Modeling and Projection of Respondent Driven Network Samples

by

Zhihe Zhuang

B.S., Tianjin University of Finance and Economics, 2016

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Statistics
College of Arts and Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2018

Approved by:

Major Professor
Dr. Perla E. Reyes Cuellar

Copyright

Abstract

The term network has become part of our everyday vocabulary. The more popular are perhaps the social ones, but the concept also includes business partnerships, literature citations, biological networks, among others. Formally, networks are defined as sets of items and their connections. Often modeled as the mathematic object known as a graph, networks have been studied extensively for several years, and research is widely available. In statistics, a variety of modeling techniques and statistical terms have been developed to analyze them and predict individual behaviors. Specifically, certain statistics like degree distribution, clustering coefficient, and so on are considered important indicators in traditional social network studies. However, while conventional network models assume that the whole network population is known, complete information is not always available. Thus, different sampling methods are often required when the population data is inaccessible. Less time has been dedicated to studying the accuracy of these sampling methods to produce a representative sample. As such, the aim of this report is to identify the capacity of sampling techniques to reflect the features of the original network. In particular, we study Anti-cluster Respondent Driven Sampling (AC-RDS). We also explore whether standard modeling techniques paired with sample data could estimate statistics often used in the study of social networks.

Respondent Driven Sampling (RDS) is a chain referral approach to study rare and/or hidden populations. Originating from the link-tracing design, RDS has been further developed into a series of methods utilized in social network studies, such as locating target populations or estimating the number and proportion of needle-sharing among drug addicts. However, RDS does not always perform as well as expected. When the social network contains tight communities (or clusters) with few connections between them, traditional RDS tends to oversample one community, introducing bias. AC-RDS is a special Markov chain process that collects samples across communities, capturing the whole network. With special

referral requests, the initial seeds are more likely to refer to the individuals that are outside their communities. In this report, we fitted the Exponential Random Graph Model (ERGM) and a Stochastic Block Model (SBM) to an empirical study of the Facebook friendship network of 1034 participants. Then, given our goal of identifying techniques that will produce a representative sample, we decided to compare two version of AC-RDSs, in addition to traditional RDS, with Simple Random Sampling (SRS). We compared the methods by drawing 100 network samples using each sampling technique, then fitting an SBM to each sample network we used the results to project the network into one of population size. We calculated essential network statistics, such as degree distribution, of each sampling method and then compared the result to the original network observed statistics.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge my major professor Dr. Perla Reyes. Without her help with great patience, i would have never be able to accomplish my Master's report and defense. Throughout the whole second year of Master program, she consistently inspired me with statistical knowledge. Not only she taught me everything she knew as much as she could, she also motivated me to study in the area of interest and work independently. Also, I would like to appreciate the help from my committee member: Dr. Michael Higgins and Dr. Gyuhyeong Goh. They gave me great advise and helped me understand my report even better. They were not only my teacher in the class and office, they were also great friend I could talk to and hang out with.

I would also like to thank my family for giving me spiritual and financial support and being always there for me whenever I needed them. Finally, I would like to thank the heads of Statistics department in KSU for hiring me as a graduate teaching assistant. With this opportunity, I was able to be financially independent. Also, I learned how to inspire people with statistical knowledge and ensure my future career path.

Dedication

To my girlfriend Kathlyn, for the "Peter Rabbit" movie, the Monday "surprise" gifts, and for the endless support.

Chapter 1

Introduction

Networks (also known as graphs) are easily seen in everyday life: a friendship network, the Internet, companies or organizations network, human's neural networks, metabolic networks, books or papers citation networks. With the growing popularity of social networks many social media websites, such as Facebook, LinkedIn or Twitter, have gained thousands of millions of users. The topology of these social networks helps users promote connections and exchange of information. It has also become an attractive way to reach target populations, for uses as diverse as the people engaged in social media, from marketing companies trying to sell their products to government and health officials looking to estimate the size of populations at risk (Handcock et al., 2015) or to understand how information sharing in online communities may affect health behaviors (Balatsoukas et al., 2015). We will define network as a set of items and their connections. The interconnected items are represented by mathematical abstractions called vertices (or nodes), and the links that connect pairs of vertices are called edges. On one hand, if edges point in a certain direction such as the relationships in a prey-predator network, it is called directed network. On the other hand, when all the edges are bidirectional or when the direction is of no interest, it is called undirected network. Moreover, one network may contain both directed and undirected edges.

There exists a long list of methods and approaches that can be taken to model, analyze and conduct inference for network data, two of the them are introduced in Chapter 2.

However, for networks with billions of nodes and edges, computation and inference might not be achieved within a reasonable amount of time and money. When data is too massive to be processed thoroughly, a sampling approach seems a natural choice. In addition, for many cases as pointed out in Shalizi and Rinaldo (2013), data is only available for a sampled sub-network. The increase of available data and the need to analyze it have resulted in the proliferation of models for network data (Easley and Kleinberg, 2010; Goldenberg et al., 2010; Newman, 2010). Typically, however, models are developed assuming we have access to the entire network and conclusions based on a sampled sub-network are generalized to the whole network/population. Shalizi and Rinaldo (2013) showed that the assumption of consistency under sampling that is required to make such generalizations is violated by many popular models. They discussed how the class of exponential random graph models (ERGM) and other similar models require strong assumptions to be able to use sampled data to draw conclusions about the population network. One key issue that Shalizi and Rinaldo (2013) left unanswered is how to obtain information from a part of the network and then draw conclusions about the whole population.

In the case of sampling methodologies, we can go as far back as Goodman (1961) snowball sampling, which evolved into response driven sampling (RDS). RDS is still the most used way to investigate and draw conclusions about hard to find populations. More recent examples of efforts into sampling networks include Blagus et al. (2015) and Rezvanian and Meybodi (2015). Less time has been dedicated to studying the accuracy of these sampling methods to produce a representative sample. Zhou (2015) explored how simple random and stratified sampling methods paired with modeling techniques can potentially reproduce a whole network using a simulated high school student friendship network, and showing that, among the approaches been tested, stratified sample with an stochastic blockmodel approach will produce the best results in terms of network metrics.

The main objective of this report is to identify the capacity of sampling techniques to reflect the features of the original network. In particular, we study RDS and Anti-cluster Respondent Driven Sampling (AC-RDS). We also explore whether standard modeling techniques paired with sample data could estimate statistics often used in the study of social

networks. Thus, we extend Zhou (2015) in two directions, 1) the methodologies were tested in a small simulated network (104 nodes), here we use a real friendship network of 1034 nodes that being still small can give us some insight into the scalability of the methods; and 2) we use RDS and AC-RDS, sampling methods that are often associated with networks.

In the following section, we will introduce the friendship network being used as a case study. We will define network, network statistics of interest and traditional network models in Chapter 2. We describe ERGM in detail in Section 2.2. Section 2.3 will cover stochastic blockmodels an alternative Bayesian method to model networks that we expect has the potential of producing better results under sampling. Chapter 3 describes the sampling techniques and inference approaches that we explored to assess whether network information can be estimated using only a sub-network. Finally, Chapter 4 discusses our findings and directions for future work.

1.1 Case Study: Facebook Friendship Ego-network

For illustration purposes, we will use data extracted from Facebook. In particular, one of the undirected facebook friendship networks originally used for Stanford Network Analysis Platform (SNAP) (Leskovec and Krevl, 2014), which is a platform for social and information network analysis based on Python. The Facebook network data was collected from survey participants using this Facebook app. (Leskovec and Mcauley, 2012) This is a friendship ego-network since edges are captured by Facebook friend list, which means each individual on the participant's friend list is connected to this participant. 10 sample ego-networks were collected in this survey. Since no edge across ego-network is recorded, we assume there is no connection between each ego-network. Being one of our objective test the scalability of the methods, we simply choose the largest of ten ego-networks, where 53498 undirected edges are built among 1034 nodes.

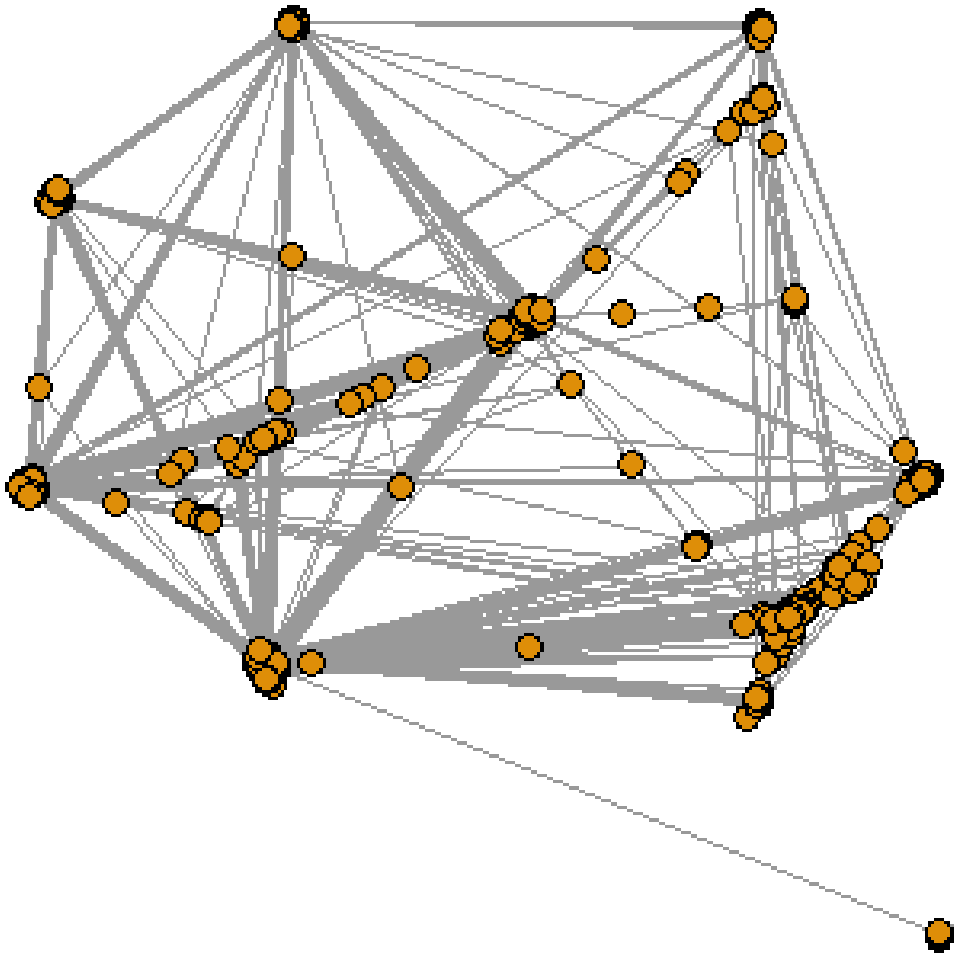

Figure 1.1: The Facebook graph. Each dot represents a subject (node) and each line denotes the friendship connection (edge) between subjects on each side of the line.

Chapter 2

Network Models for Statistical Inference

2.1 Network Properties

Rapoport (1977) was one of the first theorists that found the common properties of networks and modeled them mathematically. He studied the degree distribution in all kinds of networks using random graphs, the simplest model of a network. A random graph is a graph in which properties such as the number of graph vertices, graph edges and connections between them are determined in some random way, for instance, edge probabilities between two vertices can distribute uniformly in the (0,1) interval. We will give a more detailed mathematical definition of random graph in Section 2.2. In this Section, we will discuss some important network properties defined in Newman (2003).

2.1.1 Mean Shortest Distance between Vertex Pairs

Consider an undirected network with a fixed n number of vertices, there are $\frac{1}{2}n(n+1)$ possible edges in this network. If we treat the distance from each vertex to itself as zero, then the

mean shortest distance between pairs is defined as:

$$\ell = \frac{1}{\frac{1}{2}n(n+1)} \sum_{i \leq j} d_{ij}$$

where d_{ij} is the shortest distance from vertex i to vertex j, and n is the total number of vertices in the network. In this report, for distance we mean that the geodesic distance between vertex i and vertex j. It is the shortest path (in the number of edges) through the network from one vertex to another. One problem with the quantity ℓ arises when networks have more than one component. In graph theory, a component of a network is a subgraph that can be reached from a vertex by paths running along edges of the graph. Therefore when there is more than one component, we could have vertex pairs with no connecting path. If one assigns infinite shortest distance d_{ij} to such pairs, then the value of ℓ becomes infinite. A way to avoid this kind of problem is to define ℓ to be the harmonic mean shortest distance between all pairs, i.e., the reciprocal of the average of the reciprocals:

$$\ell^{-1} = \frac{1}{\frac{1}{2}n(n+1)} \sum_{i \leq j} d_{ij}^{-1} \tag{2.1}$$

In equation 2.1, infinite values of d_{ij} contribute nothing to the sum and thus, nothing to the quantity ℓ, either.

2.1.2 Clustering Coefficient

Different from the random behavior of a random graph, network clustering is a commonly seen property. In many such networks, it is found that if vertex A is connected to vertex B and vertex B is connected to vertex C, then there is a higher probability that vertex A is also connected to vertex C. For instance, in a social network, it simply means that the friend of your friend is also likely to be your friend. To interpret it mathematically, clustering means to measure the number of triangles - set of three connected vertices forming a triangle. The

clustering coefficient C is defined as:

$$C = \frac{3 \times \textit{number of triangles in the network}}{\textit{number of connected triples of vertices}}$$

where a connected triple means we have one single vertex with edges going to the other two vertices. Similarly, the clustering coefficient of a node is the number of triangles that pass through this vertex, relative to the maximum number of 3-vertex-loops that could pass through the node. It is always a number between 0 and 1.

Alternatively, we can define a clustering coefficient for specific vertex i as:

$$C_i = \frac{\textit{number of triangles to vertex } i}{\textit{number of triples centered on vertex } i} \tag{2.2}$$

If both numerator and denominator are zero, we put $C_i = 0$. Then, the average of the clustering coefficients for all vertices in the network is:

$$C = \frac{1}{n_i} \sum_i C_i \tag{2.3}$$

Normally equation 2.2 is easier to calculate by hand. However, equation 2.3 is easier to obtain for a computer and is widely used in data analysis. Generally speaking, no matter which formula is used, the clustering coefficient measures the density of triangles in a network. Also the coefficient of a real world network tends to be higher than that of a random graph with similar number vertices and edges.

2.1.3 Degree Distribution

A vertex in an undirected network has degree k if the number of edges connected to that vertex is k. The vertex degree distribution of an undirected network gives the number (or fraction in some formulas) of vertices with degree k for $k = 0, 1, \ldots$. In a directed network, the in-degree of a vertex k is the number of incoming edges and the out-degree is the number of outgoing edges. In this report, we will focus on the undirected network case.

Usually p_k is defined to be the fraction of vertices with degree k. It is interpreted as the probability that a randomly chosen vertex has degree k. A plot of p_k for a network can be obtained by drawing a histogram of the vertex degrees. This histogram is the degree distribution of the network. A popular formula to describe the degree distribution is to use the plot of the cumulative distribution function:

$$P_k = \sum_{k'=k}^{\infty} p_{k'}$$

which is the probability that the degree is greater than or equal to k. For example, in a random graph defined by Erdos and Rényi (1960), each edge is either presented or not with constant probability 0.5. Therefore the degree distribution of that random graph is Binomial, or Poisson in the limit of large graph size. Real world networks are hardly random, hence it is unlikely for us to find its degree distribution strictly following Binomial or Poisson distributions. They are highly right-skewed most of time. Many degree distributions follow power law in their tails, i.e., $p_k \sim k^{-a}$ for some constant exponent a. Networks with power law degree distributions has been studied extensively in literature. They are sometimes referred to as scale-free networks.

2.1.4 Community Structure

A network is said to have community structure if the vertices of the network can be easily grouped into sets of vertices such that each set of vertices is densely connected internally. Networks representing social interactions tend to show this kind of community structure, where groups of vertices (i.e., groups of people) have a higher density of edges within them than between them. People may be dividing based on easy-to-measure characteristics such as age, gender, company, political preferences and so on, or due to some other harder to observe feature like preferences or personality. Identifying those community structures in a network would provide useful insights into the process driving the network. The traditional method is called cluster analysis or sometimes called hierarchical clustering. This method

requires defining a similarity measurement between any two vertices and then grouping similar vertices into communities according to this measurement. In social network literature, the so-called block models are basically divisions of networks into communities or blocks based on some criterion. Two vertices are said to be structurally equivalent if two vertices have the same neighbors. However, exactly the same structural equivalence is hard to find, but approximate equivalence is often used for doing hierarchical clustering.

2.2 Exponential Random Graphs Models

Strauss (1986) considers exponential random graph, also called p^* models. Exponential random graph models (ERGM) represent a general class of models based on exponential theory for specifying the probability distribution underlying a set of random graphs or networks. Instead of modeling the edges, ERGM treats the whole graph as a random variable $\mathbf{Y}$ and defines a probability model for $P(\mathbf{Y} = \mathbf{y})$, which is defined in equation 2.4. The support of $\mathbf{Y}$ is the space with all possible graphs among n nodes. Within this framework, one can obtain maximum-likelihood estimates for the parameters of a specified model for a given network data; simulate additional networks with the underlying probability distribution implied by that model and perform various types of model comparison. The basic expression for the ERGM class can be written as:

$$P(\mathbf{Y} = \mathbf{y}) = \frac{\exp(\boldsymbol{\theta}' g(\mathbf{y}))}{k(\boldsymbol{\theta})}, \tag{2.4}$$

where $\mathbf{Y}$ is the random variable for the state of the network (with realization $\mathbf{y}$), $g(\mathbf{y})$ is the vector of model statistics for network $\mathbf{y}$. $\boldsymbol{\theta}$ is the vector of coefficients for those statistics, and $k(\boldsymbol{\theta})$ represents the quantity in the numerator summed over all possible networks (typically constrained to be all networks with the same node set as $\mathbf{y}$). This can be rewritten in terms of the log-odds of a single actor pair given the rest:

$$logit(Y_{ij} = 1 | \mathbf{y}_{ij}^c) = \theta' \delta(y_{ij}),$$

where Y_{ij} is the random variable for the state of the actor pair i, j (with realization y_{ij}), which means the presence or absence of an edge between vertex i and vertex j. $\mathbf{y}_{ij}^c$ denotes the complement of y_{ij}, i.e., all dyads in the network except for y_{ij}. That means all the random variables associated with potential pairs in the network except for y_{ij}. The variable $\delta(y_{ij})$ equals $g(y_{ij}^+) - g(y_{ij}^-)$, where y_{ij}^+ is defined as y_{ij}^c along with y_{ij} set to 1, and y_{ij}^- is defined as y_{ij}^c along with y_{ij} set to 0.

In other words, $\delta(y_{ij})$ equals the value of $g(\mathbf{y})$ when $y_{ij} = 1$ minus the value of $g(\mathbf{y})$ when $y_{ij} = 0$, but all other dyads are as in $g(\mathbf{y})$. This emphasizes the log-odds of an individual tie conditional on all others. $g(\mathbf{y})$ is known as the statistics of the model, and $\delta(y_{ij})$ as the "change statistics" for actor pair y_{ij}.

Here, we will consider only the simplest possible model, the Bernoulli or Erdös-Rényi model, which contains only an edge term and therefore is estimated by a essential log-linear model. When covariate information about nodes, also known as attributes, is available a linear function $\mathbf{X}\boldsymbol{\beta}$ can be included in $g(\mathbf{y})$.

2.2.1 Case Study ERGM

The ERGM package in `R` the user to fit exponential-family random graph (ERG) models to network datasets. We fitted an ERGM to the Facebook network, and see how the probability of connection between actor i and actor j are determined in this model. The fitted model is:

$$logit(\hat{Y}_{i,j}) = -2.43827 edges + 1.55330 degree(30) - 1.70358 degree(50) + 30.81076 degree(100)$$

where *edges* is equal to y_{ij}, i.e. it is equal to one whenever $y_{ij} = 1$, zero otherwise; $degree(k)$ is an indicator of whether $y_{ij} = 1$ which will increase the number of vertices with degree k.

Table 2.1: ERGM Estimates and Standard Errors

	Estimate	Std.Error
edges	-2.43827	0.05279
degree(30)	1.55330	2.58835
degree(50)	-1.70358	0.68759
degree(100)	30.81076	0.63444

- Edges accounts for connections between nodes.
- Degree(k) is a binary variable that indicates whether or not each node of the actor pair i, j will have degree k.

2.3 Stochastic Blockmodels

Stochastic blockmodels (Wang and Wong, 1987; White et al., 1976) is an approach in statistics and network analysis that extends the notion of model-based clustering to networks. The idea is to separate a graph in classes of actors with common properties. Actors are usually grouped in classes of equivalence. In terms of equivalence, we mean that pairwise relationships within the group exhibit similar structure, while relationships between different groups exhibit different structures. Revealing those hidden structures of a network is the heart of many data analysis problems. In this report, we will build a hierarchical Bayesian model for an exchangeable network to identify grouping patterns, in specific, a type of stochastic blockmodels. Given a network, the goal in stochastic blockmodels is to divide the vertices such that pairs of vertices are grouped together if their connecting pattern to the other groups in the network is similar.

In the case of acyclic (nodes cannot connect to themselves), directed networks, Bayesian stochastic blockmodels are hierarchical models that for $i, j = 1, \ldots, I$ and $j \neq i$.

$$y_{ij}|\xi_i, \xi_j, \boldsymbol{\Theta} \sim \psi(y_{ij}|\theta_{\xi_i,\xi_j}, \boldsymbol{\nu}) \quad \xi_i|\mathbf{w} \sim \sum_{k=1}^{K} w_k \delta_k \quad (\mathbf{w}, \boldsymbol{\Theta}, \boldsymbol{\nu}|\boldsymbol{\varsigma}, \boldsymbol{\lambda}) \sim p(\mathbf{w}|\boldsymbol{\varsigma}) p(\boldsymbol{\Theta}|\boldsymbol{\lambda}) p(\boldsymbol{\nu}) \quad (2.5)$$

where $\boldsymbol{\varsigma}$ and $\boldsymbol{\lambda}$ are vectors of hyperparameters (which will be assigned hyperpriors $p(\boldsymbol{\varsigma})$ and $p(\boldsymbol{\lambda})$), $\boldsymbol{\Theta} = \{\theta_{k,l}\}$ is a $N \times N$ matrix, $\mathbf{w} = (w_1, \ldots, w_N)'$ is such that $\sum_{k=1}^{N} w_k = 1$, δ_k denotes the degenerate probability distribution placing probability 1 on k, and $\psi(y|\theta, \boldsymbol{\nu})$ is

parametric kernel indexed by the parameters θ and $\boldsymbol{\nu}$, where θ is a random effect, and $\boldsymbol{\nu}$ is a vector of fixed effects. In the case of undirected networks, a similar definition applies with the added constrains $y_{ij} = y_{ji}$ and $\theta_{k,l} = \theta_{l,k}$.

The formulation in (2.5) is extremely flexible and easily interpretable. The latent variables $\boldsymbol{\xi} = (\xi_1, \ldots, \xi_I)'$ act as (unobserved) faction indicators; the prior probability that any two subjects are assigned to the same cluster is given by $\mathsf{E}\left\{\sum_{k=1}^{N} w_k^2\right\}$. Binary networks (i.e., those where $y_{ij} \in \{0, 1\}$, so that $y_{ij} = 1$ if actor i interacts with actor j, and $y_{ij} = 0$ otherwise) could be accommodated by taking $y_{ij}|\xi_i, \xi_j, \boldsymbol{\Theta} \sim Bernoulli(\theta_{\xi_i,\xi_j})$ and selecting (for computational convenience) $\theta_{k,l} \sim beta(a, b)$. In this case, the entries $\theta_{k,l}$ give the probability that an interaction occurs between actor from factions k and l. On the other hand, count data could be incorporated by taking $y_{ij}|\xi_i, \xi_j, \boldsymbol{\Theta} \sim Poisson(\theta_{\xi_i,\xi_j})$ and $\theta_{k,l} \sim \mathsf{G}(a, b)$. Now, $\theta_{k,l}$ is the intensity of the interaction between factions k and l.

One example of a stochastic blockmodel is the infinite relational blockmodel (IRM) (Kemp et al., 2006; Xu et al., 2012). In the IRM, $N = \infty$ and

$$\theta_{k,l} \sim_{iid} H^{\boldsymbol{\lambda}} \qquad w_k = v_k \prod_{s<k}(1 - v_s) \qquad v_k \sim_{iid} Beta(1, \eta) \qquad k, l = 1, 2, \ldots \tag{2.6}$$

where $H^{\boldsymbol{\lambda}}$ is a parametric distribution indexed by the hyperparmeter $\boldsymbol{\lambda}$. This structure for the weights, which is strongly connected to the stick-breaking construction of the Dirichlet process (Sethuraman, 1994), implies that the joint distribution of $\boldsymbol{\xi} = (\xi_1, \ldots, \xi_I)$ obtained *after* integrating out $\mathbf{w}$ can be described by a sequence of predictive distributions where $\xi_1 = 1$ and

$$\xi_i|\xi_{i-1}, \ldots, \xi_1 \sim \sum_{k=1}^{K^{i-1}} \frac{m_k^{i-1}}{\eta + i - 1}\delta_k + \frac{\eta}{\eta + i - 1}\delta_{K^{i-1}+1} \qquad 2 \leq i \leq I \tag{2.7}$$

where δ_a denotes the degenerate probability distribution on a, $K^{i-1} = \max_{j<i}\{\xi_j\}$ is the number of unique values among $\xi_1, \ldots, \xi_{i-1}$, $m_k^{i-1} = \sum_{j=1}^{i-1} \mathbf{1}_{(\xi_j = k)}$ is the number of indicators equal to k among $\xi_1, \ldots, \xi_{i-1}$, and $\eta > 0$ is a constant. This sequence of predictive distributions is sometimes called the Chinese restaurant process (CRP) (see Figure 2.1), and implies

that $\mathsf{Pr}(\xi_i = \xi_j) = \sum_{k=1}^{\infty} \mathsf{E}\{w_k\} = 1/(1+\eta)$ for all i and j.

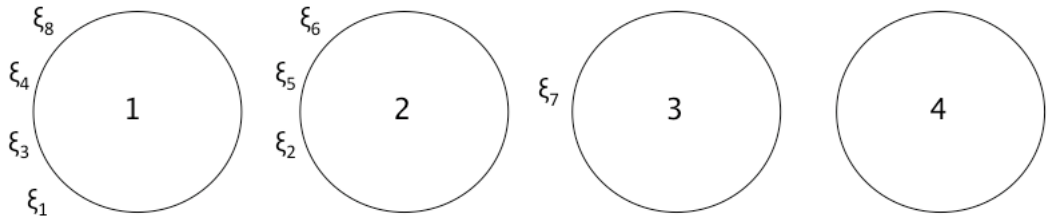

Figure 2.1: The Chinese restaurant process. Circles denote infinite number of tables and the letters around them are the customers sitting at that table.

Because the seating arrangement showed in Figure 2.1 can be described using the analogy of sitting customers at a Chinese Restaurant. ξ_i means the table is occupied by customer i. Customer 1 sits at table 1; customer i sits at any of the occupied tales with probability proportional to the number of customers sitting at that table, and sits at a new table with probability proportional to η. In Figure 2.1, the next customer (number 9) would sit on table 1 with probability $4/(\eta+8)$, on table 2 with probability $3/(\eta+8)$, on table 3 with probability $1/(\eta+8)$, or on table 4 with probability $\eta/(\eta+8)$

Based on Bayes' theorem, the hierarchical priors described above, times the observed relationships between any two actors i and j, will determine the posterior probability of connection between pairs of actors.

Under the Bayesian Nonparametric framework, the Stochastic Blockmodel produces an estimation of the parameter distributions instead of just a point estimate. We chose a Bernoulli-Beta model for y_{ij}. The joint posterior distribution of all the parameters in the model can be describe by the equation below:

$$p(\mathbf{\Theta}, \boldsymbol{\xi}, \lambda, \eta | \mathbf{Y}) \propto \prod_{\mathbf{i=1}}^{\mathbf{I}} \prod_{\mathbf{j=1, j \neq i}}^{\mathbf{I}} \psi(\mathbf{y_{ij}} | \theta_{\xi_i, \xi_j}) \mathbf{p}(\mathbf{\Theta}|\boldsymbol{\lambda}) \mathbf{p}(\boldsymbol{\xi}|\eta) \mathbf{p}(\boldsymbol{\lambda}) \mathbf{p}(\eta) \tag{2.8}$$

where $\boldsymbol{\lambda} = (a_D, b_D, a_{OD}, b_{OD})$, $\psi(y_{ij}|\theta_{\xi_i,\xi_j})$ is assumed $Bernoulli(\theta_{\xi_i,\xi_j})$; and for the prior $p(\mathbf{\Theta}|\boldsymbol{\lambda})$, we set diagonal and off-diagonal elements independent from each other, $\theta_{l,l} \sim_{iid} beta(a_D, b_D)$ for diagonal elements, and $\theta_{l,l} \sim_{iid} beta(a_{OD}, b_{OD})$ for off- diagonal elements.

