# 3D graphs with NetworkX, VTK, and ParaView

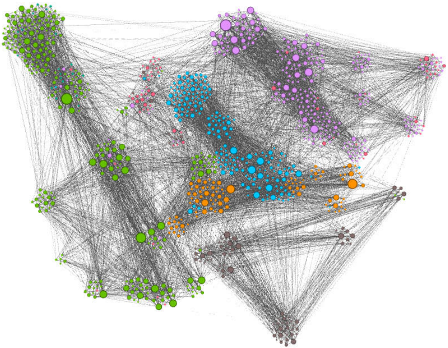
Alex Razoumov
alex.razoumov@westgrid.ca

WestGrid / Compute Canada

copy of these slides and other files at `http://bit.ly/3dgraphs`
- will download 3dgraphs.zip

# 2D graphs

- Many tools, most popular ones are Gephi, Cytoscape (both open source)
- You can find a copy of the Gephi webinar notes (March 2016) at `http://bit.ly/gephibits`



How can we extend this to 3D? And do we really want to?

# 3D graphs

- Force Atlas 3D plugin for Gephi `http://bit.ly/1QcLuLK` gives a 2D projection with nodes as spheres at (x,y,z) and the proper perspective and lighting, but can't interact with the graph in 3D
- Functional brain network visualization tools, e.g., Connectome Viewer `http://cmtk.org/viewer`
- GraphInsight was a fantastic tool, free academic license, embedded Python shell – went to the dark side in the fall 2013 (purchased by a bank, no longer exists, can still find demo versions and youtube videos)
- Walrus `http://www.caida.org/tools/visualization/walrus` was a research project, latest update in 2005, old source still available but people seem to have trouble compiling and running it now
- Network3D from Microsoft seems to be a short-lived research project, Windows only
- BioLayout Express 3D `http://www.biolayout.org/download` is Ok, written in Java, development stopped in 2014 but still works, only the commercial tool maintained ( $500)
- ORA NetScenes from Carnegie Mellon for "networked text visualization", not bad, Windows only, not open-source, licensing not clear (more of a demo license, they reserve the right to make it paid)
- Number of other research projects not targeting end users, e.g., `http://www.opengraphiti.com` (pain to compile: tends to pick `/usr/bin/python`, only `Mac/Linux`), or WebGL projects `https://youtu.be/qHkjSxbnzAU` that really require programming knowledge
  - `https://markwolff.shinyapps.io/QMtriplot17C/` is a nice WebGL example in R + Shiny

Is there any good, open-source, cross-platform, currently maintained, user-friendly dedicated 3D graph visualization tool?

# What are the options?

- Code your own graph visualization in JavaScript and WebGL
- NetworkX + MayaVi `http://bit.ly/1MyvIA8`; MayaVi's terminal tends to slow down after complex visualizations (bug both in Windows and Mac implementations)
- Colleague of mine suggested using a chemistry tool Jmol to visualize graphs (e.g., `http://www.vesnam.com/Rblog/viznets4` creates graphs with R and displays them with Jmol); "would require some customisations to trigger the selections of adjacent nodes when clicking on one"; Jmol works as a Java Application on the desktop and as Java applet and JavaScript in the browser

# NetworkX + VTK + ParaView

- Our solution: NetworkX + VTK + ParaView
  - advantage: (1) using general-purpose visualization tool; (2) everything is scriptable; (3) can scale directly to $10^{\sim 5.5}$ nodes, with a little extra care to $10^{\sim 7.5}$ nodes, and with some thought to $10^{\sim 9.5}$ nodes
  - disadvantages: graphs are static 3D objects, can't click on a node, highlight connections, move nodes, etc. (but we can script these interactions!)
  - note: in the current implementation edges are displayed as straight lines; possible to use vtkArcSource or vtkPolyLine to create arcs and store them as vtkPolyData

