Estimating the fundamental matrix of a random walk transition matrix

Clara De Paolis Kaluza

Math250- Graph Algorithms

Final project

Abstract

The analysis of distances between nodes in networks that contain many low-degree vertices and some high-degree hubs requires more sophisticated distance metrics than, for example, simple shortest-path distances. In the study of one type of these networks, protein-protein interaction networks, a promising distance metric is the diffusion state distance, but calculating requires an expensive matrix inversion. Through reforming the problem as a linear system theoretical performance improvements are possible, although the experimental results shown here show numerical instability issues.

1. Problem Statement and Motivation

1.1. Motivation

Genome sequencing allows for the study of all the proteins expressed by the genome of an organism (the proteome). For most organisms, some of these proteins have known biological functions, but for many proteins their biological function is unknown. A protein-protein interaction (PPI) network relates the structure of the proteins expressed by a genome, representing proteins that physically interact as connected nodes in a graph. These graphs capture physical interactions between proteins, including those with known function and those with unknown functions. Therefore if a proper distance metric can be determined between nodes, the structure of these networks can be used to discover the function of uncharacterized proteins[5].

However, these networks can be very complex, and furthermore, determining an appropriate distance metric is not straightforward. Some nodes in the network are considered "hubs," connecting many proteins that are not functionally similar. Therefore, a distance metric such as a simple shortest path is not useful in identifying functionally similar proteins. The work in [2] shows a promising distance metric, the diffusion state distance (DSD), but calculating it exactly requires the inversion of a matrix corresponding to a large network even for simple organisms. A more efficient method to calculate this metric is needed, especially in order to analyze the much larger PPI networks of more complex organisms and to apply this metric to other, larger networks. The main objective of this project is to implement an efficient way to estimate the DSD while avoiding an exact calculation of the fundamental matrix, an inverse of an $n \times n$ matrix, which is approximately an $O(n^3)$ operation.

2. Background and Assumptions

2.1. Problem Setup

To more precisely define the problem, consider a connected graph $G = (V, E)$ where nodes $v \in V$ represent proteins expressed by a genome and edges $e \in E$ represent physical interaction between those proteins. The number of edges e connected to any node i is the degree of that node, represented by $d(i)$. If we define the probability of a transition from node i to any of its neighboring nodes j as uniform then, the state transition matrix $\mathbf{P}$ for this network is given by

$$(\mathbf{P})_{ij} = \begin{cases} \frac{1}{d(i)} & e(i,j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

For the Markov chain this network represents, the matrix $\mathbf{W} = \lim_{n\to\infty} \mathbf{P}^n$ is such that each row of $\mathbf{W}$ is equal to $\boldsymbol{\pi}^T$, the vector describing the steady state distribution for each $v \in V$. Finally, the *fundamental matrix* of $\mathbf{P}$ is given by $\mathbf{Z} = (\mathbf{I} - \mathbf{P} + \mathbf{W})^{-1}$.

The distance metric proposed in [2] between two nodes (proteins) u and v is then defined as:

$$\text{DSD}(u, v) = ||(\mathbf{b}_u^T - \mathbf{b}_v^T)(\mathbf{I} - \mathbf{P} + \mathbf{W})^{-1}||_1 \tag{2}$$

where $\mathbf{b}_i$ is a basis vector, where all entries are zero except entry in the i^{th} position is one.

2.2. Defining The Graph Problem

For compactness, define the following two vectors $\mathbf{b}_{uv}^T := (\mathbf{b}_u^T - \mathbf{b}_v^T)$ and $\mathbf{x}_{uv}^T := \mathbf{b}_{uv}^T(\mathbf{I} - \mathbf{P} + \mathbf{W})^{-1}$ so that $\text{DSD}(u, v) = \|\mathbf{x}_{uv}^T\|_1 = \|\mathbf{x}_{uv}\|_1$. By rearranging the terms, we can state this problem in the familiar form of solving a linear system:

$$\mathbf{x}_{uv}^T = \mathbf{b}_{uv}^T(\mathbf{I} - \mathbf{P} + \mathbf{W})^{-1}$$

$$\begin{aligned} ((\mathbf{I} - \mathbf{P} + \mathbf{W})^T)^{-1}\mathbf{b}_{uv}^T &= \mathbf{x}_{uv} \\ (\mathbf{I} - \mathbf{P} + \mathbf{W})^T\mathbf{x}_{uv} &= \mathbf{b}_{uv} \\ (\mathbf{I} - \mathbf{P}^T + \mathbf{W}^T)\mathbf{x}_{uv} &= \mathbf{b}_{uv} \end{aligned}$$

