University of Stuttgart
Germany

Complex Network Systems

NetworkX tutorial

Ilche Georgievski
ilche.georgievski@iaas.uni-stuttgart.de
Room: U38 0.353

2019/2020
Winter

- Installation
- Basic classes
- Generating graphs
- Analysing graphs
- Save/load
- Plotting (Matplotlib)

- Python package for the creation, manipulation, and analysis of the structure, dynamics and functions of complex networks
  - API
  - Graph implementation
  - Interface to existing numerical algorithms
  - Load and store networks in standard and nonstandard data formats

- Excellent online documentation

Installation

- Install manually from https://pypi.org/project/networkx/
- use pip `$ sudo pip install networkx`
- or use Debian package manager `$ sudo apt-get install python-networkx`

Initialise a graph

- Constructor
  - No parameters for an empty graph
  - List of edges as node pairs

```
import networkx as nx
G = nx.Graph()
G.add_node("pumpkin")
G.add_edge(1,2)
print(G.nodes())
# ['pumpkin', 1, 2]
print(G.edges())
# [(1, 2)]
```

Graph types

- Graph: undirected simple (allows self-loops)
- DiGraph: directed simple (allows self-loops)
- MultiGraph: undirected with parallel edges
- MultiDiGraph: directed with parallel edges
- Convert do undirected: `g.to_undirected()`
- Convert to directed: `g.to_directed()`

To construct, use standard python syntax

```
g = nx.Graph()
d = nx.DiGraph()
m = nx.MultiGraph()
md = nx.MultiDiGraph()
g.to_directed()
md.to_undirected()
```

Explore nodes and edges

- Nodes and edges can be viewed using set-like views `g.nodes` and `g.edges`
- `g.nodes()` returns a node view
- `g.edges()` returns an edge view
- Lookup access `g.nodes[x]` and via iteration `g.edges.items()`
- Instead of a view, other container types can be specified (list, set, dict, tuple)
  - `list(g.nodes)` and `list(g.edges)`

Add and remove nodes

- `add_nodes_from()` **and** `remove_nodes_from()` **take any iterable collection and any object**

```
g = nx.Graph()
g.add_node("pumpkin")
g.add_nodes_from(["calcium", "kale", "broccoli"])
g.add_nodes_from('ABC')
h = nx.path_graph(5)
g.add_nodes_from(h)
print(g.nodes)
# ['pumpkin', 'calcium', 'kale', 'broccoli', 'A', 'B', 'C', 0, 1, 2, 3, 4]

g.remove_node("kale")
g.remove_nodes_from(['A', 4])
print(g.nodes)
# ['pumpkin', 'calcium', 'broccoli', 'B', 'C', 0, 1, 2, 3]
```

Add and remove edges

- Adding an edge between non-existent nodes will automatically add those nodes
- `add_edges_from()` takes any iterable collection and any type (anything that has a `__iter()__` method)

```
g = nx.Graph([('a', 'b'), ('b', 'c'), ('c', 'a')])
g.add_edge('a', 'd')
g.add_edges_from([('d', 'c'), ('d', 'b')])
print(g.nodes)
# ['a', 'b', 'c', 'd']
print(g.edges)
# [('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]
g.remove_edge('c', 'a')
print(g.edges)
# [('a', 'b'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]
```

Adding node and edge attributes

- An attribute is implemented as a dictionary (key, value) associated with a node or edge
  - dictionary keys are attribute names (immutable)
- Type indifferent, but it needs to be hashable
  - i.e., cannot use `list`, it must be `tuple`

```
Traceback (most recent call last):
  File "C:/Users/ilche/storage/cloud/work/us/teaching/cns/2019/exercises/03-networkx-tutorial/basic_networkx.py", line 43, in <module>
    g.add_node([1,2])
  File "C:\Users\ilche\storage\cloud\work\us\teaching\cns\2019\exercises\03-networkx-tutorial\venv\lib\site-packages\networkx\classes\graph.py", line 516, in add_node
    if node_for_adding not in self._node:
TypeError: unhashable type: 'list'
```

NetworX does not enforce consistency among attribute dictionaries

Node attributes

- Add attributes when adding nodes
- When several nodes are added with `add_nodes_from()`, only one set of attributes for all of them can be specified

```
g = nx.Graph()
g.add_node('nuts', edible=True)
g.add_nodes_from(['kale', 'broccoli'], edible=False)
print(g.nodes['nuts']['edible'])
# True
print(g.nodes['broccoli'])
# {'edible': False}
```

