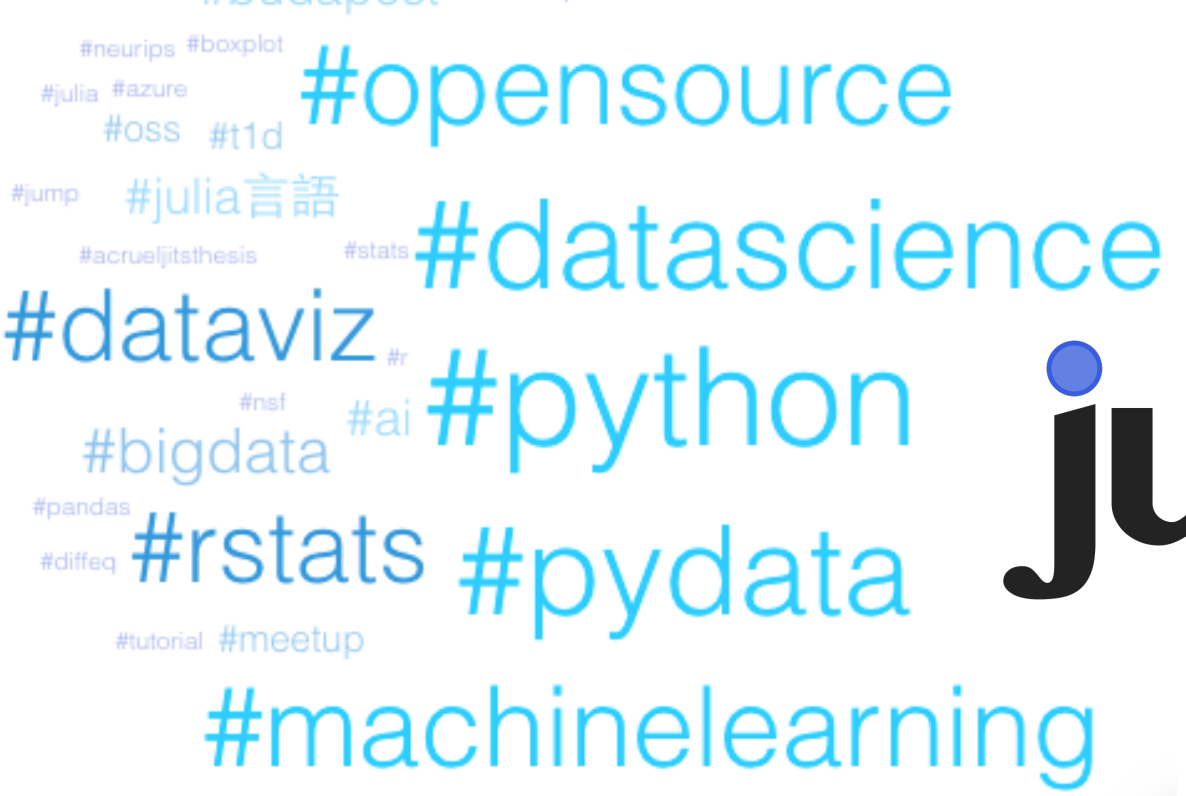


julia for Network Science

Follow along slides: XXX

Analytics Program
University of Chicago
April 1, 2019

with David F. Gleich
Computer Science
Purdue

Huda Nassar
Computer Science
Purdue

We already have Python, Matlab and others, why Julia?

Just a bit of history…

From the reddit post in 2012. We want a language that is…

> We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

We already have Python, Matlab and others, why Julia?

Just a bit of history…

From the reddit post in 2012. We want a language that is…

> We want a language that's **open source**, with a liberal license. We want the **speed of C** with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, **familiar mathematical notation like Matlab**. We want something as usable **for general programming as Python**, as easy for statistics as R, as natural for string processing as Perl, as **powerful for linear algebra as Matlab**, as good at gluing programs together as the shell. Something that is **dirt simple to learn**, yet keeps the most serious hackers happy. We want it **interactive** and we want it **compiled**.

Julia's syntax is nothing to be afraid of.

