

# NetworkX: Network Analysis with Python

---

**Salvatore Scellato**

From a tutorial presented at the 30th SunBelt Conference

*“NetworkX introduction: Hacking social networks using the Python programming language”*

by Aric Hagberg & Drew Conway

# Outline

---

1. Introduction to NetworkX
2. Getting started with Python and NetworkX
3. Basic network analysis
4. Writing your own code
5. You are ready for your own analysis!

# 1. Introduction to NetworkX.

# Introduction to NetworkX - network analysis

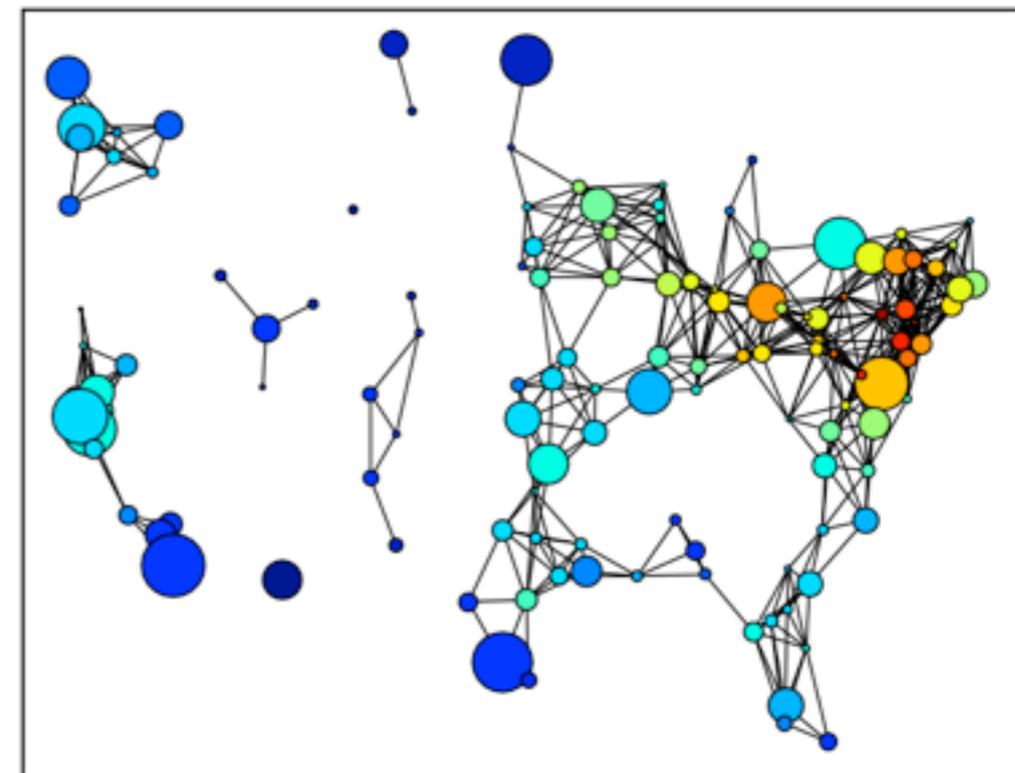
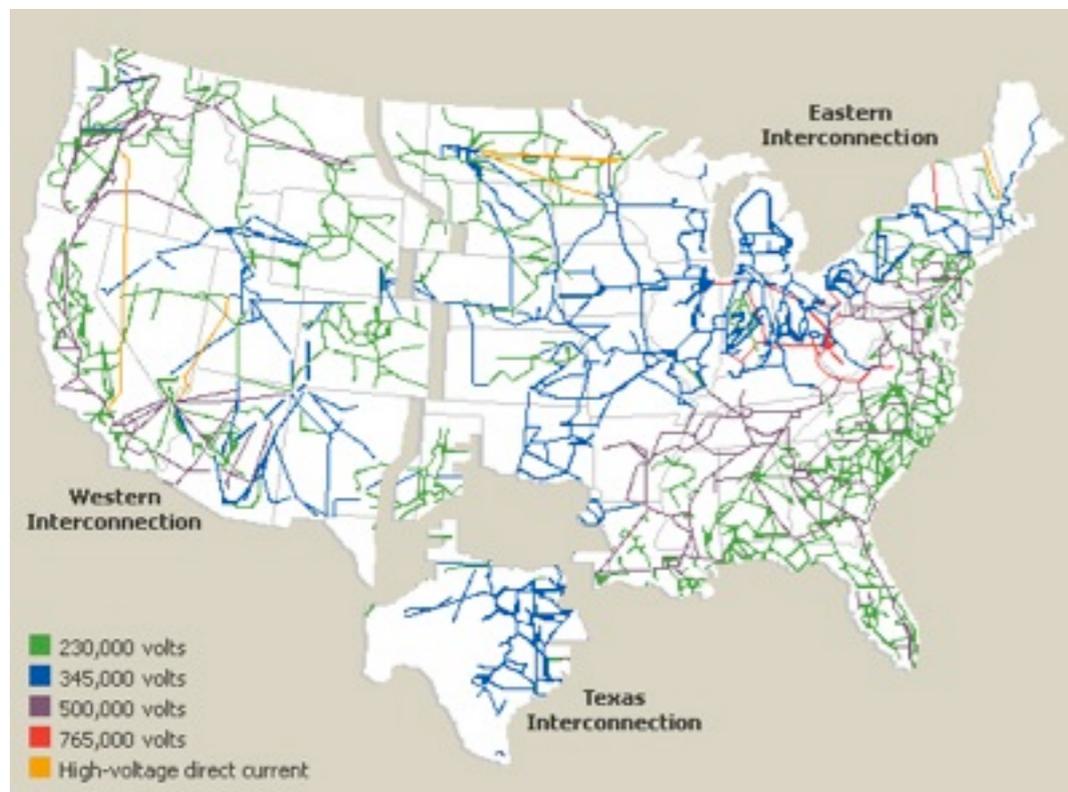
---

