Algorithms and Models for Tensors and Networks with Applications in Data Science

by

Manda Winlaw

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Applied Mathematics

Waterloo, Ontario, Canada, 2016

Author's Declaration

Statement of Contributions

The work in Chapter 2 which presents the PNCG algorithm along with several variants and their properties and demonstrates its effectiveness in computing the rank-R canonical tensor decomposition was published as a journal article [27]. The article was co-authored by my supervisor Hans De Sterck and I but was written by me and a shortened version of the paper also written by me was submitted to the Thirteenth Copper Mountain Conference on Iterative Methods for which I received an award for best student paper. The work in Chapter 3 which demonstrates the effectiveness of the PNCG algorithm in computing the latent factor model decomposition is part of a paper accepted to the 21st IEEE International Conference on Parallel and Distributed Systems [103] and won the best paper award. This paper was co-authored with Michael Hynes, Anthony Caterini and Hans De Sterck. My contribution was to examine the performance of the algorithm in serial and I wrote the parts of the paper not directly related to the parallel algorithm and its performance. It is the work that I wrote and contributed to that is replicated in this thesis.

All of the work in Chapters 4, 5, and 6 on graph generators was done in collaboration with my supervisor Hans De Sterck and Geoffery Sanders at the Lawrence Livermore National Laboratory in Livermore, California. However, all the work written in this thesis was written and performed by me and we are in the process of developing the work into several manuscripts for publication.

Abstract

Big data plays an increasingly central role in many areas of research including optimization and network modeling. We consider problems applicable to large datasets within these two branches of research. We begin by presenting a nonlinearly preconditioned nonlinear conjugate gradient (PNCG) algorithm to increase the convergence speed of iterative unconstrained optimization methods. We provide a concise overview of several PNCG variants and their properties and obtain a new convergence result for one of the PNCG variants under suitable conditions. We then use the PNCG algorithm to solve two different problems: computing the rank-R canonical tensor decomposition and finding the solution to a latent factor model where latent factor models are often used as important building blocks in many practical recommendation systems. For both problems, the alternating least squares (ALS) algorithm is typically used to find a solution and as such we consider it as a nonlinear preconditioner. Note that the ALS algorithm can be viewed as a nonlinear preconditioner for the NCG algorithm or alternatively, NCG can be viewed as an acceleration process for ALS. We demonstrate numerically that the convergence acceleration mechanism in PNCG often leads to important pay-offs for difficult tensor decomposition problems, with convergence that is significantly faster and more robust than for the stand-alone NCG or ALS algorithms. As well, we show numerically that the PNCG algorithm requires many fewer iterations and less time to reach desired ranking accuracies than stand-alone ALS in solving latent factor models.

We next turn to problems within the field of network or graph modeling. A network is a collection of points joined together by lines and networks are used in a broad variety of fields to represent connections between objects. Many large real-world networks share similar properties which has garnered considerable interest in developing models that can replicate these properties. We begin our discussion of graph models by closely examining the Chung-Lu model [4, 22, 23]. The Chung-Lu model is a very simple model where by design the expected degree sequence of a graph generated by the model is equal to a user-supplied degree sequence, $k = (k_1, \ldots, k_n)$, where k_i is the user-supplied degree of node i and n is the number of nodes in the graph. We explore what happens both theoretically and numerically when simple changes are made to the model and when the model assumptions are violated. As well, we consider an algorithm used to generate instances of the Chung-Lu model that is designed to be faster than the traditional algorithm but find that it only generates instances of an approximate Chung-Lu model. We explore the properties of this approximate model under a variety of conditions and examine how different the expected degree sequence is from the user-supplied degree sequence. We also explore several ways of improving this approximate model to reduce the approximation error in the expected degree sequence and note that when the assumptions of the original model are violated this error remains very large. We next design a new graph generator to match the community structure found in real-world networks as measured using the clustering coefficient and assortativity coefficient. Our graph generator uses information generated from a clustering algorithm run on the original network to build a synthetic network. Using several real-world networks, we test our algorithm numerically by creating a synthetic network and then comparing the properties to the real network properties as well as to the properties of another popular graph generator, BTER, developed by Seshadhri, Kolda and Pinar [94, 58]. Our graph generator does well at preserving the clustering coefficient and typically outperforms BTER in matching the assortativity coefficient, particularly when the assortativity coefficient is negative.

Acknowledgements

I would like to thank my supervisor Prof. Hans De Sterck for his continued support of my Ph.D study and research, for giving me the freedom to explore research areas that interested me and introducing me and allowing me to work with researchers around the world. His guidance has helped me immensely in both conducting my research and in the writing of this thesis. I greatly appreciated his patience, motivation, and immense knowledge.

Besides my supervisor, I would like to thank the researchers at Lawrence Livermore National Laboratory and in particular Geoffery Sanders and Van Henson. Working with them throughout my Ph.D study has not only provided me with financial assitance but has greatly improved my Ph.D experience both academically and personally. I also recognize that this research would not have been possible without the financial assistance of NSERC.

I would like to thank my family: my parents, Murray and Dorothy for providing continued support throughout my life and for motivating me to always continue learning, my sisters, Marlo and Kara for always being there for me and last but certainly not least my husband, Ian, who has been my rock and provided the support and love I needed to complete this thesis.

Dedication

I would like to dedicate this thesis to my family and in particular my husband, Ian. Your love and support has meant everything to me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Overview

Data science can be defined broadly as the extraction of knowledge from data. With an increased focus on massive datasets there has been renewed interest in the data sciences field. Data science itself encompasses a broad range of fields including optimization, statistics, machine learning, artificial intelligence and databases. In this thesis, we focus on two different problems within the data sciences field. Optimization algorithms play a large role in data science so in the first part of this thesis we look at a particular optimization algorithm, the nonlinear conjugate gradient algorithm (NCG), and examine how to increase its convergence speed using a nonlinear preconditioner. We begin in Chapter 2 by presenting the nonlinearly preconditioned nonlinear conjugate gradient algorithm (PNCG) as a means of computing rank-R tensor decompositions. The alternating least squares (ALS) algorithm is often considered the workhorse algorithm for computing the rank-R canonical tensor approximation, but for certain problems its convergence can be very slow. The nonlinear conjugate gradient method has been proposed as an alternative to ALS [1], but the results indicated that NCG is usually not faster than ALS. To improve the convergence speed of NCG, we consider a nonlinearly preconditioned nonlinear conjugate gradient algorithm for computing the rank-R canonical tensor decomposition. Our approach uses ALS as a nonlinear preconditioner in the NCG algorithm. Alternatively, NCG can be viewed as an acceleration process for ALS. We demonstrate numerically that the convergence acceleration mechanism in PNCG often leads to important pay-offs for difficult tensor decomposition problems, with convergence that is significantly faster and more robust than for the stand-alone NCG or ALS algorithms. We consider several approaches for incorporating the nonlinear preconditioner into the NCG algorithm that have been described in the literature previously and have met with success in certain application areas. However, it appears that the nonlinearly preconditioned NCG approach has received relatively little attention in the broader community and remains underexplored both theoretically and experimentally. Thus, this thesis provides in one place a concise overview of several PNCG variants and their properties that have only been described in a few places scattered throughout the literature, and it systematically compares the performance of these PNCG variants for the tensor decomposition problem, drawing further attention to the usefulness of nonlinearly preconditioned NCG as a general tool. In addition, we briefly discuss the convergence of the PNCG algorithm. In particular, we obtain a new convergence result for one of the PNCG variants under suitable conditions, building on known convergence results for non-preconditioned NCG.

Having shown that the PNCG algorithm can increase convergence speeds for the rank-R canonical tensor decomposition we then use the algorithm in Chapter 3 to show that it can accelerate convergence in latent factor models used to build recommendation systems which are popular at many online business including Amazon [64], Netflix [10], and Spotify [50]. In these latent factor models we are trying to decompose a matrix where many of the values are missing and once again ALS is often used to compute this decomposition. Using ALS as a nonlinear preconditioner we apply the PNCG algorithm to a popular latent factor model. Using a movie ratings matrix from MovieLens [75] we demonstrate that PNCG with ALS as a nonlinear preconditioner can significantly improve the convergence speed of ALS using two different measures of convergence. One is the traditional measure from optimization, the gradient norm of the objective function, where a method is said to converge if the gradient norm is zero to a specified tolerance. The second is a measure based on the rankings of the predicted ratings.

The second part of this thesis focuses on another problem within the data sciences field: network or graph modeling. A network can be defined very simply as a collection of points joined together by lines and networks can represent connections between entities in a wide variety of fields including engineering, science, medicine, and sociology. Despite originating from a variety of different fields, many large real-world networks have similar properties and there has been significant interest in developing models that can replicate these properties. In addition, there is substantial interest in developing algorithms known as graph generators that can create synthetic graphs that replicate the properties of real-world networks. Often, these algorithms are designed to generate synthetic graphs where an underlying abstract graph generation model is the starting point but this is not always the case. By building models that mimic the patterns and properties of real networks this helps us to understand the implications of these patterns and determine which patterns are important. By developing algorithms to synthesize real networks we can also examine which growth processes are plausible for the growth of real-world networks and which are not. Model development research serves an additional purpose. High-quality, large-scale network data is often not available, because of economic, legal, technological, or other obstacles [19] and thus there are many instances where the systems of interest cannot be represented by a given real-world network. When there is no single exemplar network, the systems must instead be modeled as a collection of networks in which the variation among them may be just as important as their common features. If we can create synthetic networks that capture both the essential features of a system and realistic variability by modeling these networks either using a formal abstract model or a practical graph generator then we can use such synthetic graphs to perform tasks such as simulations, analysis, and decision making as well as testing clustering algorithms and anomaly detection algorithms.

We begin our discussion of graph models in Chapter 4, where we examine a very simple random graph model, the Chung-Lu model [4, 22, 23]. In the Chung-Lu model, the probability of each edge is a function of a user-supplied degree sequence, $k = (k_1, \dots, k_n)$, where k_i is the user-supplied degree of node i and n is the number of nodes in the graph. By design the user-supplied degree sequence is equal to the expected degree sequence of the graph model (i.e $E(\mathbf{D}_i) = k_i$ where $\mathbf{D}_i$ is the degree of node i in the model). The Chung-Lu model has a very simple algorithm which can be used to generate synthetic graphs which represent instances of the model. However, this algorithm is computationally expensive, which has led to the development of faster algorithms. We explore one such algorithm, the Fast Chung-Lu (FCL) algorithm [94, 58] and observe that it does not in fact generate instances of the Chung-Lu model but an approximation to the Chung-Lu model. We explicitly formulate this approximate model and look at its properties, in particular, we look at the approximation error in the expected degree distribution and find that the expected degree of every node in the approximate model is below the expected degree in the original Chung-Lu model (which is equal to k_i). In addition to exploring the approximation error in the FCL model we also explore the implications of violating one of the central assumptions in the Chung-Lu model. In the Chung-Lu model, the user-supplied degree sequence k must satisfy the constraint $k_i^2 \leq 2m \; \forall \; i$, where $m = \frac{1}{2}\sum_i k_i$ is the number of edges in the graph. We examine what happens in the Chung-Lu model and the FCL model when this assumption no longer holds. For the Chung-Lu model, the result is another approximate (or extended) Chung-Lu model where the approximation error in the expected degree distribution can be quite large and for the FCL model the model remains the same however the approximation error for the expected degree distribution worsens.

The approximation error in the expected degree distribution in the FCL model can be quite large and in Chapter 5 we present several different methods for modifying the model and reducing the approximation error. We will see in Chapter 4 that the FCL algorithm requires drawing m edges to place in the graph. However, because there can be duplicate edges or self-edges which we discard the resulting graph instance may not have m edges. One simple improvement we consider is increasing the number of edges drawn. We look at several different alternatives for increasing the number of draws and find that by changing the number of draws the difference between the expected degree in the FCL model and the user-supplied degree (i.e. the expected degree in the original Chung-Lu model) is smaller for some nodes, however, for some nodes we increase the expected degree above that of the user-supplied degree and increase the approximation error. The next improvement we consider is an algorithm that generates instances by first using the FCL algorithm and then looking at the resulting instance to determine any additional edges needed. This algorithm does improve the approximation error but the approximation error is still not insignificant. Our final attempt at improving the approximation error relies on determining optimal probabilities for matching the expected degree sequence to the actual degree sequence in the FCL model. This results in a constrained optimization problem that is difficult to solve and we turn to fixed point methods for an approximate solution. This method which finds "improved " probabilities performs the best at reducing the approximation error in the degree distribution. We also examine the improvement proposed in [85] and compare it to the improvements we suggest and find that for the problems we consider our improved probabilities method still performs best. In the final part of this chapter, in Section 5.6, we explore all the previously introduced model improvements when the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated. While all the improvements we suggest do reduce the approximation error we note that the approximation error can remain very large. This leads us to conclude that when the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated the FCL model (including any of the improvements) should not be used to generate instances to compare with other graph generator instances, when evaluating the effectiveness of these other graph generators. In the Chung-Lu model we expect the expected degree sequence to equal the input degree sequence and in the FCL model we either expect (incorrectly) for the expected degree sequence to equal the input degree sequence or for the approximation error to be small. Thus, we expect other properties of the FCL model to be similar to properties of the Chung-Lu model. Suppose we compare an instance generated by the FCL algorithm to an instance generated by another graph generator and then draw conclusions relating the original Chung-Lu model to the model underlying this other generator. If the approximation error is quite large as in the case where the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated then these conclusions are likely incorrect.

In Chapter 6 we develop our own graph generator. As mentioned previously, graph generators are designed to match the properties of real-world networks. One network property of particular interest is the community structure which can be defined in many different ways. As we will see in Chapter 6 two common ways to measure the community structure of a network is through the clustering coefficient and the assortativity coefficient. We build a graph generator that tries to capture these properties by using a clustering algorithm. Using the Louvain clustering algorithm [13] to generate clusters of our input network we then use these clusters and the subgraphs they induce to generate synthetic graphs. Using a synthetic network generated from our algorithm we compare the properties to the real network properties as well as to the properties of another popular graph generator, BTER, developed by Seshadhri, Kolda and Pinar [94, 58]. We find that our graph generator does well at preserving the clustering coefficient and typically outperforms BTER in matching the assortativity coefficient, particularly when the assortativity coefficient is negative. We also compare our synthetic graph to one generated using a simple edge switching algorithm [79], an algorithm that is a building block of our graph generator. In our graph generator the clustering coefficient remains close to the original clustering coefficient, however, the edge switching algorithm greatly reduces the clustering coefficient. As well, the edge switching algorithm can change the assortativity coefficient by a large amount, but this behaviour is once again not seen in instances generated by our graph generator.

Finally, in Chapter 7 we conclude by looking at areas of future research in both topic areas.

Chapter 2

A Nonlinearly Preconditioned Conjugate Gradient Algorithm for Rank-R Canonical Tensor Approximation

2.1 Introduction

In this chapter, we consider a nonlinearly preconditioned nonlinear conjugate gradient (PNCG) algorithm for computing a canonical rank-R tensor approximation using the Frobenius norm as a distance metric. The current workhorse algorithm for computing the canonical tensor decomposition is the alternating least squares (ALS) algorithm [18, 45, 56]. The ALS method is simple to understand and implement, but for certain problems its convergence can be very slow [98, 56]. In [1], the nonlinear conjugate gradient (NCG) method is considered as an alternative to ALS for solving canonical tensor decomposition problems. However, [1] found that NCG is usually not faster than ALS. In this chapter, we show how incorporating ALS as a nonlinear preconditioner into the NCG algorithm (or, equivalently, accelerating ALS by the NCG algorithm) may lead to significant convergence acceleration for difficult canonical tensor decomposition problems.

Our approach is among extensive, recent, research activity on nonlinear preconditioning for nonlinear iterative solvers [36, 106, 101, 25, 17], including nonlinear GMRES and NCG. This work builds on original contributions dating back as far as the 1960s [7, 24, 89, 83], but much of this early work is not well-known in the broader community and large parts of the landscape remain unexplored experimentally and theoretically [17]; the recent paper [17] gives a comprehensive overview of the state of the art in nonlinear preconditioning and provides interesting new directions.

In this chapter, we consider nonlinear preconditioning of NCG for the canonical tensor decomposition problem. We consider several approaches for incorporating the nonlinear preconditioner into the NCG algorithm that are described in the literature (see [9, 24, 70, 106, 17]). Early references to nonlinearly preconditioned NCG include [9] and [24]. Both propose the NCG algorithm as a solution method for solving nonlinear elliptic partial differential equations (PDEs) and while both present NCG algorithms that include a possible nonlinear preconditioner, [24] actually uses a block nonlinear SSOR method as the nonlinear preconditioner in their numerical experiments. Hager and Zhang's survey paper [44] describes a linearly preconditioned NCG algorithm, but does not discuss general nonlinear preconditioning for NCG. More recent work on nonlinearly preconditioned NCG includes [106], which uses parallel coordinate descent as a nonlinear preconditioner for one variant of NCG applied to $L_1 - L_2$ optimization in signal and image processing. The recent overview paper [17] on nonlinear preconditioning also briefly mentions nonlinearly preconditioned NCG, but discusses a different variant than [9], [24], [70] and [106]. In Section 2.3, the differences between the PNCG variants of [9, 24, 70, 106, 17] will be explained. In Section 2.3 we will also prove a new convergence result for one of the PNCG variants, building on known convergence results for non-preconditioned NCG. In Section 4, extensive numerical tests on a set of standard test tensors systematically compare the performance of the PNCG variants using ALS as the nonlinear preconditioner, and demonstrate the effectiveness of the overall approach.

As mentioned above, we apply the PNCG algorithm to the tensor decomposition problem which can be described as follows. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ be an N-way or Nth-order tensor of size $I_1 \times I_2 \times \ldots \times I_N$. Let $\mathcal{A}_R \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ be a canonical rank-R tensor given by

$$\mathcal{A}_R = \sum_{r=1}^{R} \mathbf{a}_r^{(1)} \circ \ldots \circ \mathbf{a}_r^{(N)} = [\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]. \tag{2.1}$$

The canonical tensor $\mathcal{A}_R$ is the sum of R rank-one tensors, with the rth rank-one tensor composed of the outer product of N column vectors $\mathbf{a}_r^{(n)} \in \mathbb{R}^{I_n}$, $n = 1, \ldots, N$. We are interested in finding $\mathcal{A}_R$ as an approximation to $\mathcal{X}$ by minimizing the following function:

$$f(\mathcal{A}_R) = \frac{1}{2} \|\mathcal{X} - \mathcal{A}_R\|_F^2, \tag{2.2}$$

where $\|\cdot\|_F$ denotes the Frobenius norm of the N-dimensional array.

The decomposition of $\mathcal{X}$ into $\mathcal{A}_R$ is known as the canonical tensor decomposition. Popularized by Carroll and Chang [18] as CANDECOMP and by Harshman [45] as PARAFAC in the 1970s, the decomposition is commonly referred to as the CP decomposition where the 'C' refers to CANDECOMP and the 'P' refers to PARAFAC. The canonical tensor decomposition is commonly used as a data analysis technique in a wide variety of fields including chemometrics, signal processing, neuroscience and web analysis [56, 3].

The ALS algorithm for CP decomposition was first proposed in papers by Carroll and Chang [18] and Harshman [45]. For simplicity we present the algorithm for a three-way tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$. In this case, the objective function (2.2) simplifies to

$$f(\widehat{\mathcal{X}}) = \frac{1}{2}\|\mathcal{X} - \widehat{\mathcal{X}}\|_F^2 \text{ with } \widehat{\mathcal{X}} = \sum_{r=1}^{k} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r = [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]. \tag{2.3}$$

The ALS approach fixes **B** and **C** to solve for **A**, then fixes **A** and **C** to solve for **B**, then fixes **A** and **B** to solve for **C**. This process continues until some convergence criterion is satisfied. Once all but one matrix is fixed, the problem reduces to a linear least-squares problem. Since we are solving a nonlinear equation for a block of variables while holding all the other variables fixed the ALS algorithm is in fact a block nonlinear Gauss-Seidel algorithm. The algorithm can easily be extended to N-way tensors by fixing all but one of the matrices. The ALS method is simple to understand and implement, but can take many iterations to converge. It is not guaranteed to converge to a global minimum or even a stationary point of (2.2). We can only guarantee that the objective function in (2.2) is nonincreasing at every step of the ALS algorithm. As well, if the ALS algorithm does converge to a stationary point, the stationary point can be heavily dependent on the starting guess. See [56, 100] for a discussion on the convergence of the ALS algorithm.

A number of algorithms have been proposed as alternatives to the ALS algorithm. See [1, 56, 98, 42] and the references therein for examples. Acar, Dunlavy and Kolda [1] recently applied a standard NCG algorithm to solve the problem. They find that NCG is usually not faster than ALS, even though it has its advantages in terms of overfactoring (i.e. computing the CP decomposition with R greater than the actual rank of the tensor) and its ability to solve coupled factorizations [1, 2]. In an earlier paper, Paatero [84] uses the linear conjugate gradient algorithm to solve the normal equations associated with the CP decomposition and suggests the possible use of a linear preconditioner to increase the convergence speed, however, no extensive numerical testing of the algorithm is performed. Inspired by the nonlinearly preconditioned nonlinear GMRES method of [25], we propose in this thesis to accelerate the NCG approach of [1] by considering the use of ALS as a nonlinear preconditioner for NCG.

In terms of notation, throughout this chapter we use CG to refer to the linear conjugate gradient algorithm applied to a symmetric positive definite (SPD) linear system without preconditioning, and PCG refers to CG for SPD linear systems with (linear) preconditioning. Similarly, NCG refers to the nonlinear conjugate gradient algorithm for optimization problems without preconditioning, and PNCG refers to the class of (nonlinearly) preconditioned nonlinear conjugate gradient methods for optimization.

The remainder of the chapter is structured as follows. In Section 2.2, we introduce the standard nonlinear conjugate gradient algorithm for unconstrained continuous optimization. Section 2.3 gives a concise description of several variants of the PNCG algorithm that we collect from the literature and describe systematically, and it discusses their relation to the PCG algorithm in the linear case, followed by a brief convergence discussion highlighting our new convergence result. In Section 2.4 we follow the experimental procedure of Tomasi and Bro [98] to generate test tensors that we use to systematically compare the several PNCG variants we have described with the standard ALS and NCG algorithms. Section 2.5 concludes.

2.2 Nonlinear Conjugate Gradient Algorithm

The NCG algorithm for continuous optimization is an extension of the CG algorithm for linear systems. The CG algorithm minimizes the convex quadratic function

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}, \tag{2.4}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is an SPD matrix. Equivalently, the CG algorithm can be viewed as an iterative method for solving the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$. The NCG algorithm is adapted from the CG algorithm and can be applied to any unconstrained optimization problem of the form

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) \tag{2.5}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a continuously differentiable function bounded from below. The general form of the NCG algorithm is summarized in Algorithm 1.

The NCG algorithm is a line search algorithm that generates a sequence of iterates $\mathbf{x}_i$, $i \geq 1$ from the initial guess $\mathbf{x}_0$ using the recurrence relation

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k. \tag{2.6}$$

Algorithm 1: Nonlinear Conjugate Gradient Algorithm (NCG)

```
Input: $\mathbf{x}_0$
Evaluate $\mathbf{g}_0 = \nabla f(\mathbf{x}_0)$;
Set $\mathbf{p}_0 \leftarrow -\mathbf{g}_0, k \leftarrow 0$;
while $\mathbf{g}_k \neq 0$ do
    Compute $\alpha_k$;
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$;
    Evaluate $\mathbf{g}_{k+1} = \mathbf{g}(x_{k+1}) = \nabla f(x_{k+1})$;
    Compute $\beta_{k+1}$;
    $\mathbf{p}_{k+1} \leftarrow -\mathbf{g}_{k+1} + \beta_{k+1}\mathbf{p}_k$;
    $k \leftarrow k+1$;
end
```

The parameter $\alpha_k > 0$ is the step length and $\mathbf{p}_k$ is the search direction generated by the following rule:

$$\mathbf{p}_{k+1} = -\mathbf{g}_{k+1} + \beta_{k+1}\mathbf{p}_k, \quad \mathbf{p}_0 = -\mathbf{g}_0, \tag{2.7}$$

where β_{k+1} is the update parameter and $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$ is the gradient of f evaluated at $\mathbf{x}_k$. In the CG algorithm, α_k is defined as

$$\alpha_k^{CG} = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}, \tag{2.8}$$

and β_{k+1} is defined as

$$\beta_{k+1}^{CG} = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_\mathbf{k}}, \tag{2.9}$$

where $\mathbf{r}_k = \nabla\phi(\mathbf{x}_k) = \mathbf{A}\mathbf{x}_k - \mathbf{b}$ is the residual. In the nonlinear case α_k is determined by a line search algorithm and β_{k+1} can assume various different forms. We consider three different forms in this chapter, given by

$$\beta_{k+1}^{FR} = \frac{\mathbf{g}_{k+1}^T \mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{g}_k}, \tag{2.10}$$

$$\beta_{k+1}^{PR} = \frac{\mathbf{g}_{k+1}^T (\mathbf{g}_{k+1} - \mathbf{g}_k)}{\mathbf{g}_k^T \mathbf{g}_k}, \tag{2.11}$$

$$\beta_{k+1}^{HS} = \frac{\mathbf{g}_{k+1}^T (\mathbf{g}_{k+1} - \mathbf{g}_k)}{(\mathbf{g}_{k+1} - \mathbf{g}_k)^T \mathbf{p}_k}. \tag{2.12}$$

Fletcher and Reeves [38] first showed how to extend the conjugate gradient algorithm to the nonlinear case. By replacing the residual, $\mathbf{r}_k$, with the gradient of the nonlinear objective f, they obtained a formula for β_{k+1} of the form β_{k+1}^{FR}. The variant β_{k+1}^{PR} was developed by Polak and Ribière [88] and the Hestenes-Stiefel [48] formula is given by Equation (2.12). For all three versions, it can easily be shown that, if a convex quadratic function is optimized using the NCG algorithm and the line search is exact then $\beta_{k+1}^{FR} = \beta_{k+1}^{PR} = \beta_{k+1}^{HS} = \beta_{k+1}^{CG}$ where β_{k+1}^{CG} is given by Equation (2.9), see [81, Section 5.2].

2.3 Preconditioned Nonlinear Conjugate Gradient Algorithm

In this section we give a concise description of several variants of PNCG that have been proposed in a few places in the literature but have not been discussed and compared systematically in one place, briefly discuss some of their relevant properties, and prove a new convergence property for one of the variants. Before we introduce PNCG we describe the PCG algorithm for linear systems. We do this because it will be useful for interpreting some of the variants for β_{k+1} in the PNCG algorithm. In particular, one variant of the β_{k+1} formulas has the property that PNCG applied to the convex quadratic function (2.4) is equivalent to PCG under certain conditions on the line search and the preconditioner.

2.3.1 Linearly Preconditioned Linear Conjugate Gradient Algorithm

Preconditioning the conjugate gradient algorithm is commonly used in numerical linear algebra to speed up convergence [92, p. 261]. The rate of convergence of the linear conjugate gradient algorithm can be bounded by examining the eigenvalues of the matrix $\mathbf{A}$ in (2.4). For example, if the eigenvalues occur in r distinct clusters the CG iterates will approximately solve the problem in r steps [81, p. 117, 99, p. 299]. Thus, one way to improve the convergence of the CG algorithm is to transform the linear system $\mathbf{Ax} = \mathbf{b}$ to improve the eigenvalue distribution of $\mathbf{A}$. Consider a change of variables from $\mathbf{x}$ to $\widehat{\mathbf{x}}$ via a symmetric positive definite matrix $\mathbf{C}$ such that $\widehat{\mathbf{x}} = \mathbf{Cx}$. This process is known as preconditioning. The new objective function is

$$\widehat{\phi}(\widehat{\mathbf{x}}) = \frac{1}{2}\widehat{\mathbf{x}}^T(\mathbf{C}^{-T}\mathbf{A}\mathbf{C}^{-1})\widehat{\mathbf{x}} - (\mathbf{C}^{-T}\mathbf{b})^T\widehat{\mathbf{x}}, \tag{2.13}$$

and the new linear system is

$$(\mathbf{C}^{-T}\mathbf{A}\mathbf{C}^{-1})\widehat{\mathbf{x}} = \mathbf{C}^{-T}\mathbf{b}. \tag{2.14}$$

Thus, the convergence rate will depend on the eigenvalues of the matrix $\mathbf{C}^{-T}\mathbf{A}\mathbf{C}^{-1}$. If we choose $\mathbf{C}$ such that the condition number of $\mathbf{C}^{-T}\mathbf{A}\mathbf{C}^{-1}$ is smaller than the condition number of $\mathbf{A}$ or such that eigenvalues of $\mathbf{C}^{-T}\mathbf{A}\mathbf{C}^{-1}$ are clustered, then hopefully the preconditioned CG algorithm will converge faster than the regular CG algorithm. The preconditioned conjugate gradient algorithm is given in Algorithm 2, expressed in terms of the original variable $\mathbf{x}$ using the SPD preconditioning matrix $\mathbf{P} = \mathbf{C}^{-1}\mathbf{C}^{-T}$. Note that the preconditioned linear system (2.14) can equivalently be expressed as $\mathbf{PAx} = \mathbf{Pb}$, where $\mathbf{PA}$ has the same eigenvalues as $\mathbf{C}^{-T}\mathbf{A}\mathbf{C}^{-1}$. In Algorithm 2, we do not actually form the matrix $\mathbf{P}$ explicitly. Instead, we solve the linear system $\mathbf{M}\mathbf{y}_k = \mathbf{r}_k$ for $\mathbf{y}_k$ with $\mathbf{M} = \mathbf{P}^{-1} = \mathbf{C}^T\mathbf{C}$ and use $\mathbf{y}_k$ in place of $\mathbf{P}\mathbf{r}_k$. See [81, Algorithm 5.3] for the PCG algorithm written in this way. Algorithm 2 is written in terms of $\mathbf{P}$ to compare the PCG algorithm with the preconditioned NCG algorithm in what follows.

Algorithm 2: Linearly Preconditioned Linear Conjugate Gradient Algorithm (PCG)

```
Input: $\mathbf{x}_0$, Preconditioner $\mathbf{P} = \mathbf{C}^{-1}\mathbf{C}^{-T}$
Evaluate $\mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b}$;
Evaluate $\mathbf{p}_0 = -\mathbf{P}\mathbf{r}_0, k \leftarrow 0$;
while $\mathbf{r}_k \neq 0$ do
    $\alpha_k \leftarrow \frac{\mathbf{r}_k^T\mathbf{P}\mathbf{r}_k}{\mathbf{p}_k\mathbf{A}\mathbf{p}_k}$;
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k\mathbf{p}_k$;
    $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k\mathbf{A}\mathbf{p}_k$;
    $\beta_{k+1} \leftarrow \frac{\mathbf{r}_{k+1}^T\mathbf{P}\mathbf{r}_{k+1}}{\mathbf{r}_k\mathbf{P}\mathbf{r}_k}$;
    $\mathbf{p}_{k+1} \leftarrow -\mathbf{P}\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{p}_k$;
    $k \leftarrow k + 1$;
end
```

2.3.2 Linearly Preconditioned Nonlinear Conjugate Gradient Algorithm

We can also apply a linear change of variables, $\widehat{\mathbf{x}} = \mathbf{C}\mathbf{x}$, to the NCG algorithm as is explained in review paper [44]. The linearly preconditioned NCG algorithm expressed in terms of the original variable $\mathbf{x}$ can be described by the following equations:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{2.15}$$

$$\mathbf{p}_{k+1} = -\mathbf{P}\mathbf{g}_{k+1} + \breve{\beta}_{k+1}\mathbf{p}_k, \quad \mathbf{p}_0 = -\mathbf{P}\mathbf{g}_0, \tag{2.16}$$

where $\mathbf{P} = \mathbf{C}^{-1}\mathbf{C}^{-T}$. The formulas for $\breve{\beta}_{k+1}$ remain the same as before (Equations (2.10)–(2.12)), except that $\mathbf{g}_k$ and $\mathbf{p}_k$ are replaced by $\mathbf{C}^{-T}\mathbf{g}_k$ and $\mathbf{C}\mathbf{p}_k$, respectively. Thus we obtain linearly preconditioned versions of the β_{k+1} parameters of Equations (2.10)–(2.12):

$$\breve{\beta}^{FR}_{k+1} = \frac{\mathbf{g}_{k+1}^T \mathbf{P}\mathbf{g}_{k+1}}{\mathbf{g}_k^T \mathbf{P}\mathbf{g}_k}, \tag{2.17}$$

$$\breve{\beta}^{PR}_{k+1} = \frac{\mathbf{g}_{k+1}^T \mathbf{P}(\mathbf{g}_{k+1} - \mathbf{g}_k)}{\mathbf{g}_k^T \mathbf{P}\mathbf{g}_k}, \tag{2.18}$$

$$\breve{\beta}^{HS}_{k+1} = \frac{\mathbf{g}_{k+1}^T \mathbf{P}(\mathbf{g}_{k+1} - \mathbf{g}_k)}{(\mathbf{g}_{k+1} - \mathbf{g}_k)^T \mathbf{p}_k}. \tag{2.19}$$

If we use the linearly preconditioned NCG algorithm with these $\breve{\beta}_{k+1}$ formulas to minimize the convex quadratic function, $\phi(\mathbf{x})$, defined in Equation (2.4), using an exact line search, where $\mathbf{g}_k = \mathbf{r}_k$, then the algorithm is the same as the PCG algorithm described in Algorithm 2. This can easily be shown in the same way as Equations (2.10)–(2.12) are shown to be equivalent to Equation (2.9) in the linear case without preconditioning [44, 81]. Hager and Zhang's survey paper [44] describes this linearly preconditioned NCG algorithm, and also notes that $\mathbf{P}$ can be chosen differently in every step (see [77]). While a varying $\mathbf{P}$ does introduce a certain type of nonlinearity into the preconditioning process, the preconditioning in every step remains a linear transformation, and is thus different from the more general nonlinear preconditioning to be described in the next section, which employs a general nonlinear transformation in every step.

2.3.3 Nonlinearly Preconditioned Nonlinear Conjugate Gradient Algorithm

Suppose instead, we wish to introduce a nonlinear transformation of $\mathbf{x}$. In particular, suppose we consider a nonlinear iterative optimization method such as Gauss-Seidel. Let $\overline{\mathbf{x}}_k$ be the preliminary iterate generated by one step of a nonlinear iterative method, i.e., we write

$$\overline{\mathbf{x}}_k = P(\mathbf{x}_k), \tag{2.20}$$

which we will use as a nonlinear preconditioner. We assume that P is a fixed point method of $\nabla f(\mathbf{x}) = 0$. Thus, any stationary point of the optimization problem in Equation (2.5) is a fixed point of P and vice versa (i.e. $\mathbf{x}^* = P(\mathbf{x}^*) \Leftrightarrow \mathbf{x}^*$ is a stationary point of $f(\mathbf{x})$). Now define the direction generated by the nonlinear preconditioner as

$$\overline{\mathbf{g}}_k = \mathbf{x}_k - \overline{\mathbf{x}}_k = \mathbf{x}_k - P(\mathbf{x}_k). \tag{2.21}$$

In nonlinearly preconditioned NCG, one considers the nonlinearly preconditioned direction, $\overline{\mathbf{g}}_k$, instead of the gradient, $\mathbf{g}_k$, in formulating the NCG method [9, 24, 70, 106, 17]. This idea can be motivated by the linear preconditioning of CG, where $\mathbf{g}_k = \mathbf{r}_k$ is replaced by the preconditioned gradient $\mathbf{P}\mathbf{g}_k = \mathbf{P}\mathbf{r}_k$ in certain parts of Algorithm 2. This corresponds to replacing the Krylov space for CG, which is formed by the gradients $\mathbf{g}_k = \mathbf{r}_k$, with the left-preconditioned Krylov space for PCG, which is formed by the preconditioned gradients $\mathbf{P}\mathbf{g}_k = \mathbf{P}\mathbf{r}_k$. In a similar way, we replace the nonlinear gradients $\mathbf{g}_k$ with the nonlinearly preconditioned directions $\overline{\mathbf{g}}_k$. Note that this approach is called nonlinear left-preconditioning in [17], which also considers nonlinear right-preconditioning.

Thus, our nonlinearly preconditioned NCG algorithm is given by the following equations:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{2.22}$$

$$\mathbf{p}_{k+1} = -\overline{\mathbf{g}}_{k+1} + \overline{\beta}_{k+1}\mathbf{p}_k, \quad \mathbf{p}_0 = -\overline{\mathbf{g}}_0, \tag{2.23}$$

instead of Equations (2.6) and (2.7) or Equations (2.15) and (2.16). The formulas for $\overline{\beta}_{k+1}$ in Equation (2.23) are modified versions of the β_{k+1} from Equations (2.10)–(2.12) that incorporate $\overline{\mathbf{g}}_k$. However, there are several different ways to modify the β_{k+1} to incorporate $\overline{\mathbf{g}}_k$, leading to several different variants of $\overline{\beta}_{k+1}$. Algorithm 3 summarizes the PNCG algorithm, and Table 2.1 summarizes the variants of $\overline{\beta}_{k+1}$ we consider in this chapter for PNCG. Note that the nonlinear preconditioner is not guaranteed to produce a descent direction. Thus, if $-\overline{\mathbf{g}}_{k+1}$ is not a descent direction (i.e. $-\overline{\mathbf{g}}_{k+1}^T\mathbf{g}_{k+1} > 0$) we modify it so it is a decent direction. As we will see in Theorem 2, if $-\overline{\mathbf{g}}_{k+1}$ is a descent direction and we use a line search algorithm to find α_{k+1} that satisfies the strong Wolfe conditions then $\mathbf{p}_{k+1}$ is also a descent direction (i.e $\mathbf{p}_{k+1}^T\mathbf{g}_{k+1} < 0$).

The first set of $\overline{\beta}_{k+1}$ variants we consider are the $\widetilde{\beta}_{k+1}$ shown in column 1 of Table 2.1. The $\widetilde{\beta}_{k+1}$ formulas are derived by replacing all occurrences of $\mathbf{g}_k$ with $\overline{\mathbf{g}}_k$ in the formulas for β_{k+1}, Equations (2.10)–(2.12):

$$\widetilde{\beta}_{k+1}^{FR} = \frac{\overline{\mathbf{g}}_{k+1}^T\overline{\mathbf{g}}_{k+1}}{\overline{\mathbf{g}}_k^T\overline{\mathbf{g}}_k}, \tag{2.24}$$

Algorithm 3: Nonlinearly Preconditioned Nonlinear Conjugate Gradient Algorithm (PNCG)

```
Input: $\mathbf{x}_0$
Evaluate $\overline{\mathbf{x}}_0 = P(\mathbf{x}_0)$;
Set $\overline{\mathbf{g}}_0 = \mathbf{x}_0 - \overline{\mathbf{x}}_0$;
Set $\mathbf{p}_0 \leftarrow -\overline{\mathbf{g}}_0, k \leftarrow 0$;
while $\mathbf{g}_k \neq 0$ do
    Compute $\alpha_k$;
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$;
    $\overline{\mathbf{g}}_{k+1} \leftarrow \mathbf{x}_{k+1} - P(\mathbf{x}_{k+1})$;
    if $-\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1} > 0$ then
        $\overline{\mathbf{g}}_{k+1} \leftarrow -\overline{\mathbf{g}}_{k+1}$;
    end
    Compute $\overline{\beta}_{k+1}$;
    $\mathbf{p}_{k+1} \leftarrow -\overline{\mathbf{g}}_{k+1} + \overline{\beta}_{k+1}\mathbf{p}_k$;
    $k \leftarrow k+1$;
end
```

$$\widetilde{\beta}_{k+1}^{PR} = \frac{\overline{\mathbf{g}}_{k+1}^T(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{\overline{\mathbf{g}}_k^T \overline{\mathbf{g}}_k}, \tag{2.25}$$

$$\widetilde{\beta}_{k+1}^{HS} = \frac{\overline{\mathbf{g}}_{k+1}^T(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)^T \mathbf{p}_k}. \tag{2.26}$$

This is a straightforward generalization of the β_{k+1} expressions in Equations (2.10)–(2.12), and the systematic numerical comparisons to be presented in Section 2.4 indicate that these choices lead to efficient PNCG methods. The PR variant of this formula is used in [17] in the context of PDE solvers.

However, the reader may note that Equations (2.17)–(2.19) suggest different choices for the $\overline{\beta}_{k+1}$ formulas, variants which reduce to the PCG update formulas in the linear case. Indeed, suppose we apply Algorithm 3 to the convex quadratic problem, (2.4), with an exact line search, using a symmetric stationary linear iterative method such as symmetric Gauss-Seidel or Jacobi as a preconditioner. We begin by writing the stationary iterative method in general form as

$$\overline{\mathbf{x}}_k = P(\mathbf{x}_k) = \mathbf{x}_k - \mathbf{P}\mathbf{r}_k, \tag{2.27}$$

where the SPD preconditioning matrix $\mathbf{P}$ is often written as $\mathbf{M}^{-1}$ and $\mathbf{r}_k = \mathbf{g}_k$. The search

Table 2.1: Variants of $\overline{\beta}_{k+1}$ for the Nonlinearly Preconditioned Nonlinear Conjugate Gradient Algorithm (PNCG).

	$\widetilde{\beta}_{k+1}$	$\widehat{\beta}_{k+1}$
Fletcher-Reeves	$\widetilde{\beta}^{FR}_{k+1} = \dfrac{\overline{\mathbf{g}}^T_{k+1}\overline{\mathbf{g}}_{k+1}}{\overline{\mathbf{g}}^T_k\overline{\mathbf{g}}_k}$	$\widehat{\beta}^{FR}_{k+1} = \dfrac{\mathbf{g}^T_{k+1}\overline{\mathbf{g}}_{k+1}}{\mathbf{g}^T_k\overline{\mathbf{g}}_k}$
Polak-Ribière	$\widetilde{\beta}^{PR}_{k+1} = \dfrac{\overline{\mathbf{g}}^T_{k+1}(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{\overline{\mathbf{g}}^T_k\overline{\mathbf{g}}_k}$	$\widehat{\beta}^{PR}_{k+1} = \dfrac{\mathbf{g}^T_{k+1}(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{\mathbf{g}^T_k\overline{\mathbf{g}}_k}$
Hestenes-Stiefel	$\widetilde{\beta}^{HS}_{k+1} = \dfrac{\overline{\mathbf{g}}^T_{k+1}(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)^T\mathbf{p}_k}$	$\widehat{\beta}^{HS}_{k+1} = \dfrac{\mathbf{g}^T_{k+1}(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{(\mathbf{g}_{k+1} - \mathbf{g}_k)^T\mathbf{p}_k}$

direction $\overline{\mathbf{g}}_k$ from Equation (2.21) simply becomes

$$\overline{\mathbf{g}}_k = \mathbf{x}_k - \overline{\mathbf{x}}_k = \mathbf{P}\mathbf{r}_k = \mathbf{P}\mathbf{g}_k. \tag{2.28}$$

This immediately suggests a generalization of the linearly preconditioned NCG parameters $\breve{\beta}_{k+1}$ of Equations (2.17)–(2.19) to the case of nonlinear preconditioning: replacing all occurrences of $\mathbf{P}\mathbf{g}_k$ with $\overline{\mathbf{g}}_k$ we obtain the expressions

$$\widehat{\beta}^{FR}_{k+1} = \frac{\mathbf{g}^T_{k+1}\overline{\mathbf{g}}_{k+1}}{\mathbf{g}^T_k\overline{\mathbf{g}}_k}, \tag{2.29}$$

$$\widehat{\beta}^{PR}_{k+1} = \frac{\mathbf{g}^T_{k+1}(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{\mathbf{g}^T_k\overline{\mathbf{g}}_k}, \tag{2.30}$$

$$\widehat{\beta}^{HS}_{k+1} = \frac{\mathbf{g}^T_{k+1}(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{(\mathbf{g}_{k+1} - \mathbf{g}_k)^T\mathbf{p}_k}. \tag{2.31}$$

Expressions of this type have been used in [9, 24, 70, 106]. It is clear that the PNCG algorithm with this second set of expressions, which are listed in the right column of Table 2.1, reduces to the PCG algorithm in the linear case, since the $\widehat{\beta}_{k+1}$ reduce to the $\breve{\beta}_{k+1}$ in the case of a linear preconditioner, and the $\breve{\beta}_{k+1}$ in turn reduce to the β_{k+1} from the PCG algorithm when solving an SPD linear system. For completeness, we state this formally in the following theorem.

Theorem 1. *Let $\boldsymbol{A}$ and $\mathbf{P}$ be SPD matrices. Then PNCG (Algorithm 3) with $\widehat{\beta}_{k+1}^{FR}$, $\widehat{\beta}_{k+1}^{PR}$ or $\widehat{\beta}_{k+1}^{HS}$ of Table 2.1 applied to the convex quadratic problem $\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\boldsymbol{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}$ using an exact line search and a symmetric linear stationary iterative method with preconditioning matrix $\mathbf{P}$ as the preconditioner, reduces to PCG (Algorithm 2) applied to the linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with the same preconditioner.*

Thus, for the nonlinearly preconditioned NCG method, we have two sets of $\overline{\beta}_{k+1}$ formulas: the $\widehat{\beta}_{k+1}$ formulas have the property that the PNCG algorithm reduces to the PCG algorithm in the linear case, whereas the $\widetilde{\beta}_{k+1}$ formulas do not enjoy this property. Due to this property, one might expect the $\widehat{\beta}_{k+1}$ formulas to perform better, but our numerical tests show that this is not necessarily the case. Hence, we will use both the $\widetilde{\beta}_{k+1}$ and $\widehat{\beta}_{k+1}$ formulas in our numerical tests.