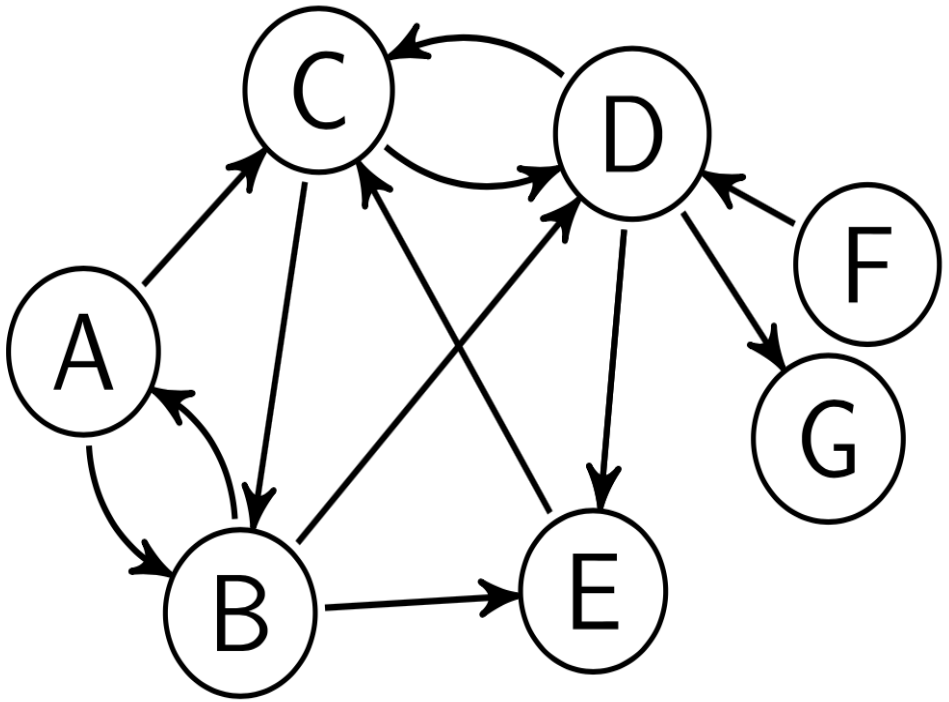
Iterative Fibonacci in Julia, Python, and Matlab

```
function fibonacci(n)
  Fn = [1,1]
  for i = 3:n
    c = Fn[1]+Fn[2]
    Fn[1] = Fn[2]
    Fn[2] = c
  end
  return Fn[2]
end
```

```
def fib(n):
    Fn = [1,1]
    for i in xrange(2,n):
        c = Fn[0] + Fn[1]
        Fn[0] = Fn[1]
        Fn[1] = c
    return Fn[1]
```

```
function F = fibonacci(n)
  Fn = [1 1];
  for i = 3:n
    c = Fn(1) + Fn(2);
    Fn(1) = Fn(2);
    Fn(2) = c;
  end
  F = Fn(2);
end
```

Graphs/Networks/Matrices are the same.



$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \end{array}
\begin{array}{c}
\begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\
\left[\begin{array}{ccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\end{array}
$$