From the definition of the state transition matrix in (1), $\mathbf{P}$ can be represented in terms of the matrices that describe a graph, namely, the degree matrix $\mathbf{D}$, the adjacency matrix $\mathbf{A}$, and the graph Laplacian $\mathbf{L}$.

$$\begin{aligned} &\mathbf{P} = \mathbf{D}^{-1}\mathbf{A} \\ \Rightarrow\ &\mathbf{P}^T = \mathbf{A}\mathbf{D}^{-1} \\ \Rightarrow\ &(\mathbf{I} - \mathbf{P}^T) = \mathbf{I} - \mathbf{A}\mathbf{D}^{-1} = (\mathbf{D} - \mathbf{A})\mathbf{D}^{-1} = \mathbf{L}\mathbf{D}^{-1} \end{aligned}$$

So the linear system to be solved becomes

$$(\mathbf{I} - \mathbf{P}^T + \mathbf{W}^T)\mathbf{x}_{uv} = (\mathbf{L}\mathbf{D}^{-1} + \mathbf{W}^T)\mathbf{x}_{uv} = \mathbf{b}_{uv} \tag{3}$$

To further simplify, consider the Sherman-Morrison formula for inverting the sum of a square invertible matrix $\mathbf{A}$ and the outer product of two vectors $\mathbf{u}$ and $\mathbf{v}$:

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T\mathbf{A}^{-1}}{1 + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}} \tag{4}$$

with the restriction that $1 + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u} \neq 0$.

For this problem, the first matrix is $(\mathbf{I} - \mathbf{P}^T)$, and the two column vectors are the vector of all ones $\mathbf{e}$ and π, the stationary distribution of the Markov chain described by $\mathbf{P}$ so that $\mathbf{W} = \mathbf{e}\pi^T$

So substituting into the above equation, we get a solution to finding the fundamental matrix $\mathbf{Z}$:

$$\begin{aligned}
\mathbf{Z}^T = (\mathbf{I} - \mathbf{P}^T + \mathbf{W}^T)^{-1} &= (\mathbf{I} - \mathbf{P}^T + (\mathbf{e}\pi^T)^T)^{-1} = (\mathbf{I} - \mathbf{P}^T + \pi\mathbf{e}^T)^{-1} \\
&= (\mathbf{I} - \mathbf{P}^T)^{-1} - \frac{(\mathbf{I} - \mathbf{P}^T)^{-1}\pi\mathbf{e}^T(\mathbf{I} - \mathbf{P}^T)^{-1}}{1 + \mathbf{e}^T(\mathbf{I} - \mathbf{P}^T)^{-1}\pi} \\
&= (\mathbf{L}\mathbf{D}^{-1})^{-1} - \frac{(\mathbf{L}\mathbf{D}^{-1})^{-1}\pi\mathbf{e}^T(\mathbf{L}\mathbf{D}^{-1})^{-1}}{1 + \mathbf{e}^T(\mathbf{L}\mathbf{D}^{-1})^{-1}\pi} \\
&= \mathbf{D}\mathbf{L}^{-1} - \frac{\mathbf{D}\mathbf{L}^{-1}\pi\mathbf{e}^T\mathbf{D}\mathbf{L}^{-1}}{1 + \mathbf{e}^T\mathbf{D}\mathbf{L}^{-1}\pi}
\end{aligned}$$

Substituting into (3), the linear system becomes

$$\begin{aligned}
\mathbf{x}_{uv} &= \mathbf{Z}^T\mathbf{b}_{uv} \\
&= \left(\mathbf{D}\mathbf{L}^{-1} - \frac{\mathbf{D}\mathbf{L}^{-1}\pi\mathbf{e}^T\mathbf{D}\mathbf{L}^{-1}}{1 + \mathbf{e}^T\mathbf{D}\mathbf{L}^{-1}\pi}\right)\mathbf{b}_{uv} \\
&= \mathbf{D}\mathbf{L}^{-1}\mathbf{b}_{uv} - \frac{\mathbf{D}\mathbf{L}^{-1}\pi\mathbf{e}^T\mathbf{D}\mathbf{L}^{-1}\mathbf{b}_{uv}}{1 + \mathbf{e}^T\mathbf{D}\mathbf{L}^{-1}\pi} \\
&= \mathbf{D}\mathbf{y}_{uv} - \frac{\mathbf{D}\mathbf{z}\,\mathbf{e}^T\mathbf{D}\mathbf{y}_{uv}}{1 + \mathbf{e}^T\mathbf{D}\mathbf{z}} \\
&= \mathbf{D}\mathbf{y}_{uv} - \left(\frac{\mathbf{e}^T\mathbf{D}\mathbf{y}_{uv}}{1 + \mathbf{e}^T\mathbf{D}\mathbf{z}}\right)\mathbf{D}\mathbf{z}
\end{aligned}$$