$p(\boldsymbol{\xi}|\eta) \sim CRP(\eta)$ introduced in equation (2.6). Hyperparameters a_{OD} and a_D follow a $gamma(\alpha_a, \beta_a)$ prior; and b_{OD} and b_D follow a $gamma(\alpha_b, \beta_b)$ prior. Finally, $p(\eta) \sim gamma(a, b)$.

Since the posterior distribution does not have a closed form we used an MCMC sampler to explore the joint posterior distribution from equation (2.8). The MCMC uses a Gibbs sampler to iteratively draw from the following full conditionals:

1. $p(\boldsymbol{\xi}|\boldsymbol{\lambda}, \eta, \mathbf{Y})$
2. $p(\boldsymbol{\Theta}|\boldsymbol{\xi}, \boldsymbol{\lambda}, \eta, \mathbf{Y})$
3. $p(\boldsymbol{\lambda}|\boldsymbol{\xi}, \mathbf{Y})$
4. $p(\eta|, \boldsymbol{\xi})$

2.3.1 Case Study Posterior Grouping

We applied the algorithm described in Section 2.3. The initial $\boldsymbol{\xi}$ vector is generated randomly from the prior distribution and each parameter and hyperparameter is set to produce noinformative flat priors, η for CRP is set to 0.5, hyper parameters for controlling $p(a_D, b_D, a_{OD}, b_{OD})$ are all equal to 2, parameters for controlling $p(\eta)$ are $a = 1$, $b = 1$. The running time grows exponentially, by approximately 30.5 hours per 1000 iteration after burn-in. Since we keep one set of parameters after each 10 iterations, we are able to get 100 likelihoods and 100 sets of vector X_is and matrix θ_{ξ_i,ξ_j}. By keep checking the likelihood every 1000 iteration, we are able to stop the simulation as the likelihood remain stable in a certain level. The likelihood increase dramatically, then the increasing trend becomes slower after 100 iterations and fluctuates only in a small interval after 600 iterations. We kill the simulation when we get 1200 posterior samples.

Figure 2.3 shows a clear grouping pattern. About 17 groups were identified. By analyzing figure 2.4, we can see that some of the groups have many connections within, but a little with other groups. In contrast, some groups have some connections within and several connections

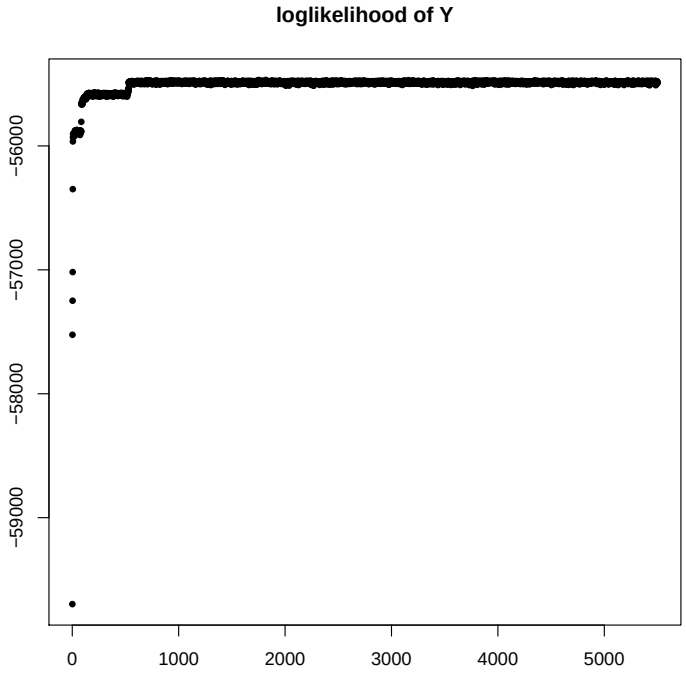

Figure 2.2: Estimated marginal loglikelihood of adjacency matrix $\mathbf{Y}$ from MCMC run.

with one or two other communities. The flexibility of SBM has allowed to discover patterns in the network that would have been impossible to detect otherwise.

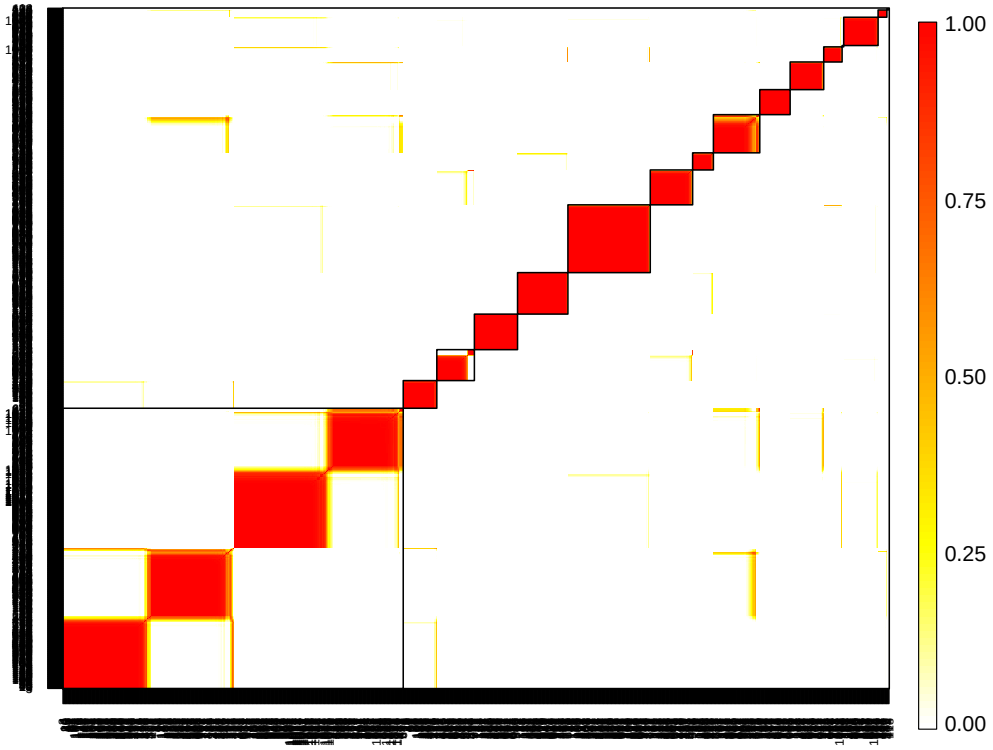

Figure 2.3: Estimated posterior probability of two subjects belonging to the same group. Lines indicate identified groups.

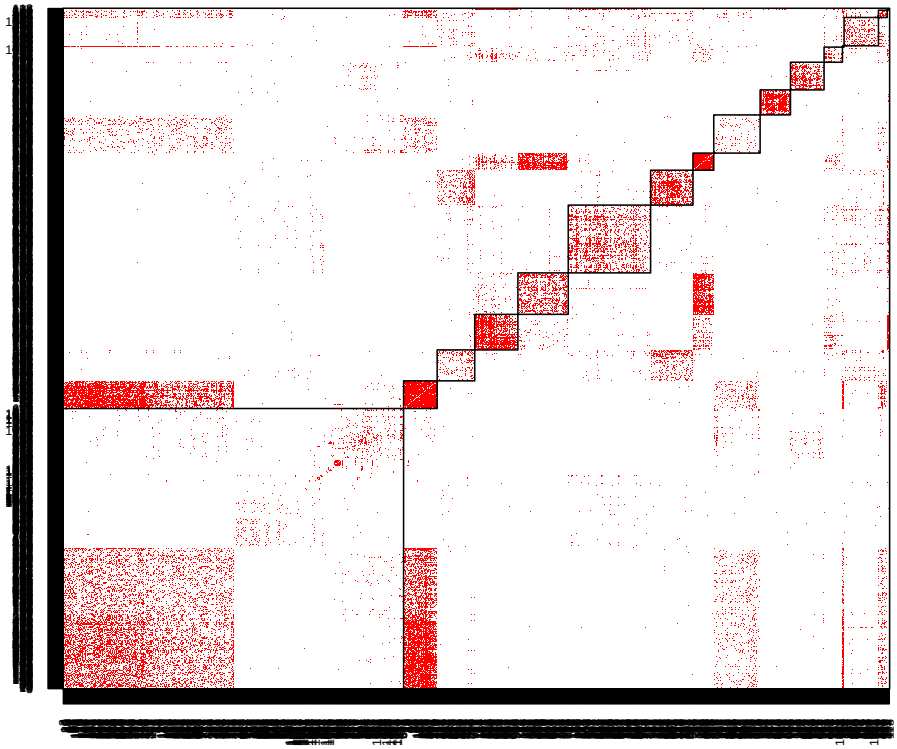

Figure 2.4: Adjacency matrix $\mathbf{Y}$ sorted by estimated groups. Lines indicate identified groups.

Chapter 3

Sampling and Projecting Networks

3.1 Introduction

The main goal of this paper is to test the efficiency and efficacy of different types of sampling technique, including a variety of respondent driven sampling. By drawing a small portion from the population network and projecting it into a simulated network of approximately the same size of the original population network, we are able to study how the network information, such as degree distribution, is being measured and translated into population parameters. The necessity of network-based sampling method is due to the difficulty of employing MCMC network-based algorithm onto high-dimensional networks. Attias (2000) mentioned that the computations in a MCMC framework is intractable, even for a very simple data. Twofold reasons are causing the inconvenience of reducing computer running time. First, the MCMC network-based method is a Markov chain, which is a single thread process that require to update each parameter and hyperparameter by the information on the previous iteration. It means no matter how many cores the processor has, the MCMC process will not accelerate by using all the core power. Second, the computational effort increase exponentially as the network dimension increases. When the number of subjects increase from n to $n+1$, the dimension of adjacency matrix increase from n^2 to n^2+2n+1. Furthermore, most of the complex real-world network data are high-dimensional. Given

the fact that the Facebook ego-friendship network we use in this paper just includes a portion of individuals involved in the survey, most of the network data are even larger and more unpractical to fit in any complex network models. Hence, inevitably, drawing samples becomes a important approach for social network analysis.

In this paper, we are going to employ four network-based sampling method to explore the whole population: Simple Random Sample (SRS), Respondent Driven Sampling (RDS), Anti-Cluster Respondent Driven Sampling (AC-RDS) and AC-RDS with double the sample size. The motivation of using SRS as a comparison boils down to a previous research. Zhou (2015) applied multiple sampling methods on faux.mesa.high, a small simulated dataset in ERGM package in `R`, and proved that SRS is a valid method to estimate whole network statistics, such as degree distribution. Even though Zhou (2015) found that stratified sampling might be able to gain a better result, stratified sampling method always require us to have more information about the population, such as race, age, etc, which might be difficult on large networks. Moreover, sampling method are designed to fulfill the purpose of research. Some sampling methods, such as RDS, are a necessity for certain researches, even though there might be other methods that has the potential of producing better results.

Another set of methods that are important in social network analysis are RDS and AC-RDS. They contribute in drawing a statistically representative sample when common network-based sampling methods are unpractical and a proper sampling frame is hard to be established (Spreen, 1992). Mainly, there are two cases we need such sampling methods: rare populations and hard-to-reach populations.

Rare population features the small proportion relative to whole population. Sometimes, they are also geographically dispersed (Heckathorn and Cameron, 2017). Locating rare populations is time-consuming and the cost of that is substantial, even exceeding that of the general sampling process. Due to the fact of that, it is infeasible to draw a large-enough representative sample and any detail, reliable statistical inferences. Hard-to-reach or "hidden" populations are characterized by some attributes of individuals that make themselves hard to be reached by outsiders or to response to the surveys. Usually, it involves stigma, illegal activities or socially unacceptable behaviors. Main difficulties of studying such populations

include that members in "hidden" population are more reluctant to cooperate, and tend to give false answers when it comes to sensitive questions. Moreover, "hidden" populations lack a sampling frame. In the following, we will briefly introduce the origin of such kind of sampling methods, and then what are the drawbacks and differences between each method. Finally, we will explain why we need to use Anti-Cluster RDS in our case study, Facebook friendship ego-network.

3.2 Sampling Methods for Networks

3.2.1 Snowball Sampling

Snowball sampling, also known as link-tracing sampling, was first proposed by (Goodman, 1961). However, (Heckathorn and Cameron, 2017) pointed out that this method was not capable of finding hidden population. Nowadays, the method is further developed to find hidden populations and named by link-tracing sampling more often. Contributions include Granovetter (1976) who applied snowball sampling to a large network data. He illustrated the importance of network sampling technique on a large dataset and provided an unbiased estimator and confident intervals for network density.

The snowball sampling process starts from a "convenient sample" (Heckathorn and Cameron, 2017). A group of initial subjects (or seeds) are drawn from the population and each subject recruits another group of subjects based on the friendship connection. In such way, several waves are developed from the "seed" subjects to a sufficient large portion of the population.

Clearly, snowball sample is not the perfect means to study the rare and the hidden population, since a huge bias may exist when neither the initial seeds nor the waves are selected by random. Even though bias can be possibly reduced after a sufficient large number of waves, we still have no idea about the magnitude and direction of the bias.(Erickson, 1979)

3.2.2 Respondent Driven Sampling

Respondent Driven Sampling (RDS) was first proposed by Heckathorn (1997). This was specifically designed to solve hidden-population problem, but different approaches were developed in the past few years to satisfy various research interests. Goel and Salganik (2009) examined a RDS as MCMC and claimed that variance increase due to the sampling design, that is, encouraging each subject recruits more than one subject. Gile (2011) provide a novel treatment of RDS with the feature of without-replacement sampling process and demonstrated the superior performance of estimating population mean when the size of hidden population is known.

The main different between snowball sampling and RDS is that, on the contrary of conventional thinking, Heckathorn (1997) proved that RDS can reach a balance independent to the convenient sample that was drawn as seed. After sufficient length of waves, the natural bias attenuates and, ultimately, RDS can draw a reliable sample.

Design Effect (DE) is the ratio of variances of estimated observed sample using RDS against estimated observed sample using SRS. It measures the increase in sample size required to achieve the same accuracy as that of an SRS. Salganik (2006) suggested that design effect $\hat{DE} = 2$ should be used to calculate the RDS sample size. In other words, RDS with double sample size might have better chance to create representative samples.

Anti-Cluster RDS

Anti-cluster Respondent Driven Sampling was first proposed by Khabbazian et al. (2017), as a novel design to solve the "referral bottlenecks" problem. When networks are divided by the similar connecting patterns and multiple clusters are formed, it is possible that there are very few connections between each group (given that, in reality, networks are not completely random or follow any specific distribution). In traditional RDS random walk designs, researchers only hire one or two individuals as the initial subject to start waves. Since subjects in a cluster may have very few connection to "outsiders", the performance of RDS is highly relative to the initial subjects, which are not picked by random. In other words, if a person

that is within an isolated group is picked, he/she is very likely to pick other "insider", and the person who is picked is likely to pick "insider" again. Finally, we may oversample one cluster, instead of covering the whole population.

To circumvent "referral bottlenecks", AC-RDS is featured by two types of referral requests: (A) Please refer contacts who do not know many of your contacts; (B) Please refer contacts who have many contacts who do not know you. Traditional RDS models assume a simple random walk on a Markov chain, However, this brand-new design assigns different weight on each vertex. The weight matrix is provided in Khabbazian et al. (2017). Authors confirmed the effectiveness of AC-RDS by a simulated study and with a Add-Health network study and observed that the covariances of the samples collected in AC-RDS are significantly smaller than the ones in RDS.

3.3 Projecting a Network Sample

In the previous chapter, we have shown that stochastic blockmodels can find useful grouping information from a network. However, in reality, for large or hard to find population of actors, it might be difficult to get information on all actors or all links between them. For example, in Ribeiro and Towsley (2010), networks from Flickr and Youtube were studied having millions of vertices and edges. The large size of these social networks makes it costly querying the entire network, particularly if the goal is to monitor these networks regularly over time. In addition, only few people or organizations have complete access to the data. Without knowing the true underlying structure of a population network, sampling becomes a natural way to solve this issue. A further statistical question in such case emerges: how well the properties of the true network can be modeled from those of the sampled network. In what follows, we will explore some traditional sampling techniques. Furthermore, we provide some evidence on whether and how a sampled network can be used to estimate the true population network and to what extent the degree distribution of the estimated network reflects that of the true network. Different sampling methods can be applied based on how we can access the network data and what is the goal of sampling. In some cases, the entire

network data could be accessed fully then a random edge or vertex can be selected. It could also be accessed restrictively when the network is hidden but allows analyzing (Handcock et al., 2015).

Here, we will explore how well we can reproduce the whole network based on a sample.

3.3.1 Learning from a Sample, Traditionally

In simple terms, the objective of survey methods is to select a representative subset of the population that will produce unbiased estimators with minimum variance under some restrictions usually associated with costs. If the sampling strategy is based on a probabilistic model, we could make generalizations and use sample statistics as estimates for population parameters. Moreover, population totals can be obtained by projecting or rescaling the observed values:

$$\sum_{i=1}^{n} \nu_i y_i \approx TotalY = \sum_{i=1}^{N} Y_i$$

where ν_i is the inverse of the probability of selection for the i-th subject.

3.3.2 Learning from a Network Sample

Bearing with the same objective, projection of network sample would aim to project each subject by a certain factor that is proportional to statistics relative to the subject. The subjects that are cloned represent the ones belonging to the same community and the projection factors indicate how many individuals of the whole population they represent. In such way, we expand the sample network to almost the same size of the original network where samples are drawn. In our case, subjects with more connections are more likely to be recruited, which cause a selection bias. To diminish this bias, the projection factor that we use must be proportional to the inverse of number of subjects' connections or degree. The

RDS projection factor is described as such:

$$xf_i = \frac{N}{\sum_{j=1}^{n} 1/degree(j)} \left(\frac{1}{degree(i)} \right) \tag{3.1}$$

where N is the target sample size, $degree(i)$ is the number of degree of subject i. In the sample network of $i = 1, \ldots, n$, subject i is expanded to the size of xf_i, so that the sample network of size n is expanded to one of size N. xf_i is not an integer, unfortunately. To project the graph associated with each vertex we can only directly clone the vertices. Thus, we round xf_i to its nearest integer. Next step is simulating the absence and presence of edges with the posterior probability of $\boldsymbol{\Theta}$ that is obtained using the same method in Section 2.3.

3.4 Simulation Experiment

We conducted a simulation experiment to compare the four sampling methods: SRS, RDS, AC-RDS and AC-RDS with design effect equal to 2, i.e. double the sample size, hence fore, denoted as AC-RDS(DE=2). First, we use each sampling method to draw random samples of size n_b, a hundred times. Specifically, for SRS we randomly selected $n_b = 103$ subjects and kept all the connections between subjects in the sample, but lost all the other connections; for the other methos, one seed is randomly select in each sample, then, subjects are required to refer two contacts that they know, without replacement. The referral type of RDS is a simple Random Walk with the same weight on each edge, while the referral type of AC-RDS is an Anti-cluster Random Walk with the referral request of A and B defined in section 3.2.2. For both AC-RDS and RDS, we selected $n_b = 100$ subjects, for AC-RDS(DE=2) we used $n_b = 200$.

Second, we apply the same MCMC algorithm, that we used to detect clusters in the whole Facebook network, to each sample (see Section 2.3), we obtain a n_b-dimensional vector of final groups ($\boldsymbol{\xi}_{n_b}$), a $n_b \times n_b$ matrix of posterior mean probability of two subjects to have a connection ($\boldsymbol{\Theta}_{n_b}$) and a $n_b \times n_b$ matrix of posterior mean probability that two subjects

belong to the same group ($group_{n_b}$). Third, we set the projection factor for SRS as $xf = 10$ and calculate the projection factor for each RDS method, with equation(3.1). As mentioned before, xf_i is rounded to the nearest integer, becoming f_i the projection factor for subject i. In such way, the population size after projection for SRS is fixed as 1030 and the one for other methods was controlled between $(N \pm 10) = [1024, 1044]$ by slightly adjusting f_i's when needed. The projection algorithm for obtaining an $N \times N$ $\mathbf{\Theta}$ matrix is illustrated as following:

$$\begin{bmatrix} \theta_{11} & \theta_{12} & \cdots & \theta_{1n} \\ \theta_{21} & \theta_{22} & & \vdots \\ \vdots & & \ddots & \vdots \\ \theta_{n1} & \cdots & \cdots & \theta_{nn} \end{bmatrix} \rightarrow \begin{bmatrix} \theta_{111} & \cdots & \theta_{11f_1} & \theta_{121} & \cdots & \theta_{11f_2} & \cdots & \theta_{1nf_n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & & \vdots \\ \theta_{11f_1} & \cdots & & \theta_{12f_1} & \cdots & & & \vdots \\ \theta_{211} & \cdots & \theta_{21f_1} & \theta_{221} & \cdots & \theta_{22f_2} & & \vdots \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & & \vdots \\ \theta_{21f_2} & \cdots & & \theta_{22f_2} & \cdots & & & \vdots \\ \vdots & & & & & & \ddots & \vdots \\ \theta_{n1f_n} & \cdots & & & & & & \theta_{nnf_n} \end{bmatrix}$$

where f_i is the projection factor for node i, and $\theta_{ij1} = \theta_{ij2} = \cdots = \theta_{ijf_n}$, which indicate the times that each node reproduces.

Finally, given θ_{ij} we can simulate directly from $y_{ij} \sim Bernoulli(\theta_{ij})$. Therefore, we can simulate one sample network $\mathbf{Y}_N$ from each sample $\mathbf{Y}_{n_b}$, so that we are able to obtain 100 simulations of the Facebook friendship network for each network sampling technique.

3.4.1 Simulation Experiment Results

After fitting 100 samples with each sampling method and projecting 100 simulated friendship network, we check how close the projected network was to the real one by comparing some essential network properties. First for the degree distribution. we calculate the interquartile interval, which is the interval between lower quartile and upper quartile, of 100 networks.

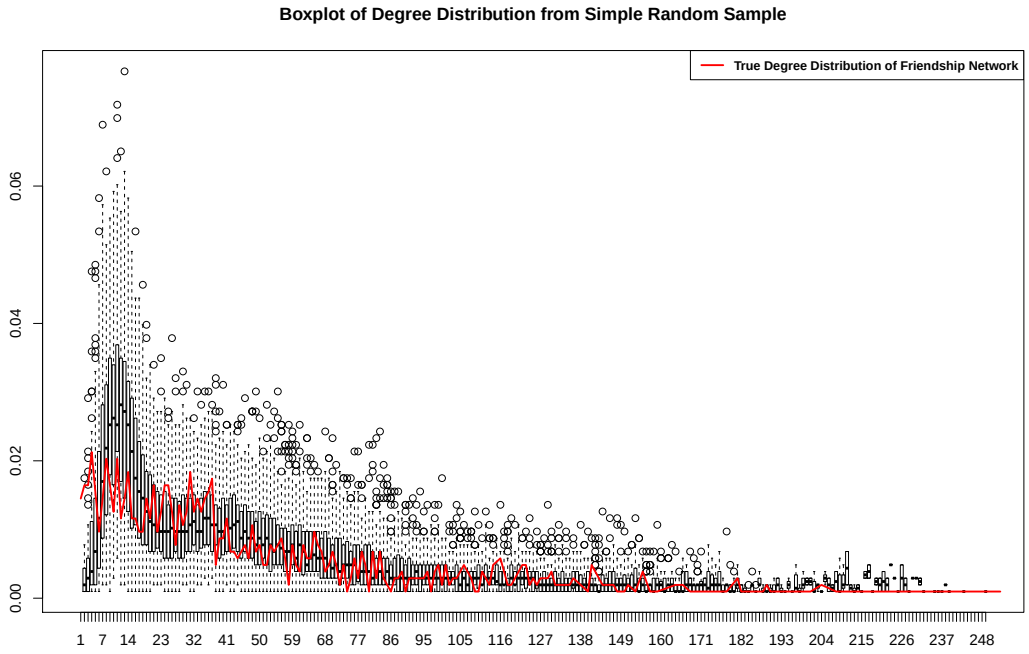

Figure 3.1: Boxplot of estimated degree distribution from SRS along with the true degree distribution.

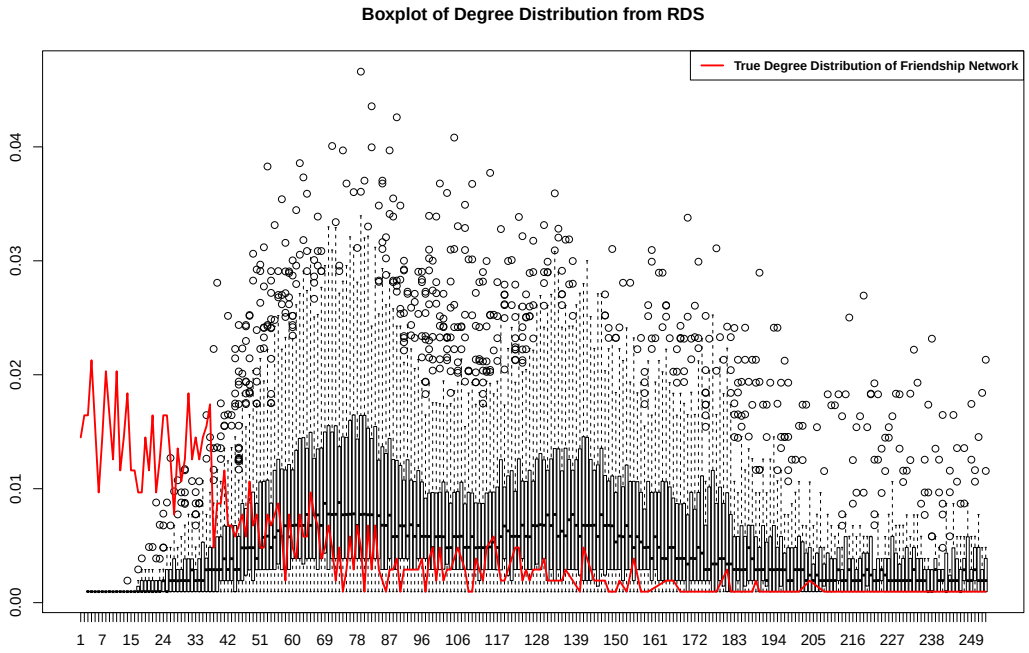

Figure 3.2: Boxplot of estimated degree distribution from RDS along with the true degree distribution.

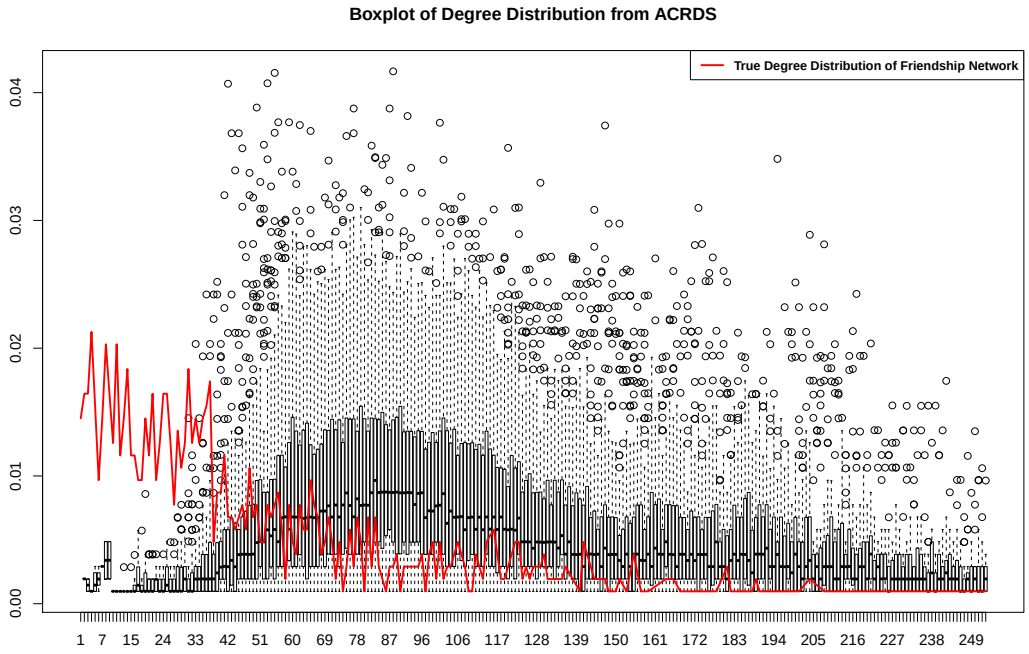

Figure 3.3: Boxplot of estimated degree distribution from ACRDS along with the true degree distribution.