Real World Graphs

Collaboration Networks

Nodes: Authors

Edges: Co-authorship

Example: Collaboration of Arxiv AstroPhysics

Social Networks

Nodes: People

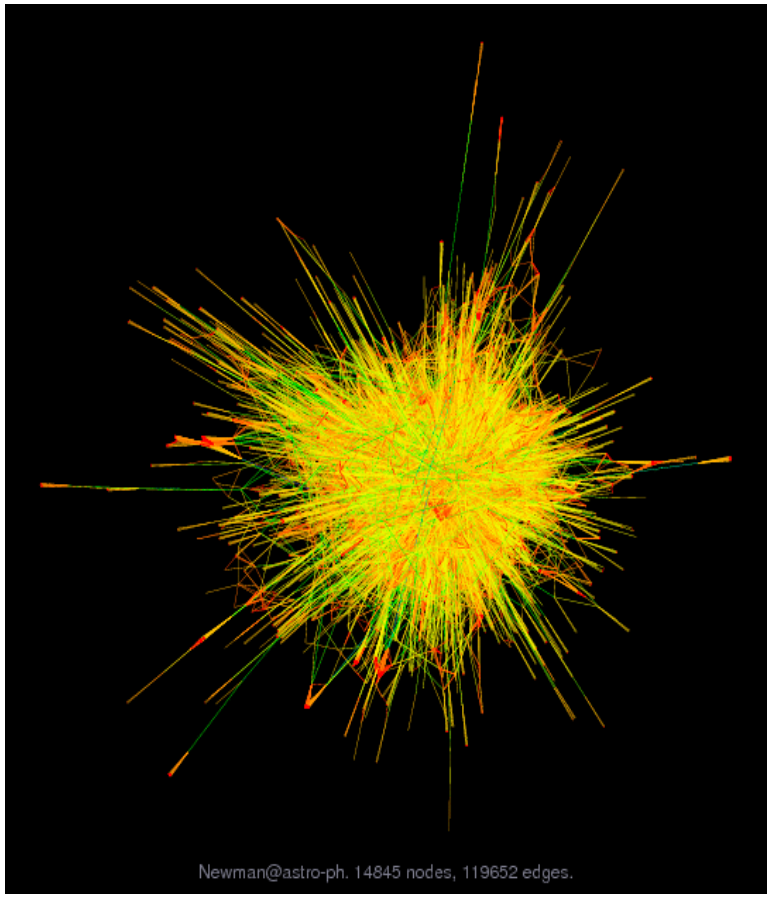
Edges: Social Interaction

Example: Facebook

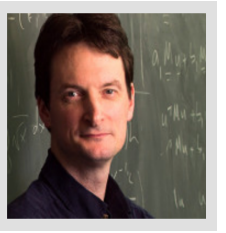
Technological Networks

Edges: Physical Infrastructure

Example: Power-grid



Collaboration network of scientists posting preprints on the astrophysics archive at www.arxiv.org, 1995-1999, as compiled by M. Newman.



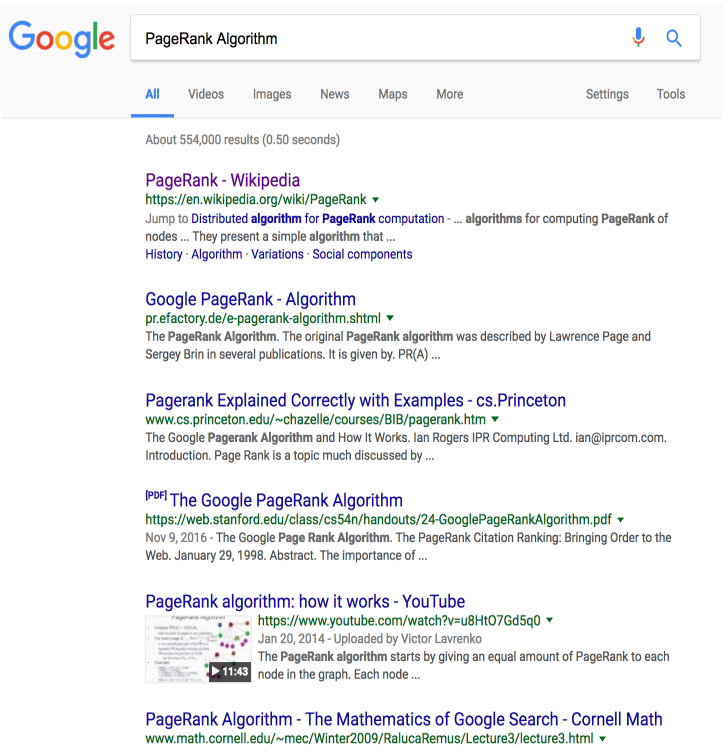
SuiteSparse Matrix Collection
Formerly the University of Florida Sparse Matrix Collection

SNAP

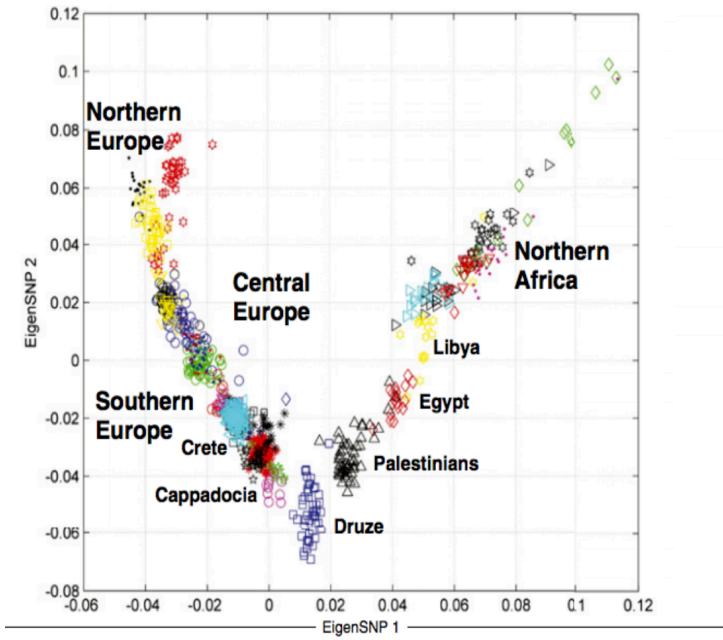
Problems on Graphs

Matrix Completion

Ranking



Exploratory Data Analysis



P. Paschou, **P. Drineas**, et al, Maritime route of colonization of Europe, *Proceedings of the National Academy of Sciences*, doi:10.1073/pnas.1320811111, 2014.