(1) We'll use NetworkX + VTK to create a graph, position nodes, optionally compute graph statistics, and write everything to a VTK file; we'll do this in Python 2.7 (VTK for Python 3 is not quite ready)

(2) Load that file into ParaView

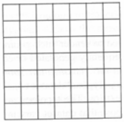
- ParaView comes with its own Python shell and VTK, but it is somewhat tricky to install NetworkX there

# What is VTK?

- 3D **Visualization Toolkit** software system for 3D computer graphics, image processing, and visualization
- Open-source and cross-platform (Windows, Mac, Linux, other Unix variants)
- Suppors OpenGL hardware acceleration
- C++ class library, with interpreted interface layers for Python, Java, Tcl/Tk
- Supports wide variety of visualization and processing algorithms for polygon rendering, ray tracing, mesh smoothing, cutting, contouring, Delaunay triangulation, etc.
- Supports many data types: scalar, vector, tensor, texture, arrays of arrays
- Supports many 2D/3D spatial discretizations: structured and unstructured meshes, particles, polygons, etc. – see next slide
- Includes a suite of 3D interaction widgets, integrates nicely with several popular cross-platform GUI toolkits (Qt, Tk)
- Supports parallel processing and parallel I/O
- Base layer of several really good 3D visualization packages (ParaView, VisIt, MayaVi, and several others)

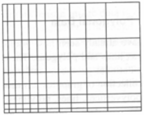
# VTK 2D/3D data: 6 major discretizations (mesh types)

- **Image Data/Structured Points**: *.vti, points on a regular rectangular lattice, scalars or vectors at each point



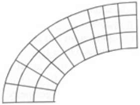
(a) Image Data

- **Rectilinear Grid**: *.vtr, same as Image Data, but spacing between points may vary, need to provide steps along the coordinate axes, not coordinates of each point



(b) Rectilinear Grid

- **Structured Grid**: *.vts, regular topology and irregular geometry, need to indicate coordinates of each point



(c) Structured Grid

# VTK 2D/3D data: 6 major discretizations (mesh types)

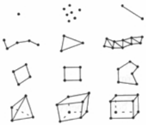
- **Particles/Unstructured Points**: *.particles

(d) Unstructured Points

- **Polygonal Data**: *.vtp, unstructured topology and geometry, point coordinates, 2D cells only (i.e. no polyhedra), suited for maps

(e) Polygonal Data

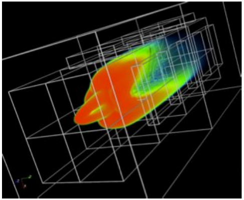
- **Unstructured Grid**: *.vtu, irregular in both topology and geometry, point coordinates, 2D/3D cells, suited for finite element analysis, structural design

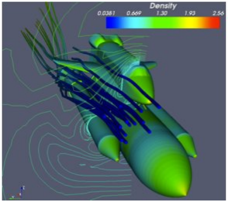


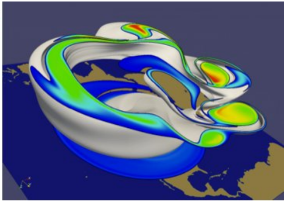
(f) Unstructured Grid

# ParaView as GUI frontend to VTK classes

- 3D visualization tool for extremely large datasets
- Scales from laptops to supercomputers with $10^{5.5}$ cores
- Open source, binary downloads for Linux/Mac/Windows from `http://www.paraview.org`
- Interactive GUI and Python scripting
- Client-server architecture
- Uses MPI for distributed-memory parallelism on HPC clusters
- Based on VTK (developed by the same folks), fully supports all VTK classes and data types
- Huge array of visualization features





# Installation

- For your OS install ParaView from http://www.paraview.org/download
- For your OS install Python 2.7 Miniconda distribution from `http://conda.pydata.org/miniconda.html`
  - in Miniconda, as of this writing, VTK not yet available in Python 3.5
- Start the command shell (terminal in MacOS/Linux, DOS prompt in Windows) and then install two Python packages:

```
conda install vtk
conda install networkx
```