Next we investigate aspects of convergence of the PNCG algorithm. For the NCG algorithm without preconditioning, global convergence can be proved for the Fletcher-Reeves method applied to a broad class of objective functions, in the sense that

$$\liminf_{k\to\infty} \|\mathbf{g}_k\| = 0, \tag{2.32}$$

when the line search satisfies the strong Wolfe conditions (see [44, 81] for a general discussion on NCG convergence). Global convergence cannot be proved in general for the Polak-Ribière or Hestenes-Stiefel variants. Nevertheless, these methods are also widely used and may perform better than Fletcher-Reeves in practice. Global convergence can be proved for variants of these methods in which every search direction $\mathbf{p}_k$ is guaranteed to be a descent direction ($\mathbf{g}_k^T\mathbf{p}_k < 0$), and in which the iteration is restarted periodically with a steepest-descent step.

It should come as no surprise that general convergence results for the PNCG algorithm are also difficult to obtain: use of a nonlinear preconditioner only exacerbates the already considerable theoretical difficulties in analyzing the convergence properties of these types of nonlinear optimization methods. However, with the use of the following theorem we will be able to establish global convergence for a restarted version of the $\widehat{\beta}_k^{FR}$ variant of the PNCG algorithm with a line search satisfying the strong Wolfe conditions, under the condition that the nonlinear preconditioner produces descent directions. Since the proof is dependent on the line search satisfying the strong Wolfe conditions we include the conditions for completeness.

Strong Wolfe Conditions. The strong Wolfe conditions require the step length parameter, α_k, in the update equation, $x_{k+1} = x_k + \alpha_k p_k$, of any line search method to satisfy the following:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k, \tag{2.33}$$

$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq c_2 |\nabla f_k^T p_k|, \tag{2.34}$$

with $0 < c_1 < c_2 < 1$.

Condition (2.33) is known as the *sufficient decrease* or *Armijo condition* and condition (2.34) is known as the *curvature condition*. The proof of our theorem relies on condition (2.34). We will use this condition to help show that the PNCG search directions $\mathbf{p}_k$ obtained using $\widehat{\beta}_k^{FR}$ are descent directions when the nonlinear preconditioner produces descent directions. To show this we follow the proof technique in [81, Lemma 5.6].

Theorem 2. *Consider the PNCG algorithm given in Algorithm 3 with* $\overline{\beta}_{k+1} = \widehat{\beta}_{k+1}^{FR}$ *and where* α_k *satisfies the strong Wolfe conditions. Let* $P(\mathbf{x})$ *be a nonlinear preconditioner such that* $-\overline{\mathbf{g}}(\mathbf{x}_k) = P(\mathbf{x}_k) - \mathbf{x}_k$ *is a descent direction for all* k, *i.e.,* $-\mathbf{g}_k^T \overline{\mathbf{g}}_k < 0$. *Suppose the objective function* f *is bounded below in* $\mathbb{R}^n$ *and* f *is continuously differentiable in an open set* $\mathcal{N}$ *containing the level set* $\mathcal{L} := \{\mathbf{x} : f(\mathbf{x}) \leq f(\mathbf{x}_0)\}$, *where* $\mathbf{x}_0$ *is the starting point of the iteration. Assume also that the gradient* $\mathbf{g}_k$ *is Lipschitz continuous on* $\mathcal{N}$. *Then,*

$$\sum_{k \geq 0} \cos^2 \theta_k \|\mathbf{g}_k\|^2 < \infty, \tag{2.35}$$

where

$$\cos \theta_k = \frac{-\mathbf{g}_k^T \mathbf{p}_k}{\|\mathbf{g}_k\| \|\mathbf{p}_k\|}. \tag{2.36}$$

Proof. We show that $\mathbf{p}_k$ is a descent direction, i.e., $\mathbf{g}_k^T \mathbf{p}_k < 0 \ \forall \ k$. Then condition (2.35) follows directly from [81, Theorem 3.2] which states that condition (2.35) holds for any iteration of the form $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ provided that the above conditions hold for α_k, f and $\mathbf{g}_k$, and where $\mathbf{p}_k$ is a descent direction.

Instead of proving that $\mathbf{g}_k^T \mathbf{p}_k < 0$ directly, we will prove the following:

$$-\frac{1}{1 - c_2} \leq \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_k^T \overline{\mathbf{g}}_k} \leq \frac{2c_2 - 1}{1 - c_2}, \quad k \geq 0, \tag{2.37}$$

where $0 < c_2 < \frac{1}{2}$ is the constant from the curvature condition of the strong Wolfe conditions:

$$|\mathbf{g}_{k+1}^T \mathbf{p}_k| \leq c_2 |\mathbf{g}_k^T \mathbf{p}_k|. \tag{2.38}$$

Note, that the function $t(\xi) = (2\xi - 1)/(1 - \xi)$ is monotonically increasing on the interval $[0, \frac{1}{2}]$ and that $t(0) = -1$ and $t(\frac{1}{2}) = 0$. Thus, because $c_2 \in (0, \frac{1}{2})$, we have

$$-1 < \frac{2c_2 - 1}{1 - c_2} < 0. \tag{2.39}$$

Also note that since $-\overline{\mathbf{g}}_k$ is a descent direction, $\mathbf{g}_k^T \overline{\mathbf{g}}_k > 0$. So, if (2.37) holds then $\mathbf{g}_k^T \mathbf{p}_k < 0$ and $\mathbf{p}_k$ is a descent direction.

We use an inductive proof to show that (2.37) is true. For $k = 0$, we use the definition of $\mathbf{p}_0$ to get,

$$\frac{\mathbf{g}_0^T \mathbf{p}_0}{\mathbf{g}_0^T \overline{\mathbf{g}}_0} = \frac{-\mathbf{g}_0^T \overline{\mathbf{g}}_0}{\mathbf{g}_0^T \overline{\mathbf{g}}_0} = -1. \tag{2.40}$$

From (2.39) we have

$$\frac{\mathbf{g}_0^T \mathbf{p}_0}{\mathbf{g}_0^T \overline{\mathbf{g}}_0} = -1 \leq \frac{2c_2 - 1}{1 - c_2}. \tag{2.41}$$

Note, that the function $t(\xi) = -1/(1 - \xi)$ is monotonically decreasing on the interval $[0, \frac{1}{2}]$ and that $t(0) = -1$ and $t(\frac{1}{2}) = -2$. Thus, because $c_2 \in (0, \frac{1}{2})$, we have

$$-2 < -\frac{1}{1 - c_2} < -1. \tag{2.42}$$

Thus,

$$\frac{\mathbf{g}_0^T \mathbf{p}_0}{\mathbf{g}_0^T \overline{\mathbf{g}}_0} = -1 \geq -\frac{1}{1 - c_2}. \tag{2.43}$$

Now suppose

$$-\frac{1}{1 - c_2} \leq \frac{\mathbf{g}_l^T \mathbf{p}_l}{\mathbf{g}_l^T \overline{\mathbf{g}}_l} \leq \frac{2c_2 - 1}{1 - c_2}, \quad l = 1, \ldots, k. \tag{2.44}$$

We need to show that (2.37) is true for $k + 1$. Using the definition of $\mathbf{p}_{k+1}$ we have,

$$\begin{aligned} \frac{\mathbf{g}_{k+1}^T \mathbf{p}_{k+1}}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} &= \frac{\mathbf{g}_{k+1}^T \left(-\overline{\mathbf{g}}_{k+1} + \widehat{\beta}_{k+1}^{FR} \mathbf{p}_k \right)}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \\ &= -1 + \widehat{\beta}_{k+1}^{FR} \frac{\mathbf{g}_{k+1}^T \mathbf{p}_k}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}}. \end{aligned} \tag{2.45}$$

From the Wolfe condition, Equation (2.38), and the inductive hypothesis, which implies that $\mathbf{g}_k^T \mathbf{p}_k < 0$, we can write

$$c_2 \mathbf{g}_k^T \mathbf{p}_k \leq \mathbf{g}_{k+1}^T \mathbf{p}_k \leq -c_2 \mathbf{g}_k^T \mathbf{p}_k. \tag{2.46}$$

Combining this with Equation (2.45), we have

$$-1 + c_2 \widehat{\beta}_{k+1}^{FR} \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \leq \frac{\mathbf{g}_{k+1}^T \mathbf{p}_{k+1}}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \leq -1 - c_2 \widehat{\beta}_{k+1}^{FR} \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \tag{2.47}$$

So,

$$\begin{aligned}
\frac{\mathbf{g}_{k+1}^T \mathbf{p}_{k+1}}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} &\geq -1 + c_2 \widehat{\beta}_{k+1}^{FR} \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \\
&= -1 + c_2 \left(\frac{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}}{\mathbf{g}_k^T \overline{\mathbf{g}}_k} \right) \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \\
&= -1 + c_2 \left(\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_k^T \overline{\mathbf{g}}_k} \right) \\
&\geq -1 - \frac{c_2}{1 - c_2} \\
&= -\frac{1}{1 - c_2},
\end{aligned}$$

and

$$\begin{aligned}
\frac{\mathbf{g}_{k+1}^T \mathbf{p}_{k+1}}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} &\leq -1 - c_2 \widehat{\beta}_{k+1}^{FR} \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \\
&= -1 - c_2 \left(\frac{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}}{\mathbf{g}_k^T \overline{\mathbf{g}}_k} \right) \frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_{k+1}^T \overline{\mathbf{g}}_{k+1}} \\
&= -1 - c_2 \left(\frac{\mathbf{g}_k^T \mathbf{p}_k}{\mathbf{g}_k^T \overline{\mathbf{g}}_k} \right) \\
&\leq -1 + \frac{c_2}{1 - c_2} \\
&= \frac{2c_2 - 1}{1 - c_2}.
\end{aligned}$$

□

We can now easily establish that convergence holds for a restarted version of the PNCG algorithm with $\widehat{\beta}_{k+1}^{FR}$ if a nonlinear preconditioner is used that produces descent directions: If we use the steepest decent direction as the search direction on every mth iteration of the algorithm and then restart the PNCG algorithm with $\mathbf{p}_{m+1} = -\overline{\mathbf{g}}_{m+1} = -\mathbf{x}_m + P(\mathbf{x}_m)$, then Equation (2.35) of Theorem 2 is still satisfied for the combined process and

$$\liminf_{k \to \infty} \|\mathbf{g}_k\| = 0, \tag{2.48}$$

since $\cos\theta_k = 1$ for the steepest descent steps [81, p. 128]. Thus we are guaranteed overall global convergence for this method. Note that the proof for global convergence of NCG using β_{k+1}^{FR} without restarting [81, Theorem 5.7] does not carry over to the case of unrestarted PNCG with $\widehat{\beta}_{k+1}^{FR}$.

For (2.35) to hold we must assume that $\beta_{k+1} = \widehat{\beta}_{k+1}^{FR}$. However, if we use a more restrictive line search we can show that (2.35) holds for any variant of β_{k+1} provided that the remaining assumptions of Theorem 2 hold. Suppose we use an "ideal" line search at every step of the PNCG algorithm where a line search is considered ideal if α_k is a stationary point of $f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$. If α_k is a stationary point of $f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$ then $\nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k = \mathbf{g}_{k+1}^T \mathbf{p}_k = 0$ and using the definition of $\mathbf{p}_k$ we have

$$\begin{aligned}
\mathbf{g}_k^T \mathbf{p}_k &= \mathbf{g}_k^T \left(-\overline{\mathbf{g}}_k + \beta_k \mathbf{p}_{k-1}\right) \\
&= -\mathbf{g}_k^T \overline{\mathbf{g}}_k + \beta_k \mathbf{g}_k^T \mathbf{p}_{k-1} \\
&= -\mathbf{g}_k^T \overline{\mathbf{g}}_k < 0,
\end{aligned}$$

provided $-\overline{\mathbf{g}}_k$ is a descent direction. Thus, $\mathbf{p}_k$ is a descent direction for all k and (2.35) holds for all variants of β_{k+1}. This implies that any restarted version of the PNCG algorithm with an ideal line search is guaranteed to converge provided $-\overline{\mathbf{g}}_k$ is a descent direction. See [24] for a similar proof when $\overline{\mathbf{g}}_k = \mathbf{M}^{-1}\mathbf{g}_k$ and $\mathbf{M}$ is a positive definite matrix, which guarantees that $-\overline{\mathbf{g}}_k$ is a descent direction. We should also note that performing an ideal line search at every step of the PNCG algorithm is often prohibitively expensive and thus not used in practice.

Both convergence results require that the nonlinearly preconditioned directions $-\overline{\mathbf{g}}_k = P(\mathbf{x}_k) - \mathbf{x}_k$ be descent directions. If one assumes a continuous preconditioning function $P(\mathbf{x})$ such that $-\overline{\mathbf{g}}(\mathbf{x}) = P(\mathbf{x}) - \mathbf{x}$ is a descent direction for all $\mathbf{x}$ in a neighbourhood of an isolated local minimizer $\mathbf{x}^*$ of a continuously differentiable objective function $f(\mathbf{x})$, then this implies that the nonlinear preconditioner satisfies the fixed-point condition $\mathbf{x}^* = P(\mathbf{x}^*)$, which is a natural condition for a nonlinear preconditioner. It is often the case in nonlinear optimization that convergence results only hold under restrictive conditions and are mainly of theoretical value. In practice, numerical results may show satisfactory convergence behaviour for much broader classes of problems. Our numerical results will show that this is also the case for PNCG applied to canonical tensor decomposition: While the ALS preconditioner satisfies the fixed-point property, it is not guaranteed to produce descent directions. Our PNCG algorithm accounts for this and convergence was generally observed numerically for all the PNCG variants we considered, with the $\widetilde{\beta}_{PR}$ variant producing the fastest results in most cases.

2.3.4 Application of the PNCG Algorithm to the CP Optimization Problem

Thus far we have described the PNCG algorithm in very general terms. The algorithm can be applied to any continuously differentiable function bounded from below using any nonlinear iterative method as a preconditioner. We now discuss how to apply Algorithm 3 to the CP optimization problem. The two quantities that are most important in the computation of Algorithm 3 are the gradient, $\mathbf{g}_k$, and the preconditioned value, $P(\mathbf{x}_k)$. Not only is the gradient used in some of the formulas for β_{k+1}, it is also used in calculating the step length parameter, α_k. We choose to use the ALS algorithm as a preconditioner since it is the standard algorithm used to solve the CP decomposition problem. We briefly revisit the ALS algorithm before discussing the computation of the gradient, $\mathbf{g}_k$. The CP optimization problem for an N-way tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is given by

$$\min f(\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}) = \frac{1}{2} \|\mathcal{X} - [\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]\|_F^2, \tag{2.49}$$

where $\mathbf{A}^{(n)},\ n = 1, \ldots, N$, is a factor matrix of size $I_n \times R$ and the following size parameters are defined:

$$K = \prod_{l=1}^{N} I_l, \quad \overline{K}^{(n)} = \prod_{l=1, l\neq n}^{N} I_l. \tag{2.50}$$

Rather than solve (2.49) for $\mathbf{A}^{(1)}$ through $\mathbf{A}^{(N)}$ simultaneously, the ALS algorithm solves for each factor matrix one at a time. The exact solution for each factor matrix is given by

$$\mathbf{A}^{(n)} = \mathbf{X}_{(n)} \mathbf{A}^{(-n)} (\mathbf{\Gamma}^{(n)})^{\dagger}, \tag{2.51}$$

where

$$\mathbf{\Gamma}^{(n)} = \mathbf{\Upsilon}^{(1)} * \cdots * \mathbf{\Upsilon}^{(n-1)} * \mathbf{\Upsilon}^{(n+1)} * \cdots * \mathbf{\Upsilon}^{(N)}, \tag{2.52}$$

$$\mathbf{\Upsilon}^{(n)} = \mathbf{A}^{(n)T} \mathbf{A}^{(n)}, \tag{2.53}$$

$$\mathbf{A}^{(-n)} = \mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)}, \tag{2.54}$$

where $\odot$ is the Khatri-Rho or Hadamard product [56], $*$ denotes the elementwise product, and $\mathbf{X}_{(n)} \in \mathbb{R}^{I_n \times \overline{K}^{(n)}}$ is the mode-n matricization of $\mathcal{X}$, obtained by stacking the n-mode fibers of $\mathcal{X}$ in its columns in a regular way as defined in [56]. For more details of the derivation of Equation (2.51) see [56].

The primary cost of solving for $\mathbf{A}^{(n)}$ is multiplying the matricized tensor, $\mathbf{X}_{(n)}$, with the Khatri-Rao product, $\mathbf{A}^{(-n)}$. The matrix $\mathbf{X}_{(n)}$ is of size $I_n \times \overline{K}^{(n)}$ and $\mathbf{A}^{(-n)}$ is of size

$\overline{K}^{(n)} \times R$ where $\overline{K}^{(n)} = K/I_n$. Thus the cost of computing Equation (2.51), measured in terms of the number of operations, is $\mathcal{O}(KR)$. One iteration of the ALS algorithm requires us to solve for each factor matrix, thus each iteration of the ALS algorithm has a computational cost of $\mathcal{O}(NKR)$.

We now discuss the gradient of the objective function in (2.49). It can be written as a vector of matrices

$$\nabla f(\mathcal{A}_R) = \mathbf{G}(\mathcal{A}_R) = (\mathbf{G}^{(1)}, \ldots, \mathbf{G}^{(n)}), \tag{2.55}$$

where $\mathbf{G}^{(n)} \in \mathbb{R}^{I_n \times R}$, $n = 1, \ldots, N$. Each matrix $\mathbf{G}^{(n)}$, $n = 1, \ldots, N$, is given by

$$\mathbf{G}^{(n)} = -\mathbf{X}_{(n)}\mathbf{A}^{(-n)} + \mathbf{A}^{(n)}\mathbf{\Gamma}^{(n)}. \tag{2.56}$$

The derivation of (2.56) can be found in [1]. From Equation (2.56) we can see that the cost of computing one gradient matrix, $\mathbf{G}^{(n)}$, is dominated by the calculation of $\mathbf{X}_{(n)}\mathbf{A}^{(-n)}$ and thus the computational cost of computing the gradient, $\nabla f(\mathcal{A}_R)$, is $\mathcal{O}(NKR)$, the same as one iteration of the ALS algorithm.

2.4 Numerical Results

To test our PNCG algorithm we randomly generate artificial tensors of different sizes, ranks, collinearity, and heteroskedastic and homoskedastic noise, which constitute standard test problems for CP decomposition [98]. We then compare the performance of the CP factorization using the PNCG algorithm with results from using the ALS and NCG algorithms.

2.4.1 Problem Description

The artificial tensors are generated using the methodology of [98]. All the tensors we consider are 3-way tensors. Each dimension has the same size but we consider tensors of three different sizes, $I = 20$, 50 and 100. The factor matrices $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$, and $\mathbf{A}^{(3)}$ are generated randomly so that the collinearity of the factors in each mode is set to a particular level C. The steps necessary to create the factor matrices are outlined in [98]. Thus,

$$\frac{\mathbf{a}_r^{(n)T}\mathbf{a}_s^{(n)}}{\|\mathbf{a}_r^{(n)}\|\|\mathbf{a}_s^{(n)}\|} = C, \tag{2.57}$$

for $r \neq s,\ r, s = 1, \dots, R$ and $n = 1, 2, 3$. As in [1], the values of C we consider are 0.5 and 0.9, where higher values of C make the problem more difficult. We consider two different values for the rank, $R = 3$ and $R = 5$. For each combination of R and C we generate a set of factor matrices. Once we have converted these factors into a tensor and added noise our goal is to recover these underlying factors using the different optimization algorithms. From a given set of factor matrices we are able to generate nine test tensors by adding different levels of homoskedastic and heteroskedastic noise. Homoskedastic noise refers to noise with constant variance whereas heteroskedastic noise refers to noise with differing variance. The noise ratios we consider for homoskedastic and heteroskedastic noise are $l_1 = 1, 5, 10$ and $l_2 = 0, 1, 5$, respectively, see [98, 1]. Suppose $\mathcal{N}_1, \mathcal{N}_2 \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ are random tensors with entries chosen from a standard normal distribution. Then we generate the test tensors as follows. Let the original tensor be

$$\mathcal{X} = [\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \mathbf{A}^{(3)}]\!]. \tag{2.58}$$

Homoskedastic noise is added to give:

$$\mathcal{X}' = \mathcal{X} + (100/l_1 - 1)^{-\frac{1}{2}} \frac{\|\mathcal{X}\|}{\|\mathcal{N}_1\|} \mathcal{N}_1, \tag{2.59}$$

and then heteroskedastic noise is added to give:

$$\mathcal{X}'' = \mathcal{X}' + (100/l_2 - 1)^{-\frac{1}{2}} \frac{\|\mathcal{X}'\|}{\|\mathcal{N}_2 * \mathcal{X}'\|} \mathcal{N}_2 * \mathcal{X}'. \tag{2.60}$$

The optimization algorithms are applied to the test tensor $\mathcal{X}''$ and in the case where $l_2 = 0$, $\mathcal{X}'' = \mathcal{X}'$. To test the performance of each optimization algorithm we apply the algorithm to the test tensor $\mathcal{X}''$ using 20 different random starting values, where the same 20 starting values are used for each algorithm. Thus for each size, $I = 20, 50$ and 100 we generate 36 test tensors since we consider 2 different ranks, 2 different collinearity values, 1 set of factor matrices for each combination of C and R and 9 different levels of noise and for each of these test tensors we apply a given optimization algorithm 20 different times using different random starting values.

2.4.2 Results

We begin by presenting numerical results for the smallest case where $I = 20$. All numerical experiments where performed on a Linux Workstation with a Quad-Core Intel Xeon 3.16GHz processor and 8GB RAM. We use the NCG algorithm from the Poblano toolbox for MATLAB [30] which uses the Moré-Thuente line search algorithm [74]. We use the same line search algorithm for the PNCG algorithm. We use the default line search parameters specified in the Poblano toolbox which are as follows: 10^{-4} for the sufficient decrease condition tolerance, 10^{-2} for the curvature condition tolerance, an initial step length of 1 and a maximum of 20 iterations. The ALS algorithm we use is from the tensor toolbox for MATLAB [8]; however, we use a different normalization of the factors (as explained below) and we use the gradient norm as a stopping condition instead of the relative function change. In the CP decomposition, it is often useful to assume that the columns of the factor matrices, $\mathbf{A}^{(n)}$, are normalized to length one with the weights absorbed into a vector $\boldsymbol{\lambda} \in \mathbb{R}^k$. Thus

$$\mathcal{X} \approx \sum_{r=1}^{k} \lambda_r a_r^{(1)} \circ \ldots \circ a_r^{(N)}. \tag{2.61}$$

In our ALS algorithm the factors are normalized such that $\boldsymbol{\lambda}$ is distributed evenly over all the factors. Also note that, while the gradient norm is used as a stopping condition for the ALS algorithm, the calculation of the gradient is not included in the timing results for the ALS algorithm. For all three algorithms, ALS, NCG and PNCG, there are three stopping conditions; all are set to the same value for each algorithm. They are as follows: 10^{-9} for the gradient norm divided by the number of variables, $\|\mathbf{G}(\mathcal{A}_R)\|_2/N$ where N is the number of variables in $\mathcal{X}$, 10^4 for the maximum number of iterations and 10^5 for the maximum number of function evaluations.

For $I = 20$ and $R = 3$, Table 2.2 summarizes the results for each algorithm while Table 2.3 summarizes the results for $I = 20$ and $R = 5$. For each value of the rank, R, there are two possible values for the collinearity, $C = 0.5$ and $C = 0.9$. Once the collinearity has been fixed, a test tensor is created and there are nine different combinations of homoskedastic and heteroskedastic noise added to each test tensor. We then generate 20 different initial guesses with components chosen randomly from a uniform distribution between 0 and 1. Each algorithm is tested using each of these initial guesses. Thus, for each collinearity value there are 180 CP decompositions performed by each algorithm. Each table reports the overall timings for the 180 CP decompositions. The timing is written in the form $a \pm b$ where a is the mean time and b is the standard deviation. The first number in brackets represents the number of CP decompositions that converge before reaching the maximum number of iterations or function evaluations out of a possible 180. All timing calculations are performed for the converged runs only. The second number in brackets represents the number of runs where the algorithm is able to recover the original set of factor matrices. We use a measure, defined in [98], known as congruence to determine if an algorithm is able to recover the original factors where the congruence between two rank-one tensors,

$\mathcal{X} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c}$ and $\mathcal{Y} = \mathbf{p} \circ \mathbf{q} \circ \mathbf{r}$ is defined as

$$\text{cong}(\mathcal{X}, \mathcal{Y}) = \frac{|\mathbf{a}^T\mathbf{p}|}{\|\mathbf{a}\|\|\mathbf{p}\|} \cdot \frac{|\mathbf{b}^T\mathbf{q}|}{\|\mathbf{b}\|\|\mathbf{q}\|} \cdot \frac{|\mathbf{c}^T\mathbf{r}|}{\|\mathbf{c}\|\|\mathbf{r}\|}. \tag{2.62}$$

If the congruence is above 0.97 ($\approx 0.99^3$) for every component rank-one tensor then we say that the algorithm has successfully recovered the original factor matrices. Since the CP decomposition is unique up to a permutation of the component rank-one tensors, we consider all permutations when calculating congruences and choose the permutation that results in the greatest sum of congruences of rank-one tensors. We also calculate the congruences for all runs regardless of whether or not they converge.

Table 2.2: Speed Comparison with $R = 3$ and $I = 20$.

Optimization Method	Mean Time (Seconds)	
	$C = 0.5$	$C = 0.9$
ALS	**0.1644 ± 0.0185 (180) (180)**	5.3182 ± 1.1356 **(180) (99)**
NCG - β^{FR}	0.8617 ± 0.6658 **(178) (180)**	4.1649 ± 3.8092 **(172) (98)**
PNCG - $\widetilde{\beta}^{FR}$	1.1707 ± 0.5962 **(180) (180)**	1.7556 ± 0.5792 **(180) (99)**
PNCG - $\widehat{\beta}^{FR}$	1.5308 ± 0.7196 **(180) (180)**	2.1131 ± 1.3339 **(180) (99)**
NCG - β^{PR}	0.5170 ± 0.5300 **(179) (180)**	3.5328 ± 2.7377 **(167) (97)**
PNCG - $\widetilde{\beta}^{PR}$	0.3434 ± 0.2611 **(180) (180)**	**0.9676 ± 0.2020 (180) (99)**
PNCG - $\widehat{\beta}^{PR}$	0.4087 ± 0.2592 **(180) (180)**	0.9979 ± 0.3077 **(180) (99)**
NCG - β^{HS}	0.4457 ± 0.3458 **(178) (180)**	3.1265 ± 2.0725 **(167) (98)**
PNCG - $\widetilde{\beta}^{HS}$	0.4969 ± 0.4234 **(180) (180)**	3.3269 ± 3.6214 **(180) (99)**
PNCG - $\widehat{\beta}^{HS}$	0.4675 ± 0.3095 **(180) (180)**	1.0267 ± 0.4513 **(180) (99)**

From the results it is clear that when $C = 0.5$, ALS is the fastest algorithm. The results also indicate that for a given formula for β, NCG is faster that either PNCG algorithm. However, when the collinearity is 0.5, it is known that the problem is relatively easy [98, 1, 25], so we don't necessarily expect the preconditioned algorithm to outperform the standard algorithm, and the additional time needed to perform the preconditioning may actually slow the algorithm down relative to the original algorithm. The results change

Table 2.3: Speed Comparison with $R = 5$ and $I = 20$.

Optimization Method	Mean Time (Seconds)	
	$C = 0.5$	$C = 0.9$
ALS	**0.2517** $\pm$ **0.0663** **(180)** **(180)**	13.8499 $\pm$ 5.8256 **(106)** **(20)**
NCG - β^{FR}	0.9723 $\pm$ 0.3944 **(180)** **(180)**	9.5120 $\pm$ 6.7666 **(94)** **(20)**
PNCG - $\widetilde{\beta}^{FR}$	2.4235 $\pm$ 1.0916 **(180)** **(180)**	4.4674 $\pm$ 1.6256 **(104)** **(20)**
PNCG - $\widehat{\beta}^{FR}$	3.0196 $\pm$ 1.8507 **(179)** **(180)**	7.1644 $\pm$ 5.4327 **(106)** **(20)**
NCG - β^{PR}	0.5730 $\pm$ 0.3607 **(180)** **(180)**	7.0099 $\pm$ 4.3507 **(83)** **(20)**
PNCG - $\widetilde{\beta}^{PR}$	1.7628 $\pm$ 12.6569 **(180)** **(180)**	**2.7751** $\pm$ **1.9319** **(109)** **(20)**
PNCG - $\widehat{\beta}^{PR}$	1.2049 $\pm$ 2.0744 **(180)** **(180)**	4.1549 $\pm$ 5.0031 **(108)** **(20)**
NCG - β^{HS}	0.5285 $\pm$ 0.3131 **(180)** **(180)**	6.9515 $\pm$ 4.6067 **(85)** **(20)**
PNCG - $\widetilde{\beta}^{HS}$	0.9940 $\pm$ 1.1962 **(180)** **(180)**	5.1334 $\pm$ 5.6721 **(107)** **(20)**
PNCG - $\widehat{\beta}^{HS}$	1.4841 $\pm$ 3.4182 **(180)** **(180)**	5.5534 $\pm$ 12.4827 **(108)** **(20)**

when we look at the more difficult problem, $C = 0.9$. In this case, the PNCG algorithm with the $\widetilde{\beta}^{PR}$ variant is the fastest. The ALS algorithm is the slowest and for a given formula for β, both PNCG algorithms are faster than the NCG algorithm by a factor between 2 and 4: nonlinear preconditioning significantly speeds up the NCG algorithm. The one exception is for $R = 3$ where the PNCG algorithm with $\widetilde{\beta}^{HS}$ is slower than the NCG algorithm. However, in this case the number of convergent runs for $\widetilde{\beta}^{HS}$ is 100% while only 92.78% of the runs for β^{HS} are convergent. We also see from Tables 2.2 and 2.3 that the number of times each algorithm is able to recover the original factor matrices successfully is approximately the same for each algorithm for a given combination of R and C. In the case where $R = 5$ and $C = 0.9$, this number is quite low, however, these results closely match the results found in [1] and we note that all of the successful runs occur when there is very little noise ($l_1 = 1$ and l_2=0).

Returning to the timing results displayed in Tables 2.2 and 2.3, we recognize that the results may be dominated by a small number of difficult problems. Even with fixed problem parameters, a problem can be difficult (and require a large amount of iterations to converge) or easy depending on the particular random realization of the test tensor and/or the initial guess. Including the standard deviation helps to describe the effects of this bias; however, the timing results don't account for the problems where the algorithm fails to converge within the prescribed resource limit. One way to overcome this is to use the performance profiles suggested by Dolan and Moré in [28].

Suppose that we want to compare the performance of a set of algorithms or solvers $\mathcal{S}$ on a test set $\mathcal{P}$. Suppose there are n_s algorithms and n_p problems. For each problem $p \in \mathcal{P}$ and algorithm $s \in \mathcal{S}$ let $t_{p,s}$ be the computing time required to solve problem p using algorithm s. In order to compare algorithms we use the best performance by any algorithm as a baseline and define the performance ratio as

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : \ s \in \mathcal{S}\}}. \tag{2.63}$$

Although we may be interested in the performance of algorithm s on a given problem p, a more insightful analysis can be performed if we can obtain an overall assessment of the algorithm's performance. We can do this by defining the following:

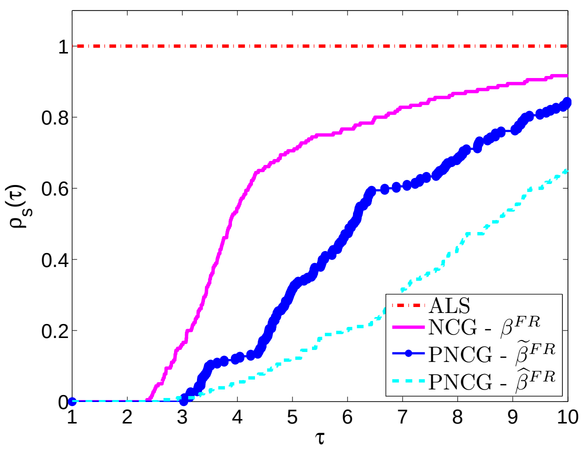
$$\rho_s(\tau) = \frac{1}{n_p}\text{size}\{p \in \mathcal{P} : r_{p,s} \leq \tau\}. \tag{2.64}$$

For algorithm $s \in \mathcal{S}$, $\rho_s(\tau)$ is the fraction of problems p for which the performance ratio $r_{p,s}$ is within a factor $\tau \in \mathbb{R}$ of the best ratio (which equals one). Thus, $\rho_s(\tau)$ is the cumulative distribution function for the performance ratio and we refer to it as the performance profile. By visually examining the performance profiles of each algorithm we can compare the algorithms in $\mathcal{S}$. In particular, algorithms with large fractions $\rho_s(\tau)$ are preferred.

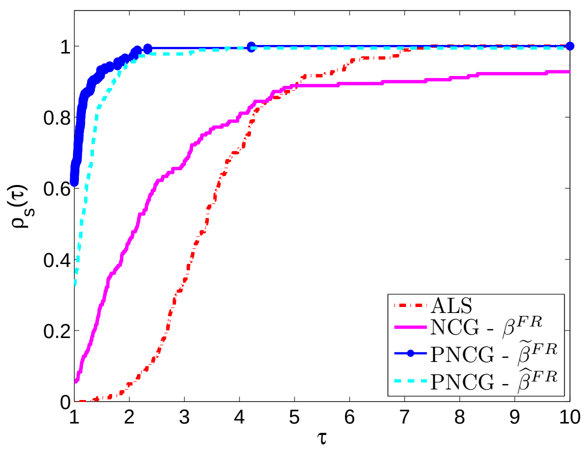
Since the performance profile, $\rho_s : \mathbb{R} \mapsto [0, 1]$, is a cumulative distribution function it is nondecreasing. In addition, it is a piecewise constant function, continuous from the right at each breakpoint. The value of ρ_s at $\tau = 1$ is the fraction of problems for which the algorithm wins over the rest of the algorithms. In other words, $\rho_s(1)$ is the fraction of wins for each solver. For larger values of τ, algorithms with high values of ρ_s relative to the other algorithms indicate robust solvers.

To examine the performance profiles of each algorithm more easily we group the NCG and PNCG algorithms according to formula for β_{k+1}, either FR, PR or HS. Thus

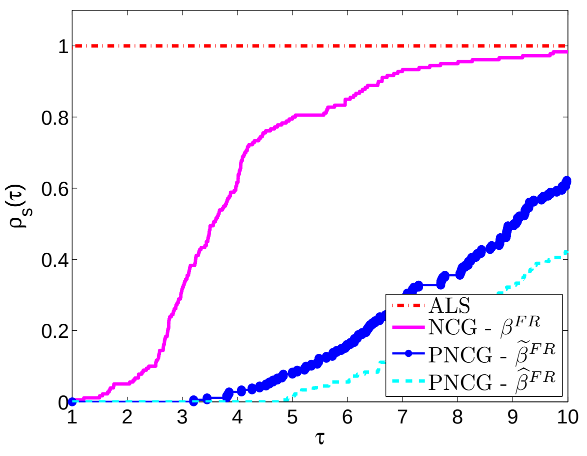
$$\mathcal{S}_1 = \{\text{ALS, NCG with } \beta^{FR}, \text{PNCG with } \widetilde{\beta}^{FR}, \text{PNCG with } \widehat{\beta}^{FR}\}, \tag{2.65}$$
$$\mathcal{S}_2 = \{\text{ALS, NCG with } \beta^{PR}, \text{PNCG with } \widetilde{\beta}^{PR}, \text{PNCG with } \widehat{\beta}^{PR}\}, \tag{2.66}$$
$$\mathcal{S}_3 = \{\text{ALS, NCG with } \beta^{HS}, \text{PNCG with } \widetilde{\beta}^{HS}, \text{PNCG with } \widehat{\beta}^{HS}\}. \tag{2.67}$$



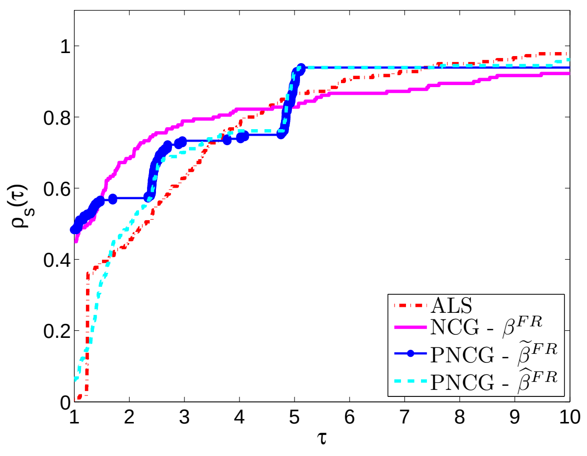
(a) $R = 3$, Collinearity$= 0.5$



(b) $R = 3$, Collinearity$= 0.9$



(c) $R = 5$, Collinearity$= 0.5$



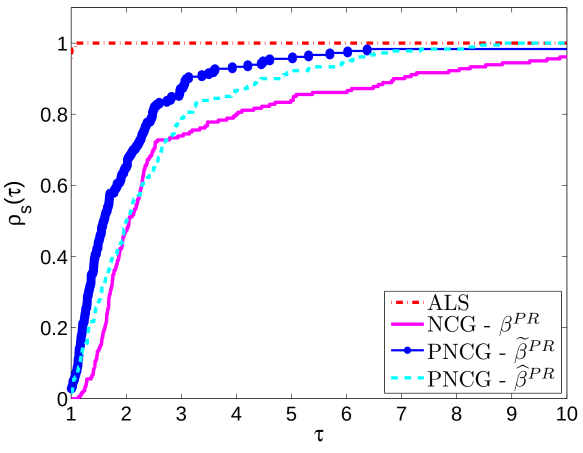
(d) $R = 5$, Collinearity$= 0.9$

Figure 2.1: Performance profiles for the algorithms in $\mathcal{S}_1$ with $I = 20$.

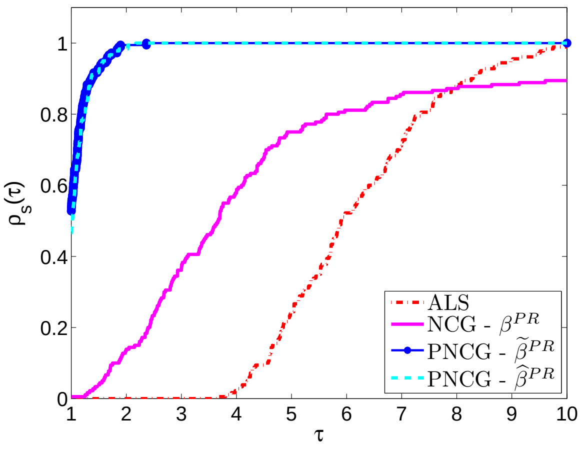
Figure 2.1 plots the performance profiles of the algorithms in $\mathcal{S}_1$. In Figure 2.1a, $R = 3$ and $C = 0.5$. This is an easy problem and from the performance profiles we can see that not only is ALS the fastest it is also the most robust. We can increase the difficulty of the problem by increasing the collinearity to 0.9 and Figure 2.1b shows the performance profiles of each algorithm in $\mathcal{S}_1$ when $C = 0.9$. Since $\rho_s(1)$ indicates what fraction of the 180 trials each algorithm is the fastest, we see that PNCG with $\widetilde{\beta}_{FR}$ is the fastest algorithm in the largest percentage of runs. When $\tau = 3$ approximately 70% of the 180 NCG runs are within three times the fastest time and approximately 40% of the ALS runs are within three times the fastest time. However, as τ increases to 10 we notice that approximately all of the ALS and PNCG runs are within ten times the fastest time but only 90% of the NCG runs are within ten times the fastest time. This suggests that the NCG algorithm without nonlinear preconditioning is not nearly as robust as the other algorithms. In Figures 2.1c and 2.1d, $R = 5$ and $C = 0.5$ and 0.9 respectively. For $C = 0.5$, the performance profiles look similar in Figures 2.1a and 2.1c where $R = 3$ and 5 respectively. For $C = 0.9$, the performance profiles in Figure 2.1b where $R = 3$ and Figure 2.1d where $R = 5$ differ, however, in both cases, PNCG with $\widetilde{\beta}^{FR}$ is the fastest in the largest percentage of runs and NCG is the least robust algorithm having the smallest value at $\tau = 10$.

Figures 2.2 and 2.3 plot the performance profiles for the algorithms in $\mathcal{S}_2$ and $\mathcal{S}_3$, respectively. Once again we see similar results as those displayed in Figure 2.1.

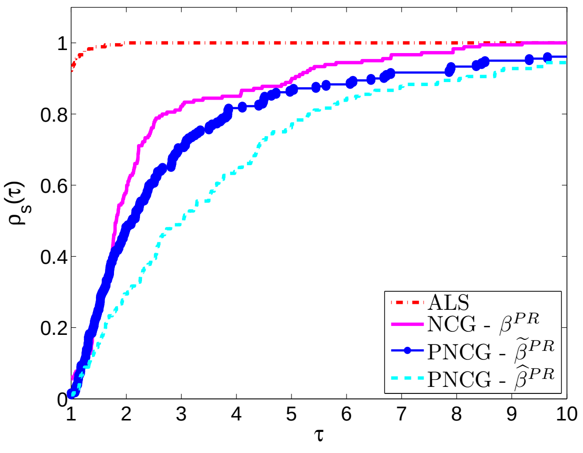
Our next challenge is to examine the performance of the PNCG algorithm when we increase the tensor size. To better understand the performance, we focus on the results for the algorithms in $\mathcal{S}_2$ since the results are similar for the algorithms in $\mathcal{S}_1$ and $\mathcal{S}_3$. We consider two different size parameters, $I = 50$ and $I = 100$. Table 2.4 reports the timing results when $I = 50$ and Table 2.5 contains the results when $I = 100$. As we increase the size of the tensors we see that the results remain similar to the case where $I = 20$. Regardless of the rank, R, the easy problem for which the collinearity is 0.5, can easily be solved by ALS. Again, when we move to the more difficult problem of $C = 0.9$, the PNCG algorithms perform the best (except for $I = 100$ and $R = 5$). These results are further reflected in the performance profiles shown in Figures 2.4 and 2.5. ALS dominates regardless of rank and size when $C = 0.5$, but Figures 2.4b, 2.4d, 2.5b, and 2.5d suggest that PNCG with the $\widetilde{\beta}^{PR}$ variant is the fastest for $C = 0.9$ except when $I = 100$ and $R = 5$, where ALS appears faster. The figures also indicate that the NCG algorithm without nonlinear preconditioning is the least robust. In the case when $I = 100$, $R = 5$ and the collinearity is 0.9, we note that Table 2.5 shows that the ALS algorithm is the fastest on average, while Figure 2.5d shows that the fastest run is most often for PNCG with the $\widetilde{\beta}^{PR}$ variant. Both variants of the PNCG algorithm are more robust than the NCG algorithm, while ALS is the most robust in this case. So we can say that, while PNCG appears significantly faster than ALS for all difficult ($C = 0.9$) problems when the number of factors R and the tensor size I are relatively small, ALS becomes competitive again with PNCG when R and I are large. Note, however, that the line search parameters in the NCG and PNCG algorithms were the same for every problem, and it may be possible to improve both the NCG and PNCG results by fine-tuning these parameters. We also see from Tables 2.4 and 2.5 that the ability of NCG to successfully recover the original



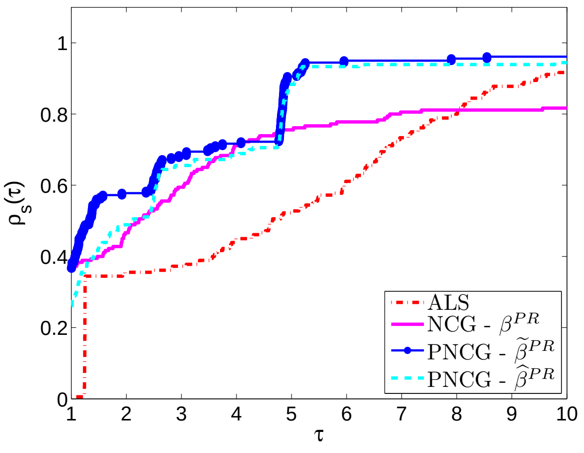
(a) $R = 3$, Collinearity= 0.5



(b) $R = 3$, Collinearity= 0.9



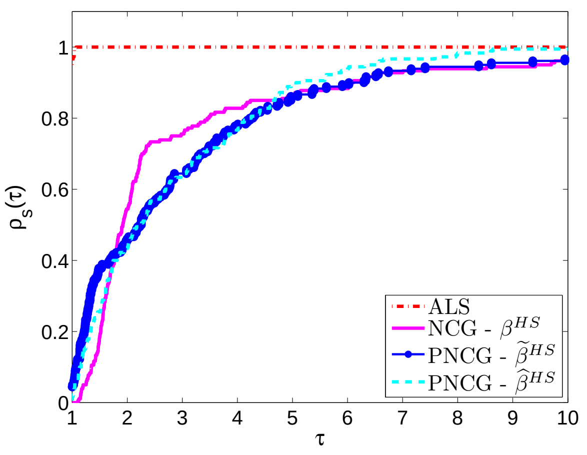
(c) $R = 5$, Collinearity= 0.5



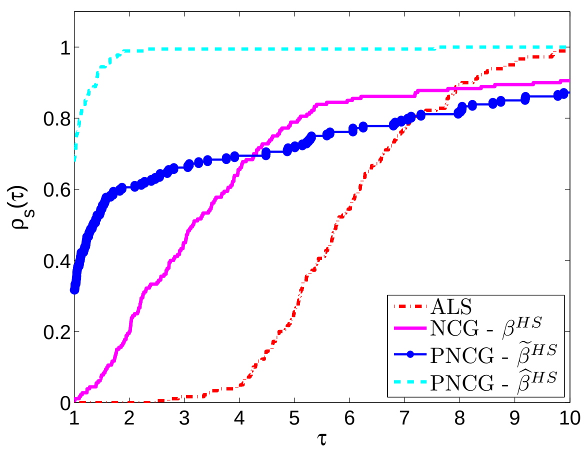
(d) $R = 5$, Collinearity= 0.9

Figure 2.2: Performance profiles for the algorithms in $\mathcal{S}_2$ with $I = 20$.