Follow along slides @ https://www.cs.purdue.edu/homes/hnassar/uchicago-julia.pdf

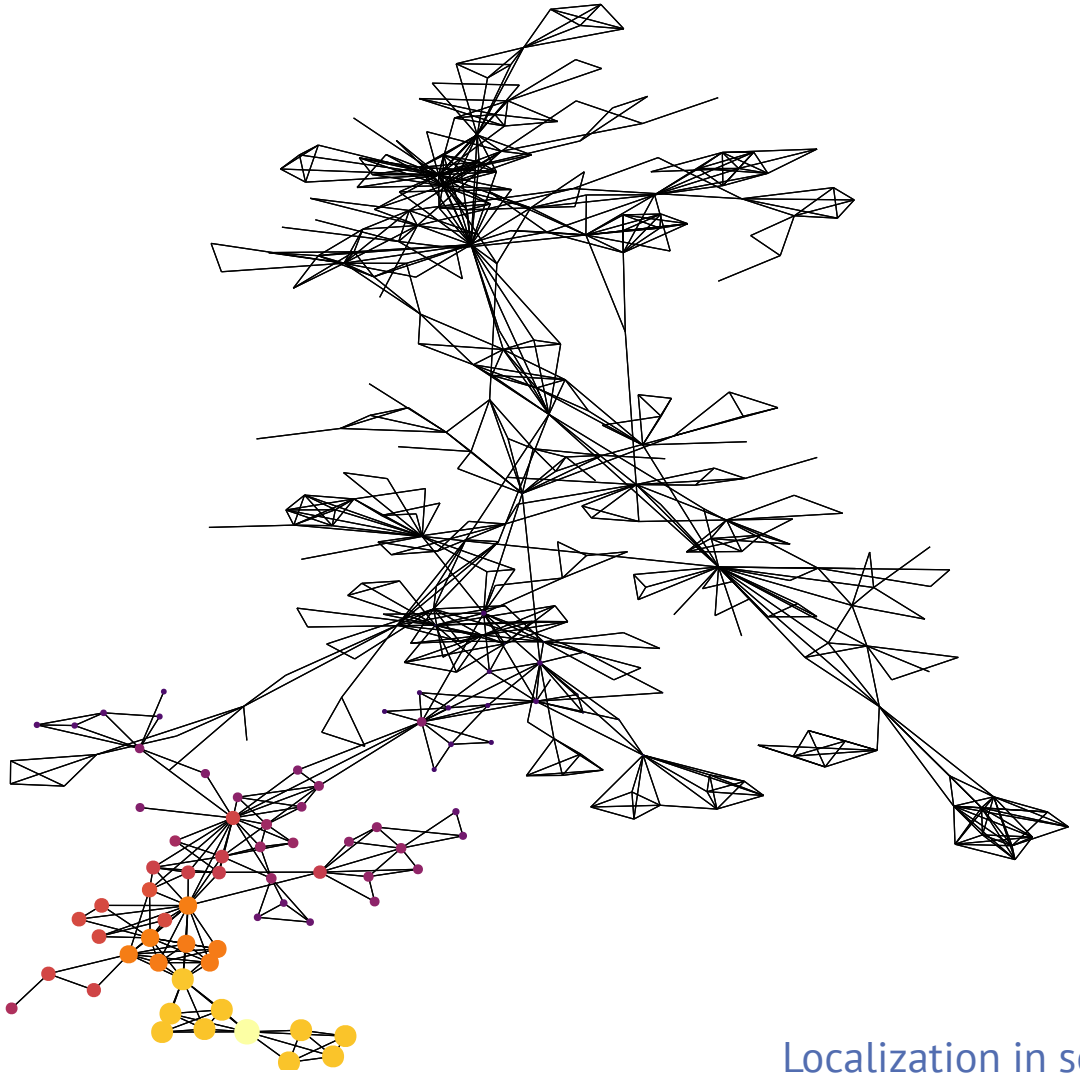
Many operations we want to perform on networks are (in fact) linear algebra.