- Add or modify attributes of existing nodes and edges
  - `set_node_attributes(g, node_dict, att_name)`

Edge attributes

- Add attributes when adding edges

```
g = nx.Graph()
g.add_edge('nuts', 'copper', weight=0)
g.add_edges_from([('a', 'b', {'color': 'red'}), ('b', 'c', {'weight': 2.2})])
print(g.edges['nuts', 'copper']['weight'])
# 0
print(g.edges['b', 'c'])
# {'weight': 2.2}
g['nuts']['copper']['weight'] = 0.9
print(g.edges['nuts', 'copper'])
# {'weight': 0.9}
```

- Add or modify attributes of existing edges
  - `set_edge_attributes(g, edge_dict, att_name)`

Merge nodes

- `nx.contracted_nodes(F, u, v)`
  - merges node *v* into node *u* in the graph *F*
  - reassignes all edges previously incident to *v*, to *u*
  - if option `self_loops=False` is not passed, it converts an edge from *v* to *u* to a self-loop
  - as a side effect, it creates a new node attribute called contraction

- Can be used to eliminate duplicate nodes in a network

Simple properties

- Number of nodes

```
g = nx.Graph()
len(g)
g.number_of_nodes()
g.order()
```

- Number of edges

```
g = nx.Graph()
g.number_of_edges()
g.size()
```

- Check node membership

```
g = nx.Graph()
v = g.has_node('nuts')
print(v)
# False
```

- Check edge presence

```
g = nx.Graph()
g.add_edge('nuts', 'copper')
v = g.has_edge('nuts', 'copper')
print(v)
# True
```

Neighbours

- Iterating over edges (can be useful for efficiency)

```
g = nx.Graph()
nx.add_path(g, [0, 1, 2, 3])
for e in g.edges.items():
    print(e)
# ((0, 1), {})
# ((1, 2), {})
# ((2, 3), {})
for n, nbrs in g.adjacency():
    print((n, nbrs))
# (0, {1: {}})
# (1, {0: {}, 2: {}})
# (2, {1: {}, 3: {}})
# (3, {2: {}})
```

Degrees

```
g = nx.Graph()
g.add_edges_from([(1, 2), (2, 1), (2, 3), (3, 2), (3, 4)])
d1 = g.degree(1)
print(d1)
# 1
d23 = g.degree([2, 3])
print(d23)
# [(2, 2), (3, 2)]
d = g.degree
print(d)
# [(1, 1), (2, 2), (3, 2), (4, 1)]
```

Simple graph generators

- Complete graph
  - `nx.complete_graph(50)`
- Chain
  - `nx.path_graph(5)`
- Bipartite
  - `nx.complete_bipartite_graph(n1, n2)`

Random graph generators

- Preferential attachment
  - `nx.barbasi_albert_graph(n, m)`
- $G_{n,p}$
  - `nx.gnp_random_graph(n, p)`
  - `nx.gnm_random_graph(n, m)`
  - `nx.watts_strogatz_graph(n, k, p)`

Algorithms

- Some imported in the top-level `networkx`, while others can be found in `networkx.algorithms`
- Bipartite
- Centrality
- Clique
- Clustering
- Communities
- Components
- Connectivity
- Directed acyclic graphs
- Distance measures
- Isolates
- Shortest path
- Triads

A few useful functions

- Subgraphs
  - `g.subgraph([1, 2, 3])`
  - `nx.strongly_connected_components(g)`
- Operations on graphs
  - `nx.union(g, h)`
  - `nx.intersection(g, h)`
  - `nx.complement(g)`
- Shortest path
  - `nx.shortest_path(g, source, target)`
  - `nx.betweenness_centrality(g)`
- Clustering
  - `nx.average_clustering(g)`
- Diameter
  - `nx.diameter(g)`

Read a network from a CSV file

- Read a network from a file and construct a graph

- CSV edge reader
  - open an edge list file:
    - `with open("healthy_nutrition.csv") as infile:`
  - create a CSV reader for the file:
    - `csv.reader(infile)`
  - put the list of pairs into the graph constructor:
    - `nx.Graph(csv_reader)`