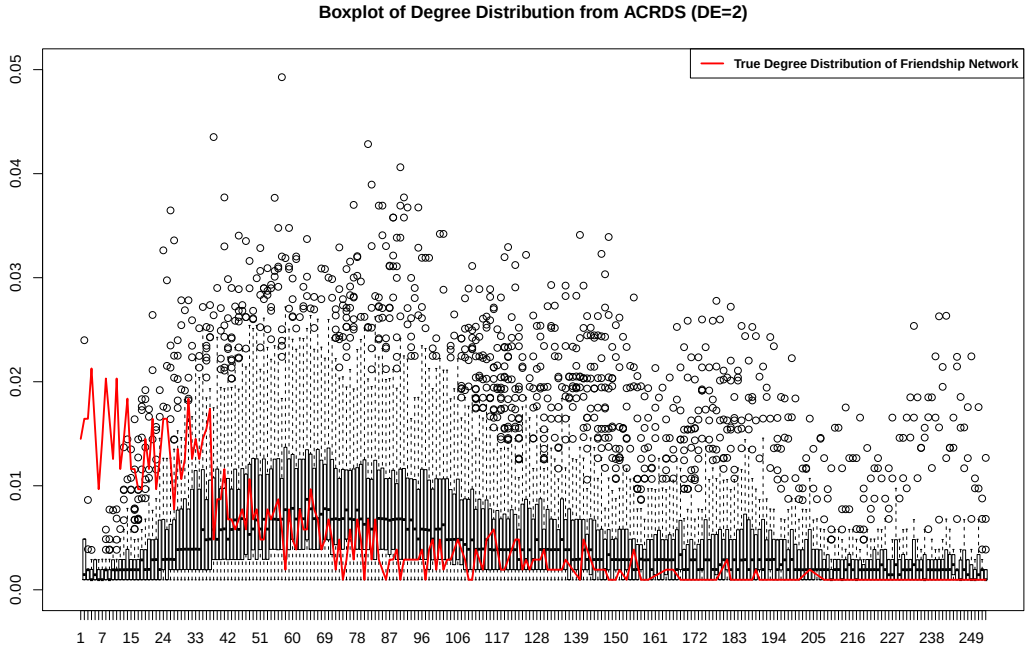

Figure 3.4: Boxplot of estimated degree distribution from ACRDS (DE=2) along with the true degree distribution.

To illustrate the output, we generate the degree distribution boxplots and degree distribution quartile interval plots along with true degree distribution shown in Figures 3.1, 3.2, 3.3, and 3.4.

Figure 3.1, shows that SRS paired with SBM and our projection approach produces a reasonably close estimation of the degree distribution, with the exception of the left tails, where the prevalence of degrees bellow 10 tends to be underestimated. This is similar to what Zhou (2015) found, but at a larger scale, with a small network the method was only underestimating the probability of degree zero, here we see that problem extends to other small degrees.

In contrast, RDS and AC-RDS, figures 3.2, and 3.3, produced a poor fit. As described earlier, AC-RDS was developed to fix RDS tendency to oversample high degree nodes, we see this reflected in a reduction of the second mode around degree 130 that appears in RDS estimated degree distribution (figure 3.2) but it is missing in AC-RDS degree distribution (figure 3.3). Both RDS methods produced biased degree distributions toward higher degrees, shifting the mode from degrees below 50 to degrees around 75.

However, AC-RDS with double sample size, figure 3.4, performed better than RDS and AC-RDS, figure 3.2, and 3.3, worse than SRS 3.1. By drawing 20% from the whole Facebook network with AC-RDS method, we are able to project simulated networks that is closer to the true one, but still far from being representative.

In addition, we evaluated two other network metrics. Results are presented in table 3.1, as with degree distribution SRS is the closest to the true value for the mean shortest distance, however AC-RDS methods are closer for the clustering coefficient. It appears that maybe AC-RDS could potentially be a better option than SRS in some cases. Further analysis is needed to confirm or denied this observation.

Table 3.1: Networks Statistics Comparison

	True	SRS	RDS	AC-RDS	AC-RDS (DE=2)
Mean shortest distance	2.95	2.74	2.099	2.07	2.20
Clustering coefficient	0.5342	0.2454	0.3113	0.3078	0.3072

- Column one was calculated from 1034×1034 whole Facebook network
- Column two is the mean value of 100 simulated networks, projected from 103×103 simple random sample using Stochastic Blockmodel.
- Column three is the mean value of 100 simulated networks, projected from 100×100 respondent driven sample using Stochastic Blockmodel.
- Column four is the mean value of 100 simulated networks, projected from 100×100 anti-clustering respondent driven sample using Stochastic Blockmodel.
- Column five is the mean value of 100 simulated networks, projected from 200×200 anti-clustering respondent driven sample using Stochastic Blockmodel.

Chapter 4

Conclusions

In this report we explored the statistical challenge of using a sampled network data to reproduce the population or complete network. We presented the concept of network, discussed different properties of complex networks and compared the population network estimations based on stochastic blockmodels and ERGM. We used as case study a Facebook friendship network gathered via survey. Although this network is in itself a sample from the whole Facebook's network, it is a more realistic representation of the problem than a simulated network. The first part of Chapter 2 defined some network properties followed by the introduction of Exponential Random Graph Models and its application to model the Facebook network. It was shown that without including nodes attributes and using only network attributes, such as degree or triangles statistics, ERGM did not fit the Facebook data well. The last part of Chapter 2 includes basic concepts and the application of stochastic blockmodels, which are used to decompose a network in classes of actors with common properties so certain grouping patterns can be found. We obtained a clear clustering pattern. The Facebook data was found to be clearly separated into approximately 17 groups with similar friendship preferences.

In Chapter 3, we described sampling techniques often related with the study of networks, and inference approaches that we explored to assess whether network information can be estimated using only a sub-network. We provide some evidence on whether and how a sampled

network can be used to estimate the true population network and to what extent the degree distribution and other metrics of the estimated network reflects those of the true network. Four sampling methods were evaluated: SRS, RDS, AC-RDS, and AC-RDS(DE=2), samples of about $n = 100$ nodes were drawn using each method with the exception of the last one that used samples of $n = 200$ vertices; SBM were used to model the sampled network of size n;, then we expanded the matrices of estimated probabilities of connection to simulate whole population networks ($N \approx 1034$).

As one may expect, SRS was proved as a reasonable approach to draw representative samples. after 100 samples are projected, SRS gives a reasonably close estimation of degree distribution, except for the range of degree below 10. Meanwhile, decent quartile intervals are given, between which the true degree distribution fall. This means the conclusion of Zhou (2015) is proven to be able to reproduced even on a large population network. Additionally, SRS perform better on a small network, since only the probability of zero degree is underestimated. On the contrary, RDS and AC-RDS, figures 3.2, and 3.3, do not perform as well as expected. Not only the estimations of lower degree distribution are poor, the estimations of higher degree distribution are far from the true one, even though AC-RDS does reduce the bias toward higher degree nodes. Meanwhile, AC-RDS with double sample size (figure 3.4) gives the best results among the three RDS methods. The median of degree distribution is closer to the true degree than the other two methods. As we described earlier, AC-RDS was developed to fix the tendency of RDS to oversample in single highly connected subgroups or in high degree nodes, we see this in a reduction of the second mode around degree 130 that appears in RDS estimated degree distribution (figure 3.2) but it is missing in AC-RDS degree distribution (fig 3.3). The first mode appears in figure 3.2 and 3.3 reduces in the AC-RDS with double sample size (figure 3.4). In general, both RDS might not be the best option to draw representative samples of the whole population network, since they produce a natural bias to high degrees, and shift the mode of degree distributions from below 50 to around 75. To produce samples with the same power as SRS, a larger sample size might be needed.

In addition, we evaluated two other network metrics. Results are presented in table 3.1,

as with degree distribution SRS is the closest to the true value for the mean shortest distance, however AC-RDS methods are closer for the clustering coefficient. It appears that maybe AC-RDS could potentially be a better option than SRS in some cases. Further analysis is needed to confirm or denied this observation.

Bibliography

Hagai Attias. A variational baysian framework for graphical models. In *Advances in neural information processing systems*, pages 209–215, 2000.

Panos Balatsoukas, Catriona M Kennedy, Iain Buchan, John Powell, and John Ainsworth. The role of social network technologies in online health promotion: a narrative review of theoretical and empirical factors influencing intervention effectiveness. *Journal of medical Internet research*, 17(6), 2015.

Neli Blagus, Lovro Šubelj, Gregor Weiss, and Marko Bajec. Sampling promotes community structure in social and information networks. *Physica A: Statistical Mechanics and its Applications*, 432:206–215, 2015.

David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

Bonnie H Erickson. Some problems of inference from chain data. *Sociological methodology*, 10:276–302, 1979.

Krista J Gile. Improved inference for respondent-driven sampling data with application to hiv prevalence estimation. *Journal of the American Statistical Association*, 106(493): 135–146, 2011.

Sharad Goel and Matthew J Salganik. Respondent-driven sampling as markov chain monte carlo. *Statistics in medicine*, 28(17):2202–2229, 2009.

Anna Goldenberg, Alice X Zheng, Stephen E Fienberg, and Edoardo M Airoldi. A survey of statistical network models. *Foundations and Trends® in Machine Learning*, 2(2):129–233, 2010.

Leo A Goodman. Snowball sampling. *The annals of mathematical statistics*, pages 148–170, 1961.

Mark Granovetter. Network sampling: Some first steps. *American journal of sociology*, 81 (6):1287–1303, 1976.

Mark S Handcock, Krista J Gile, and Corinne M Mar. Estimating the size of populations at high risk for hiv using respondent-driven sampling data. *Biometrics*, 71(1):258–266, 2015.

Douglas D Heckathorn. Respondent-driven sampling: a new approach to the study of hidden populations. *Social problems*, 44(2):174–199, 1997.

Douglas D Heckathorn and Christopher J Cameron. Network sampling: From snowball and multiplicity to respondent-driven sampling. *Annual review of sociology*, 43:101–119, 2017.

Charles Kemp, Joshua B Tenenbaum, Thomas L Griffiths, Takeshi Yamada, and Naonori Ueda. Learning systems of concepts with an infinite relational model. In *AAAI*, volume 3, page 5, 2006.

Mohammad Khabbazian, Bret Hanlon, Zoe Russek, Karl Rohe, et al. Novel sampling design for respondent-driven sampling. *Electronic Journal of Statistics*, 11(2):4769–4812, 2017.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

Jure Leskovec and Julian J Mcauley. Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547, 2012.

Mark Newman. *Networks: an introduction*. Oxford university press, 2010.

Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2): 167–256, 2003.

Anatol Rapoport. Contribution to the theory of random and biased nets. In *Social Networks*, pages 389–409. Elsevier, 1977.

Alireza Rezvanian and Mohammad Reza Meybodi. Sampling social networks using shortest paths. *Physica A: Statistical Mechanics and its Applications*, 424:254–268, 2015.

Bruno Ribeiro and Don Towsley. Estimating and sampling graphs with multidimensional random walks. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 390–403. ACM, 2010.

Matthew J Salganik. Variance estimation, design effects, and sample size calculations for respondent-driven sampling. *Journal of Urban Health*, 83(1):98, 2006.

Jayaram Sethuraman. A constructive definition of dirichlet priors. *Statistica sinica*, pages 639–650, 1994.

Cosma Rohilla Shalizi and Alessandro Rinaldo. Consistency under sampling of exponential random graph models. *Annals of statistics*, 41(2):508, 2013.

Marinus Spreen. Rare populations, hidden populations, and link-tracing designs: What and why? *Bulletin of Sociological Methodology/Bulletin de Methodologie Sociologique*, 36(1): 34–58, 1992.

David Strauss. On a general class of models for interaction. *SIAM review*, 28(4):513–527, 1986.

Yuchung J Wang and George Y Wong. Stochastic blockmodels for directed graphs. *Journal of the American Statistical Association*, 82(397):8–19, 1987.

Harrison C White, Scott A Boorman, and Ronald L Breiger. Social structure from multiple networks. i. blockmodels of roles and positions. *American journal of sociology*, 81(4): 730–780, 1976.

Zhao Xu, Volker Tresp, Kai Yu, and Hans-Peter Kriegel. Infinite hidden relational models. *arXiv preprint arXiv:1206.6864*, 2012.

Shu Zhou. Exploring network models under sampling. 2015.

Appendix A

Stochastic Blockmodel for Networks

This part is C++ codes for MCMC Gibbs sampler for the whole Facebook network.

```
/* nohup g++ -I/usr/local/include -L/usr/local/lib -lgsl -lgslcblas -lm -g -o Allian_BeB
   ./Allian_BeBin_PDPv2_1.exe & */


/*
 *  Allian_BeBin_PDP.cpp
 *
 *
 *  Created by Abel Rodriguez on 5/10/06.
 * Modified by Perla Reyes 10/11/11.
 * Modified by Perla Reyes 11/8/11 to read initial xi values.
 * Modified by Perla Reyes 11/15/11 to run a beta-binomial model.
 * Modified by Perla Reyes 11/15/11 to use a Poisson DP.
 * Modified by Perla Reyes 11/23/11 to compute post-likelihoods.
 * Modified by Perla Reyes 6//01/18 to Run Facebookdata and large n
 * Modified by Perla Reyes 6//4/18 adding thining.
 */
```

```
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <time.h>
#include <math.h>
#include <string.h>
#include <vector>
#include <stdio.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_cdf.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_sf_gamma.h>
#include <gsl/gsl_sf_log.h>

#include "./utilities2.h"

using namespace std;

const int n = 103;
const bool sym = true;/* to run symmetric or asymmetric networks*/
```

```
bool elim_and_relabel(int n, int *xi, int fp){
  int i;
  int label_fp = xi[fp];
  xi[fp] = -1;
  //Check if this is the only item with that label
  bool sw = true;

  for(i = 0; i<=n-1; i++){
    if((i!=fp) && (xi[i]==label_fp)){
      sw = false;
      break;
    }
  }
  if(sw==true){
    for(i = 0; i<=n-1 ;i++){
      if(xi[i] > label_fp){
xi[i] = xi[i]-1;
      }
    }
    return true;
  }else{
    return false;
  }
}

double logpredictive_gg(int n, double sumz, double sumz2, double tau2){
  return -0.5*n*log(2*M_PI) - 0.5*log(n + 1/tau2) - 0.5*log(tau2) - 0.5*sumz2 + 0.5*gsl_
}
```

```
double loglikelihood_g(double z, double mean, double var){
return -0.5*log(2*M_PI) - 0.5*log(var) - 0.5*gsl_pow_2(z-mean)/var;
}

double logpredictive_bb(int n, double sumz, double a, double b){
  return gsl_sf_lnbeta(a + sumz, b + n - sumz) - gsl_sf_lnbeta(a,b);
}

double gen_alpha(double alpha, int n, int L, double aalpha, double balpha, gsl_rng *r){
  double eta  = gsl_ran_beta (r, alpha+1.0, n);
double odr  = (aalpha + L - 1)/(n*(balpha - log(eta)));
double prob = odr/(1.0 + odr);
  double u    = gsl_ran_flat(r, 0.0, 1.0);
  if(u < prob){
    alpha = gsl_ran_gamma(r, aalpha+L, 1.0/(balpha-log(eta)));
  }else{
    alpha = gsl_ran_gamma(r, aalpha+L-1, 1.0/(balpha-log(eta)));
  }
  return(alpha);
}

void update_alpha_nu(double *alpha, double *nu, int n, int *L, int *xi, double aalpha, d

  int i, k;
   vector<int> ncat;
   double sumy = 0;
   double sumz = 0;
   double x = gsl_ran_beta (r, *alpha + 1.0, n-1);
```

```
  ncat.resize(*L, 0);
  for(i = 0; i<=n-1 ;i++){
    ncat[xi[i]] += 1;
  }
  // display_ncat(ncat);

  if(*L > 1){
    for(k=0; k <= *L-2; k++){
      sumy += gsl_ran_bernoulli(r, (*alpha/(*alpha + (k+1)* *nu)) );
      //  cout << " k= "<< k << ", "<<(alpha/(alpha + (k+1)*nu))<<", sumy= "<< sumy <<en
      if(ncat[k] > 1){
for(i = 1; i<=ncat[k]-1; i++){
  sumz += gsl_ran_bernoulli(r, (1-*nu)/(i-*nu) );
  //   cout << " i= "<< i  << ", "<< (1-nu)/(i-nu) << ", sumz= "<< sumz<<endl;
}
      }
    }
  }
  if(ncat[*L-1] > 1){
    for(i =1; i<=ncat[*L-1]-1; i++){
      sumz += gsl_ran_bernoulli(r, (1-*nu)/(i-*nu) );
      //  cout << " i= "<< i  << ", "<< (1-nu)/(i-nu) << ", sumz= "<< sumz<<endl;
    }
  }
  //  cout << "Antes " << alpha << " " << nu<<endl;
  *alpha = gsl_ran_gamma(r, aalpha + sumy, 1.0/(balpha - log(x)));

  *nu = gsl_ran_beta (r, anu + *L-1 - sumy, bnu + sumz);
  //  cout << "Despues " << alpha << " " << nu<<endl;
```

```
}

double update_xi_wllik(int n, int *L, int *Z, int *xi, double aD, double bD, double aOD,

    int i,j,k,l;
    int cxi;
    bool sw;
    vector<double> q;
    vector<int> ncat;
    vector<int> nz;
    vector< vector<int> > NZ;
    vector<double> sumz;
    vector< vector<double> > SUMZ;
    double llik;

  ncat.resize(*L, 0);
  for(i = 0; i<=n-1 ;i++){
    ncat[xi[i]] += 1;
  }

  /*complete summary statistics */
  for(l=0; l < *L; l++){
    NZ = extend_B(NZ,0);
    SUMZ = extend_B(SUMZ,0);
  }

  for(i = 0; i<=n-1 ;i++){
    for(j = i+1; j<=n-1 ;j++){
```

```
      if( (sym == true) && (xi[i]==xi[j]) ){
      SUMZ[xi[i]][xi[j]] += Z[i*n+j];
      NZ[xi[i]][xi[j]] += 1;
     // cout << "i,j " <<i<<j << " xi[i] equal "<< xi[i] <<", Z= "<<  Z[i*n+j]<< endl;
      }else{
        //cout << "i,j " <<i<<j << " xi[i]= "<< xi[i] << "xi[j]= "<< xi[j]<<endl;
        //cout <<"Z["<<i*n+j<<"]= "<<  Z[i*n+j]<<"Z["<<j*n+i<<"]= "<<  Z[j*n+i]<< endl;

      SUMZ[xi[i]][xi[j]] += Z[i*n+j];
      SUMZ[xi[j]][xi[i]] += Z[j*n+i];
      NZ[xi[i]][xi[j]] += 1;
      NZ[xi[j]][xi[i]] += 1;
//cout << "i,j " <<i<<j << "error"<< endl;
      }
    }
  }
/*  cout << "Sum zeta " << endl;
  display_B(SUMZ);
cout << "n zeta " << endl;
  display_B(NZ);
cout << "Sum zeta^2 " << endl;
display_B(SUMZ2);
  */
  //  i = 4;
  for(i = 0; i<=n-1 ;i++){
    cxi = xi[i];
 /*   cout<< "i=" << i << "L=" <<*L<<"cxi=" <<cxi<< endl;
    cout << "Sum zeta " << endl;
    display_B(SUMZ);
```

```
    cout << "n zeta " << endl;
    display_B(NZ);
*/
    sw  = elim_and_relabel(n, xi, i);

    if(sw==true){ //if i was alone in its category
      *L = *L - 1;
      ncat.erase(ncat.begin() + cxi);

      // Finds suff stats for Z_{i,.} and Z_{.,i} the i-th row/column
      nz.resize(*L, 0);
      sumz.resize(*L, 0.0);
      for(l = 0; l<=*L-1 ;l++){
       nz[l]    = 0;
      sumz[l]  = 0.0;
      }
      for(j = 0; j<=n-1 ;j++){
       if(j!=i){
         nz[xi[j]]    += 1;
         sumz[xi[j]]  += Z[i*n+j];
      }
      }

      // remove i-th subject from the sufficient statistics
      NZ = contract_B(NZ, cxi+1);
      SUMZ = contract_B(SUMZ, cxi+1);

    }else{
      ncat[cxi]--; //removes i from the number of subjects in its group
```

```
// Finds suff stats for Z_{i,.} and Z_{.,i} the i-th row/column
nz.resize(*L, 0);
sumz.resize(*L, 0.0);
for(l = 0; l<=*L-1 ;l++){
 nz[l]    = 0;
 sumz[l]  = 0.0;
}
for(j = 0; j<=n-1 ;j++){
if(j!=i){
  nz[xi[j]]    += 1;
  sumz[xi[j]]  += Z[i*n+j];
}
}

// remove i-th subject from the sufficient statistics
for(l=0; l < *L; l++){
if( (sym == true) && (cxi==l) ){
  if(sumz[l] > SUMZ[cxi][l]) {cout << "Negativo SUM same group" <<endl;}
  if(nz[l] > NZ[cxi][l]) {cout << "Negativo N same group" <<endl;}
  SUMZ[cxi][l] -= sumz[l];
  NZ[cxi][l] -= nz[l];
} else if(sym == true){
  if(sumz[l] > SUMZ[cxi][l]) {cout << "Negativo SUM diff group Row" <<endl;}
  if(nz[l] > NZ[cxi][l]) {cout << "Negativo N diff group Row" <<endl;}
  if(sumz[l] > SUMZ[l][cxi]){
    cout << "Negativo SUM diff group Col, sumz[l]=" << sumz[l] << "> SUMZ[l][cxi]=
    cout << "l, cxi= " <<l<<","<<cxi<< " SUMZ[cxi][l]= " << SUMZ[cxi][l]<<endl;
  }
```

```
        if(nz[l] > NZ[l][cxi]) {cout << "Negativo N diff group CoL" <<endl;}
        SUMZ[cxi][l] -= sumz[l];
        SUMZ[l][cxi] -= sumz[l];
        NZ[cxi][l] -= nz[l];
        NZ[l][cxi] -= nz[l];
      //cout << "i,j " <<i<<j << "xi[i] no equal, Z= "<<  Z[i*n+j]<< endl;
      }else{
        cout << "Error, Special Code missing" << endl;
      }
      }

    } // if(sw==true){

  //computing probabilities
    q.resize(*L+1, 0.0);
    for(l = 0; l<=*L-1 ;l++){
      q[l] = log(ncat[l] - nu);
       //cout << " l = "<<l<<endl;
      if(sym == true){
for(k = 0; k<=*L-1 ;k++){
//    cout << " nk = "<<NZ[l][k]<<endl;
//  cout << " sk = "<<SUMZ[l][k]<<endl;
  if(l==k){
    q[l] += logpredictive_bb(NZ[l][k]+nz[k], SUMZ[l][k]+sumz[k], aD, bD);
    q[l] -= logpredictive_bb(NZ[l][k], SUMZ[l][k], aD, bD);
  }else{
    q[l] += logpredictive_bb(NZ[l][k]+nz[k], SUMZ[l][k]+sumz[k], aOD, bOD);
    q[l] -= logpredictive_bb(NZ[l][k], SUMZ[l][k], aOD, bOD);
  }
```

```
}
      }else{
cout << "Error, Special Code missing" << endl;
      }


    }


    q[*L] = log(*L * nu + alpha);
    for(l = 0; l<=*L-1 ;l++){
      q[*L] += logpredictive_bb(nz[l], sumz[l], aD, bD);
  }
    //  display_a(q);
    //cout << "###############" << endl;

    q = standarize_log_prob(q);
    //display_a(q);
    //cout << "%%%%%%%%%%%%%" << endl;

    //  display_lineararray_int(n, xi);
    xi[i] = own_ran_discrete_unif(q, r);
    //xi[i] = 2;
    //display_lineararray_int(n, xi);
    //cout << "^^^^^^^^^^^^^^^" << endl;
    //cout << "L  = " << L << endl;
    //display_B(NZ);

    if(xi[i] == *L){
      ncat.insert(ncat.begin() + xi[i], 0);
```

```
      // return the i-th subject to the sufficient statistics
      NZ = extend_B(NZ,0);
      SUMZ = extend_B(SUMZ,0.0);
      for(l=0; l < *L; l++){
if( (sym == true) && (xi[i]==l) ){
  SUMZ[xi[i]][l] += sumz[l];
  NZ[xi[i]][l] += nz[l];
} else if(sym == true){
  SUMZ[xi[i]][l] += sumz[l];
  SUMZ[l][xi[i]] += sumz[l];
  NZ[xi[i]][l] += nz[l];
  NZ[l][xi[i]] += nz[l];
//cout << "i,j " <<i<<j << "xi[i] no equal, Z= "<<  Z[i*n+j]<< endl;
}else{
  cout << "Error, Special Code missing" << endl;
}
      }
      *L = *L + 1;
    }else{
      // return the i-th subject to the sufficient statistics
      for(l=0; l < *L; l++){
if( (sym == true) && (xi[i]==l) ){
  SUMZ[xi[i]][l] += sumz[l];
  NZ[xi[i]][l] += nz[l];
} else if(sym == true){
  SUMZ[xi[i]][l] += sumz[l];
  SUMZ[l][xi[i]] += sumz[l];
  NZ[xi[i]][l] += nz[l];
  NZ[l][xi[i]] += nz[l];
```

```
//cout << "i,j " <<i<<j << "xi[i] no equal, Z= "<<  Z[i*n+j]<< endl;
}else{
  cout << "Error, Special Code missing" << endl;
}
      }
    }
    ncat[xi[i]]++;
    /*display_lineararray_int(n, xi);
  display_ncat(ncat);
  display_B(B);
  display_B(NZ);*/
  }

 //cout << "UPDATED XI: " <<*L<< endl;

  llik = gsl_sf_lngamma(alpha+1) -  gsl_sf_lngamma(alpha + n) - gsl_sf_log(alpha + *L*nu
  for(k = 0; k<=*L-1 ; k++){
    llik += lgamma(ncat[k] - nu) + gsl_sf_log(alpha + k*nu);
    for(l = k; l<=*L-1; l++){
      if(l==k){
llik += logpredictive_bb(NZ[k][l], SUMZ[k][l], aD, bD);
      }else{
llik += logpredictive_bb(NZ[k][l], SUMZ[k][l], aOD, bOD);
if(sym == false) llik += logpredictive_bb(NZ[l][k], SUMZ[l][k], aOD, bOD);
      }
    }
  }

  return(llik);
```

```
}

vector<double> sample_Theta(int n, int L, int *Z, int *xi, double aD, double bD, double
  int i,j,k,l;
  double post_a = 0.0;
  double post_b  = 0.0;

  double S[L][L];
  double N[L][L];
  for(l = 0; l<=L-1 ;l++){
    for(k = l; k<=L-1 ;k++){
      S[l][k]= 0.0;
      S[k][l]=0.0;
      N[l][k]= 0.0;
      N[k][l]=0.0;
    }
  }
  vector<double> B(L*L, 0);

  for(i = 0; i<=n-1 ;i++){
    for(j = i+1; j<=n-1 ;j++){
      if( (sym == true) && (xi[i]==xi[j]) ){
S[xi[i]][xi[j]] += Z[i*n+j];
N[xi[i]][xi[j]] += 1.0;
      }else{
S[xi[i]][xi[j]] += Z[i*n+j];
S[xi[j]][xi[i]] += Z[j*n+i];
N[xi[i]][xi[j]] += 1.0;
N[xi[j]][xi[i]] += 1.0;
```

```
      }
    }
  }

  for(k = 0; k<=L-1 ; k++){
    for(l = k; l<=L-1; l++){
      if(l==k){
post_a = aD + S[k][l];
post_b = aD + N[k][l] - S[k][l];
      }else{
post_a = aOD + S[k][l];
post_b = aOD + N[k][l] - S[k][l];
      }
      B[l*L + k] = gsl_ran_beta (r, post_a, post_b);
      B[k*L + l] = B[l*L + k];
    }
  }

  return(B);
}




int main(){
  int i,j,k,l;
  int inValue;
  int print_status=0;
  int keep_status=0;
  int *Y;
```

```
  Y = new int[n*n];  //memory allocated
  float *pairwise_xi = new float[n*n];
  float *mean_Theta = new float[n*n];

  int L = 1;
  int xi[n];
  double llik;

/*Initialize Simulation parameters*/
  double aD = 2.00;
  double bD = 2.00;
  double aOD = 2.00;
  double bOD = 2.00;

  double aalpha = 1.0;
  double balpha = 1.0;
  double alpha = aalpha/balpha;
  //double alpha = 1.5;
  double anu = 1.0;
  double bnu = 9.0;
  double nu = anu/(anu + bnu);
  //double nu = 0.24;

  int iter = 1000;
  int burn = 500;
  int print_each = 100;
  int keep_each = 5;

/*Input files */
```

```
  ifstream in_Y("Yfb.txt");
  ifstream in_xi("FB_XIrandom2.txt");
//  ifstream in_xi("FB_XIn.txt");

/*Output files */
  ofstream out_xi("xi_v5_Rdm2.txt");
  ofstream out_llik("llik_v5_Rdm2.txt");
  ofstream out_status("Status_v5_Rdm2.txt");
  ofstream out_alpha("alpha_v5_Rdm2.txt");
  ofstream out_nu("nu_v5_Rdm2.txt");
  ofstream out_pairxi("pairwise_xi_v5_Rdm2.txt");
  ofstream out_Theta("Theta_v5_Rdm2.txt");

/*
//initialize pairwise
  float pairwise_xi[n][n];
 // float mean_Theta[n][n];
  for(i=0; i<=n-1; i++){
    for(j=i; j<=n-1; j++){
      pairwise_xi[i][j] = 0.0;
      pairwise_xi[j][i] = 0.0;
   //   mean_Theta[i][j] = 0.0;
  //    mean_Theta[j][i] = 0.0;
    }
  }
*/
/*Initialize the random variate generators*/
  const gsl_rng_type *T;
  gsl_rng *r;
```

```
  time_t calendar_time;
  gsl_rng_env_setup();
  T = gsl_rng_default;
  r = gsl_rng_alloc (T);
  calendar_time = time(NULL);
  //gsl_rng_set(r,89487);
  gsl_rng_set(r,calendar_time);
  out_status << "Random " << ctime(&calendar_time) << endl;

/*Load the incidence matrix Y and covariates*/
  if(!in_Y){
    out_status << "Could not open Y file. " << endl;
    return 1;
  }
  if(!in_xi){
    out_status << "Could not open xi file. " << endl;
    return 1;
  }

// Reading and inicializing big arrays
  k = 0;
  for(i = 0; i<=n-1 ;i++){
    in_xi >> xi[i];
    // xi[i]=0;
    if(xi[i]+1 > L) L=xi[i]+1;
    for(j = 0; j<=n-1 ;j++){
      in_Y >> *(Y+k);
      pairwise_xi[k] = 0;
      mean_Theta[k] = 0;
```

```
      k++;
    }
  }
  cout << "xi"<<endl;
  display_lineararray_int(n, xi);

  cout << "Y" <<endl;
  for(i =0; i<=20; i++){
    cout << *(Y+(i*n)) << endl;
    cout << pairwise_xi[i*n] << endl;
  }
  in_Y.close();
  in_xi.close();

  calendar_time = time(NULL);
  out_status << "Simulation Starts on " << ctime(&calendar_time)<<endl;

  // Initialize theta
  vector<double> theta(L*L);
/*  for(l = 0; l< L; l++){
    B = extend_B(B,0.5);
}*/
  display_a(theta);

  keep_status=keep_each;
    cout << "last i=" << iter*keep_each+burn-1 <<endl;

      //Simulation #N.iter <- burn + iter * keep_each
  for(i = 0; i<=iter*keep_each+burn-1 ;i++){
```

```
// cout << "i=" << i <<endl;

  llik = update_xi_wllik(n, &L, Y, xi, aD, bD, aOD, bOD, alpha, nu, r, sym);
  update_alpha_nu(&alpha, &nu, n, &L, xi, aalpha, balpha, anu, bnu, r);
  out_llik << llik << "\n";

  if((i>burn-1) && (i-burn+1==keep_status) ){
   // cout<< "inside keep"<<endl;

   //posterior sample for theta;
   theta = sample_Theta(n, L, Y, xi, aD, bD, aOD, bOD, r, sym);

    //saving alpha
    out_alpha << alpha << "\n";
    out_nu << nu << "\n";

  //--- saving B and Update pairwise_xi and mean_Theta (row*num_cols + col)
    for(j = 0; j<=n-1; j++){
     for(k = 0; k<=n-1; k++){
      if(xi[j]==xi[k]){
        pairwise_xi[j*n + k] += 1.0/iter;
      }
       //mean_Theta[j*n + k] += 0.5/iter;
       mean_Theta[j*n + k] += theta[xi[j]*L + xi[k]]/iter;
     }
    }
    keep_status += keep_each;

  }
```

```
  // display_B(B);
  //display_lineararray_int(n, xi);
  //cout << "alpha = " << alpha << endl;

    if(i == print_status){
      calendar_time = time(NULL);
      out_status << "Simulation "<< i << " Ends on " << ctime(&calendar_time)<< endl;
       //saving xi;
      for(j = 0; j<=n-1; j++){
        out_xi << xi[j] << "   ";
      }
      out_xi << "\n";

      print_status += print_each;
    }
  }

//saving pairwise_xi and meanB
  for(j = 0; j<=n-1; j++){
    for(k = 0; k<=n-1; k++){
      out_pairxi << pairwise_xi[j*n + k] << ",";
      out_Theta << mean_Theta[j*n + k] << ",";
    }
    out_pairxi << endl;
    out_Theta << endl;
  }
```

```
  out_xi.close();
  calendar_time = time(NULL);
  out_status << "Simulation Completed on " << ctime(&calendar_time);
  out_status.close();
  out_alpha.close();
  out_nu.close();
  out_llik.close();
  out_pairxi.close();
  out_Theta.close();

  delete [] Y;
  delete [] pairwise_xi;
  delete [] mean_Theta;

}
```