Network Operation	Matrix Operation
Ranking (eg. PageRank)	$(\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x} = (1-\alpha)\mathbf{v}$
Network Alignment (eg. IsoRank)	$\mathbf{x} = \alpha(\boldsymbol{A} \otimes \boldsymbol{B})\mathbf{x} + (1-\alpha)\mathbf{h}$
Diffusion (eg. HeatKernel)	$\mathbf{x} = \exp\left(-t(\boldsymbol{I} - \boldsymbol{P})\right)\mathbf{s}$
Community Detection (eg. Random walks)	$\mathbf{x} = \boldsymbol{P}^k\mathbf{s}$
Clustering (eg. Spectral Clustering)	$\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$

These give simple Julia implementations

```
x = (I - alpha*P) \ (1-alpha)*v  # PageRank
x = expm(-t*(I-P))*v             # Heat Kernel
```

The project that caused my first Julia experiment: generate really large graphs and compute seeded PageRank



Strong Localization in Personalized PageRank
Nassar, Kloster, Gleich, WAW2015
Localization in seeded PageRank (Internet Math, 2017)

I was a Matlab user before Julia, but "Matlab+mex" was not good enough

- Interface is harder
- Weak compatibility of older codes with new versions
- Harder to try out ideas and analyze intermediate results
- Inconsistent indexing
- Common source of frustration, bugs and wasted time ☹

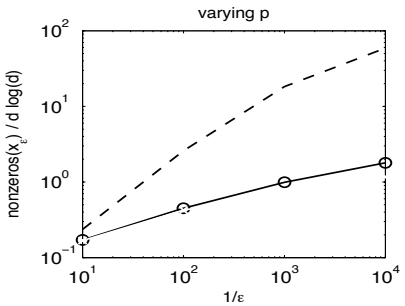
Julia's wrappers make it easy to use existing C code

Previously,

run
c code

save it to the
hard disk

read it into
matlab



run experiments

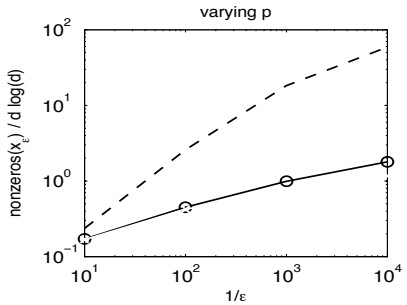
Julia's wrappers make it easy to use existing C code

Previously,

run
c code

save it to the
hard disk

read it into
matlab



run experiments

file size ~ 40GB
when network is large

too much time!

O(n) algorithm for
compute time

Julia's wrappers make it easy to use existing C code

The Julia solution:

1. use existing C code
2. use Julia function call of the form

```
ccall ( (:check_graphical_sequence, libpath),
          Cint, # return value
          (Int64, Ptr{Int64}), # arg types
          n, degs); # actual args
```

3. wrap inside a Julia function

Result: Can generate a graph with 1B nodes and 2B+ edges in less than an hour

nodes/edges	time
1K/2K+	0.001 secs
1M/2M+	5 secs
100M/200M+	260 secs
1B/2B+	3570 secs

Julia's wrappers make it easy to use existing C code

To compare the the 2 approaches

With Julia

nodes/edges	time
1K/2K+	0.001 secs
1M/2M+	5 secs
100M/200M+	260 secs
1B/2B+	3570 secs

Previously,

nodes/edges	time
1K/2K+	0.2 secs (200x)
1M/2M+	41 secs (10x)
100M/200M+	3300 secs (8x)
1B/2B+	9941 secs (2.7 hours) (3x)

First project: a success! Then I wanted to explore Julia

Julia is fast seems to be the selling point.

Let's test the following codes: Find fib(1), fib(2),..., fib(50)

```
function fibonacci(n)
  Fn = [1,1]
  for i = 3:n
    c = Fn[1]+Fn[2]
    Fn[1] = Fn[2]
    Fn[2] = c
  end
  return Fn[2]
end
```

```
def fib(n):
    Fn = [1,1]
    for i in xrange(2,n):
        c = Fn[0] + Fn[1]
        Fn[0] = Fn[1]
        Fn[1] = c
    return Fn[1]
```

```
function F = fibonacci(n)
  Fn = [1 1];
  for i = 3:n
    c = Fn(1) + Fn(2);
    Fn(1) = Fn(2);
    Fn(2) = c;
  end
  F = Fn(2);
end
```

Julia is fast seems to be the selling point.