Store a network

Format	Attributes	Reader	Writer
Adjacency list	Not stored	nx.read_adjlist()	nx.write_adjlist()
Edge list	Not stored	nx.read_edgelist()	nx.write_edgelist()
Graph exchange XML format	Stored	nx.read_gexf()	nx.write_gexf()
Graph modeling language	Stored	nx.read_gml()	nx.write_gml()
GraphML	Stored	nx.read_graphml()	nx.write_graphml()
Pajek NET	Not stored	nx.read_pajek()	nx.write_pajek()
Pickle	Stored	nx.read_gpickle()	nx.write_gpickle()
YAML	Stored	nx.read_yaml()	nx.write_yaml()

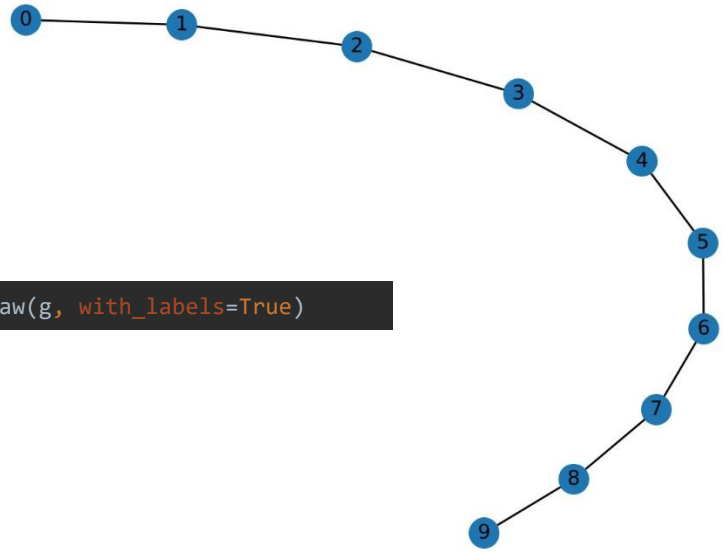
Matplotlib

- Python package which emulates matlab funcitonality
  - Well documented at https://matplotlib.org/contents.html
- Interfaces nicely with NetworkX
- Depends on Numpy, which provides multidimensional array support
  - https://numpy.org/
- May be needed for plotting

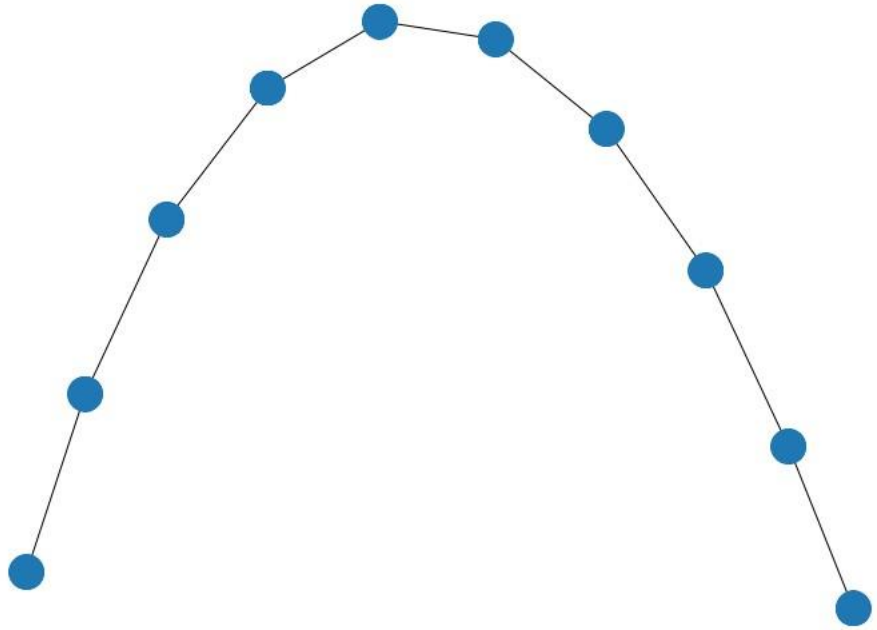
```
import matplotlib.pyplot as plt
plt.plot(range(10), range(10))
plt.show()
```

Basic graph drawing

```
import networkx as nx
import matplotlib.pyplot as plt
g = nx.path_graph(10)
nx.draw(g)
plt.savefig("C:/Users/../path_graph.pdf")
```