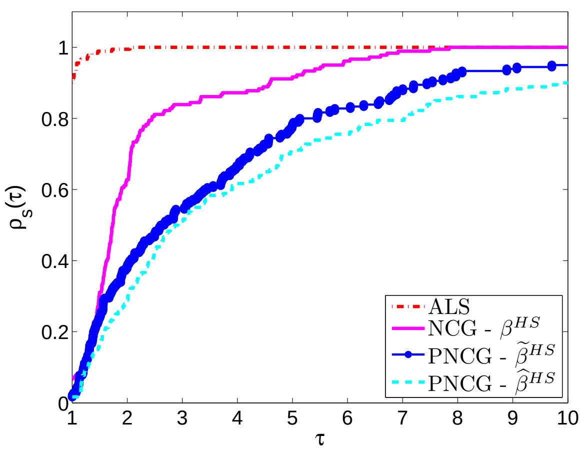
factor matrices is less than both PNCG variants and ALS in some cases. When $I = 50$ and $C = 0.9$ the difference is small for both $R = 3$ and $R = 5$. The difference is more significant when $I = 100$, $C = 0.9$ and $R = 5$, while there is no difference when $R = 3$. In all cases, the number of successes is essentially the same for both variants of PNCG and ALS. Thus, the main conclusion from our numerical tests is that nonlinear preconditioning can dramatically improve the speed and robustness of NCG: PNCG is significantly faster and more robust than NCG for all difficult ($C = 0.9$) CP problems we tested.



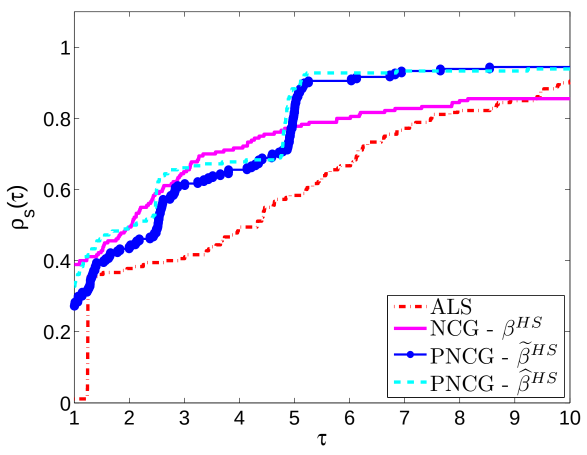
(a) $R = 3$, Collinearity= 0.5



(b) $R = 3$, Collinearity= 0.9



(c) $R = 5$, Collinearity= 0.5



(d) $R = 5$, Collinearity= 0.9

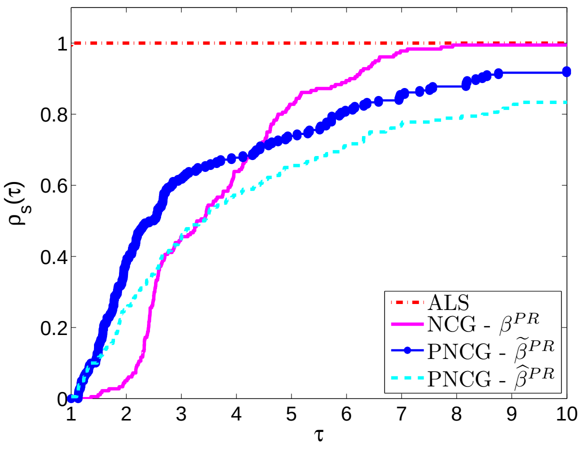
Figure 2.3: Performance profiles for the algorithms in $\mathcal{S}_3$ with $I = 20$.

Table 2.4: Speed Comparison with $I = 50$.

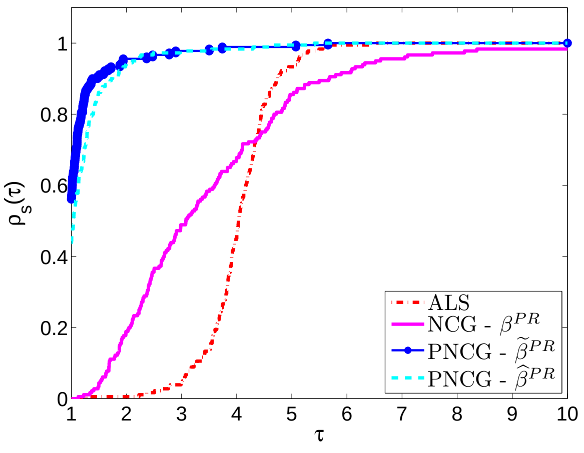
	Optimization Method	Mean Time (Seconds)	
		$C = 0.5$	$C = 0.9$
$R = 3$	ALS	**0.1988 ± 0.0368 (180) (180)**	5.1981 ± 0.3444 **(180) (180)**
	NCG - β^{PR}	0.7170 ± 0.2830 **(180) (180)**	4.4516 ± 1.9664 **(179) (171)**
	PNCG - $\widetilde{\beta}^{PR}$	0.8335 ± 0.9137 **(180) (180)**	**1.6320 ± 1.1064 (180) (180)**
	PNCG - $\widehat{\beta}^{PR}$	1.1722 ± 1.4899 **(180) (180)**	1.6676 ± 0.7855 **(180) (180)**
$R = 5$	ALS	**0.3357 ± 0.1509 (180) (180)**	10.4698 ± 3.0988 **(159) (120)**
	NCG - β^{PR}	1.6522 ± 1.2236 **(180) (180)**	14.6827 ± 10.1787 **(142) (116)**
	PNCG - $\widetilde{\beta}^{PR}$	3.8331 ± 13.5605 **(179) (179)**	**7.4386 ± 12.2583 (155) (120)**
	PNCG - $\widehat{\beta}^{PR}$	6.1021 ± 26.0100 **(179) (180)**	10.4150 ± 25.0737 **(156) (120)**

Table 2.5: Speed Comparison with $I = 100$.

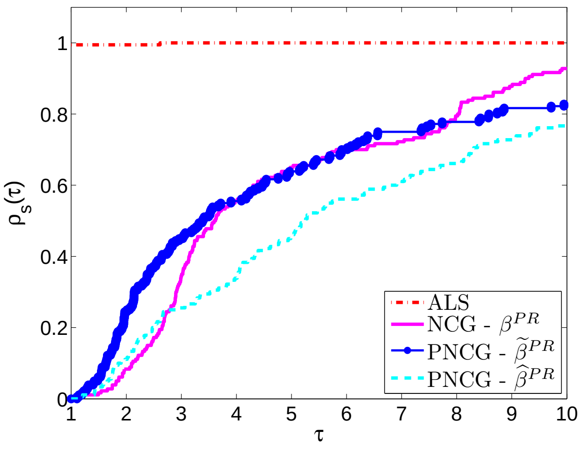
	Optimization Method	Mean Time (Seconds)	
		$C = 0.5$	$C = 0.9$
$R = 3$	ALS	**1.9006 ± 0.7043 (180) (180)**	47.3505 ± 4.3030 **(180) (180)**
	NCG - β^{PR}	14.3840 ± 6.1019 **(180) (180)**	94.9786 ± 89.6489 **(180) (180)**
	PNCG - $\widetilde{\beta}^{PR}$	15.3848 ± 24.8887 **(180) (180)**	**28.2346 ± 30.9428 (180) (180)**
	PNCG - $\widehat{\beta}^{PR}$	20.8161 ± 31.6531 **(180) (180)**	34.8675 ± 46.9708 **(180) (180)**
$R = 5$	ALS	**1.9770 ± 0.4002 (180) (180)**	**57.1086 ± 5.5332 (180) (179)**
	NCG - β^{PR}	14.8031 ± 6.2776 **(180) (180)**	124.5449 ± 95.9350 **(178) (138)**
	PNCG - $\widetilde{\beta}^{PR}$	44.2358 ± 205.5225 **(180) (179)**	103.7680 ± 257.0952 **(178) (178)**
	PNCG - $\widehat{\beta}^{PR}$	66.7177 ± 157.0857 **(180) (180)**	151.7887 ± 356.2924 **(180) (179)**



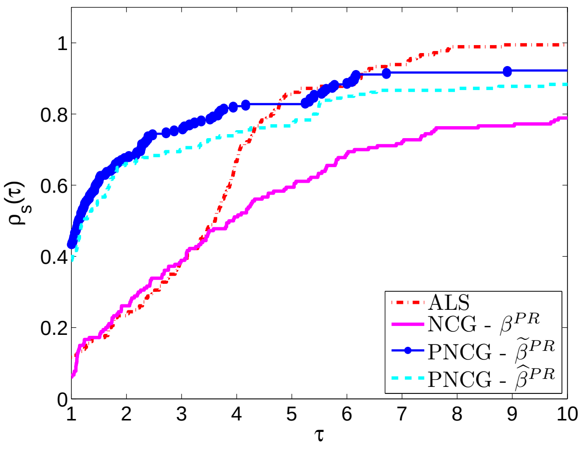
(a) $R = 3$, Collinearity= 0.5



(b) $R = 3$, Collinearity= 0.9



(c) $R = 5$, Collinearity= 0.5

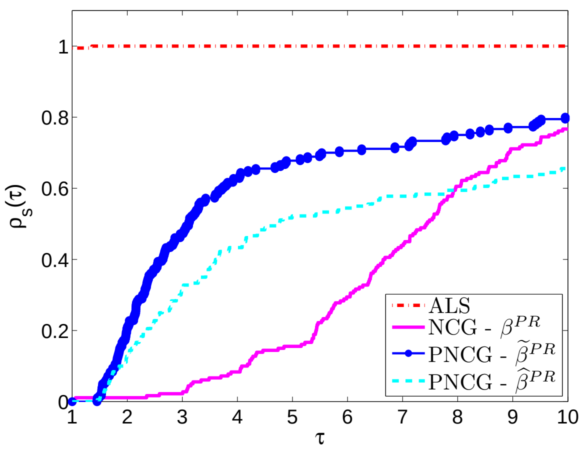


(d) $R = 5$, Collinearity= 0.9

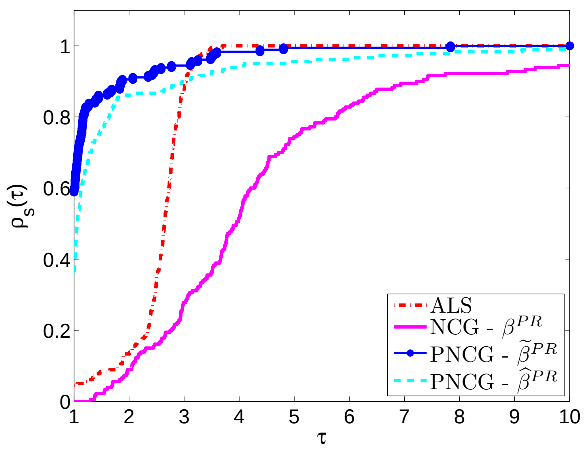
Figure 2.4: Performance profiles for the algorithms in $\mathcal{S}_2$ with $I = 50$.

2.5 Conclusion

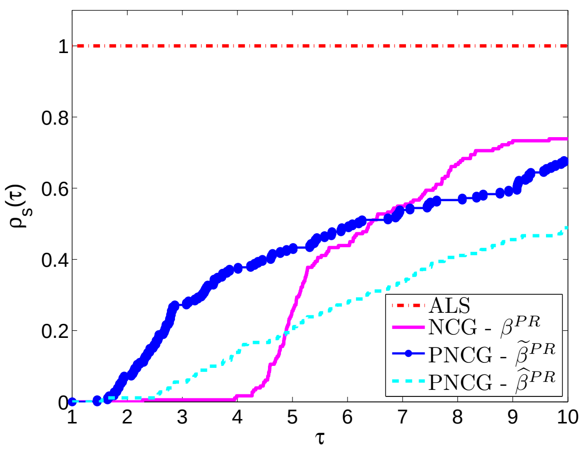
We have proposed an algorithm for computing the canonical rank-R tensor decomposition that applies ALS as a nonlinear preconditioner to the nonlinear conjugate gradient algorithm. We consider the ALS algorithm as a preconditioner because it is the standard algorithm used to compute the canonical rank-R tensor decomposition but it is known to



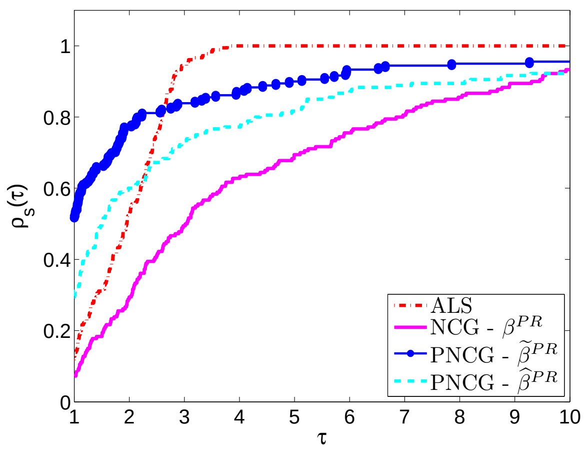
(a) $R = 3$, Collinearity= 0.5



(b) $R = 3$, Collinearity= 0.9



(c) $R = 5$, Collinearity= 0.5



(d) $R = 5$, Collinearity= 0.9

Figure 2.5: Performance profiles for the algorithms in $\mathcal{S}_2$ with $I = 100$.

converge very slowly for certain problems, for which acceleration by NCG is expected to be beneficial. We have considered several approaches for incorporating the nonlinear preconditioner into the NCG algorithm that have been described in the literature [9, 24, 70, 106, 17], corresponding to two different sets of preconditioned formulas for the standard FR, PR and HS update parameter, β, namely the $\widetilde{\beta}$ and $\widehat{\beta}$ formulas. If we use the $\widehat{\beta}$ formulas and apply the PNCG algorithm using an SPD preconditioner to a convex quadratic function using an exact line search, then the PNCG algorithm simplifies to the PCG algorithm. Also, we proved a new convergence result for one of the PNCG variants under suitable conditions, building on known convergence results for non-preconditioned NCG when line searches are used that satisfy the strong Wolfe conditions. Note that it is very easy to extend existing NCG software with the nonlinear preconditioning mechanism. Our simulation code and examples can be found at www.math.uwaterloo.ca/~hdesterc/pncg.html.

Following the methodology of [98] we create numerous test tensors and perform extensive numerical tests comparing the PNCG algorithm to the ALS and NCG algorithms. We consider a wide range of tensor sizes, ranks, factor collinearity and noise levels. Results in [1] showed that ALS is normally faster than NCG. In this chapter, we show that NCG preconditioned with ALS (or, equivalently, ALS accelerated by NCG) is often significantly faster than ALS by itself, for difficult problems. For easy problems, where the collinearity is 0.5, ALS outperforms all other algorithms. However, when the problem becomes more difficult and the collinearity is 0.9, the PNCG algorithm is often the fastest algorithm. The only case where ALS is faster is when we consider our largest tensor size and highest rank. The performance profiles of each algorithm also show that for the more difficult problems, PNCG is consistently both more robust and faster than the NCG algorithm. For our optimization problems, we generally obtain convergent results for all of the six variants of the PNCG algorithm we considered. It is interesting that for the PDE problems of [17], out of the $\widetilde{\beta}$ variants, only $\widetilde{\beta}^{PR}$ was found viable. It appears that the $\widehat{\beta}$ variants were not investigated in [17]. We did find for our test tensors that the $\widetilde{\beta}^{PR}$ formula, which does not reduce to PCG in the linear case, converges the fastest for most cases.

The PNCG algorithm discussed in this chapter is formulated under a general framework. While this approach has met with success previously in certain application areas [9, 24, 70, 106, 17] and may offer promising avenues for further applications, it appears that the nonlinearly preconditioned NCG approach has received relatively little attention in the broader community and remains underexplored both theoretically and experimentally. It will be interesting to investigate the effectiveness of PNCG for other nonlinear optimization problems. Other nonlinear least-squares optimization problems for which ALS solvers are available are good initial candidates for further study. In fact, in the next chapter we explore we how to use the PNCG algorithm to accelerate the convergence of ALS-based optimization methods for collaborative filtering models where collaborative filtering algorithms are important building blocks in many practical recommendation systems. However, as with PCG for SPD linear systems [92, Chapter 10], it is fully expected that devising effective preconditioners for more general nonlinear optimization problems will be highly problem-dependent while at the same time being crucial for gaining substantial performance benefits.

Chapter 3

A Nonlinearly Preconditioned Conjugate Gradient Algorithm for Recommendation Systems

3.1 Introduction and Background

Recommendation systems are designed to analyze available user data to recommend items such as movies, music, or other goods to consumers, and have become an increasingly important part of most successful online businesses. One strategy for building recommendation systems is known as collaborative filtering whereby items are recommended to users by collecting preferences or taste information from many users. Collaborative filtering methods provide the basis for many recommendation systems [14] and have been used by online businesses such as Amazon [64], Netflix [10], and Spotify [50].

Most collaborative filtering methods can be classified as either one of two types: neighborhood methods or latent factor models. Originally, neighborhood methods estimated unknown ratings using the ratings of similar users [47]. However, an item-oriented approach was later proposed as an alternative [93, 64] whereby ratings are estimated using the known ratings made by the same user on similar items. Not only are these item-oriented methods more accurate, they may scale better [11, 93, 96] and they provide a better explanation of user behavior since users are familiar with items previously preferred by them but do not know the users similar to them as assumed under the user-oriented approach. Latent factor models offer an alternative approach to collaborative filtering. In a latent factor model the user ratings are used to characterize both users and items in terms of latent features or factors. For movies, for example, a latent factor might measure the degree to which a movie is a comedy versus a drama and might quantify how much a user prefers comedies over dramas.

Low-rank matrix factorizations are the simplest means of realizing a latent factor model and while no method appears to be optimal on its own, low-rank matrix factorizations have repeatedly demonstrated better accuracy than other methods such as nearest neighbor models and restricted Boltzmann machines [10, 29]. The goal of low-rank matrix factorization is to determine from the user-item ratings matrix, $\mathbf{R}$, low-rank user ($\mathbf{U}$) and item ($\mathbf{M}$) matrices in which each row in these matrices represents a latent feature or factor of the data and $\mathbf{R} \approx \mathbf{U}^T\mathbf{M}$. Once the user and item matrices are computed, they are used to build the recommendation system. In the simplest case, the product of $\mathbf{U}$ and $\mathbf{M}$ is used to predict unknown ratings. Regardless, computing user and item matrices is the first step in building a variety of recommendation systems, so computing the factorization of $\mathbf{R}$ quickly is important.

The matrix factorization problem is closely related to the singular value decomposition (SVD), but the SVD can only be applied to matrices with no missing values. However, since $\mathbf{R} \approx \mathbf{U}^T\mathbf{M}$, one way to find the user and item matrices is by minimizing the squared difference between the approximated and actual value for the known ratings in $\mathbf{R}$. Minimizing this difference is typically done by one of two algorithms: stochastic gradient descent or alternating least squares (ALS) [59]. Stochastic gradient descent, popularized by Funk [39], is relatively easy to implement. ALS can be easily parallelized and can efficiently handle models that incorporate implicit data [49], but it is well-known that ALS can require a large number of iterations to converge. Thus, we propose using the PNCG algorithm introduced in the previous chapter with ALS as the nonlinear preconditioner to compute the user and item matrices to significantly improve the convergence speed relative to the ALS algorithm. We demonstrate that PNCG with ALS as a nonlinear preconditioner can significantly improve the convergence speed by testing both the ALS and PNCG algorithms on a movie ratings matrix from MovieLens [75]. We show that the PNCG algorithm significantly improves convergence speed using two different measures of convergence. One is the traditional measure from optimization, the gradient norm of the objective function, where a method is said to converge if the gradient norm is zero to a specified tolerance. The second is a measure based on the rankings of the predicted ratings.

The rest of the chapter is organized as follows. In Section 3.2 we present a detailed problem formulation. In Section 3.3, we present the ALS algorithm for collaborative filtering and a very brief review of the PNCG algorithm. In Section 3.4, we compare the performance of the ALS and PNCG algorithms. In Section 3.5, we conclude.

3.2 Problem Description

The optimization approach we present in this chapter is in principle applicable to any collaborative filtering model that uses ALS. For definitiveness, we choose a specific latent factor model, the matrix factorization model from [59] and [105], and implement our approach for this model. Given the data we use, we present the model in terms of users and movies instead of the more generic users and items framework.

Let the matrix of user-movie rankings be represented by $\mathbf{R} = \{r_{ij}\}_{n_u \times n_m}$ where r_{ij} is the rating given to movie j by user i, n_u is the number of users and n_m is the number of items. Note that for any user i and movie j the value of r_{ij} is either a real number or is missing, where a vast number of values are usually missing. For example, the MovieLens 20M dataset [75] with 138,493 users and 27,278 movies contains only 20 million rankings, which accounts for less than 1% of the total possible rankings. In the low-rank factorization of $\mathbf{R}$, each movie j is associated with a vector $\mathbf{m}_j \in \mathbb{R}^{n_f}$, $j = 1, \ldots, n_m$ and each user i is associated with a vector $\mathbf{u}_i \in \mathbb{R}^{n_f}$, $i = 1, \ldots, n_u$. The elements of $\mathbf{m}_j$ measure the degree that movie j possesses each factor or feature and $\mathbf{u}_i$ similarly measures the degree that user i possesses each factor or feature. The dot product, $\mathbf{u}_i^T \mathbf{m}_j$ captures the interaction between user i and movie j and is an approximation of user i's rating of movie j which we denote $\hat{r}_{ij}$ (i.e. $\hat{r}_{ij} = \mathbf{u}_i^T \mathbf{m}_j$). Let $\mathbf{U} = [\mathbf{u_i}] \in \mathbb{R}^{n_f \times n_u}$ be the user feature matrix and $\mathbf{M} = [\mathbf{m_j}] \in \mathbb{R}^{n_f \times n_m}$ the movie feature matrix. Our goal is to map the ratings matrix $\mathbf{R}$ to $\mathbf{U}^T \mathbf{M}$. The challenge of computing the matrices $\mathbf{U}$ and $\mathbf{M}$ is accomplished by minimizing the following mean-squared loss function:

$$\begin{aligned} \mathcal{L}_\lambda(\mathbf{R}, \mathbf{U}, \mathbf{M}) = \frac{1}{n} \sum_{(i,j) \in \mathcal{I}} (r_{ij} - \mathbf{u}_i^T \mathbf{m}_j)^2 + \\ \lambda\left(\sum_i n_{u_i} \|\mathbf{u}_i\|^2 + \sum_j n_{m_j} \|\mathbf{m}_j\|^2\right), \end{aligned} \tag{3.1}$$

where $\mathcal{I}$ is the index set of known ratings and has size n, n_{u_i} denotes the number of ratings of user i and n_{m_j} is the number of ratings of movie j. The term $\lambda(\sum_i n_{u_i} \|\mathbf{u}_i\|^2 + \sum_j n_{m_j} \|\mathbf{m}_j\|^2)$ is a Tikhonov regularization [97] term commonly included in the loss function to prevent overfitting [105]. The full optimization problem can be stated as follows,

$$\min_{\mathbf{U}, \mathbf{M}} \mathcal{L}_\lambda(\mathbf{R}, \mathbf{U}, \mathbf{M}). \tag{3.2}$$

There are numerous algorithms available to solve the optimization problem in (3.2) and as mentioned previously ALS and stochastic gradient descent are the most popular. We will propose using the PNCG algorithm with ALS as a preconditioner, and will show that it converges much faster than ALS.

The model represented by the loss function in Equation (3.1) is very simple. Koren, Bell and Volinsky [59] provide an overview of some extensions to the basic model including models that incorporate bias, include implicit feedback and add temporal dynamics. In all of these models, a prediction rule for $\hat{r}_{ij}$ is derived, where $\hat{r}_{ij} = \mathbf{u}_i^T \mathbf{m}_j$ in the model we examine. The objective in each model is to find the unknown parameters in the model and for each model we do so by minimizing the mean squared error between the known ratings and the prediction (possibly including a regularization term in the loss function to prevent overfitting). Thus, our approach has broad applicability, since ALS and our PNCG algorithm can also be used to find the unknown parameters in these more sophisticated models.

3.3 Optimization Algorithms

Before presenting the PNCG algorithm we begin by describing the standard ALS algorithm used to solve the optimization problem described in (3.2).

3.3.1 Alternating Least Squares Algorithm

The optimization problem in (3.2) is not convex. However, if we fix one of the unknowns, either $\mathbf{U}$ or $\mathbf{M}$, then the optimization problem becomes quadratic and we can solve for the remaining unknown by solving the least squares problem. This is the central idea behind ALS. To find a solution to the optimization problem given in (3.2) we start by initializing $\mathbf{M}$ with random values. Then, we can fix $\mathbf{M}$ and solve for $\mathbf{U}$ by minimizing Equation (3.1). The next step is to fix $\mathbf{U}$ and solve for $\mathbf{M}$ by minimizing Equation (3.1). We can repeat these alternating steps until we have reached a specified stopping criteria.

The primary work in the ALS algorithm is in finding the least squares solution to Equation (3.1) for each of the unknowns. Consider the case where $\mathbf{M}$ is fixed and we are seeking the least squares solution to Equation (3.1) for $\mathbf{U}$. We can determine the solution to Equation (3.1) by setting all the elements of the gradient of Equation (3.1) equal to zero. The elements of the gradient of Equation (3.1) are given by the partial derivatives, $\frac{\partial \mathcal{L}_\lambda}{\partial u_{ki}}$, where u_{ki} is the (k,i)th element of $\mathbf{U}$, $k = 1\ldots,n_f$, $i = 1,\ldots,n_u$. Setting all the partial derivatives to zero gives the following solution for $\mathbf{U}$,

$$\begin{aligned}
&\frac{\partial \mathcal{L}_\lambda}{\partial u_{ki}} = 0, \quad \forall\, i,k \\
\Rightarrow\ &\sum_{j\in\mathcal{I}_i} 2(\mathbf{u}_i^T\mathbf{m}_j - r_{ij})m_{kj} + 2\lambda n_{u_i} u_{ki} = 0, \quad \forall\, i,k \\
\Rightarrow\ &\sum_{j\in\mathcal{I}_i} m_{kj}\mathbf{u}_i^T\mathbf{m}_j + \lambda n_{u_i} u_{ki} = \sum_{j\in\mathcal{I}_i} m_{kj} r_{ij}, \quad \forall\, i,k \\
\Rightarrow\ &(\mathbf{M}_{\mathcal{I}_i}\mathbf{M}_{\mathcal{I}_i}^T + \lambda n_{u_i}\mathbf{I})\mathbf{u}_i = \mathbf{M}_{\mathcal{I}_i}\mathbf{R}^T(i,\mathcal{I}_i), \quad \forall\, i \\
\Rightarrow\ &\mathbf{u}_i = \mathbf{A}_i^{-1}\mathbf{V}_i, \quad \forall\, i
\end{aligned} \tag{3.3}$$

where $\mathbf{A}_i = \mathbf{M}_{\mathcal{I}_i}\mathbf{M}_{\mathcal{I}_i}^T + \lambda n_{u_i}\mathbf{I}$, $\mathbf{v}_i = \mathbf{M}_{\mathcal{I}_i}\mathbf{R}^T(i,\mathcal{I}_i)$ and $\mathbf{I}$ is the $n_f \times n_f$ identity matrix. The elements of $\mathbf{M}$ are represented by m_{kj}, $k = 1,\ldots,n_f$, $j = 1,\ldots,n_m$, $\mathcal{I}_i$ is the index set of movies user i has rated, and $\mathbf{M}_{\mathcal{I}_i}$ represents the sub-matrix of $\mathbf{M}$ where columns $j \in \mathcal{I}_i$ are selected. Similarly, $\mathbf{R}(i,\mathcal{I}_i)$ is a row vector that represents the ith row of $\mathbf{R}$ with only the columns in $\mathcal{I}_i$ included. We get a similar solution for the columns of $\mathbf{M}$ when we fix $\mathbf{U}$ and find the least squares solution to Equation (3.1) for $\mathbf{M}$:

$$\mathbf{m}_i = \mathbf{A}_j^{-1}\mathbf{v}_j,\ \forall\, j \tag{3.4}$$

where $\mathbf{A}_j = \mathbf{U}_{\mathcal{I}_j}\mathbf{U}_{\mathcal{I}_j}^T + \lambda n_{m_j}\mathbf{I}$, $\mathbf{v}_j = \mathbf{U}_{\mathcal{I}_j}\mathbf{R}(\mathcal{I}_j, j)$. $\mathcal{I}_j$ is the index set of users that have rated movie j, $\mathbf{U}_{\mathcal{I}_j}$ represents the sub-matrix of $\mathbf{U}$ where columns $i \in \mathcal{I}_j$ are selected, and $\mathbf{R}(\mathcal{I}_j, j)$ is a column vector that represents the jth column of $\mathbf{R}$ with only the rows in $\mathcal{I}_j$ included.
Algorithm 4 summarizes the ALS algorithm used to solve the optimization problem given in (3.2). From Algorithm 4 we note that each of the columns of $\mathbf{U}$, and the columns of $\mathbf{M}$, can be computed independently. This suggests an easy way for the algorithm to be implemented in parallel as is done in [105]. However, the algorithm can still require many iterations to converge and we propose using ALS as a preconditioner in the PNCG algorithm to improve convergence speed.

3.3.2 PNCG Algorithm

The PNCG algorithm we use to find an optimal solution to the problem in (3.2) is given by Algorithm 3 from the previous chapter where $\mathbf{x}^T = \left[\mathbf{u}_1^T\ \mathbf{u}_2^T \ldots \mathbf{u}_{n_u}^T\ \mathbf{m}_1^T\ \mathbf{m}_2^T \ldots \mathbf{m}_{n_m}^T\right] \in \mathbb{R}^{n_f\times(n_u+n_m)}$ and $\overline{\mathbf{x}}_k = P(\mathbf{x}_k)$ denotes one iteration of the ALS algorithm described in Algo-

Algorithm 4: Alternating Least Squares Algorithm (ALS)

```
Initialize $\mathbf{M}$ with random values;
while Stopping criteria have not been satisfied do
    for $i = 1, \ldots, n_u$ do
        $\mathbf{u}_i \leftarrow \mathbf{A}_i^{-1}\mathbf{v}_i$;
    end
    for $i = 1, \ldots, n_m$ do
        $\mathbf{m}_i = \mathbf{A}_j^{-1}\mathbf{v}_j$;
    end
end
```

rithm 4. Note, that we only consider one possibility for $\overline{\beta}_{k+1}$ in our numerical experiments:

$$\widehat{\beta}_{k+1}^{PR} = \frac{\mathbf{g}_{k+1}^T(\overline{\mathbf{g}}_{k+1} - \overline{\mathbf{g}}_k)}{\mathbf{g}_k^T\overline{\mathbf{g}}_k}, \tag{3.5}$$

where we note that if the given expression for $\overline{\beta}_{k+1}$ is used in Algorithm 3 and we applied Algorithm 3 to a convex quadratic problem with an exact line search with P a preconditioning matrix, $\mathbf{P}$, then Algorithm 3 would be equivalent to the preconditioned conjugate gradient algorithm with the same preconditioner.

The primary computational cost in Algorithm 3 comes from computing the ALS iteration, $P(\mathbf{x}_k)$, and from computing α_k, the line search parameter. To compute α_k we use a line search algorithm. An exact line search algorithm would find the α_k that minimizes $f(\mathbf{x}_k + \alpha_k\mathbf{p}_k)$. However, this can require a large number of iterations so most line search algorithms try to efficiently find an α_k that moves the solution towards convergence. These algorithms typically require both function and gradient evaluations to find the best α_k and we can see from the computations in (3.3) that calculating the gradient is less expensive than an ALS iteration since we don't need to find the inverse of a matrix to compute the gradient. Thus, if we don't require a large number of iterations to determine α_k then using the PNCG algorithm with ALS as a preconditioner doesn't increase the computational cost per iteration by too much relative to the ALS algorithm. However, this small additional cost per iteration may be warranted if it leads to a large decrease in the number of iterations required for convergence. As in the previous chapter, we use the Moré-Thuente [74] line search algorithm which is designed to satisfy the strong Wolfe conditions, which consist of a sufficient decrease condition and a curvature condition.

3.4 Numerical Experiments

We analyze the performance of the ALS and PNCG algorithm by examining their performance on the MovieLens [75] data. The entire MovieLens data set has 138,493 users and 27,278 movies and just over 20 million ratings where each user has rated at least 20 movies. Note that there are some movies with no ratings and if we only consider movies with at least one rating then there are 26,744 movies. We will consider subsets of the data and examine how the algorithms perform as we increase the size of the ratings matrix. In creating our subsets we try to exclude outlier users; users with either a large number of movie ratings or a very small number of movie ratings. As an example of how we do this, suppose we want to create a subset of ratings with 100 users and 20 movies. We begin by creating a sorted list of the users based on the number of movies they have rated (descending order). The user in the middle of this list, user c, has the median number of ratings. We then include users $c-50$ to $c+49$ from the sorted list in our sample. Once we have chosen the users, we use the same process to determine the movies based on ratings given by our chosen users.

All numerical experiments were performed on a Linux Workstation with a Quad-Core Intel Xeon 3.16GHz processor and 8GB RAM. For the PNCG algorithm we use the Moré-Thuente line search algorithm from the Poblano toolbox for MATLAB [30]. The line search parameters are as follows: 10^{-4} for the sufficient decrease condition tolerance, 10^{-2} for the curvature condition tolerance, an initial step length of 1 and a maximum of 20 iterations. The gradient norm is used as a stopping criteria for both ALS and PNCG but since the gradient is not used in the calculation of the ALS algorithm, the calculation of the gradient is not included in the timing results for the ALS algorithm. The stopping conditions shared by both ALS and PNCG are as follows: 10^{-6} for the gradient norm divided by the number of variables, $(n_m + n_u) \times n_f$, and 10^4 for the maximum number of iterations. In addition, PNCG has a maximum set on the total number of function evaluations equal to 10^7.

For our tests we consider ratings matrices of 6 different sizes: $n_u \times n_m = 100 \times 20$, 200×40, 400×80, 800×160, 1600×320 and 3200×640, where n_u is the number of users and n_m is the number of movies. For each ratings matrix, ALS and PNCG are used to solve the optimization problem in (3.2) with $\lambda = 0.1$ and $n_f = 10$ and the algorithms are run using 20 different random starting values until one of the stopping criteria is reached. Note that for each ratings matrix, the 20 starting values are the same for ALS and PNCG. In Table 3.1 we summarize the timing results for the different problem sizes. The time is written in the form $a \pm b$ where a is the mean time in seconds and b is the standard deviation. Runs that do not converge based on the gradient norm tolerance are not included in the mean and standard deviation calculations; the only run that did not converge to the specified gradient norm tolerance before reaching the maximum number of iterations was a 1600×320 ALS run. In the third column of Table 3.1 we calculate the acceleration factor of PNCG. The value is simply the mean time for convergence of ALS divided by the mean time for convergence of PNCG. We see from this table that PNCG significantly improves the convergence speed of the ALS algorithm for all problem sizes. Similarly, Table 3.2 summarizes the number of iterations required to reach convergence for each algorithm. Again, the results are calculated based on converged runs only. In the third column of Table 3.2 we calculate the acceleration factor of PNCG based on iteration count and we see that PNCG also significantly improves convergence in terms of iteration count.

Table 3.1: Timing Results of ALS and PNCG for different problem sizes (Mean Time in Seconds).

Problem Size	Time (Seconds)		Acceleration
$n_u \times n_m$	ALS	PNCG	Factor
100×20	8.32 ± 3.87	1.95 ± 0.59	4.27
200×40	19.80 ± 10.25	4.52 ± 1.30	4.38
400×80	56.50 ± 38.06	12.22 ± 4.25	4.62
800×160	162.0 ± 89.94	47.57 ± 20.02	3.41
1600×320	330.61 ± 120.3	116.2 ± 31.56	2.84
3200×640	960.8 ± 364.0	303.7 ± 111.3	3.16

Table 3.2: Iteration Results of ALS and PNCG for different problem sizes.

Problem Size	Iterations		Acceleration
$n_u \times n_m$	ALS	PNCG	Factor
100×20	1326 ± 622.4	105.7 ± 29.4	12.54
200×40	1574 ± 814.1	125.2 ± 35.3	12.58
400×80	2181 ± 1466	158.8 ± 54.3	12.74
800×160	3048 ± 1689	290.4 ± 128.1	10.50
1600×320	3014 ± 1098	302.9 ± 86.3	9.95
3200×640	4231 ± 1602	329.6 ± 127.5	12.84

From Tables 3.1 and 3.2 it is clear that PNCG accelerates the convergence of the ALS algorithm, using the gradient norm as the measure of convergence. However, since we are also interested in using the factor matrices, $\mathbf{U}$ and $\mathbf{M}$, generated by each algorithm, to make recommendations we would also like to examine the convergence of the algorithms using these recommendations. Rather than examine the full recommendations themselves (i.e. $\hat{\mathbf{R}} = \mathbf{U}^T\mathbf{M}$), we consider the rankings. In particular, we are interested in the rankings of the top t movies (e.g. top 20 movies) for each user. We want to explore how these rankings change as we increase the number of iterations of both ALS and PNCG. If the rankings of the top t movies for each user do not change after a small number of iterations then we can say that this algorithm computes an accurate solution, for the purposes of providing accurate recommendations, quickly. Thus, predicted user movie rankings can provide us with an additional measure by which to determine how quickly ALS and PNCG converge. Our ranking accuracy metric will be an average across users. For each user, we will use a modified version of the Kendall-Tau [52] distance to determine how close two different movie rankings are. The Kendall-Tau distance computes the difference between two ranking vectors based on the entire vector and we modify the formula to instead compute the difference between ranking vectors based only on the rankings of the top t items. Suppose we want to compare two different movie rankings for a single user. Let $\mathbf{p}_1$ and $\mathbf{p}_2$ be two different rankings of movies for user i. Our ranking measure is based on computing the number of inversions or switches required to turn $\mathbf{p}_2$ into $\mathbf{p}_1$ but only for the top t movies. We begin by finding the top t movies in $\mathbf{p}_1$. Then for each of those movies we determine what rank they have in $\mathbf{p}_2$ and see how many inversions are required in $\mathbf{p}_2$ so that movie has the same rank in $\mathbf{p}_1$. Consider the following example, where $\mathbf{p}_1 = [3, 4, 2, 5, 6, 1]$, and $\mathbf{p}_2 = [6, 3, 1, 2, 4, 5]$. So in $\mathbf{p}_1$, user i ranks movie 6 highest, $\mathbf{p}_1(6) = 1$, and in $\mathbf{p}_2$, user i ranks movie 3 highest, $\mathbf{p}_2(3) = 1$. Suppose we are only interested in the rankings of the top 2 movies. To compute our ranking metric we first need to find the top ranked movie in $\mathbf{p}_1$ which we've already stated is movie 6. However, in $\mathbf{p}_2$, movie 6 is ranked 5th. We need to compute the number of inversions required to move the rank of movie 6 from 5th to 1st which in this case is 4. We then need to find the 2nd highest ranked movie in $\mathbf{p}_1$ which is movie 3. In $\mathbf{p}_2$ movie 3 is ranked 1st. We need to compute the number of inversions required to move the rank of movie 3 from 1st to 2nd. However, in our first step we moved movie 6 from 5th to 1st using inversions so $\mathbf{p}_2$ is actually now $\tilde{\mathbf{p}}_2 = [6, 4, 2, 3, 5, 1]$ and movie 3 is already in 2nd so we don't need to make any inversions. So the total number of switches required to match $\mathbf{p}_2$ to $\mathbf{p}_1$ for the top 2 rankings is $s = 4$. We want to normalize our ranking metric to have a value of 1 if no inversions are needed (i.e. $\mathbf{p}_1 = \mathbf{p}_2$ in the top t spots) and 0 if the maximum number of inversions are needed. To calculate the maximum number of inversions we note that this occurs if $\mathbf{p}_2$ is in the exact opposite order from $\mathbf{p}_1$. In this case, $n_m - 1$ inversions are needed in the first step, $n_m - 2$ inversions in the second step and so on until we need to make $n_m - t$ inversions in the t-th step. Thus the maximum number of inversions, is $s_{max} = (n_m - 1) + (n_m - 2) + \ldots + (n_m - t) = \frac{t}{2}(2n_m - t - 1)$ and our ranking accuracy metric for a given user i is

$$q_i = 1 - \frac{s}{s_{max}}, \tag{3.6}$$

and in our example $q_i = 1 - \frac{4}{9} = \frac{5}{9}$. To calculate the total ranking accuracy metric we take the average value of q_i across all users.

We can use this ranking metric to evaluate the accuracy of ALS and PNCG as a function of the number of iterations and runtime. Suppose we take the final solution of a given algorithm, where convergence to a final solution is determined by the gradient norm, and then use the factor matrices to generate predictions, $\hat{\mathbf{R}} = \mathbf{U}^T\mathbf{M}$. Then based on these predictions we can also generate a final ranking vector for each user, $\overline{\mathbf{p}}_i,\ i = 1, \ldots, n_u$. We can also consider the predictions generated at every iteration k of the algorithm, $\hat{\mathbf{R}}_k = \mathbf{U}_k^T\mathbf{M}_k$ and generate a ranking vector, $\mathbf{p}_{i,k}$ for each user. We can then use $\mathbf{p}_{i,k}$ and $\overline{\mathbf{p}}_i$ to compute our ranking metric. Note that $\overline{\mathbf{p}}_i$ is computed with high accuracy but can potentially be different for each optimization method. In our numerical experiments, $\overline{\mathbf{p}}_i$ is the ranking vector generated when the normalized gradient norm is less than or equal to 10^{-6} for each method. This gives us a measure of how close to convergence each iteration is in terms of our desired ranking. If the ranking metric for iteration k is x then we say that at iteration k the algorithm has reached $x\%$ ranking accuracy.

Tables 3.3, 3.4, 3.5 and 3.6 summarize the time needed for each algorithm to reach a certain level of ranking accuracy for ratings matrices with different sizes. Note that in all 4 cases we are comparing the top 20 movie rankings for each user. Once again we use 20 different random starting values so the time is written in the form $a \pm b$ where a is the mean time in seconds and b is the standard deviation. To better understand the tables, consider Table 3.3. The first entry of the first column displays the time needed for ALS to get a ranking accuracy of 70% or a ranking metric equal to 0.7 (i.e. the time needed for ALS to compute k iterations where k is the first iteration where the average q_i across users is greater than or equal to 0.7). In general, we can see that PNCG reaches a certain level of accuracy in less time than ALS. In the third column of Tables 3.3, 3.4, 3.5 and 3.6 we have computed the acceleration factor of PNCG. We note that at 70% accuracy this value is less than 1 for all problem sizes which implies that ALS is faster at reaching 70% accuracy for all problem sizes. However, if we want a more accurate solution then PNCG is much faster than ALS. We can also see this from Figure 3.1 which shows the average time needed for both algorithms to reach a given ranking accuracy for the 400×80 ratings matrix. We see that both ALS and PNCG reach a ranking accuracy of approximately 75% very quickly. However, it then takes ALS a long time to increase the ranking accuracy from 75% to 100% while the time need for PNCG to increase the ranking accuracy from

75% to 100% is much smaller. For example, PNCG reaches 90% ranking accuracy in about 7 seconds whereas standalone ALS needs about 37 seconds.