Vast amounts of network data are being generated and collected

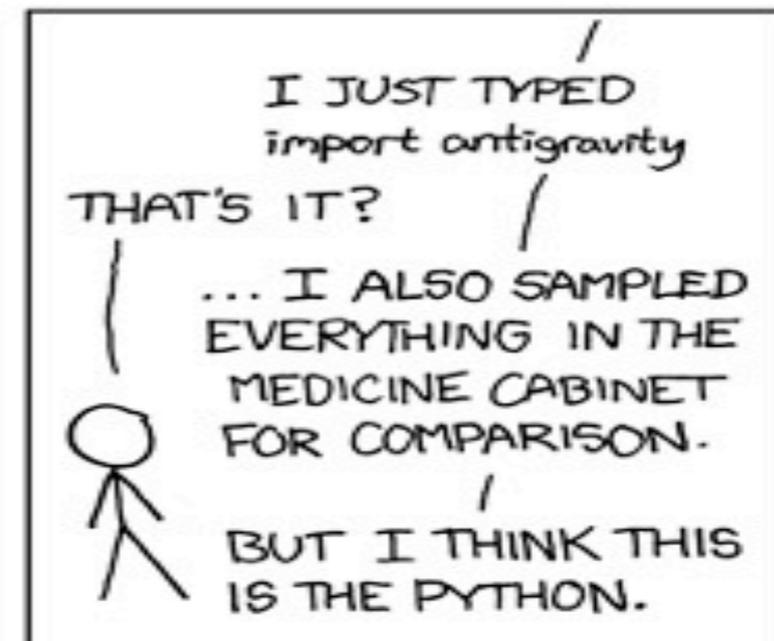
- **Sociology:** web pages, mobile phones, social networks
- **Technology:** Internet routers, vehicular flows, power grids

*How can we analyse these networks?*

*Python + NetworkX!*



# Introduction to NetworkX - Python awesomeness



# Introduction to NetworkX - Python in one slide

---

Python is an interpreted, general-purpose high-level programming language whose design philosophy emphasises code readability.

*“there should be one (and preferably only one) obvious way to do it”.*

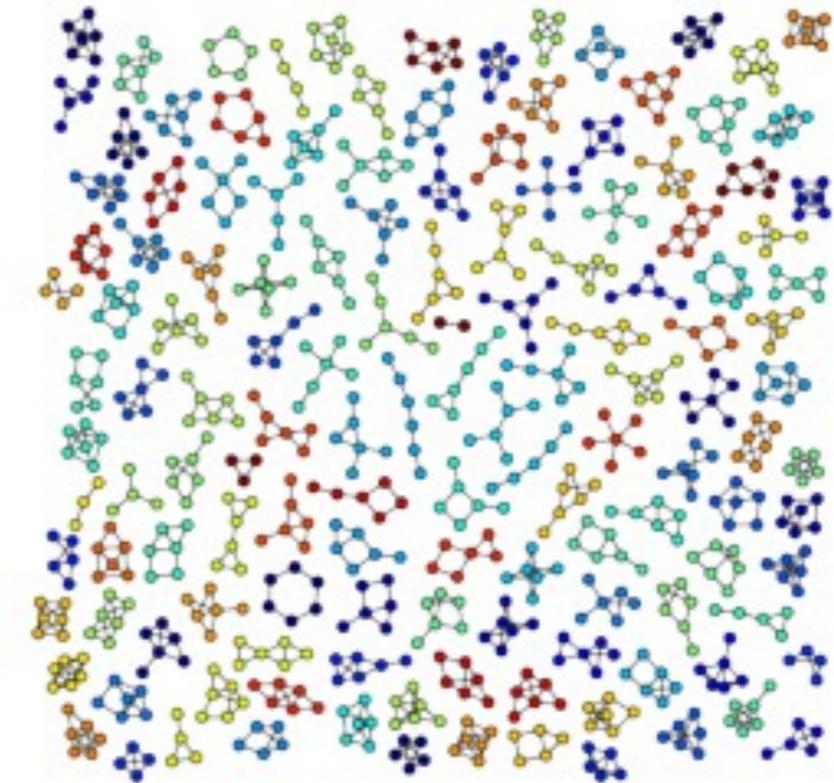
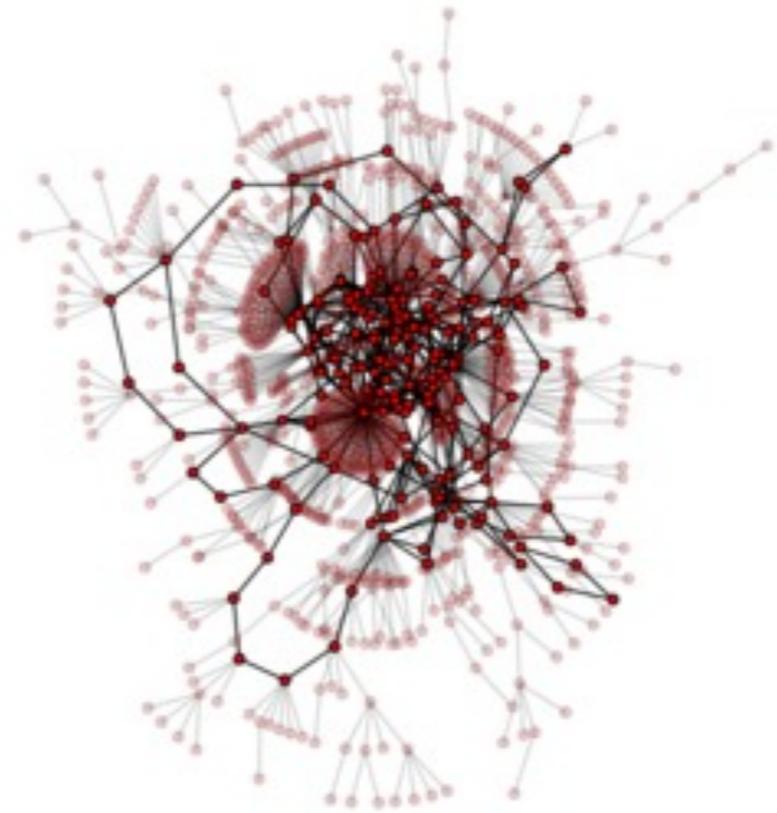
- Use of **indentation** for block delimiters (!!!!)
- **Multiple programming paradigms**: primarily object oriented and imperative but also functional programming style.
- **Dynamic type** system, automatic memory management, **late binding**.
- Primitive **types**: int, float, complex, string, bool.
- Primitive **structures**: list, tuple, dictionary, set.
- Complex **features**: generators, lambda functions, list comprehension, list slicing.