Appendix B

Self-defined Functions

Codes for a variety of self-defined functions

```
#########################################################
#  D.Dahl's (2003) SAMS algorithm     #
#     #
# for Relational Data Model      #
#        #
# Model Y_{i,i',j} | zeta, XI, Theta ~ Bernoulli(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
#
# theta_{lk,j} ~ Beta(a,b)     #
#     #
# NOTE: For computations requires:     #
# source("funcCollections.R")     #
# v.2 update to work with funcCollections.R     #
#  WARNING doesn't work for Poisson models     #
#     #
# By Perla Reyes     #
#   FALL 2010     #
#     v.2 FALL 2011     #
```

```
#####################################################


######    SAMS function  ####################
# makes 1 step of SAMS algorithm
# Input: Y, xi, hyper, direct
# Y array of matrices
# xi unique vector for all matrices in Y
#     hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
# one for each matrix of Y
# Output: new.xi=list(new.xi, Q.new.xi)
#################################################

func.SAMS <- function(Y, xi, hyperD, hyperOD, direct, model){
  n <- dim(Y)[1]
  rpair = sample(1:n, 2)
  a = rpair[1] ## subject index;
  b = rpair[2]

  if( xi[a] == xi[b] ){
    move = func.SPLIT(a, b, Y, xi, hyperD, hyperOD, direct, model)
  }else{
   move = func.MERGE(a, b, Y, xi, hyperD, hyperOD, direct, model)
  }

  new.xi = func.MH_update(move, Y, xi, hyperD, hyperOD, direct, model)

return(new.xi)
```

```
}

###############  MH_update ##################
# computes MH ratio and decides the new state
# Input:  move, Y, xi, hyper, direct
# Y array of matrices
# move = list(S.ab, S.a, S.b, Q, split)
# hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
# one for each matrix of Y
# Output: list(new.xi, Q.new.xi)
################################################

func.MH_update <- function(move, Y, xi, hyperD, hyperOD, direct, model){

  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
  K = max(xi)
  m.a = length(move$S.a)
  m.b = length(move$S.b)

## marg of y's in S.a, S.b and S.ab
  log.pr.a = dmarg.group(move$S.a, Y, xi, hyperD, hyperOD, direct, model, log=T)
  log.pr.b = dmarg.group(move$S.b, Y, xi, hyperD, hyperOD, direct, model, log=T)
  log.pr.ab = dmarg.group(move$S.ab, Y, xi, hyperD, hyperOD, direct, model, log=T)
```

```
# both log.pr.a and log.pr.b count the "ab" and "ba" subgroups, so to "substract" the ex
  log.pr.both = 0
  for(j in 1:J){
   if(model[j]=='B'){
    if(direct[j]){
      log.pr.both = log.pr.both + dmarg.Y.B(m.a*m.b, sum(Y[move$S.a,move$S.b,j]), hyperO
    }else{
      log.pr.both = log.pr.both + dmarg.Y.B(m.a*m.b, sum(Y[move$S.a,move$S.b,j])+sum(Y[m
    }
   }else if(model[j]=='P'){ ## Poisson
   ## NO CODED, YET.
   }
  }

  log.merged.ratio = (log.pr.ab + log(move$Q)) - (lbeta(m.a,m.b) + log(hyperD$eta) + log
  MH.ratio = min( 1, exp(log.merged.ratio*(-1)^move$split) )

  new.xi = xi
  if( rbinom(1, size=1, p=MH.ratio) ) {
   if( move$split ){
     new.xi[move$S.a] = K+1
     new.xi[move$S.b] = K+2
   # do we want to keep track of who is where?
     new.xi = as.numeric(factor(new.xi))
   }else{
     new.xi[move$S.ab] = K+1
     new.xi = as.numeric(factor(new.xi))
   }
```

```
   Q.new.xi <- MH.ratio
  }else{
   Q.new.xi <- 1 - MH.ratio
  }

return(list(new.xi=new.xi, Q.new.xi=Q.new.xi))
}


########  SPLIT  ###################################
# splits a group and saves Pr(split)
# Input: a, b, Y, xi, hyper, direct
# Y array of matrices
#     hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
# one for each matrix of Y
# Output: move = list(S.ab, S.a, S.b, Q, split=TRUE)
#############################################################

func.SPLIT <- function(a, b, Y, xi, hyperD, hyperOD, direct, model){

  xi_ab = xi[a]  ## group index;
  n <- dim(Y)[1]
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
```

```
  S.ab = (1:n)[xi == xi_ab]
  S.a = a;
  S.b = b;
  U = suff.ab(a, b, Y, xi, direct, model)

  xi.rest = xi #keeping track of the others
  xi.rest[S.ab] = 0
  xi.rest[-S.ab] = as.numeric(factor(xi.rest[-S.ab]))
  K.rest = max(xi.rest)

  m.l = c(rep(0,K.rest),1,1) #number of obs per group
         # K.rest+1 & K.rest+2 are the "a" & "b" group, resp
  if(K.rest > 0){
   for(l in 1:K.rest) m.l[l] = sum(xi.rest==l)
  }

  S.ab.small = S.ab[S.ab !=a & S.ab != b]  ## removes a & b
  if (length(S.ab.small) <= 1) permuted.S.ab = S.ab.small else permuted.S.ab = sample(S.
  log.Q = 0

  for (k in permuted.S.ab) {
    pr.ab = split.prob(k, S.a, S.b, U, Y, xi.rest, m.l, hyperD, hyperOD, direct, model)
    pr.a = pr.ab[1]
    pr.b = pr.ab[2]

    if( rbinom(1,p=pr.a, size=1) ){
     ######## k-th obs goes to S.a

#update suff.ab
```

```
for(j in 1:J){
    if(model[j]=='B'){
      if(direct[j]){ ### directed
        if(K.rest > 0){
          for(l in 1:K.rest){
            S.l = (1:n)[xi.rest==l]
           U$a.sn[,l,j] = U$a.sn[,l,j] + m.l[l]
            U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[k,S.l,j])
            U$a.sy[2,l,j] = U$a.sy[2,l,j] + sum(Y[S.l,k,j])
          }
        }
         # the aa subgroup
        l = K.rest+1
        S.l = S.a
       U$a.sn[,l,j] = U$a.sn[,l,j] + 2*m.l[l]
        U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[k,S.l,j]) + sum(Y[S.l,k,j])
        U$a.sy[2,l,j] = U$a.sy[1,l,j]
         # K.rest+2 is the "b" group
        l = K.rest+2
        S.l = S.b
       U$a.sn[,l,j] = U$a.sn[,l,j] + m.l[l]
        U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[k,S.l,j])
        U$a.sy[2,l,j] = U$a.sy[2,l,j] + sum(Y[S.l,k,j])

       # the ab & ba subgroups for "b" suff stats
       U$b.sn[,K.rest+1,j] = U$a.sn[,K.rest+2,j]
       U$b.sy[,K.rest+1,j] = U$a.sy[,K.rest+2,j]

      }else{   ### undirected
```

```
          for(l in 1:(K.rest+2)){
           # K.rest+1, K.rest+2 are the "a", "b" group, resp
            if(l <= K.rest) S.l = (1:n)[xi.rest==l] else if(l == K.rest+1) S.l = S.a els

           U$a.sn[1,l,j] = U$a.sn[1,l,j] + m.l[l]
            U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[S.l,k,j]) + sum(Y[k,S.l,j])
          }
           # the ab subgroup
         U$b.sn[1,K.rest+1,j] = U$a.sn[1,K.rest+2,j]
         U$b.sy[1,K.rest+1,j] = U$a.sy[1,K.rest+2,j]
        }  #end of if(direct)
      }else if(model[j]=='P'){ ## Poisson
   ## NO CODED, YET.
      }
      }  #end of for(j in 1:J)
      m.l[K.rest+1] = m.l[K.rest+1] + 1

      S.a = c(S.a,k)
      log.Q = log.Q + log(pr.a)

    }else{
########### k-th obs goes to S.b
#update suff.ab
  for(j in 1:J){
      if(model[j]=='B'){
        if(direct[j]){ ### directed
          for(l in 1:(K.rest+1)){
         # K.rest+1 is the "a" group
            if(l <= K.rest) S.l = (1:n)[xi.rest==l] else S.l = S.a
```

```
            U$b.sn[,l,j] = U$b.sn[,l,j] + m.l[l]
             U$b.sy[1,l,j] = U$b.sy[1,l,j] + sum(Y[k,S.l,j])
             U$b.sy[2,l,j] = U$b.sy[2,l,j] + sum(Y[S.l,k,j])
           }
# the bb subgroup
  l = K.rest+2
  S.l = S.b
  U$b.sn[,l,j] = U$b.sn[,l,j] + 2*m.l[l]
          U$b.sy[1,l,j] = U$b.sy[1,l,j] + sum(Y[k,S.l,j]) + sum(Y[S.l,k,j])
          U$b.sy[2,l,j] = U$b.sy[1,l,j]

         # the ab & ba subgroups change "a" suff stats
         U$a.sn[,K.rest+2,j] = U$b.sn[,K.rest+1,j]
         U$a.sy[,K.rest+2,j] = U$b.sy[,K.rest+1,j]

        }else{   ### undirected
          for(l in 1:(K.rest+2)){
           # K.rest+1, K.rest+2 are the "a", "b" group, resp
            if(l <= K.rest) S.l = (1:n)[xi.rest==l] else if(l == K.rest+1) S.l = S.a els

            U$b.sn[1,l,j] = U$b.sn[1,l,j] + m.l[l]
             U$b.sy[1,l,j] = U$b.sy[1,l,j] + sum(Y[S.l,k,j]) + sum(Y[k,S.l,j])
           }
           # the ab subgroup
         U$a.sn[1,K.rest+2,j] = U$b.sn[1,K.rest+1,j]
         U$a.sy[1,K.rest+2,j] = U$b.sy[1,K.rest+1,j]
        }  #end of if(direct)
      }else if(model[j]=='P'){ ## Poisson
```

```
   ## NO CODED, YET.
      }
      } #end of for(j in 1:J)
      m.l[K.rest+2] = m.l[K.rest+2] + 1
      S.b = c(S.b,k)
      log.Q = log.Q + log(pr.b)
    } #end of if for allocation

  } # end of for (k in permuted.S.ab)
  pr.Q <- exp(log.Q)

return(move = list(S.ab = S.ab, S.a = S.a, S.b = S.b, Q=pr.Q,  split=TRUE))
}


############    MERGE    ########################
# merges groups and computes Pr(imaginary split)
# Input: a, b, Y, xi, hyper, direct
# hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
# one for each matrix of Y
# Output: move = list(S.ab, S.a, S.b, Q, split=FALSE)
################################################

func.MERGE <- function(a, b, Y, xi, hyperD, hyperOD, direct, model){

  xi_a = xi[a]
  xi_b = xi[b]
  n <- dim(Y)[1]
```

```
if(length(dim(Y))<3){
  J <- 1
  Y <- array(Y,dim=c(n,n,J))
}else{
  J <- dim(Y)[3]
}
S.a = (1:n)[xi == xi_a]
S.b = (1:n)[xi == xi_b]
S.ab = c(S.a,S.b)

U = suff.ab(a, b, Y, xi, direct, model)

xi.rest = xi #keeping track of the others
xi.rest[S.ab] = 0
xi.rest[-S.ab] = as.numeric(factor(xi.rest[-S.ab]))
K.rest = max(xi.rest)

m.l = c(rep(0,K.rest),1,1) #number of obs per group
       # K.rest+1 & K.rest+2 are the "a" & "b" group, resp
if(K.rest > 0){
 for(l in 1:K.rest) m.l[l] = sum(xi.rest==l)
}

Hyp.S.ab = S.ab[S.ab !=a & S.ab != b]  ## removes a & b
if (length(Hyp.S.ab)<=1) permuted.S.ab = Hyp.S.ab else permuted.S.ab = sample(Hyp.S.ab
log.Q = 0
Hyp.S.a = a
Hyp.S.b = b
```

```
  for (k in permuted.S.ab) {

    pr.ab = split.prob(k, Hyp.S.a, Hyp.S.b, U, Y, xi.rest, m.l, hyperD, hyperOD, direct,
    pr.a = pr.ab[1]
    pr.b = pr.ab[2]

    if( sum(k == S.a) > 0 ){
     ######## k-th obs goes to Hyp.S.a

#update suff.ab
  for(j in 1:J){
   if(model[j]=='B'){
        if(direct[j]){ ### directed
          if(K.rest > 0){
            for(l in 1:K.rest){
              S.l = (1:n)[xi.rest==l]
             U$a.sn[,l,j] = U$a.sn[,l,j] + m.l[l]
              U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[k,S.l,j])
              U$a.sy[2,l,j] = U$a.sy[2,l,j] + sum(Y[S.l,k,j])
            }
          }
           # the aa subgroup
          l = K.rest+1
          S.l = Hyp.S.a
         U$a.sn[,l,j] = U$a.sn[,l,j] + 2*m.l[l]
          U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[k,S.l,j]) + sum(Y[S.l,k,j])
          U$a.sy[2,l,j] = U$a.sy[1,l,j]
           # K.rest+2 is the "b" group
          l = K.rest+2
```

```
        S.l = Hyp.S.b
       U$a.sn[,l,j] = U$a.sn[,l,j] + m.l[l]
        U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[k,S.l,j])
        U$a.sy[2,l,j] = U$a.sy[2,l,j] + sum(Y[S.l,k,j])

       # the ab & ba subgroups for "b" suff stats
       U$b.sn[,K.rest+1,j] = U$a.sn[,K.rest+2,j]
       U$b.sy[,K.rest+1,j] = U$a.sy[,K.rest+2,j]

      }else{   ### undirected
        for(l in 1:(K.rest+2)){
         # K.rest+1, K.rest+2 are the "a", "b" group, resp
          if(l <= K.rest) S.l = (1:n)[xi.rest==l] else if(l == K.rest+1) S.l = Hyp.S.a

         U$a.sn[1,l,j] = U$a.sn[1,l,j] + m.l[l]
          U$a.sy[1,l,j] = U$a.sy[1,l,j] + sum(Y[S.l,k,j]) + sum(Y[k,S.l,j])
        }
         # the ab subgroup
       U$b.sn[1,K.rest+1,j] = U$a.sn[1,K.rest+2,j]
       U$b.sy[1,K.rest+1,j] = U$a.sy[1,K.rest+2,j]
      }  #end of if(direct)
   }else if(model[j]=='P'){ ## Poisson
## NO CODED, YET.
   }
   } #end of for(j in 1:J)
   m.l[K.rest+1] = m.l[K.rest+1] + 1

   Hyp.S.a = c(Hyp.S.a,k)
   log.Q = log.Q + log(pr.a)
```

```
    }else{
########### k-th obs goes to Hyp.S.b
#update suff.ab
  for(j in 1:J){
      if(model[j]=='B'){
        if(direct[j]){ ### directed
          for(l in 1:(K.rest+1)){
         # K.rest+1 is the "a" group
            if(l <= K.rest) S.l = (1:n)[xi.rest==l] else S.l = Hyp.S.a

           U$b.sn[,l,j] = U$b.sn[,l,j] + m.l[l]
            U$b.sy[1,l,j] = U$b.sy[1,l,j] + sum(Y[k,S.l,j])
            U$b.sy[2,l,j] = U$b.sy[2,l,j] + sum(Y[S.l,k,j])
          }
# the bb subgroup
  l = K.rest+2
  S.l = Hyp.S.b
  U$b.sn[,l,j] = U$b.sn[,l,j] + 2*m.l[l]
          U$b.sy[1,l,j] = U$b.sy[1,l,j] + sum(Y[k,S.l,j]) + sum(Y[S.l,k,j])
          U$b.sy[2,l,j] = U$b.sy[1,l,j]

         # the ab & ba subgroups change "a" suff stats
         U$a.sn[,K.rest+2,j] = U$b.sn[,K.rest+1,j]
         U$a.sy[,K.rest+2,j] = U$b.sy[,K.rest+1,j]

        }else{   ### undirected
          for(l in 1:(K.rest+2)){
           # K.rest+1, K.rest+2 are the "a", "b" group, resp
```

```
            if(l <= K.rest) S.l = (1:n)[xi.rest==l] else if(l == K.rest+1) S.l = Hyp.S.a

           U$b.sn[1,l,j] = U$b.sn[1,l,j] + m.l[l]
            U$b.sy[1,l,j] = U$b.sy[1,l,j] + sum(Y[S.l,k,j]) + sum(Y[k,S.l,j])
          }
           # the ab subgroup
         U$a.sn[1,K.rest+2,j] = U$b.sn[1,K.rest+1,j]
         U$a.sy[1,K.rest+2,j] = U$b.sy[1,K.rest+1,j]
       }  #end of if(direct)
     }else if(model[j]=='P'){ ## Poisson
  ## NO CODED, YET.
     }
     } #end of for(j in 1:J)
     m.l[K.rest+2] = m.l[K.rest+2] + 1
     Hyp.S.b = c(Hyp.S.b,k)
     log.Q = log.Q + log(pr.b)
   } #end of if for allocation

 } # end of for (k in permuted.S.ab)
 pr.Q <- exp(log.Q)

return(move = list(S.ab = S.ab, S.a = S.a, S.b = S.b, Q=pr.Q, split=FALSE))
}

######## dmarg.group  ###################################
# density function of the joint marginal of the "k" group
#  i.e. Pr(Y_ij: i or j \in S.k )
# Input: S.k, Y, xi, hyper, direct, log = FALSE
# Y array of matrices
```

```
# hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
# one for each matrix of Y
# log = TRUE the output is the log(density)
# Output: density
#########################################################

dmarg.group <- function(S.k, Y, xi, hyperD, hyperOD, direct, model, log = FALSE){
  n <- dim(Y)[1]
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
  xi.rest = xi #keeping track of the others
  xi.rest[S.k] = 0
  xi.rest[-S.k] = as.numeric(factor(xi.rest[-S.k]))
  K.rest = max(xi.rest)

  U = suff.group(S.k, Y, xi, direct, model)

  log.pr.Sk = 0

#find the prob per group
  for(j in 1:J){
   if(model[j]=='B'){ ## Binary
    if(direct[j]){ ### directed
     if(K.rest > 0){
```

```
        for(l in 1:K.rest){ # for the other groups
          log.pr.Sk = log.pr.Sk + dmarg.Y.B(U$k.sn[1,l,j], U$k.sy[1,l,j], hyperOD, log=T
# xi_k,l (row) + l,xi_k (col)
        }
      }
         # K.rest+1 is the "k" group
      l = K.rest+1
      log.pr.Sk = log.pr.Sk + dmarg.Y.B(U$k.sn[1,l,j], U$k.sy[1,l,j], hyperD, log=T)
# kk subgroup

    }else{   ### undirected
      for(l in 1:(K.rest+1)){   # K.rest+1 is the "k" group
       if(l <= K.rest) hyper=hyperOD else hyper=hyperD;
        log.pr.Sk = log.pr.Sk + dmarg.Y.B(U$k.sn[1,l,j], U$k.sy[1,l,j], hyper, log=T)
  }
    } #end of if(direct)
   }else if(model[j]=='P'){ ## Poisson
   ## NO CODED, YET.
   }
  }  #end of for(j in 1:J)

  density = if(log) log.pr.Sk else exp(log.pr.Sk)

return(density)
}



######## split.prob  ###################################
# computes the prob's of "k" to join S.a or S.b
```

```
# Input: k, S.a, S.b, U, Y, xi.rest, m.l, hyper, direct
# Y array of matrices
# U = list(a.sn=a.sn, a.sy=a.sy, b.sn=b.sn, b.sy=b.sy)
# hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
# one for each matrix of Y
# Output: c(pr.a, pr.b)
#########################################################

split.prob <- function(k, S.a, S.b, U, Y, xi.rest, m.l, hyperD, hyperOD, direct, model){
  n <- dim(Y)[1]
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
  K.rest <- max(xi.rest)

  log.pr.Sa = log(length(S.a)) # p(y[k] | y[S.a])
  log.pr.Sb = log(length(S.b)) # p(y[k] | y[S.b])

#find the prob per group
 for(j in 1:J){
  if(model[j]=='B'){ ## Binary
   if(direct[j]){ ### directed
    if(K.rest > 0){
       for(l in 1:K.rest){ # for the other groups
         S.l = (1:n)[xi.rest==l]
```

```
        log.pr.Sa = log.pr.Sa + dmarg.Y.B(U$a.sn[1,l,j] + m.l[l], U$a.sy[1,l,j] + sum(Y
# xi_k,l (row)
        log.pr.Sa = log.pr.Sa + dmarg.Y.B(U$a.sn[2,l,j] + m.l[l], U$a.sy[2,l,j] + sum(Y
# l,xi_k (col)

        log.pr.Sb = log.pr.Sb + dmarg.Y.B(U$b.sn[1,l,j] + m.l[l], U$b.sy[1,l,j] + sum(Y
# xi_k,l (row)
        log.pr.Sb = log.pr.Sb + dmarg.Y.B(U$b.sn[2,l,j] + m.l[l], U$b.sy[2,l,j] + sum(Y
# l,xi_k (col)
   }
     }
        # K.rest+1 is the "a" group
     l = K.rest+1
     S.l = S.a
        # aa subgroup
     log.pr.Sa = log.pr.Sa + dmarg.Y.B(U$a.sn[1,l,j] + 2*m.l[l], U$a.sy[1,l,j] + sum(Y[S
        # ab and ba subgroups
     log.pr.Sb = log.pr.Sb + dmarg.Y.B(U$b.sn[1,l,j] + m.l[l], U$b.sy[1,l,j] + sum(Y[k,S
# xi_k,l (row)
    log.pr.Sb = log.pr.Sb + dmarg.Y.B(U$b.sn[2,l,j] + m.l[l], U$b.sy[2,l,j] + sum(Y[S.l,
# l,xi_k (col)
        # K.rest+1, K.rest+2 are the "a", "b" group, resp
     l = K.rest+2
     S.l = S.b
        # ab and ba subgroups
     log.pr.Sa = log.pr.Sa + dmarg.Y.B(U$a.sn[1,l,j] + m.l[l], U$a.sy[1,l,j] + sum(Y[k,S
# xi_k,l (row)
     log.pr.Sa = log.pr.Sa + dmarg.Y.B(U$a.sn[2,l,j] + m.l[l], U$a.sy[2,l,j] + sum(Y[S.l
```

```
# l,xi_k (col)
         # bb subgroup
     log.pr.Sb = log.pr.Sb + dmarg.Y.B(U$b.sn[1,l,j] + 2*m.l[l], U$b.sy[1,l,j] + sum(Y[S

   }else{   ### undirected
     for(l in 1:(K.rest+2)){
         # K.rest+1, K.rest+2 are the "a", "b" group, resp
       if(l <= K.rest){
         S.l = (1:n)[xi.rest==l];
         hyper.a = hyperOD;
         hyper.b = hyperOD;
       }else if(l == K.rest+1){
         S.l = S.a;
         hyper.a = hyperD;
         hyper.b = hyperOD;
       }else{
     S.l = S.b;
     hyper.a = hyperOD;
         hyper.b = hyperD;
       }


       log.pr.Sa = log.pr.Sa + dmarg.Y.B(U$a.sn[1,l,j] + m.l[l], U$a.sy[1,l,j] + sum(Y[S


       log.pr.Sb = log.pr.Sb + dmarg.Y.B(U$b.sn[1,l,j] + m.l[l], U$b.sy[1,l,j] + sum(Y[S
 }
   } #end of if(direct)
  }else if(model[j]=='P'){ ## Poisson
   ## NO CODED, YET.
```

```
  }
} #end of for(j in 1:J)

  log.pr = c(log.pr.Sa,log.pr.Sb)
  pr = exp(log.pr -  max(log.pr))
  pr.a = pr[1]/sum(pr)
  pr.b = 1- pr.a

return(prob.ab=c(pr.a,pr.b))
}




#
# File: funcCollections.R
#
# Data: 01/27/2011
# Modification to include different models for Y_{i,i',j}
#
# Date: 12/01/2010
# Complete list of functions to run Sequential Allocation for
# a collection of adjacency matrices
#

#####################################################################
#####################################################################
#
# File: funcSuffStats.R
#
```

```
# Functions to compute and update sufficient stats for a collection of adjacency matrice
#
# Model
#  Binomial Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Bernoulli(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Beta(hyperD$.a.bin,hyperD$b.bin)   l=k
#    Beta(hyperOD$.a.bin,hyperOD$b.bin) l \neq k
#
#  Poisson Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Poisson(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Gamma(hyperD$a.pois,hyperD$b.pois)   l=k
#    Gamma(hyperOD$a.pois,hyperOD$b.pois) l \neq k

############    suff.xi    ######################
# Computes the matrices of suff stats by group
#
# Input: Y, xi, direct, model
# Y array of matrices
# if xi has zeros those actors and not considered
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: list(U.sn, U.sy, U.pf) // n and sum of y's
# and log(prod y!) if Poisson//
#################################################

suff.xi <- function(Y,xi,direct,model){
  n <- dim(Y)[1]
  if(length(dim(Y))<3){
```

```
  J <- 1
  Y <- array(Y,dim=c(n,n,J))
 }else{
  J <- dim(Y)[3]
 }
 K <- max(xi)

 U.sn = array(0,dim=c(K,K,J))
 U.sy = array(0,dim=c(K,K,J))
 if(sum(model=='P')>0) U.lpf = array(0,dim=c(K,K,J))

for(j in 1:J){
 if(model[j]=='B'){
  if(direct[j]){ ### directed
   if(K != 1){
   for( l in 1:(K-1) ){
     m_l = sum(xi==l)
      U.sn[l,l,j] = m_l*(m_l-1)
      U.sy[l,l,j] = sum(Y[xi==l,xi==l,j])
      for( k in (l+1):K ){
    m_k = sum(xi==k)
        U.sn[l,k,j] = U.sn[k,l,j] = m_l*m_k
        U.sy[l,k,j] = sum(Y[xi==l,xi==k,j])
        U.sy[k,l,j] = sum(Y[xi==k,xi==l,j])
      }
    }
    # the first "for" need to stop earlier than the second
    m_l = sum(xi==K)
    U.sn[K,K,j] = m_l*(m_l-1)
```

```
   U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
  }else{ ### K = 1
   m_l = sum(xi==K)
   U.sn[K,K,j] = m_l*(m_l-1)
   U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
  }
 }else{   ### undirected
  if(K != 1){
   for( l in 1:(K-1) ){
    m_l = sum(xi==l)
    U.sn[l,l,j] = m_l*(m_l-1)/2
    U.sy[l,l,j] = sum(Y[xi==l,xi==l,j])
     for( k in (l+1):K ){
   m_k = sum(xi==k)
       U.sn[l,k,j] = m_l*m_k
       U.sy[l,k,j] = sum(Y[xi==l,xi==k,j]) + sum(Y[xi==k,xi==l,j])
     }
   }
   # the first "for" need to stop earlier than the second
   m_l = sum(xi==K)
   U.sn[K,K,j] = m_l*(m_l-1)/2
   U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
  }else{ ### K = 1
   m_l = sum(xi==K)
   U.sn[K,K,j] = m_l*(m_l-1)/2
   U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
  }
 } ## end of if(direct[j])
}else if(model[j]=='P'){
```

```
if(direct[j]){ ### directed
 if(K != 1){
 for( l in 1:(K-1) ){
   m_l = sum(xi==l)
    U.sn[l,l,j] = m_l*(m_l-1)
    U.sy[l,l,j] = sum(Y[xi==l,xi==l,j])
    U.lpf[l,l,j] = sum(lfactorial(Y[xi==l,xi==l,j]))
    for( k in (l+1):K ){
  m_k = sum(xi==k)
      U.sn[l,k,j] = U.sn[k,l,j] = m_l*m_k
      U.sy[l,k,j] = sum(Y[xi==l,xi==k,j])