where $\mathbf{y}_{uv} = \mathbf{L}^{-1}\mathbf{b}_{uv}$ and $\mathbf{z} = \mathbf{L}^{-1}\pi$. Therefore to solve this system, the following two linear systems of graph Laplacian must be solved:

$$\mathbf{L}\mathbf{y}_{uv} = \mathbf{b}_{uv} \text{ and } \mathbf{L}\mathbf{z} = \pi \tag{5}$$

3. Methods

3.1. Approach

Using the solution to the systems in (5), the original problem can be solved by following the approach outlined in Algorithm 1. Here, we use an aggregation-based algebraic multigrid approach to solve the two linear systems in steps 1 and 2, however this method can be substituted by an alternative linear system solving method.

Algorithm 1 Compute DSD(u, v) given $\mathbf{b}_{uv}$, graph Laplacian $\mathbf{L} = \mathbf{D} - \mathbf{A}$, and π

1. Solve $\mathbf{L}\mathbf{y}_{uv} = \mathbf{b}_{uv}$
2. Solve $\mathbf{L}\mathbf{z} = \pi$
3. Compute $\mathbf{f} = \mathbf{D}\mathbf{y}_{uv}$ and $\mathbf{g} = \mathbf{D}\mathbf{z}$
4. Compute $\mathbf{x}_{uv} = \mathbf{f} - \left(\dfrac{\mathbf{e}^T\mathbf{f}}{1 + \mathbf{e}^T\mathbf{g}}\right)\mathbf{g}$
5. Compute DSD$(u, v) = \|\mathbf{x}_{uv}^T\|_1 = \|\mathbf{x}_{uv}\|_1$

3.2. Solving the Linear Systems

The two linear systems could be solved using several possible methods to achieve a performance gain over directly computing the fundamental matrix (if it can even be computed), but the method implemented in this work is an aggregation-based algebraic multigrid (AMG) method. AMG methods can be used to solve graph Laplacian systems and take the general form of forming aggregates to form P_l then constructing the graph Laplacian for each coarser level using $L_{l+1} = P_l^T L_l P_l$, then recursively calling the AMG cycle algorithm (Algorithm 2)

Algorithm 2 Algebraic Multigrid with Aggregations, AMGCyle(x_l, L_l, b_l, l)

- if at coarsest level:
  - solve $x = L_l^{-1} b_l$ directly
- else:
  - Pre-smoothing (update to x_l)
  - Compute residual $r_l \leftarrow b_l - L_l x_l$
  - Restriction $r_c \leftarrow P_l^T r_l$
  - Coarse-grid correction (recursive call to AMG Cycle)
  - Prolongation $x_l \leftarrow x_l + P_l e_{l+1}$
  - Post-smoothing (update to x_l)

4. Results

4.1. Solving $\mathbf{L}\mathbf{y}_{uv} = \mathbf{b}_{uv}$

Table 1 shows the performance of AMG in solving $\mathbf{L}\mathbf{y}_{uv} = \mathbf{b}_{uv}$. Since the DSD between each pair or nodes in the PPI is needed, the calculation must be performed for each pair of unique nodes (since DSD$(u, u) = 0$ and DSD(u, v) =DSD(v, u), as shown in [2]).

Species	$\|V\|$	$\|E\|$	$d_{\max}$	d_{avg}	method	setup time(s)	num iters	solve time(s)
worm	5,281	13,829	225	5.237				
					two-level AMG	2.327	23	3.641
					V-cycle AMG	2.358	24	3.138
					W-cycle AMG	2.389	23	5.193
mouse	6,596	18,697	714	5.669				
					two-level AMG	2.100	29	10.959
					V-cycle AMG	3.274	30	8.824
					W-cycle AMG	3.133	29	10.97
yeast	6,096	216,531	3,472	71.040				
					two-level AMG	2.434	10	269.132
					V-cycle AMG	3.209	10	277.139
					W-cycle AMG	4.059	10	270.089
human	15,129	155,866	9,388	20.605				
					two-level AMG	5.556	14	1543.422
					V-cycle AMG	9.300	*	*
					W-cycle AMG	9.644	*	*