# Introduction to NetworkX

---

*“Python package for the creation, manipulation and study of the structure, dynamics and functions of complex networks.”*

- Data structures for representing many types of networks, or graphs
- Nodes can be any (hashable) Python object, edges can contain arbitrary data
- Flexibility ideal for representing networks found in many different fields
- Easy to install on multiple platforms
- Online up-to-date documentation
- First public release in April 2005



# Introduction to NetworkX - design requirements

---

- **Tool** to study the structure and dynamics of social, biological, and infrastructure networks
- Ease-of-use and **rapid development**
- **Open-source** tool base that can easily grow in a multidisciplinary environment with non-expert users and developers
- An easy **interface to existing code** bases written in C, C++, and FORTRAN
- To painlessly slurp in relatively large **nonstandard data sets**

# Introduction to NetworkX - object model

---

## **NetworkX defines no custom node objects or edge objects**

- node-centric view of network
- nodes can be any hashable object, while edges are tuples with optional edge data (stored in dictionary)
- any Python object is allowed as edge data and it is assigned and stored in a Python dictionary (default empty)

## **NetworkX is all based on Python**

- Instead, other projects use custom compiled code and Python: Boost Graph, igraph, Graphviz
- Focus on computational network modelling not software tool development
- Move fast to design new algorithms or models
- Get immediate results

# Introduction to NetworkX - how to choose

---

## **When should I USE NetworkX to perform network analysis?**

- Unlike many other tools, it is designed to handle data on a scale relevant to modern problems.
- Most of the core algorithms rely on extremely fast legacy code
- Highly flexible graph implementations (a graph/node can be anything!)
- Extensive set of native readable and writable formats
- Takes advantage of Python's ability to pull data from the Internet or databases

## **When should I AVOID NetworkX to perform network analysis?**

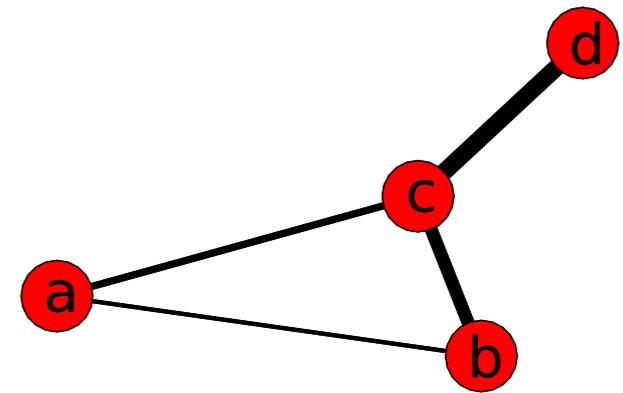
- Large-scale problems that require faster approaches (i.e. massive networks with 100M/1B edges)
- Better use of memory/threads than Python (large objects, parallel computation)

# Introduction to NetworkX - quick example

---

Use Dijkstra's algorithm to find the shortest path in a weighted and unweighted network:

```
>>> import networkx as nx
>>> g = nx.Graph()
>>> g.add_edge('a', 'b', weight=0.1)
>>> g.add_edge('b', 'c', weight=1.5)
>>> g.add_edge('a', 'c', weight=1.0)
>>> g.add_edge('c', 'd', weight=2.2)
>>> print nx.shortest_path(g, 'b', 'd')
['b', 'c', 'd']
>>> print nx.shortest_path(g, 'b', 'd', weighted=True)
['b', 'a', 'c', 'd']
```



# Introduction to NetworkX - Python's Holy Trinity

---



Python's primary library for **mathematical** and **statistical computing**.

Containing sub-libs for

- Numeric optimisation
- Linear algebra
- ..and many others

The primary data type in SciPy is an **array**, so data manipulation is similar to that of MATLAB.



NumPy is an extension of the SciPy data type to include **multidimensional arrays and matrices**.

Provides many functions for working on arrays and matrices.

Both SciPy and NumPy rely on the C library LAPACK for very fast implementation.



matplotlib is **primary plotting library in Python**

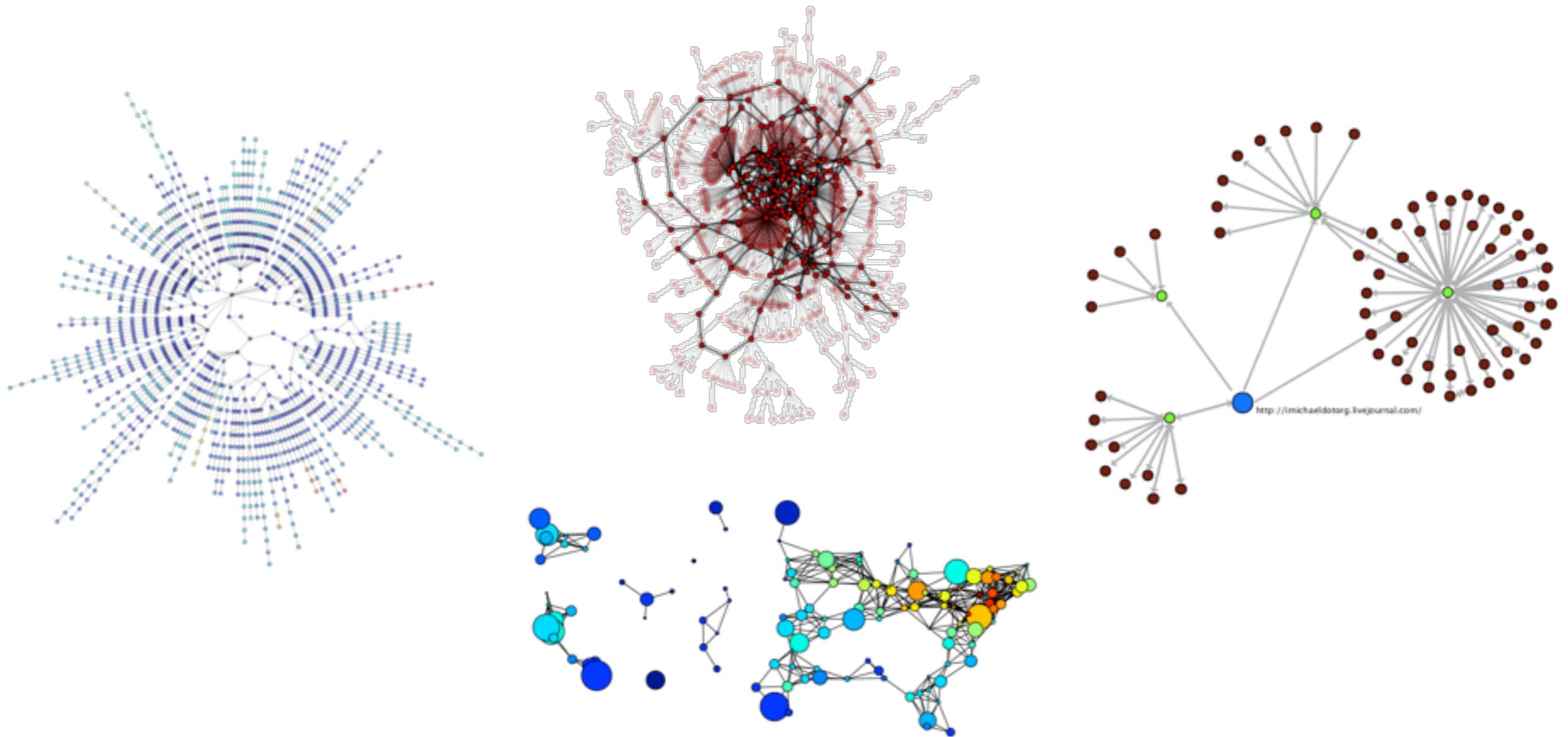
- Supports 2- and 3-D plotting
- API allows embedding in apps