      U.sy[k,l,j] = sum(Y[xi==k,xi==l,j])
      U.lpf[l,k,j] = sum(lfactorial(Y[xi==l,xi==k,j]))
      U.lpf[k,l,j] = sum(lfactorial(Y[xi==k,xi==l,j]))
    }
  }
  # the first "for" need to stop earlier than the second
  m_l = sum(xi==K)
  U.sn[K,K,j] = m_l*(m_l-1)
  U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
  U.lpf[K,K,j] = sum(lfactorial(Y[xi==K,xi==K,j]))
 }else{ ### K = 1
  m_l = sum(xi==K)
  U.sn[K,K,j] = m_l*(m_l-1)
  U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
  U.lpf[K,K,j] = sum(lfactorial(Y[xi==K,xi==K,j]))
 }
}else{   ### undirected
 if(K != 1){
```

```
    for( l in 1:(K-1) ){
     m_l = sum(xi==l)
     U.sn[l,l,j] = m_l*(m_l-1)/2
     U.sy[l,l,j] = sum(Y[xi==l,xi==l,j])
      U.lpf[l,l,j] = sum(lfactorial(Y[xi==l,xi==l,j]))
      for( k in (l+1):K ){
    m_k = sum(xi==k)
        U.sn[l,k,j] = m_l*m_k
        U.sy[l,k,j] = sum(Y[xi==l,xi==k,j]) + sum(Y[xi==k,xi==l,j])
        U.lpf[l,k,j] = sum(lfactorial(Y[xi==l,xi==k,j])) + sum(lfactorial(Y[xi==k,xi==l
      }
    }
    # the first "for" need to stop earlier than the second
    m_l = sum(xi==K)
    U.sn[K,K,j] = m_l*(m_l-1)/2
    U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
    U.lpf[K,K,j] = sum(lfactorial(Y[xi==K,xi==K,j]))
   }else{ ### K = 1
    m_l = sum(xi==K)
    U.sn[K,K,j] = m_l*(m_l-1)/2
    U.sy[K,K,j] = sum(Y[xi==K,xi==K,j])
    U.lpf[K,K,j] = sum(lfactorial(Y[xi==K,xi==K,j]))
   }
  } ## end of if(direct[j])
 } ## end of Poisson
} ## end of for(j in 1:J)

if(sum(model=='P')>0) output=list(U.sn=U.sn, U.sy=U.sy, U.lpf=U.lpf) else output=list(U
```

```
return(output)
}


############    update.suff.xi_i    ########################
#   Updadtes the sufficient statistics
# once xi_i has been updated to value "ind"
# i.e., actor "i" joins group "ind"
#
# Input: Zz, ind, direct, model
# Zz=list(Z.sn, Z.sy, z.sn, z.sy)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: list( U.sn, U.sy, U.lpf) // n and sum of y's //
# and log(prod y!) if Poisson//
################################################

update.suff.xi_i <- function(Zz, ind, direct, model){
# ind = 2
  K.n <- dim(Zz$Z.sn)[1]
  Pois = sum(model=='P')>0

  if(length(dim(Zz$Z.sn))<3){
    J <- 1
    Z.sn <- array(Zz$Z.sn,dim=c(K.n,K.n,J))
    Z.sy <- array(Zz$Z.sy,dim=c(K.n,K.n,J))
    z.sn <- array(Zz$z.sy,dim=c(2,K.n,J))
    z.sy <- array(Zz$z.sy,dim=c(2,K.n,J))
```

```
   if(Pois){
Z.lpf <- array(Zz$Z.lpf,dim=c(K.n,K.n,J))
    z.lpf <- array(Zz$z.lpf,dim=c(2,K.n,J))
   }
 }else{
   J <- dim(Zz$Z.sn)[3]
   Z.sn = Zz$Z.sn
   Z.sy = Zz$Z.sy
   z.sn = Zz$z.sn
   z.sy = Zz$z.sy
   if(Pois){
     Z.lpf = Zz$Z.lpf
     z.lpf = Zz$z.lpf
   }
 }

 if(ind > K.n){
   U.sn = array(0,dim=c(ind,ind,J))
   U.sy = array(0,dim=c(ind,ind,J))
   if(Pois) U.lpf = array(0,dim=c(ind,ind,J))
   for(j in 1:J){
     U.sn[1:K.n,1:K.n,j] = Z.sn[,,j]
     U.sy[1:K.n,1:K.n,j] = Z.sy[,,j]
     if(model[j]=='B'){
      if(direct[j]){
       U.sn[ind,,j] = c(z.sn[1,,j],0)  # xi_i,l block (row)
       U.sy[ind,,j] = c(z.sy[1,,j],0)
       U.sn[,ind,j] = c(z.sn[2,,j],0)  # l,xi_i block (col)
       U.sy[,ind,j] = c(z.sy[2,,j],0)
```

```
     }else{
      U.sn[,ind,j] = c(z.sn[1,,j],0)  # l,xi_i block (col)
      U.sy[,ind,j] = c(z.sy[1,,j],0)
     }
    }else if(model[j]=='P'){
     U.lpf[1:K.n,1:K.n,j] = Z.lpf[,,j]
     if(direct[j]){
      U.sn[ind,,j] = c(z.sn[1,,j],0)  # xi_i,l block (row)
      U.sy[ind,,j] = c(z.sy[1,,j],0)
      U.lpf[ind,,j] = c(z.lpf[1,,j],0)
      U.sn[,ind,j] = c(z.sn[2,,j],0)  # l,xi_i block (col)
      U.sy[,ind,j] = c(z.sy[2,,j],0)
      U.lpf[,ind,j] = c(z.lpf[2,,j],0)
     }else{
      U.sn[,ind,j] = c(z.sn[1,,j],0)  # l,xi_i block (col)
      U.sy[,ind,j] = c(z.sy[1,,j],0)
      U.lpf[,ind,j] = c(z.lpf[1,,j],0)
     }

    } #end of Poisson
  }
}else{
  U.sn = Z.sn #need to change xi_i-th row and col
  U.sy = Z.sy
  if(Pois) U.lpf = Z.lpf
  for(j in 1:J){
   if(model[j]=='B'){
    if(direct[j]){
      for(l in 1:K.n){
```

```
      # xi_i,l block (row)
     U.sn[ind,l,j] = U.sn[ind,l,j] + z.sn[1,l,j]
     U.sy[ind,l,j] = U.sy[ind,l,j] + z.sy[1,l,j]
        # l,xi_i block (col)
     U.sn[l,ind,j] = U.sn[l,ind,j] + z.sn[2,l,j]
     U.sy[l,ind,j] = U.sy[l,ind,j] + z.sy[2,l,j]
   }
 }else{
   for(l in 1:K.n){
    r.ind = min(ind,l)
    c.ind = max(ind,l)
#affecting the l,xi_i block
    U.sn[r.ind,c.ind,j] = U.sn[r.ind,c.ind,j] + z.sn[1,l,j]
    U.sy[r.ind,c.ind,j] = U.sy[r.ind,c.ind,j] + z.sy[1,l,j]
   }
 }
}else if(model[j]=='P'){
 if(direct[j]){
   for(l in 1:K.n){
      # xi_i,l block (row)
     U.sn[ind,l,j] = U.sn[ind,l,j] + z.sn[1,l,j]
     U.sy[ind,l,j] = U.sy[ind,l,j] + z.sy[1,l,j]
     U.lpf[ind,l,j] = U.lpf[ind,l,j] + z.lpf[1,l,j]
        # l,xi_i block (col)
     U.sn[l,ind,j] = U.sn[l,ind,j] + z.sn[2,l,j]
     U.sy[l,ind,j] = U.sy[l,ind,j] + z.sy[2,l,j]
     U.lpf[l,ind,j] = U.lpf[l,ind,j] + z.lpf[2,l,j]
   }
 }else{
```

```
       for(l in 1:K.n){
        r.ind = min(ind,l)
        c.ind = max(ind,l)
     #affecting the l,xi_i block
        U.sn[r.ind,c.ind,j] = U.sn[r.ind,c.ind,j] + z.sn[1,l,j]
        U.sy[r.ind,c.ind,j] = U.sy[r.ind,c.ind,j] + z.sy[1,l,j]
        U.lpf[r.ind,c.ind,j] = U.lpf[r.ind,c.ind,j] + z.lpf[1,l,j]
       }
      }
     } # end of Poisson
    } # end of for(j in 1:J)
  } # end of if(ind > K.n)

 if(Pois) output=list(U.sn=U.sn, U.sy=U.sy, U.lpf=U.lpf) else output=list(U.sn=U.sn, U.s

return(output)
}

############    suff.xi_i    ###################################
# updadtes the sufficient statistics for xi_i | xi^(i)
# takes xi_i out from the count and creates vectors for xi_i,l
#
# Input: Y, xi, U, i, direct,model
# U=list(U.sn, U.sy)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: list(Z.sn, Z.sy, Z.lpf, z.sn, z.sy, z.lpf)
# // n and sum of y's and log(prod y!) if Poisson//
```

```
#################################################################

suff.xi_i <- function(Y, xi, U, i, direct, model){
#i = 4
  xi_i = xi[i]
  xi.n = xi
  xi.n[i] = 0
  xi.n[-i] = as.numeric(factor(xi.n[-i]))
  K = max(xi)
  K.n = max(xi.n)
  Pois = sum(model=='P')>0

  if(length(dim(U$U.sn))<3){
   J <- 1
    U.sn <- array(U$U.sn,dim=c(K,K,J))
    U.sy <- array(U$U.sy,dim=c(K,K,J))
    if(Pois) U.lpf <- array(U$U.lpf,dim=c(K,K,J))

  }else{
   J <- dim(U$U.sn)[3]
    U.sn = U$U.sn
    U.sy = U$U.sy
    if(Pois) U.lpf <- U$U.lpf
  }

  Z.sn = array(0, dim=c(K.n,K.n,J))
  Z.sy = array(0, dim=c(K.n,K.n,J))
  z.sn = array(0, dim=c(2,K.n,J))
  z.sy = array(0, dim=c(2,K.n,J))
```

```
if(Pois){
  Z.lpf = array(0, dim=c(K.n,K.n,J))
  z.lpf = array(0, dim=c(2,K.n,J))
}

if(K.n < K){
 for(j in 1:J){
    Z.sn[1:K.n,1:K.n,j] = U.sn[-xi_i,-xi_i,j]
    Z.sy[1:K.n,1:K.n,j] = U.sy[-xi_i,-xi_i,j]

    if(model[j]=='B'){    ## Binary
     if(direct[j]){ ## directed
      for(k in 1:K.n){
       l = ifelse(k>=xi_i,k+1,k) # picking the right column from U
       z.sn[1,k,j] = U.sn[xi_i,l,j]  # xi_i,l block (row)
       z.sy[1,k,j] = U.sy[xi_i,l,j]
       z.sn[2,k,j] = U.sn[l,xi_i,j]  # l,xi_i block (col)
       z.sy[2,k,j] = U.sy[l,xi_i,j]
       z.sy[2,k,j] = U.sy[l,xi_i,j]
      }
     }else{ ## undirected
      for(k in 1:K.n){
       l = ifelse(k>=xi_i,k+1,k) # picking the right row || column
       z.sn[1,k,j] = U.sn[xi_i,l,j] + U.sn[l,xi_i,j]
       z.sy[1,k,j] = U.sy[xi_i,l,j] + U.sy[l,xi_i,j]
      }
     }
    }else if(model[j]=='P'){ ## Poisson
     Z.lpf[1:K.n,1:K.n,j] = U.lpf[-xi_i,-xi_i,j]
```

```
      if(direct[j]){ ## directed
       for(k in 1:K.n){
        l = ifelse(k>=xi_i,k+1,k) # picking the right column from U
        z.sn[1,k,j] = U.sn[xi_i,l,j]  # xi_i,l block (row)
        z.sy[1,k,j] = U.sy[xi_i,l,j]
        z.lpf[1,k,j] = U.lpf[xi_i,l,j]
        z.sn[2,k,j] = U.sn[l,xi_i,j]  # l,xi_i block (col)
        z.sy[2,k,j] = U.sy[l,xi_i,j]
        z.lpf[2,k,j] = U.lpf[l,xi_i,j]
       }
      }else{ ## undirected
       for(k in 1:K.n){
        l = ifelse(k>=xi_i,k+1,k) # picking the right row || column
        z.sn[1,k,j] = U.sn[xi_i,l,j] + U.sn[l,xi_i,j]
        z.sy[1,k,j] = U.sy[xi_i,l,j] + U.sy[l,xi_i,j]
        z.lpf[1,k,j] = U.lpf[xi_i,l,j] + U.lpf[l,xi_i,j]
       }
     }
}  #end of POISSON
   } # end of for(j in 1:J)
 }else{
  for(j in 1:J){
   Z.sn[,,j] = U.sn[,,j] #need to change xi_i-th row and col
    Z.sy[,,j] = U.sy[,,j]
    if(model[j]=='B'){ #Binary
     if(direct[j]){
       for(l in 1:K.n){
  z.sn[,l,j] <- sum(xi.n==l) #same n for xi_i,l and l,xi_i blocks
        z.sy[1,l,j] = sum(Y[i,xi.n==l,j])  # xi_i,l block (row)
```

```
      Z.sn[xi_i,l,j] = Z.sn[xi_i,l,j] - z.sn[1,l,j]
      Z.sy[xi_i,l,j] = Z.sy[xi_i,l,j] - z.sy[1,l,j]

      z.sy[2,l,j] = sum(Y[xi.n==l,i,j])  # l,xi_i block (col)
      Z.sn[l,xi_i,j] = Z.sn[l,xi_i,j] - z.sn[2,l,j]
      Z.sy[l,xi_i,j] = Z.sy[l,xi_i,j] - z.sy[2,l,j]
     }
   }else{ ## undirected
    for(l in 1:K.n){
      r.ind = min(xi_i,l)
      c.ind = max(xi_i,l)
#affecting the l,xi_i block
      z.sn[1,l,j] = sum(xi.n==l)
      z.sy[1,l,j] = sum(Y[i,xi.n==l,j]) + sum(Y[xi.n==l,i,j])
      Z.sn[r.ind,c.ind,j] = Z.sn[r.ind,c.ind,j] - z.sn[1,l,j]
      Z.sy[r.ind,c.ind,j] = Z.sy[r.ind,c.ind,j] - z.sy[1,l,j]
    }
   }
  }else if(model[j]=='P'){ ## Poisson
    Z.lpf[,,j] = U.lpf[,,j]
    if(direct[j]){
     for(l in 1:K.n){
z.sn[,l,j] <- sum(xi.n==l) #same n for xi_i,l and l,xi_i blocks
      z.sy[1,l,j] = sum(Y[i,xi.n==l,j])  # xi_i,l block (row)
      z.lpf[1,l,j] = sum(lfactorial(Y[i,xi.n==l,j]))
  Z.sn[xi_i,l,j] = Z.sn[xi_i,l,j] - z.sn[1,l,j]
      Z.sy[xi_i,l,j] = Z.sy[xi_i,l,j] - z.sy[1,l,j]
      Z.lpf[xi_i,l,j] = Z.lpf[xi_i,l,j] - z.lpf[1,l,j]
```

```
        z.sy[2,l,j] = sum(Y[xi.n==l,i,j])  # l,xi_i block (col)
        z.lpf[2,l,j] = sum(lfactorial(Y[xi.n==l,i,j]))
        Z.sn[l,xi_i,j] = Z.sn[l,xi_i,j] - z.sn[2,l,j]
        Z.sy[l,xi_i,j] = Z.sy[l,xi_i,j] - z.sy[2,l,j]
        Z.lpf[l,xi_i,j] = Z.lpf[l,xi_i,j] - z.lpf[2,l,j]
       }
     }else{ ## undirected
      for(l in 1:K.n){
       r.ind = min(xi_i,l)
       c.ind = max(xi_i,l)
  #affecting the l,xi_i block
       z.sn[1,l,j] = sum(xi.n==l)
       z.sy[1,l,j] = sum(Y[i,xi.n==l,j]) + sum(Y[xi.n==l,i,j])
       z.lpf[1,l,j] = sum(lfactorial(Y[i,xi.n==l,j])) + sum(lfactorial(Y[xi.n==l,i,j]))
       Z.sn[r.ind,c.ind,j] = Z.sn[r.ind,c.ind,j] - z.sn[1,l,j]
       Z.sy[r.ind,c.ind,j] = Z.sy[r.ind,c.ind,j] - z.sy[1,l,j]
       Z.lpf[r.ind,c.ind,j] = Z.lpf[r.ind,c.ind,j] - z.lpf[1,l,j]
      }
     }


    } #end of Poisson
   } # end of for(j in 1:J)
  } # end of if(K.n < K)

 if(Pois) output=list(Z.sn=Z.sn, Z.sy=Z.sy, Z.lpf=Z.lpf, z.sn=z.sn, z.sy=z.sy, z.lpf=z.l

return(output)
}
```

```
############    suff.ab   ##############################
#  Computes the sufficient statistics for the initial split
# when S.a only has a, and S.b only has b on it.
#  In the output the columns are each group, the a and b groups
# are the second to last and the last column, respectively.
#  The first row is a,l (row) subgroups
# and the second is l,a (col) subgroups.
#
# Input: a, b. Y, xi, direct,model
# Y array of matrices
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: list(a.sn, a.sy, a.lpf, b.sn, b.sy, b.lpf)
# // n and sum of y's for a and b//
# // and log(prod y!) if Poisson//
#################################################################

suff.ab <- function(a, b, Y, xi, direct,model){
#i = 4
  xi_a = xi[a]
  xi_b = xi[b]
  n <- dim(Y)[1]
  Pois = sum(model=='P')>0
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
```

```
 }
 S.ab = (1:n)[xi == xi_a | xi == xi_b]

 xi.rest = xi
 xi.rest[S.ab] = 0
 xi.rest[-S.ab] = as.numeric(factor(xi.rest[-S.ab]))

 K.rest = max(xi.rest)

 a.sn = b.sn = array(0,dim=c(2,K.rest+2,J))
 a.sy = b.sy = array(0,dim=c(2,K.rest+2,J))
 if(Pois) a.lpf = b.lpf = array(0,dim=c(2,K.rest+2,J))

for(j in 1:J){
 if(model[j]=='B'){
  if(direct[j]){ ### directed
    if(K.rest > 0){
      for( l in 1:K.rest ){
       a.sn[,l,j] = b.sn[,l,j] = sum(xi.rest==l)
        a.sy[1,l,j] = sum(Y[a,xi.rest==l,j]) # xi_a,l (row)
        b.sy[1,l,j] = sum(Y[b,xi.rest==l,j])
        a.sy[2,l,j] = sum(Y[xi.rest==l,a,j]) # l,xi_a (col)
        b.sy[2,l,j] = sum(Y[xi.rest==l,b,j])
      }
    }
    a.sn[,K.rest+1,j] = 0 #K.rest+1 is the "a" group
    a.sy[,K.rest+1,j] = 0
    b.sn[,K.rest+1,j] = 1
    b.sy[1,K.rest+1,j] = Y[b,a,j]
```

```
    b.sy[2,K.rest+1,j] = Y[a,b,j]


    a.sn[,K.rest+2,j] = 1 #K.rest+2 is the "b" group
    a.sy[1,K.rest+2,j] = Y[a,b,j]
    a.sy[2,K.rest+2,j] = Y[b,a,j]
    b.sn[,K.rest+2,j] = 0
    b.sy[,K.rest+2,j] = 0


  }else{   ### undirected
    if(K.rest > 0){
      for( l in 1:K.rest ){
       a.sn[1,l,j] = b.sn[1,l,j] = sum(xi.rest==l)
        a.sy[1,l,j] = sum(Y[xi.rest==l,a,j]) + sum(Y[a,xi.rest==l,j])
        b.sy[1,l,j] = sum(Y[xi.rest==l,b,j]) + sum(Y[b,xi.rest==l,j])
      }
    }
    a.sn[1,K.rest+1,j] = 0 #K.rest + 1 is the "a" group
    a.sy[1,K.rest+1,j] = 0
    b.sn[1,K.rest+1,j] = 1
    b.sy[1,K.rest+1,j] = Y[a,b,j] + Y[b,a,j]


    a.sn[1,K.rest+2,j] = 1 #K.rest + 2 is the "b" group
    a.sy[1,K.rest+2,j] = Y[a,b,j] + Y[b,a,j]
    b.sn[1,K.rest+2,j] = 0
    b.sy[1,K.rest+2,j] = 0
  }
 }else if(model[j]=='P'){
  if(direct[j]){ ### directed
    if(K.rest > 0){
```

```
  for( l in 1:K.rest ){
   a.sn[,l,j] = b.sn[,l,j] = sum(xi.rest==l)
    a.sy[1,l,j] = sum(Y[a,xi.rest==l,j]) # xi_a,l (row)
    a.lpf[1,l,j] = sum(lfactorial(Y[a,xi.rest==l,j]))
    b.sy[1,l,j] = sum(Y[b,xi.rest==l,j])
    b.lpf[1,l,j] = sum(lfactorial(Y[b,xi.rest==l,j]))
    a.sy[2,l,j] = sum(Y[xi.rest==l,a,j]) # l,xi_a (col)
    a.lpf[2,l,j] = sum(lfactorial(Y[xi.rest==l,a,j]))
    b.sy[2,l,j] = sum(Y[xi.rest==l,b,j])
    b.lpf[2,l,j] = sum(lfactorial(Y[xi.rest==l,b,j]))
  }
}
a.sn[,K.rest+1,j] = 0 #K.rest+1 is the "a" group
a.sy[,K.rest+1,j] = 0
a.lpf[,K.rest+1,j] = 0
b.sn[,K.rest+1,j] = 1
b.sy[1,K.rest+1,j] = Y[b,a,j]
b.sy[2,K.rest+1,j] = Y[a,b,j]
b.lpf[1,K.rest+1,j] = lfactorial(Y[b,a,j])
b.lpf[2,K.rest+1,j] = lfactorial(Y[a,b,j])

a.sn[,K.rest+2,j] = 1 #K.rest+2 is the "b" group
a.sy[1,K.rest+2,j] = Y[a,b,j]
a.sy[2,K.rest+2,j] = Y[b,a,j]
a.lpf[1,K.rest+2,j] = lfactorial(Y[a,b,j])
a.lpf[2,K.rest+2,j] = lfactorial(Y[b,a,j])
b.sn[,K.rest+2,j] = 0
b.sy[,K.rest+2,j] = 0
b.lpf[,K.rest+2,j] = 0
```

```
  }else{   ### undirected
    if(K.rest > 0){
      for( l in 1:K.rest ){
       a.sn[1,l,j] = b.sn[1,l,j] = sum(xi.rest==l)
        a.sy[1,l,j] = sum(Y[xi.rest==l,a,j]) + sum(Y[a,xi.rest==l,j])
        a.lpf[1,l,j] = sum(lfactorial(Y[xi.rest==l,a,j])) + sum(lfactorial(Y[a,xi.rest=
        b.sy[1,l,j] = sum(Y[xi.rest==l,b,j]) + sum(Y[b,xi.rest==l,j])
        b.lpf[1,l,j] = sum(lfactorial(Y[xi.rest==l,b,j])) + sum(lfactorial(Y[b,xi.rest=
      }
    }
    a.sn[1,K.rest+1,j] = 0 #K.rest + 1 is the "a" group
    a.sy[1,K.rest+1,j] = 0
    a.lpf[1,K.rest+1,j] = 0
    b.sn[1,K.rest+1,j] = 1
    b.sy[1,K.rest+1,j] = Y[a,b,j] + Y[b,a,j]
    b.lpf[1,K.rest+1,j] = lfactorial(Y[a,b,j]) + lfactorial(Y[b,a,j])

    a.sn[1,K.rest+2,j] = 1 #K.rest + 2 is the "b" group
    a.sy[1,K.rest+2,j] = Y[a,b,j] + Y[b,a,j]
    a.lpf[1,K.rest+2,j] = lfactorial(Y[a,b,j]) + lfactorial(Y[b,a,j])
    b.sn[1,K.rest+2,j] = 0
    b.sy[1,K.rest+2,j] = 0
    b.lpf[1,K.rest+2,j] = 0
  }
 } #end of Poisson

}
if(Pois) output=list(a.sn=a.sn, a.sy=a.sy, a.lpf=a.lpf, b.sn=b.sn, b.sy=b.sy, b.lpf=b.l
```

```
return(output)
}


############    suff.group   ##############################
#  Computes the sufficient statistics for one particular
# set of subjects, assuming they form a group.
#  In the output the columns are each group, the given group
# is the last column. The first row is k,l (row) subgroups
# and the second is l,k (col) subgroups.
#
# Input: S.k, Y, xi, direct, model
# Y array of matrices
# S.k = vector of indices
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: list(k.sn, k.sy, k.lpf ) // n and sum of y's for k
# // and log(prod y!) if Poisson//
#################################################################

suff.group <- function(S.k, Y, xi, direct, model){
  n <- dim(Y)[1]
  Pois = sum(model=='P')>0
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
```

```
  J <- dim(Y)[3]
 }
 m.k = length(S.k)
 xi.rest = xi
 xi.rest[S.k] = 0
 xi.rest[-S.k] = as.numeric(factor(xi.rest[-S.k]))

 K.rest = max(xi.rest)

 k.sn = array(0,dim=c(2,K.rest+1,J))
 k.sy = array(0,dim=c(2,K.rest+1,J))
 if(Pois)   k.lpf = array(0,dim=c(2,K.rest+1,J))

for(j in 1:J){
 if(model[j]=='B'){
  if(direct[j]){ ### directed
    if(K.rest > 0){
      for( l in 1:K.rest ){
       k.sn[,l,j] = m.k*sum(xi.rest==l)
        k.sy[1,l,j] = sum(Y[S.k,xi.rest==l,j]) # xi_k,l (row)
        k.sy[2,l,j] = sum(Y[xi.rest==l,S.k,j]) # l,xi_k (col)
      }
    }
   #K.rest+1 is the "k" group
    k.sn[,K.rest+1,j] = m.k*(m.k-1)
    k.sy[,K.rest+1,j] = sum(Y[S.k,S.k,j])

  }else{   ### undirected
    if(K.rest > 0){
```

```
    for( l in 1:K.rest ){
     k.sn[1,l,j] = m.k*sum(xi.rest==l)
      k.sy[1,l,j] = sum(Y[xi.rest==l,S.k,j]) + sum(Y[S.k,xi.rest==l,j])
    }
  }
  k.sn[1,K.rest+1,j] = m.k*(m.k-1)/2 #K.rest+1 is the "k" group
  k.sy[1,K.rest+1,j] = sum(Y[S.k,S.k,j])
 }
}else if(model[j]=='P'){
 if(direct[j]){ ### directed
  if(K.rest > 0){
    for( l in 1:K.rest ){
     k.sn[,l,j] = m.k*sum(xi.rest==l)
      k.sy[1,l,j] = sum(Y[S.k,xi.rest==l,j]) # xi_k,l (row)
      k.lpf[1,l,j] = sum(lfactorial(Y[S.k,xi.rest==l,j]))
      k.sy[2,l,j] = sum(Y[xi.rest==l,S.k,j]) # l,xi_k (col)
      k.lpf[2,l,j] = sum(lfactorial(Y[xi.rest==l,S.k,j]))
    }
  }
  #K.rest+1 is the "k" group
  k.sn[,K.rest+1,j] = m.k*(m.k-1)
  k.sy[,K.rest+1,j] = sum(Y[S.k,S.k,j])
  k.lpf[,K.rest+1,j] = sum(lfactorial(Y[S.k,S.k,j]))

 }else{   ### undirected
  if(K.rest > 0){
    for( l in 1:K.rest ){
     k.sn[1,l,j] = m.k*sum(xi.rest==l)
      k.sy[1,l,j] = sum(Y[xi.rest==l,S.k,j]) + sum(Y[S.k,xi.rest==l,j])
```

```
        k.lpf[1,l,j] = sum(lfactorial(Y[xi.rest==l,S.k,j])) + sum(lfactorial(Y[S.k,xi.r
      }
    }
    k.sn[1,K.rest+1,j] = m.k*(m.k-1)/2 #K.rest+1 is the "k" group
    k.sy[1,K.rest+1,j] = sum(Y[S.k,S.k,j])
    k.lpf[1,K.rest+1,j] = sum(lfactorial(Y[S.k,S.k,j]))
  }
 } #end of Poisson
}
if(Pois) output=list(k.sn=k.sn, k.sy=k.sy, k.lpf=k.lpf) else output=list(k.sn=k.sn, k.s

return(output)
}

#################################################################
#################################################################
#
# File: funcModel.R
#
# Functions to compute marginal density and sample theta for

# Model
#  Binomial Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Bernoulli(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Beta(hyperD$.a.bin,hyperD$b.bin)   l=k
#    Beta(hyperOD$.a.bin,hyperOD$b.bin) l \neq k
#  Poisson Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Poisson(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Gamma(hyperD$a.pois,hyperD$b.pois)   l=k
```

```
#    Gamma(hyperOD$a.pois,hyperOD$b.pois) l \neq k

############    dmarg.Y.B  ########################
# density function of the marginal of y vector
# assuming all yi's are "BINARY" outcomes
# and come from the same subgroup
#
# Input: n, sum.y, hyper, log = FALSE
# hyper = list(a.bin=1/2, b.bin=1/2, ...)
# log = TRUE the output is the log(density)
# Output: density or Pr(Y)
################################################

dmarg.Y.B <- function(n, sum.y, hyper, log = FALSE){

hyp = hyper
if(n<sum.y) {print(n); print(sum.y)}

log.pr.y = lbeta(hyp$a.bin + sum.y, hyp$b.bin + n - sum.y) - lbeta(hyp$a.bin,hyp$b.bin)

density = if(log) log.pr.y else exp(log.pr.y)

return(density)
}

############    dmarg.Y.P  ########################
# density function of the marginal of y vector
# assuming all yi's are "POISSON" outcomes
# and come from the same subgroup
```

```
#
# Input: n, sum.y, log.prod.y, hyper, log = FALSE
# hyper = list(a.pois=1/2, b.pois=1/2, ...)
# log = TRUE the output is the log(density)
# Output: density or Pr(Y)
################################################

dmarg.Y.P <- function(n, sum.y, log.prod.y, hyper, log = FALSE){

hyp = hyper

log.pr.y = hyp$a.pois*log(hyp$b.pois) + lgamma(sum.y + hyp$a.pois) - (lgamma(hyp$a.pois)

density = if(log) log.pr.y else exp(log.pr.y)

return(density)
}

############    sample.Theta    ########################
# Sampling from p(Theta | xi^(i), Y)
#
# Input: Y, zeta, XI, hyperD, hyperOD, direct, model
# hyper = list(a,b, eta, ...)
# direct = c(F,..) F symmetric, T o.w.
# model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: matrix Theta
################################################
sample.Theta <- function(Y, zeta, XI, hyperD, hyperOD, direct, model){
```

```
 n <- dim(Y)[1]
 if(length(dim(Y))<3){
   J <- 1
   Y <- array(Y,dim=c(n,n,J))
   XI <- matrix(XI,1,n)
 }else{
   J <- dim(Y)[3]
 }
 max.K <- max(XI)
 R <- max(zeta)

  ## sampling theta_{lk}
 Theta <- array(0,dim=c(max.K,max.K,J))

for(j in 1:J){
 xi <- XI[zeta[j],]
 K <- max(xi)
   ## the sufficient statistics
 U <- suff.xi(Y[,,j],xi,direct[j],model[j])

 if(model[j]=="B"){
   if(direct[j]){
     for(l in 1:K){
      for(k in l:K){
        if(l==k) hyp=hyperD else hyp=hyperOD
        Theta[l,k,j] = rbeta(1,hyp$a.bin[j] + U$U.sy[l,k,], hyp$b.bin[j] + U$U.sn[l,k,]