Table 1: Results of solving the linear system $\mathbf{L}\mathbf{y}_{uv} = \mathbf{b}_{uv}$ with a tolerance for the relative residual of for the largest connected component of the PPI network for several species.
The number of proteins (vertices in G) is indicated by $|V|$ the number of edges by $|E|$, the maximal degree by $d_{\max}$, and the average degree by d_{avg}.
*For the human PPI network, the V-cycle AMG method reached a relative residual of 2.021×10^{-8} at 12 iterations before starting to increase again. For the W-cylce, the relative residual reached 8.647×10^{-6} at 5 iterations before increasing

4.2. Solving $\mathbf{L}\mathbf{z} = \pi$

Using the same method as above was used to solve $\mathbf{L}\mathbf{y}_{uv} = \mathbf{b}_{uv}$, the AMG method for solving $\mathbf{L}\mathbf{z} = \pi$ does not converge, with the rate oscillating around 1, therefore the residual never converging towards the tolerance set by the algorithm. To investigate this behavior, the system $\mathbf{L}\mathbf{z} = \pi$ was solved using two different solvers available through Python's SciPy and NumPy packages[3]. SciPy provides a sparse matrix solver `scipy.sparse.linalg.spsolve` and Numpy provides a solver `numpy.linalg.solve`. When used on this linear system, the former found $\mathbf{z} = (4.876 \times 10^{11})\mathbf{e}$ and the latter found $\mathbf{z} = (6.546 \times 10^{12})\mathbf{e}$ for the worm PPI network, where $\mathbf{e}$ denotes the vector of all ones. Both solutions are clearly very far from both the starting guesses for $\mathbf{z}$ ($\mathbf{z} = \text{ones}(n, 1)$ and $\mathbf{z} = \text{zeros}(n, 1)$) which were attempted with the AMG method. However, changing the initial guess for $\mathbf{z}$ to $\mathbf{z} = 10^{11}\text{ones}(n, 1)$ did not lead to converging behavior either, suggesting a different cause of the numerical issue.

5. Conclusion

The theory presented in this work shows that the calculation of the fundamental matrix corresponding to a Markov chain can be replaced by solving a set of linear systems and some matrix-vector and vector-vector multiplications. Although the experimental results show numerical issues

with one of the linear systems, changes to the implemented method or substituting for another solving method may yield positive results.

5.1. Future Work and Applications

The goal of this work is to develop a method to calculate the DSD distance metric when finding the fundamental matrix is unfeasible. The PPI networks presented here are relatively small, containing thousands or tens of thousands of vertices. However, this same distance metric can prove useful in identifying similar entities in other much larger graphs which excibit similar properties as the PPI networks, namely networks that contain hubs which link otherwise unrelated entities to each other over potentially short paths. Examples of such networks include large social networks or a network representing linked article on Wikipedia or linked webpages on the Internet. In 2011, the largest connected component of the Facebook social network was found to contain 99.91% of Facebook's 721 million active(signed in within 28 days of the analysis) users, with the number of edges $|E|$ in the entire graph equal to 68.7 billion, although the edges in the largest component were not specified [4]. A more recent statistic showed 1.65 billion active in March 2016 [1]. When undertaking analyses of such networks, a matrix inversion at $O(n^3)$ is prohibitive and an efficient algorithm such as that proposed by the theory in the project would prove even more useful.

References

[1] *Facebook: Company Info- Stats*, `http://newsroom.fb.com/Company-Info/`, Accessed: 2016-05-05.

[2] Mengfei Cao, Hao Zhang, Jisoo Park, Noah M. Daniels, Mark E. Crovella, Lenore J. Cowen, and Benjamin Hescott, *Going the distance for protein function prediction: A new distance metric for protein interaction networks*, PLoS ONE **8** (2013), no. 10, 1–12.

[3] Eric Jones, Travis Oliphant, Pearu Peterson, et al., *SciPy: Open source scientific tools for Python*, 2001–, [Online; accessed 2016-05-05].

[4] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow, *The anatomy of the facebook social graph*, CoRR **abs/1111.4503** (2011).

[5] Alexei Vazquez, Alessandro Flammini, Amos Maritan, and Alessandro Vespignani, *Global protein function prediction from protein-protein interaction networks*, Nature biotechnology **21** (2003), no. 6, 697–700 (English), Copyright - Copyright Nature Publishing Group Jun 2003; Last updated - 2013-02-06.