Let's test the following codes: Find fib(1), fib(2),…, fib(50)

```
julia> include("fibonacci.jl")
fibonacci (generic function with 1 method)

julia> fibonacci(10)
55

julia> global totaltime = 0
0

julia> for i = 1:50
           global totaltime
           totaltime += @elapsed f = fibonacci(i
       end

julia> totaltime/50
4.3269999999999984e-7
```

4.32e-7

```
$ python
Python 2.7.14 |Anaconda, Inc.| (default, Oct 13 2
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license"
>>> import time
>>> import fibonacci
>>> fibonacci.fib(10)
55
>>> totaltime = 0
>>> for i in xrange(1,51):
...     start = time.time();f=fibonacci.fib(i);en
...     totaltime += end-start
...
>>> totaltime/50
7.753372192382813e-06
```

7.75e-6

```
>> totaltime = 0

totaltime =

     0

>> for i = 1:50
       tic;f = fibonacci(i);t=toc;
       totaltime = totaltime+t;
   end
>> totaltime/50

ans =

   9.0780e-05
```

9.07e-5

more at https://julialang.org/benchmarks/

But why? Major advantage: Julia is compiled, it uses a JIT compiler

```
julia> function add2(n)
           return n+2
       end
add2 (generic function with 1 method)
```

```
julia> @time add2(1)
  0.000032 seconds (5 allocations: 240 bytes)
3

julia> @time add2(1)
  0.000003 seconds (4 allocations: 160 bytes)
3
```

But why? Major advantage: Julia is compiled, it uses a JIT compiler

```
julia> function add2(n)
           return n+2
       end
add2 (generic function with 1 method)
```

```
julia> @time add2(1)
  0.000032 seconds (5 allocations: 240 bytes)
3
```

→ compiled here ←

```
julia> @time add2(1)
  0.000003 seconds (4 allocations: 160 bytes)
3
```

But why? Major advantage: Julia is compiled, it uses a JIT compiler

```
julia> function add2(n)
           return n+2
       end
add2 (generic function with 1 method)
```

```
julia> @time add2(1)
  0.000032 seconds (5 allocations: 240 bytes)
3
```

→ compiled here ←

```
julia> @time add2(1)
  0.000003 seconds (4 allocations: 160 bytes)
3
```

```
julia> @code_llvm add2(3)

; Function add2
; Location: REPL[588]:2
define i64 @julia_add2_37220(i64) {
top:
; Function +; {
; Location: int.jl:53
  %1 = add i64 %0, 2
;}
  ret i64 %1
}
```

```
julia> @code_native add2(3)
        .section        __TEXT,__text,regular,pure_instructions
; Function add2 {
; Location: REPL[588]:2
; Function +; {
; Location: REPL[588]:2
        decl    %eax
        leal    2(%edi), %eax
;}
        retl
        nopw    %cs:(%eax,%eax)
;}
```

MatrixNetworks.jl was born after I realized the speed of Julia

Basic premise: Treat a matrix like a network and a network like a matrix (our research group's mission is to exploit structure in this setting).

- **Primitives**: Manage common implementation primitives (connected components, shortest paths, etc.)
- **Research generated algorithms**: Provide "production ready" algorithms (when we get them, i.e. no research code)

MatrixNetworks.jl was born after I realized the speed of Julia

Basic pren matrix
(our resea).

- **Primiti** ected compo
- **Resear** orithms (when

MatrixNetworks

This package consists of a collection of network algorithms. In short, the major difference between MatrixNetworks.jl and packages like LightGraphs.jl or Graphs.jl is the way graphs are treated.

In LightGraphs.jl, graphs are created through Graph() and DiGraph() which are based on the representation of G as G = (V,E). Similar types exist in Graphs.jl (EdgeList, AdjacencyList, IncidenceList, Graph) - this is again based on viewing a graph G as a set of nodes and edges. Our viewpoint is different.

MatrixNetworks is based on the philosophy that there should be no distinction between a matrix and a network - thus the name.

For example, `d,dt,p = bfs(A,1)` computes the bfs distance from the node represented by row 1 to all other nodes of the graph with adjacency matrix A. (A can be of type `SparseMatrixCSC` or `MatrixNetwork`). This representation can be easier to work with and handle.

The package provides documentation with sample runs for all functions - viewable through Juila's REPL. These sample runs come with sample data, which makes it easier for users to get started on `MatrixNetworks`.