All plots are highly customisable and ready for professional publication.

# Introduction to NetworkX - drawing and plotting

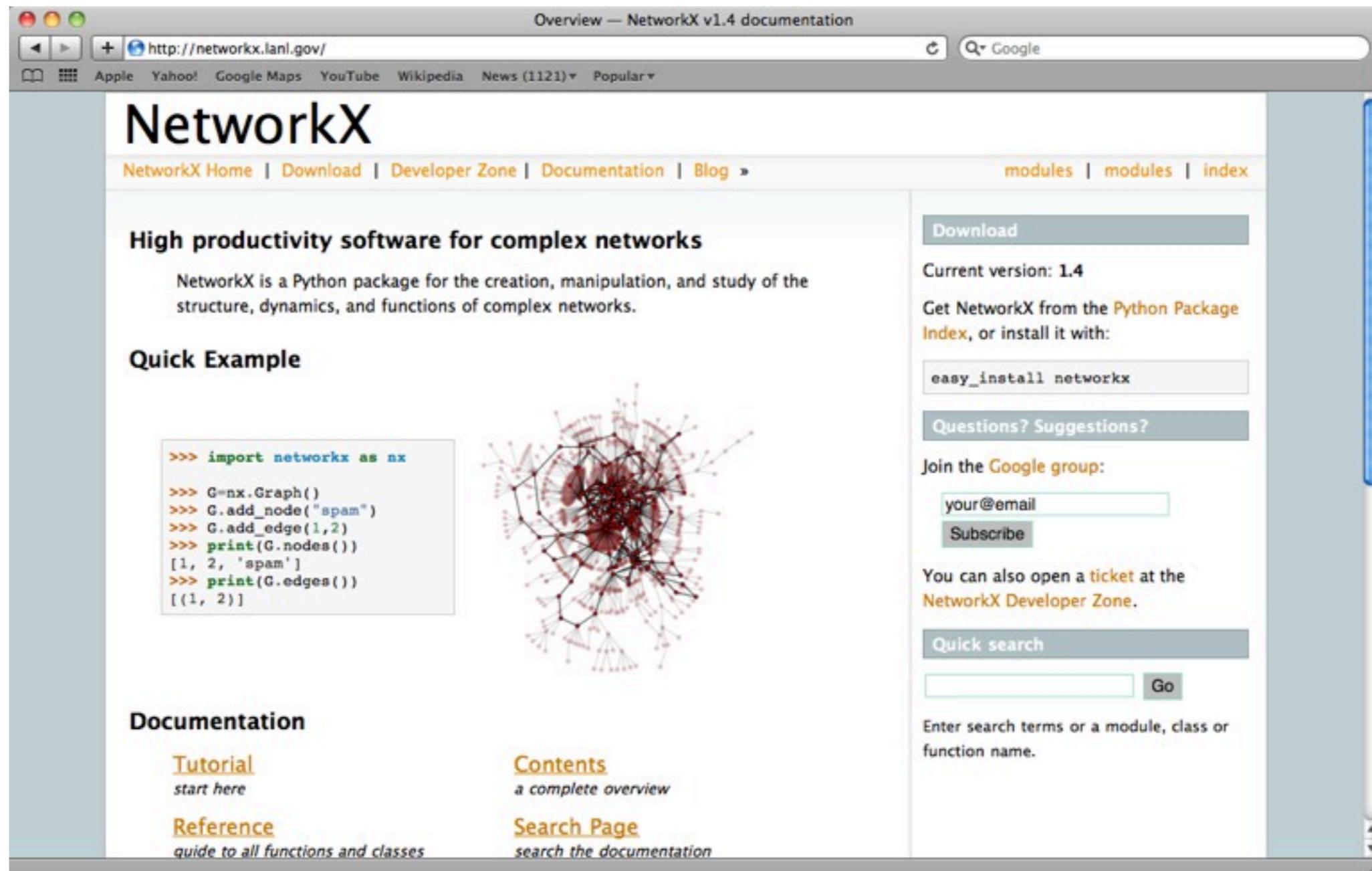
---

- It is possible to draw small graphs within NetworkX and to export network data and draw with other programs (i.e., GraphViz, matplotlib)