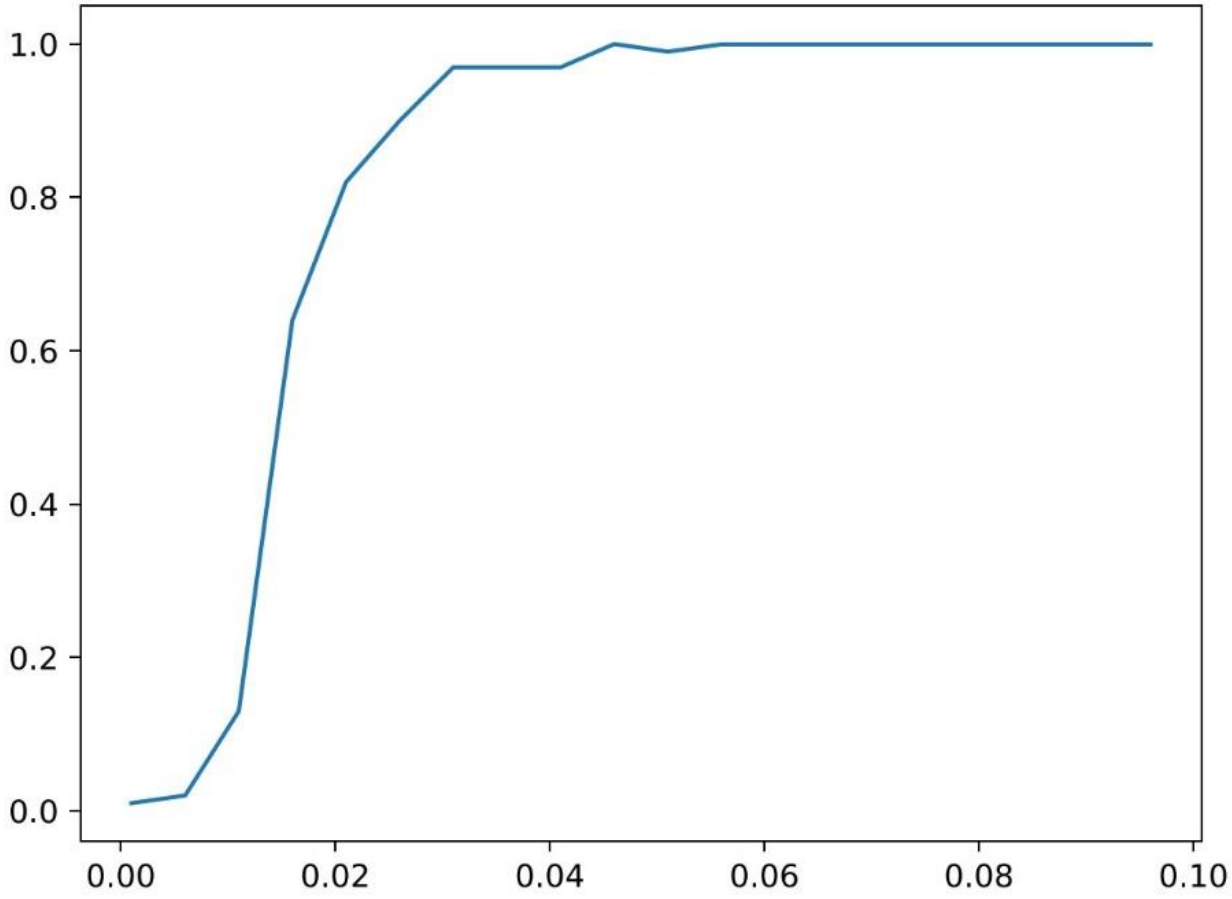

```
nx.draw(g, with_labels=True)
```



Basic data plotting

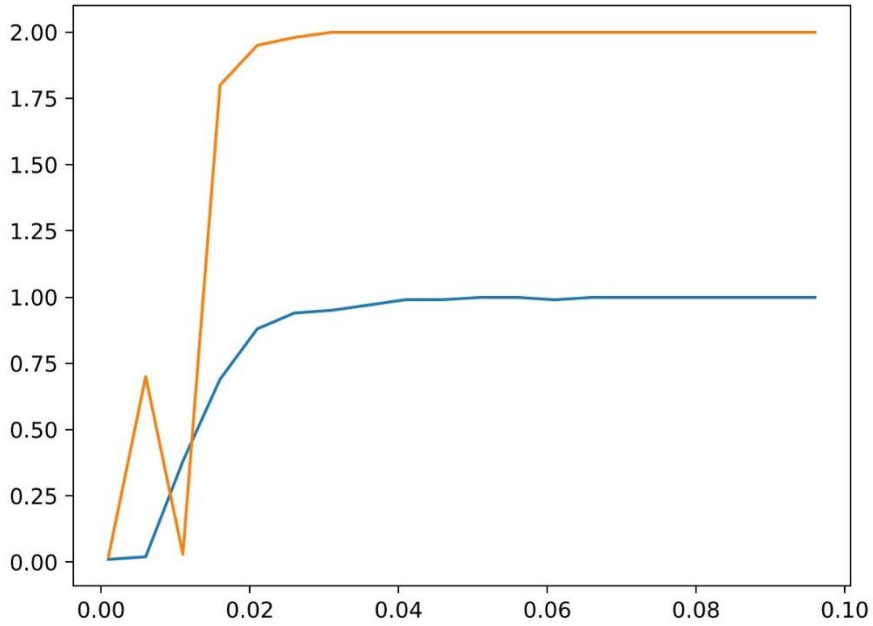
```
def get_phase_curve(ps, n):
    cs = []
    for p in ps:
        g = nx.gnp_random_graph(n, p)
        s = [g.subgraph(c).copy() for c in nx.connected_components(g)]
        c = s[0].order()
        cs.append(float(c) / 100)
    return cs


ps = np.arange(0.001, 0.1, 0.005)
plt.plot(ps, get_phase_curve(ps, 100))