        Theta[k,l,j] = rbeta(1,hyp$a.bin[j] + U$U.sy[k,l,], hyp$b.bin[j] + U$U.sn[k,l,]
      }
```

```
    }
   }else{
    for(l in 1:K){
     for(k in l:K){
       if(l==k) hyp=hyperD else hyp=hyperOD
       Theta[l,k,j] = rbeta(1,hyp$a.bin[j] + U$U.sy[l,k,], hyp$b.bin[j] + U$U.sn[l,k,]
     }
    }
   }
 }else if(model[j]=="P"){
  if(direct[j]){
    for(l in 1:K){
     for(k in l:K){
       if(l==k){ hyp=hyperD }else{ hyp=hyperOD }
       Theta[l,k,j] = rgamma(1,shape=(hyp$a.pois + U$U.sy[l,k,]), rate=(hyp$b.pois + U
       Theta[k,l,j] = rgamma(1,shape=(hyp$a.pois + U$U.sy[k,l,]), rate=(hyp$b.pois + U
     }
    }
   }else{
    for(l in 1:K){
     for(k in l:K){
       if(l==k){ hyp=hyperD }else{ hyp=hyperOD }
       Theta[l,k,j] = rgamma(1,shape=(hyp$a.pois + U$U.sy[l,k,]), rate=(hyp$b.pois + U
     }
    }
   }
 }  # end of model
}  # end of for(j in 1:J)
```

```
return(Theta)
}

############    sample.lambda    ########################
# Sampling from p(lambda | Y)
#
# Input: Y, zeta, XI, hyperD, hyperOD, direct, model
# hyper = list(a,b, eta, ...)
# direct = c(F,..) F symmetric, T o.w.
# model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: hyperD, hyperOD
#################################################
sample.lambda <- function(Y, zeta, XI, hyperD, hyperOD, direct, model){

  n <- dim(Y)[1]
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
    XI <- matrix(XI,1,n)
  }else{
    J <- dim(Y)[3]
  }
  max.K <- max(XI)
  R <- max(zeta)

  a.D = hyperD$a.bin
  b.D = hyperD$b.bin
  a.OD = hyperOD$a.bin
```

```
  b.OD = hyperOD$b.bin
  alpha.a = hyperD$alpha.a
  beta.a = hyperD$beta.a
  alpha.b = hyperD$alpha.b
  beta.b = hyperD$beta.b
  kappa = hyperD$kappa

 for(j in 1:J){
  #print(j)
  xi <- XI[zeta[j],]
  K <- max(xi)
    ## the sufficient statistics
  U <- suff.xi(Y[,,j],xi,direct[j],model[j])
  p.aD = exp(rnorm(1,log(a.D[j]),kappa))
  p.bD = exp(rnorm(1,log(b.D[j]),kappa))
  p.aOD = exp(rnorm(1,log(a.OD[j]),kappa))
  p.bOD = exp(rnorm(1,log(b.OD[j]),kappa))

  ratio.D = dgamma(p.aD, shape=alpha.a, rate=beta.a, log=T) + dgamma(p.bD, shape=alpha.b
  ratio.OD = dgamma(p.aOD, shape=alpha.a, rate=beta.a, log=T) + dgamma(p.bOD, shape=alph

  if(model[j]=="B"){
    if(direct[j]){
      for(l in 1:K){
       for(k in l:K){
         if(l==k){
ratio.D = ratio.D + dmarg.Y.B(U$U.sn[l,k,], U$U.sy[l,k,], hyper=list(a.bin=p.aD, b.bin=p
         }else{
ratio.OD = ratio.OD + dmarg.Y.B(U$U.sn[l,k,], U$U.sy[l,k,], hyper=list(a.bin=p.aOD, b.bi
```

```
        }
       }
      }

    }else{
      for(l in 1:K){
       for(k in l:K){
         if(l==k){
ratio.D = ratio.D + dmarg.Y.B(U$U.sn[l,k,], U$U.sy[l,k,], hyper=list(a.bin=p.aD, b.bin=p
         }else{
ratio.OD = ratio.OD + dmarg.Y.B(U$U.sn[l,k,], U$U.sy[l,k,], hyper=list(a.bin=p.aOD, b.bi
         }
       }
      }
    }
  }else if(model[j]=="P"){
### DOESN'T WORK YET
  }  # end of model

  ## updating
  if(rbinom(1,1,p=min(exp(ratio.D),1))){
   a.D[j] = p.aD
   b.D[j] = p.bD
  }
  if(rbinom(1,1,p=min(exp(ratio.OD),1))){
   a.OD[j] = p.aOD
   b.OD[j] = p.bOD
  }
 }  # end of for(j in 1:J)
```

```
  hyperD$a.bin = a.D
  hyperD$b.bin = b.D
  hyperOD$a.bin = a.OD
  hyperOD$b.bin = b.OD

return(list(hyperD=hyperD, hyperOD=hyperOD))
}


############    meanPost.Theta    ########################
# Computes the mean from p(Theta | xi^(i), Y)
#
# Input: Y, zeta, XI, hyperD, hyperOD, direct, model
# hyper = list(a,b, eta, ...)
# direct = c(F,..) F symmetric, T o.w.
# model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: matrix Theta
#################################################
meanPost.Theta <- function(Y, zeta, XI, hyperD, hyperOD, direct, model){

  n <- dim(Y)[1]
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
    XI <- matrix(XI,1,n)
  }else{
    J <- dim(Y)[3]
```

```
 }
 max.K <- max(XI)
 R <- max(zeta)

  ## sampling theta_{lk}
 Theta <- array(0,dim=c(max.K,max.K,J))

for(j in 1:J){
 xi <- XI[zeta[j],]
 K <- max(xi)
   ## the sufficient statistics
 U <- suff.xi(Y[,,j],xi,direct[j],model[j])

 if(model[j]=="B"){
   if(direct[j]){
     for(l in 1:K){
      for(k in l:K){
        if(l==k) hyp=hyperD else hyp=hyperOD
        Theta[l,k,j] = (hyp$a.bin[j] + U$U.sy[l,k,])/(hyp$a.bin[j] + hyp$b.bin[j] + U$U
        Theta[k,l,j] = (hyp$a.bin[j] + U$U.sy[k,l,])/(hyp$a.bin[j] + hyp$b.bin[j] + U$U
      }
     }
   }else{
     for(l in 1:K){
      for(k in l:K){
        if(l==k) hyp=hyperD else hyp=hyperOD
        Theta[l,k,j] = (hyp$a.bin[j] + U$U.sy[l,k,])/(hyp$a.bin[j] + hyp$b.bin[j] + U$U
      }
     }
```

```
    }
  }else if(model[j]=="P"){
   if(direct[j]){
      for(l in 1:K){
       for(k in l:K){
         if(l==k){ hyp=hyperD }else{ hyp=hyperOD }
         Theta[l,k,j] = (hyp$a.pois + U$U.sy[l,k,])/(hyp$b.pois + U$U.sn[l,k,])
         Theta[k,l,j] = (hyp$a.pois + U$U.sy[k,l,])/(hyp$b.pois + U$U.sn[k,l,])
       }
      }
    }else{
      for(l in 1:K){
       for(k in l:K){
         if(l==k){ hyp=hyperD }else{ hyp=hyperOD }
         Theta[l,k,j] = (hyp$a.pois + U$U.sy[l,k,])/(hyp$b.pois + U$U.sn[l,k,])
       }
      }
    }
  }  # end of model
 }  # end of for(j in 1:J)

return(Theta)
}

############    sample.alpha    ########################
# Sampling from p(alpha | L, theta, Y)
#   where alpha is the mixing paramenter of the DP
#
# Input: alpha, n, L, hyper
```

```
# alpha = actual value
# L = number of groups on the actual theta (xi)
# hyper = list(..., a.alpha, b.alpha)
# a.alpha, b.alpha = param of prior(alpha)
#
# Output: new.alpha
################################################

sample.alpha <- function(alpha,n,L,hyper){
  a <- hyper$a.alpha
  b <- hyper$b.alpha
  eta <- rbeta(1,alpha+1,n)
  odds <- (a+L-1)/(n*(b-log(eta)))
  w <- odds/(1+odds)
  if( rbinom(1,size=1,prob=w) ){
   new.alpha <- rgamma(1,shape=(a+L),rate=(b-log(eta)))
  }else{
   new.alpha <- rgamma(1,shape=(a+L-1),rate=(b-log(eta)))
  }

return(new.alpha)
}

############    gen.alpha    ########################
# Sampling from prior p(alpha)
#   where alpha is the mixing paramenter of the DP
# and assuming prior gamma(shape=a.alpha,rate=b.alpha)
#
# Input: hyper = list(..., a.alpha, b.alpha)
```

```
# a.alpha, b.alpha = param of prior(alpha)
#
# Output: new.alpha
################################################

gen.alpha <- function(hyper){
  new.alpha <- rgamma(1,shape=hyper$a.alpha,rate=hyper$b.alpha)

return(new.alpha)
}

############    d.alpha    ########################
# density prior p(alpha)
#   where alpha is the mixing paramenter of the DP
# and assuming prior gamma(shape=a.alpha,rate=b.alpha)
#
# Input: alpha, hyper = list(..., a.alpha, b.alpha), log = FALSE
# a.alpha, b.alpha = param of prior(alpha)
#
# Output: d.alpha
################################################

d.alpha <- function(alpha,hyper, log=FALSE){
  density <- dgamma(alpha,shape=hyper$a.alpha,rate=hyper$b.alpha, log=log)

return(density)
}

#################################################################
```

```
#################################################################
#
# File: funcGibbsUpdate.R
#
# Functions to run a Gibbs Sampler for

# Model
#  Binomial Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Bernoulli(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Beta(hyperD$.a.bin,hyperD$b.bin)   l=k
#    Beta(hyperOD$.a.bin,hyperOD$b.bin) l \neq k


#  Poisson Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Poisson(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Gamma(hyperD$a.pois,hyperD$b.pois)   l=k
#    Gamma(hyperOD$a.pois,hyperOD$b.pois) l \neq k


# NOTE: For suff stats computations requires:
# source("funcSuffStats.R")
# source("funcModel.R")

############    sample.xi    ########################
# Sampling from p(xi_i | xi^(i), Y)
#
# Input: Y, xi, U, hyperD, hyperOD, direct, model,
# U=list(U.sn, U.sy, U.lpf)
# hyper = list(a,b, eta)
```

```
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: updated xi, U and Q.xi (prob of updated xi)
################################################

sample.xi <- function(Y,xi,U,hyperD,hyperOD,direct,model){
n <- dim(Y)[1]
Pois = sum(model=='P')>0
if(length(dim(Y))<3){
  J <- 1
  Y <- array(Y,dim=c(n,n,J))
}else{
  J <- dim(Y)[3]
}

log.Q = 0

  i = 1
  for( i in 1:n){
   #print(c("i",i))
   #print(1)
   xi.n = xi
   xi.n[i] = 0
   xi.n[-i] = as.numeric(factor(xi.n[-i]))
   K = max(xi)
   K.n = max(xi.n)

  ## the sufficient statistics
```

```
 #print(2)
 Zz <- suff.xi_i(Y, xi, U, i, direct, model)
 Z.sn <- Zz$Z.sn
 Z.sy <- Zz$Z.sy
 z.sn <- Zz$z.sn
 z.sy <- Zz$z.sy
 if(Pois){
   Z.lpf <- Zz$Z.lpf
   z.lpf <- Zz$z.lpf
 }

## the group probabilities
 q <- log(table(xi.n[xi.n!=0]))
 q[K.n+1] <- log(hyperD$eta)

 for(j in 1:J){
  if(model[j]=="B"){ ### Binary
   hypD = list(a.bin=hyperD$a.bin[j], b.bin=hyperD$b.bin[j])
   hypOD = list(a.bin=hyperOD$a.bin[j], b.bin=hyperOD$b.bin[j])
   if(direct[j]){ ####directed
     #k<-1
     for( k in 1:K.n){
       q[k] = q[k] + dmarg.Y.B(Z.sn[k,k,j]+z.sn[1,k,j]+z.sn[2,k,j], Z.sy[k,k,j]+z.sy[1
       #l<-2
       for(l in (1:K.n)[-k]){
         #print(3)
        #for the k,l group (row) and then the l,k (col)
         q[k] = q[k] + dmarg.Y.B(Z.sn[k,l,j]+z.sn[1,l,j], Z.sy[k,l,j]+z.sy[1,l,j], hyp
       }
```

```
    }
    for(l in 1:K.n){
   #for the k,l group (row) and then the l,k (col)
      q[K.n+1] = q[K.n+1] + dmarg.Y.B(z.sn[1,l,j], z.sy[1,l,j], hyper=hypOD, log=T) +
    }
  }else{ #####undirected
    #k<-1
    for( k in 1:K.n){
      #l<-1
      for(l in 1:K.n){
        r=min(k,l)
        c=max(k,l)
        #print(3)
        if(r==c) hyp = hypD else hyp = hypOD;
        q[k] = q[k] + dmarg.Y.B(Z.sn[r,c,j]+z.sn[1,l,j], Z.sy[r,c,j]+z.sy[1,l,j], hyp
      }
    }
    for(l in 1:K.n){
      q[K.n+1] = q[K.n+1] + dmarg.Y.B(z.sn[1,l,j], z.sy[1,l,j], hyper=hypOD, log=T)
    }
  } # end of if(direct[j])
 }else if(model[j]=="P"){ ### Poisson
  if(direct[j]){ ####directed
    #k<-1
    for( k in 1:K.n){
      q[k] = q[k] + dmarg.Y.P(Z.sn[k,k,j]+z.sn[1,k,j]+z.sn[2,k,j], Z.sy[k,k,j]+z.sy[1
      #l<-2
      for(l in (1:K.n)[-k]){
        #print(3)
```

```
       #for the k,l group (row) and then the l,k (col)
        q[k] = q[k] + dmarg.Y.P(Z.sn[k,l,j]+z.sn[1,l,j], Z.sy[k,l,j]+z.sy[1,l,j], Z.l
      }
    }
    for(l in 1:K.n){
   #for the k,l group (row) and then the l,k (col)
      q[K.n+1] = q[K.n+1] + dmarg.Y.P(z.sn[1,l,j], z.sy[1,l,j], z.lpf[1,l,j], hyper=h
    }
  }else{ #####undirected
    #k<-1
    for( k in 1:K.n){
      #l<-1
      for(l in 1:K.n){
        r=min(k,l)
        c=max(k,l)
        #print(3)
        if(r==c) hyp = hyperD else hyp = hyperOD;
        q[k] = q[k] + dmarg.Y.P(Z.sn[r,c,j]+z.sn[1,l,j], Z.sy[r,c,j]+z.sy[1,l,j], Z.l
      }
    }
    for(l in 1:K.n){
      q[K.n+1] = q[K.n+1] + dmarg.Y.P(z.sn[1,l,j], z.sy[1,l,j], z.lpf[1,l,j], hyper=h
    }
  } # end of if(direct[j])
 }
} # end of for(j 1:J)

# sampling the new value of xi.[i]
#print(q)
```

```
   #print(xi)
   #print(z.sy)
   q <- exp(q - max(q))
   ind <- sample(1:(K.n+1),1, prob=q)
   xi.n[i] <- ind
   log.Q = log.Q + log(q[ind])

   #update sufficient stats if at least one change
   if(sum(xi.n!=xi) > 0){
    #print(4)
    U <- update.suff.xi_i(Zz, ind, direct, model)
   }
   #print(xi.n)
   xi <- xi.n
  }

return(list(xi=xi, U=U, Q.xi=exp(log.Q)))
}

############    sample.zeta    ########################
# Sampling from p(zeta_r | .... )
#
# Input: Y, zeta, XI, hyperD, hyperOD, direct, model
# hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: updated zeta, XI, ETA.XI
################################################
```

```
sample.zeta <- function(Y, zeta, XI, hyperD, hyperOD, direct, model){

  n <- dim(Y)[1]
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
  ETA.XI <- hyperD$ETA.XI

    j = 1
  for( j in 1:J){
   zeta.n = zeta
   zeta.n[j] = 0
   zeta.n[-j] = as.numeric(factor(zeta.n[-j]))
   R = max(zeta)
   R.n = max(zeta.n)
   #ETA.XI.n <- ETA.XI

   if(R > R.n){
    XI.n <- XI[-zeta[j],]
   }else{
    XI.n <- XI
   }
    #we need a new one for R.n+1
   #XI.n <- rbind(XI.n,func.CRP(n,mean(ETA.XI[-j])))
   hypD <- hyperD
```

```
hypOD <- hyperOD
hypD$eta <- hypOD$eta <- mean(hyperD$ETA.XI)
Assig <- Initial.xi(Y[,,j], hypD, hypOD, direct[j], model[j])
XI.n <- rbind(XI.n,Assig$xi)

q <- log(table(zeta.n[zeta.n!=0]))
q[R.n+1] <- log(hyperD$eta.zeta)
#r=1
for(r in 1:(R.n+1)){
 xi <- XI.n[r,]
 K <- max(xi)
 U <- suff.xi(Y[,,j],xi,direct[j],model[j])

 if(model[j]=="B"){ ### Binary
   hD = list(a.bin=hyperD$a.bin[j], b.bin=hyperD$b.bin[j])
   hOD = list(a.bin=hyperOD$a.bin[j], b.bin=hyperOD$b.bin[j])
   if(direct[j]){ ####directed
    for( k in 1:K){
      q[r] = q[r] + dmarg.Y.B(U$U.sn[k,k,1], U$U.sy[k,k,1], hyper=hD, log=T) #for the
      for(l in (1:K)[-k]){
       #for the k,l group (row) and then the l,k (col)
        q[r] = q[r] + dmarg.Y.B(U$U.sn[k,l,1], U$U.sy[k,l,1], hyper=hOD, log=T) + dma
      }
    }
   }else{ #####undirected
    for( k in 1:K ){
      for(l in 1:K){
        ridx=min(k,l)
        cidx=max(k,l)
```

```
        if(ridx==cidx) hyp = hD else hyp = hOD;
        q[r] = q[r] + dmarg.Y.B(U$U.sn[ridx,cidx,1], U$U.sy[ridx,cidx,1], hyper=hyp,
      }
    }
  } # end of if(direct[j])
 }else if(model[j]=="P"){ ### Poisson
  if(direct[j]){ ####directed
    for( k in 1:K ){
      q[r] = q[r] + dmarg.Y.P(U$U.sn[k,k,1], U$U.sy[k,k,1], U$U.lpf[k,k,1], hyper=hyp
      for(l in (1:K)[-k]){
       #for the k,l group (row) and then the l,k (col)
        q[r] = q[r] + dmarg.Y.P(U$U.sn[k,l,1], U$U.sy[k,l,1], U$U.lpf[k,l,1], hyper=h
      }
    }
  }else{ #####undirected
    for( k in 1:K){
      for(l in 1:K){
        ridx=min(k,l)
        cidx=max(k,l)
        if(ridx==cidx) hyp = hyperD else hyp = hyperOD;
        q[r] = q[r] + dmarg.Y.P(U$U.sn[ridx,cidx,1], U$U.sy[ridx,cidx,1], U$U.lpf[rid
      }
    }
  } # end of if(direct[j])
 } #end of if(model[j])

}
q <- exp(q - max(q))
index <- sample(1:(R.n+1),1, prob=q)
```

```
    zeta.n[j] <- index
    XI <- XI.n[1:max(zeta.n),]
    if(index==(R.n+1)){
      ETA.XI[j] <- mean(ETA.XI[-j])
    }else{
      temp <- ETA.XI[-j]
      ETA.XI[j] <- temp[(zeta.n==index)[-j]][1]
    }
    zeta <- zeta.n

  }
  return(list(zeta=zeta, XI=XI, ETA.XI=ETA.XI))
}

#####################################################################
#####################################################################

#
# File: Collections.R
#
# Date: 10/15/2010
# Functions to compute and update sufficient stats for a collection of matrices

# Model Y_{i,i',j} | zeta, XI, Theta ~ Bernoulli(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Beta(a,b)

# NOTE: For computations requires:
# source("funcSuffStats.R")
# source("funcModel.R")
```

```
# source("funcGibbsUpdate.R")


############    func.CRP    ##############################
# Assigns entities to groups following a simple Poyla Urn
#
# Input: size, alpha
# alpha = DP parameter
# Output: xi (vector of group allocations)
############################################################

func.CRP <- function(size, alpha){
  xi <- rep(0,size)
  xi[1] <- 1
  for(i in 2:size){
   K <- max(xi)
   q <- table(xi[xi!=0])
   q[K+1] <- alpha
   xi[i] <- sample((K+1),1,prob=q)
  }
return(xi)
}

############   Initial.xi   ##############################
# Assigns actors to groups using posterior prob
#
# Input: Y, hyperD, hyperOD, direct, model, pi
# Y array of matrices
#     hyper = list(a,b, eta)
```

```
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# pi = a permutation of 1:n to use if we want to fix it
# Output: list(xi, Q.xi) (allocations and prob to get them)
############################################################

Initial.xi <- function(Y, hyperD, hyperOD, direct, model, pi=NULL){
  n <- dim(Y)[1]
  Pois = sum(model=='P')>0
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
  if(is.null(pi)){
    pi <- sample(n)
  }
  xi <- rep(0,n)
  xi[pi[1]] <- 1
  q <- c(1,hyperD$eta)/(1+hyperD$eta)
  xi[pi[2]] <- sample(2,1,prob=q)
  U <- suff.xi(Y,xi,direct,model)
  log.Q = log(q[xi[pi[2]]])

  #i=6
  for(i in pi[3:n]){
   K <- max(xi)
```

```
## suff statistics
    z.sn <- array(0,dim=c(2,K,J))
    z.sy <- array(0,dim=c(2,K,J))
    if(Pois) z.lpf <- array(0,dim=c(2,K,J))

    for(j in 1:J){
     if(model[j]=='B'){ ## Binary
      if(direct[j]){
        for(l in 1:K){
    z.sn[,l,j] <- sum(xi==l) #same n for xi_i,l and l,xi_i blocks
          z.sy[1,l,j] = sum(Y[i,xi==l,j])  # xi_i,l block (row);
          z.sy[2,l,j] = sum(Y[xi==l,i,j])  # l,xi_i block (col)
        }
      }else{ ## undirected
        for(l in 1:K){
  #affecting the l,xi_i block
          z.sn[1,l,j] = sum(xi==l)
          z.sy[1,l,j] = sum(Y[i,xi==l,j]) + sum(Y[xi==l,i,j])
        }
     }
    }else if(model[j]=='P'){ ## Poisson
      if(direct[j]){
        for(l in 1:K){
    z.sn[,l,j] <- sum(xi==l) #same n for xi_i,l and l,xi_i blocks
          z.sy[1,l,j] = sum(Y[i,xi==l,j])  # xi_i,l block (row);
          z.lpf[1,l,j] = sum(lfactorial(Y[i,xi==l,j]))
          z.sy[2,l,j] = sum(Y[xi==l,i,j])  # l,xi_i block (col)
      z.lpf[2,l,j] = sum(lfactorial(Y[xi==l,i,j]))
```

```
      }
    }else{ ## undirected
      for(l in 1:K){
#affecting the l,xi_i block
        z.sn[1,l,j] = sum(xi==l)
        z.sy[1,l,j] = sum(Y[i,xi==l,j]) + sum(Y[xi==l,i,j])
        z.lpf[1,l,j] = sum(lfactorial(Y[i,xi==l,j])) + sum(lfactorial(Y[xi==l,i,j]))
      }
    }


   }
  }
 q <- log(table(xi[xi!=0]))
 q[K+1] <- log(hyperD$eta)

  for(j in 1:J){
   if(model[j]=='B'){  ## Binary
    hD = list(a.bin=hyperD$a.bin[j], b.bin=hyperD$b.bin[j])
    hOD = list(a.bin=hyperOD$a.bin[j], b.bin=hyperOD$b.bin[j])
    if(direct[j]){ ####directed
     #k<-1
     for( k in 1:K){
      q[k] = q[k] + dmarg.Y.B(U$U.sn[k,k,j]+2*z.sn[1,k,j], U$U.sy[k,k,j]+z.sy[1,k,j]+z
      #l<-2
      for(l in (1:K)[-k]){
       #print(3)
      #for the k,l group (row) and then the l,k (col)
       q[k] = q[k] + dmarg.Y.B(U$U.sn[k,l,j]+z.sn[1,l,j], U$U.sy[k,l,j]+z.sy[1,l,j], h
      }
```

```
 }
 for(l in 1:K){
#for the k,l group (row) and then the l,k (col)
   q[K+1] = q[K+1] + dmarg.Y.B(z.sn[1,l,j], z.sy[1,l,j], hyper=hOD, log=T) + dmarg
 }
}else{ #####undirected
  #k<-1
 for(k in 1:K){
  #l<-1
  for(l in 1:K){
   r=min(k,l)
   c=max(k,l)
   if(r==c) hyp = hD else hyp = hOD;
    #print(3)
   q[k] = q[k] + dmarg.Y.B(U$U.sn[r,c,j]+z.sn[1,l,j], U$U.sy[r,c,j]+z.sy[1,l,j], h
  }
 }
 for(l in 1:K){
  q[K+1] = q[K+1] + dmarg.Y.B(z.sn[1,l,j], z.sy[1,l,j], hyper=hOD, log=T)
 }
}
}else if(model[j]=='P'){ ## Poisson
 if(direct[j]){ ####directed
  #k<-1
  for( k in 1:K){
   q[k] = q[k] + dmarg.Y.P(U$U.sn[k,k,j]+2*z.sn[1,k,j], U$U.sy[k,k,j]+z.sy[1,k,j]+z
   #l<-2
   for(l in (1:K)[-k]){
    #print(3)
```

```
        #for the k,l group (row) and then the l,k (col)
         q[k] = q[k] + dmarg.Y.P(U$U.sn[k,l,j]+z.sn[1,l,j], U$U.sy[k,l,j]+z.sy[1,l,j], U
        }
       }
       for(l in 1:K){
       #for the k,l group (row) and then the l,k (col)
        q[K+1] = q[K+1] + dmarg.Y.P(z.sn[1,l,j], z.sy[1,l,j], z.lpf[1,l,j], hyper=hyperOD
       }
      }else{ #####undirected
        #k<-1
       for(k in 1:K){
        #l<-1
        for(l in 1:K){
          r=min(k,l)
          c=max(k,l)
          if(r==c) hyp = hyperD else hyp = hyperOD;
#print(3)
          q[k] = q[k] + dmarg.Y.P(U$U.sn[r,c,j]+z.sn[1,l,j], U$U.sy[r,c,j]+z.sy[1,l,j], U
        }
       }
       for(l in 1:K){
        q[K+1] = q[K+1] + dmarg.Y.P(z.sn[1,l,j], z.sy[1,l,j], z.lpf[1,l,j], hyper=hyperOD
       }
      }


      }


   } ## end of for(j in 1:J)
```

```
    q <- exp(q - max(q))
    q <- q/sum(q)
    xi[i] <- sample((K+1),1,prob=q)
    if(Pois) Zz=list(Z.sn=U$U.sn, Z.sy=U$U.sy, Z.lpf=U$U.lpf, z.sn=z.sn, z.sy=z.sy, z.lp
   U <- update.suff.xi_i(Zz,xi[i],direct,model)
   log.Q = log.Q + log(q[xi[i]])

  } ## end of for(i in pi[3:n])
  Q.xi=exp(log.Q)

return(list(xi=xi, Q.xi=Q.xi))
}



############   Prob.Initial.xi   ##############################
#    Computes the probability of a given assignment of actors
# to groups, assuming the posterior prob was used for such
# assignment
#
# Input: Y, xi, hyperD, hyperOD, direct, model, log=F, pi = NULL
# Y array of matrices
#     hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# log = TRUE the output is the log(density)
# pi = a permutation of 1:n to use if we want to fix it
# Output: density or Pr(the given allocation)
############################################################
```

```
Prob.Initial.xi <- function(Y, xi, hyperD, hyperOD, direct, model, log=FALSE, pi=NULL){
  n <- dim(Y)[1]
  Pois = sum(model=='P')>0
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
  if(is.null(pi)){
    pi <- sample(n)
  }
  Hyp.xi <- rep(0,n)
  New.xi <- rep(0,n)
  Hyp.xi[pi[1]] <- 1
  New.xi[xi==xi[pi[1]]] <- 1

  q <- c(1,hyperD$eta)/(1+hyperD$eta)
  if(New.xi[pi[2]]==1){
    Hyp.xi[pi[2]] <- 1
    log.Q = log(q[1])
  }else{
    Hyp.xi[pi[2]] <- 2
    New.xi[xi==xi[pi[2]]] <- 2
    log.Q = log(q[2])
  }
  U <- suff.xi(Y,Hyp.xi,direct,model)
```

```
  #i=6
  for(i in pi[3:n]){
   K <- max(Hyp.xi)

## suff statistics
    z.sn <- array(0,dim=c(2,K,J))
    z.sy <- array(0,dim=c(2,K,J))