# Introduction to NetworkX - official website

<http://networkx.lanl.gov/>

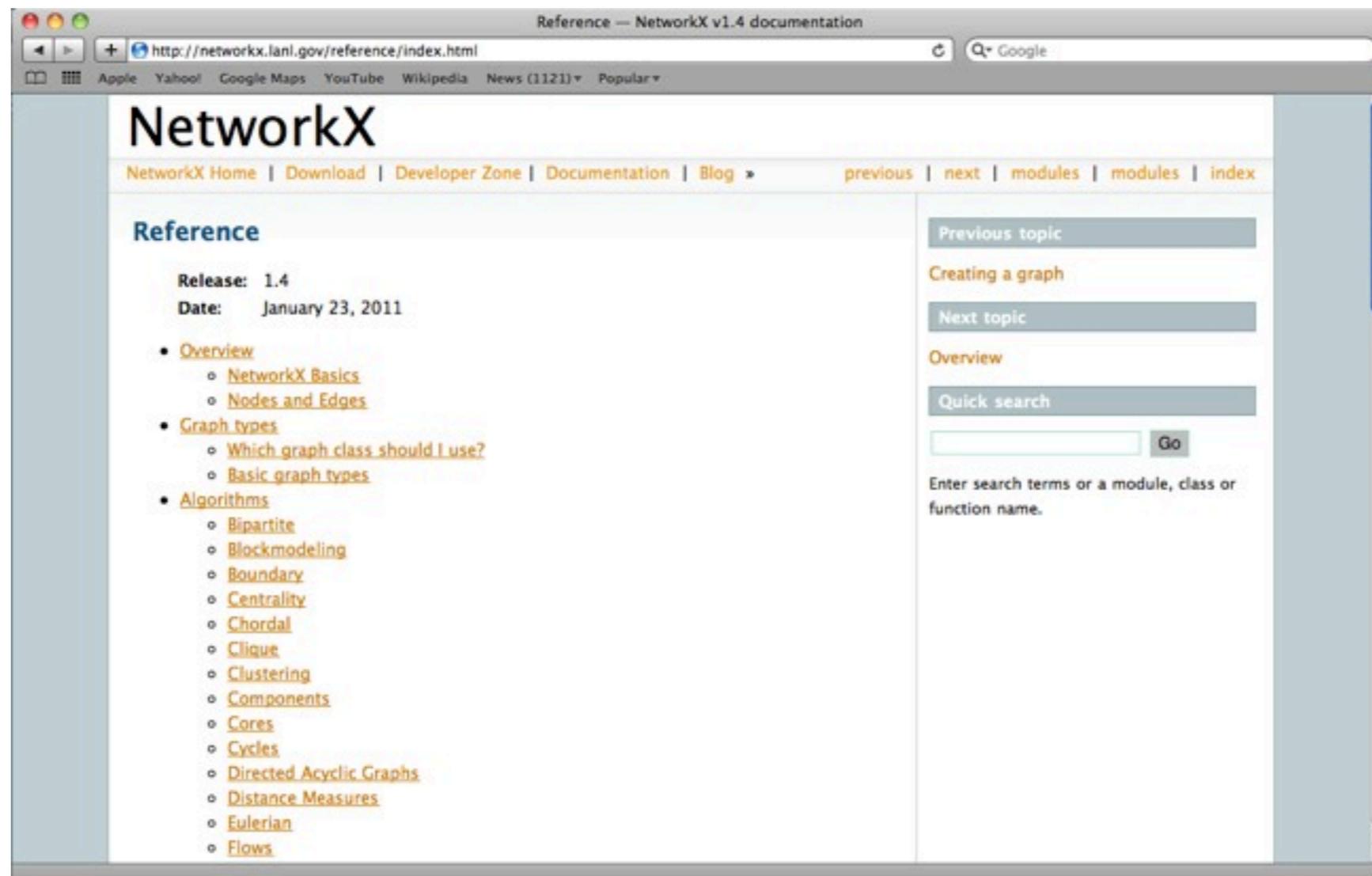


The screenshot shows a web browser window displaying the NetworkX v1.4 documentation page. The browser's address bar shows the URL <http://networkx.lanl.gov/>. The page title is "Overview — NetworkX v1.4 documentation". The main content area features the "NetworkX" logo, a navigation menu with links for "NetworkX Home", "Download", "Developer Zone", "Documentation", and "Blog", and a secondary menu for "modules", "modules", and "index". The main heading is "High productivity software for complex networks", followed by a brief description: "NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks." Below this is a "Quick Example" section with a code block showing a Python script that creates a graph with two nodes and an edge. To the right of the code is a 3D visualization of a complex network graph. The "Documentation" section includes links for "Tutorial" (start here), "Reference" (guide to all functions and classes), "Contents" (a complete overview), and "Search Page" (search the documentation). On the right side of the page, there is a "Download" section with the current version (1.4) and instructions on how to get the package from the Python Package Index. Below that is a "Questions? Suggestions?" section with a "Join the Google group:" link and a form to enter an email address and a "Subscribe" button. At the bottom right, there is a "Quick search" section with a search input field and a "Go" button.

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

# Introduction to NetworkX - online resources

Online **documentation** and active **mailing list** with helpful developers and contributors (<http://networkx.lanl.gov/reference/index.html>)



## 2. Getting started with Python and NetworkX.

# Getting started - import NetworkX

---

Start Python (interactive or script mode) and import NetworkX:

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

There are different Graph classes for undirected and directed networks. Let's create a basic Graph class

```
>>> g = nx.Graph() # empty graph
```

The graph **g** can be grown in several ways. NetworkX includes many graph generator functions and facilities to read and write graphs in many formats.

# Getting started - add nodes

---

*# One node at a time*

```
>>> g.add_node(1) # method of nx.Graph
```

*# A list of nodes*

```
>>> g.add_nodes_from([2, 3])
```

*# A container of nodes*

```
>>> h = nx.path_graph(10)
```

```
>>> g.add_nodes_from(h) # g now contains the nodes of h
```

*# In contrast, you can remove any node of the graph*

```
>>> g.remove_node(2)
```

# Getting started - node entities

---

A node can be **any hashable object** such as strings, numbers, files, functions, and more. This provides important flexibility to all your projects.

```
>>> import math
>>> g.add_node(math.cos) # cosine function
>>> fh=open('tmp.txt','w') # file handle
>>> g.add_node(fh)
>>> print g.nodes()
[<built-in function cos>, <open file 'tmp.txt', mode 'w'
at 0x30dc38>]
```

# Getting started - add edges

---

```
# Single edge
```

```
>>> g.add_edge(1,2)
```

```
>>> e=(2,3)
```

```
>>> g.add_edge(*e) # unpack edge tuple
```

```
# List of edges
```

```
>>> g.add_edges_from([(1,2), (1,3)])
```

```
# Container of edges
```

```
>>> g.add_edges_from(h.edges())
```

```
# In contrast, you can remove any edge of the graph
```

```
>>> g.remove_edge(1,2)
```