- Start Python and test your Miniconda installation:

```
import vtk
import networkx as nx
```

# NetworkX graphs

- NetworkX is a Python package for the creation, manipulation, and analysis of complex networks
- Documentation at `http://networkx.github.io`

```
import networkx as nx

# return all names (attributes and methods) inside nx
dir(nx)

# generate a list (of 105) built-in graph types
# with Python's ``list comprehension''
[x for x in dir(nx) if '_graph' in x]
```

# NetworkX layouts

```
# generate a list built-in graph layouts
[x for x in dir(nx) if '_layout' in x]
# will print ['circular_layout',
# 'fruchterman_reingold_layout', 'random_layout',
# 'shell_layout', 'spectral_layout', 'spring_layout']

# can always look at the help pages
help(nx.circular_layout)
```

- `spring_` and `fruchterman_reingold_` are the same, so really 5 built-in layouts
- can use 3rd-party layouts (you'll see at least one later in this presentation)
- `circular_`, `random_`, `shell_` are fixed layouts
- `spring_` and `spectral_` are force-directed layouts: linked nodes attract each other, non-linked nodes are pushed apart

# NetworkX layouts

- Layouts typically return a *dictionary*, with each element being a 2D/3D coordinate array indexed by the node's number (or name)

```
# generate a random graph
H = nx.gnm_random_graph(10,50)

# the first element of the dictionary is a 2D array
# (currently only dim=2 is supported)
nx.shell_layout(H,dim=3)[0]
nx.circular_layout(H,dim=3)[0]

# the first element of the dictionary is a 3D array
nx.spring_layout(H,dim=3)[0]
nx.random_layout(H,dim=3)[0]
nx.spectral_layout(H,dim=3)[0]
```

# Custom Python function to write graphs as VTK

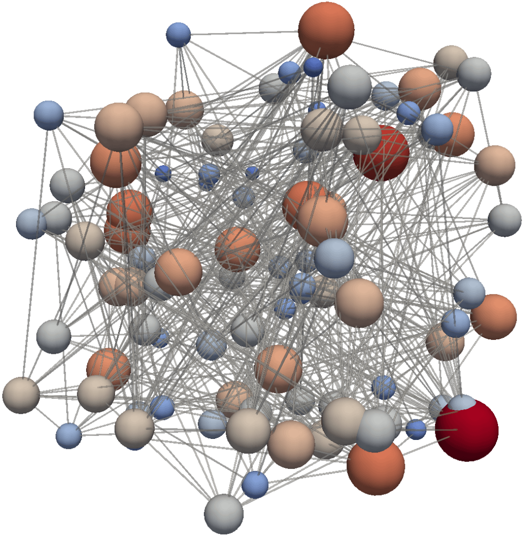
- Function writeObjects() in writeNodesEdges.py
- Stores graphs as vtkPolyData or vtkUnstructuredGrid

```
def writeObjects(nodeCoords,
                 edges = [],
                 scalar = [], name = '', power = 1,
                 scalar2 = [], name2 = '', power2 = 1,
                 nodeLabel = [],
                 method = 'vtkPolyData',
                 fileout = 'test'):
    """
    Store points and/or graphs as vtkPolyData or vtkUnstructuredGrid.
    Required argument:
    - nodeCoords is a list of node coordinates in the format [x,y,z]
    Optional arguments:
    - edges is a list of edges in the format [nodeID1,nodeID2]
    - scalar/scalar2 is the list of scalars for each node
    - name/name2 is the scalar's name
    - power/power2 = 1 for r~scalars, 0.333 for V~scalars
    - nodeLabel is a list of node labels
    - method = 'vtkPolyData' or 'vtkUnstructuredGrid'
    - fileout is the output file name (will be given .vtp or .vtu extension)
    """
```