Package Installation:

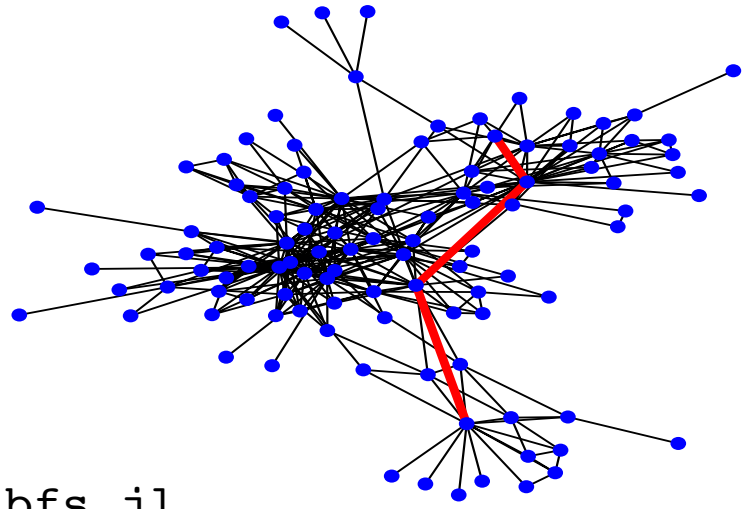
To install package

```
Pkg.add("MatrixNetworks")
using MatrixNetworks
```

Example

Stuff in MatrixNetworks.jl

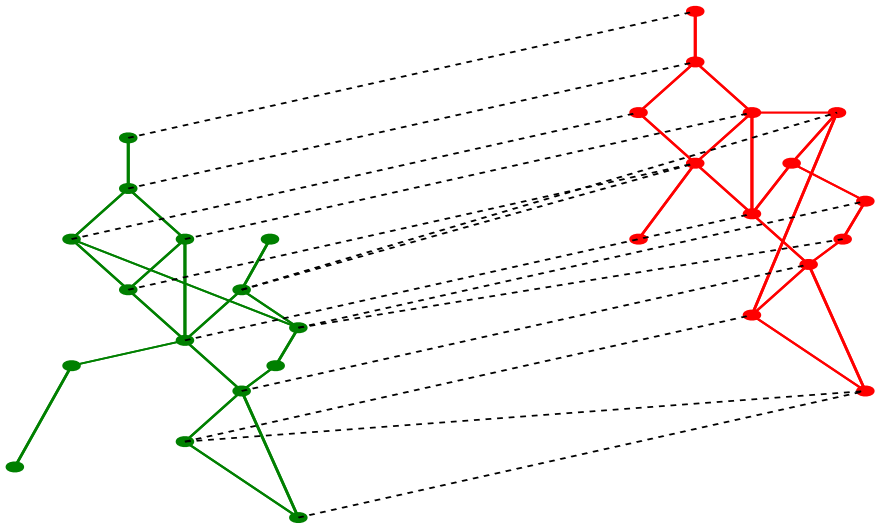
- Basics: Connected Components, Paths, etc.
- Diffusions (PageRank + Heat Kernel)
- Spectral clustering (custom ARPACK wrapper for type stability)
- Random graph models



`plot_bfs.jl`

Next: I worked on a network alignment package `NetworkAlignment.jl`

Goal: want to implement various state-of-the-art network alignment algorithms. One problem boils down to solving a bipartite matching problem on each row of the matrix



What is the best way of matching graph A (green) to graph B (red) using edges in **L (dashed lines)**?

Julia makes it easy to manipulate large sparse matrices

The Julia solution:

1. Simply re-write the entire code in native Julia
    1. Make use of any Julia feature available
    2. Write type-stable julia code (will return to this in 2 slides)

Result: Our Julia implementation beats the C++ implementation in most cases (both very well written)

Problem size	time
12 nodes	0.003 secs
16952 nodes	0.18 secs
4971629 nodes	40 secs

Julia makes it easy to manipulate large sparse matrices

To compare the the 2 approaches

With Julia

Problem size	time
12 nodes	0.003 secs
16952 nodes	0.18 secs
4971629 nodes	40 secs

Previously (with C++),

Problem size	time
12 nodes	0.018 secs
16952 nodes	0.24 secs
4971629 nodes	50 secs

Julia is fast, with a catch!

Types matter

```
In [18]: @btime flipcoin_then_add(rand(1000))
         @btime flipcoin_then_add_typed(rand(1000))
```

```
  10.286 µs (1 allocation: 7.94 KiB)
  3.425 µs (1 allocation: 7.94 KiB)
```

Julia is fast, with a catch!

Types matter

```
In [18]: @btime flipcoin_then_add(rand(1000))
         @btime flipcoin_then_add_typed(rand(1000))
```

```
  10.286 µs (1 allocation: 7.94 KiB)
  3.425 µs (1 allocation: 7.94 KiB)
```

Memory allocation matters

```
In [22]: n = 100
         @btime build_fibonacci_no_allocation(n);
         @btime build_fibonacci_preallocate(n);
```

```
  1.265 µs (7 allocations: 2.27 KiB)
  255.648 ns (1 allocation: 896 bytes)
```

Julia is fast, with a catch!