A. Appendix: Code

To run code:
need python 3 and the packages networkx, numpy, scipy, argparse, and timeit
to execute:

```
> python dsdAMG.py -f [ppi file name without the .ppi extension]
```

for example

```
> python dsdAMG.py -f worm
```

```
1
2  import networkx as nx
3  import argparse
4  import numpy as np
5  import scipy.sparse
6  import dsd_solveAMG
7
8  ##########################################################################################################################
9  ## The following code is adapted from code authored by Ben Hescott:
10 ##
11 #Precondition: adj is a NetworkX adjacency matrix of a connected undirected graph
12 #Postcondition: Returns a NetworkX matrix of transintion probabilties for
13 #               a random walk in the graph represented by adjacency.
14 def createTransitionMatrix(adj):
15     # number of nodes
16     n      = np.size(adj[0])
17     # initialize the matrices
18     p      = np.zeros((n, n))
19     degree = np.zeros((n, 1))
20     #for every node calculate the transition probability
21     for j in range(n):
22         degree[j] = sum(adj[j])
23         # compute the transition matrix of the markov chain
24         if degree[j] != 0:
25             p[j] = adj[j]/degree[j]
26     return p, degree
27
28
29 # Takes in a networkx graph, 'graph', and a list of nodes in the graph
30 # and returns the adjacency matrix of the graph with the ordering
31 # in 'nodelist' as a numpy array
32 def createAdjacencyMatrix(graph, nodelist):
33     return np.array(nx.adjacency_matrix(graph, nodelist).todense())
34
35 # Return the canonical node ordering, which is the nodes of the graph
36 # in sorted order
37 def getNodeOrdering(graph):
38     return sorted(graph.nodes())
39 ##########################################################################################################################
40 def basis_uv(u,v, n):
41
42     #b_uv = scipy.sparse.coo_matrix(([1,-1], (u, v)), shape=(n, 1))
43
44     b_uv = np.zeros((n,1))
```

```
45      b_uv[u,0] = 1
46      b_uv[v,0] = -1
47      return b_uv
48
49
50  def createMatrix(G):
51      nodeList  = getNodeOrdering(G)
52      adj = createAdjacencyMatrix(G, nodeList)
53
54      # number of nodes
55      n = np.size(adj[0])
56      _, degree = createTransitionMatrix(adj)
57
58      # create w using the fact that the
59      # steady state of an undirected random walk
60      # is proportional to node degree
61      pi = (degree)/sum(degree)
62
63      # create degree matrix with correct ordering
64      D = scipy.sparse.dia_matrix((degree.T, [0]), shape=(n,n))
65
66      L = D - scipy.sparse.csc_matrix(adj)
67      return L, pi, D
68
69  def main():
70      parser = argparse.ArgumentParser()
71      parser.add_argument("-f", required=True, help="PPI file")
72      args = parser.parse_args()
73      ppi = args.f
74      # ppi = 'worm'
75      ppi_file = '../../data/' + ppi + '.ppi'
76
77      # Create graph and needed matrices and vectors
78      G = nx.read_edgelist(ppi_file, nodetype=str)
79      L, pi, D = createMatrix(G)
80      n = len(pi)
81      (u,v) = (0,1)
82      b_uv = basis_uv(u,v,n)
83
84      print('AMG Solve for ', ppi)
85      for cycle in ['W']:
86          print('AMG Cycle type', cycle)
87          # Set up AMG levels and parameters
88          # only needs to be done once for solving for linear systems,
89          # unless different cycle_types are necessary
90          amgData, amgParam = dsd_solveAMG.setup_AMG(L,cycle_type=cycle)
91
92          # Solve the two linear systems Ly_{uv}=b_{uv} and Lz=pi
93          # z = dsd_solveAMG.solve_AMG(amgData, pi, amgParam)
94          y_uv = dsd_solveAMG.solve_AMG(amgData, b_uv, amgParam)
95
96          # Compute the vertices for finding fundamental matrix (I - P + W)
97          # f = D.dot(y_uv)
98          #g = D.dot(z)
99          #x_uv = f - ((np.ones(n).dot(f))/(1 + np.ones(n).dot(g))).dot(g)
100
```

```
        # Compute DSD
        #dsd_uv = np.linalg.norm(x_uv)
if __name__ == "__main__":
    main()
```

```
import numpy as np
from AMG_Setup import *
from AMG_Solve import *


# solve graph Laplacian using AMG
#
# adapted from matlab code from @ Xiaozhe Hu, Tufts University


def setup_AMG(L, cycle_type='V'):
    """