Table 3.3: Ranking Accuracy Timing Results. Problem Size: 100×20

Ranking Accuracy	Time (Seconds)		Acceleration Factor
	ALS	PNCG	
70%	0.17 ± 0.09	0.18 ± 0.04	0.93
80%	1.60 ± 1.70	0.59 ± 0.40	2.71
90%	4.83 ± 3.05	1.19 ± 0.72	4.05
100%	15.18 ± 7.44	2.24 ± 0.72	6.78

Table 3.4: Ranking Accuracy Timing Results. Problem Size: 200×40

Ranking Accuracy	Time (Seconds)		Acceleration Factor
	ALS	PNCG	
70%	0.27 ± 0.11	0.38 ± 0.12	0.72
80%	4.72 ± 8.07	1.36 ± 0.87	3.47
90%	12.49 ± 13.06	2.62 ± 1.30	4.77
100%	34.67 ± 18.02	5.07 ± 1.57	6.83

Table 3.5: Ranking Accuracy Timing Results. Problem Size: 400×80

Ranking Accuracy	Time (Seconds)		Acceleration Factor
	ALS	PNCG	
70%	0.37 ± 0.17	0.65 ± 0.16	0.58
80%	7.88 ± 7.03	2.87 ± 1.72	2.74
90%	37.08 ± 37.62	6.92 ± 4.47	5.36
100%	101.29 ± 68.04	14.07 ± 5.25	7.20

3.5 Conclusion

In this chapter, we have demonstrated how the PNCG algorithm with ALS as a nonlinear preconditioner can significantly improve the convergence speed of ALS-based collaborative

Table 3.6: Ranking Accuracy Timing Results. Problem Size: 800 × 160

Ranking Accuracy	Time (Seconds)		Acceleration Factor
	ALS	PNCG	
70%	0.76 ± 0.34	1.43 ± 0.35	0.53
80%	19.83 ± 13.67	12.61 ± 10.13	1.57
90%	103.91 ± 70.97	33.25 ± 22.34	3.12
100%	310.98 ± 168.67	59.88 ± 28.20	5.19

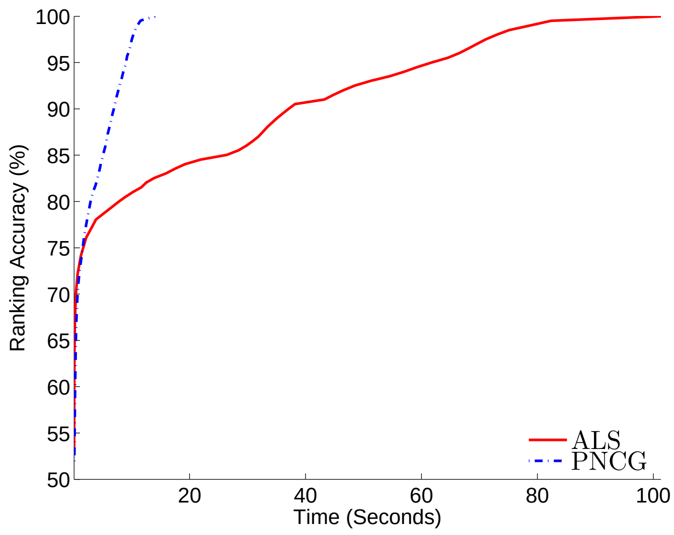


Figure 3.1: Average ranking accuracy versus time for problem size: 400 × 80

filtering algorithms. We showed this in the context of a simple latent factor model, but our acceleration approach can be used with any collaborative filtering model that uses ALS. We expect that our acceleration approach will be especially useful for advanced ALS-based collaborative filtering methods that achieve low Root Mean Square Error (RMSE), since these methods require solving the optimization problem accurately, and that is precisely where accelerated PNCG shows the most benefit over standalone ALS. For example, it would be straightforward to extend our approach to the ALS-based implicit feedback algorithm of [49].

Chapter 4

An In-Depth Analysis of the Chung-Lu Model

4.1 Introduction

In very simple terms a network can be defined as a collection of points joined together by lines. Thus, networks can be used to represent connections between entities in a wide variety of fields including engineering, science, medicine, and sociology. Many large real-world networks share a surprising number of properties, leading to model development research whereby formal models are developed to describe real-world networks. These formal models often have associated algorithms which allow us to generate instances of these models to compare with real networks. However, in other cases, an algorithm or procedure will be developed to generate a synthetic graph, a graph designed to match many of the properties of a real-world network, where the underlying model may not be explicitly defined. These algorithms or generative processes are often referred to as graph generators. Modeling real-world networks either with a formal model or a generative process serves two purposes. First, building models that mimic the patterns and properties of real networks helps to understand the implications of these patterns and helps determine which patterns are important. If we develop a generative process to synthesize networks we can also examine which growth processes are plausible and which are not. Secondly, high-quality, large-scale network data is often not available, because of economic, legal, technological, or other obstacles [19]. Thus, there are many instances where the systems of interest cannot be represented by a single real-world network. As one example, consider the field of cybersecurity, where systems require testing across diverse threat scenarios and validation across diverse network structures. In these cases, where there is no single exemplar network, the systems must instead be modeled as a collection of networks in which the variation among them may be just as important as their common features. By modeling these networks either using a formal model or a practical graph generator, we can create synthetic networks that capture both the essential features of a system and the realistic variability. Then we can use such synthetic graphs to perform tasks such as simulations, analysis, and decision making. For example, we can use these synthetic networks to examine how diseases propagate through a population. We can also use synthetic graphs to test the performance of graph analysis algorithms, including clustering algorithms and anomaly detection algorithms.

One of the most well-known formal graph models is the Erdős Rényi random graph model [33]. In the classic Erdős Rényi random graph model each edge is chosen with uniform probability and the degree distribution is binomial, limiting the type of graphs that can be modeled using the Erdős Rényi framework [80, p. 424]. The Chung-Lu model [4, 22, 23] is an extension of the Erdős Rényi model that allows for more general degree distributions. The probability of each edge is no longer uniform and is a function of a user-supplied degree sequence, $k = (k_1, \ldots, k_n)$, where k_i is the user-supplied degree of node i and n is the number of nodes in the graph. By design the user-supplied degree sequence is equal to the expected degree sequence of the model (i.e $E(\mathbf{D}_i) = k_i$ where $\mathbf{D}_i$ is the degree of node i in the model). This property makes it an easy model to work with theoretically and since the Chung-Lu model is a special case of a random graph model with a given degree sequence, many of its properties are well known and have been studied extensively [22, 23, 104, 71, 73]. It is also an attractive model for many real-world networks, particularly those with power-law degree distributions and it is sometimes used as a benchmark for comparison with other graph models despite some of its limitations [94, 86]. We know for example, that the average clustering coefficient is too low relative to most real world networks. As well, measures of degree affinity are also too low relative to most real-world networks of interest. However, despite these limitations or perhaps because of them, the Chung-Lu model is often used as a basis for comparing new graph models and generators.

To compare the Chung-Lu model to other graph models or graph generators we often use instances of the models or generators we are interested in. However, the standard algorithm used to generate instances of the Chung-Lu model has a runtime that is $\mathcal{O}(n^2)$ where n is the number of nodes in the original graph. Thus, if we are dealing with a very large graph this algorithm can be prohibitively expensive. Miller and Hagberg [67] introduce an algorithm with expected runtime that is $\mathcal{O}(n + m)$ where m is the expected number of edges of the model. They argue that for graphs with finite average degree $n = \mathcal{O}(m)$ and the expected runtime is $\mathcal{O}(n)$. As part of their Block Two-Level Erdős Rényi graph generator, Kolda et al. [94, 58] introduce a Fast Chung-Lu algorithm with an expected runtime of $\mathcal{O}(m)$ where m, in this case, is the number of edges in the original graph. They claim that their Fast Chung-Lu algorithm generates instances of the Chung-Lu model. In [58], the authors do note that in their Fast Chung-Lu algorithm, duplicate edges and self-edges may be created, but that they simply discard these edges, stating that in their experiments the practical impact was small. In this chapter, we are interested in examining the impact of discarding these edges. What we discover is that by removing these duplicate edges and self-edges the Fast Chung-Lu algorithm does not actually generate instances of the Chung-Lu model, but instead generates instances of a different model, one that is an approximation to the true Chung-Lu model. We will present this new model, which we refer to as either the $\mathcal{O}(|Edges|)$ or $\mathcal{O}(m)$ Chung-Lu model, and examine the size of the approximation error and determine how different factors can affect it. Pfeiffer et al. [85] also note the bias in networks generated using the Fast Chung-Lu algorithm however we provide a much more in-depth analysis of this approximation bias. In particular, since the original Chung-Lu model allows for self-edges we are interested in examining the approximation error as a result of removing self-edges separately from the impact of removing duplicate edges. Table 4.1 summarizes our initial results where the Bernoulli Chung-Lu (BCL) model refers to the original Chung-Lu model, for reasons that will become apparent in the next section, and the $\mathcal{O}(m)$ Chung-Lu (MCL) model describes the model that the Fast Chung-Lu algorithm generates instances of, the approximation to the BCL model. As shown in Table 4.1 models with and without the possibility of generating self-edges are treated separately which results in a total of 4 different models to compare. For each model, we compare the occurrence probability of an edge with two distinct endpoint nodes and the occurrence probability of a self-edge where the endpoint nodes are the same. We also compare the expected degree. As we mentioned previously, in the BCL model with self-edges, the model originally presented by Chung and Lu, the expected degree, $E(\mathbf{D}_i)$, of each node i is equal to the input degree, k_i, of node i. This property does not hold for each of the subsequent models we explore so we present in Table 4.1 the formula for the expected degree, which we derive in detail later in this chapter.

Table 4.1 summarizes the properties of 4 different models and in the following sections we present these models and derive the probabilities of edges and the expected degree. Note that in all of our discussions we will only be concerned with undirected, connected graphs although much of the analysis does not require the graph to be connected. We begin our discussion by presenting the BCL model and we describe the standard $\mathcal{O}(n^2)$ algorithm that can be used to generate instances of this model. We calculate the expected degree for each node and using the degree sequence from a real-world network we compute instances of the model to demonstrate numerically that for each node, the degree average approaches the expected degree as the number of instances grows, as is predicted by the law of large numbers. In Section 4.3, we present the MCL model, a new model and the model that the Fast Chung-Lu algorithm from [94, 58] actually generates instances of. Using the degree sequence from a real-world network we demonstrate numerically the differences between the MCL model and the BCL model. In Section 4.4 we explore how removing self-edges from both the BCL and MCL models leads to new models and impacts the properties of these model as compared to the BCL and MCL models with self-edges.

As we will see in Sections 4.2 and 4.4, one of the assumptions made in both BCL models is that $k_i^2 \leq 2m \; \forall \; i$, where m is the number of edges in the graph. However, there are many real-world graphs for which this condition does not hold. In Section 4.5 we explore how the properties of both the BCL and the MCL models change when this constraint is violated. In the case of the BCL models, the models actually have to be extended which leads to two additional BCL models. Finally, in Section 4.6 we focus on another network property that is often important when modeling networks, the triangle count and the related clustering coefficient. We calculate the expected triangle count for the various Chung-Lu models we have presented and show how these values are different from the values found in real networks.

4.2 Bernoulli Chung-Lu Model

In the model originally presented by Chung and Lu, as previously mentioned, each edge is chosen with a different probability. As such, each edge can be represented as a random variable and we can formally define the probability distribution for each edge. Edges between two distinct nodes can be represented as Bernoulli random variables and this is why we refer to this model as the Bernoulli Chung-Lu model. Written in terms of the graph adjacency matrix, we have

$$\mathbf{A}_{ij} \sim \text{Bernoulli}(p_{ij}), \;\; 1 \leq i < j \leq n, \; \mathbf{A}_{ji} = \mathbf{A}_{ij}. \tag{4.1}$$

For self-edges, the probability distribution is more general although it is very similar to the Bernoulli distribution. Written in terms of the adjacency matrix, self-edges, $\mathbf{A}_{ii}$, $i = 1, \ldots, n$, can be defined as random variables with support, $S(\mathbf{A}_{ii}) = \{0, 2\}$, and probability mass function given by the following

$$P(\mathbf{A}_{ii} = a_{ii}) = \begin{cases} 1 - p_{ii} & a_{ii} = 0, \\ p_{ii} & a_{ii} = 2, \end{cases} \tag{4.2}$$

		Probability of Edge (i,j)	Probability of Edge (i,i)	Expected Degree $E(\mathbf{D}_i)$
Bernoulli Chung-Lu	Self-Edges: Model I	$\frac{k_i k_j}{2m}$	$\frac{k_i^2}{4m}$	k_i
	No Self-Edges: Model III	$\frac{k_i k_j}{2m}$	0	$k_i - \frac{k_i^2}{2m}$
$\mathcal{O}(m)$ Chung-Lu	Self-Edges: Model II	$1 - \left(1 - 2\frac{k_i k_j}{4m^2}\right)^m < \frac{k_i k_j}{2m}$	$1 - \left(1 - \frac{k_i^2}{4m^2}\right)^m < \frac{k_i^2}{4m}$	$\sum_{j \neq i} 1 - \left(1 - 2\frac{k_i k_j}{4m^2}\right)^m + 1 - \left(1 - \frac{k_i^2}{4m^2}\right)^m < k_i$
	No Self-Edges: Model IV	$1 - \left(1 - 2\frac{k_i k_j}{4m^2}\right)^m < \frac{k_i k_j}{2m}$	0	$\sum_{j \neq i} 1 - \left(1 - 2\frac{k_i k_j}{4m^2}\right)^m < k_i - \frac{k_i^2}{2m}$

Table 4.1: Model Summary

where $i = 1, \ldots, n$. Why is the support of each self-edge equal to $\{0, 2\}$ and not $\{0, 1\}$? Essentially, this is to ensure that all edges are counted equally. In general, if there is an edge in an undirected graph between two distinct nodes i and j then both a_{ij} and a_{ji} are 1. If edges are to be counted equally, then self-edges should also appear in the adjacency matrix twice but since there is only one diagonal matrix element, a_{ii}, both appearances need to be recorded in the same element.

We have not yet defined the probabilities, $p_{ij},\ i, j = 1, \ldots, n$, in the probability distributions, (4.1) and (4.2). In the BCL model they have a very specific form and are based on the only input into the BCL model, a sequence of degrees, $k = (k_1, \ldots, k_n)$, where k_i is the degree of node i and n is the number of nodes. Often, this degree sequence is taken from a real network we are interested in modeling. In the BCL model, $p_{ij},\ i, j = 1, \ldots, n$, has the following form

$$p_{ij} = \begin{cases} \frac{k_i k_j}{2m} & i \neq j, \\ \frac{k_i^2}{4m} & i = j, \end{cases} \tag{4.3}$$

where $m = \frac{1}{2} \sum_i k_i$ is the number of edges in the graph and we assume $k_i^2 \leq 2m\ \forall\ i$ which is a sufficient condition to ensure that $p_{ij} \leq 1\ \forall\ i, j$.

One of the central properties of the BCL model is that the expected degree sequence is actually given by the input degree sequence, k. In other words, the expected degree of node i is equal to k_i. To show this we note that in the model, the degree of node i is a random variable $\mathbf{D}_i$, $i = 1, \ldots, n$, given by

$$\mathbf{D}_i = \sum_j \mathbf{A}_{ij},\ i = 1, \ldots, n. \tag{4.4}$$

We have

$$E(\mathbf{D}_i) = E\left(\sum_j \mathbf{A}_{ij}\right) = \sum_j E(\mathbf{A}_{ij}). \tag{4.5}$$

For edges with distinct node endpoints, $\mathbf{A}_{ij}$ is a Bernoulli random variable and thus its expected value is given by

$$E(\mathbf{A}_{ij}) = p_{ij} = \frac{k_i k_j}{2m},\ i \neq j. \tag{4.6}$$

For self-edges we can use the probability mass function given in (4.2) to determine the expected value

$$\begin{aligned} E(\mathbf{A}_{ii}) &= \sum a_{ii} P(\mathbf{A}_{ii} = a_{ii}) \\ &= 0 \cdot P(\mathbf{A}_{ii} = 0) + 2 \cdot P(\mathbf{A}_{ii} = 2) \\ &= 2p_{ii} = \frac{k_i^2}{2m}. \end{aligned} \tag{4.7}$$

Thus, the expected degree of node i is given by

$$\begin{aligned} E(\mathbf{D}_i) &= \sum_j E(\mathbf{A}_{ij}) = \sum_{j \neq i} E(\mathbf{A}_{ij}) + E(\mathbf{A}_{ii}) \\ &= \sum_{j \neq i} \frac{k_i k_j}{2m} + \frac{k_i^2}{2m} \\ &= \sum_j \frac{k_i k_j}{2m} = \frac{k_i}{2m} \sum_j k_j = k_i \frac{2m}{2m} \\ &= k_i. \end{aligned} \tag{4.8}$$

This is the central property of the BCL model.

In practice, model properties are often examined by using an algorithm to generate instances of the model. We use Algorithm 5 to generate instances of the BCL model. Of course, in any given instance of the BCL model the node degree will not match the expected degree; however, by the law of large numbers, as we increase the number of instances the average degree for each node should approach the expected degree for each node. We explore the validity of our algorithm by ensuring that this theory holds using a small real network. Consider the arXiv general relativity collaboration network [62] which is an author collaboration network based on data from the general relativity section of the arXiv preprint server and has 4158 nodes and 13422 edges. The largest degree node has 81 neighbors. Thus, $k_i^2 \leq 2m \ \forall \ i$, since the maximum value of k_i^2 is 6561 and $2m$ is equal to 26844. If we use Algorithm 5 to generate a collection of instances of the BCL model then the average degree for each node should approach the expected degree as the collection size increases. Given that we will present several different versions of the BCL model throughout this thesis, we need to provide different names to each version. We refer to the BCL model presented in this section as both the BCL model and Model I. We will also often refer to this model as the original Chung-Lu model since it was the model originally proposed by Chung and Lu [22, 23]. Using the degree sequence from the general relativity collaboration network to define the probabilities, p_{ij}, $i, j = 1, \ldots, n$ in Equation (4.3) we use Algorithm 5 to generate 10000 instances of the Chung-Lu model. Figure 4.1

plots the difference between the expected degree of each node and the average degree of each node where we have ordered the nodes from smallest degree to largest degree based on their degree size in the original graph. The expected degree of each node is of course equal to the actual degree of each node in the original graph as shown by Equation (4.8). The average degree of each node is calculated using l instances of the Chung-Lu model where we vary l from 10 to 10000. We can see from the plots that as we increase the number of instances in our average degree calculation the difference between the average and the expected degree decreases where the largest difference between the two values is seen in large degree nodes. This indicates that Algorithm 5 does in fact generate instances of the BCL model.

Additionally, we can also compare the expected number of edges in the graph with the average number of edges, calculated from l instances of the model. Like the degree of node i, $\mathbf{D}_i$, we can define the random variable $\mathbf{M}$ as the number of edges in the model where $\mathbf{M}$ is defined as

$$\mathbf{M} = \frac{1}{2}\sum_i \mathbf{D}_i = \frac{1}{2}\sum_i \sum_j \mathbf{A}_{ij} \tag{4.9}$$

Then the expected value of $\mathbf{M}$ can be calculated as follows

$$\begin{aligned} E(\mathbf{M}) &= \frac{1}{2}\sum_i E(\mathbf{D}_i) \\ &= \frac{1}{2}\sum_i k_i \\ &= \frac{1}{2}(2m) \\ &= m. \end{aligned} \tag{4.10}$$

From our numerical experiments with $l = 10000$ the average number of edges is 13421.12. Compare this to 13422 which is the expected number of edges in the graph (and the actual number of edges in the original graph). Table 4.2 lists the average number of edges for

$l = 10, 100, 1000, 5000$ and 10000.

Algorithm 5: Bernoulli Chung-Lu Algorithm

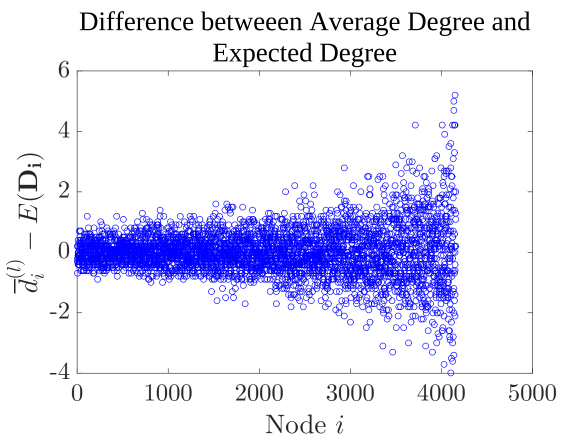
```
for $i = 1$ to $n$ do
    for $j = i$ to $n$ do
        Draw a random number from the uniform distribution on [0,1];
        if The random number is less than or equal to $p_{ij}$ from Equation (4.3) then
            /* Add edge $(i, j)$ to the graph */
            if $i \neq j$ then
                $a_{ij} = a_{ji} = 1$;
            else
                $a_{ii} = 2$;
            end
        end
    end
end
```

Table 4.2: Average Number of Edges for l Instances of the BCL Model.

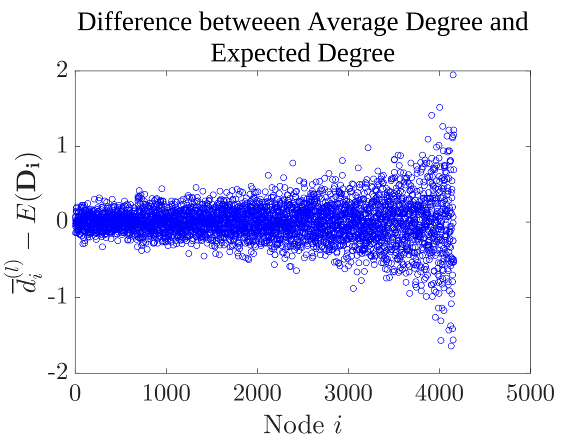
Number of Instances (l)	10	100	1000	5000	10000
Average Number of Edges	13442.5	13420.31	13421.65	13421.21	13421.12

4.3 $\mathcal{O}(|Edges|)$ Chung-Lu Model

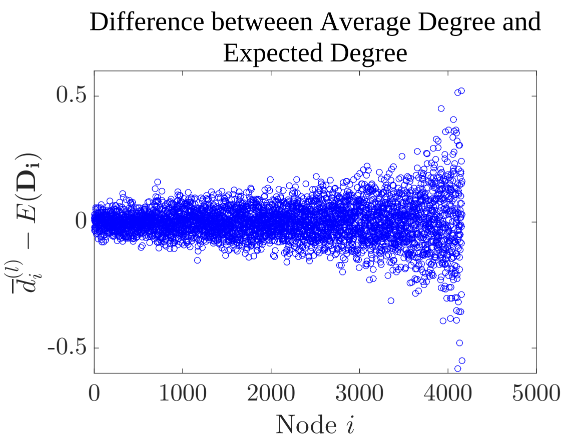
One of the issues with the BCL model and in particular the algorithm used to generate instances of the BCL model, Algorithm 5, is that Algorithm 5 requires $\mathcal{O}(n^2)$ operations. We must visit every edge and decide with probability p_{ij} if the edge should be included in the graph. This algorithm can become prohibitively expensive as the size of the graphs we consider grows. However, we can devise an algorithm that generates instances of a model that requires only $\mathcal{O}(m)$ operations where $m = \frac{1}{2}\sum_i k_i$ is the number of edges in the graph and where the probability that an edge will appear in the graph is approximately the same in this model as in the BCL model. This algorithm is very similar to the Fast Chung-Lu algorithm presented in [94, 58]; however we allow for self-edges. Before we present this



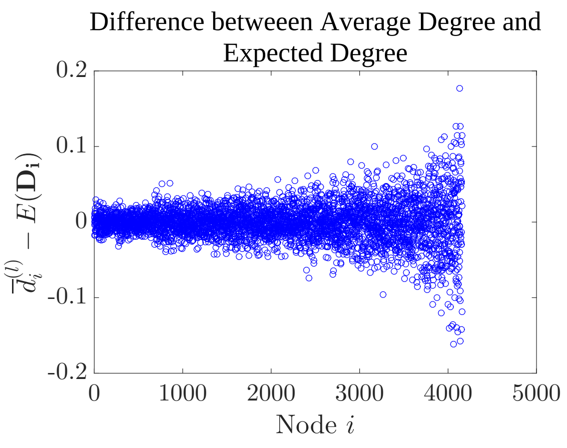
(a) $l = 10$



(b) $l = 100$



(c) $l = 1000$



(d) $l = 10000$

Figure 4.1: General Relativity Collaboration Network: Comparing Average Degree and Expected Degree from Model I.

algorithm we begin by presenting a generative process which will allow us to determine the exact probabilities of the edges in the model.

Suppose we place all the elements of the adjacency matrix a_{ij}, $i, j = 1, \ldots, n$ in a bag. We are no longer treating the elements as random variables so we write the elements as a_{ij} instead of $\mathbf{A}_{ij}$. In this section, we will assume that the elements representing self-edges,

a_{ii}, $i = 1, \ldots, n$, are in the bag. We create our graph by drawing m elements of the adjacency matrix from the bag. After each draw we return the elements to the bag so that we allow elements to be drawn more than once. If the element a_{ij} is drawn, a_{ij} and a_{ji} are both set equal to 1. If an element has already been drawn the values of a_{ij} and a_{ji} remain equal to 1. If the element a_{ii} is drawn then $a_{ii} = 2$ and this value doesn't change if a_{ii} is drawn subsequently. We begin by considering the case where $i \neq j$. We want to know what the probability of an edge between vertices i and j is after m draws. In other words, we want to know the probability of having an edge between vertices i and j is in our graph. This is not simply equal to the probability of drawing a_{ij} or a_{ji} on a single draw. To calculate the probability of having an edge between vertices i and j in the graph we note that an edge between nodes i and j appears in the graph because either a_{ij} or a_{ji} is drawn at least once. More formally, let us define three mutually exclusive and exhaustive events that can occur on a single draw, $\mathbf{E}_1 = \{a_{ij} \text{ is drawn}\}$, $\mathbf{E}_2 = \{a_{ji} \text{ is drawn}\}$, and $\mathbf{E}_3 = \{\text{Another element of the adjacency matrix is drawn}\}$. Let p_{ij} be the probability that event $\mathbf{E}_1$ happens, p_{ji} be the probability that event $\mathbf{E}_2$ happens and $1 - p_{ij} - p_{ji}$ be the probability that event $\mathbf{E}_3$ happens. On m independent draws let $\mathbf{X}_i$ be the number of occurrences of event $\mathbf{E}_i$, $i = 1, 2, 3$. Then the vector $\mathbf{X} = (\mathbf{X}_1, \mathbf{X}_2)$ has a multinomial distribution with joint pdf given by

$$f(x_1, x_2) = \frac{m!}{x_1! x_2! x_3!} p_1^{x_1} p_2^{x_2} p_3^{x_3} = \frac{m!}{x_1! x_2! x_3!} p_{ij}^{x_1} p_{ji}^{x_2} (1 - p_{ij} - p_{ji})^{x_3}, \tag{4.11}$$

where $x_3 = m - x_1 - x_2$ and $p_3 = 1 - p_1 - p_2$. In fact, we could define the vector $\widehat{\mathbf{X}} = (\mathbf{X}_{11}, \mathbf{X}_{12}, \ldots, \mathbf{X}_{ij}, \ldots, \mathbf{X}_{n(n-1)})$ where $\mathbf{X}_{ij}$ is the number of occurrences of event $\mathbf{E}_{ij} = \{a_{ij} \text{ is drawn}\}$, $i, j = 1, \ldots, n$, on m independent draws. Then $\widehat{\mathbf{X}}$ has a multinomial distribution with joint pdf given by

$$f(x_{11}, x_{12}, \ldots, x_{ij}, \ldots, x_{n(n-1)}) = \frac{m!}{x_{11}! x_{12}! \cdots x_{ij}! \cdots x_{n(n-1)}! x_{nn}!} p_{11}^{x_{11}} p_{12}^{x_{12}} \cdots p_{ij}^{x_{ij}} \cdots p_{n(n-1)}^{x_{n(n-1)}} p_{nn}^{x_{nn}}, \tag{4.12}$$

where $\sum_i \sum_j p_{ij} = 1$ and $x_{nn} = m - \sum_{(i,j) \neq (n,n)} x_{ij}$. The vector $\mathbf{X}$ is simply a condensed version of $\widehat{\mathbf{X}}$ where we have simply condensed the $n^2 - 2$ events where we draw an adjacency matrix element that is not a_{ij} or a_{ji} into one event. As noted $\sum_i \sum_j p_{ij} = 1$. Returning to our original goal of calculating the probability of an edge between nodes i and j in the graph, we can use the joint pdf in (4.11) to calculate this value. We want to know the probability that either $\mathbf{X}_1$ or $\mathbf{X}_2$ is at least 1. This is equivalent to one minus the probability that both are zero. Written formally we have

$$\begin{aligned} p_{ij}(m) &:= P(\text{There is an edge between nodes } i \text{ and } j \text{ in the graph after } m \text{ draws}) \\ &= 1 - f(\mathbf{X}_1 = 0, \mathbf{X}_2 = 0) \\ &= 1 - (1 - p_{ij} - p_{ji})^m, \; i \neq j. \end{aligned} \tag{4.13}$$

If we assume that $p_{ij} = p_{ji}$ we can use the binomial theorem to simplify the formula for $p_{ij}(m)$:

$$(1 - 2p_{ij})^m = \sum_{k=0}^{m} \binom{m}{k} (-1)^k (2p_{ij})^k. \tag{4.14}$$

This give us

$$\begin{aligned} p_{ij}(m) &= 1 - (1 - 2p_{ij})^m \\ &= 1 - \sum_{k=0}^{m} \binom{m}{k} (-1)^k (2p_{ij})^k \\ &= \sum_{k=1}^{m} \binom{m}{k} (-1)^{k+1} (2p_{ij})^k \\ &= 2mp_{ij} + \mathcal{O}(p_{ij}^2), \; i \neq j. \end{aligned} \tag{4.15}$$

Returning to the case where $i = j$, we note that a self-edge appears in the graph at node i if A_{ii} is drawn at least once. So to calculate the probability of a self-edge appearing in the graph, we define two mutually exclusive and exhaustive events that can occur on a single draw, $\mathbf{E} = \{a_{ii} \text{ is drawn}\}$, and $\overline{\mathbf{E}} = \{\text{Another element of the adjacency matrix is drawn}\}$. Let p_{ii} be the probability that event $\mathbf{E}$ happens, and $1 - p_{ii}$ be the probability that event $\overline{\mathbf{E}}$ happens. On m independent draws let $\mathbf{X}$ be the number of occurrences of event $\mathbf{E}$. Then the vector $\mathbf{X}$ has a binomial distribution with probability distribution function given by

$$f(x) = \binom{m}{x} p_{ii}^x (1 - p_{ii})^{m-x} \tag{4.16}$$

We want to know the probability that the value of $\mathbf{X}$ is greater than or equal to 1. This is equal to $1 - P(\mathbf{X} = 0)$. Using the binomial distribution in Equation (4.16) we get

$$\begin{aligned} p_{ii}(m) &:= P(\text{There is a self-edge at node } i \text{ in the graph after } m \text{ draws}) \\ &= 1 - f(\mathbf{X} = 0) \\ &= 1 - (1 - p_{ii})^m. \end{aligned} \tag{4.17}$$

Using the binomial theorem to simplify we get

$$\begin{aligned} p_{ii}(m) &= 1-(1-p_{ii})^m \\ &= 1-\sum_{k=0}^{m}\binom{m}{k}(-1)^k(p_{ii})^k \\ &= \sum_{k=1}^{m}\binom{m}{k}(-1)^{k+1}(p_{ii})^k \\ &= mp_{ii}+\mathcal{O}(p_{ii}^2). \end{aligned} \tag{4.18}$$

We can combine the two cases to get the probability of edge (i, j) being in the graph after m draws as

$$p_{ij}(m) = \begin{cases} 1-(1-2p_{ij})^m \approx 2mp_{ij} & i \neq j, \\ 1-(1-p_{ij})^m \approx mp_{ii} & i = j, \end{cases} \tag{4.19}$$

where we have assumed that $p_{ij} = p_{ji}$. Suppose we let $p_{ij} = \frac{k_i k_j}{(2m)^2}$ $\forall\, i, j$ so that $\sum_i \sum_j p_{ij} = 1$, then

$$p_{ij}(m) = \begin{cases} 1-\left(1-2\frac{k_i k_j}{(2m)^2}\right)^m = \frac{k_i k_j}{2m}+\mathcal{O}(p_{ij}^2) & i \neq j, \\ 1-\left(1-\frac{k_i^2}{(2m)^2}\right)^m = \frac{k_i^2}{4m}+\mathcal{O}(p_{ii}^2) & i = j. \end{cases} \tag{4.20}$$

If we ignore the higher order terms then we get

$$p_{ij}(m) \approx \begin{cases} \frac{k_i k_j}{2m} & i \neq j, \\ \frac{k_i^2}{4m} & i = j. \end{cases} \tag{4.21}$$

Thus, the probability of having an edge between nodes i and j in the graph is approximately equal to the value given by Equation (4.3), the probability of choosing an edge between nodes i and j in the BCL model. Our new model is defined by Equation (4.20) and the following algorithm outlines the above generative process necessary to generate instances of our new model.

Algorithm 6: Approximate Chung-Lu Algorithm

```
for $k = 1$ to $m$ do
    Draw element $a_{ij}$ with probability $\frac{k_i k_j}{(2m)^2}$;
    /* Add edge $(i, j)$ to the graph */
    if $i \neq j$ then
        $a_{ij} = a_{ji} = 1$;
    else
        $a_{ii} = 2$;
    end
end
```

The way that Algorithm 6 is written requires us to perform $\mathcal{O}(n^2)$ calculations to determine the probabilities of drawing each edge on a single draw (i.e. the probability of drawing a_{ij}), so in fact this algorithm is still $\mathcal{O}(n^2)$. However, suppose on each draw, instead of drawing the edges themselves we independently draw the nodes of each edge. So, on a single draw, we draw node i with probability $\frac{k_i}{2m}$ and we draw node j with probability $\frac{k_j}{2m}$. Since the two events are independent, the probability of drawing the ordered pair of nodes (i, j), or oriented edge (i, j), is equal to $\frac{k_i k_j}{(2m)^2}$. This only requires us to calculate n probabilities and since $m \geq n - 1$ in any connected graph, our algorithm is now $\mathcal{O}(m)$. This algorithm is described by Algorithm 7. Both Algorithms 6 and 7 generate instances of the same model; however, Algorithm 7 has a much faster run-time. Since Algorithm 7 is $\mathcal{O}(m)$ we refer to the underlying model as the $\mathcal{O}(m)$ Chung-Lu (MCL) model, $\mathcal{O}(|\text{Edges}|)$ Chung-Lu model or as Model II.

Algorithm 7: $\mathcal{O}(|\text{Edges}|)$ Chung-Lu Algorithm

```
for $k = 1$ to $m$ do
    Draw node $i$ with probability $\frac{k_i}{2m}$;
    Draw node $j$ with probability $\frac{k_j}{2m}$;
    /* Add edge $(i, j)$ to the graph */
    if $i \neq j$ then
        $a_{ij} = a_{ji} = 1$;
    else
        $a_{ii} = 2$;
    end
end
```

Note that we have defined $p_{ij}(m)$ in the MCL model as the probability of edge (i, j) being in the graph. Equivalently, $p_{ij}(m)$ can be defined as the probability of choosing edge (i, j) to be in the graph. Given this interpretation, we can then represent each edge as a random variable like we did in the BCL model. Edges can again be represented as elements of the adjacency matrix where edges between distinct nodes follow a Bernoulli distribution as in (4.1) and self-edges are random variables with the probability mass function given by (4.2). The difference between the BCL model and the MCL model is in the probability p_{ij}. In the BCL model p_{ij} is defined in Equation (4.3) and in the MCL model p_{ij} is defined as $p_{ij}(m)$, Equation (4.20). In other words, we could use Algorithm 5 with $p_{ij}(m)$ given by the exact values in Equation (4.20) in place of p_{ij} from Equation (4.3) and this would be generate instances of the MCL model. Algorithm 5 with $p_{ij} = p_{ij}(m)$ from the exact values in Equation (4.20) is equivalent to Algorithms 7 and 6. We can use this equivalence to calculate the expected degree of each node in the MCL model. As noted in the discussion of the BCL model, since the adjacency matrix elements are random variables so too is the degree of each node and the expected degree of each node. For the MCL model, $E(\mathbf{A}_{ij}) = p_{ij}(m)$ if $i \neq j$ and $E(\mathbf{A}_{ii}) = 2p_{ii}(m)$. If we use the approximation in Equation (4.21) for $p_{ij}(m)$ then we find $E(\mathbf{D}_i) = k_i$ as in (4.8). However if we include the higher order terms in $p_{ij}(m)$ we get the following:

$$\begin{aligned} E(\mathbf{D}_i) &= E\left(\sum_j \mathbf{A}_{ij}\right) = \sum_j E(\mathbf{A}_{ij}) \\ &= \sum_{j\neq i} p_{ij}(m) + 2p_{ii}(m) \\ &= \sum_{j\neq i} 1 - (1-2p_{ij})^m + 2(1-(1-p_{ii})^m) \\ &= n + 1 - \sum_{j\neq i}\left(1 - \frac{k_i k_j}{2m^2}\right)^m - 2\left(1 - \frac{k_i^2}{4m^2}\right)^m, \end{aligned} \tag{4.22}$$

which should be approximately equal to k_i. We should note however that the expected degree in Equation (4.22) will always be less than k_i. To see this, we examine the probabilities in Equation (4.20):

$$p_{ij}(m) = \begin{cases} 1 - \left(1 - 2\frac{k_i k_j}{(2m)^2}\right)^m = \frac{k_i k_j}{2m} + \sum_{l=2}^m \binom{m}{l}(-1)^{l+1}(\frac{k_i k_j}{2m^2})^l & i \neq j, \\ 1 - \left(1 - \frac{k_i^2}{(2m)^2}\right)^m = \frac{k_i^2}{4m} + \sum_{l=2}^m \binom{m}{l}(-1)^{l+1}(\frac{k_i^2}{4m^2})^l & i = j. \end{cases} \tag{4.23}$$

In Appendix A we prove that $0 < \frac{k_i k_j}{2m^2} < 1$ and then we can use Bernoulli's Inequality

[95, Theorem 5.1] to show that $1-\left(1-2\frac{k_i k_j}{(2m)^2}\right)^m < 1-\left(1-m\frac{k_i k_j}{2m^2}\right) = \frac{k_i k_j}{2m}$, thus $p_{ij}(m)$ is strictly less than $\frac{k_i k_j}{2m}$ for $i \neq j$. For self-edges, we also have that the maximum value for $p_{ii}(m)$ is strictly less than $\frac{k_i^2}{4m}$ for $i \neq j$. Given that $E(\mathbf{A}_{ij}) = p_{ij}(m)$ for $i \neq j$ and $E(\mathbf{A}_{ii}) = 2p_{ii}(m)$ this implies that $E(\mathbf{D}_i) < k_i$.

We can once again explore the performance of our algorithm using the general relativity collaboration network. Using Algorithm 7 to generate $l = 10000$ instances of the MCL model we can examine the difference between the average degree for each node and the expected degree. Figure 4.2 plots the difference between the average degree and the expected degree, given by Equation (4.22), for $l = 10, 100, 1000$ and 10000. We can see from the figures that as the number of instances of the graph increase the difference between the average and the expected degree decreases, as expected, suggesting our algorithm correctly generates instances of the MCL model. We can also use the general relativity collaboration network to explore the difference between the expected degree given by Equation (4.22) and the expected degree under the BCL model, which is k_i, $\forall\ i$. Figure 4.3 plots the difference between the expected degree and the actual degree. This clearly illustrates that the MCL model may have significant deviations from the target degree distribution for high degree nodes where the error for the largest degree node is approximately 3%.

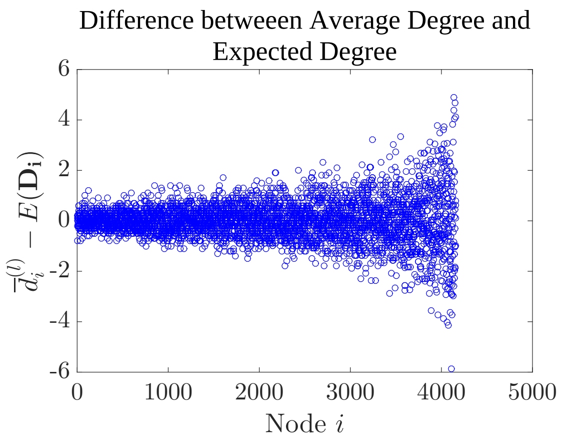
4.4 Excluding Self-Edges

4.4.1 Bernoulli Chung-Lu Model

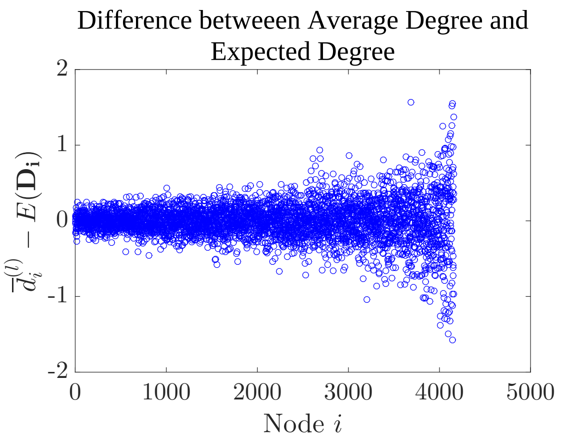
Most of the real-world networks we are interested in studying do not have self-edges. So we would like to build models that do not allow for self-edges. This often involves taking existing models and modifying them to exclude the possiblity of self-edges. To do this in the BCL is fairly straight forward. We still represent each edge as a random variable, where distinct node edges have a Bernoulli distribution and self-edges have the probability mass function given in (4.2). However, to remove the possibility of self-edges we simply set the probability of choosing a self-edge to 0. More formally, we re-define the probabilities in (4.3) as

$$p_{ij} = \begin{cases} \frac{k_i k_j}{2m} & i \neq j, \\ 0 & i = j, \end{cases} \tag{4.24}$$

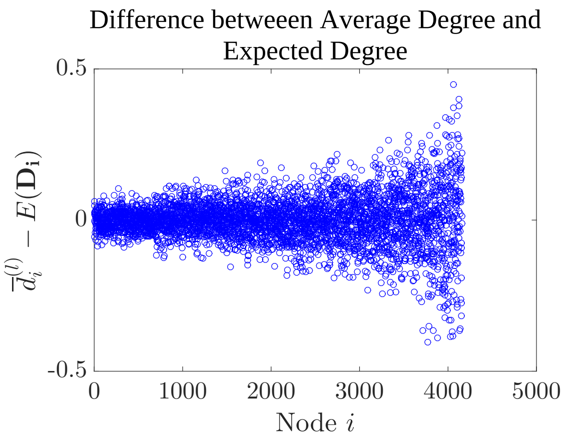
where we still assume $k_i^2 \leq 2m\ \forall\ i$ to ensure $p_{ij} \leq 1\ \forall\ i, j$. This is, of course, a sufficient but not necessary condition to ensure that the probabilities are well defined. Note that by



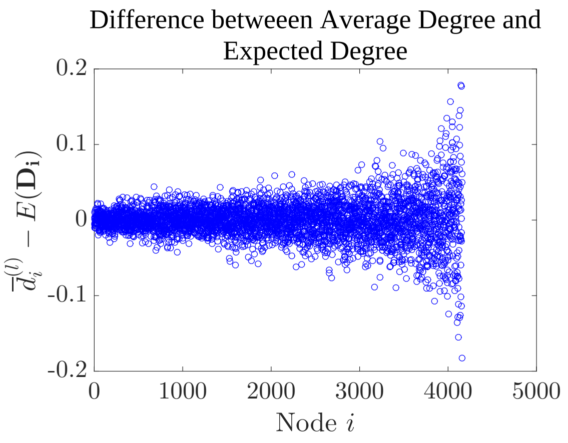
(a) $l = 10$



(b) $l = 100$



(c) $l = 1000$



(d) $l = 10000$

Figure 4.2: General Relativity Collaboration Network: Comparing Average Degree and Expected Degree from Model II.

removing self-edges from the graph we no longer have the nice property that $E(\mathbf{D}_i) = k_i$.

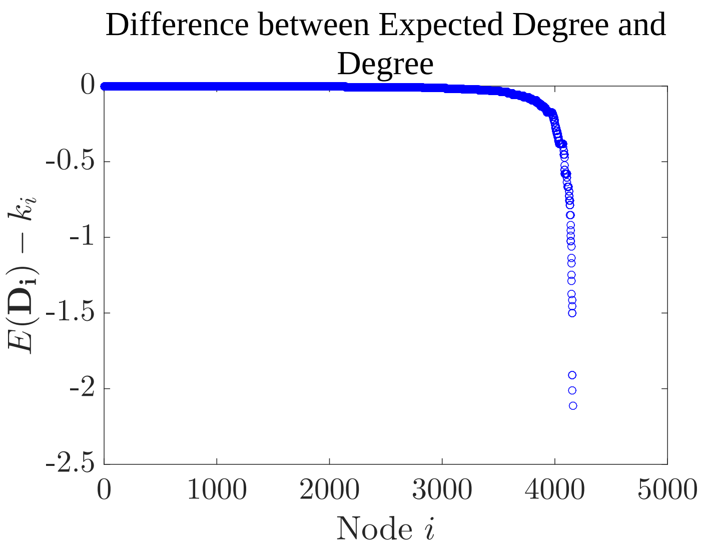


Figure 4.3: General Relativity Collaboration Network: Comparing Expected Degree from Model II and Actual Degree.

The expected degree is now given by the following

$$\begin{aligned} E(\mathbf{D}_i) &= E\left(\sum_j \mathbf{A}_{ij}\right) = \sum_j E(\mathbf{A}_{ij}) = \sum_{j\neq i} \frac{k_i k_j}{2m} \\ &= \sum_{j\neq i} \frac{k_i k_j}{2m} = \frac{k_i}{2m} \sum_{j\neq i} k_j = k_i \frac{2m - k_i}{2m} \\ &= k_i - \frac{k_i^2}{2m}, \end{aligned} \tag{4.25}$$

where $E(\mathbf{A}_{ii}) = 0 \cdot 1 + 2 \cdot 0 = 0,\ i = 1, \ldots, n$. We have assumed that $\frac{k_i^2}{2m} \leq 1$ which means that in most cases $E(\mathbf{D}_i) \approx k_i$. We use the general relativity collaboration network to illustrate the difference between the BCL model (with self-edges) and BCL without self-edges which we also refer to as Model III. Figure 4.4 plots the difference between the expected degree, Equation (4.25) and the degree in the original graph. Note that the difference is equal to $E(\mathbf{D}_i) - k_i = -\frac{k_i^2}{2m}$. From Figure 4.4 we can see that all the values are less than one in magnitude as expected.