plt.savefig("C:/Users/…/phase.pdf")
```

Phase change plot



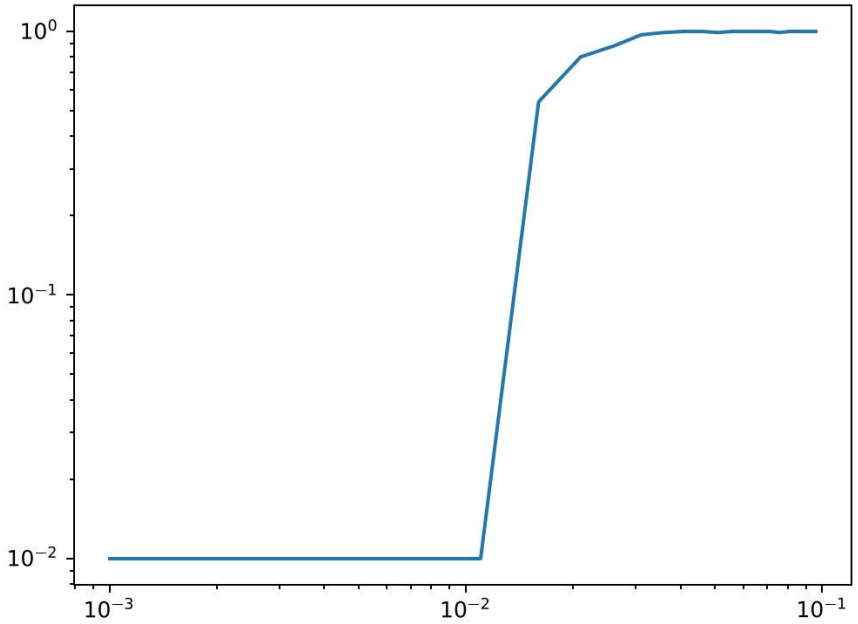
Plotting multiple series

```
plt.clf()
ps = np.arange(0.001, 0.1, 0.005)
plt.plot(ps, get_phase_curve(ps, 100))
plt.plot(ps, get_phase_curve(ps, 200))
plt.savefig("C:/Users/…/phase_100_200.pdf")
```



Log plot

```
ps = np.arange(0.001, 0.1, 0.005)
cs = get_phase_curve(ps, 100)
plt.loglog(ps, cs)  # also  see  semilog
plt.savefig("C:/Users/…/phase_log_log.pdf")
```