    if(Pois) z.lpf <- array(0,dim=c(2,K,J))

    for(j in 1:J){
     if(model[j]=='B'){ ## Binary
      if(direct[j]){
        for(l in 1:K){
    z.sn[,l,j] <- sum(Hyp.xi==l) #same n for xi_i,l and l,xi_i blocks
          z.sy[1,l,j] = sum(Y[i,Hyp.xi==l,j])  # xi_i,l block (row);
          z.sy[2,l,j] = sum(Y[Hyp.xi==l,i,j])  # l,xi_i block (col)
        }
      }else{ ## undirected
        for(l in 1:K){
  #affecting the l,xi_i block
          z.sn[1,l,j] = sum(Hyp.xi==l)
          z.sy[1,l,j] = sum(Y[i,Hyp.xi==l,j]) + sum(Y[Hyp.xi==l,i,j])
        }
     }
    }else if(model[j]=='P'){ ## Poisson
     if(direct[j]){
       for(l in 1:K){
    z.sn[,l,j] <- sum(Hyp.xi==l) #same n for xi_i,l and l,xi_i blocks
          z.sy[1,l,j] = sum(Y[i,Hyp.xi==l,j])  # xi_i,l block (row);
```

```
        z.lpf[1,l,j] = sum(lfactorial(Y[i,Hyp.xi==l,j]))
        z.sy[2,l,j] = sum(Y[Hyp.xi==l,i,j])  # l,xi_i block (col)
        z.lpf[2,l,j] = sum(lfactorial(Y[Hyp.xi==l,i,j]))
      }
    }else{ ## undirected
      for(l in 1:K){
#affecting the l,xi_i block
        z.sn[1,l,j] = sum(Hyp.xi==l)
        z.sy[1,l,j] = sum(Y[i,Hyp.xi==l,j]) + sum(Y[Hyp.xi==l,i,j])
        z.sy[1,l,j] = sum(lfactorial(Y[i,Hyp.xi==l,j])) + sum(lfactorial(Y[Hyp.xi==l,i
      }
    }
   } # end of Poisson
  }
 q <- log(table(Hyp.xi[Hyp.xi!=0]))
 q[K+1] <- log(hyperD$eta)

  for(j in 1:J){
   if(model[j]=='B'){  ## Binary
    hD = list(a.bin=hyperD$a.bin[j], b.bin=hyperD$b.bin[j])
    hOD = list(a.bin=hyperOD$a.bin[j], b.bin=hyperOD$b.bin[j])
    if(direct[j]){ ####directed
     #k<-1
     for( k in 1:K){
      q[k] = q[k] + dmarg.Y.B(U$U.sn[k,k,j]+2*z.sn[1,k,j], U$U.sy[k,k,j]+z.sy[1,k,j]+z
      #l<-2
      for(l in (1:K)[-k]){
       #print(3)
      #for the k,l group (row) and then the l,k (col)
```

```
    q[k] = q[k] + dmarg.Y.B(U$U.sn[k,l,j]+z.sn[1,l,j], U$U.sy[k,l,j]+z.sy[1,l,j], h
   }
  }
  for(l in 1:K){
 #for the k,l group (row) and then the l,k (col)
    q[K+1] = q[K+1] + dmarg.Y.B(z.sn[1,l,j], z.sy[1,l,j], hyper=hOD, log=T) + dmarg
  }
 }else{ #####undirected
   #k<-1
  for(k in 1:K){
   #l<-1
   for(l in 1:K){
    r=min(k,l)
    c=max(k,l)
    if(r==c) hyp = hD else hyp = hOD;
     #print(3)
    q[k] = q[k] + dmarg.Y.B(U$U.sn[r,c,j]+z.sn[1,l,j], U$U.sy[r,c,j]+z.sy[1,l,j], h
   }
  }
  for(l in 1:K){
   q[K+1] = q[K+1] + dmarg.Y.B(z.sn[1,l,j], z.sy[1,l,j], hyper=hOD, log=T)
  }
 }
}else if(model[j]=='P'){ ## Poisson
 if(direct[j]){ ####directed
  #k<-1
  for( k in 1:K){
   q[k] = q[k] + dmarg.Y.P(U$U.sn[k,k,j]+2*z.sn[1,k,j], U$U.sy[k,k,j]+z.sy[1,k,j]+z
   #l<-2
```

```
        for(l in (1:K)[-k]){
         #print(3)
        #for the k,l group (row) and then the l,k (col)
         q[k] = q[k] + dmarg.Y.P(U$U.sn[k,l,j]+z.sn[1,l,j], U$U.sy[k,l,j]+z.sy[1,l,j], U
        }
       }
       for(l in 1:K){
       #for the k,l group (row) and then the l,k (col)
        q[K+1] = q[K+1] + dmarg.Y.P(z.sn[1,l,j], z.sy[1,l,j], z.lpf[1,l,j], hyper=hyperOD
       }
      }else{ #####undirected
        #k<-1
       for(k in 1:K){
        #l<-1
        for(l in 1:K){
          r=min(k,l)
          c=max(k,l)
          if(r==c) hyp = hyperD else hyp = hyperOD;
#print(3)
          q[k] = q[k] + dmarg.Y.P(U$U.sn[r,c,j]+z.sn[1,l,j], U$U.sy[r,c,j]+z.sy[1,l,j], U
        }
       }
       for(l in 1:K){
        q[K+1] = q[K+1] + dmarg.Y.P(z.sn[1,l,j], z.sy[1,l,j], z.lpf[1,l,j], hyper=hyperOD
       }
      }


     }
```

```
   } ## end of for(j in 1:J)

    q <- exp(q - max(q))
    q <- q/sum(q)

    if(New.xi[i] != 0){
      Hyp.xi[i] <- New.xi[i]
      log.Q = log.Q + log(q[Hyp.xi[i]])
    }else{
      Hyp.xi[i] <- K+1
      New.xi[xi==xi[i]] <- K+1
      log.Q = log.Q + log(q[K+1])
    }

    if(Pois) Zz=list(Z.sn=U$U.sn, Z.sy=U$U.sy, Z.lpf=U$U.lpf, z.sn=z.sn, z.sy=z.sy, z.lp
   U <- update.suff.xi_i(Zz,Hyp.xi[i],direct,model)

  } ## end of for(i in pi[3:n])
  density = if(log) log.Q else exp(log.Q)

return(density)
}


############    d.Y.given.xi    ########################
# density function of the marginal of Y.j relationship
# given the groups specified by xi
# Input: Y.j, xi, hyperD, hyperOD, direct.j, log = FALSE
# hyper = list(a=1/2, b=1/2, ...)
```

```
# direct.j = F symmetric, T o.w.
# model.j = B binary, P poisson.
# log = TRUE the output is the log(density)
# Output: density or Pr(Y)
################################################

d.Y.given.xi <- function(Y.j, xi, hyperD, hyperOD, direct.j, model.j, log = FALSE){
 n <- dim(Y)[1]
 U <- suff.xi(Y.j,xi,direct.j,model.j)
 K <- max(xi)
 log.pr.y = 0

 if(model.j=='B'){ ## Binary
  if(direct.j){ ### directed
    if(K != 1){
    for( l in 1:(K-1) ){
      log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[l,l,1], U$U.sy[l,l,1], hyperD, log=T) # the
       for( k in (l+1):K ){
         log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[l,k,1], U$U.sy[l,k,1], hyperOD, log=T) +
       }
     }
     log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[K,K,1], U$U.sy[K,K,1], hyperD, log=T) # the
    }else{ ### K = 1
     log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[K,K,1], U$U.sy[K,K,1], hyperD, log=T)
    }
  }else{   ### undirected
    if(K != 1){
    for( l in 1:(K-1) ){
      log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[l,l,1], U$U.sy[l,l,1], hyperD, log=T) # the
```

```
      for( k in (l+1):K ){
        log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[l,k,1], U$U.sy[l,k,1], hyperOD, log=T)
      }
    }
    log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[K,K,1], U$U.sy[K,K,1], hyperD, log=T) # the
   }else{ ### K = 1
    log.pr.y = log.pr.y + dmarg.Y.B(U$U.sn[K,K,1], U$U.sy[K,K,1], hyperD, log=T)
   }
 } ## end of if(direct.j)

}else if(model.j=='P'){ ## Poisson
 if(direct.j){ ### directed
   if(K != 1){
   for( l in 1:(K-1) ){
     log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[l,l,1], U$U.sy[l,l,1], U$U.lpf[l,l,1], hype
      for( k in (l+1):K ){
        log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[l,k,1], U$U.sy[l,k,1], U$U.lpf[l,k,1], h
      }
    }
    log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[K,K,1], U$U.sy[K,K,1],  U$U.lpf[K,K,1], hype
   }else{ ### K = 1
    log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[K,K,1], U$U.sy[K,K,1], U$U.lpf[K,K,1], hyper
   }

 }else{   ### undirected
   if(K != 1){
   for( l in 1:(K-1) ){
     log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[l,l,1], U$U.sy[l,l,1], U$U.lpf[l,l,1], hype
      for( k in (l+1):K ){
```

```
          log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[l,k,1], U$U.sy[l,k,1], U$U.lpf[l,k,1], h
        }
      }
      log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[K,K,1], U$U.sy[K,K,1], U$U.lpf[K,K,1], hyper
     }else{ ### K = 1
      log.pr.y = log.pr.y + dmarg.Y.P(U$U.sn[K,K,1], U$U.sy[K,K,1],  U$U.lpf[K,K,1], hype
     }

   } ## end of if(direct.j)
  } ## end of Poisson

  density = if(log) log.pr.y else exp(log.pr.y)

return(density)
}

### function for update ###
update.pairwise <- function(pairwise,xi){
  n <- length(xi)
  for(i in 1:n){
    for(j in 1:n){
      if(xi[i]==xi[j]){
        pairwise[i,j] <- pairwise[i,j] + 1
      }
    }
  }
  return(pairwise)
}
```

```
### functions for ploting ###
plotrelation <- function(pairwise, n, xi, outfile=NULL){
  colorscale <- c("white", rev(heat.colors(100)))
  axislabels<-as.character(c(1:n))
  subset<-order(xi)
  colorcode<-xi
  min.col <- round(100*min(pairwise))
  max.col <- round(100*max(pairwise))
  if(!is.null(outfile)){
    pdf(outfile)
  }
  par(mar=c(5,4,2,2)+-.1)
  image(seq(1,n), seq(1,n), pairwise[subset,subset], col=colorscale[min.col:max.col], xl
  axis(1,at=seq(1,n),labels= FALSE)
  mtext(side=1, text= axislabels[subset], at=seq(1,n), las=2, cex=.75, line = 1, col=col
  axis(2,at=seq(1,n),labels= FALSE)
  mtext(side=2, text= axislabels[subset], at=seq(1,n), las=2, cex=.75, line = 1, col=col
  box()
  if(!is.null(outfile)){
    dev.off()
  }
}

##########
##to compute the likelihood for multiple relations

dalpha.prior <- function(alpha, beta, zeta, a.alpha, b.alpha, log=F){
  z <- dgamma(alpha, shape=a.alpha, rate=b.alpha, log=T)
  for(k in 1:max(zeta)){
```

```
    z <- z + dgamma(beta[zeta==k][1], shape=a.alpha, rate=b.alpha, log=T)
  }
  if(log==F){
    z <- exp(z)
  }
return(z)
}

dDP <- function(xi, alpha, log=F){
  I <- length(xi)
  m.l <- table(xi)
  L <- length(m.l)
  z <- lgamma(alpha) - lgamma(alpha+I) + L*log(alpha) + sum(lgamma(m.l))
  if(log==F){
    z <- exp(z)
  }
return(z)
}

deta <- function(zeta, xi, alpha, beta, log=F){
  z <- dDP(xi=zeta, alpha=alpha, log=T)
  for(k in 1:max(zeta)){
    z <- z + dDP(xi=as.vector(xi[k,]), alpha=beta[zeta==k][1], log=T)
  }
  if(log==F){
    z <- exp(z)
  }
return(z)
}
```

```
dY.given.eta <- function(y, zeta, xi, RR,  hyperD, hyperOD, log=F){
  z = 0
  for(r in 1:RR){
   LL <- max(xi[zeta[r],])
    for(k in 1:LL){
      for(l in 1:LL){
       if(k==l){
         a.base=hyperD$a.bin[r]
         b.base=hyperD$b.bin[r]
       }else{
         a.base=hyperOD$a.bin[r]
         b.base=hyperOD$b.bin[r]
       }
        ytemp <- as.vector(y[xi[zeta[r],]==k, xi[zeta[r],]==l, r])
        ytemp <- ytemp[!is.na(ytemp)]  #Remember that Y has diagonal elements that are N
        ysum <- sum(ytemp)
        yn <- length(ytemp)
        if(yn!=0){
          z = z + lbeta(a.base + ysum, b.base + yn - ysum) - lbeta(a.base,b.base)
        }
      }
    }
  }
  if(log==F){
    z <- exp(z)
  }
return(z)
}
```

```
#
# File: SAt_CollectV3.R
#
# Data: 01/27/2011
# Modification to include different models for Y_{i,i',j}
#
# Date: 12/01/2010
# Functions to split/merge collections of relational matrices
# runs "t" Gibbs updates after copy/generating xi.a, xi.b & xi.ab

# Model
#  Binomial Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Bernoulli(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Beta(hyperD$.a.bin,hyperD$b.bin)   l=k
#    Beta(hyperOD$.a.bin,hyperOD$b.bin) l \neq k
#
#  Poisson Data
#  Y_{i,i',j} | zeta, XI, Theta ~ Poisson(theta_{xi_{zeta_j,i},xi_{zeta_j,i'},j})
# theta_{lk,j} ~ Gamma(hyperD$a.pois,hyperD$b.pois)   l=k
#    Gamma(hyperOD$a.pois,hyperOD$b.pois) l \neq k

# NOTE: For computations requires:
# source("funcSuffStats.R")
# source("funcModel.R")
# source("SAMS.R")
# source("funcGibbsUpdate.R")
# source("Collections.R")
```

```
######    SAt_Collect function  ####################
# makes 1 step of Sequential Allocation algorithm
#   with "t" Gibbs updates in between
# Input: t, Y, zeta, xi, hyperD, hyperOD, direct, model
# Y array of matrices
#     hyper = list(a,b, eta)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: new.zeta = list(new.zeta, new.XI, new.hyperD, new.hyperOD, MH.ratio)
################################################

SAt_Collect <- function(t, Y, zeta, XI, hyperD, hyperOD, direct, model){
  J <- dim(Y)[3]
  rpair = sample(1:J, 2)
  a = rpair[1] ## subject index;
  b = rpair[2]

  if( zeta[a] == zeta[b] ){
    move = func.SPLITt_Collect(t, a, b, Y, zeta, XI, hyperD, hyperOD, direct, model)
  }else{
   move = func.MERGEt_Collect(t, a, b, Y, zeta, XI, hyperD, hyperOD, direct, model)
  }

  new.zeta = SA.MH_update(a,b,move, Y, zeta, XI, hyperD, hyperOD, direct, model)

return(new.zeta)
```

```
}


###############  SA.MH_update ##################
# computes MH ratio and decides the new state
# Input:  a,b,move, Y, zeta, XI, hyperD, hyperOD, direct, model
# Y array of matrices
# move = list(S.ab, S.a, S.b, xi.ab, xi.a, xi.b,
#                   eta.a, eta.b, eta.ab,
# Q.xi.ab, Q.xi.a, Q.xi.b, split)
# hyper = list(a,b, eta, a.alpha, b.alpha, eta.zeta, ETA.XI)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: list(new.zeta, new.XI, new.hyperD, new.hyperOD, MH.ratio)
################################################

SA.MH_update <- function(a,b,move, Y, zeta, XI, hyperD, hyperOD, direct, model){

  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }
  R = max(zeta)
  n.a = length(move$S.a)
  n.b = length(move$S.b)
  m.a = table(move$xi.a)
```

```
m.b = table(move$xi.b)
m.ab = table(move$xi.ab)
log.p.xi.a = sum(lgamma(m.a)) + length(m.a)*log(move$eta.a) + lgamma(move$eta.a) - lga
log.p.xi.b = sum(lgamma(m.b)) + length(m.b)*log(move$eta.b) + lgamma(move$eta.b) - lga
log.p.xi.ab = sum(lgamma(m.ab)) + length(m.ab)*log(move$eta.ab) + lgamma(move$eta.ab)

log.pr.aORb = log.p.xi.a + log.p.xi.b
log.pr.ab = log.p.xi.ab

for(j in move$S.ab){
  hypD = list(a.bin=hyperD$a.bin[j], b.bin=hyperD$b.bin[j], a.pois=hyperD$a.pois, b.po
  hypOD = list(a.bin=hyperOD$a.bin[j], b.bin=hyperOD$b.bin[j], a.pois=hyperD$a.pois, b
 log.pr.ab = log.pr.ab + d.Y.given.xi(Y[,,j], move$xi.ab, hypD, hypOD, direct[j], mode
 if( sum(j == move$S.a) > 0){ # j-th relation is in S.a
   log.pr.aORb = log.pr.aORb + d.Y.given.xi(Y[,,j], move$xi.a, hypD, hypOD, direct[j],
 }else{  # j-th relation is in S.b
   log.pr.aORb = log.pr.aORb + d.Y.given.xi(Y[,,j], move$xi.b, hypD, hypOD, direct[j],
 }
}

log.Q.split = log(move$Q.xi.a) + log(move$Q.xi.b)
log.Q.merge = log(move$Q.xi.ab) + (n.a + n.b - 2)*log(2)

log.merged.ratio = (log.pr.ab + log.Q.split) - (lbeta(n.a,n.b) + log(hyperD$eta.zeta)
MH.ratio = min( 1, exp(log.merged.ratio*(-1)^move$split) )

new.zeta = zeta
new.XI = XI
new.ETA.XI = hyperD$ETA.XI
```

```
  if( rbinom(1, size=1, p=MH.ratio) ) {
   if( move$split ){
     new.XI <- XI[-zeta[a],]
     new.XI <- rbind(new.XI,move$xi.a,move$xi.b)
     new.zeta[move$S.a] = R+1
     new.zeta[move$S.b] = R+2
     new.zeta = as.numeric(factor(new.zeta))
     new.ETA.XI[move$S.a] = move$eta.a
     new.ETA.XI[move$S.b] = move$eta.b
   }else{
     new.XI <- XI[-c(zeta[a],zeta[b]),]
     new.XI <- rbind(new.XI,move$xi.ab)
     new.zeta[move$S.ab] = R+1
     new.zeta = as.numeric(factor(new.zeta))
     new.ETA.XI[move$S.ab] = move$eta.ab
   }
  }
  hyperD$ETA.XI = new.ETA.XI
  hyperOD$ETA.XI = new.ETA.XI

return(list(new.zeta=new.zeta, new.XI=new.XI, new.hyperD=hyperD, new.hyperOD=hyperOD, MH
}


########  SPLITt_Collect  ############################
# splits a collection and saves Pr(split)
#  with "t" Gibbs updates
# Input: t, a, b, Y, zeta, XI, hyperD, hyperOD, direct, model
# Y array of matrices
```

```
#     hyper = list(a,b, eta, eta.zeta, ETA.XI)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: move = list(S.ab, S.a, S.b, xi.ab, xi.a, xi.b, eta.a, eta.b, eta.ab, Q.xi.ab,
#########################################################

func.SPLITt_Collect <- function(t, a, b, Y, zeta, XI, hyperD, hyperOD, direct, model){

  zeta_ab = zeta[a]  ## group index;

  n <- dim(Y)[1]
  if(length(dim(Y))<3){
    J <- 1
    Y <- array(Y,dim=c(n,n,J))
  }else{
    J <- dim(Y)[3]
  }

  S.ab = (1:J)[zeta == zeta_ab]
  hyperD$eta <- hyperOD$eta <- eta.ab <- hyperD$ETA.XI[a]   ## group parameter;
  xi.ab = XI[zeta_ab,]
  Q.xi.ab = Prob.Initial.xi(Y[,,S.ab], xi.ab, hyperD, hyperOD, direct[S.ab], model[S.ab]
  for( iter in 1:t ){
    U <- suff.xi(Y[,,S.ab],xi.ab,direct[S.ab], model[S.ab])
    Assig <- sample.xi(Y[,,S.ab], xi.ab, U, hyperD, hyperOD, direct[S.ab], model[S.ab])
    xi.ab <- Assig$xi
  }
  Q.xi.ab <- d.alpha(eta.ab,hyperD)*Q.xi.ab*Assig$Q.xi
```

```
S.a = a;
S.b = b;
S.ab.small = S.ab[S.ab !=a & S.ab != b]  ## removes a & b
if(length(S.ab.small)<=1){ permuted.S.ab = S.ab.small }else{ permuted.S.ab = sample(S.

for (j in permuted.S.ab) {
  if( rbinom(1,p=1/2, size=1) ){
    S.a = c(S.a,j)
  }else{
    S.b = c(S.b,j)
  }
} # end of for (j in permuted.S.ab)

fixed.pi = sample(n)
hyperD$eta <- hyperOD$eta <- eta.a <- gen.alpha(hyperD)
xi_Q.a = Initial.xi(Y[,,S.a], hyperD, hyperOD, direct[S.a], model[S.a], pi=fixed.pi)
xi.a = xi_Q.a$xi
Q.xi.a = xi_Q.a$Q.xi
for( iter in 1:t ){
  U <- suff.xi(Y[,,S.a],xi.a,direct[S.a],model[S.a])
  Assig <- sample.xi(Y[,,S.a],xi.a,U,hyperD,hyperOD,direct[S.a],model[S.a])
  xi.a <- Assig$xi
}
Q.xi.a <- d.alpha(eta.a,hyperD)*Q.xi.a*Assig$Q.xi

hyperD$eta <- hyperOD$eta <- eta.b <- gen.alpha(hyperD)
xi_Q.b = Initial.xi(Y[,,S.b],hyperD, hyperOD, direct[S.b], model[S.b], pi=fixed.pi)
xi.b = xi_Q.b$xi
```

```
  Q.xi.b = xi_Q.b$Q.xi
  for( iter in 1:t ){
    U <- suff.xi(Y[,,S.b],xi.b,direct[S.b],model[S.b])
    Assig <- sample.xi(Y[,,S.b],xi.b,U,hyperD,hyperOD,direct[S.b],model[S.b])
    xi.b <- Assig$xi
  }
  Q.xi.b <- d.alpha(eta.b,hyperD)*Q.xi.b*Assig$Q.xi

return(move = list(S.ab = S.ab, S.a = S.a, S.b = S.b, xi.ab=xi.ab, xi.a=xi.a, xi.b=xi.b,
}


########  MERGEt_Collect  ##########################
# merges collections and computes Pr(imaginary split)
#  with "t" Gibbs updates
# Input: t, a, b, Y, zeta, XI, hyperD, hyperOD, direct, model
# Y array of matrices
#     hyper = list(a,b, eta, eta.zeta, ETA.XI)
# direct = c(F,..) F symmetric, T o.w.
#     model = c("B", "P") B binary, P poisson.
# one for each matrix of Y
# Output: move = list(S.ab, S.a, S.b, xi.ab, xi.a, xi.b, eta.a, eta.b, eta.ab, Q.xi.ab,
#########################################################

func.MERGEt_Collect <- function(t, a, b, Y, zeta, XI, hyperD, hyperOD, direct, model){

  zeta_a = zeta[a]
  zeta_b = zeta[b]
  n <- dim(Y)[1]
```

```
if(length(dim(Y))<3){
  J <- 1
  Y <- array(Y,dim=c(n,n,J))
}else{
  J <- dim(Y)[3]
}
S.a = (1:J)[zeta == zeta_a]
S.b = (1:J)[zeta == zeta_b]

fixed.pi = sample(n)
hyperD$eta <- hyperOD$eta <- eta.a <- hyperD$ETA.XI[a]
xi.a = XI[zeta_a,]
Q.xi.a = Prob.Initial.xi(Y[,,S.a], xi.a, hyperD, hyperOD, direct[S.a], model[S.a], log
for( iter in 1:t ){
  U <- suff.xi(Y[,,S.a],xi.a,direct[S.a],model[S.a])
  Assig <- sample.xi(Y[,,S.a],xi.a,U,hyperD,hyperOD,direct[S.a],model[S.a])
  xi.a <- Assig$xi
}
Q.xi.a <- d.alpha(eta.a,hyperD)*Q.xi.a*Assig$Q.xi

hyperD$eta <- hyperOD$eta <- eta.b <- hyperD$ETA.XI[b]
xi_Q.b = Initial.xi(Y[,,S.b],hyperD,hyperOD,direct[S.b],model[S.b],pi=fixed.pi)
xi.b = XI[zeta_b,]
Q.xi.b = Prob.Initial.xi(Y[,,S.b], xi.b, hyperD, hyperOD, direct[S.b], model[S.b], log
for( iter in 1:t ){
  U <- suff.xi(Y[,,S.b],xi.b,direct[S.b],model[S.b])
  Assig <- sample.xi(Y[,,S.b],xi.b,U,hyperD,hyperOD,direct[S.b],model[S.b])
  xi.b <- Assig$xi
}
```

```
  Q.xi.b <- d.alpha(eta.b,hyperD)*Q.xi.b*Assig$Q.xi

  S.ab = c(S.a,S.b)
  hyperD$eta <- hyperOD$eta <- eta.ab <- gen.alpha(hyperD)
  xi_Q.ab = Initial.xi(Y[,,S.ab],hyperD,hyperOD,direct[S.ab],model[S.ab])
  xi.ab = xi_Q.ab$xi
  Q.xi.ab = xi_Q.ab$Q.xi
  for( iter in 1:t ){
    U <- suff.xi(Y[,,S.ab],xi.ab,direct[S.ab],model[S.ab])
    Assig <- sample.xi(Y[,,S.ab],xi.ab,U,hyperD,hyperOD,direct[S.ab],model[S.ab])
    xi.ab <- Assig$xi
  }
  Q.xi.ab <- d.alpha(eta.ab,hyperD)*Q.xi.ab*Assig$Q.xi

return(move = list(S.ab = S.ab, S.a = S.a, S.b = S.b, xi.ab=xi.ab, xi.a=xi.a, xi.b=xi.b,
}



##Heatmap Functions######
## MACROS
clust2matrix <- function(ind,JJ){
  M <- matrix(0,JJ,JJ)
  for(i in 1:(JJ-1)){
    for(j in (i+1):JJ){
      if(ind[i]==ind[j]){
        M[i,j] <- 1
      }
    }
  }
```

```
  return(M)
}

optimalclust <- function(pairwise, K0, initclust, JJ){
  OO <- pairwise/max(pairwise) - K0
  clust <- initclust
  oclust <- clust
  oclust[1] <- oclust[1]+1

  while(!all(oclust==clust)){
    oclust <- clust
    for(i in 1:JJ){
      sclust <- clust[-i]
      nclust <- rep(0,JJ)
      nclust[-i] <- as.numeric(factor(sclust))
      nc <- max(nclust)
      ut <- rep(0,nc+1)
      for(k in 1:(nc+1)){
        nclust[i] <- k
        MM <- clust2matrix(nclust,JJ)
        ut[k] <- sum(MM*OO)
      }
      nclust[i] <- order(ut,decreasing=T)[1]
      clust <- nclust
      #print(i)
    }
    #print("iter")
  }
  return(clust)
```

```
}

reorder.pairwise.alt <- function(pairwise,start,JJ){
  nused <- 1
  used <- start
  nnonused <- JJ-1
  nonused <- seq(1,JJ)[-start]
  for(l in 2:JJ){
    dista <- rep(0,nnonused)
    for(j in 1:nnonused){
      #print(j)
      dista[j] <- pairwise[used[l-1],nonused[j]]
    }
    maxdist.or <- order(dista,decreasing=T)
    used <- c(used,nonused[maxdist.or[1]])
    nonused <- nonused[-maxdist.or[1]]
  }
  return(used)
}

heatmapplot <- function(pairwise, used, olab, subsetaxis, outfile=FALSE, llc=FALSE){
  JJ <- dim(pairwise)[1]
#  x11()
  colorscale <- c("white",rev(heat.colors(100)))
  min.col <- round(100*min(pairwise))+1
  max.col <- round(100*max(pairwise))+1
  # nf <- layout(matrix(c(1,2),nrow=1,ncol=2), c(7,1), TRUE)
  par(mar=c(2.5,2.5,0.5,0.5))
  #Display heatmap
```

```
  image(seq(1,JJ),seq(1,JJ),pairwise[used,used], axes=F, xlab="", ylab="", col=colorscal
  axis(1,at=subsetaxis,labels=(olab[used])[subsetaxis],las=2,cex.axis=0.5)
  axis(2,at=subsetaxis,labels=(olab[used])[subsetaxis],las=2,cex.axis=0.5)
  box()
  #Add squares marking the clusters
  if(any(llc!=FALSE)){
    for(i in 1:(length(llc)-1)){
      rect(llc[i]+0.5,llc[i]+0.5,llc[i+1]+0.5,llc[i+1]+0.5,border="black")
    }
  }
  #Display color scale
  # par(mar=c(3,0,0,0))
  # plot(1:100,1:100,xlim=c(0,2),ylim=c(0,100),type="n",axes=F,xlab ="",ylab ="")
  # yposr <- (1:100)
  # rect(0, yposr-.5, 0.5, yposr+.5,col = c("white",rev(heat.colors(100))), border=F)
  # rect(0, .5, 0.5, 100.5,col = "transparent")
  # text(0.42,c(yposr[1],yposr[25],yposr[50],yposr[75],yposr[100]), c("0","0.25","0.5","
  if(outfile!=FALSE){
    dev.print(device = pdf, outfile)
  }
}
```