# Getting started - access nodes and edges

---

```
>>> g.add_edges_from([(1,2) ,(1,3)])
>>> g.add_node('a')
>>> g.number_of_nodes() # also g.order()
4
>>> g.number_of_edges() # also g.size()
2
>>> g.nodes()
[1, 2, 3, 'a']
>>> g.edges()
[(1, 2), (1, 3)]
>>> g.neighbors(1)
[2, 3]
>>> g.degree(1)
2
```

# Getting started - Python dictionaries

---

NetworkX takes advantage of **Python dictionaries** to store node and edge measures. The `dict` type is a data structure that represents a key-value mapping.

```
# Keys and values can be of any data type
```

```
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],42:True}
```

```
# Can retrieve the keys and values as Python lists (vector)
```

```
>>> fruit_dict.keys()
```

```
[ "orange" , "apple" , 42 ]
```

```
# Or create a (key,value) tuple
```

```
>>> fruit_dict.items()
```

```
[ ("orange", [0.23,0.11]), ("apple", 1), (42, True) ]
```

```
# This becomes especially useful when you master Python list-comprehension
```

# Getting started - access nodes and edges

---

Any NetworkX graph behaves like a Python dictionary with nodes as primary keys (**only for access!**)

```
>>> g.add_node(1, time='5pm')
>>> g.node[1]['time']
'5pm'
>>> g.node[1] # Python dictionary
{'time': '5pm'}
```

The special edge attribute '**weight**' should always be numeric and holds values used by algorithms requiring weighted edges.

```
>>> g.add_edge(1, 2, weight=4.0 )
>>> g[1][2]['weight'] = 5.0 # edge already added
>>> g[1][2]
{'weight': 5.0}
```

# Getting started - node and edge iterators

---

Many applications require iteration over nodes or over edges: simple and easy in NetworkX

```
>>> g.add_edge(1,2)
>>> for node in g.nodes():
    print node, g.degree(node)
1, 1
2, 1

>>> g.add_edge(1,3,weight=2.5)
>>> g.add_edge(1,2,weight=1.5)
>>> for n1,n2,attr in g.edges(data=True): # unpacking
    print n1,n2,attr['weight']
1, 2, 1.5
1, 3, 2.5
```

# Getting started - directed graphs

---

```
>>> dg = nx.DiGraph()
>>> dg.add_weighted_edges_from([(1,4,0.5), (3,1,0.75)])
>>> dg.out_degree(1,weighted=True)
0.5
>>> dg.degree(1,weighted=True)
1.25
>>> dg.successors(1)
[4]
>>> dg.predecessors(1)
[3]
```

Some algorithms work only for undirected graphs and others are not well defined for directed graphs. If you want to treat a directed graph as undirected for some measurement you should probably convert it using `Graph.to_undirected()`

# Getting started - multigraphs

---

NetworkX provides classes for graphs which allow **multiple edges** between any pair of nodes, `MultiGraph` and `MultiDiGraph`.

This can be powerful for some applications, but many algorithms are not well defined on such graphs: shortest path is one example.

Where results are not well defined you should convert to a standard graph in a way that makes the measurement well defined.

```
>>> mg = nx.MultiGraph()
>>> mg.add_weighted_edges_from([(1,2,.5), (1,2,.75),
(2,3,.5)])
>>> mg.degree(weighted=True)
{1: 1.25, 2: 1.75, 3: 0.5}
```

# Getting started - graph operators

---

Classic graph operations

`subgraph (G, nbunch)` - induce subgraph of G on nodes in **nbunch**

`union (G1, G2)` - graph union

`disjoint_union (G1, G2)` - graph union assuming all nodes are different

`cartesian_product (G1, G2)` - return Cartesian product graph

`compose (G1, G2)` - combine graphs identifying nodes common to both

`complement (G)` - graph complement

`create_empty_copy (G)` - return an empty copy of the same graph class

`convert_to_undirected (G)` - return an undirected representation of G

`convert_to_directed (G)` - return a directed representation of G

# Getting started - graph generators

---

```
# small famous graphs
```

```
>>> petersen=nx.petersen_graph()
```

```
>>> tutte=nx.tutte_graph()
```

```
>>> maze=nx.sedgewick_maze_graph()
```

```
>>> tet=nx.tetrahedral_graph()
```

```
# classic graphs
```

```
>>> K_5=nx.complete_graph(5)
```

```
>>> K_3_5=nx.complete_bipartite_graph(3,5)
```

```
>>> barbell=nx.barbell_graph(10,10)
```

```
>>> lollipop=nx.lollipop_graph(10,20)
```

```
# random graphs
```

```
>>> er=nx.erdos_renyi_graph(100,0.15)
```

```
>>> ws=nx.watts_strogatz_graph(30,3,0.1)
```

```
>>> ba=nx.barabasi_albert_graph(100,5)
```

```
>>> red=nx.random_lobster(100,0.9,0.9)
```

# Getting started - graph I/O

---

NetworkX is able to read/write graphs from/to files using common graph formats:

- edge lists
- adjacency lists
- GML
- GEXF
- Python pickle
- GraphML
- Pajek
- LEDA
- YAML

We will see how to read/write **edge lists**.

# Getting started - read and write edge lists

---

## General read/write format

```
>>> g = nx.read_format("path/to/file.txt", ...options...)
>>> nx.write_format(g, "path/to/file.txt", ...options...)
```

## Read and write edge lists

```
g = nx.read_edgelist(path, comments='#', create_using=None,
delimiter=' ', nodetype=None, data=True, edgetype=None, encoding='utf-8')
nx.write_edgelist(g, path, comments='#',
delimiter=' ', data=True, encoding='utf-8')
```

## Formats

- Node pairs with no data:

```
1 2
```

- Python dictionary as data:

```
1 2 {'weight':7, 'color':'green'}
```

- Arbitrary data:

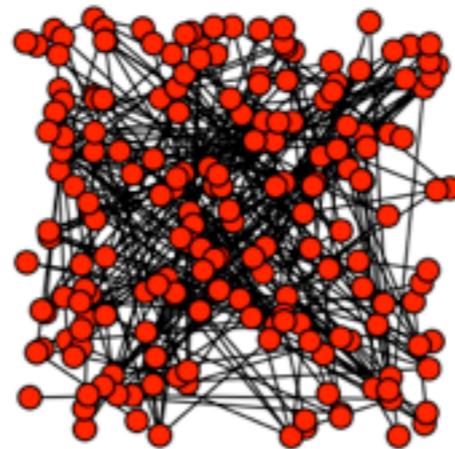
```
1 2 7 green
```

# Getting started - draw a graph

---

NetworkX is not primarily a graph drawing package but it provides basic drawing capabilities by using **matplotlib**. For more complex visualization techniques it provides an interface to use the open source **GraphViz** software package.

```
>>> import pylab as plt #import Matplotlib plotting interface
>>> g = nx.erdos_renyi_graph(100, 0.15)
>>> nx.draw(g)
>>> nx.draw_random(g)
>>> nx.draw_circular(g)
>>> nx.draw_spectral(g)
>>> plt.savefig('graph.png')
```



Note that the drawing package in NetworkX is not (yet!) compatible with Python versions 3.0 and above.