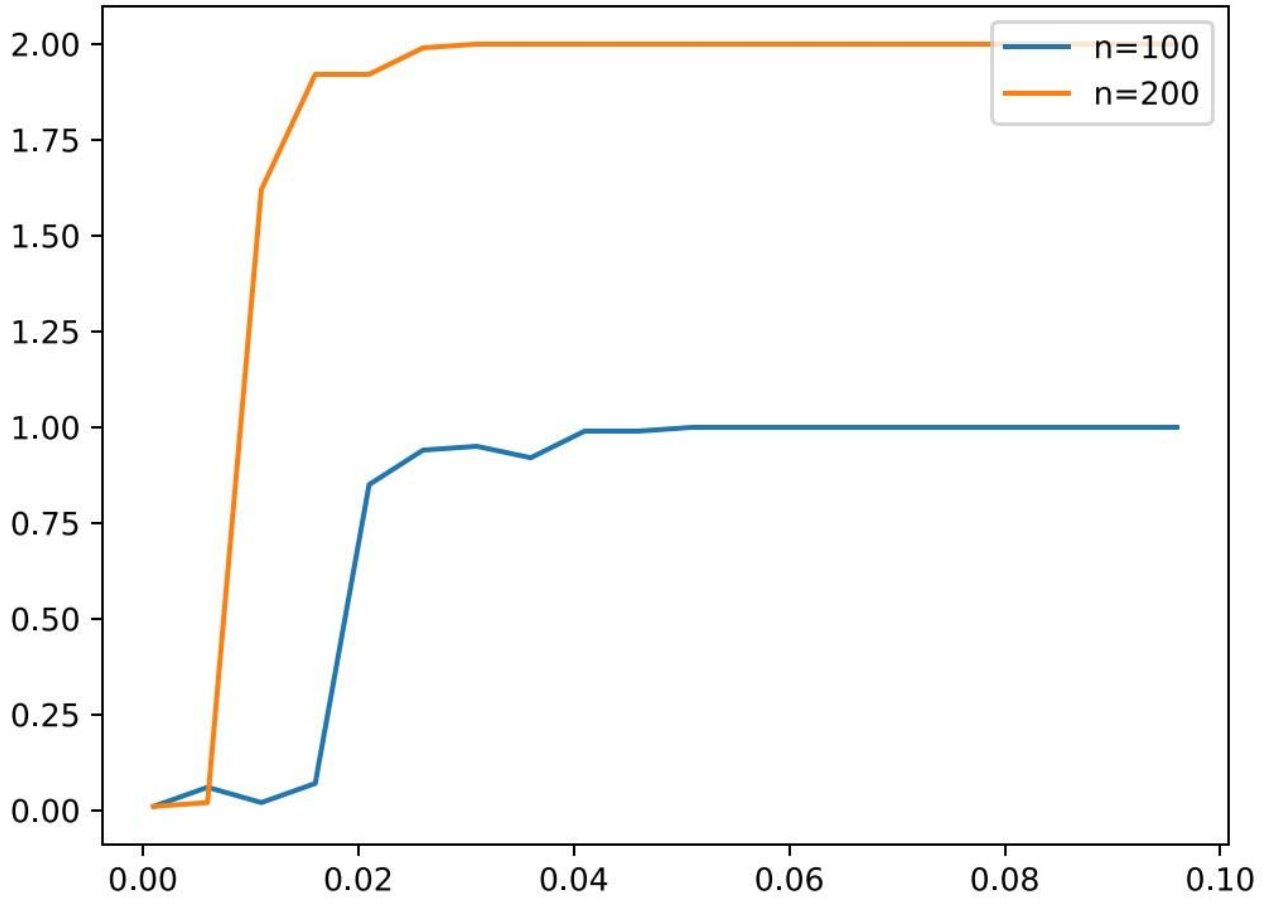


Legends

- Each call to `plt.plot` returns a handle object for the series of type `matplotlib.lines.Line2D`
- To add a legend, use method `plt.legend([handles], [labels])`
- Can control the placement with keyword argument `loc=[1,…,10]` (upper right, lower right, ….)

```
ps = np.arange(0.001, 0.1, 0.005)
h_100 = plt.plot(ps, get_phase_curve(ps, 100))
h_200 = plt.plot(ps, get_phase_curve(ps, 200))
plt.legend(['n=100', 'n=200'], loc=1)
plt.savefig("C:/Users/…/phase_legend.pdf")
```

Legends


n=100
n=200
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.00
0.02
0.04
0.06
0.08
0.10

Resources

- NetworkX Docs, https://networkx.github.io/documentation/stable/
- Matplotlib Docs, https://matplotlib.org/contents.html
- Zinoviev, D.. Complex Network Analysis in Python: Recognize, Construct, Visualize, Analyze, Interpret, The Pragmatic Bookshelf, 2018.
- Rosen, E., NetworkX tutorial, Stanford University, 2011.