    Sets up levels and parameters for solving Lx=b using cycle-type specified

    Parameters
    ----------
    L: Graph Laplacian
    cycle_type: AMG cycle type, either 'TL' for two-level, 'V' for V-cycle (default), or 'W' for W-cycle

    Returns
    -------
    amgData
    amgParams
    """

    #---------------------
    # AMG parameters
    #---------------------
    amgParam = {}
    amgParam.update({'print_level': 1})  # how much information to print when using AMG solve only
                                         # 0: print nothing | positive number print information
    # setup phase parameters
    amgParam.update({'max_level': 20})  # maximal number of level in AMG
    amgParam.update({'coarsest_size': 100})  # size of the coarest level

    # solve pahse parameters
    amgParam.update({'cycle_type':cycle_type})  # ''TL: Two-level | 'V': V-cycle | 'W': W-cycle
    amgParam.update({'n_presmooth': 1})  # number of presmoothing
    amgParam.update({'n_postsmooth': 1})  # number of postsmoothing

    amgParam.update({'max_it': 100})  # when AMG is used as standalone solver, maximal number of iterations that is
    amgParam.update({'tol': 1e-8})    # when AMG is used as standalone solver, tolerance for the reletive residual

    #---------------------
    # setup phase
    #---------------------
    amgData = AMG_Setup(L, amgParam)

    return amgData, amgParam
```

```
50
51  def solve_AMG(amgData, b, amgParam):
52      """
53      Solve Lx=b through AMG
54
55      Parameters
56      ----------
57      amgData: AMG data produced through AMG_Setup
58      b: Right-hand side
59      amgParam: AMG parameters produced through AMG_Setup
60      cycle_type: AMG cycle type, either 'TL' for two-level, 'V' for V-cycle (default), or 'W' for W-cycle
61
62      Returns
63      -------
64      x: solution to Lx=b
65      """
66      n = b.shape[0]
67      x = np.zeros((n,1)) #initial guess
68      (x, k, err) = AMG_Solve(amgData, b, x, amgParam)
69
70      return x
```

```
1   import numpy as np
2   import scipy.sparse
3   from timeit import default_timer as timer
4   import support_scripts, form_aggregates
5
6   def AMG_Setup(Lf, amgParam):
7       # Setup phase for AMG method
8       #
9       # adapted from Matlab code by@ Xiaozhe Hu, Tufts University
10
11      #----------------
12      # local variable
13      #----------------
14      print_level = amgParam['print_level']
15      max_level = amgParam['max_level']
16
17      if amgParam['cycle_type']=='TL':
18          max_level =2
19
20      coarsest_size = amgParam['coarsest_size']
21
22      level = 0
23
24      #----------------
25      # AMG information
26      #----------------
27      AMG_Data={level:{} for level in range(max_level)}
28
29      #----------------
30      # finest level
31      #----------------
32      AMG_Data[0].update({'L':Lf,
```

```
                        'N':Lf.shape[0],
                        'DL': scipy.sparse.tril(Lf, format='csr'),
                        'DU': scipy.sparse.triu(Lf, format='csr'),
                        'D': Lf.diagonal(),
                        'max_level': 0})
    #----------------
    # main loop
    #----------------
    print('-------------------------------------------------')
    print('              Calling AMG setup    ')
    print('-------------------------------------------------')

    setup_start = timer()

    while (level < max_level-1) and (AMG_Data[level]['N'] > coarsest_size):

        #----------------
        # form aggregation
        #----------------
        # implement your own aggregation algorithm
        # input: L{level} -- graph Laplacian on current level
        # output: aggregation -- information about aggregates
        #                                          ( aggregation(i) = j mean the i-th vertex belong to aggregates j
        #              num_agg -- number of aggregations
        #
        (aggregation, num_agg) = form_aggregates.form_aggregates(AMG_Data[level]['L'])


        #----------------
        # generate prolongation
        #----------------
        AMG_Data[level]['P'] = support_scripts.generate_unsmoothed_P(aggregation, num_agg)

        #----------------
        # generate restriction
        #----------------