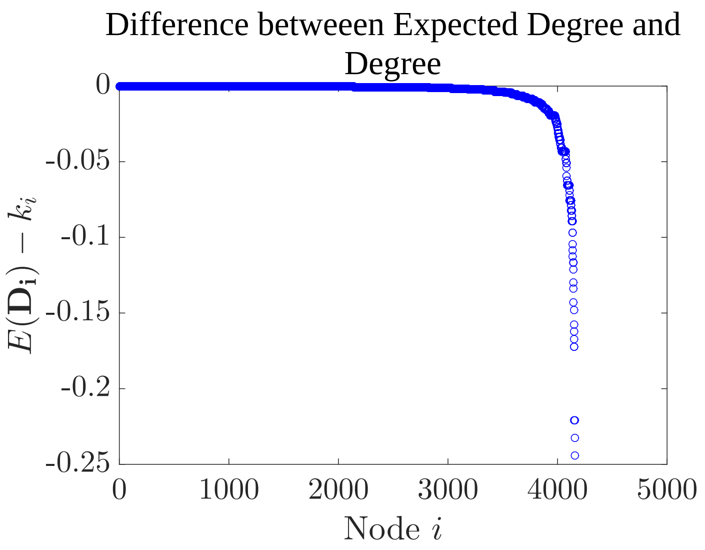


Figure 4.4: General Relativity Collaboration Network: Comparing Expected Degree from Model III and Actual Degree.

4.4.2 $\mathcal{O}(|Edges|)$ Chung-Lu Model

As we previously mentioned, the algorithm to generate instances of the BCL model is computationally expensive. Even if we remove self-edges from the model the algorithm is still $\mathcal{O}(n^2)$. In the case with self-edges, the MCL model proved an attractive alternative since it is an approximation to the BCL model but the algorithm used to generate instances of the model is much less computationally expensive. We would like to find a similar algorithm for the case without self-edges. To do so we return to the generative process we used to build the MCL model. Suppose we drew elements of the adjacency matrix from a bag that doesn't include the self-edges (i.e. We remove the a_{ii} elements to begin with and only draw the a_{ij} elements). This would modify p_{ij}, the probability of drawing edge (i, j) on a single draw, since $\sum_{i,j} p_{ij}$ must equal 1 and now $p_{ii} = 0 \; \forall \; i$. This would of course modify $p_{ij}(m)$, the probability of edge (i, j) being in the graph after m draws. However, what enables us to turn the generative process in the MCL model into a $\mathcal{O}(m)$ algorithm is the ability to draw the nodes of an edge independently as opposed to drawing the edge itself. This is what makes the algorithm $\mathcal{O}(m)$ as opposed to $\mathcal{O}(n^2)$. Once we require that self-edges be excluded then we can no longer draw edge nodes independently since once we draw the first node the probability of drawing the second node changes (i.e. the probability of drawing node i if it has already been drawn is 0). So this rules out the idea of building an $\mathcal{O}(m)$ algorithm by drawing edges from a bag where the self-edges are simply not included. However, a much simpler alternative exists. Suppose that on a given draw, the probability of drawing an element of the adjacency matrix remains the same but we exclude self-edges by simply doing nothing if $i = j$ (i.e. a_{ii} remains equal to 0). So in this case, the probability on a single draw of drawing element a_{ij} is still given by the following

$$p_{ij} = \begin{cases} \frac{k_i k_j}{4m^2} & i \neq j, \\ \frac{k_i^2}{4m^2} & i = j, \end{cases} \tag{4.26}$$

but the probability of having edge (i, j) in the graph after m draws is now given by

$$p_{ij}(m) = \begin{cases} \frac{k_i k_j}{2m} + \mathcal{O}(p_{ij}^2) & i \neq j, \\ 0 & i = j. \end{cases} \tag{4.27}$$

So $p_{ij}(m)$ remains the same for $i \neq j$, while by design $p_{ii}(m)$ now equals 0 for all i. This model is equivalent to the BCL model with the probabilities in Equation (4.3) replaced with the probabilities given in Equation (4.27) and we refer to this model as both the MCL model without self-edges and Model IV. The algorithm that generates instances of this model is given by Algorithm 8. This is the algorithm referred to as the Fast Chung-Lu model in [94, 58] and from the above analysis we note that this algorithm does not generate instances of the BCL model with self-edges.

Algorithm 8: $\mathcal{O}(|\text{Edges}|)$ Chung-Lu Algorithm without Self-Edges.

```
for $k = 1$ to $m$ do
    Draw node $i$ with probability $\frac{k_i}{2m}$;
    Draw node $j$ with probability $\frac{k_j}{2m}$;
    /* Add edge $(i, j)$ to the graph */
    if $i \neq j$ then
        $a_{ij} = a_{ji} = 1$;
    end
end
```

Of course, by not including self-edges in the model we have also changed the expected degree for all nodes. If we ignore the higher order terms in Equation (4.27) then the expected degree is given by

$$\begin{aligned} E(\mathbf{D}_i) &= E(\sum_j \mathbf{A}_{ij}) = \sum_j E(\mathbf{A}_{ij}) = \sum_{j\neq i} E(\mathbf{A}_{ij}) \\ &= \sum_{j\neq i} p_{ij}(m) = \sum_{j\neq i} \frac{k_i k_j}{2m} \\ &= k_i - \frac{k_i^2}{2m}, \end{aligned} \tag{4.28}$$

which is the same as in Model III, the BCL model without self-edges. However, if we don't ignore the higher order terms in $p_{ij}(m)$ then the expected degree is calculated as follows

$$\begin{aligned} E(\mathbf{D}_i) &= \sum_{j\neq i} p_{ij}(m) = \sum_{j\neq i} \left(1 - \left(1 - 2\frac{k_i k_j}{(2m)^2}\right)^m\right) \\ &= \sum_{j\neq i} \left(1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^m\right), \end{aligned} \tag{4.29}$$

where $E(\mathbf{D}_i) < k_i - \frac{k_i^2}{2m}$ since we showed in Section 4.3 that $1 - \left(1 - 2\frac{k_i k_j}{(2m)^2}\right)^m < \frac{k_i k_j}{2m}$. We return to the general relativity collaboration network to examine how these model properties differ in a real network. We note that there are no self-edges in the original network. Figure 4.5 plots the difference between the expected degree in Model IV and the upper bound on the expected degree, Equation (4.28), which is also the expected degree in Model III. Figure 4.6 plots the difference between the expected degree in Model IV and the actual degree. For the largest degree node this difference is approximately 3%. We also note that the expected number of edges in Model IV for the general relativity network is $\sum_n E(\mathbf{D}_i) = 13333.55$ which is approximately 100 edges fewer than in the actual graph, which is $m = 13422$, and where the expected number of edges in Model III is 13413.01.

4.4.3 Summary

In the previous sections, we presented the 4 models along with their properties that we that we outlined in Table 4.1. We also used a real-world network to examine the approximation error present in the BCL model without self-edges, and the MCL models both with and without self-edges. We next look at what happens to these models when the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated.

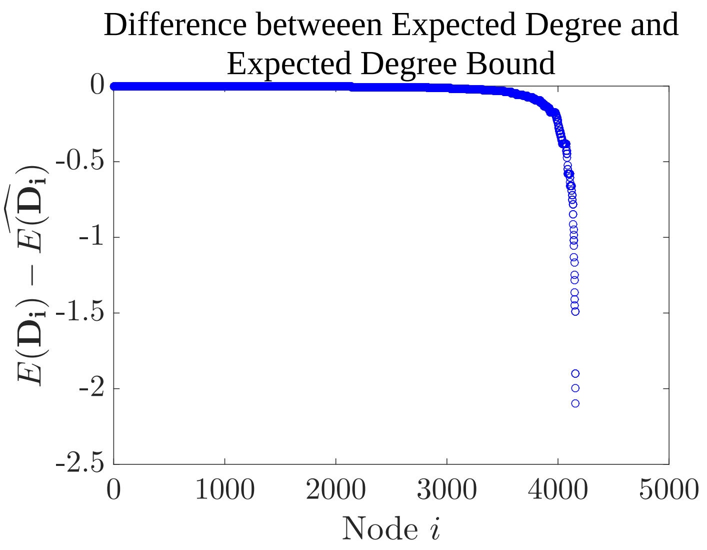


Figure 4.5: General Relativity Collaboration Network: Comparing Expected Degree from Model IV and Expected Degree Upper Bound, the Expected Degree from Model III.

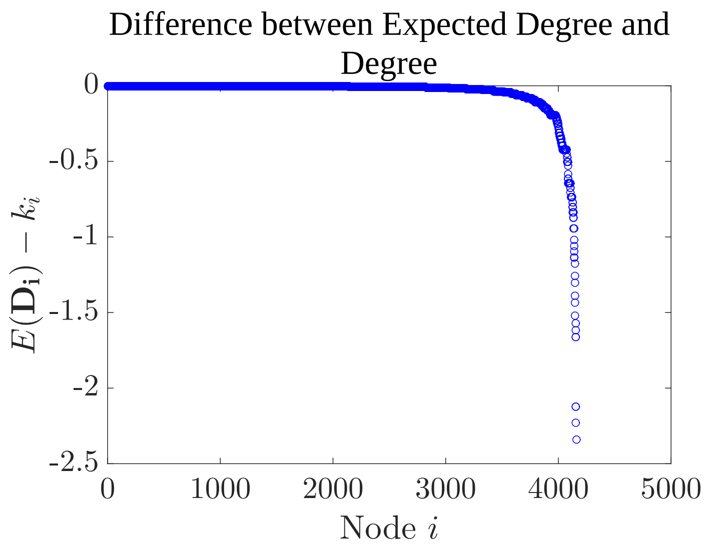


Figure 4.6: General Relativity Collaboration Network: Comparing Expected Degree from Model IV and the Degree, the Expected Degree from Model I.

4.5 Constraint Violation

4.5.1 Bernoulli Chung-Lu Model

Suppose we use as an input into either the Bernoulli Chung-Lu model or the $\mathcal{O}(m)$ Chung-Lu model a degree sequence $k = (k_i, \dots, k_n)$ where the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated. To see why this constraint is important we revisit the edge probability formulas from the original BCL model which are given as follows

$$p_{ij} = \begin{cases} \frac{k_i k_j}{2m} & i \neq j, \\ \frac{k_i^2}{4m} & i = j. \end{cases} \tag{4.30}$$

The constraint is important because it provides a sufficient condition to guarantee that these probabilities are well-defined (i.e. $0 \leq p_{ij} \leq 1 \; \forall \; i, j$). Suppose we have two nodes i and j where $k_i^2 > 2m$ and $k_j^2 > 2m$. This implies that $k_i k_j > 2m$ and $p_{ij} > 1$. In fact, for all pairs of nodes i and j with $k_i^2 > 2m$, and $k_j^2 > 2m$, p_{ij} in (4.30), is greater than 1. In order for the probabilities to be well-defined we need to redefine p_{ij} in (4.30). In the case where the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated the probability of choosing an edge between nodes i and j can be defined by

$$p_{ij} = \begin{cases} \min\left\{\frac{k_i k_j}{2m}, 1\right\} & i \neq j, \\ \min\left\{\frac{k_i^2}{4m}, 1\right\} & i = j. \end{cases} \tag{4.31}$$

If the probabilities are defined as above then $0 \leq p_{ij} \leq 1 \; \forall \; i, j$. If the constraint $k_i^2 \leq 2m \; \forall \; i$ holds then the probabilities reduce to those in Equation (4.30). Thus, Model I, the BCL model with self-edges, is imbedded in this model, which we refer to as Model V. We can calculate the expected degree for Model V as

$$\begin{aligned} E(\mathbf{D}_i) = E\left(\sum_j \mathbf{A}_{ij}\right) &= \sum_j E(\mathbf{A}_{ij}) = \sum_{j \neq i} \min\left\{\frac{k_i k_j}{2m}, 1\right\} + 2 \min\left\{\frac{k_i^2}{4m}, 1\right\} \\ &= \sum_{j \neq i} \min\left\{\frac{k_i k_j}{2m}, 1\right\} + \min\left\{\frac{k_i^2}{2m}, 2\right\}, \end{aligned} \tag{4.32}$$

where $E(\mathbf{D}_i) \leq k_i$ and if the constraint $k_i^2 \leq 2m \; \forall \; i$ holds then $E(\mathbf{D}_i) = k_i$. The difference between k_i and $E(\mathbf{D}_i)$ can be written as

$$k_i - E(\mathbf{D}_i) = \frac{k_i^2}{2m} - \min\left\{\frac{k_i^2}{2m}, 2\right\} + \sum_{j \neq i}\left(\frac{k_i k_j}{2m} - \min\left\{\frac{k_i k_j}{2m}, 1\right\}\right) \tag{4.33}$$

where we use the following to rewrite k_i,

$$k_i = \sum_j \frac{k_i k_j}{2m} = \sum_{j \neq i} \frac{k_i k_j}{2m} + \frac{k_i^2}{2m}. \tag{4.34}$$

From this equation we can see that for $i \neq j$, it is the difference between $\frac{k_i k_j}{2m}$ and 1, when $\frac{k_i k_j}{2m} > 1$, that contributes to the difference between k_i and $E(\mathbf{D}_i)$. For self-edges, it is the difference between $\frac{k_i^2}{2m}$ and 2, when $\frac{k_i^2}{2m} > 2$, that contributes to the difference between k_i and $E(\mathbf{D}_i)$.

To explore this difference empirically we now consider a network that violates the constraint $k_i^2 \leq 2m \; \forall \; i$. The network we will examine is an autonomous system graph which maps the traffic flow across the internet [82]. It has 6474 nodes and 12572 edges. The largest degree node has 1458 neighbors, thus $k_i^2 > 2m$ for at least 1 node and in fact, for this network, k_i^2 is greater than $2m$ for 491 nodes. This implies that there are 120295 probabilities, p_{ij}, $i \neq j$, that need to be redefined and set equal to 1. This amounts to less than 1% of all probabilities, p_{ij}, $i \neq j$. However, these violations can have a noticeable impact on the model and its expected degree. Figures 4.7 and 4.8 compare the expected degree of Model V to k_i and we can see that the error on the largest degree node is approximately 35%.

4.5.2 ($\mathcal{O}(|Edges|)$) Chung-Lu Model

In the MCL model all the probabilities are still well-defined even if the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated. To see this, we use Equation (4.20), which defines the probability of choosing an edge between nodes i and j and which we include here for ease of exposition:

$$p_{ij}(m) = \begin{cases} 1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^m & i \neq j, \\ 1 - \left(1 - \frac{k_i^2}{4m^2}\right)^m & i = j. \end{cases} \tag{4.35}$$

Even if $k_i^2 > 2m$ for more than one node i, it is still true that both $0 < 1 - \frac{k_i k_j}{2m^2} < 1$ and $0 < 1 - \frac{k_i^2}{4m^2} < 1$ since the proof in Appendix A does not assume $k_i^2 \leq 2m$. This implies that $0 \leq p_{ij}(m) \leq 1, \; \forall \; i, j$. This implies that our model does not change and that the expected degree for each node is still well defined and is given by Equation (4.22). However, as noted in Section 4.3, the expected degree in the MCL model can be very different from k_i, the expected degree in the original BCL model, and as we will show this difference is

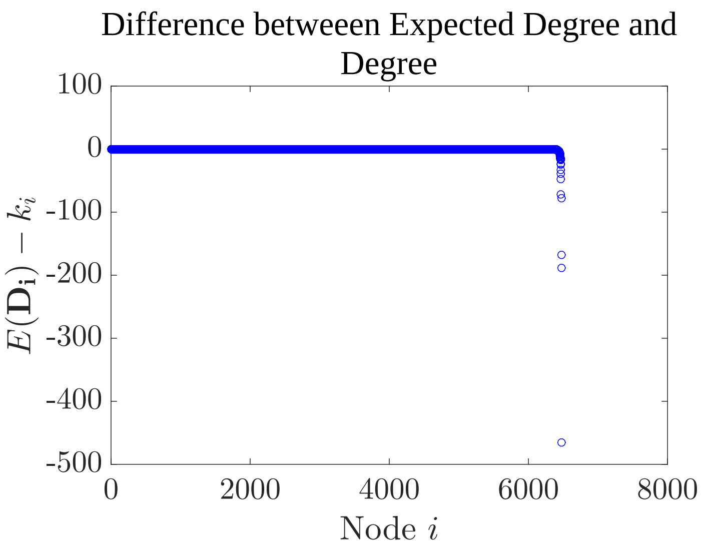


Figure 4.7: Autonomous System Network (Constraint Violated): Comparing Expected Degree from Model V and Actual Degree.

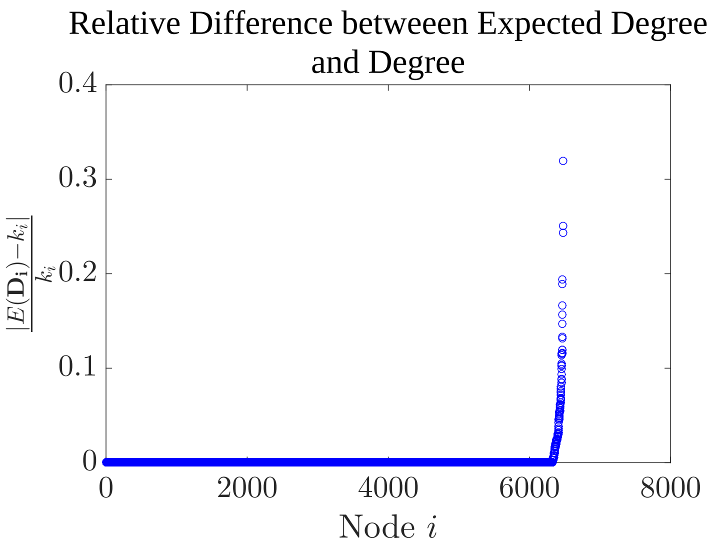


Figure 4.8: Autonomous System Network (Constraint Violated): Relative difference between Expected Degree from Model V and Actual Degree.

exacerbated when the constraint is violated. As we noted previously, the maximum value of $p_{ij}(m)$ is $\frac{k_ik_j}{2m}$ for $i \neq j$ and $\frac{k_i^2}{4m}$ for self-edges. However, when the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated, and $\frac{k_ik_j}{2m} > 1$ for some $i \neq j$, since $p_{ij}(m)$ is still a well-defined probability, we know that 1 is in fact a lower upper bound. The same is true if $\frac{k_i^2}{4m} > 1$ for some node i; $p_{ii}(m)$ still has an upper bound of 1. Using this knowledge we can actually calculate a tighter upper bound on the expected degree for Model II when the degree constraint is violated. Recall from Section 4.3 that the upper bound on $E(\mathbf{D}_i)$ was k_i when $k_i^2 \leq 2m \; \forall \; i$ (see Table 4.1). To calculate our new upper bound, we note that for nodes i and j where $\frac{k_ik_j}{2m} \leq 1$, the upper bound on $p_{ij}(m)$ is still $\frac{k_ik_j}{2m}$, and if $\frac{k_ik_j}{2m} > 1$ then the upper bound on $p_{ij}(m)$ is 1. For self-edges, if $\frac{k_i^2}{4m} \leq 1$, then the upper bound on $p_{ii}(m)$ is still $\frac{k_i^2}{4m}$, and if $\frac{k_i^2}{4m} > 1$ then the upper bound on $p_{ii}(m)$ is 1. Combining this information we get the upper bound on the expected degree of node i for Model II as

$$\widehat{E(\mathbf{D}_i)} = \min\left\{\frac{k_i^2}{2m}, 2\right\} + \sum_{j \neq i} \min\left\{\frac{k_ik_j}{2m}, 1\right\} \tag{4.36}$$

which is equal to the expected degree, Equation (4.32), from Model V and if the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated, $\widehat{E(\mathbf{D}_i)} < k_i$. Also note that if the constraint is not violated then the maximum difference between $p_{ij}(m)$ and $\frac{k_ik_j}{2m}$ is 1 since both are between 0 and 1. However, if $\frac{k_ik_j}{2m} > 1$ the difference can be much larger than 1 which implies that the difference between the expected degree of node i in the MCL model, Model II, and k_i, can be much larger when the constraint is violated than when the constraint holds. Empirically, we can study this model using a real-world network to generate instances of the model. We use the degree sequence from the same autonomous system graph that was used with Model V. Figure 4.9 plots the difference between the expected degree and the actual degree, which is the expected degree in the original BCL model, Model I. Figures 4.10 and 4.11 break this difference down by plotting the difference between the expected degree and the upper bound on the expected degree and the difference between the upper bound on the expected degree and the actual degree, respectively. We can see for some large degree nodes this difference can be quite large. In fact, for the largest degree node which has degree 1458, this difference is approximately 600. This is also reflected in the number of edges. The expected number of edges for this graph can be calculated from the expected degree of each node and is equal to 11445.185225. The upper bound on the expected number of edges can be calculated as

$$\widehat{E(\mathbf{M})} = \frac{1}{2}\sum_i \widehat{E(\mathbf{D}_i)}, \tag{4.37}$$

where $\widehat{E(\mathbf{D}_i)}$ is given by Equation (4.36). For the autonomous system graph, $\widehat{E(\mathbf{M})} =$ 11836.836939. Note that the total number of edges in the original autonomous system graph is 12572. Thus there are approximately 1000 fewer edges expected in the graph than are in the original graph which is approximately 9% of the total number of edges. This is a large deviation, of which people may not be aware and the goal of our work is to quantify and illustrate these deviations.

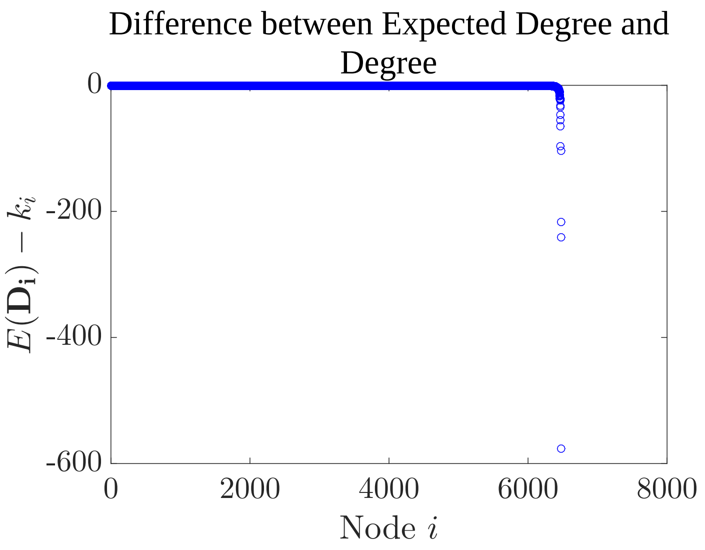


Figure 4.9: Autonomous System Network (Constraint Violated): Comparing Expected Degree from Model II and Actual Degree.

4.5.3 Removing Self-Edges

Sections 4.5.1 and 4.5.2 analyze what happens to Models I and II when the constraint $k_i^2 \leq 2m \ \forall \ i$ is violated, respectively. We are still left to analyze Models III and IV (no self-edges) in the case of the degree constraint violation. Model III is simply Model I without self-edges, thus our analysis of what happens to Model III when the degree constraint is violated will be similar to our analysis in Section 4.5.1. Similarly, Model IV is simply Model II without self-edges and thus our analysis of what happens to Model IV when the degree constraint is violated will be similar to our analysis from Section 4.5.2. In Model III, the BCL model without self-edges, the probability of choosing edge (i, j) is

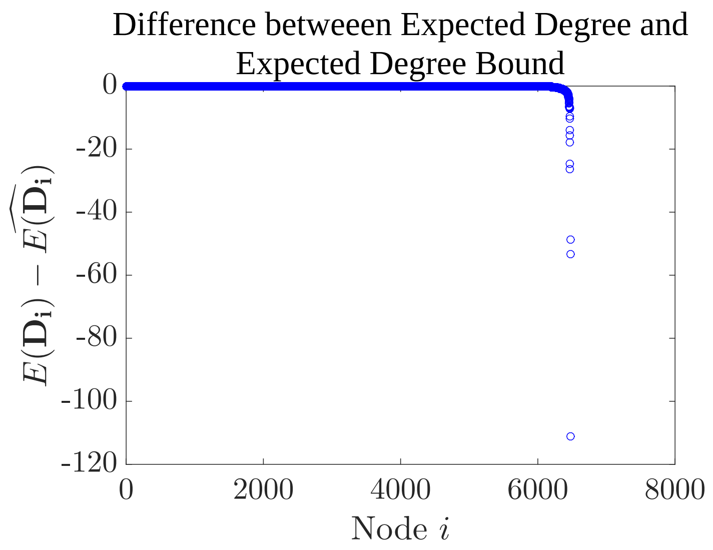


Figure 4.10: Autonomous System Network (Constraint Violated): Comparing Expected Degree and Expected Degree Bound from Model II.

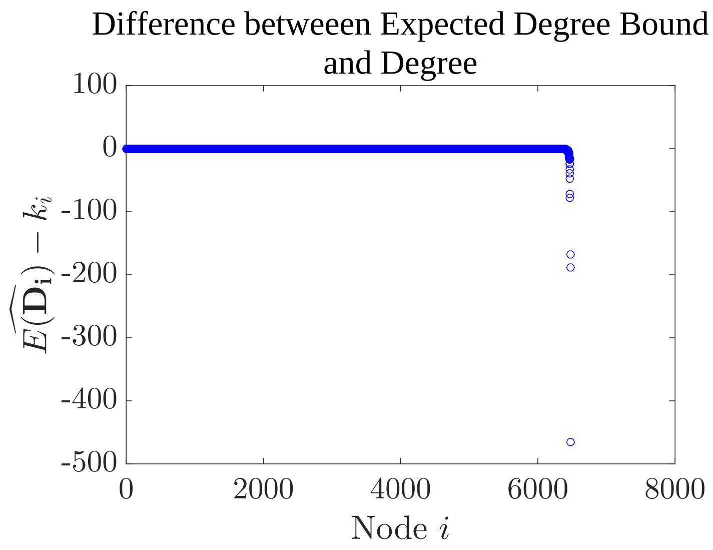


Figure 4.11: Autonomous System Network (Constraint Violated): Comparing Expected Degree Bound from Model II and Actual Degree.

given by

$$p_{ij} = \begin{cases} \frac{k_i k_j}{2m} & i \neq j, \\ 0 & i = j. \end{cases} \tag{4.38}$$

If the degree constraint is violated, then not all the probabilities are necessarily well-defined. So if we suppose that not all the probabilities in Equation (4.38) are well-defined we need to redefine the probabilities. In the case where the degree constraint is violated we define the probability of choosing edge (i, j) as follows

$$p_{ij} = \begin{cases} \min\left\{\frac{k_i k_j}{2m}, 1\right\} & i \neq j, \\ 0 & i = j, \end{cases} \tag{4.39}$$

and where we refer to the model defined by the probabilities in Equation (4.39) as Model VI. We can calculate the expected degree for Model VI as

$$\begin{aligned} E(\mathbf{D}_i) = E\left(\sum_j \mathbf{A}_{ij}\right) &= \sum_j E(\mathbf{A}_{ij}) \\ &= \sum_{j \neq i} \min\left\{\frac{k_i k_j}{2m}, 1\right\}, \end{aligned} \tag{4.40}$$

where $E(\mathbf{D}_i) \leq k_i - \frac{k_i^2}{2m}$ and if the constraint $k_i^2 \leq 2m \;\forall\; i$ holds then $E(\mathbf{D}_i) = k_i - \frac{k_i^2}{2m}$. The difference between k_i and $E(\mathbf{D}_i)$ can be written as

$$k_i - E(\mathbf{D}_i) = \frac{k_i^2}{2m} + \sum_{j \neq i}\left(\frac{k_i k_j}{2m} - \min\left\{\frac{k_i k_j}{2m}, 1\right\}\right). \tag{4.41}$$

We once again use the autonomous system graph to explore this model numerically. The expected degree is given by Equation (4.40) and in Figure 4.12 we plot the difference between $E(\mathbf{D}_i)$ and the actual degree, k_i, where k_i is also the expected degree value in Model I. Figure 4.13 plots the difference between the expected degree and the expected degree bound, which is given by $\widehat{E(\mathbf{D}_i)} = k_i - \frac{k_i^2}{2m}$. Note that $E(\mathbf{D}_i) \leq \widehat{E(\mathbf{D}_i)} < k_i$ and the difference between $\widehat{E(\mathbf{D}_i)}$ and k_i can be quite large, as can be seen in Figure 4.14. Contrast this to the case where $k_i^2 \leq 2m \;\forall\; i$, where the maximum difference between k_i and $k_i - \frac{k_i^2}{2m}$ is 1 since $\frac{k_i^2}{2m} \leq 1 \;\forall\; i$. However, the majority of the difference between $E(\mathbf{D}_i)$ and k_i is in the difference between $E(\mathbf{D}_i)$ and $\widehat{E(\mathbf{D}_i)}$ as can be seen in Figure 4.13.

To generate instances of Model VI we could use the $\mathcal{O}(n^2)$ Algorithm 5 with the probabilities from Equation 4.39. However, we could also use the $\mathcal{O}(m+n)$ algorithm introduced by [67] since it also generates instances of Model VI. Of course, if the degree constraint is satisfied Model VI reduces to Model III, the BCL model without self-edges, and the $\mathcal{O}(m+n)$ algorithm generates instances of this model when the degree constraint is satisfied. Thus, the $\mathcal{O}(m+n)$ algorithm offers an alternative algorithm for generating instances of these models and is inexpensive, relative to the the $\mathcal{O}(n^2)$ algorithm. Even though Model III is an approximation to the original Chung-Lu model it is still a better approximation than Model IV since the upper bound on the expected degree of node i in Model IV in given by the expected degree of node i in Model III. One could then make the argument that rather than using the $O(m)$ algorithm, Algorithm 8, we should use the $O(m+n)$ algorithm given by [67] since it generates instances that are a better approximation to the original Chung-Lu model. Similarly, as we will see below, when the degree constraint is violated, Model IV does not change and the upper bound on the expected degree of node i is given by the expected degree of node i in Model VI. This again suggests that using the algorithm proposed by [67] is preferable over Algorithm 8 since it generates instances that are a better approximation to the original Chung-Lu model. However, in our experience the $O(m+n)$ was about twice as slow as Algorithm 8 at generating instances which can be a significant reducation in speed if the graph is very large and thus it still remains interesting to explore the properties of Model IV and Algorithm 8.

Finally, we consider Model IV, the MCL model with no self-edges, under the assumption that the constraint $k_i^2 \leq 2m \ \forall \ i$ is violated. The probability of choosing edge (i,j) in this model is given by

$$p_{ij}(m) = \begin{cases} 1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^m & i \neq j, \\ 0 & i = j. \end{cases} \tag{4.42}$$

When the constraint $k_i^2 \leq 2m \ \forall \ i$ is violated, from Section 4.5.2 we know that $p_{ij}(m)$ is still well-defined for $i \neq j$ and since $p_{ii}(m) = 0 \ \forall \ i$, $p_{ij}(m)$ is still well-defined for all i, j. Thus, our model does not change and the expected degree is still given by Equation (4.29). As in Section 4.5.2, we can put an upper bound on the expected degree and compare this to the expected degree from Model I. As we noted in Section 4.5.2 the maximum value for $p_{ij}(m)$ is $\min\left\{1, \frac{k_i k_j}{2m}\right\}$ for $i \neq j$. Thus, the upper bound on the expected degree of node i is

$$\widehat{E(\mathbf{D}_i)} = \sum_{j \neq i} \min\left\{\frac{k_i k_j}{2m}, 1\right\}, \tag{4.43}$$

which is the same as $E(\mathbf{D}_i)$ from Model VI. Note that the expected degree from Model

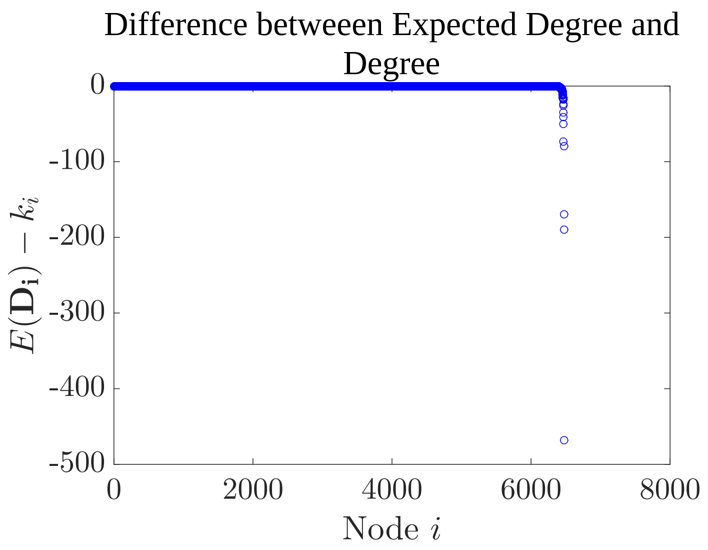


Figure 4.12: Autonomous System Network (Constraint Violated): Comparing Expected Degree from Model VI and Actual Degree.

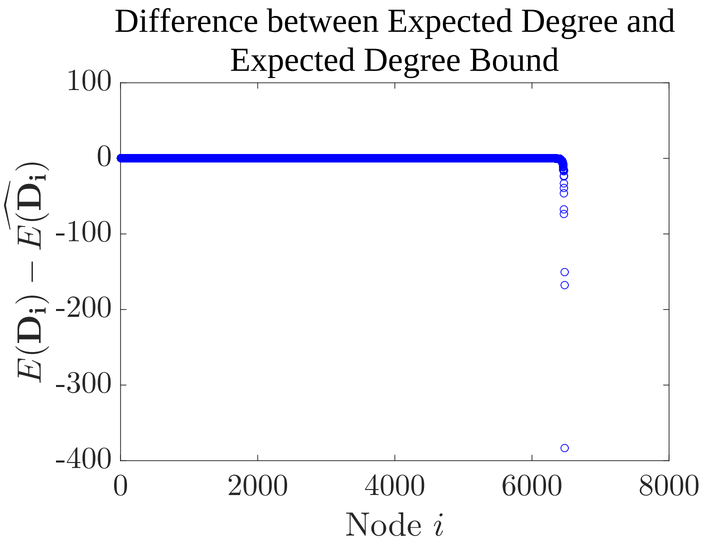


Figure 4.13: Autonomous System Network (Constraint Violated): Comparing Expected Degree and Expected Degree Bound from Model VI.

VI will always be strictly greater than the expected degree from Model IV. To see this we

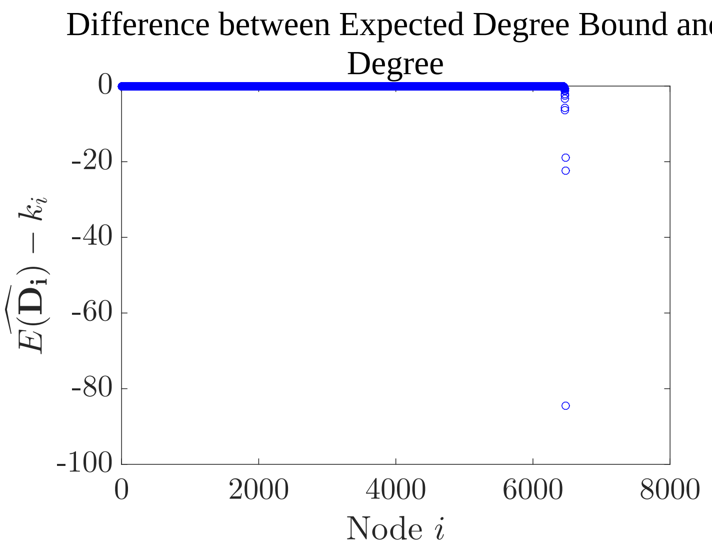


Figure 4.14: Autonomous System Network (Constraint Violated): Comparing Expected Degree Bound from Model VI and Actual Degree.

note that for nodes $i \neq j$ where $\frac{k_i k_j}{2m} > 1$, the probability of edge (i, j) being in Model VI is 1 and in Model IV is $1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^m < 1$. For nodes $i \neq j$ where $\frac{k_i k_j}{2m} \leq 1$, the probability of edge (i, j) being in Model VI is $\frac{k_i k_j}{2m}$ and in Model IV is $1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^m < \frac{k_i k_j}{2m}$. Note that we have the following relationship for Model IV,

$$E(\mathbf{D}_i) < \widehat{E(\mathbf{D}_i)} \leq k_i - \frac{k_i^2}{2m} < k_i, \tag{4.44}$$

where $k_i - \frac{k_i^2}{2m}$ is the expected degree from Model III and k_i is the expected degree from Model I. To see how large these differences can be empirically we again use the autonomous system graph. Figure 4.15 summarizes these differences by plotting the difference between the expected degree and the degree in the original graph which is quite large for some large degree nodes. Figure 4.16 plots the difference between the expected degree and the expected degree bound, while Figure 4.17 plots the difference between the expected degree bound and the actual degree.

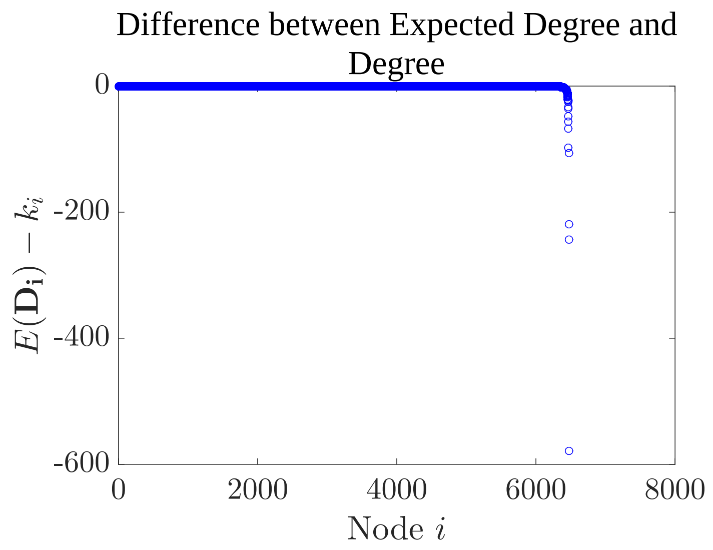


Figure 4.15: Autonomous System Network (Constraint Violated): Comparing Expected Degree from Model IV and Actual Degree.

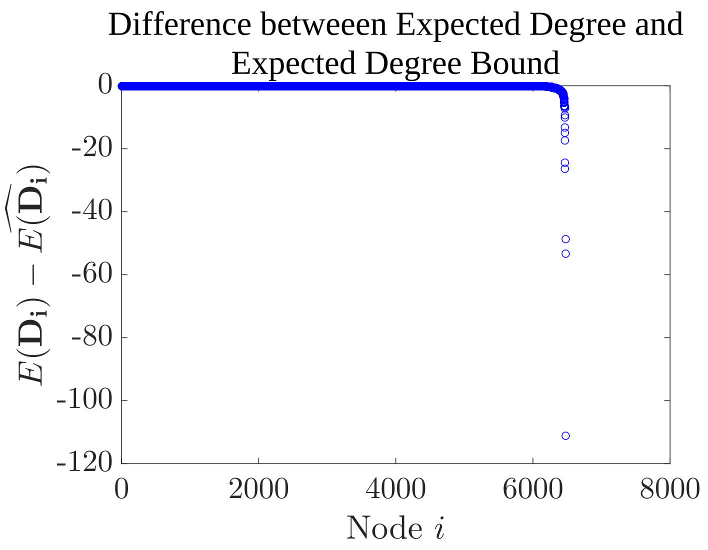


Figure 4.16: Autonomous System Network (Constraint Violated): Comparing Expected Degree and Expected Degree Bound from Model IV.

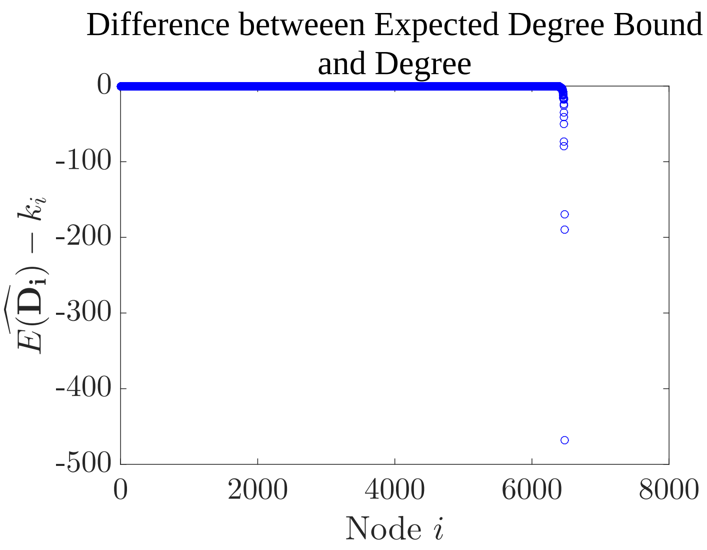


Figure 4.17: Autonomous System Network (Constraint Violated): Comparing Expected Degree Bound from Model IV and Actual Degree.

4.5.4 Summary

We have examined both the Bernoulli and $\mathcal{O}(m)$ Chung-Lu models with and without self-edges when the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated. When the constraint is violated in the BCL model, both with and without self-edges, we actually need to define new models. However, when the constraint is violated in the MCL model, both with and without self-edges, the models do not change. The following table summarizes the models presented when the constraint $k_i^2 \leq 2m \; \forall \; i$ is violated and note that for all four models, when the constraint is violated, the degree of certain nodes, particularly large degree nodes, in any given instance of the model can be much less than the degree in the original graph.