## 3. Basic network analysis.

# Basic network analysis - graph properties

---

Let's load the Hartford drug users network: it's a directed graph with integers as nodes.

```
hartford = nx.read_edgelist('hartford.txt',  
                           create_using=nx.DiGraph(), nodetype=int)
```

```
N, K = hartford.order(), hartford.size()  
avg_deg = float(K)/N  
print "Nodes: ", N  
print "Edges: ", K  
print "Average degree: ", avg_deg
```

# Basic network analysis - degree distribution

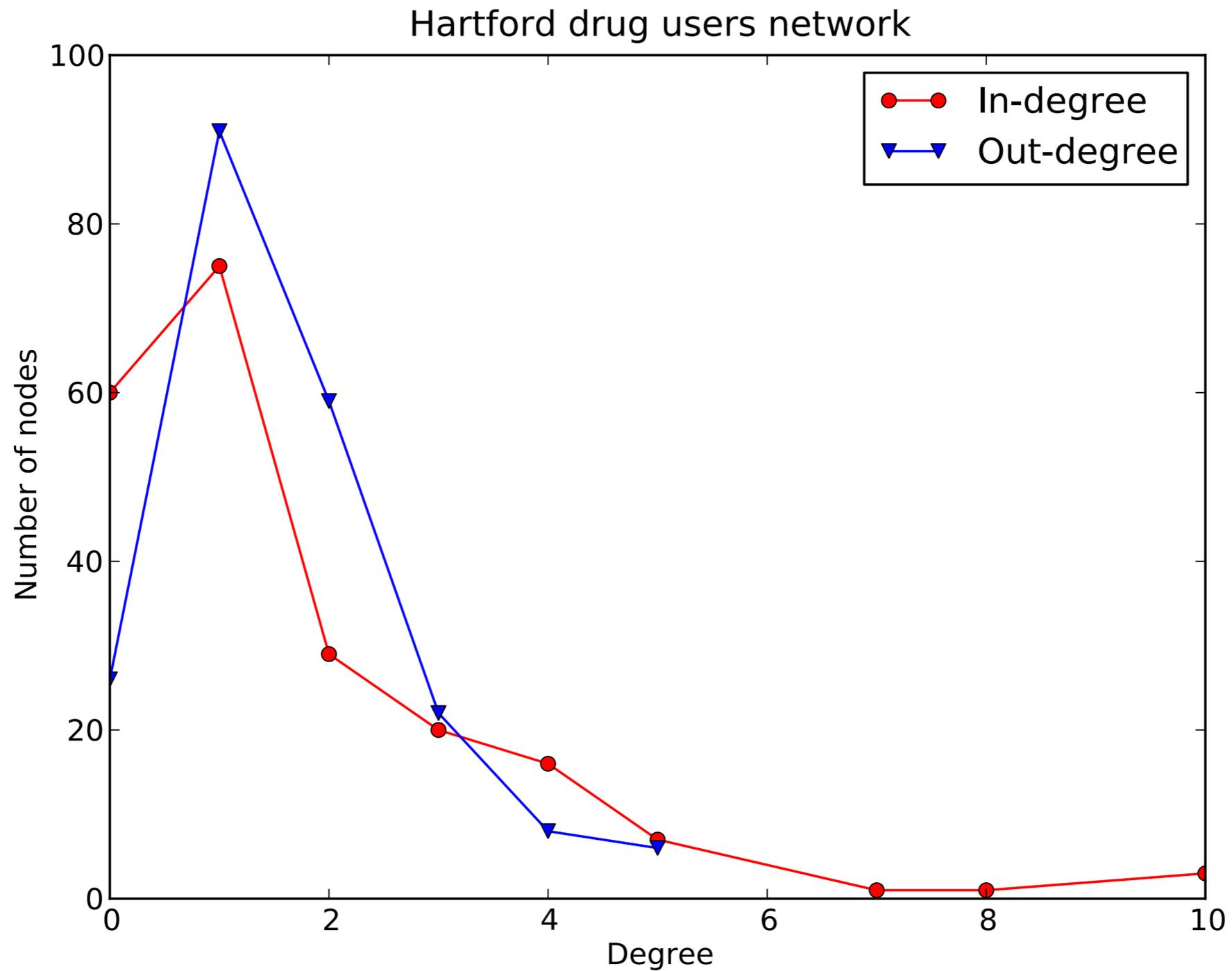
---

Let's compute in- and out-degree distribution of the graph and plot them. Don't try this method with massive graphs, it's slow...!

```
in_degrees = hartford.in_degree() # dictionary node:degree
in_values = sorted(set(in_degrees.values()))
in_hist = [in_degrees.values().count(x) for x in in_values]

plt.figure()
plt.plot(in_values, in_hist, 'ro-') # in-degree
plt.plot(out_values, out_hist, 'bv-') # out-degree
plt.legend(['In-degree', 'Out-degree'])
plt.xlabel('Degree')
plt.ylabel('Number of nodes')
plt.title('Hartford drug users network')
plt.savefig('hartford_degree_distribution.pdf')
plt.close()
```

# Basic network analysis - degree distribution



# Basic network analysis - clustering coefficient

---

We can get the clustering coefficient of individual nodes or of all the nodes (but the first we convert the graph to an undirected one):

```
hartford_ud = hartford.to_undirected()

# Clustering coefficient of node 0
print nx.clustering(hartford_ud, 0)

# Clustering coefficient of all nodes (in a dictionary)
clust_coefficients = nx.clustering(hartford_ud)

# Average clustering coefficient
ccs = nx.clustering(hartford_ud)
avg_clust = sum(ccs.values()) / len(ccs)
```