Types matter

```
In [18]: @btime flipcoin_then_add(rand(1000))
         @btime flipcoin_then_add_typed(rand(1000))
```

```
  10.286 μs (1 allocation: 7.94 KiB)
  3.425 μs (1 allocation: 7.94 KiB)
```

Memory allocation matters

```
In [22]: n = 100
         @btime build_fibonacci_no_allocation(n);
         @btime build_fibonacci_preallocate(n);
```

```
  1.265 μs (7 allocations: 2.27 KiB)
  255.648 ns (1 allocation: 896 bytes)
```

Memory access matters

```
In [28]: @btime matrix_sum_rows(A)
         @btime matrix_sum_cols(A)
         @btime matrix_sum_index(A)
```

```
  2.247 s (1 allocation: 16 bytes)
  122.958 ms (1 allocation: 16 bytes)
  131.959 ms (1 allocation: 16 bytes)
```

Julia is fast, with a catch!

Types matter

```
In [18]: @btime flipcoin_then_add(rand(1000))
         @btime flipcoin_then_add_typed(rand(1000))
```

```
  10.286 µs (1 allocation: 7.94 KiB)
  3.425 µs (1 allocation: 7.94 KiB)
```

Memory allocation matters

```
In [22]: n = 100
         @btime build_fibonacci_no_allocation(n);
         @btime build_fibonacci_preallocate(n);
```

```
  1.265 µs (7 allocations: 2.27 KiB)
  255.648 ns (1 allocation: 896 bytes)
```

Memory access matters

```
In [28]: @btime matrix_sum_rows(A)
         @btime matrix_sum_cols(A)
         @btime matrix_sum_index(A)
```

```
  2.247 s (1 allocation: 16 bytes)
  122.958 ms (1 allocation: 16 bytes)
  131.959 ms (1 allocation: 16 bytes)
```

More at:
https://github.com/nassarhuda/JuliaTutorials/

Performance Tips for Julia

Moving on to the hands-on portion of this talk. A Julia tutorial

Go to: juliabox.com

tutorials/

broader-topics-and-ecosystem/

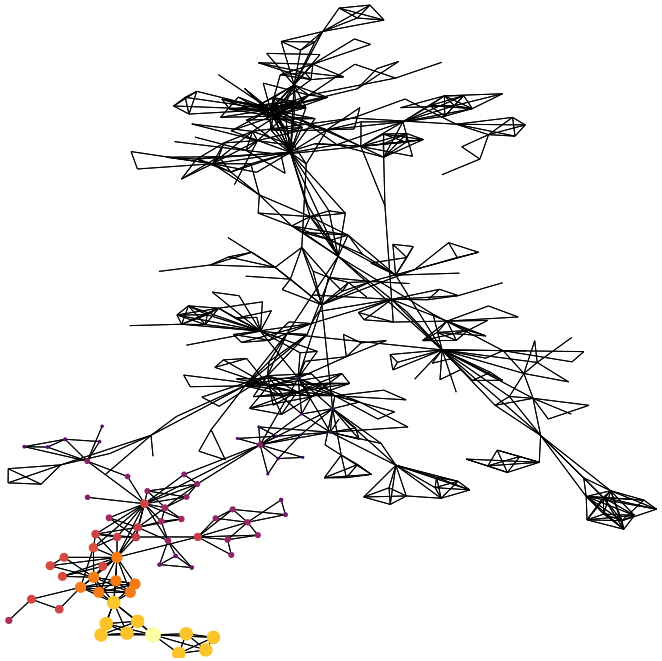
intro-to-julia-for-data-science/

Thank you!
I have stickers if anyone wants ☺

Some resources:

- Slack: https://slackinvite.julialang.org/
- Discourse: https://discourse.julialang.org/
- Github: https://github.com/JuliaLang/julia
- Intro to Julia for data science @ https://www.youtube.com/watch?v=SLE0vz85Rqo (Alternatively: Google "Julia data science Huda Nassar")

The project that caused my first Julia experiment: generate really large graphs and compute seeded PageRank



For seeded PageRank, the random walk always restarts on the same node. The solution is mathematically non-zero but practically "zero" on most of the nodes. Q: How many "non-zero"s?

My task

1. Generate a random graph with a power law degree distribution using a C program
2. Compute seeded PageRank
3. Sort vector and compute number of large entries.

Strong Localization in Personalized PageRank
Nassar, Kloster, Gleich, WAW2015
Localization in seeded PageRank (Internet Math, 2017)