bidirectional UTM-WGS84 converter for python, Retrieved on February 5, 2019 from https://github.com/Turbo87/utm

[CrLo02] Crowl, D.A., and Louvar, J.F. (2011). Chemical Process Safety: Fundamentals with Applications, 3 edition. Upper Saddle River, NJ: Prentice Hall, 720p.

[ELLT12] Eto, J.H., LaCommare, K.H., Larsen, P.H., Todd, A., and Fisher, E. (2012). An Examination of Temporal Trends in Electricity Reliability Based on Reports from U.S. Electric Utilities. Lawrence Berkeley National Laboratory Report Number LBNL-5268E. Berkeley, CA: Ernest Orlando Lawrence Berkeley National Laboratory, 68p

[Folium] python-visualization/folium. (n.d.). Retrieved on February 5, 2019 from https://github.com/python-visualization/folium

[GaCl18] Gazoni, E. and Clark, C. (2018) openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files, Retrieved on May 4, 2018 from https://openpyxl.readthedocs.io.

[HaSS08] Hagberg, A.A., Schult, D.A., and Swart, P.J. (2008). Exploring network structure, dynamics, and function using NetworkX. In Proceedings of the 7th Python in Science Conference (SciPy2008), August 19-24, Pasadena, CA, USA.

[Hunt07] Hunter, J.D. (2007). Matplotlib: A 2D graphics environment. Computing in Science and Engineering, 9(3), 90-95.

[ICC12] International Code Council. (2011). 2012 International Fire Code, Appendix B - Fire-Flow Requirements for Buildings. Country Club Hills, IL: International Code Council, ISBN: 978-1-60983-046-5.

[JCMG11] Joyner, D., Certik, O., Meurer, A., and Granger, B.E. (2012). Open source computer algebra systems, SymPy. ACM Communications in Computer Algebra, 45(4), 225-234.

[Lamb01] Lambert, A. (2001). What do we know about pressure-leakage relationships in distribution systems. Proceedings of the IWA Specialised Conference 'System Approach to Leakage Control and Water Distribution Systems Management', Brno, Czech Republic, 2001, May 16-18, 89-96.

[LWFZ17] Liu, H., Walski, T., Fu, G., Zhang, C. (2017). Failure Impact Analysis of Isolation Valves in a Water Distribution Network. Journal of Water Resources Planning and Management 143(7): 04017019.

[Mcki13] McKinney, W. (2013). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. Sebastopal, CA: O'Reilly Media, 1 edition, 466p.

[NIAC09] National Infrastructure Advisory Council (NIAC). (2009). Critical Infrastructure Resilience, Final Report and Recommendations, U.S. Department of Homeland Security, Washington, D.C., Accessed September 20, 2014. http://www.dhs.gov/xlibrary/assets/niac/niac_critical_infrastructure_resilience.pdf.

[OsKS02] Ostfeld, A., Kogan, D., and Shamir, U. (2002). Reliability simulation of water distribution systems - single and multiquality. Urban Water, 4(1), 53-61.

[Ross00] Rossman, L.A. (2000). EPANET 2 Users Manual. Cincinnati, OH: U.S. Environmental Protection Agency. U.S. Environmental Protection Agency Technical Report, EPA/600/R–00/057, 200p.

[SOKZ12] Salomons, E., Ostfeld, A., Kapelan, Z., Zecchin, A., Marchi, A., and Simpson, A. (2012). The battle of the water networks II - Problem description. Water Distribution Systems Analysis Conference 2012, September 24-27, Adelaide, South Australia, Australia. Retrieved on May 23, 2017 from https://emps.exeter.ac.uk/media/universityofexeter/emps/research/cws/downloads/WDSA2012-BWNII-ProblemDescription.pdf.

[SPHC16] Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2016). plotly: Create interactive web graphics via Plotly's JavaScript graphing library [Software].

[Todi00] Todini, E. (2000). Looped water distribution networks design using a resilience index based heuristic approach. Urban Water, 2(2), 115-122.

[USEPA14] United States Environmental Protection Agency. (2014). Systems Measures of Water Distribution System Resilience. Washington DC: U.S. Environmental Protection Agency. U.S. Environmental Protection Agency Technical Report, EPA 600/R–14/383, 58p.

[USEPA15] United States Environmental Protection Agency. (2015). Water Security Toolkit User Manual. Washington DC: U.S. Environmental Protection Agency. U.S. Environmental Protection Agency Technical Report, EPA/600/R-14/338, 187p.

[VaCV11] van der Walt, S., Colbert, S.C., and Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. Computing in Science and Engineering, 13, 22-30.

[WaSM88] Wagner, J.M., Shamir, U., and Marks, D.H. (1988). Water distribution reliability: Simulation methods. Journal of Water Resources Planning and Management, 114(3), 276-294.

[WaWC06] Walski, T., Weiler, J. Culver, T. (2006). Using Criticality Analysis to Identify Impact of Valve Location. Water Distribution Systems Analysis Symposium. Cincinnati, OH, American Society of Civil Engineers: 1-9.

[WWQP06] Wald, D.J., Worden, B.C., Quitoriano, V., and Pankow, K.L. (2006). ShakeMap manual: Technical manual, user's guide, and software guide. United States Geologic Survey, Retrieved on April 25, 2017 from http://pubs.usgs.gov/tm/2005/12A01/

[WCSG03] Walski, T.M., Chase, D.V., Savic, D.A., Grayman, W., Beckwith, S. (2003). Advanced Water Distribution Modeling and Management. HAESTAD Press, Waterbury, CT, 693p.

EPA
United States
Environmental Protection
Agency

PRESORTED STANDARD
POSTAGE & FEES PAID
EPA
PERMIT NO. G-35

Office of Research and Development (8101R)
Washington, DC 20460

Official Business
Penalty for Private Use
$300