# Basic network analysis - node centralities

---

Now, we will extract the main connected component; then we will compute **node centrality measures**.

```
hartford_components =  
    nx.connected_component_subgraphs(hartford_ud)  
hartford_mc = hartford_components[0]  
  
# Betweenness centrality  
bet_cen = nx.betweenness_centrality(hartford_mc)  
# Closeness centrality  
clo_cen = nx.closeness_centrality(hartford_mc)  
# Eigenvector centrality  
eig_cen = nx.eigenvector_centrality(hartford_mc)
```

# Basic network analysis - most central nodes

---

To find the most central nodes we will learn Python's list comprehension technique to do basic data manipulation on our centrality dictionaries.

```
def highest_centrality(cent_dict):  
    """Returns a tuple (node,value) with the node  
with largest value from Networkx centrality dictionary."""  
    # Create ordered tuple of centrality data  
    cent_items=[(b,a) for (a,b) in cent_dict.iteritems()]  
  
    # Sort in descending order  
    cent_items.sort()  
    cent_items.reverse()  
  
    return tuple(reversed(cent_items[0]))
```



# Basic network analysis - plotting results

---

```
def centrality_scatter(dict1,dict2,path="",
                      ylab="",xlab="",title="",line=False):
    # Create figure and drawing axis
    fig = plt.figure(figsize=(7,7))
    ax1 = fig.add_subplot(111)

    # Create items and extract centralities
    items1 = sorted(dict1.items())
    items2 = sorted(dict2.items())
    xdata=[b for a,b in items1]
    ydata=[b for a,b in items2]

    # Add each actor to the plot by ID
    for p in xrange(len(items1)):
        ax1.text(x=xdata[p], y=ydata[p],s=str(items1[p][0]), color="b")
```

# Basic network analysis - plotting results

---

...continuing....

```
if line:
    # use NumPy to calculate the best fit
    slope, yint = plt.polyfit(xdata,ydata,1)
    xline = plt.xticks()[0]
    yline = map(lambda x: slope*x+yint,xline)
    ax1.plot(xline,yline,ls='--',color='b')

# Set new x- and y-axis limits
plt.xlim((0.0,max(xdata)+(.15*max(xdata))))
plt.ylim((0.0,max(ydata)+(.15*max(ydata))))
# Add labels and save
ax1.set_title(title)
ax1.set_xlabel(xlab)
ax1.set_ylabel(ylab)
plt.savefig(path)
```

# Basic network analysis - export results

---

Even though NetworkX and the complementing scientific computing packages in Python are powerful, it may often be useful or necessary to **output** your data for additional analysis because:

- suite of tools lacks your specific need
- you require alternative visualisation
- you want to store results for later analysis

In most cases this will entail either **exporting** the raw network data, or metrics from some network analysis

1. NetworkX can write out network data in as many formats as it can read them, and the process is equally straightforward
2. When you want to export metrics we can also use Python's built-in XML and CSV libraries, or simply write to a text file.

# Basic network analysis - write results to file

---

Let's export a CSV file with node IDs and the related centrality values on each line: this can be then used to plot without computing again all centrality measures.

```
results = [(k,bet_cen[k],clo_cen[k],eig_cen[k])
            for k in hartford_mc]
f = open('hartford_results.txt','w')
for item in results:
    f.write(','.join(map(str,item)))
    f.write('\n')
f.close()
```

## 4. Writing your own code.

# Write your own code - BFS

---

With Python and NetworkX it's easy to write any graph-based algorithm

```
from collections import deque

def breadth_first_search(g, source):
    queue = deque([(None, source)])
    enqueued = set([source])
    while queue:
        parent, n = queue.popleft()
        yield parent, n
        new = set(g[n]) - enqueued
        enqueued |= new
        queue.extend([(n, child) for child in new])
```

# Write your own code - network triads

---

Extract all unique triangles in a graph with integer node IDs

```
def get_triangles(g):
    for n1 in g.nodes:
        neighbors1 = set(g[n1])
        for n2 in filter(lambda x: x>n1, nodes):
            neighbors2 = set(g[n2])
            common = neighbors1 & neighbors2
            for n3 in filter(lambda x: x>n2, common):
                yield n1,n2,n3
```

# Write your own code - average neighbours' degree

---

Compute the average degree of each node's neighbours (long and one-liner version).

```
def avg_neigh_degree(g):
    data = {}
    for n in g.nodes():
        if g.degree(n):
            data[n] = float(sum(g.degree(i) for i in g[n])) /
g.degree(n)
    return data
```

```
def avg_neigh_degree(g):
    return dict((n, float(sum(g.degree(i) for i in g[n])) /
g.degree(n)) for n in g.nodes() if g.degree(n))
```

5. You are ready for your own analysis!

# What you have learnt today about NetworkX

---

- How to **create graphs** from scratch, with generators and by loading local data
- How to compute **basic network measures**, how they are stored in NetworkX and how to manipulate them with list comprehension
- Getting **data out** of NetworkX as raw network data or analytics
- How to use matplotlib to **visualize and plot** results (useful for final report!)
- How to use and include NetworkX features to **design your own algorithms/analysis**

# Useful links

---

- **Code&data used in this lecture:** <http://www.cl.cam.ac.uk/~ss824/stna-examples.tar.gz>
- **NodeXL:** a graphical front-end that integrates network analysis into Microsoft Office and Excel. (<http://nodexl.codeplex.com/>)
- **Pajek:** a program for network analysis for Windows (<http://pajek.imfm.si/doku.php>).
- **Gephi:** an interactive visualization and exploration platform (<http://gephi.org/>)
- **Power-law Distributions in Empirical Data:** tools for fitting heavy-tailed distributions to data (<http://www.santafe.edu/~aaronc/powerlaws/>)
- **GraphViz:** graph visualization software (<http://www.graphviz.org/>)
- **Matplotlib:** full documentation for the plotting library (<http://matplotlib.sourceforge.net/>)

# Questions?

---

**Salvatore Scellato**

**Email:** [salvatore.scellato@cl.cam.ac.uk](mailto:salvatore.scellato@cl.cam.ac.uk)

**Web:** [www.cl.cam.ac.uk/~ss824/](http://www.cl.cam.ac.uk/~ss824/)