        AMG_Data[level]['R'] = AMG_Data[level]['P'].transpose()

        #----------------
        # compute coarse grid matrix
        #----------------
        AMG_Data[level+1]['L'] = AMG_Data[level]['R'].dot(AMG_Data[level]['L']).dot(AMG_Data[level]['P'])

        AMG_Data[level+1]['N'] = AMG_Data[level+1]['L'].shape[0]

        #----------------
        # extra information for smoothers
        #----------------
        AMG_Data[level+1]['DL'] = scipy.sparse.tril(AMG_Data[level+1]['L'], format='csr')
        AMG_Data[level+1]['DU'] = scipy.sparse.tril(AMG_Data[level+1]['L'], format='csr')
        AMG_Data[level+1]['D'] = AMG_Data[level+1]['L'].diagonal()

        #---------------
        # update
        #---------------
        level += 1
```

```
89
90
91      setup_duration = timer() - setup_start
92
93      # construct the data structure
94      for l in range(level+1):
95          AMG_Data[l]['max_level'] = level
96
97
98      # print information
99      if print_level > 0:
100
101         total_N = 0
102         total_NNZ = 0
103
104         print('-------------------------------------------------')
105         print(' # Level\t|\t# Row\t|\t# Nonzero\t|\tAvg. NNZ/Row\t|')
106         print('-------------------------------------------------')
107
108         for i in range(level+1):
109                 nonzero_i = len(AMG_Data[i]['L'].nonzero()[0])
110                 N_i = AMG_Data[i]['N']
111                 total_N += N_i
112                 total_NNZ += nonzero_i
113
114                 print('\t%2d\t|\t%9d\t|\t%10d\t|\t%7.3f\t|' % (i, N_i, nonzero_i, nonzero_i/N_i))
115
116
117         print('-------------------------------------------------')
118         print(' Grid complexity: %0.3f | Operator complexity: %0.3f ' % (total_N/AMG_Data[0]['N'], total_NNZ/len(AM
119         print('-------------------------------------------------')
120
121     # print cputime
122     print('-------------------------------------------------')
123     print('        AMG setup costs', setup_duration, 'seconds')
124     print('-------------------------------------------------')
125
126
127     return AMG_Data
128
```

```
1   import numpy as np
2   from timeit import default_timer as timer
3   import support_scripts
4   from AMG_Cycle import *
5
6   def AMG_Solve(amgData, b, x, amgParam):
7       # Solve phase for AMG method
8       #
9       # adapted from Matlab code by@ Xiaozhe Hu, Tufts University
10
11      # parameters
12      print_level = amgParam['print_level']
13      max_it = amgParam['max_it']
```

```
    tol = amgParam['tol']

    # prepare solve
    level = 0
    err = np.zeros((max_it+1,1))

    r = b - amgData[0]['L'].dot(x)
    err[0] = np.linalg.norm(r)

    # print
    print('-------------------------------------------------')
    print('              Calling AMG solver    ')
    print('-------------------------------------------------')

    if print_level > 0:
        print('-------------------------------------------------')
        print(' # It |  ||r||/||r0||  |     ||r||      |   Rate.  |')
        print('-------------------------------------------------')
        print(' %4d |  %e  |  %e  | %f |' % (0, 1.0, err[0], 0.0))


    # main loop
    solve_start = timer()

    for k in range(max_it):

        # call multigrid
        x = AMG_Cycle(amgData, b, x, level, amgParam)

        # compute residual
        r = b - amgData[level]['L'].dot(x)

        # compute error
        err[k+1] = np.linalg.norm(r)

        # display
        if print_level > 0:
            print(' %4d |  %e  |  %e  | %f |' % (k+1, err[k+1]/err[0], err[k+1], err[k+1]/err[k]))


        if (err[k+1]/err[0]) < tol:
            break


    solve_duration = timer() - solve_start

    # cut err
    err = err[:k+1]

    # print
    print('-------------------------------------------------')
    if k == max_it:
        print('        AMG reached maximal number of iterations ')
    else:
        print('        AMG converged or reached max iterations')
        print('        Number of iterations =', k+1)
```

```
70
71      print('        Relative residual =', err[-1]/err[0])
72      print('-----------------------------------------------------')
73      print('        AMG solve costs', solve_duration, 'seconds')