4.6 Triangle Counts and Clustering Coefficients

One of the nice properties of the BCL model, as originally proposed, is the fact that $E(\mathbf{D}_i) = k_i \; \forall \, i$. This makes it an attractive model, since it can be used to model networks with any degree sequence. However, researchers are often interested in building models that match other network properties, including the network community structure. The network community structure is often measured by the clustering coefficient which can be

		Probability of Edge (i,j)	Probability of Edge (i,i)	Expected Degree $E(\mathbf{D}_i)$	Expected Degree Bound $\widehat{E(\mathbf{D}_i)}$
Bernoulli	Self-Edges Model V	$\min\left\{\frac{k_ik_j}{2m}, 1\right\}$	$\min\left\{\frac{k_i^2}{4m}, 1\right\}$	$\sum_{j\neq i} \min\left\{\frac{k_ik_j}{2m}, 1\right\} + \min\left\{\frac{k_i^2}{2m}, 2\right\}$	k_i
Chung-Lu	No Self-Edges Model VI	$\min\left\{\frac{k_ik_j}{2m}, 1\right\}$	0	$\sum_{j\neq i} \min\left\{\frac{k_ik_j}{2m}, 1\right\}$	$k_i - \frac{k_i^2}{2m}$
$\mathcal{O}(m)$	Self-Edges Model II	$1 - \left(1 - 2\frac{k_ik_j}{4m^2}\right)^m < \min\left\{\frac{k_ik_j}{2m}, 1\right\}$	$1 - \left(1 - \frac{k_i^2}{4m^2}\right)^m < \min\left\{\frac{k_i^2}{4m}, 1\right\}$	$\sum_{j\neq i} 1 - \left(1 - 2\frac{k_ik_j}{4m^2}\right)^m + 1 - \left(1 - \frac{k_i^2}{4m^2}\right)^m$	$\sum_{j\neq i} \min\left\{\frac{k_ik_j}{2m}, 1\right\} + \min\left\{\frac{k_i^2}{2m}, 2\right\}$
Chung-Lu	No Self-Edges Model IV	$1 - \left(1 - 2\frac{k_ik_j}{4m^2}\right)^m < \min\left\{\frac{k_ik_j}{2m}, 1\right\}$	0	$\sum_{j\neq i} 1 - \left(1 - 2\frac{k_ik_j}{4m^2}\right)^m$	$\sum_{j\neq i} \min\left\{\frac{k_ik_j}{2m}, 1\right\}$

Table 4.3: Model Summary – Degree Constraint Violated

defined for each node i as follows

$$C_i = \frac{\text{Number of pairs of neighbors of } i \text{ that are connected}}{\text{Number of pairs of neighbors of } i} \tag{4.45}$$

or

$$C_i = \frac{\text{Number of unique triangles node } i \text{ belongs to}}{\text{Number of pairs of neighbors of } i}. \tag{4.46}$$

The clustering coefficient can be written as a random variable, $\mathbf{C}_i$, where $\mathbf{C}_i$ is a function of the random variables $\mathbf{A}_{ij}$, $j = 1, \ldots, n$. We can then compare the realizations of this random variable from our model instances and its expected value with the clustering coefficients from the real network. Unfortunately, the expected value of the clustering coefficient, $E(\mathbf{C}_i)$, is not easily determined. Our analysis would be greatly simplified if we could write the following

$$E(\mathbf{C}_i) = \frac{E(\text{Number of unique triangles node } i \text{ belongs to})}{E(\text{Number of pairs of neighbors of } i)}, \tag{4.47}$$

however this equation does not hold in general. Despite the fact that $E(\mathbf{C}_i)$ cannot be written as above we note that in the BCL model, Model I, $E(\mathbf{D}_i) = k_i$, and thus on average the number of pairs of neighbors of node i is the same in any instance of Model I as in the real network. It has been noted empirically that the BCL model generates instances that under-estimate the clustering coefficient for many real-world networks [80, p. 450]. Thus, if $\mathbf{C}_i = c_i$ is much less on average in the model instances then it is due to the lack of triangles for node i. In all the other models we have introduced, $E(\mathbf{D}_i) < k_i$, so the number of pairs of neighbors of node i is on average less in a given model instance than in the real network. Thus, for $\mathbf{C}_i = c_i$ to be less on average for a given model instance relative to the real-world network, the number of triangles for node i must be less on average in the model instance relative to the real network by a greater proportion than the number of pairs of neighbors. Not only does the number of triangles for each node play an important role in determining the clustering coefficient but it can also be used as a measure of community structure independent of the clustering coefficient. Hence, we focus on determining its value in our models as a way to look at community structure and to examine the difference between our models and real networks. We begin by noting that the number of triangles for a given node i in the original Chung-Lu model, Model I, is a random variable that can be defined as

$$\mathbf{T}_i = \sum_{j \neq i} \left(\sum_{k \neq i, k \neq j} \mathbf{A}_{ij} \mathbf{A}_{ik} \mathbf{A}_{jk} \right), \ \forall\, i, \tag{4.48}$$

where a triangle occurs at node i if there is an edge between nodes i and j, between nodes i and k and between nodes k and j where the nodes i, j and k are all unique nodes. Note that every triangle is counted twice in the above sum since A_{jk} and A_{kj} both appear. Since $\mathbf{T}_i$ is a random variable we can compute the expected value of $\mathbf{T}_i$ as

$$\begin{aligned} E(\mathbf{T}_i) &= \sum_{j\neq i}\sum_{k\neq i,j} \mathbf{A}_{ij}\mathbf{A}_{ik}\mathbf{A}_{jk} \\ &= \sum_{j\neq i}\sum_{k\neq i,j} E(\mathbf{A}_{ij}\mathbf{A}_{ik}\mathbf{A}_{jk}) \\ &= \sum_{j\neq i}\sum_{k\neq i,j} E(\mathbf{A}_{ij})E(\mathbf{A}_{ik})E(\mathbf{A}_{jk}) \\ &= \sum_{j\neq i}\sum_{k\neq i,j} p_{ij}p_{ik}p_{jk}, \end{aligned} \tag{4.49}$$

where $E(\mathbf{A}_{ij}\mathbf{A}_{ik}\mathbf{A}_{jk}) = E(\mathbf{A}_{ij})E(\mathbf{A}_{ik})E(\mathbf{A}_{jk})$ because each edge is drawn independently. Equation (4.3) defines $p_{ij}, i, j = 1, \ldots, n$, in Model I which allows us to simplify $E(\mathbf{T}_i)$ as follows

$$\begin{aligned} E(\mathbf{T}_i) &= \sum_{j\neq i}\sum_{k\neq i,j} p_{ij}p_{ik}p_{jk} \\ &= \sum_{j\neq i}\sum_{k\neq i,j} \left(\frac{k_ik_j}{2m}\right)\left(\frac{k_ik_k}{2m}\right)\left(\frac{k_jk_k}{2m}\right) \\ &= \frac{k_i^2}{(2m)^3}\sum_{j\neq i} k_j^2 \sum_{k\neq i,j} k_k^2 \\ &= \frac{k_i^2}{(2m)^3}\sum_{j\neq i} k_j^2\left(\mathcal{K}_2 - k_i^2 - k_j^2\right) \\ &= \frac{k_i^2}{(2m)^3}\sum_{j\neq i} \mathcal{K}_2k_j^2 - k_i^2k_j^2 - k_j^4 \\ &= \frac{k_i^2}{(2m)^3}\left(\mathcal{K}_2(\mathcal{K}_2 - k_i^2) - k_i^2(\mathcal{K}_2 - k_i^2) - (\mathcal{K}_4 - k_i^4)\right) \\ E(\mathbf{T}_i) &= \frac{k_i^2}{(2m)^3}\left((\mathcal{K}_2 - k_i^2)^2 - (\mathcal{K}_4 - k_i^4)\right), \end{aligned} \tag{4.50}$$

where we define $\mathcal{K}_2 = \sum_{i=1}^n k_i^2$ and $\mathcal{K}_4 = \sum_{i=1}^n k_i^4$. Note that the expected value of the number of triangles at node i is a function of the degrees and is not related to the number of triangles in the network since the number of triangles is not an input into the model. If we were interested in better matching the expected number of triangles at node i to the actual number of triangles at node i in the original graph, t_i, then a possible strategy would be to make p_{ij} a function of t_i and t_j. However, $E(\mathbf{D}_i)$ would then be a function of t_i and would no longer be equal to k_i. Also note that random variable $\mathbf{T}_i$ is not a function of any self-edges which means that neither is $E(\mathbf{T}_i)$ and thus for Model III, the BCL model without self-edges, $E(\mathbf{T}_i)$ is also given by Equation (4.50). For Models II and IV the MCL models with and without self-edges, respectively, we can determine $E(\mathbf{T}_i)$ if we replace p_{ij} in Equation (4.49) with $p_{ij}(m)$, $i \neq j$ from Equation (4.20) as follows

$$\begin{aligned} E(\mathbf{T}_i) &= \sum_{j\neq i}\sum_{k\neq i,j} p_{ij}(m)p_{ik}(m)p_{jk}(m) \\ &= \sum_{j\neq i}\sum_{k\neq i,j}\left(1-\left(1-2\frac{k_ik_j}{(2m)^2})\right)^m\right)\left(1-\left(1-2\frac{k_ik_k}{(2m)^2}\right)^m\right)\left(1-\left(1-2\frac{k_jk_k}{(2m)^2}\right)^m\right), \end{aligned} \tag{4.51}$$

which cannot be easily simplified. However, as we have noted before the probability of an edge between distinct nodes in Models II and IV is lower than in Models I and III. This implies that $E(\mathbf{T}_i)$ will be lower in Models II and IV than in Models I and III. To illustrate the differences between $E(\mathbf{T}_i)$ in our models and in a real network we revisit the general relativity collaboration network. Figure 4.18 plots the difference between the expected number of triangles in Models I and III and the actual number of triangles in the general relativity collaboration network where the nodes are still ordered by degree from smallest to largest. This difference is quite large for some nodes. Figure 4.19 plots the relative difference between the expected number of triangles in Models I and III and the actual number of triangles for each node with at least one triangle in the real network. For Models II and IV, Figure 4.20 plots the difference between the expected number of triangles and the actual number of triangles and Figure 4.21 plots the relative difference between the expected number of triangles and the actual number of triangles. From these figures it is hard to determine if the expected number of triangles in Models I and III is closer to the actual triangle count or if the expected number of triangles in Models II and IV is closer to the actual triangle count. Figure 4.22 plots the difference between the expected number of triangles in Models II and IV and the upper bound which is the expected number of triangles in Models I and III. From the figure we can see that the expected number of triangles in Models I and III is higher than in Models II and IV and thus the expected number of triangles in Models I and III is closer to the true triangle count. In general, since $E(\mathbf{T}_i)$ is not a function of T_i we can not draw any conclusions about whether $E(\mathbf{T}_i)$ will be closer to t_i in Models I and III or in Models II and IV.

In the above analysis we have assumed that the constraint, $k_i^2 \leq 2m \; \forall \; i$, holds. If this constraint no longer holds, then our above analysis needs to be revised to account for the

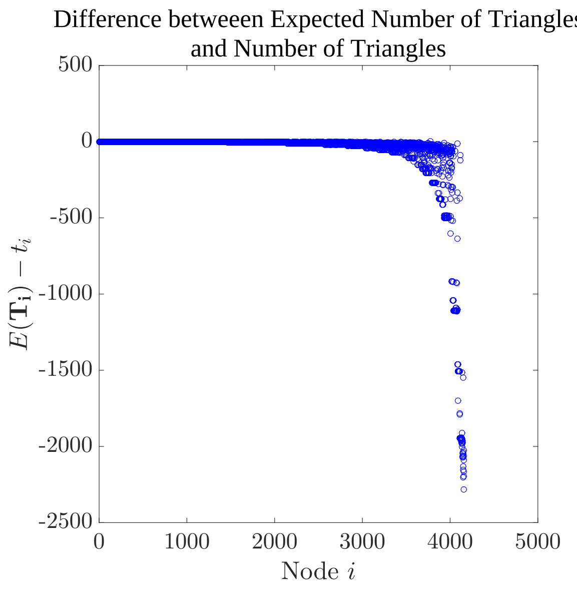


Figure 4.18: General Relativity Collaboration Network: Comparing Expected Number of Triangles from Models I and III and Actual Number of Triangles.

constraint violation. As we mentioned in Section 4.5, if the constraint, $k_i^2 \leq 2m \; \forall \; i$, is violated then instead of using Models I and III we use Models V and VI, respectively. For both Model V and VI, $E(\mathbf{T}_i)$ is the same, and can be determined by using Equation (4.31) to define p_{ij}, $i \neq j$, and substituting this into Equation (4.49). This gives us the following

$$\begin{aligned} E(\mathbf{T}_i) &= \sum_{j \neq i} \sum_{k \neq i,j} p_{ij} p_{ik} p_{jk} \\ &= \sum_{j \neq i} \sum_{k \neq i,j} \min\left\{\frac{k_i k_j}{2m}, 1\right\} \min\left\{\frac{k_i k_k}{2m}, 1\right\} \min\left\{\frac{k_j k_k}{2m}, 1\right\}. \end{aligned} \tag{4.52}$$

Expression (4.52) will be less than expresssion (4.50). However, this does not imply that when the constraint is violated $E(\mathbf{T}_i)$ will be further from the true triangle count since $E(\mathbf{T}_i)$ is not a function of the number of the triangles in the original network. Expression (4.50) only provides an upper bound on expression (4.52).

For the MCL models, Models II and IV, when the constraint, $k_i^2 \leq 2m \; \forall \; i$, is violated, the models do not change and hence $E(\mathbf{T}_i)$ is still given by Equation (4.51). However, if the constraint is violated then the upper bound on $p_{ij}(m)$, $i \neq j$, is given by

$$p_{ij}(m) < \min\left\{1, \frac{k_i k_j}{2m}\right\}. \tag{4.53}$$

This implies that in this case the upper bound on $E(\mathbf{T}_i)$ is given by Equation (4.52), which is a tighter upper bound than Equation (4.50). Note that if the constraint, $k_i^2 \leq 2m \;\forall\; i$, holds then Equations (4.52) and (4.50) are the same.

Numerically, we can examine the effect of the constraint violation on the number of triangles using the autonomous system network we have examined previously. Figures 4.23 and 4.24 plot the difference between $\mathbf{E}(T_i)$ and t_i, the actual number of triangles node i belongs to, for Models V and VI. Notice how for this network $E(\mathbf{T}_i) > t_i$ for a number of nodes whereas for the general relativity network, $E(\mathbf{T}_i) < t_i$ for all nodes i. Note that for the general relativity network, which does not violate the constraint, $k_i^2 \leq 2m$, $\forall\; i$, Models I and III are equivalent to Models V and VI, respectively. In general, since $E(\mathbf{T}_i)$ is not a function of t_i we can not draw any conclusions about whether $E(\mathbf{T}_i)$ will be greater or less than t_i. Figures 4.25 and 4.26 plot the difference between between $\mathbf{E}(T_i)$ and t_i for Models II and IV. Note from the figures that Models II and IV are actually better able to match

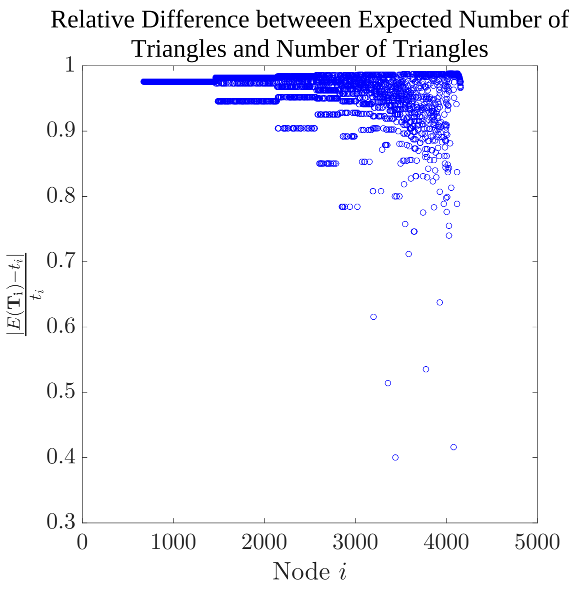


Figure 4.19: General Relativity Collaboration Network: Relative difference between Expected Number of Triangles from Models I and III and Actual Number of Triangles.

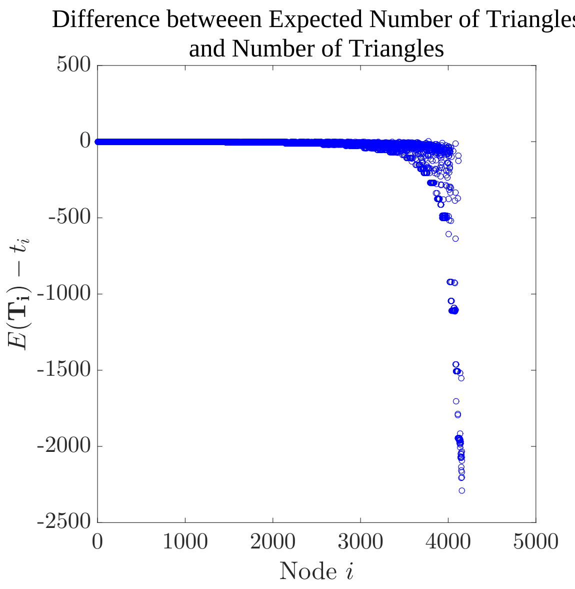


Figure 4.20: General Relativity Collaboration Network: Comparing Expected Number of Triangles from Models II and IV and Actual Number of Triangles.

the number of triangles than Models V and VI although no conclusion can be drawn in general about which set of models will better match the triangle count data.

4.7 Conclusion

The original Chung-Lu model is a simple model that has the nice property that the expected degree is given by the input degree sequence. However, we have shown that when changes are made to the original model (i.e. excluding self-edges) and the constraints on the degrees of the input degree sequence do not hold then this property no longer holds. In the case where we exclude self-edges this difference can be quite small. However, when the degrees in the degree sequence violate the constraint this difference can be quite large. We have also examined in-depth an approximate Chung-Lu model, the MCL model, and shown how large this approximation can be especially when the degree constraints are violated. It is important to know how much this approximation can effect any instance created using either Model II or Model IV. In particular, we must be aware that when the degree constraint is violated then the expected degree can in fact be quite different than the original degree sequence. We also looked at another important network property, triangle

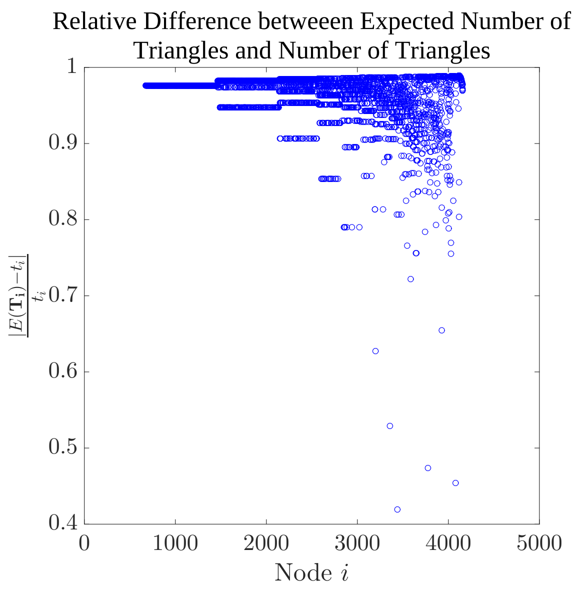


Figure 4.21: General Relativity Collaboration Network: Relative difference between Expected Number of Triangles from Models II and IV and Actual Number of Triangles.

counts, and noted that the Chung-Lu model is not designed to match this property and typically underestimates the number of triangles in networks we are interested in. In the next Chapter we will look at some ways to try and improve the MCL model and decrease the approximation error.

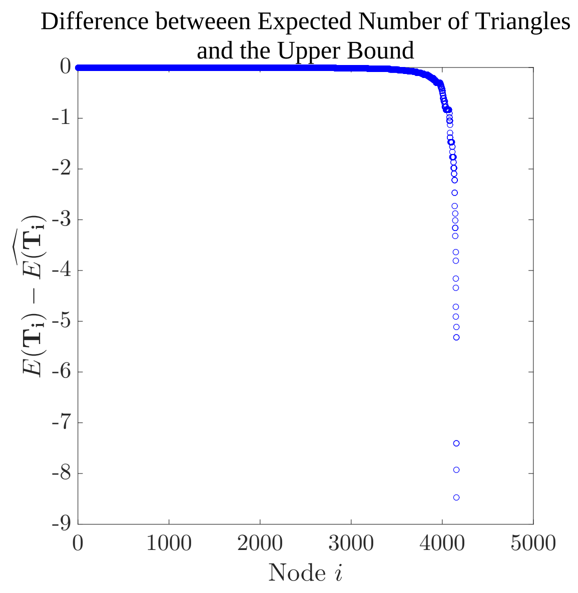


Figure 4.22: General Relativity Collaboration Network: Comparing Expected Number of Triangles from Models II and IV and the Upper Bound.

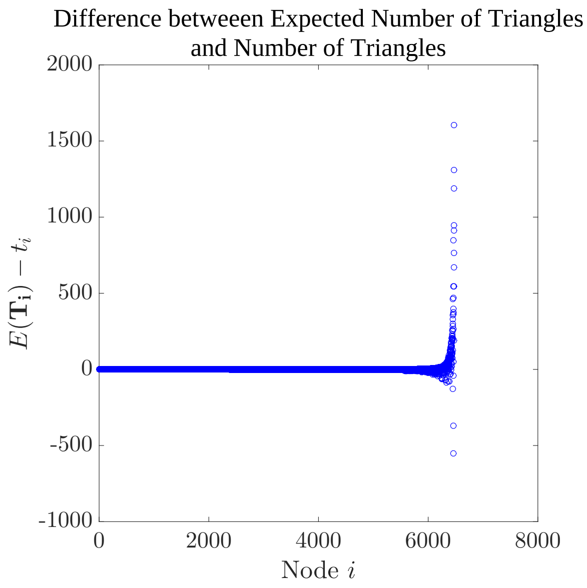


Figure 4.23: Autonomous System Network (Constraint Violated): Comparing Expected Number of Triangles from Models V and VI and Actual Number of Triangles.

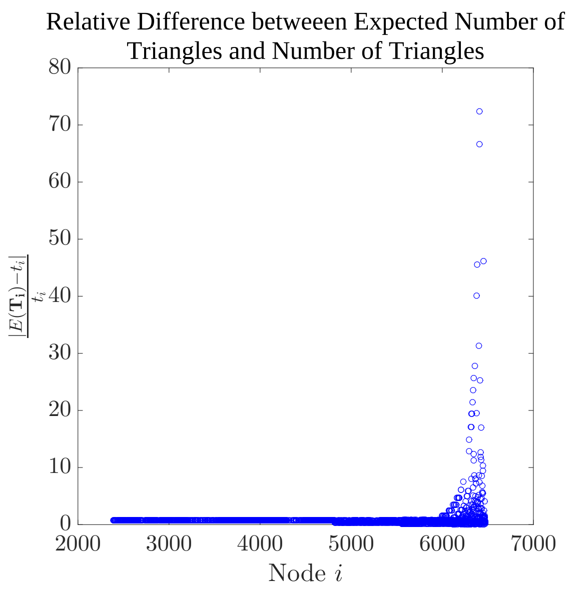


Figure 4.24: Autonomous System Network (Constraint Violated): Relative difference between Expected Number of Triangles from Models V and VI and Actual Number of Triangles.

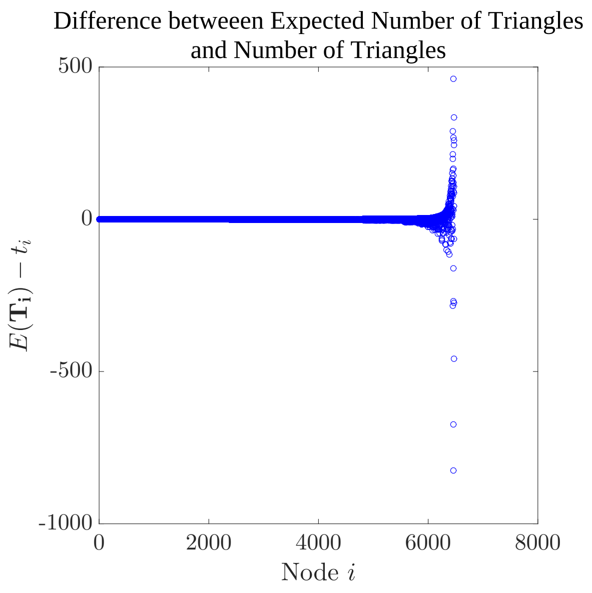


Figure 4.25: Autonomous System Network (Constraint Violated): Comparing Expected Number of Triangles from Models II and IV and Actual Number of Triangles.

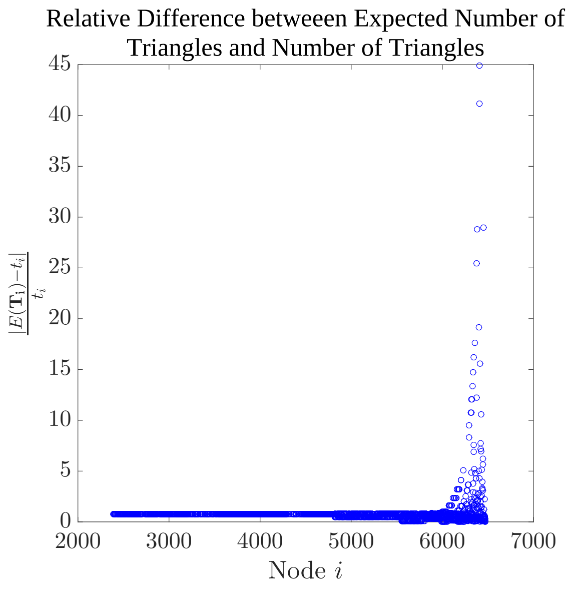


Figure 4.26: Autonomous System Network (Constraint Violated): Relative difference between Expected Number of Triangles from Models II and IV and Actual Number of Triangles.

Chapter 5

Improving the $\mathcal{O}(m)$ Chung-Lu model

5.1 Introduction

Consider the $\mathcal{O}(m)$ Chung-Lu model without self-edges, Model IV, as described in Section 4.4.2. Algorithm 8 is used to generate instances of this model and often these instances are referred to as instances of the original Chung-Lu model, Model I. As we have mentioned previously, in the original Chung-Lu model, which assumes that the constraint $k_i^2 < 2m \; \forall \, i$ is satisfied, the expected degree of node i is equal to the input degree, k_i, and the expected number of edges is equal to m, the edge count in the original network. However, as we saw in the previous Chapter, the expected degree of node i and the expected edge count can be much different under Model IV than in Model I, and thus the instances that are referred to as representations of the original Chung-Lu model can have fairly different properties than what is assumed. If we want to use these instances to compare with other graph generators we must be aware of the actual model underlying our instances and its properties to make a fair comparison. Given this, the next step in our analysis is to look at simple ways of modifying Model IV to better approximate the original Chung-Lu model, Model I. In our discussion we only consider Model IV since this is the model most used in practice (i.e. Algorithm 8 is most often used to generate instances of the "Chung-Lu" model). In Section 5.2, we examine how increasing the number of draws can change the model. In Section 5.3 we create an algorithm that first uses Algorithm 8 and then looks at the resulting instance to determine any additional edges needed. In Section 5.4 we examine what happens if we let the probabilities in Equation (4.29) have a more general form (i.e. $p_{ij} \neq \frac{k_i k_j}{(2m)^2}$). This results in a constrained optimization problem that is difficult to solve and we turn to fixed point methods for an approximate solution. Finally, in Section 5.5 we examine the improvement proposed in [85] and compare it to the improvements we suggest. In Sections 5.2, 5.3, 5.4 and 5.5, we assume that the degree constraint is not violated. In Section 5.6 we look at the improvements introduced in the previous sections when the constraint $k_i^2 < 2m \; \forall \; i$ is violated.

5.2 Drawing Additional Edges

As we mentioned previously, we are interested in modifying Model IV to improve the approximation error. For Model IV, the probability of edge (i, j) being in the graph after m draws is given by

$$p_{ij}(m) = \begin{cases} 1 - \left(1 - \frac{k_i k_j}{4m^2}\right)^m & i \neq j, \\ 0 & i = j, \end{cases} \tag{5.1}$$

where $p_{ij}(m) \approx \frac{k_i k_j}{2m}$ but $p_{ij}(m)$ is strictly less than $\frac{k_i k_j}{2m}$. For now let us assume that the input degree sequence does not violate the constraint, $k_i^2 < 2m \; \forall \; i$, and thus $\frac{k_i k_j}{2m} \leq 1$. If we could increase the probability $p_{ij}(m)$ such that it was closer to $\frac{k_i k_j}{2m}$ then $E(\mathbf{D}_i)$ would be be closer to k_i (see Equation (4.8)) and $E(\mathbf{M})$ would better approximate m. One way to increase this probability is to draw more edges. So instead of drawing m edges we draw ω edges and Algorithm 8 becomes the following:

Algorithm 9: $\mathcal{O}(\omega)$ Chung-Lu Algorithm without Self-Edges.

```
for $k = 1$ to $\omega$ do
    Draw node $i$ with probability $\frac{k_i}{2m}$;
    Draw node $j$ with probability $\frac{k_j}{2m}$;
    /* Add edge $(i, j)$ to the graph */
    if $i \neq j$ then
        $a_{ij} = a_{ji} = 1$;
    end
end
```

The probability of edge (i, j) being in the graph after ω draws is given by

$$p_{ij}(\omega) = \begin{cases} 1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^\omega & i \neq j, \\ 0 & i = j, \end{cases} \tag{5.2}$$

and we refer to the model underlying Algorithm 9 and described by Equation (5.2) as the $\mathcal{O}(\omega)$ Chung-Lu (ωCL) model without self-edges. Using Equation (4.15) we get that $p_{ij}(\omega) \approx \frac{\omega}{m} \cdot \frac{k_i k_j}{2m}$ with $p_{ij}(\omega)$ strictly less than $\frac{\omega}{m} \cdot \frac{k_i k_j}{2m}$. Since $\omega \geq m$ this implies that $p_{ij}(\omega)$ can equal $\frac{k_i k_j}{2m}$ if we choose the correct ω; however, we have only one parameter, ω, and we want to match $\frac{n(n-1)}{2}$ probabilities. In fact, for certain values of ω, $p_{ij}(\omega)$ may be larger than $\frac{k_i k_j}{2m}$ which implies that $E(\mathbf{D}_i)$ may be greater than k_i. So, how many edges do we draw? One approach is to draw as many edges as necessary to get m edges in the resulting instance of the graph. This process is described in Algorithm 10.

Algorithm 10: A Matching Edge Count Chung-Lu Algorithm (No Self-Edges).

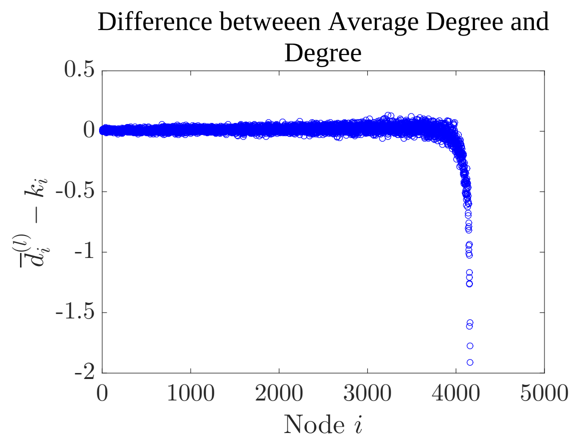
```
$\delta \leftarrow 1$;
while $\delta \leq m$ do
    Draw node $i$ with probability $\frac{k_i}{2m}$;
    Draw node $j$ with probability $\frac{k_j}{2m}$;
    /* Add edge $(i, j)$ to the graph */
    if $i \neq j$ and $a_{ij} = a_{ji} = 0$ then
        $a_{ij} = a_{ji} = 1$;
        $\delta \leftarrow \delta + 1$;
    end
end
```

The model underlying Algorithm 10 is not described by the probabilities in Equation (5.2) since the number of draws required to get m edges in a given instance of the model is variable. Instead, the probability of drawing a particular edge in the model underlying Algorithm 10 is given by the following

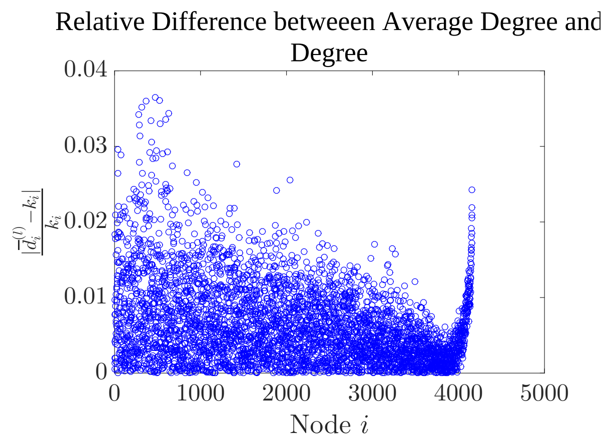
$$P(\text{There is an edge between nodes } i \text{ and } j \text{ in the graph}) =$$
$$\sum_{\omega=m}^{\infty} P(\gamma = \omega) \cdot P(\text{There is an edge between nodes } i \text{ and } j \text{ in the graph after } \omega \text{ draws}), \tag{5.3}$$

where γ is the number of draws required to get m edges in the graph and the minimum number of draws is m so $\omega \geq m$. Since we do not know $P(\gamma = \omega)$ we use the general relativity network and Algorithm 10 to generate $l = 10000$ synthetic graphs to study the properties of the underlying model numerically. The average value of ω is 13502.5 and the standard deviation is 9.01, where we recall that $m = 13422$. Figure 5.1 plots the difference and the relative difference between the average node degree and the actual degree. Figure

5.2 plots the difference between the average node degree and the expected node degree from Model IV. We saw in Figure 4.6, which plots the difference between the expected degree in Model IV and the actual degree, that the expected degree in Model IV underestimates the actual degree and we can see from Figures 5.1 and 5.2 that for large degree nodes adding edges moves the average degree closer to the actual degree relative to Model IV (since $\overline{d}_i^{(l)} > E(\mathbf{D}_i)_m$); however, the average degree still remains below the actual degree.



(a) $l = 10000$



(b) $l = 10000$

Figure 5.1: General Relativity Collaboration Network: Difference and Relative difference between Average Degree from Algorithm 10 and Actual Degree.

Suppose that we assume that $p_{ij}(\omega) = \frac{\omega}{m}\frac{k_i k_j}{2m}$ for $i \neq j$ ($p_{ij}(\omega) = 0$ for $i = j$). Then, for $i \neq j$,

$$\begin{aligned} P(\text{There is an edge between nodes } i \text{ and } j \text{ in the graph}) &= \sum_{\omega=m}^{\infty} P(\gamma = \omega)\frac{\omega}{m}\frac{k_i k_j}{2m} \\ &= \frac{1}{m}\frac{k_i k_j}{2m}\sum_{\omega=m}^{\infty} P(\gamma=\omega)\omega \\ &= \frac{E(\omega)}{m}\frac{k_i k_j}{2m} \\ &\approx \frac{\overline{\omega}}{m}\frac{k_i k_j}{2m}, \end{aligned} \tag{5.4}$$

where $\overline{\omega}$ is the average ω from l synthetic graphs generated using Algorithm 10 (i.e. $\overline{\omega} = 13502.5$ for the general relativity graph with $l = 10000$). Note that the probability of there being and edge between nodes i and j, $i \neq j$ in the ωCL model without self-edges with $\omega = \overline{\omega}$ is also approximately $\frac{\overline{\omega}}{m}\frac{k_i k_j}{2m}$. Thus, the ωCL model without self-edges with $\omega = \overline{\omega}$ could be used to approximate the model underlying Algorithm 10, assuming $p_{ij}(\omega) = \frac{\omega}{m}\frac{k_i k_j}{2m}$ is a reasonable approximation.

Therefore, returning to the general relativity network, we use the ωCL model without self-edges with $\omega = \overline{\omega} = 13502.5$ to approximate the model underlying Algorithm 10 and we plot the difference between the expected degree and the actual degree and the difference between the expected degree and the expected degree from Model IV in Figures 5.3 and 5.4, respectively. The same general pattern is seen in these Figures as in Figures 5.1a and 5.2. We should note that for approximately 90% of the nodes the expected degree under this model is actually higher than the actual degree.

Algorithm 10 has the nice property that ensures that every instance has exactly m edges but unfortunately we cannot quantify the underlying model for a given network ($P(\gamma = \omega)$ is unknown). As an alternative, we could use Algorithm 9 with ω chosen so

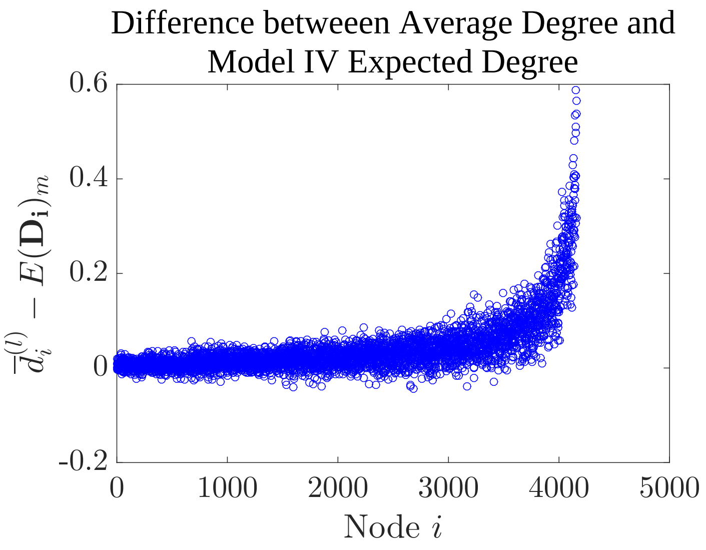


Figure 5.2: General Relativity Collaboration Network: Comparing Average Degree from Algorithm 10 ($l = 10000$) and Expected Degree from Model IV.

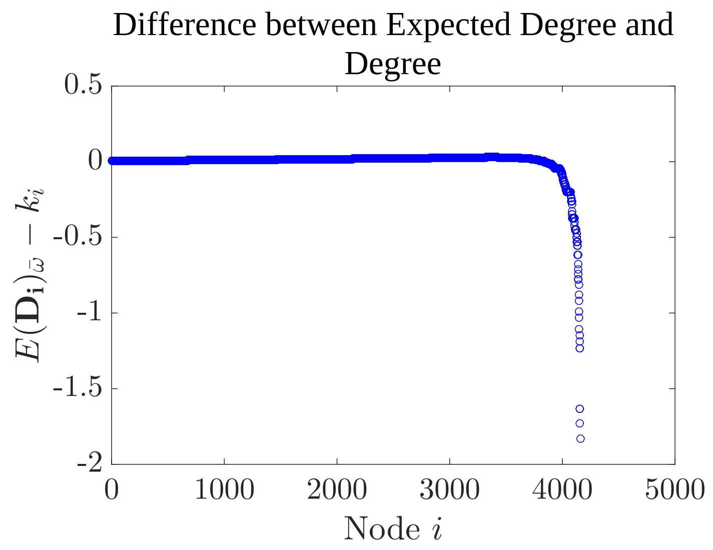


Figure 5.3: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = \overline{\omega} = 13502.5$ and Actual Degree.

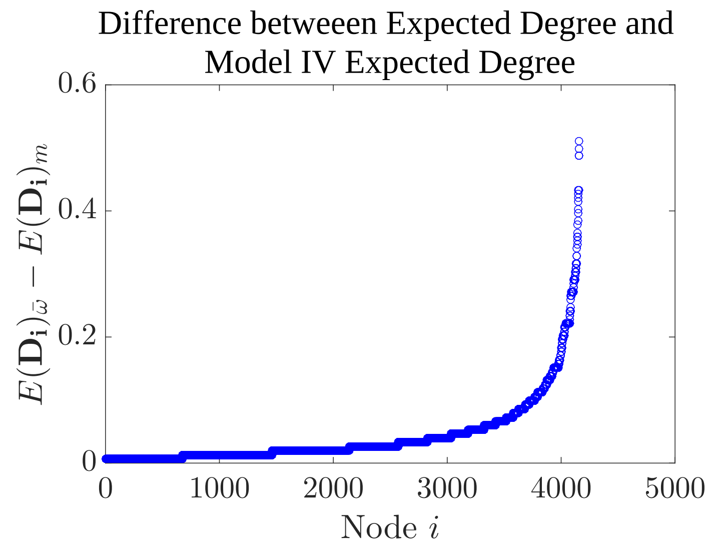


Figure 5.4: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = \overline{\omega} = 13502.5$ and Expected Degree from Model IV.

that $E(\mathbf{M}) = m$. The formula for $E(\mathbf{M})$ is given by the following

$$\begin{aligned} E(\mathbf{M}) &= \frac{1}{2}\sum_i E(\mathbf{D}_i) \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} E(\mathbf{A}_{ij}) \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} p_{ij}(\omega) \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} 1 - \left(1 - \frac{k_i k_j}{2^2}\right)^{\omega}. \end{aligned} \tag{5.5}$$

We can set $E(\mathbf{M})$ equal to m and solve for ω numerically. This gives $\omega = 13511.56 \approx 13512$. As an alternative, we could choose ω to match the degree of the highest degree node. The formula for $E(\mathbf{D}_i)$ is given by

$$\begin{aligned} E(\mathbf{D}_i) &= \sum_{j\neq i} E(\mathbf{A}_{ij}) \\ &= \sum_{j\neq i} p_{ij}(\omega) \\ &= \sum_{j\neq i} 1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^{\omega}. \end{aligned} \tag{5.6}$$

We can set $E(\mathbf{D}_i)$ equal to k_i where i is chosen to be the largest degree node and solve for ω numerically. For the general relativity network this gives $\omega = 13832.54 \approx 13833$. Finally, we consider matching all $\frac{n(n-1)}{2}$ probabilities (i.e. Find ω such that $p_{ij}(\omega) = \frac{k_i k_j}{2m}$, $i = 1, \ldots, n$, $j = i, \ldots, n$.) In this case we have $\frac{n(n-1)}{2}$ equations and only one unknown. So the resulting ω can be chosen as the least-squares solution to this overdetermined system. Since probabilities are given by

$$p_{ij}(\omega) = 1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^{\omega}, \tag{5.7}$$

where $i = 1, \ldots, n,\ j = i \ldots, n$, our system of equations is described by

$$\begin{aligned} \frac{k_i k_j}{2m} &= 1 - \left(1 - \frac{k_i k_j}{2m^2}\right)^{\omega} \\ \left(1 - \frac{k_i k_j}{2m^2}\right)^{\omega} &= 1 - \frac{k_i k_j}{2m} \\ \omega \ln\left(1 - \frac{k_i k_j}{2m^2}\right) &= \ln\left(1 - \frac{k_i k_j}{2m}\right), \end{aligned} \tag{5.8}$$

for $i = 1, \ldots, n,\ j = i \ldots, n$ where the following must hold: $\frac{k_i k_j}{2m} < 1\ \forall\ i, j \neq i$. If we place all $\frac{n(n-1)}{2} \ln\left(1 - \frac{k_i k_j}{4m^2}\right)$ terms in the vector $\mathbf{a}$ and all $\frac{n(n-1)}{2} \ln\left(1 - \frac{k_i k_j}{2m}\right)$ terms in the vector $\mathbf{b}$ we can write the system as $\mathbf{a}\omega = \mathbf{b}$ and the least squares solution is $\omega = \frac{\mathbf{a}^T\mathbf{b}}{\mathbf{a}^T\mathbf{a}}$. The least squares solution for the general relativity network is $\omega = 13720.66 \approx 13721$. For the three different ω outlined above, we use the general relativity network to plot the difference between the expected degree and the actual degree and the difference between the expected degree and the expected degree from Model IV in Figures 5.5 to 5.10. We can see from these figures that as we increase ω in order to match the node degree for the higher degree nodes we overshoot on the node degrees for the smaller degree nodes.

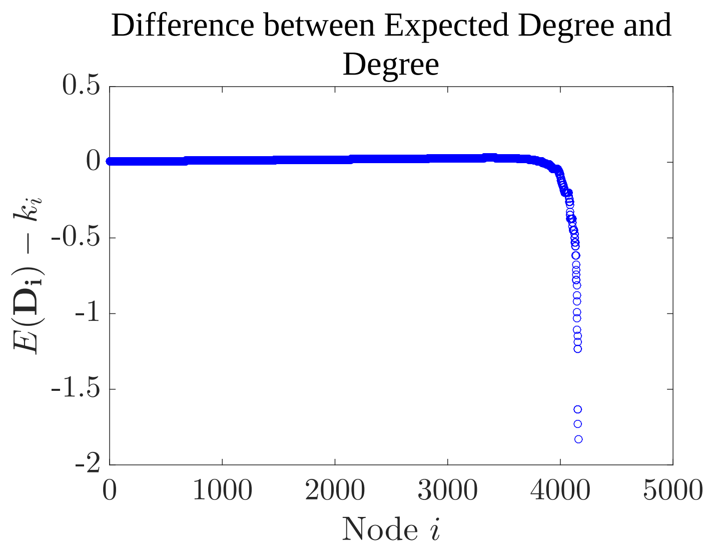


Figure 5.5: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 13511.56$ ($E(m) = m$) and Actual Degree.

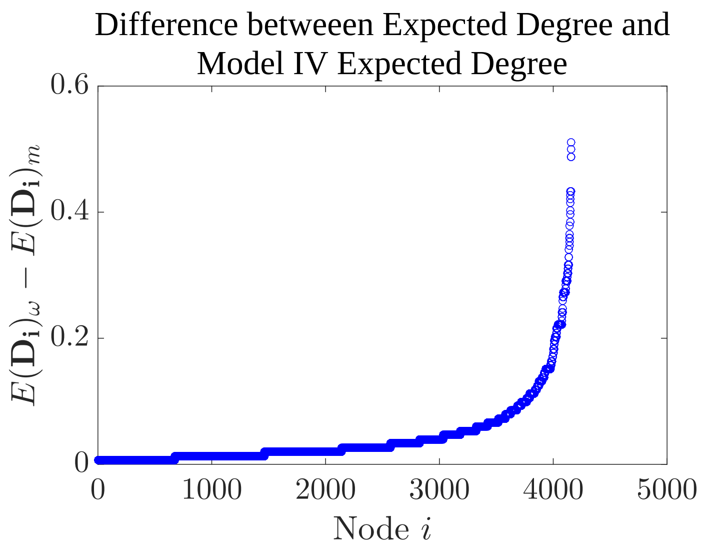


Figure 5.6: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 13511.56$ ($E(m) = m$) and Expected Degree from Model IV.