# Our first graph (randomGraph.py)

```
import networkx as nx
from writeNodesEdges import writeObjects

numberNodes, numberEdges = 100, 500
H = nx.gnm_random_graph(numberNodes,numberEdges)
print 'nodes:', H.nodes()
print 'edges:', H.edges()

# return a dictionary of positions keyed by node
pos = nx.random_layout(H,dim=3)

# convert to list of positions (each is a list)
xyz = [list(pos[i]) for i in pos]

degree = H.degree().values()
writeObjects(xyz, edges=H.edges(), scalar=degree,
             name='degree', fileout='network')
```

# Our first graph (`randomGraph.py`)



# Load this graph into ParaView

- After you run "python randomGraph.py" from the command line, to reproduce the previous slide, you have three options:

1. Load the file `network.vtp`, apply Glyph filter, apply Tube filter, edit their properties, or
2. In ParaView's menu navigate to File -> Load State and select `drawGraph.pvsm`, or
   - important: adjust the data file location!

```
$ grep Users drawGraph.pvsm
<Element index="0" value="/Users/razoumov/teaching/humanities/network.vtp"/>
<Element index="0" value="/Users/razoumov/teaching/humanities/network.vtp"/>
```

3. On a Unix-based system start ParaView and load the state with one command:

```
/Applications/paraview.app/Contents/MacOS/paraview --state=drawGraph.pvsm
```

# Labeling nodes

1. Press V to bring up Find Data dialogue
2. Find Points with ID>=0 (or other selection)
3. Make points visible in the pipeline browser
4. Check Point Labels -> ID (can also do this operation from View -> Selection Display Inspector)
5. Adjust the label font size
6. Set original data opacity to 0

Also we can label only few selected points, e.g., those with degree $\geq 10$

# Switch to spring layout

- Let's apply a force-directed layout

```
$ diff randomGraph.py randomGraph2.py
10c10
< pos = nx.random_layout(H,dim=3)
---
> pos = nx.spring_layout(H,dim=3,k=1)
```

- Run "python randomGraph2.py" from the command line
- Press Disconnect to clear everything from the pipeline browser
- Reload the state file `drawGraph.pvsm`

# Few more graphs: Moebius-Kantor graph

```
$ diff random2.py moebiusKantor.py
5,7c5,6
< H = nx.gnm_random_graph(numberNodes,numberEdges)
< print 'nodes:', H.nodes()
< print 'edges:', H.edges()
---
> H = nx.moebius_kantor_graph()
> print nx.number_of_nodes(H), 'nodes and',
      nx.number_of_edges(H), 'edges'
15a15
> print 'degree =', degree
```

- Run “python moebiusKantor.py” from the command line
- Press Disconnect to clear everything from the pipeline browser
- Reload the state file `drawGraph.pvsm`
- This time probably want to adjust nodes and edges

# Few more graphs: complete bipartite graph

Composed of two partitions with $N$ nodes in the first and $M$ nodes in the second. Each node in the first set is connected to each node in the second.

```
$ diff moebiusKantor.py  completeBipartite.py
5c5
< H = nx.moebius_kantor_graph()
---
> H = nx.complete_bipartite_graph(10,5)
```

- Run “python completeBipartite.py” from the command line
- Press Disconnect to clear everything from the pipeline browser
- Reload the state file `drawGraph.pvsm`

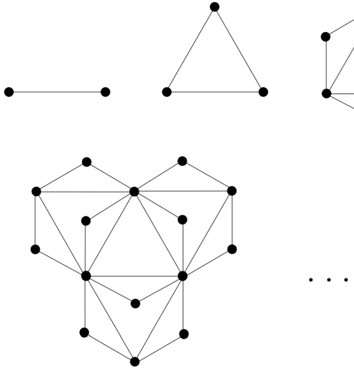
# Your own graphs

We are not limited to NetworkX's built-in graphs.
Can build our own graphs with:

```
H = nx.Graph()
H.add_node(1) # add a single node
H.add_nodes_from([2,3]) # add a list of nodes
H.add_edge(2,3) # add a single edge
H.add_edges_from([(1,2),(1,3)]) # add a list of edges
 ...
```