74      print('-----------------------------------------------------')
75
76
77
78
79      return x, k, err
```

```
1   import numpy as np
2   import scipy.sparse, scipy.sparse.linalg
3   import support_scripts
4
5   def AMG_Cycle(amgData, b, x, level, amgParam):
6       # Multigid cycle
7       # adapted from Matlab code of @ Xiaozhe Hu, Tufts University
8       # Clara De Paolis
9
10      # parameters
11
12      max_level = amgData[0]['max_level']
13
14      n_presmooth = amgParam['n_presmooth']
15      n_postsmooth = amgParam['n_postsmooth']
16      cycle_type = amgParam['cycle_type']
17
18      # coarsest level
19      if level == max_level:
20         x = scipy.sparse.linalg.spsolve(
21             (amgData[level]['L'] + 1.0e-12* scipy.sparse.eye(len(b), len(b))), b).reshape((len(x),1))
22
23      else:
24          # presmoothing
25          x = support_scripts.forward_gs(amgData[level]['L'], b, x, amgData[level]['DL'], n_presmooth)
26
27          # compute residual
28          r = b - amgData[level]['L'].dot(x)
29
30          # restriction
31          r_c = amgData[level]['R'].dot(r)
32
33          # coarse grid correction
34          e_c = np.zeros((amgData[level+1]['L'].shape[0],1))
35
36          if cycle_type=='TL':
37              # coarse grid correction for two-level method here
38              e_c = AMG_Cycle(amgData, r_c, e_c, level+1, amgParam)
39          elif cycle_type=='V':
40              # coarse grid correction for V-cycle here
41              e_c = AMG_Cycle(amgData, r_c, e_c, level+1, amgParam)
42          elif cycle_type=='W':
43              # coarse grid correction for W-cycle here
```

```
            for k in range(2):
                e_c = AMG_Cycle(amgData, r_c, e_c, level+1, amgParam)


        # prolongation
        x = x + amgData[level]['P'].dot(e_c)

        # postsmoothing
        x = support_scripts.backward_gs(amgData[level]['L'], b, x, amgData[level]['DU'], n_postsmooth)

    return x
```

```
import numpy as np
import scipy.sparse, scipy.sparse.linalg

def assembleGraphLaplace(N):
  # Adapted from matlab code
  # Copyright (C)  Xiaozhe Hu.

  e = np.ones(N)
  NN = N**2

  L1d = scipy.sparse.spdiags([-1*e, 2*e, -1*e], [-1,0,1], N, N)
  I = scipy.sparse.eye(N,N)

  L =  scipy.sparse.kron(L1d, I) + scipy.sparse.kron(I, L1d)
  L = L - scipy.sparse.spdiags(L.diagonal(), 0, NN, NN)
  L = L + scipy.sparse.spdiags(-L.sum(axis=1).T, 0, NN, NN) #row sum


  return L

def backward_gs(A, b, x, DU, nsmooth):
  # Backward Gauss-Seidel smoother
  # Adapted from matlab code from @ Xiaozhe Hu, Tufts University

  #----------------------------
  # Step 1: Main loop
  #----------------------------
  for i in range(nsmooth):
      # GS iteration
      x += scipy.sparse.linalg.spsolve(DU, (b - A.dot(x))).reshape((len(x),1))
  return x

def forward_gs(A, b, x, DL, nsmooth):
    # Forward Gauss-Seidel smoother
    # Adapted from matlab code from
    # @ Xiaozhe Hu, Tufts University

    #----------------------------
    # Step 1: Main loop
    #----------------------------
    for i in range(nsmooth):
        # GS iteration
```

```
        x += scipy.sparse.linalg.spsolve(DL, (b - A.dot(x))).reshape((len(x),1))
    return x

def generate_unsmoothed_P(aggregation, num_agg):
    # Construct unsmoothed prolongation P
    # Adapted from matlab code from
    # @ Xiaozhe Hu, Tufts University

    n = len(aggregation)
    p = scipy.sparse.csr_matrix((np.ones(n), (np.array(range(n)), aggregation)), shape=(n, num_agg))

    return p
```

```
import numpy as np
import scipy.sparse

def form_aggregates(L):

    # Heavy edge Coarsening
    n = L.shape[0]
    count = -1
    aggregates = np.zeros(n)
    for i in range(n):
        if aggregates[i]==0:
            # pick j for edge with max weight
            (_, js, w) = scipy.sparse.find(-L[i]) #find edges and weights
            e = np.where(w == max(w))[0]  # e lists the indices that match the max weight
            j = js[e[-1]]

            if aggregates[j] == 0:
                count += 1
                aggregates[i] = int(count)
                aggregates[j] = int(count)

            else:
                aggregates[i] = aggregates[j]

    num_agg = count+1

    return aggregates, num_agg
```