5.3 Two Distributions Algorithm

As the above analysis indicates, simply drawing more edges to add to our graph doesn't necessarily get us any closer to matching the degree distribution. Given this, we consider an alternative approach. We begin by using Algorithm 8, the MCL algorithm without self-edges, to build an initial instance of the graph. Then using this instance we create a "remaining degree" distribution for each node based on the remaining number of edges needed to match the degree of each node. So, for each node we calculate the following

$$r_i = k_i - d_i, \tag{5.9}$$

where k_i is the degree of node i in the original graph and d_i is the degree of node i after running Algorithm 8. For the nodes where $r_i < 0$, we set those values equal to zero, so in fact the formula for r_i is given by

$$r_i = \max\{k_i - d_i, 0\}, \tag{5.10}$$

and we define $m_r = \frac{1}{2}\sum_i r_i$. We then run Algorithm 8 again but this time with node i drawn with probability $\frac{r_i}{2m_r}$ and node j drawn with probability $\frac{r_j}{2m_r}$ and the number of

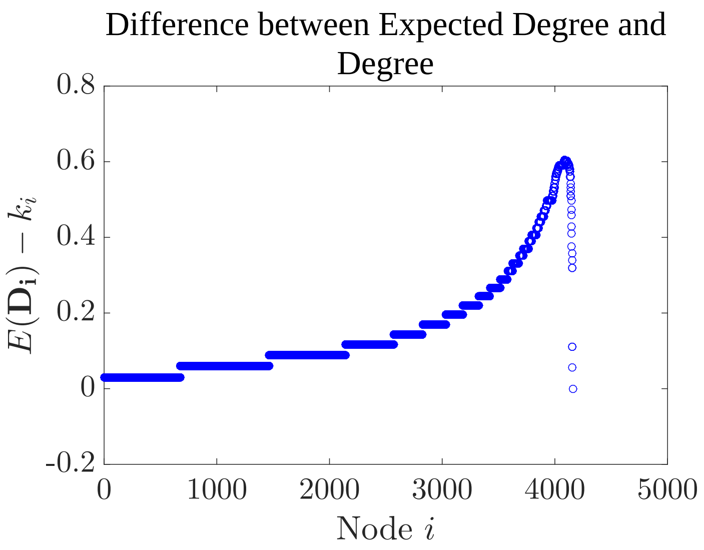


Figure 5.7: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 13832.54$ ($E(\mathbf{D}_i) = k_i$ for largest degree node) and Actual Degree.

draws equal to m_r. For this model, we have

$$p_{ij}(m_r) = 1 - \left(1 - \frac{r_i r_j}{2m_r^2}\right)^{m_r} \approx \frac{r_i r_j}{2m_r}. \tag{5.11}$$

If we assume that $p_{ij}(m_r) = \frac{r_i r_j}{2m_r}$ then the expected degree of node i in this model, the remaining degree model, is given by

$$E(\mathbf{D}_i^r) = r_i - \frac{r_i^2}{2m_r} \approx r_i = k_i - d_i. \tag{5.12}$$

Suppose that an edge between nodes i and j generated in the first round model could not be generated in the remaining degree model and vice versa (i.e. suppose the events are mutually exclusive). Then the expected degree of node i in the combined graph would just be the sum of the expected degrees from each model. However, these events are not mutually exclusive, since an edge between nodes i and j can be generated in either the

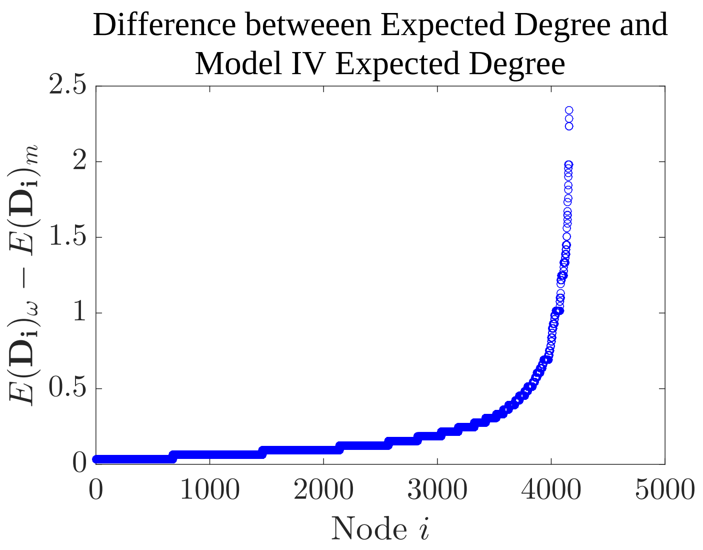


Figure 5.8: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 13832.54$ ($E(\mathbf{D}_i) = k_i$ for largest degree node) and Expected Degree from Model IV.

first round model or the remaining degree model or both. Thus,

$$\begin{aligned}
&P(\text{There is an edge between nodes } i \text{ and } j \text{ in the combined graph}) = \\
&P(\text{There is an edge between nodes } i \text{ and } j \text{ in the first round model}) + \\
&P(\text{There is an edge between nodes } i \text{ and } j \text{ in the remaining degree model}) - \\
&P(\text{There is an edge between nodes } i \text{ and } j \text{ in both the first round and the remaining degree model}).
\end{aligned} \tag{5.13}$$

Thus, the expected degree in the combined graph will be less than the sum of the expected degrees from the two models and is still strictly less than k_i so unlike the case of just adding edges the expected degree cannot overshoot the actual degree. We should note that the remaining degree distribution is based on the particular instance of the graph created using Algorithm 8 and thus will be different every time Algorithm 8 is run. The entire algorithm is summarized in Algorithm 11. Given the dependence of the remaining degree distribution on the realization of the first graph model it is difficult to analyze the actual model underlying any instance created using Algorithm 11 (i.e. finding the probability of edge (i, j) being in the graph after running the algorithm) thus we analyze Algorithm 11 numerically by using instances created from real networks. Again, we use the general relativity network and generate $l = 10000$ instances of the graph using Algorithm

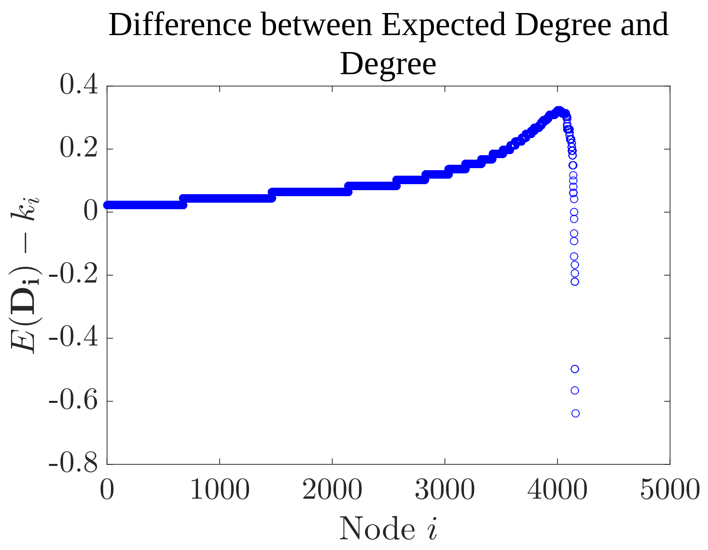


Figure 5.9: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 13720.66$ (Least-squares solution) and Actual Degree.

11. Figure 5.11 plots the difference between the average degree from $l = 10000$ instances created using Algorithm 11 and the actual degree, while 5.12 plots the difference between the average degree from $l = 10000$ instances created using Algorithm 11 and the expected degree under Model IV. From Figure 5.12 we can see that Algorithm 11 improves upon Model IV, since the average degree under Algorithm 7 is closer to the actual degree than the expected degree in Model IV is to the actual degree ($\overline{d}_i^l > E(\mathbf{D}_i)$) however it does not do as good as just simply adding edges (see Figure 5.4 where the difference between $E(\mathbf{D}_i)_\omega$ and $E(\mathbf{D}_i)$ is larger indicating that $E(\mathbf{D}_i)_\omega$ is closer to k_i than $\overline{d}_i^l$)

5.4 Optimal Probabilities

In Model IV, networks are generated by independently drawing m edges. The probability of drawing edge (i, j) on a single draw is denoted p_{ij} where $\sum_i \sum_j p_{ij} = 1$. The probability of an edge occurring between nodes i and j where $i \neq j$ in the resulting graph is given by

$$\begin{aligned} p_{ij}(m) &:= P(\text{There is an edge between nodes } i \text{ and } j \text{ in the graph after } m \text{ draws}) \\ &= 1 - (1 - p_{ij} - p_{ji})^m, \ i \neq j, \end{aligned} \tag{5.14}$$

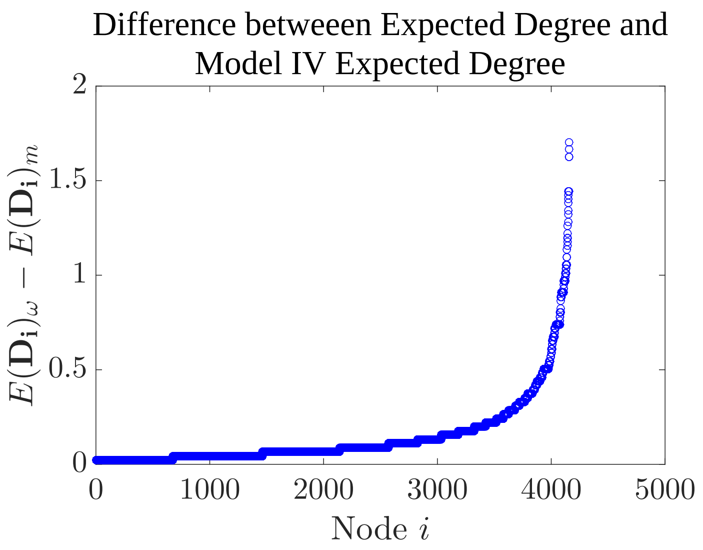


Figure 5.10: General Relativity Collaboration Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 13720.66$ (Least-squares solution) and Expected Degree from Model IV.

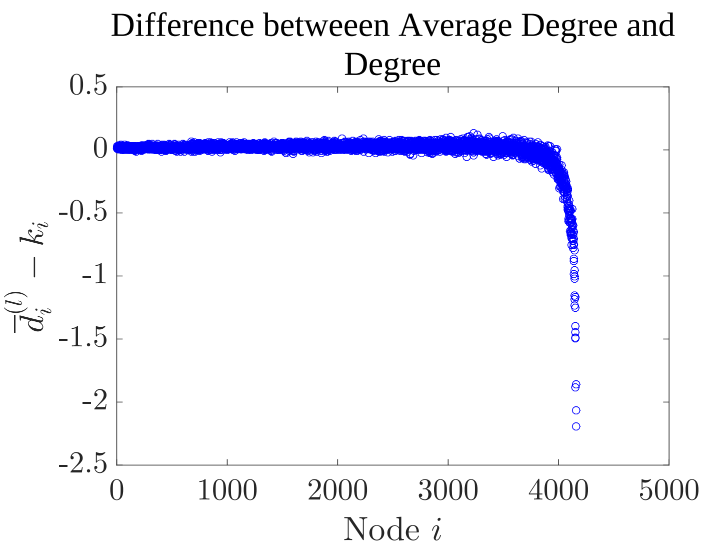


Figure 5.11: General Relativity Collaboration Network: Comparing Average Degree from Algorithm 11 and Actual Degree.

Algorithm 11: Two Distributions Chung-Lu Algorithm without Self-Edges.

```
for $k = 1$ to $m$ do
    Draw node $i$ with probability $\frac{k_i}{2m}$;
    Draw node $j$ with probability $\frac{k_j}{2m}$;
    /* Add edge $(i, j)$ to the graph */
    if $i \neq j$ then
        $a_{ij} = a_{ji} = 1$;
    end
end
for $i = 1$ to $n$ do
    Calculate $d_i = \sum_i a_{ij}$;
    Calculate $r_i = \max\{k_i - d_i, 0\}$;
end
Calculate $m_r = \frac{1}{2}\sum_i r_i$;
for $k = 1$ to $m_r$ do
    Draw node $i$ with probability $\frac{r_i}{2m_r}$;
    Draw node $j$ with probability $\frac{r_j}{2m_r}$;
    /* Add edge $(i, j)$ to the graph */
    if $i \neq j$ then
        $a_{ij} = a_{ji} = 1$;
    end
end
```

and the probability of a self-edge is given by

$$\begin{aligned} p_{ii}(m) &:= P(\text{There is a self-edge at node } i \text{ in the graph after } m \text{ draws}) \\ &= 0, \end{aligned} \tag{5.15}$$

by design. The expected degree of each node is computed as follows,

$$E(\mathbf{D}_i) = E\left(\sum_j \mathbf{A}_{ij}\right) = \sum_j E(\mathbf{A}_{ij}), \tag{5.16}$$

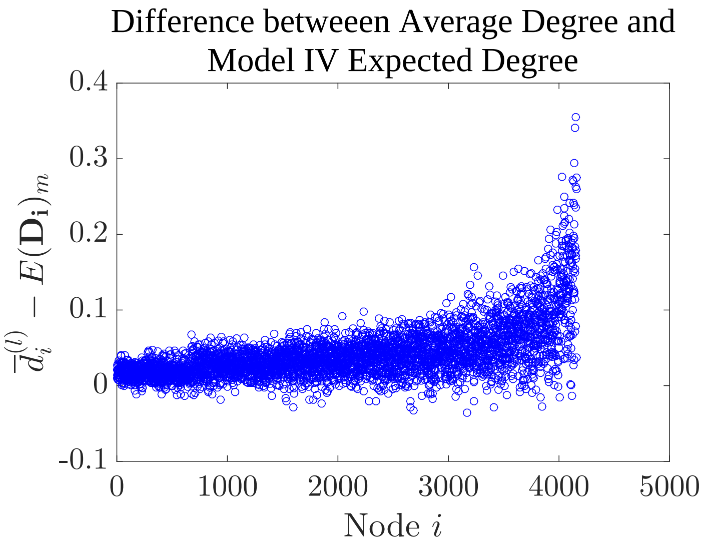


Figure 5.12: General Relativity Collaboration Network: Comparing Average Degree from Algorithm 11 and Expected Degree from Model IV.

where $E(\mathbf{A}_{ij}) = p_{ij}(m)$ if $i \neq j$ and $E(\mathbf{A}_{ii}) = 2p_{ii}(m)$. This gives us

$$\begin{aligned} E(\mathbf{D}_i) &= E\left(\sum_j \mathbf{A}_{ij}\right) = \sum_j E(\mathbf{A}_{ij}) \\ &= \sum_{j\neq i} p_{ij}(m) \\ &= \sum_{j\neq i} 1 - (1 - p_{ij} - p_{ji})^m. \end{aligned} \tag{5.17}$$

In Model IV, we assume that on each edge draw, the probability of choosing edge (i, j) is equal to $\frac{k_i k_j}{4m^2}$ and $p_{ij} = p_{ji}$. However, what if we assumed a more general form for this probability (i.e we no longer assume it is a function of node degrees). In order for the algorithm to remain $\mathcal{O}(m)$ we need the nodes to be drawn independently so we assume a general form for the probability of choosing edge (i, j) where this assumption holds. Let λ_i be the probability of choosing node i and let λ_j be the probability of drawing node j where $\sum_i \lambda_i = 1$ then $p_{ij} = \lambda_i\lambda_j$ and the expected degree can be written as

$$E(\mathbf{D}_i) = \sum_{j\neq i} 1 - (1 - 2\lambda_i\lambda_j)^m. \tag{5.18}$$

How do we determine suitable λ_i, $i = 1, \ldots, n$? One way is to match the expected degree of each node to the degree in the actual network where k_i is the degree of node i in the actual network. Thus we want to find λ_i, $i = 1, \ldots, n$ such that

$$\begin{gathered} k_i = \sum_{j \neq i} 1 - (1 - 2\lambda_i \lambda_j)^m, \quad i = 1, \ldots, n \\ \sum_i \lambda_i = 1, \quad 0 \leq \lambda_i \leq 1 \ \ \forall\, i, \end{gathered} \tag{5.19}$$

where n is the number of nodes in the network. If we ignore the constraint, $\sum_i \lambda_i = 1$, and the inequality constraints, then we can write the above as a system of n nonlinear equations and n unknowns:

$$F(\boldsymbol{\lambda}) = \mathbf{0}, \tag{5.20}$$

where

$$F_i(\boldsymbol{\lambda}) = k_i - \sum_{j \neq i} 1 - (1 - 2\lambda_i \lambda_j)^m, \quad i = 1, \ldots, n. \tag{5.21}$$

Note that we can include the equality constraint in the nonlinear system by defining $\lambda_1 = 1 - \sum_{i=2}^{n} \lambda_i$ and then substituting out λ_1. Then we have a nonlinear system of n equations and $n - 1$ unknowns, an overdetermined system.

Problem (5.19) may not have a solution however we can formulate the following constrained optimization problem

$$\min_{\boldsymbol{\lambda}} \|F(\boldsymbol{\lambda})\| \ \ s.t. \ \sum_i \lambda_i = 1, \quad 0 \leq \lambda_i \leq 1 \ \ \forall\, i, \tag{5.22}$$

where $\|F(\boldsymbol{\lambda})\|^2 = \sum_{i=1}^{n} |F_i(\boldsymbol{\lambda})|^2$, in order to find a vector $\boldsymbol{\lambda}$ that will generate graphs that approximate the degree distribution optimally in expectation, with optimality defined by the function $\|F(\boldsymbol{\lambda})\|$ in (5.22). Note that we could also choose a different functional form for our optimization function, in particular, we could weight the terms in the summation of $\|F(\boldsymbol{\lambda})\|$ (i.e. $\|F(\boldsymbol{\lambda})\|^2 = \sum_i \omega_i |F_i(\boldsymbol{\lambda})|$) to put an emphasis on high or low degree nodes, or to make the relative error uniform, etc.

Unfortunately, the optimization problem in (5.22) can be computationally expensive to solve given the large number of constraints $(2n + 1)$. Given this, our goal is not to find a solution to the optimization problem in (5.22), instead we want to find a $\boldsymbol{\lambda}$ such that the function value, $F(\boldsymbol{\lambda})$, is lower than with $\lambda_i = \frac{k_i}{2m}\ \forall\, i$. This will get us closer to an optimal solution (in terms of (5.22)) than Model IV. To do this we develop an iterative method attempting to find λ_i values such that $\|F(\boldsymbol{\lambda})\|$ is lower than $\|F(\boldsymbol{\lambda})\|$ with $\lambda_i = \frac{k_i}{2m}\ \forall\, i$. We begin by developing a fixed point method to solve $F(\boldsymbol{\lambda}) = 0$. Consider the equation for a single node, $F_i(\boldsymbol{\lambda}) = 0$, which can be re-written as follows,

$$\begin{aligned}
F_i(\boldsymbol{\lambda}) &= 0, \\
k_i - \sum_{j\neq i} 1 - (1 - 2\lambda_i\lambda_j)^m &= 0, \\
k_i - \sum_{j\neq i} \left[1 - \sum_{p=0}^{m} \binom{m}{p} (1)^p (-2\lambda_i\lambda_j)^{m-p}\right] &= 0, \\
k_i - \sum_{j\neq i} \left[1 - \left(1 + m(-2\lambda_i\lambda_j) + \sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i\lambda_j)^{m-p}\right)\right] &= 0, \\
k_i - \sum_{j\neq i} \left[2m\lambda_i\lambda_j - \sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i\lambda_j)^{m-p}\right] &= 0, \\
k_i - 2m\lambda_i \sum_{j\neq i} \lambda_j - \sum_{j\neq i}\sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i\lambda_j)^{m-p} &= 0, \\
2m\lambda_i \sum_{j\neq i} \lambda_j = k_i - \sum_{j\neq i}\sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i\lambda_j)^{m-p}&, \\
\lambda_i = \frac{k_i - \sum_{j\neq i}\sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i\lambda_j)^{m-p}}{2m\sum_{j\neq i}\lambda_j}&,
\end{aligned} \tag{5.23}$$

and from which we can get the following fixed point method

$$\lambda_i^{(q+1)} = \frac{k_i - \sum_{j<i}\sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i^{(q)}\lambda_j^{(q+1)})^{m-p} - \sum_{j>i}\sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i^{(q)}\lambda_j^{(q)})^{m-p}}{2m\left(\sum_{j<i}\lambda_j^{(q+1)} + \sum_{j>i}\lambda_j^{(q)}\right)}, \tag{5.24}$$

where we can force $\lambda_i^{(q+1)} = \lambda_i^{(q)}$ if the $\lambda_i^{(q+1)}$ given by Equation (5.24) violates the constraint $0 \leq \lambda_i^{(q+1)} \leq 1$. Using Equation (5.24) along with the constraints on λ_i, we propose the method outlined in Algorithm 12 to find an improved solution for $\boldsymbol{\lambda}$. If the constraints on $\lambda_i^{(q+1)}$ are never violated then Algorithm 12 will find a solution to $F(\boldsymbol{\lambda}) = \mathbf{0}$, however, we have no guarantee that $\sum_i \lambda_i = 1$ and by normalizing at the end we no longer have $F(\boldsymbol{\lambda}) = \mathbf{0}$; however, the idea is that we may be able to obtain a better solution (i.e. smaller

$\|F(\boldsymbol{\lambda})\|$) than if $\lambda_i = \frac{k_i}{2m}$ although there are no guarantees.

Algorithm 12: Algorithm For Finding Improved $\boldsymbol{\lambda}$

```
Input: $\boldsymbol{\lambda}^{(0)}$
$q = 0$;
Evaluate $\|F(\boldsymbol{\lambda}^{(0)})\|$;
while $\|F(\boldsymbol{\lambda}^{(q)})\| > \epsilon$ do
    for $i = 1$ to $n$ do
        $\lambda_i^{(q+1)} = \frac{k_i - \sum_{j<i} \sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i^{(q)} \lambda_j^{(q+1)})^{m-p} - \sum_{j>i} \sum_{p=2}^{m} \binom{m}{p} (1)^p (-2\lambda_i^{(q)} \lambda_j^{(q)})^{m-p}}{2m\left(\sum_{j<i} \lambda_j^{(q+1)} + \sum_{j>i} \lambda_j^{(q)}\right)}$;
        if $\lambda_i^{(q+1)} < 0$ or $\lambda_i^{(q+1)} > 1$ then
            $\lambda_i^{(q+1)} = \lambda_i^{(q)}$;
        end
    end
    $q = q + 1$;
end
$\sigma_{\boldsymbol{\lambda}} = \sum_i \lambda_i^{(q)}$;
for $i = 1$ to $n$ do
    $\overline{\lambda}_i = \frac{\lambda_i^{(q)}}{\sigma_{\boldsymbol{\lambda}}}$;
end
```

To test our algorithm we once again use the general relativity graph. We set $\lambda_i^{(0)} = \frac{k_i}{2m}\ \forall\ i$ and $\epsilon = 10^{-7}$. The algorithm requires 12 iterations to reach the desired tolerance and the inequality constraints on λ_i are never violated. However, before we normalize at the end of the algorithm we find $\sum_i \lambda_i = 1.0034$. Let $\overline{\boldsymbol{\lambda}}$ denote the improved (normalized) solution. We find that $\|F(\overline{\boldsymbol{\lambda}})\| = 4.656$ and $\|F(\boldsymbol{\lambda}^{(0)})\| = 9.876$. The maximum value of F_i, which is the difference between the actual degree of node i and the expected degree of node i goes from 2.342 to 0.534. However, the minimum value of F_i actually increases from $3.72e^{-4}$ to $6.78e^{-3}$ and if we examine the expected number of edges then we find that with $\boldsymbol{\lambda} = \boldsymbol{\lambda}^{(0)}$ the expected number of edges is 13333.55 but with $\boldsymbol{\lambda} = \overline{\boldsymbol{\lambda}}$ the expected number of edges is 13331.57 where the actual number of edges in the graph is 13422. It appears that choosing a $\boldsymbol{\lambda}$ to approximate the solution to (5.22) is better able to match the expected degree to the actual degree for higher degree nodes but at the cost of matching the expected degree to the actual degree for lower degree nodes. We can also see the difference by comparing Figures 5.13 and 5.14 where Figure 5.13 plots the difference between the expected degree using $\overline{\boldsymbol{\lambda}}$ and the actual degree while Figure 5.14 plots the difference between the expected degree with $\lambda_i = \frac{k_i}{2m}\ \forall\ i$ and the actual degree. The difference between expected degree and the actual degree is smaller with $\overline{\boldsymbol{\lambda}}$ than with $\lambda_i = \frac{k_i}{2m}\ \forall\ i$. We can also see this from Figure 5.15 which plots the difference between the expected degree using $\overline{\boldsymbol{\lambda}}$ and the expected degree with $\lambda_i = \frac{k_i}{2m}\ \forall\ i$ since this difference is positive.

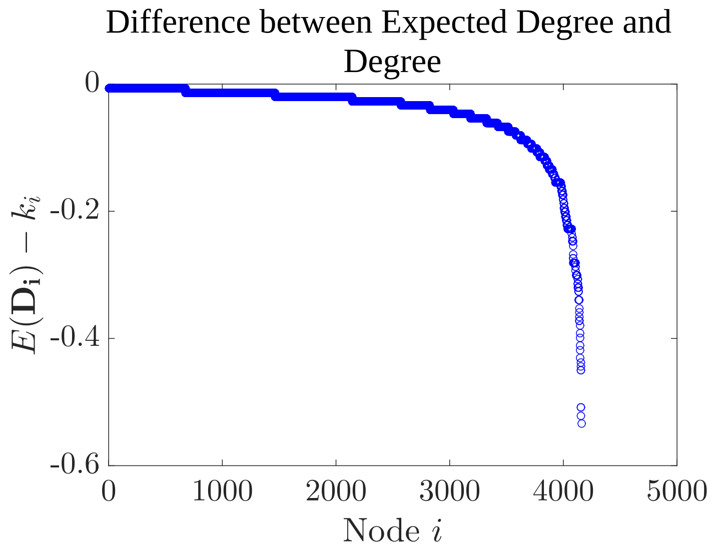


Figure 5.13: General Relativity Collaboration Network: Comparing Expected Degree with $\overline{\boldsymbol{\lambda}}$ and Actual Degree.

5.5 Corrected Fast Chung-Lu Algorithm

Pfeiffer et al. [85] also suggest an algorithm for improving the approximation error in Model IV, the MCL model without self-edges. They refer to their algorithm as the corrected Fast Chung-Lu (cFCL) algorithm. The basic idea behind their algorithm is to create a list of $2m$ nodes where each node is added to the list by drawing nodes independently with probability $\frac{k_i}{2m}$, and then pairing the nodes in the list together. If any pair of nodes is chosen more than once or a node is paired with itself then the set of nodes can be randomly permuted until no duplicate nodes are found nor any node is paired with itself. Rather than permute the entire list they suggest only permuting a few edges, the edges that are duplicates or self-edges. There graph generator is designed to replicate this process and we reproduce their algorithm in Algorithm 13. The algorithm is very similar to Algorithm 8 however when the algorithm encounters either a self-edge or a duplicate edge both nodes

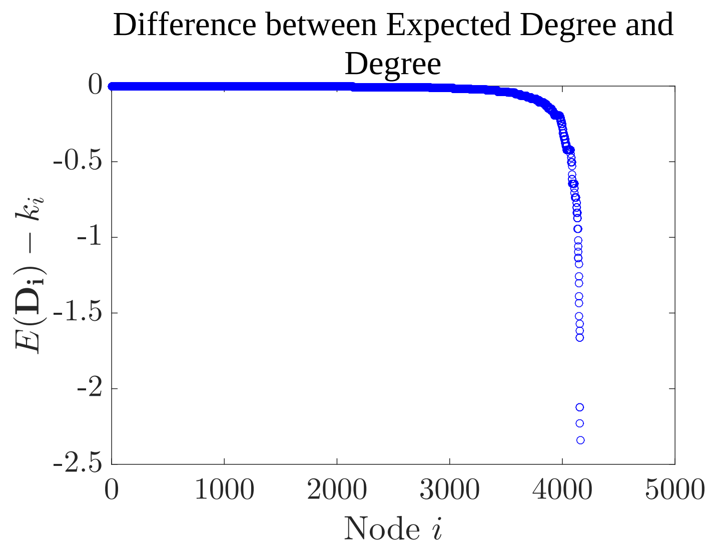


Figure 5.14: General Relativity Collaboration Network: Comparing Expected Degree from Model IV $\left(\lambda_i = \frac{k_i}{2m}\ \forall\ i\right)$ and Actual Degree.

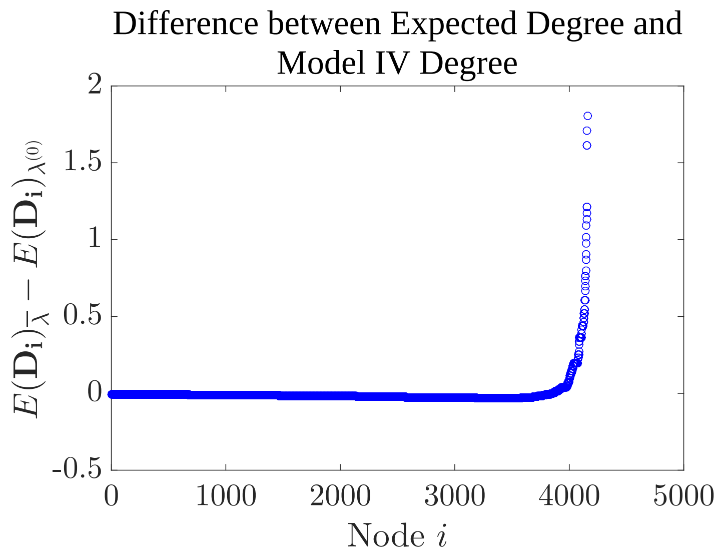


Figure 5.15: General Relativity Collaboration Network: Comparing Expected Degree with $\overline{\boldsymbol{\lambda}}$ and Expected Degree from Model IV.

are placed in a waiting queue. Then rather than continuing with regular edge insertions the algorithm attempts to select neighbours for all nodes in the queue. If a new edge for a node selected from the queue results in a duplication or a self-edge then the chosen neighbour is also placed in the queue. As the authors note, this ensures that if a node is 'due' for a new edge but is stopped from getting it due to a duplication or a self-edge then the node is 'slightly permuted' by exchanging places with a node sampled later. This process re-weights the nodes so that nodes that have been involved in a duplication or self-edge are given a higher weight to be chosen in a subsequent edge. The underlying abstract model is not explicitly defined in [85] but we can explore the properties of the model numerically by generating a large number of network instances using Algorithm 13 and then using the average degree to approximate the expected degree. We again use the general relativity network and generate $l = 10000$ instances. Figure 5.16 plots the difference between the average degree from Algorithm 13 and the actual degree while Figure 5.17 plots the difference between the average degree from Algorithm 13 and the expected degree from Model IV. The differences in Figure 5.17 are (mostly) positive suggesting that Algorithm 13 does reduce the approximation error in Model IV however some of the differences in Figure 5.16 are positive which implies that for some nodes the average degree ($\approx E(\mathbf{D}_i)$) is greater than the actual degree (i.e. Algorithm 13 may over-correct the approximation error for some nodes). We also note that the differences in Figure 5.16 are larger than in Figure 5.13 suggesting that finding the improved $\overline{\boldsymbol{\lambda}}$ provides a better approximation although there is an increased cost of finding the improved $\overline{\boldsymbol{\lambda}}$.

Algorithm 13: Corrected Fast Chung-Lu Algorithm (from [85])

```
Initialize queue;
$\delta \leftarrow 1$;
while $\delta \leq m$ do
    if queue is empty then
        Draw node $i$ with probability $\frac{k_i}{2m}$;
    else
        $i$ = pop(queue);
    end
    Draw node $j$ with probability $\frac{k_j}{2m}$;
    if $i \neq j$ and $a_{ij} = 0$ then
        $a_{ij} = a_{ji} = 1$;
        $\delta \leftarrow \delta + 1$;
    else
        push(queue,i);
        push(queue,j);
    end
end
```

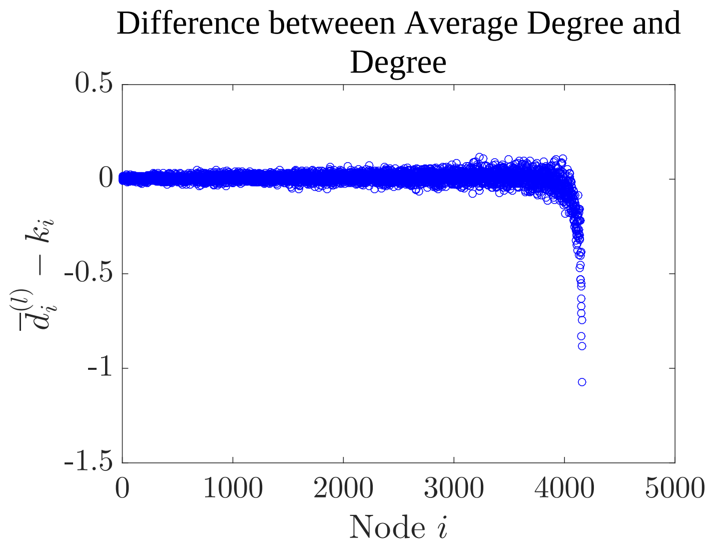


Figure 5.16: General Relativity Collaboration Network: Difference between Average Degree from Algorithm 13 ($l = 10000$) and Actual Degree.

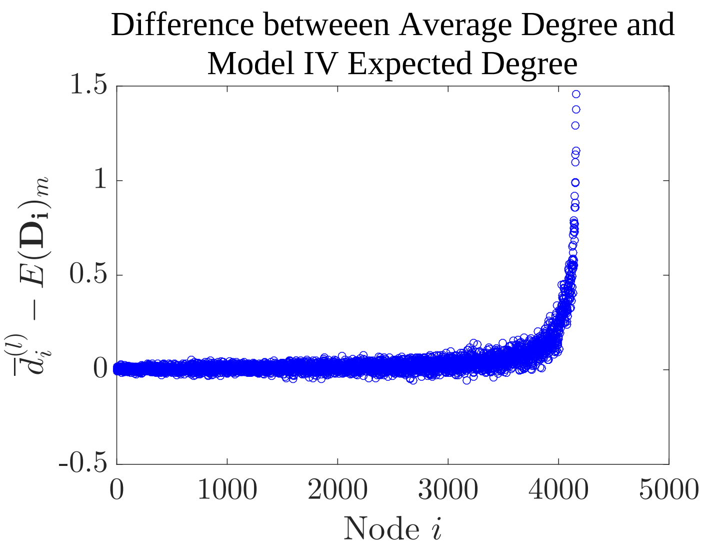


Figure 5.17: General Relativity Collaboration Network: Comparing Average Degree from Algorithm 13 ($l = 10000$) and Expected Degree from Model IV.

5.6 Constraint Violation

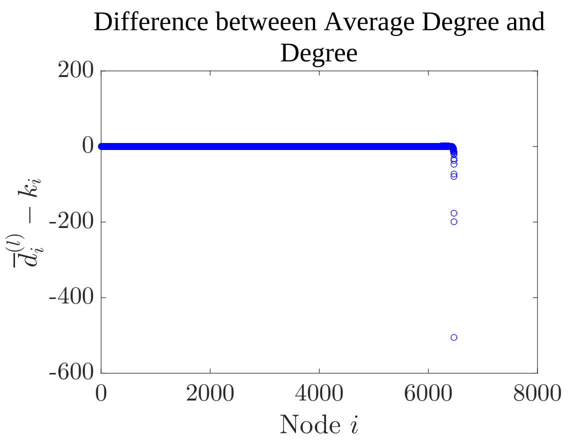
We now turn to the case where the constraint, $k_i^2 \leq 2m$, $\forall i$, is violated and look at how the different methods to improve the $\mathcal{O}(m)$ Chung-Lu model perform in this case. We begin by revisiting Algorithm 10 where each instance of the model is generated to match the edge count. Using Algorithm 10, we generate $l = 10000$ instances of the autonomous system network. Note that this network has several nodes that violate the degree constraint. Figure 5.18 plots the difference and the relative difference between the average node degree and the actual degree. Figure 5.19 plots the difference between the average node degree and the expected node degree from Model IV. As with the general relativity network, where all the degree constraints hold, increasing the number of draws to match the edge count in the original network improves the degree distribution relative to Model IV, however, in this case the difference between the average degree distribution and the actual degree distribution is still quite large. To discuss why this may be the case we revisit Algorithm 9 and the ωCL model without self-edges, since the ωCL model without self-edges with $\omega = \overline{\omega}$ can be used to approximate the model underlying Algorithm 10. Note that in the ωCL model without self-edges, the probability of edge (i, j) being in the graph after ω draws is given by Equation (5.2) even if the degree constraint is violated. We mentioned previously that $p_{ij}(\omega)$ is strictly less than $\frac{\omega}{m}\frac{k_i k_j}{2m}$. In networks where the degree constraint is violated, we can refine this as $p_{ij}(\omega) = \min\{1, \frac{\omega}{m}\frac{k_i k_j}{2m}\}$. For each node the expected degree is just the sum of these probabilities (i.e. $E(\mathbf{D}_i) = \sum_{j \neq i} p_{ij}(\omega)$). Ideally, we want $p_{ij}(\omega)$ to contribute $\frac{k_i k_j}{2m}$ to the expected degree (Note that in the case where self-edges are not allowed, we actually want this contribution to be larger). However, if $\frac{k_i k_j}{2m} > 1$ then the contribution from $p_{ij}(\omega)$ is 1 and the other $p_{ij}(\omega)$ must make up this difference. If this difference is large this may require ω to be large however we don't want to increase ω too much as that will increase the total number of edges in the graph. In the autonomous system graph, the largest degree node has degree 1458. For this node, if we look at all the nodes j where $\frac{k_i k_j}{2m} > 1$ and look at the sum of the difference between $\frac{k_i k_j}{2m}$ and 1 we get

$$\sum_{j \in J} \left(\frac{k_i k_j}{2m} - 1 \right) = 641.83 \tag{5.25}$$

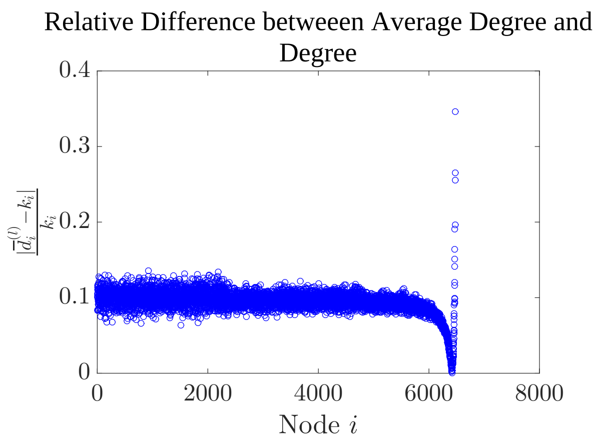
where J is the set of all nodes where $\frac{k_i k_j}{2m} > 1$. This sum is over a third of the total node degree for this node and thus the remaining $p_{ij}(\omega)$ have to contribute this amount to $E(\mathbf{D}_i)$ which explains why even increasing the number of draws may not be enough to match the degree distribution. Also note that increasing $p_{ij}(\omega)$ to match the expected degree for some nodes will increase the expected degree above the actual degree for other nodes.

Returning to the autonomous system network we note that the instances generated using Algorithm 10 had an average ω of 13890.12 and a standard deviation of 41.875. Compare this with the actual number of edges in the network which is 12572. So for this network, on average, we have to draw over 1000 more edges to match the total number of edges. Figures 5.20 and 5.21 plot the results from Algorithm 9 with $\overline{\omega} = 13890.12$ and reflect the patterns found in Figures 5.18 and 5.19. Note that the largest difference between the expected degree in the ωCL model with $\omega = \overline{\omega} = 13890.12$ and the average degree from the instances generated from Algorithm 10 with $l = 10000$ is 0.2437 suggesting that the ωCL model with $\omega = \overline{\omega} = 13890.12$ approximates the model underlying Algorithm 10 well. This can also be seen from the figures, since Figure 5.18a and 5.20 are nearly identical as are Figures 5.19 and 5.21.

Our next step is to analyze the autonomous system network and the ωCL model without self-edges with ω chosen to match certain expected values. We begin by finding ω to match the expected number of edges with the actual number of edges in the network ($E(\mathbf{M}) = m$). For the autonomous system network, we use the formula for $E(\mathbf{M})$ given by Equation (5.5) and set it equal to m. We then solve this equation numerically to find $\omega = 13889.77 \approx 13890$. This is approximately the same ω as the average ω calculated from the instances generated by Algorithm 10. Thus, Figures 5.20 and 5.21 can be used to show the difference between the expected degree and the actual degree and the expected degree



(a) $l = 10000$



(b) $l = 1000$

Figure 5.18: Autonomous System Network: Difference and Relative difference between Average Degree from Algorithm 10 and Actual Degree.

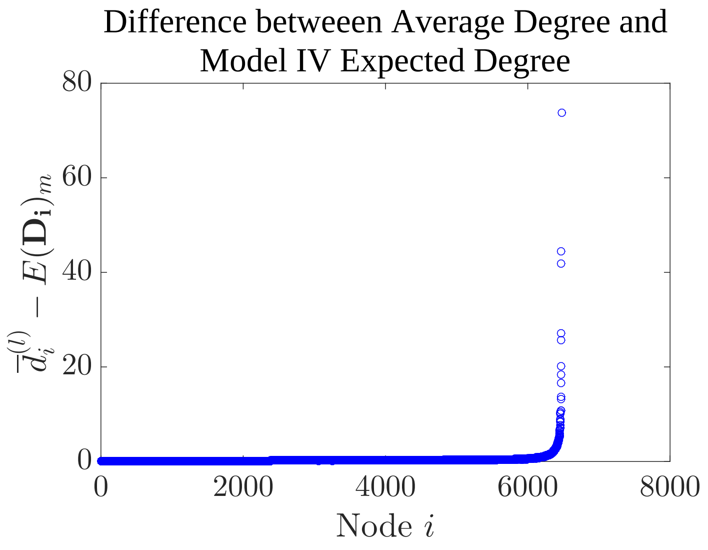


Figure 5.19: Autonomous System Network: Comparing Average Degree from Algorithm 10 and Expected Degree from Model IV.

and the expected degree from Model IV. We do note that under this model, the expected degree of the highest degree node is approximately 879, well under the actually degree of

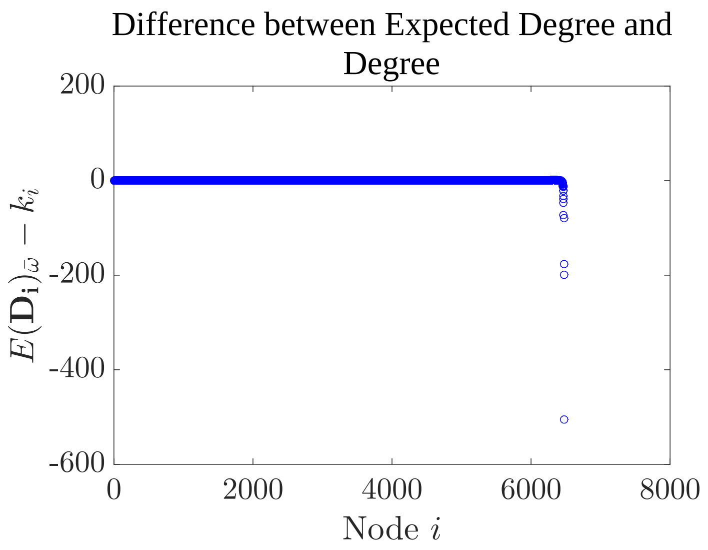


Figure 5.20: Autonomous System Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = \overline{\omega} = 13890.12$ and Actual Degree.

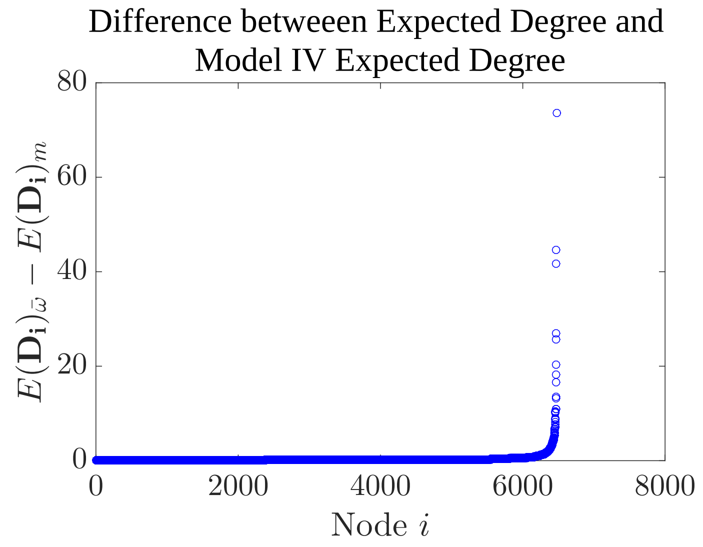


Figure 5.21: Autonomous System Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = \overline{\omega} = 13890.12$ and Expected Degree from Model IV.

1458. Given this we could also choose ω to match the expected degree of the largest degree node. In this case, for the autonomous system network, we find $\omega = 23937.66 \approx 23938$. This is almost double the number of edges in the original graph and the expected number of edges under this model is 21085.51. Figures 5.22 and 5.23 plot the difference between the expected degree and the actual degree and the difference between the expected degree and the expected degree from Model IV, respectively. With $\omega = 23937.66$, the expected degree for a large number of nodes is much greater than the actual degree. This makes it an unattractive choice for ω in practice. The remaining case for ω that we previously examined is the case where ω is chosen to match the probabilities (in the least-squares sense). The ω we found was the least squares solution to a system of equations. However, when the degree constraint is violated, $\ln\left(1 - \frac{k_i k_j}{2m}\right)$ is not defined for some nodes i and j since $1 - \frac{k_i k_j}{2m} < 1$ for some nodes i and j. Thus we cannot calculate this ω for the autonomous system network or any network where the degree constraint is violated.

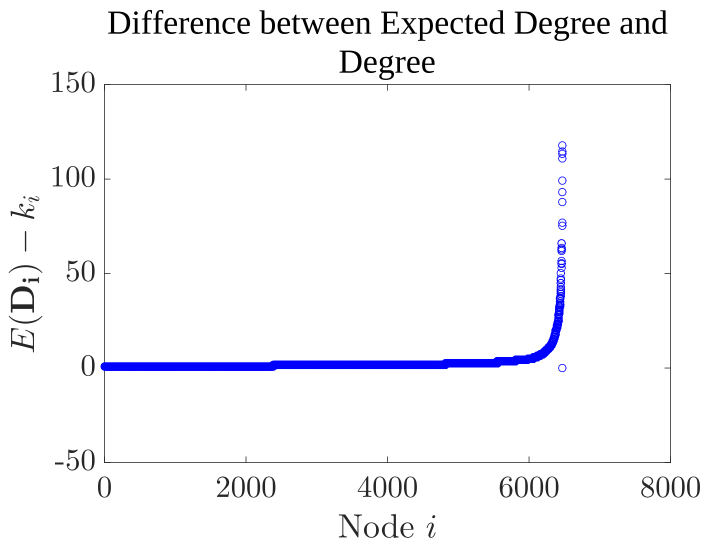


Figure 5.22: Autonomous System Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 23937.66$ ($E(\mathbf{D}_i) = k_i$ for the largest degree node) and Actual Degree.