# Dorogovtsev-Goltsev-Mendes graph

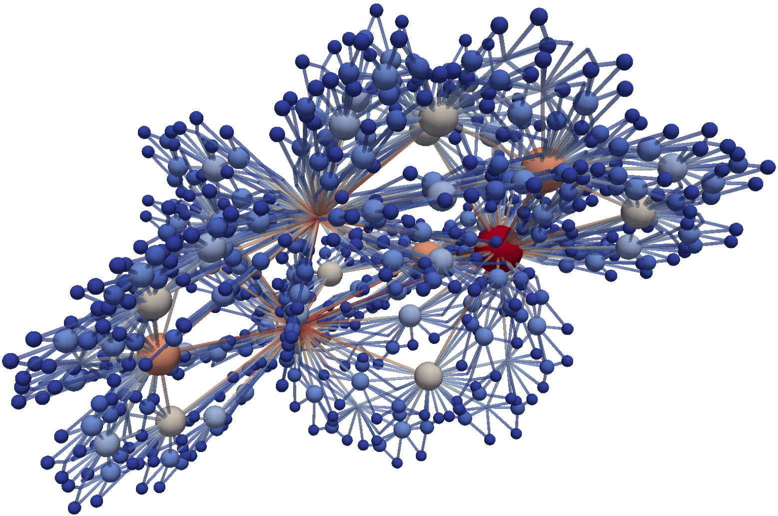
Dorogovtsev-Goltsev-Mendes graph is an interesting fractal network from `http://arxiv.org/pdf/cond-mat/0112143.pdf`. In each subsequent generation, each edge from the previous generation yields a new node, and the new graph can be made by connecting together three previous-generation graphs.



# Dorogovtsev-Goltsev-Mendes graph (dgm.py)

```
import networkx as nx
from forceatlas import forceatlas2_layout
from writeNodesEdges import writeObjects
import sys
generation = int(sys.argv[1])
H = nx.dorogovtsev_goltsev_mendes_graph(generation)

# Force Atlas 2 from https://github.com/tpoisot/nxfa2.git
pos = forceatlas2_layout(H, iterations=100, kr=0.001, dim=3)

# convert to list of positions (each is a list)
xyz = [list(pos[i]) for i in pos]

print nx.number_of_nodes(H), 'nodes and', nx.number_of_edges(H), 'edges'
degree = H.degree(H.nodes()).values()
writeObjects(xyz, edges=H.edges(), scalar=degree,
        name='degree', power=0.333,
        fileout='network')
```

# Dorogovtsev-Goltsev-Mendes graph (7th generation)



# Dorogovtsev-Goltsev-Mendes graph

- From the command line run

```
python dgm.py 1
python dgm.py 2
python dgm.py 3
python dgm.py 4
python dgm.py 7    # takes ~15 seconds on my laptop
```

- Reload the state file `drawGraph.pvsm`, adjust glyph radii, adjust edge colours/radii/opacities

# Custom layouts

Let's first make a flat graph:

```
$ diff dgm.py dgmFlat.py
9c9
< pos=forceatlas2_layout(H, iterations=100, kr=0.001, dim=3)
---
> pos=forceatlas2_layout(H, iterations=100, kr=0.001, dim=2)
12c12
< xyz = [list(pos[i]) for i in pos]
---
> xyz = [[pos[i][0], pos[i][1], 0] for i in pos]
```

Run this with "python dgmFlat.py 5", reload the state file `drawGraph.pvsm`, adjust glyph radii

# Custom layouts

Now let's offset each node in the z-direction by a function of its degree:

```
$ diff dgmFlat.py dgmOffset.py
12,13d11
< xyz = [[pos[i][0], pos[i][1], 0] for i in pos]
15a14,15
> xyz = [[pos[i][0], pos[i][1], (degree[i])**0.5/5.7] for i in pos]
```

Run this with "python dgmOffset.py 5" and colour edges by degree.