Appendix C

Sampling And Projections

Codes for creating a variety of RDS samples.

```
#This is for creating a simulation of RDS process
#Need Rtools and devtools to install "rdssim" package
setwd("C:/Users/zhihe/Documents/FB")
library("rdssim", lib.loc="~/R/win-library/3.4")
library("Matrix")

#Load adjacency matrix
load("AdMatrix.Rdata")
#Need a symmetric adjacency matrix
Y[lower.tri(Y)] <- t(Y)[lower.tri(Y)]
adj <- as.matrix.data.frame(Y)
#adj[1:10,1:10]
set.seed(335)
#Refer friends based on the number of friends they have that you don't know and you have
adjL <- adj2list(adj, ac.type = "Both")

#simulate the rds process 100 times
```

```
rds <- 100                              #number of samples
sam_size <- 103                         #sample size
sam_size <- sam_size*2                  #sample size*design effect (DE)
seedNode <- sample(1:1034, rds ,replace=F)       #create random starting seeds
rseed <- sample(1:1000, rds)      #seed for reproductive results
rdssam <- array(0,dim = c(sam_size,4,rds))
for (i in 1:rds){
rdssam[,,i] <- rdssim(adjL, referral.type = "acRW", wRreplacement = TRUE,
       nSamples=sam_size, nReferrals=2, seedNode=seedNode[i], rseed=rseed[i])
}
# head(rdssam[,,1])

#############################################################################
#In this case, RDS samples can only be create in a certain way,            #
#due to the fact that some nodes are too isolated to create enough samples.#
#Single anti clustering referal type will lead isolated nodes              #
#############################################################################

#get adjacency matrix for each sample
load("AdMatrix.Rdata")
sample_id <- matrix(,nrow = 100, ncol = sam_size)
for (i in 1:rds){
  sample_id[i,] <- sort(rdssam[,2,i], decreasing = F)
}
samY_rds <- array(0,dim = c(sam_size,sam_size,rds))
samy_rds <- array(0,dim = c(sam_size,sam_size,rds))
for (i in 1:rds){
  samY_rds[,,i] <- as.matrix(Y[sample_id[i,],sample_id[i,]])
  samy_rds[,,i] <- as.matrix(y[sample_id[i,],sample_id[i,],1])
```

```
}

# check degree distributions
# library(igraph)
# x=0:130
# y=rep(0,131)
# plot(x,y,xlim=c(0,130), ylim=c(0,.05),type="n",
#      ,xlab="Degree",ylab="Frequency",font.lab=2,font.main=2,
#      cex.lab=1.5,cex.main=1.5,cex.lab=1.5,cex=1.5)
# for (j in 1:100){
#   graph.Y=graph.adjacency(samY_rds[,,j],mode='upper',weighted=NULL,diag=FALSE)
#   degree=density(degree(graph.Y,mode='all'))
#   lines(degree,col=j)
#   }

save(rdssam, samY_rds, samy_rds, file = "sampling/RDSampling(DE=2).Rdata")
```

Codes for MCMC sampler for each sample

```
# To run in batch mode:
#    nohup R CMD BATCH --no-save ~/FB_MCMC_srsam.R ~/FB_MCMC_srsam.Rout &
#    nohup R CMD BATCH --no-save ~/AC-RDS_projection.R ~/AC-RDS_projection.Rout &
## This is the code for MCMC with 100 sample matrices (SRS/AC-RDS)
## To save time, 100 cores machine are recommended (finished in 9ish hours)
## Outputs are raw matrix that need to be sorted


#setwd("C:/Users/zhihe/Documents/FB")
library("Matrix")
```

```
library(foreach)
library(doMC)
registerDoMC()

set.seed(123)
#Load simple radom sample adjacency matrix
#load("sampling/SRSampling.Rdata")
#Load AC-RDS adjacency matrix
load("sampling/RDSampling.Rdata")

#source codes from the other R files in the working directory
source("functions/SAMSv2.R")
source("functions/funcCollections.R")
source("functions/SAt_CollectV3.R") #runs Sequential Allocation for zeta with "t" Gibbs

## MCMC ##
N <- 1000
burn.in <- 500
keep.each <- 5
print.each <- 200
N.iter <- burn.in + N * keep.each

# set parameters and initial Y & y
# For SRS
# n <- dim(samY_srs)[1]
# r <- dim(samY_srs)[3]
# For AC-RDS
n <- dim(samY_rds)[1]
r <- dim(samY_rds)[3]
```

```
#r <- 3   # To run a smaller test
direct = F
model = 'B'



# Measure system time
system <- system.time({
#run each sample matrix in each core
results <- foreach (i = 1:r,.combine = rbind) %dopar% {

Y <- matrix(0, n, n)
y <- array(0, dim = c(n, n, 1))
# For SRS
# y[,,1] <- samy_srs[,,i]
# Y <- samY_srs[,,i]
# For AC-RDS
y[,,1] <- samy_rds[,,i]
Y <- samY_rds[,,i]

## Hyperparamters ##
alpha.a = beta.a = alpha.b = beta.b = 2
hyperD = hyperOD = list(
  a.bin = rgamma(1, shape = alpha.a, rate = beta.a),
  b.bin = rgamma(1, shape = alpha.b, rate = beta.b),
  alpha.a = alpha.a,
  beta.a = beta.a,
  alpha.b = alpha.b,
  beta.b = beta.b,
  kappa = 0.75,
```

```
  a.pois = 1,
  b.pois = 2 / 5,
  eta = 0.5,
  a.alpha = 1,
  b.alpha = 1
)

### Testing XI Gibbs, setting initial XI to (1,1,...,1) ###
XI <- matrix(0,1,n)
# Assig <- Initial.xi(Y, hyperD, hyperOD, direct, model)
# XI[1, ] <- Assig$xi
XI[1,] = as.numeric(rep(1,n))

## Set empty matrices ##
groups.N = array(0, dim = c(n, n, 1))
groups.theta.N = array(0, dim = c(n, n, 1))
# eta.N <- matrix(0, N, 1)
# aD.N <- matrix(0, N, 1)
# bD.N <- matrix(0, N, 1)
# aOD.N <- matrix(0, N, 1)
# bOD.N <- matrix(0, N, 1)
# llk.out <- matrix(0, N + burn.in, 1)
# iter.llk <- 0
# llk.out2 <- matrix(0, N.iter, 1)
simul = 0


#start MCMC
for (iter in 1:N.iter) {
```

```
#updates XI
U <- suff.xi(Y, XI[1, ], direct, model)
Assig <- sample.xi(Y, XI[1, ], U, hyperD, hyperOD, direct, model)
XI[1,] <- Assig$xi

#updates theta's hyper parameters
new.hyp <- sample.lambda(Y, 1, XI , hyperD, hyperOD, direct, model)
hyperD <- new.hyp$hyperD
hyperOD <- new.hyp$hyperOD

# llk.out2[iter,1] <-
#   dY.given.eta(y, 1, XI , 1, hyperD, hyperOD, log = T) + deta(1, XI , hyperD$eta, hy
#                                                                T) + dalpha.prior(hyp
#                                                                                  hyp
#                                                                                  1,
#                                                                                  hyp
#                                                                                  hyp
#                                                                                  log

## Keeping only the interesting values
if (iter <= burn.in) {
  #iter.llk = iter.llk + 1
  #llk.out[iter.llk,1] <- llk.out2[iter,1]
  if ((iter == burn.in)&(i %% 5 == 0)){
    print(paste("burn-in",i))
  }
} else if ((iter > burn.in) &
           ((iter - burn.in) %% keep.each == 0)) {
  simul = simul + 1
```

```
#iter.llk = iter.llk + 1
#llk.out[iter.llk,1] <- llk.out2[iter,1]
# eta.N[simul,1] <- hyperD$eta
# aD.N[simul,1] <- hyperD$a.bin
# bD.N[simul,1] <- hyperD$b.bin
# aOD.N[simul,1] <- hyperOD$a.bin
# bOD.N[simul,1] <- hyperOD$b.bin
Theta <- sample.Theta(Y, 1, XI, hyperD, hyperOD, direct, model)
groups = array(0, dim = c(n, n))
groups.theta = array(0, dim = c(n, n))
L <- max(XI[1,])
for (l in 1:L) {
  subset <- (1:n)[XI[1, ] == l]
  groups[subset, subset] = 1
  groups.theta[subset, subset] <- Theta[l, l, 1]
  for (lp in min(L, (l + 1)):L) {
    subset.lp <- (1:n)[XI[1,] == lp]
    groups.theta[subset, subset.lp] <- Theta[l, lp, 1]
    groups.theta[subset.lp, subset] <- Theta[lp, l, 1]
  }
}
groups.N[,,1] = groups.N[,,1] + groups
groups.theta.N[,,1] = groups.theta.N[,,1] + groups.theta
diag(groups.theta.N[,,1]) = NA

#to let me know how far it has gone
if((simul %% print.each == 0) & (i %% 5 == 0)){
  print(paste(simul,i))
}
```

```
  }
}

# list(XI[1,],groups.theta.N[,,1],groups.N[,,1])
rbind(c(i,XI[1,]), cbind(rep(i,n),groups.theta.N[,,1]), cbind(rep(i,n),groups.N[,,1]))
# llk.out[,1]
}
# for (rr in 1:100){
#   names(results[[rr]])=c('XI','Theta','groups')
# }
})

#save(eta.N,aD.N,bD.N,aOD.N,bOD.N,groups.N,llk.out,groups.theta.N,xi,file = "results/FB_
save(system, results, file = "results/results_matrix_RDS.Rdata")

# setwd("C:/Users/zhihe/Documents/FB")
# setwd("C:/Lil john/graduate project")
# library("igraph")
# ed107 <- read.delim("facebook/107.edges", header = F, sep = "")
# G <- graph.data.frame(ed107, directed = FALSE)
# Y.sym <- as_adjacency_matrix(G,
#                              type = "both",
#                              names = TRUE,
#                              attr = NULL)
source("functions/HeatmapFunction.R")
load("AdMatrix.Rdata")
pairwise <- read.table("C:/Lil john/graduate project/6-25-2018/Archive/pairwise_xi_v5_Rd
# pairwise <- read.table("pairwise_xi_v5_Rdm2.txt", sep = ",")
pairwise <- as.matrix(pairwise[1:1034,1:1034])
```

```
diag(pairwise) <- 1
load("MP_optclust.Rdata")

n <- nrow(Y)
Z <- as.matrix(Z)

# llk <- read.table("C:/Lil john/graduate project/6-25-2018/Archive/llik_v5_Rdm2.txt")
# llk <- as.vector(llk)
# dim(llk)
# pdf("heatmap/loglike.pdf")
# plot(llk[,1],pch=20, ylab="", xlab="", main="loglikelihood of Y")
# dev.off()

f.name.Raw.G = "heatmap/Raw_GibbsCRP.pdf"
f.name.xi.G = "heatmap/xi_GibbsCRP(K=.1).pdf"



initclust <- rep(1, n)
olab = c(1:n)
subsetaxis = c(1:n)
date()
optclust <- optimalclust(pairwise, K0 = .1, initclust, n)
date()
save(olab,subsetaxis,optclust,file = "MP_optclust.Rdata")
used <- NULL
for (i in 1:length(unique(optclust))) {
  subs <- seq(1:n)[optclust == i]
  if (length(subs) == 1) {
    used <- c(used, subs)
```

```
  } else{
    pp    <- pairwise[subs, subs]
    start <-
      ceiling(order(lower.tri(pp, diag = F) * pp, decreasing = T)[1] / length(subs))
    ut     <- reorder.pairwise.alt(pp, start, length(subs))
    used <- c(used, subs[ut])
  }
}
ll   <- c(0, table(optclust[used]))
llc <- cumsum(ll)
date()
system.time({
heatmapplot(pairwise,
            used,
            olab,
            subsetaxis,
            outfile = f.name.xi.G,
            llc = llc)
heatmapplot(Z,
            used,
            olab,
            subsetaxis,
            outfile = f.name.Raw.G,
            llc = llc)
})
date()
```

Codes for projection of 100 RDS samples

```
## This is the code for:
```

```
#   sorting the raw matrices of SRS from FB_MCMC_srsam.R;
#   projecting Theta matrix;
#   Generate Y matrix;
#   Calculate degree distribution;

# setwd("C:/Users/zhihe/Documents/FB")
library(igraph)
library(foreach)
library(doMC)
registerDoMC()
load("sampling/results_matrix_RDS.Rdata")
load("sampling/RDSampling.Rdata")
############### Sorting raw result matrices #############
system <- system.time({
  n <- 103
  XI100 <- matrix(, nrow = 100, ncol = n)
  Theta100 <- array(, dim = c(n,n,100))
  Group100 <- array(, dim = c(n,n,100))
  d <- 2*n+1
  for (i in 1:(d*100)){
    if (i %% d == 1){
      XI100[(i%/%d+1),] <- results[i,2:(n+1)]
    }
    else if ((i %% d >= 2)&(i%%d <= (n+1))){
      Theta100[(i%%d-1), ,(i%/%d+1)] <- results[i,2:(n+1)]
    }
    else if ((i %% d >= (n+1))&(i %% d <= (d-1))){
      Group100[(i%%d-(n+1)), ,(i%/%d+1)] <- results[i,2:(n+1)]
    }
```

```
    else if (i %% d == 0){
      Group100[n,,i %/% d] <- results[i,2:(n+1)]
    }
  }

#################### AC-RDS #########################

  RDS.final.results <- foreach (rr = 1:100)%dopar%{
    N=1034
    #xf=N/n
    groups.theta.N=Theta100[,,rr]/1000
    diag(groups.theta.N)=0
    groups.N <- Group100
    # Degree distribution
    graph.y=graph.adjacency(samY_rds[,,rr],mode='upper',weighted=NULL,diag=FALSE)
    dd=degree(graph.y)
    # Check if dd is sequential
    #for (i in 1:10){print(length(which(samY_rds[i,,1]==1)))}

    # Projection factors
    #xf <- matrix(, nrow = 1, ncol = 103)
    xf <- rep(0,n)
    for (i in 1:n){
        xf[i] <- round(N/(sum(1/dd)*dd[i]), digits = 0)
    }
    out.matrix=matrix(0,sum(xf),sum(xf))
    # Check what xf add up to
    N <- sum(xf)
```

```
for ( i in 1:n){

    start=sum(xf[i-1:i])+1
    end=sum(xf[i-1:i])+xf[i]
    m=start:end
    # for ( j in 1:xf[i]){
    new.out=rep(groups.theta.N[i,],xf)

    out.matrix[start:end,]=matrix(rep(new.out,each=xf[i]),xf[i],sum(xf))

    diag(groups.N[,,rr])=0

    index=groups.N[i,,rr]==max(groups.N[i,,rr])


    out.matrix[m,m]=mean(groups.theta.N[i,index])
  }

diag(out.matrix)=0
################################################

set.seed(123)
# results.y=rep(0,100)
# degree.y=list()
#for (k in 1:100){

  Y=matrix(NA,N,N)

  for (i in 1:(N-1)){
```

```
        #set.seed(i+22)

        for (j in (i+1):N){
          Y[i,j]=rbinom(1,1,out.matrix[i,j])
          Y[j,i]=Y[i,j]
          # To let me know how far it goes
          if ((i %% 10 == 0) & (rr %% 10 == 0)){print(paste(rr,i))}
        }}
          diag(Y)=0
      graph.y=graph.adjacency(Y,mode='upper',weighted=NULL,diag=FALSE)
      dd=degree(graph.y)

      degree.y=table(dd)
      rdegree.y=table(dd/sum(dd))
      results.y=sum(Y)
      # Mean distance path
      me.di=mean_distance(graph.y, directed = FALSE, unconnected = TRUE)
      # Average clustering coefficient
      trans=transitivity(graph.y, type = "average")
    #}
    list(XI100, Theta100, Group100, xf, out.matrix, degree.y,rdegree.y, me.di, trans, re
  }
  for (rr in 1:100){
    names(RDS.final.results[[rr]])=c('XI100', 'Theta100', 'Group100', 'project.factor','
  }
  })

save(system, RDS.final.results,file="results/RDS.final.results.Rdata")
```

Codes for making boxplot with true degree distribution in percentages

```
# setwd("C:/Users/zhihe/Documents/FB")
load("sampling/adjust/SRS.final.results.Rdata")
load("sampling/adjust/RDS.final.results.Rdata")
load("sampling/adjust/ACRDS.final.results.Rdata")
load("sampling/adjust/ACRDS.final.results(DE=2).Rdata")
load("AdMatrix.Rdata")
library(igraph)



# True degree distribution
graph.Y=graph.adjacency(Y,mode='upper',weighted=NULL,diag=FALSE)
rdd.Y=table(degree(graph.Y))/length(degree(graph.Y))
LINK=as.numeric(names(rdd.Y))

################ Simple Random Sampling ######################
pdf("rquartilewithtrue(SRS).pdf",width=12,height=8)

cc=rep(0,100)
S=rep(0,100)
for (i in 1:100){
  cc[i] <- max(as.numeric(names(SRS.final.results[[i]]$degree)))
  # for relative degrees
  S[i] <- sum(SRS.final.results[[i]]$degree)
}
cc=max(cc)
```

```
degree.boxplot=matrix(,100,cc+1)

for (j in 1:cc+1){
  degree=NULL
  for (i in 1:100){
    degree=c(degree,SRS.final.results[[i]]$degree[names(SRS.final.results[[i]]$degree)==
  }
  # length(degree)
  if (length(degree)!=0){
    degree.boxplot[1:length(degree),j]=as.vector(degree)
  }
}
# x <- rowSums(degree.boxplot)
# x11()
boxplot(degree.boxplot[1:100,1:249], range = 1.5,main="Boxplot of Degree Distribution fr
lines(rdd.Y~LINK,type='l',col='red',lwd=2,lty=1)
legend("topright", "True Degree Distribution of Friendship Network"
       , col = "red",lty = 1,lwd = 2 ,text.font=2,cex=.8)

dev.off()



######################### RDS ############################
pdf("rquartilewithtrue(RDS).pdf",width=12,height=8)
cc=rep(0,100)
S2 <- rep(0,100)
for (i in 1:100){
  cc[i] <- max(as.numeric(names(RDS.final.results[[i]]$degree)))
```

```
  S2[i] <- sum(RDS.final.results[[i]]$degree)
}
cc=max(cc)
# cc=253

degree.boxplot=matrix(,100,cc+1)

for (j in 1:cc+1){
  degree=NULL
  for (i in 1:100){
    degree=c(degree,RDS.final.results[[i]]$degree[names(RDS.final.results[[i]]$degree)==
  }
  # length(degree)
  if (length(degree)!=0){
    degree.boxplot[1:length(degree),j]=as.vector(degree)
  }
}

# x11()
boxplot(degree.boxplot[1:100,1:253], range = 1.5,main="Boxplot of Degree Distribution fr
lines(rdd.Y~LINK,type='l',col='red',lwd=2,lty=1)
legend("topright", "True Degree Distribution of Friendship Network"
       , col = "red",lty = 1,lwd = 2 ,text.font=2,cex=.8)

dev.off()


######################### AC-RDS ############################
pdf("rquartilewithtrue(ACRDS).pdf",width=12,height=8)
```

```
cc=rep(0,100)
S3 <- rep(0,100)
for (i in 1:100){
  cc[i] <- max(as.numeric(names(ACRDS.final.results[[i]]$degree)))
  S3[i] <- sum(ACRDS.final.results[[i]]$degree)
}
cc=max(cc)
# cc=253

degree.boxplot=matrix(,100,cc+1)

for (j in 1:cc+1){
  degree=NULL
  for (i in 1:100){
    degree=c(degree,ACRDS.final.results[[i]]$degree[names(ACRDS.final.results[[i]]$degre
  }
  # length(degree)
  if (length(degree)!=0){
    degree.boxplot[1:length(degree),j]=as.vector(degree)
  }
}

# x11()
boxplot(degree.boxplot[1:100,1:253], range = 1.5,main="Boxplot of Degree Distribution fr
lines(rdd.Y~LINK,type='l',col='red',lwd=2,lty=1)
legend("topright", "True Degree Distribution of Friendship Network"
       , col = "red",lty = 1,lwd = 2 ,text.font=2,cex=.8)

dev.off()
```

```
######################### AC-RDS (DE=2) ############################
pdf("rquartilewithtrue(ACRDS2).pdf",width=12,height=8)
cc=rep(0,100)
S4 <- rep(0,100)
for (i in 1:100){
  cc[i] <- max(as.numeric(names(ACRDS.final.results.DE2[[i]]$degree)))
  S4[i] <- sum(ACRDS.final.results.DE2[[i]]$degree)
}
cc=max(cc)
# cc=253

degree.boxplot=matrix(,100,cc+1)

for (j in 1:cc+1){
  degree=NULL
  for (i in 1:100){
    degree=c(degree,ACRDS.final.results.DE2[[i]]$degree[names(ACRDS.final.results.DE2[[i
  }
  # length(degree)
  if (length(degree)!=0){
    degree.boxplot[1:length(degree),j]=as.vector(degree)
  }
}

# x11()
boxplot(degree.boxplot[1:100,1:253], ylim=c(0,.05),range = 1.5,main="Boxplot of Degree D
lines(rdd.Y~LINK,type='l',col='red',lwd=2,lty=1)
```

```
legend("topright", "True Degree Distribution of Friendship Network"
       , col = "red",lty = 1,lwd = 2 ,text.font=2,cex=.8)


dev.off()
```