# Social network (florentineFamilies.py)

Let's visualize nx.florentine_families_graph(). It returns a list of edges with the nodes indexed by the family name. The function writeObjects() expects integer ID indices instead – hence the loop below.

```
import networkx as nx
from writeNodesEdges import writeObjects
H = nx.florentine_families_graph()
nodes = H.nodes()

# index edges by their node IDs
edges = []
for edge in H.edges():
    edges.append([nodes.index(edge[0]),nodes.index(edge[1])])

# return a dictionary of positions keyed by node
pos = nx.spring_layout(H,dim=3,k=1)

# convert to list of positions (each is a list)
xyz = [list(pos[i]) for i in pos]

degree = H.degree(H.nodes()).values()
writeObjects(xyz, edges=edges, scalar=degree, name='degree',
             fileout='network', nodeLabel=nodes, power=0.333)
```

Note: turn on the labels!

# Highlighting individual nodes

Let's highlight nodes 'Strozzi', 'Tornabuoni', 'Albizzi' with colour.

```
$ diff florentineFamilies.py florentineFamilies2.py
17c17,20
< degree = H.degree(H.nodes()).values()
---
> degree = [1]*len(nodes)
> selection = ['Strozzi', 'Tornabuoni', 'Albizzi']
> for i in selection:
>     degree[nodes.index(i)] = 3
```

# Highlighting individual nodes and edges

Now let's try to highlight the selection and their edges.

$\Rightarrow$ That's very easy: simply colour the edges by node degree.

# Highlighting individual nodes and their neighbours

Let's highlight neighbours of the selected nodes.

```
$ diff florentineFamilies2.py florentineFamilies3.py
20c20,23
<       degree[nodes.index(i)] = 3
---
>       degree[ nodes.index(i) ] = 3
>       for j in list(nx.all_neighbors(H,i)):
>           degree[nodes.index(j)] = max(2.5,degree[nodes.index(j)])
```

# Eigenvector centrality (dgmCentrality.py)

Let's compute and visualize eigenvector centrality in the 5th-deneration Dorogovtsev-Goltsev-Mendes graph with our custom 3D layout.

```
import networkx as nx
from forceatlas import forceatlas2_layout
from writeNodesEdges import writeObjects
H = nx.dorogovtsev_goltsev_mendes_graph(5)
pos = forceatlas2_layout(H, iterations=100, kr=0.001, dim=2)
print nx.number_of_nodes(H), 'nodes␣and', nx.number_of_edges(H), 'edges'
degree = H.degree(H.nodes()).values()
xyz = [[pos[i][0], pos[i][1], (degree[i])**0.5/5.7] for i in pos]

# compute and print eigenvector centrality
ec = nx.eigenvector_centrality(H) # dictionary of nodes with EC as the value
ecList = [ec[i] for i in ec]
print 'degree␣=', degree
print 'eigenvector␣centrality␣=', ecList
print 'min/max␣=', min(ecList), max(ecList)

writeObjects(xyz, edges=H.edges(),
        scalar=degree, name='degree', power=0.333,
        scalar2=ecList, name2='eigenvector␣centrality', power2=0.333,
        fileout='network')
```

- Run "python dgmCentrality.py" and load into ParaView by hand
- Colour by degree, size by eigenvector centrality

# Other statistics in NetworkX

- Various centrality measures: degree, closeness, betweenness, current-flow closeness, current-flow betweenness, eigenvector, communicability, load, dispersion – see `https://networkx.readthedocs.org/en/stable/reference/algorithms.centrality.html`

- Several hundred built-in algorithms for various calculations – see `https://networkx.readthedocs.org/en/stable/reference/algorithms.html`

# Questions?