Next, we use the autonomous system network to examine the two distributions model described by Algorithm 11. Using Algorithm 11, we generate $l = 10000$ instances of the network. Since we don't know the underlying model (i.e. we don't know the probability of edge (i, j) being in the graph after running Algorithm 11), we can't analyze the model analytically. However, we can look at the average values for different properties calculated

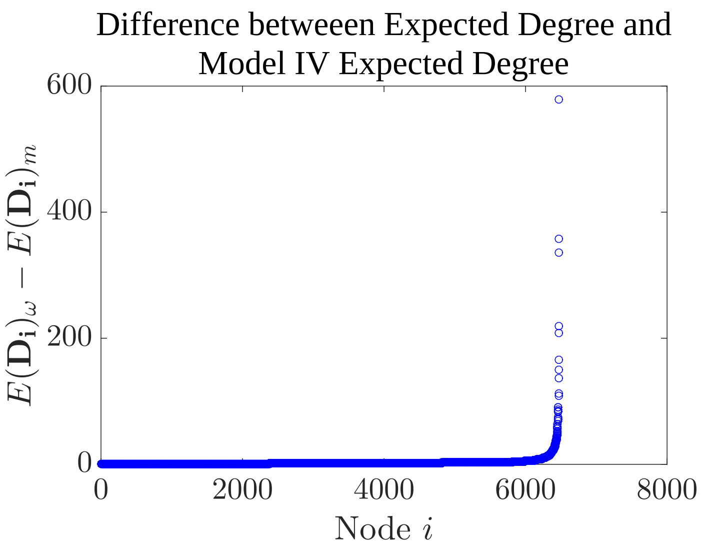


Figure 5.23: Autonomous System Network: Comparing Expected Degree from the ωCL model without self-edges with $\omega = 23937.66$ ($E(\mathbf{D}_i) = k_i$ for the largest degree node) and Expected Degree from Model IV.

from the $l = 10000$ instances. Figure 5.24 plots the difference between the average degree and the actual degree and Figure 5.25 plots the difference between the average degree and the expected degree from Model IV. While the average degree is still much lower than the actual degree for the larger degree nodes Algorithm 11 performs better than Algorithm 10 which can be seen by comparing Figures 5.25 and 5.19. The difference, $\overline{d}_i^{(l)} - E(\mathbf{D}_i)_{(m)}$, is larger for Algorithm 11 than Algorithm 10 and since the average node degree for both algorithms is less than the actual degree this implies that the average node degree is closer to the actual degree for Algorithm 11. Note that this can also be seen by comparing Figures 5.24 and 5.18a.

Next, we use Algorithm 12 on the autonomous system network to find probabilities λ_i which will better match the expected degree to the actual degree using the optimization function in (5.22). We set $\lambda_i^{(0)} = \frac{k_i}{2m}$ $\forall$ i and $\epsilon = 10^{-7}$. For the autonomous system network, the algorithm requires 58 iterations to reach the desired tolerance; however, once again the inequality constraints on λ_i are never violated. Before we normalize at the end of the algorithm we find $\sum_i \lambda_i = 1.12$. For this network, $\|F(\boldsymbol{\lambda}^{(0)})\| = 694.25$ and $\|F(\overline{\boldsymbol{\lambda}})\| = 342.42$. The maximum value of F_i is given by the largest degree node and goes from 578.84 to 233.35 but the minimum value of F_i increases from 0.00327 to 0.2035. As well, we see the same pattern with the expected number of edges as we saw with the

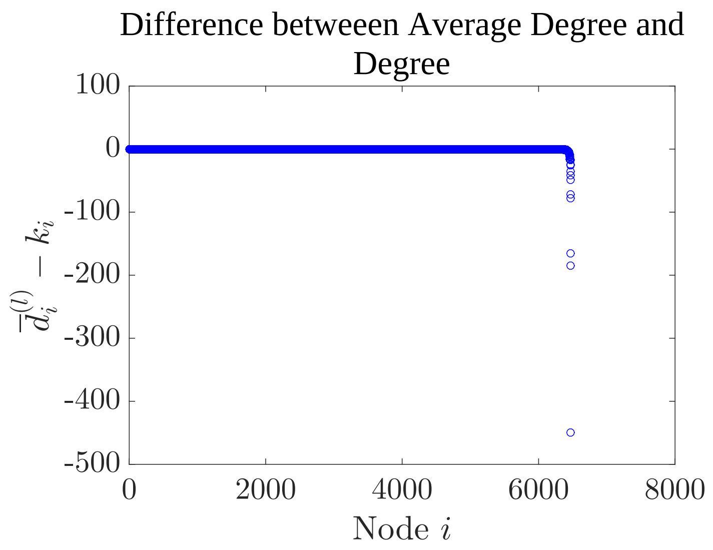


Figure 5.24: Autonomous System Network: Comparing Average Degree from Algorithm 11 and Actual Degree.

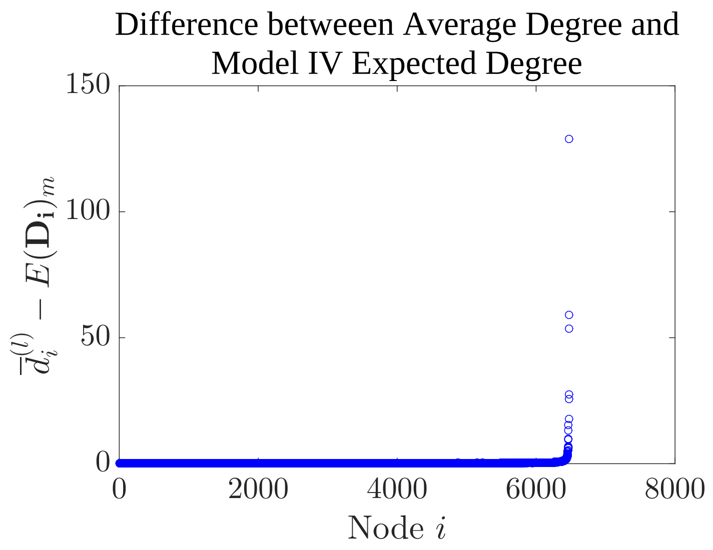


Figure 5.25: Autonomous System Network: Comparing Average Degree from Algorithm 11 and Expected Degree from Model IV.

general relativity network. The expected number of edges with $\boldsymbol{\lambda} = \boldsymbol{\lambda}^{(0)}$ is 11429.50 but with $\boldsymbol{\lambda} = \overline{\boldsymbol{\lambda}}$ the expected number of edges is 10166.16 where the actual number of edges in the graph is 12572. Again, it appears that by choosing $\overline{\boldsymbol{\lambda}}$ we are better able to match the expected degree to the actual degree for higher degree nodes but this comes at the cost of matching the expected degree to the actual degree for lower degree nodes. Figure 5.26 plots the difference between the expected degree using $\overline{\boldsymbol{\lambda}}$ and the actual degree while Figure 5.27 plots the difference between the expected degree with $\lambda_i = \frac{k_i}{2m}\ \forall\ i$ (Model IV) and the actual degree. Figure 5.28 plots the difference between the expected degree using $\overline{\boldsymbol{\lambda}}$ and the expected degree with $\lambda_i = \frac{k_i}{2m}$ which is the expected degree from Model IV. From this figure we can see that Algorithm 12 performs better than both Algorithm 11 and 10 since for the large degree nodes $|E(\mathbf{D}_i) - k_i|$ from Algorithm 12 is smaller than $|\overline{d}_i^{(l)} - k_i|$ from both Algorithms refAlgorithm:TwoDistributions and 10. However, there is still a large difference between the expected degree and the actual degree for some nodes.

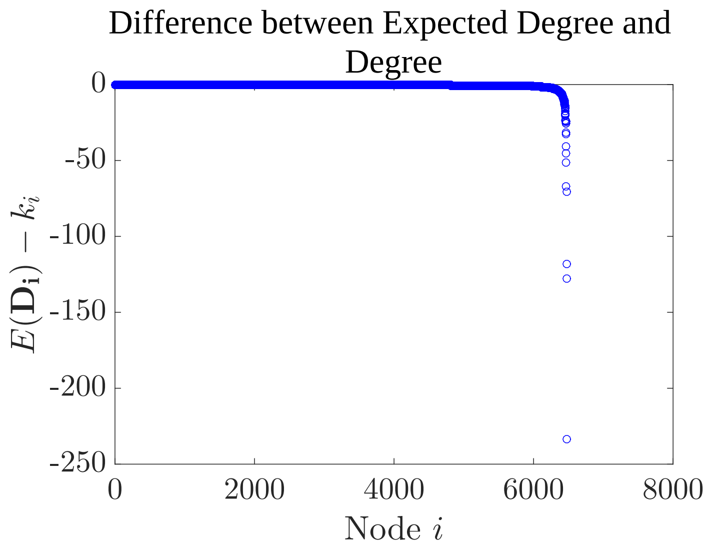


Figure 5.26: Autonomous System Network: Comparing Expected Degree with $\overline{\boldsymbol{\lambda}}$ and Actual Degree.

Finally we use Algorithm 13 on the autonomous system network and generate, $l = 10000$, instances to examine the improvement suggested by [85] in the case where the degree constraint is violated. Figure 5.29 plots the difference between the average degree from Algorithm 13 and the actual degree while Figure 5.30 plots the difference between the average degree from Algorithm 13 and the expected degree from Model IV. We see an improvement in the approximation error since the differences in Figure 5.30 are positive; however, this improvement is not as large as with the improved $\overline{\boldsymbol{\lambda}}$ as can see by comparing

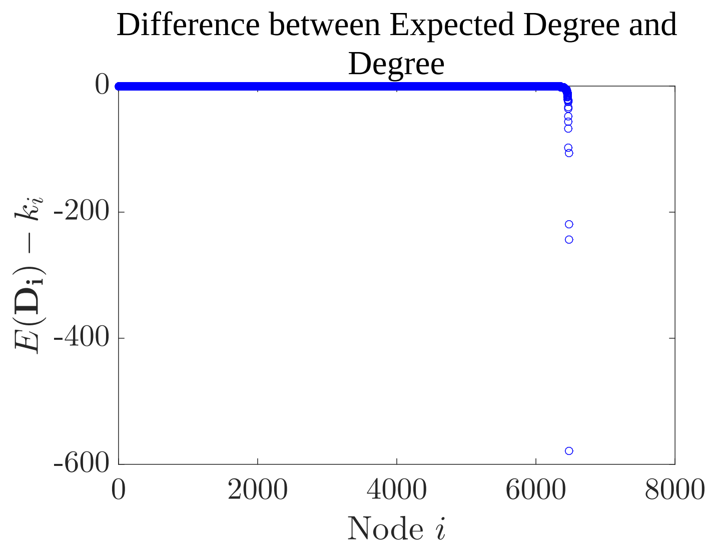


Figure 5.27: Autonomous System Network: Comparing Expected Degree from Model IV $\left(\lambda_i = \frac{k_i}{2m}\ \forall\, i\right)$ and Actual Degree.

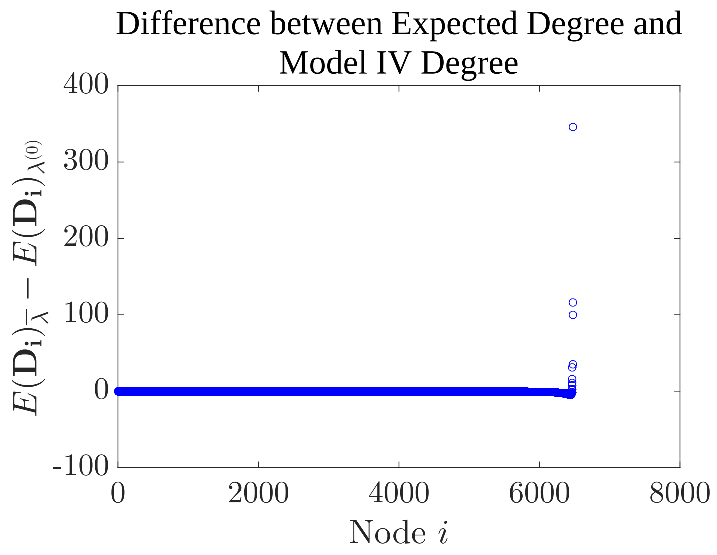


Figure 5.28: Autonomous System Network: Comparing Expected Degree with $\overline{\boldsymbol{\lambda}}$ and Expected Degree from Model IV.

Figures 5.29 and 5.26 where the differences in Figure 5.26 are smaller than in Figure 5.29.

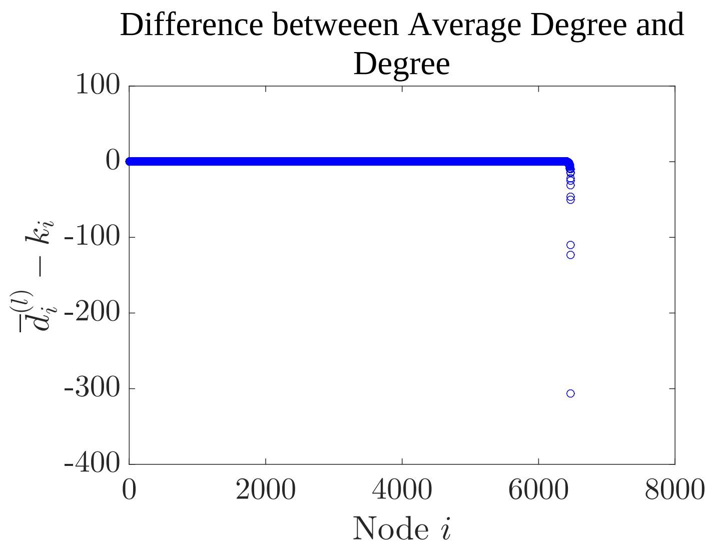


Figure 5.29: Autonomous System Network: Difference between Average Degree from Algorithm 13 ($l = 10000$) and Actual Degree.

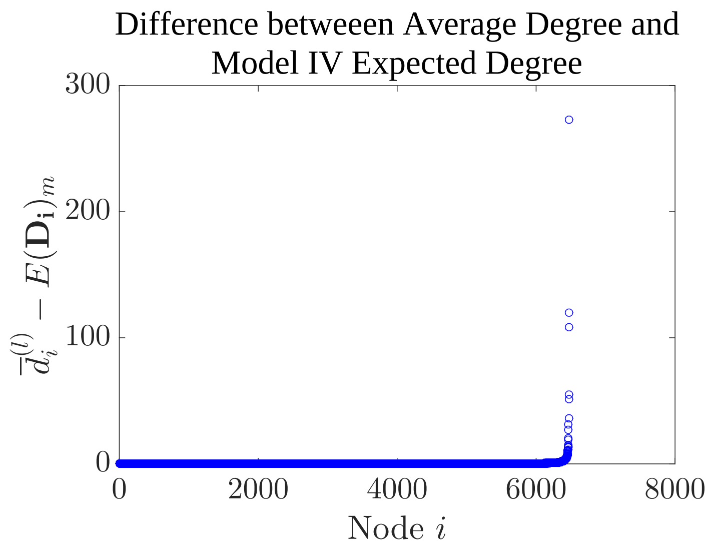


Figure 5.30: Autonomous System Network: Comparing Average Degree from Algorithm 13 ($l = 10000$) and Expected Degree from Model IV.

5.7 Conclusion

We examined some simple ways of modifying $\mathcal{O}(m)$ Chung-Lu model without self-edges to better match the expected degree distribution to the actual degree distribution. In the case where the degree constraint is not violated, the $\mathcal{O}(m)$ Chung-Lu model, Model IV, may not be a bad approximation to begin with but by drawing more edges we can improve the degree distribution for some nodes with insufficient edges although this comes at the expense of overestimating the degree distribution for other nodes. Using the two distributions algorithm, Algorithm 11, also improves the distribution, albeit by less than by simply adding edges for the network we examined, but it does not overestimate the distribution of other nodes. When the degree constraint is violated drawing more edges again improves the degree distribution but is still well under the desired value for large degree nodes. Using Algorithm 11 in this case does a better job of matching the distribution and still underestimates the number of edges which allows for the possibility of adding more edges. Algorithm 12 (improving $\boldsymbol{\lambda}$ for Model IV) underestimates the number of edges and performs better than Algorithm 11 in both the case where the degree constraints hold and when they are violated while Algorithm 13 also performs well in both cases. However, in the case where the constraint on the input degree sequence is violated the modifications to the $\mathcal{O}(m)$ Chung-Lu model still leave large differences between the expected degree and the actual degree for some nodes. This suggests along with all our previous analysis that when the degree constraint is violated, the Chung-Lu model is not a good choice for a null model. If we were to use the Chung-Lu model as a basis to compare other graph models to and we suppose (incorrectly) that the expected degree is equal to the degree to derive other model properties for the Chung-Lu model such as the clustering coefficient or assortativity then any comparison of these properties between the Chung-Lu model and other graph models is incorrect.

Chapter 6

A Clustering Graph Generator

6.1 Introduction

In the previous two chapters we examined the Chung-Lu model as well as approximate Chung-Lu models in detail. As we have noted previously, the Chung-Lu model is a very simple model and doesn't capture many of the important properties of real networks we are interested in. In this chapter, we develop a graph generator to capture more of these properties. As with any graph generator, capturing the essential properties of a real-world network is important since we can use the synthetic networks generated from a graph generator to not only better understand the network under investigation but synthetic graphs can also be used to test graph analysis algorithms, such as clustering algorithms and anomaly detection algorithms.

One network property of particular interest is the network community structure. In a network, a community can broadly be defined as a collection of nodes that share a large number of connections internally with many fewer connections to nodes outside the collection [80, p. 357]. The size and number of communities in a network can provide insight into the underlying network. As well, each community itself can be examined for patterns and properties and since the communities are smaller than the overall network this makes the analysis easier. However, the study of network community structure can often be difficult since a community can be quantified in many different ways [80, Chapter 11]. One way to evaluate community structure is to look at the number of triangles in a graph and in particular the number of triangles each node is a part of. Thus, there are a number of graph generators that try to replicate the triangle structure of real networks including the models developed by Seshadhri, Kolda and Pinar [94, 58] and Gutfraind, Meyers and Safro

[43]. Another measure of community structure is the assortativity coefficient which we will discuss in Section 6.2. Mussmann et al. [76] use the observed assortativity of a network to develop a graph generator that matches the assortativity and thereby captures the community structure of the network. In the graph generator we develop, we use a different approach to model the community structure: we use a clustering algorithm as part of our graph generator. We begin by applying a clustering algorithm to the real-world network we are trying to model. We then use the resulting clusters, as well as the subgraphs induced by these clusters, as inputs into our graph generator. This additional information is used to help develop a graph generator that better preserves the community structure compared to a graph generator focused only on the triangle structure. In our graph generator, we separate the between cluster and within cluster behavior and use techniques to generate random graphs with prescribed degree sequences to build our synthetic networks. Using inputs from a real-world network, we then look at the properties of a synthetic network generated from our algorithm and compare these properties to the real network properties as well as to the properties of another popular graph generator, BTER, developed by Seshadhri, Kolda and Pinar [94, 58]. In particular, we focus on comparing two measures of community structure, the clustering coefficient and the assortativity coefficient. We find that our graph generator does well at preserving the clustering coefficient and typically outperforms BTER in matching the assortativity coefficient, particularly when the assortativity coefficient is negative. As we will show in Section 6.5, our graph generator uses an edge switching algorithm as one of the building blocks of the algorithm. In particular, the edge switching algorithm is applied to each of the clusters. So it is also of interest to compare instances generated by our graph generator to instances generated by the edge switching algorithm applied to the entire network. For many networks, applying the edge switching algorithm to the entire graph greatly reduces the clustering coefficient; however, this is not seen in instances generated by our graph generator where the clustering coefficient remains close to the original clustering coefficient. As well, the edge switching algorithm can change the assortativity coefficient by a large amount, but this behaviour is once again not seen in instances generated by our graph generator. This suggests that by applying the edge switching algorithm to only the clusters themselves we can preserve important aspects of the community structure. Before presenting our graph generating algorithm and our numerical results, we first discuss some of the interesting properties common to many networks of interest, including some properties we have already examined in the context of the Chung-Lu model. Then we introduce some of the prevailing graph generators and present the clustering algorithm, the Louvain algorithm, we will be using to generate clusters. After presenting our algorithm and numerical results we conclude by looking at avenues for future research.

6.2 Network Properties

We refer to the networks we are interested in as "interaction networks" [94]. This describes a wide variety of networks including social networks, citation networks, collaboration networks, computer traffic networks, and gene regulation networks. Despite their diverse nature these interaction networks share many properties, including degree distribution, diameter, clustering coefficient and assortativity coefficient. We consider each of these properties in turn and describe the similarities across different interaction networks. In the discussion that follows we only consider undirected graphs (i.e. networks where edges do not have a direction).

In interaction networks, one of the most commonly examined properties is the degree distribution. The degree of a node or vertex in an undirected graph is the number of edges connected to it, thus the degree distribution simply describes the distribution of degrees in the graph. The degree distribution of a graph follows a power law distribution if the number of nodes N_k with degree k approximately satisfies

$$N_k \propto k^{-\gamma}, \tag{6.1}$$

where $\gamma > 0$ is called the power law exponent. So, graphs that have a power-law degree distribution have a lot of nodes with small degree and only a small number of nodes with large degree. Most real-world networks have a power law degree distribution. Power laws have been found in the Internet [35], the Web [16, 53], citation graphs [91], online social networks [20] and many others. Thus, any graph generator that we develop should be able to generate synthetic networks that have a power-law degree distribution.

Perhaps one of the most well known properties common to many real-world networks is known as the "small-world" property. One way to measure this property is to use the diameter of a graph. A graph is said to have diameter D when D is the smallest integer such that every pair of nodes can be connected by a path of length at most D edges. The diameter of most real-world graphs is relatively small suggesting that most nodes in the graph are relatively close to one another hence the "small-world" description. However, the diameter can be affected by outliers so we define the integer effective diameter to measure the pair-wise distance between nodes. The integer effective diameter is the minimum number of steps in which some fraction of all connected pairs of nodes can reach each other. The effective diameter has been found to be small for large real-world graphs, like the Internet, the Web, and online social networks [6, 63, 66].

Since we are particularly interested in the community structure of networks we examine one of the measures most commonly used to determine community structure, the clustering coefficient. As mentioned previously there are many ways to quantify a community in a network and thus there are many measures of community structure in a network. We present the clustering coefficient since it is one of the most popular measures for determining community structure. The clustering coefficient is defined as

$$C = \frac{3 \times \text{Total Number of Triangles}}{\text{Total Number of Wedges}}, \tag{6.2}$$

where a *wedge* is a path of length 2 and $0 \leq C \leq 1$. If connections between vertices are made at random such that the probability of an edge occurring is a function of the node degree (i.e the probability of an edge between nodes i and j is $\frac{k_i k_j}{2m}$), then the clustering coefficient takes the value

$$C = \frac{1}{n} \frac{\left[\langle k^2 \rangle - \langle k \rangle\right]^2}{\langle k \rangle^3}, \tag{6.3}$$

where $\langle k^m \rangle = \frac{1}{n} \sum_{i=1}^{n} k_i^m$ and k_i is the degree of node i [80, p. 449]. For example, if we assume that $\langle k^2 \rangle$ and $\langle k \rangle$ have fixed finite values the clustering coefficient becomes small as $n \to \infty$, thus we can expect the clustering coefficient to be very small in large graphs assuming the connections between vertices are made at random. However, the typical value for C is between 0.1 and 0.5 in many real-world networks [94] which is much larger than suggested by Equation (6.3).

We can also define the local clustering coefficient for each node as

$$C_i = \frac{\text{Number of Pairs of Neighbors of } i \text{ that are Connected}}{\text{Number of Pairs of Neighbors of } i}. \tag{6.4}$$

The local clustering coefficient is often dependent on degree, where vertices with higher degree have a lower local clustering coefficient on average [80, p. 265].

Another popular metric used to analyze networks is known as assortative mixing and in particular assortative mixing by degree which is the tendency for vertices to connect to other vertices with similar degree to their own. One way to measure assortative mixing is through the assortativity coefficient. To define the assortativity coefficient we must first define some preliminary variables. Let p_k be the probability that a randomly chosen vertex in the graph will have degree k. Now, suppose we consider a vertex reached by following a randomly chosen edge in the graph. Note that the degree of this vertex is not distributed according to p_k; instead, it is biased in favor of vertices of high degree. This is because more edges end at a high-degree vertex than at a low-degree one so the degree distribution for the vertex at the end of a randomly chosen edge is proportional to kp_k rather than p_k. We now define the remaining degree as the number of edges leaving the vertex other than the one we arrived along. It is one less than the total degree and distributed proportional to $(k+1)p_{k+1}$. The normalized distribution, q_k, of the remaining degree is

$$q_k = \frac{(k+1)p_{k+1}}{\sum_j jp_j}. \tag{6.5}$$

Let the quantity e_{jk} be defined as the the joint probability distribution of the remaining degrees of two vertices at either end of a randomly chosen edge. This quantity is symmetric on an undirected graph, $e_{jk} = e_{kj}$, and obeys the sum rules

$$\sum_{jk} e_{jk} = 1, \tag{6.6}$$

and

$$\sum_{j} e_{jk} = q_k. \tag{6.7}$$

Note that if there is no assortative mixing in a network then $e_{jk} = q_j q_k$ and if there is assortative mixing e_{jk} will differ from this value. The amount of assortative mixing can be quantified by the connected degree-degree correlation function given by

$$\sum_{jk} jk(e_{jk} - q_k q_j). \tag{6.8}$$

The value of this function is zero when there is no assortative mixing, positive for assortative mixing and negative for disassortative mixing. If we want to compare different networks we can normalize this value by its maximal value (i.e. the value it achieves on a perfectly assortative network). This value is equal to the variance $\sigma_q^2 = \sum_k k^2 q_k - [\sum_k kq_k]^2$. This gives us the following assortativity coefficient

$$r = \frac{1}{\sigma_q^2} \sum_{jk} jk(e_{jk} - q_k q_j), \tag{6.9}$$

Note that r can be shown to be a Pearson correlation coefficient so the values of r range from -1 to 1. As mentioned, a network has perfect assortative mixing if $r = 1$, is non-assortative if $r = 0$ and is completely disassortative (i.e. vertices connect to others with very different degrees) if $r = -1$. In [80, p. 267] it is noted that social networks tend to have positive r values while technological, information and biological networks tend to have negative r values.

As with the clustering coefficient we can look at assortative mixing at the local level. The local assortativity [87] is defined as

$$\rho_i = \frac{j(j+1)(\overline{k} - \mu_q)}{2m\sigma_q^2}. \tag{6.10}$$

where j is the remaining degree of node i, $\overline{k}$ is the average remaining degree of node i's neighbors, μ_q is the mean of the remaining degree distribution q_k and σ_q^2 is its variance.

The properties highlighted above are often targeted when building graph generators. In the next section we outline some of the current graph generators being used to replicate real-world graphs including some we have discussed previously.

6.3 Existing Graph Generators

The most well-known graph generating model is widely attributed to Erdős and Rényi [33] and is often referred to as the "Erdős-Rényi model" or "Erdős-Rényi random graph". In the Erdős-Rényi (ER) graph each pair of nodes has an identical, independent probability of sharing an edge. The ER graph provably violates the degree distribution power law found in real-world networks. In fact, it can be shown that the degree distribution follows a Poisson distribution which is why this graph is also referred to as the "Poisson random graph". For this reason, Erdős-Rényi graphs are rarely used to model real-world networks.

In the ER graph model every edge has an equal probability of occurring and the expected degree of each node is the same. To overcome this limitation, Chung and Lu [22, 23] introduced a graph model where the expected degree distribution is specified as noted in the previous chapters. So, Chung-Lu graphs can produce graphs with power law degree distributions. However, the generated graphs have small clustering coefficients relative to real-world networks. We should also note that the ER graph model is equivalent to the Chung-Lu graph model with degree distribution $k = (pn, pn, \ldots, pn)$ where p is the probability of nodes sharing an edge in the ER model and n is the number of nodes in the graph.

Similar to the Chung-Lu model we have the graph model that consists of the ensemble of all graphs with a prescribed degree sequence where each graph in the ensemble is equally likely [12, 68, 71, 72, 78]. In the Chung-Lu model, only the expected degree of a node is equal to the given degree and in any particular instance, the degree of a node may actually differ from the given degree, however, in the model consisting of the ensemble of all graphs with a prescribed degree sequence, in every instance, the node degree matches the given node degree. Two common algorithms for generating instances of this model are the switching algorithm [69, 79, 90, 51] and the matching algorithm [71, 78, 69]. We use both algorithms in our graph generator and we will discuss these algorithms in more detail when we present our graph generator. We should note however, that neither the matching nor the switching algorithm generate each graph from the ensemble with uniform probability and thus only approximate the model.

Next, we look at some of the more complex graph generators used to model interaction networks. The Stochastic Kronecker Graph Model [20] also known as R-MAT has proved to be one of the most successful generative strategies for modeling real-world networks and is used as the graph generator for the Graph 500 Supercomputer Benchmark [41]. R-MAT generates a graph by recursively partitioning the adjacency matrix. There are typically four partitions with the probabilities of an edge falling into one of these partitions given by a, b, c, d where $a + b + c + d = 1$. Typically, $a \geq b$, $a \geq c$, $a \geq d$ and often $b = c$. To see how an edge is placed in the R-MAT model consider the adjacency matrix in Figure 6.1. Initially, the matrix is divided into four parts. Partition b is then randomly chosen according to the probabilities a, b, c, d and then divided into four parts. From there partition a is randomly chosen again according to the probabilities a, b, c, d and divided into four parts. This process continues recursively until we actually reach the finest level of the adjacency matrix at which point we can add an edge to the graph by setting the value in the partition to 1. In the above explanation of the R-MAT model the probabilities a, b, c and d are the same at each level of the partition. In the actual R-MAT algorithm an element of randomness is introduced so that these values vary slightly for each level of the partition. The randomness is introduced to better match the properties of real-world networks and while the R-MAT model can generate graphs with power-law degree distributions, it does not generate graphs with high clustering coefficients [86].

The Block Two-Level Erdős-Rényi (BTER) model [94, 58] is designed to capture the underlying community structure in real-world networks. BTER graphs contain a scale free collection of dense ER subgraphs which the authors suggest is the underlying structure of most real-world networks. The BTER graph takes as an input the degree distribution and the clustering coefficient distribution (i.e. for each node degree the average clustering coefficient) and is able to match both the degree distribution and the local and global clustering coefficients of real-world networks fairly well. However, in BTER graphs nodes of degree d have neighbors with degree $\approx d$. Yet, there is more variation in the joint degree distribution of real-world graphs. Communities in real-world networks are not as homogeneous as depicted by the BTER graph.

There are many more graph generators than mentioned above. Chakrabarti and Falout-

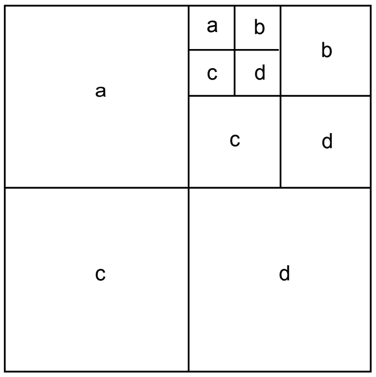


Figure 6.1: The R-MAT Model

sos [19] provide a survey of some other models. They include preferential attachment [5], small-world models [102], copying models [60] and forest fire [61]. Both BTER and R-MAT are generative models. There is another class of network generators, known as graph editing models. In these models, we start with a given network and randomly change its components until the network becomes sufficiently different from the original network. Thus, we are able to introduce variability while preserving key structural properties. The simplest example of this type of generator is the switching algorithm mentioned previously. One notable graph editing method is known as MUSKETEER [43]. The method uses a multilevel approach to build graphs that preserve some of the original network structural properties while introducing realistic variability and is able to retain many of the real-world network properties.

6.4 The Louvain Clustering Algorithm

To build a graph generator via clustering we must first choose an algorithm to cluster our real-world networks. While our graph generating algorithm is designed to use clusters from any clustering algorithm we choose to use the Louvain algorithm [13] as our initial clustering method. The Louvain algorithm finds hierarchical community structure using a heuristic modularity optimization technique. Since communities within any network can loosely be defined as densely connected nodes that share few connections with nodes outside the community any community detection or clustering algorithm will attempt to partition a network into clusters or communities of densely connected nodes where the number of connections between clusters is small. One measure of partitioning success is defined by modularity, $Q \in [0, 1]$:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j), \tag{6.11}$$

where A_{ij} represents the edge weight between node i and j, $k_i = \sum_j A_{ij}$ is the sum of edge weights adjacent to vertex i, c_i is the community node i belongs to, the δ-function $\delta(c_i, c_j)$ is 1 if nodes i and j belong to the same community and 0 otherwise and $m = \frac{1}{2} \sum_{ij} A_{ij}$ is the sum of all the edge weights. Not only is modularity used to compare the quality of partitions obtained by different clustering algorithms, where a higher modularity indicates a better clustering algorithm, but it can also be used as an objective function to optimize in designing a community detection algorithm. However, exact modularity optimization is a computationally hard problem [15]; thus, many algorithms, including the Louvain algorithm, use a heuristic approach to modularity optimization.

The Louvain algorithm can be divided into two steps that are repeated iteratively to build hierarchical clusters. In the first step of the algorithm, each node in the graph is assigned to its own community. Each node is then considered sequentially and for each node i we calculate the change in modularity from moving node i to each of its neighboring communities. A neighboring community is any community that contains a neighbor of node i. Thus if node i has m neighbors we must calculate at most m changes in modularity. Once all of the modularity calculations have been completed we determine whether or not to move node i to a new community based on the largest of these changes. If the largest change in modularity is positive we move node i to the community with the largest modularity change; otherwise, node i remains in its original cluster. Once all of the nodes have been considered we repeat this process until no nodes can be moved into new communities and thus no gains in modularity can be made. It should be noted that the order in which nodes are considered does change the algorithm's output; however, in practice, it does not seem to significantly affect the modularity obtained [13].

After the first step of the algorithm has been completed we move to the second step which consists of building a new network. Each node in the new network now represents a cluster found during the first step of the algorithm. The edges within a cluster are represented by self-loops where the edge weight is determined by the sum of the within-cluster edge weights. The edges between nodes in the new graph are determined by the edges between clusters and the edge weights are determined by the corresponding between-cluster edge weight sums. Once we have constructed the new graph we can repeat these two steps and the process can be repeated until no new clusters can be formed. At this point the algorithm stops. Figure 6.2 shows the results of applying the Louvain algorithm on a very simple graph.

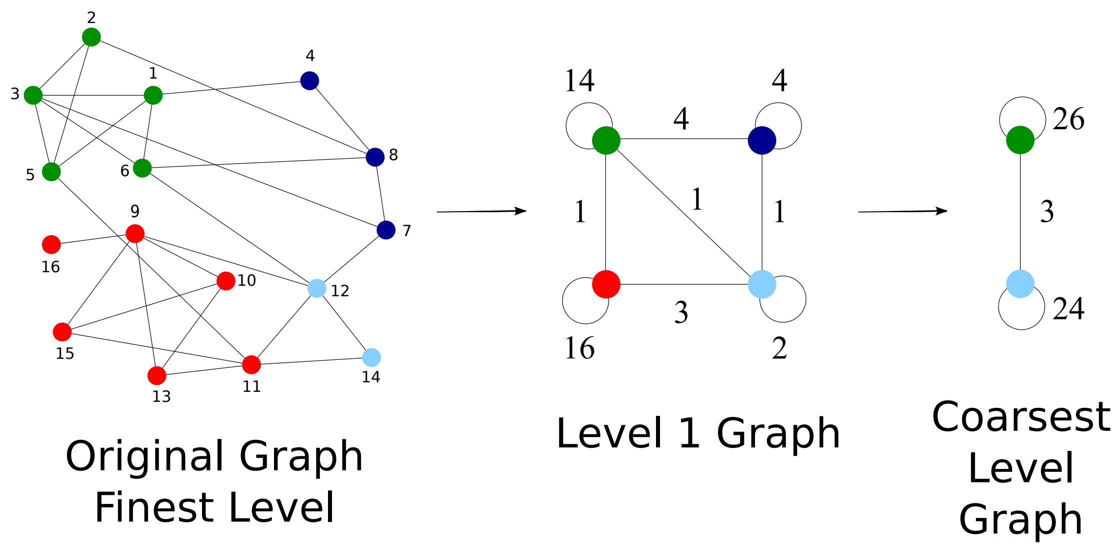


Figure 6.2: Simple Example of Louvain Clustering

6.5 The Clustering Graph Generator

As mentioned previously, our graph generator is built using the clusters generated from a clustering algorithm where we use the Louvain algorithm described in the previous section to generate clusters. We generate edges between clusters differently than edges within clusters, where we use more information to build the graph within a cluster than between clusters. Between clusters we use a modified version of the matching algorithm [71, 78, 69] also known as the configuration model. In the original matching algorithm each node is assigned a set of "stubs", which are sawn-off ends of the edges, according to the desired degree sequence. Then, pairs of stubs are chosen at random to create the edges of the network. If a self-edge or multiple edge is chosen then the entire network is discarded and the entire process starts over. Alternatively, the method can be modified so that if a multiple edge or self-edge is drawn then the network is not discarded, we simply draw another pair of stubs. The network is only discarded if no further progress is possible (i.e. all the remaining stubs would either create self-edges or multiple-edges). To better illustrate how we use a modified matching algorithm to create edges between clusters we consider the graph in Figure 6.2. Figure 6.2 depicts the original graph, and the hierarchical clusters generated by the Louvain algorithm where in the original graph the numbers represent node labels and in the level 1 and coarsest level graphs the numbers represent edge weights. For our purposes we are only interested in the original graph and the level 1 graph. The first step in our between-clusters algorithm is to calculate the external degree for each node. The external degree for node i is the number of edges between node i and any node j where node j does not belong to the same cluster as node i. For example, node 12 has external degree 4. Once the external degree for each node has been calculated we use it to create a list of stubs for each cluster where each node appears in the list with multiplicity equal to its external degree. If we let the green nodes represent cluster 1, the dark blue nodes cluster 2, the red nodes cluster 3 and the light blue nodes cluster 4, then we have the following stub lists: $C_1 = \{1, 2, 3, 5, 6, 6\}, C_2 = \{4, 7, 7, 8, 8\}, C_3 = \{9, 11, 11, 11\}, C_4 = \{12, 12, 12, 12, 14\}$. We can use these lists to create the edges but before we add an edge we must first decide what clusters any given edge will be between. To do this we use the level 1 graph given in Figure 6.2. The weights on the edges in this graph indicate how many edges there are between the clusters and we use these weights to calculate the probabilities of placing an edge between cluster I and J. The probability that an edge will be between cluster I and J is given as follows

$$P(I, J) = \frac{\text{Number of edges between clusters } I \text{ and } J}{\text{Total number of external edges}}. \tag{6.12}$$

In our example graph, $P(1, 2) = \frac{4}{10}$. Once we have chosen which clusters to place an edge between then we can randomly choose a value from C_I and C_J. If no edge exists yet between this pair of nodes then we add this edge to our generated graph and remove these values from C_I and C_J. For example, if on our first draw we chose to place an edge between clusters 1 and 2 and then chose 6 from C_1 and 7 from C_2 we would add edge (6,7) to our generated graph and C_1 and C_2 will now be $C_1 = \{1, 2, 3, 5, 6\}$, and $C_2 = \{4, 7, 8, 8\}$. We continue to add edges in this way. Ideally we would only have to repeat this process e times where e is equal to the number of external edges but because we can get duplicate edges we complete $\alpha * e$ draws where $\alpha \geq 1$. As well, we note that there is always the possibility that when we complete this procedure some unassigned values remain in the C_I lists although the number should be small. Note, we do not restart the algorithm in this case. Algorithm 14 summarizes the algorithm for adding edges between clusters described above.

Algorithm 14: Between Clusters Algorithm

```
for $I = 1$ to $nClusters$ do
    Create a list of stubs, $C_I$, based on the external degree of each node in the cluster;
    for $J = I + 1$ to $nClusters$ do
        Calculate $P(I, J) = \frac{\text{Number of edges between clusters } I \text{ and } J}{\text{Total number of external edges}}$;
    end
end
for $i = 1$ to $\alpha \cdot nExternalEdges$ do
    Choose the two clusters $I$ and $J$ where $I < J$ to put an edge between with probability $P(I, J)$;
    Randomly choose stub $p$ from $C_I$;
    Randomly choose stub $k$ form $C_J$;
    if Edge $(p, k)$ isn't in the graph then
        Add edge $(p, k)$ to the graph;
        Remove $p$ from $C_I$;
        Remove $k$ from $C_J$;
    end
end
```

The algorithm for adding edges within a cluster is based on the switching algorithm [69, 79, 90, 51]. For each cluster, we create a graph from the cluster nodes and the original internal cluster edges and apply the switching algorithm to this graph. The switching algorithm uses a Markov chain to generate a random graph with a given degree sequence [79]. To describe the switching algorithm suppose we have a graph G. We start with the original graph, G, and proceed to carry out a series of Monte Carlo switching steps. In each step a pair of edges (i, j), and (k, l) is selected at random and the ends of the edges are switched to give the edges (i, l), and (k, j). If this switch does not lead to self-edges or multiple edges then the switch is performed; otherwise it is not performed. The entire process is performed $\alpha \cdot e$ times where e is the number of edges in G and α is chosen large enough so that the Markov chain shows good mixing (i.e. is "close" to its steady state distribution). For our purposes, a Markov chain shows good mixing if the properties of interest are not changing. This switching algorithm is performed on each of the clusters to generate all the edges within clusters. Algorithm 15 describes the above algorithm. Our graph generator combines Algorithms 14 and 15 to generate synthetic networks. Note that we could have considered using a matching algorithm or alternatively the Chung-Lu model within the clusters to generate edges instead of the switching algorithm. In both these cases we would then only need the internal cluster degree for each node as opposed to the entire subgraph induced by the cluster as is needed for the switching algorithm. However, for the real-world networks we considered, many of the clusters had a small number of nodes with large internal degree (i.e. hubs) and many nodes with small internal degree. The constraint $k_i^2 \leq 2m \; \forall \; i$ where k_i is the internal cluster degree of node i and m is the sum of the internal degrees within the cluster was often violated. For the Chung-Lu model, we know that if this constraint is violated, then the underlying model is not in fact the original Chung-Lu model but an approximate (extended) Chung-Lu model where the expected degree can be quite far from the actual degree, thus making the Chung-Lu model an unattractive choice for modeling the clusters. In the case of the matching algorithm, if the constraint is violated then it can be difficult for the algorithm to run to completion without having to restart many times. This is what makes the switching algorithm attractive despite the need for more information than in either the matching algorithm or the Chung-Lu model.

Algorithm 15: Within Clusters Algorithm

```
for $I = 1$ to $nClusters$ do
    Create a graph, $G_I$, from the nodes and internal edges of cluster $I$;
    for $i = 1$ to $\alpha \cdot nInternalEdges$ do
        Choose two edges $(i, j)$, and $(k, l)$ randomly from $G_I$;
        if $i \neq l$ and $k \neq j$ and edge $(i, l)$ and $(k, j)$ are not in $G_I$ then
            Remove edges $(i, j)$, and $(k, l)$ from $G_I$;
            Add edges $(i, l)$, and $(k, j)$ to $G_I$;
        end
    end
end
```

6.6 Experimental Studies

We test our graph generator on various networks including including two collaboration networks (ca-GrQc, ca-AstroPh), one citation network (cit-HepPh), a technological network (as-735) and a social network (soc-Epinions)[62]. All five networks are treated as undirected graphs. Table 6.1 lists the basic attributes of the graphs including the number of clusters generated by the first level of the Louvain algorithm.

We evaluate our graph generator using an instance generated from our clustering graph

Graph Name	Nodes	Edges	Number of Clusters
ca-GrQc	4158	13422	868
ca-AstroPh	18772	198080	2061
cit-HepPh	34546	210789	895
as-735	6474	12572	964
soc-Epinions	75879	811480	8481

Table 6.1: Basic Attributes of the Networks

generator (CGG) algorithm. Note that for all the graphs, in all clusters, $\alpha = 100$ in the switching algorithm, Algorithm 15, to ensure good mixing. We found through numerical experiments that both the assortativity coefficient and the global clustering coefficient within the clusters had reached an equilibrium with $\alpha = 100$. We begin by comparing the properties of the instance generated by the CGG to the original network and to an instance generated by the BTER graph generator using the code from the feastpack package [57] where both the degree distribution and the clustering coefficient distribution of the original network are used as inputs. The two properties we are most interested in comparing are the clustering coefficient and the assortativity coefficient. Tables 6.2 and 6.3 list the global clustering coefficient and the assortativity coefficient for each of the models, respectively. Figures 6.3, 6.4, 6.5, 6.6 and 6.7 plot the degree distribution, the local clustering coefficient distribution and the local assortativity distribution of the synthetic networks for each of the networks under study. From Table 6.2 we can see that the CGG tends to underestimate the global clustering coefficient while BTER overestimates if for some graphs, sometimes by a factor of 2 or more. The CGG does a much better job of matching the assortativity coefficient which can be seen from Table 6.3. This can also be seen in Figures 6.3c, 6.4c, 6.5c and 6.6c. In the case of the citation network, cit-HepPh, and the social network, soc-Epinions, CGG is able to match the sign of the assortativity coefficient whereas BTER is not.

As we saw in Section 6.5, the edge switching algorithm, Algorithm 15, is used on each of the clusters. The edge switching algorithm when applied to the entire network is a means of generating a random graph with prescribed degree sequence. We are interested

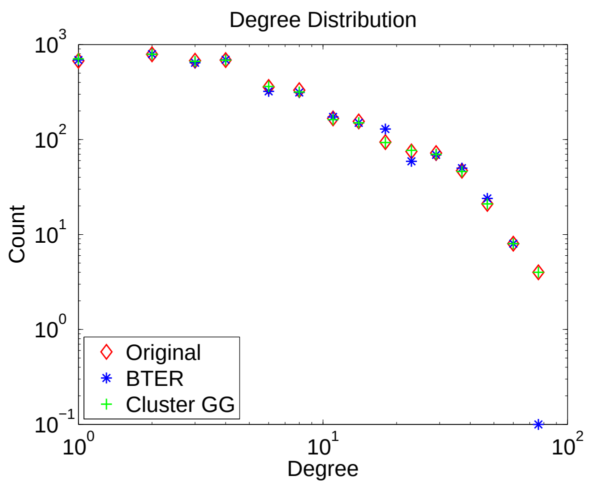
Graph Name	Original	CGG	BTER
ca-GrQc	0.629	0.514	0.531
ca-AstroPh	0.318	0.108	0.326
cit-HepPh	0.146	0.016	0.154
as-735	0.01	0.016	0.06
soc-Epinions	0.066	0.016	0.107

Table 6.2: Global Clustering Coefficient of Each Model.

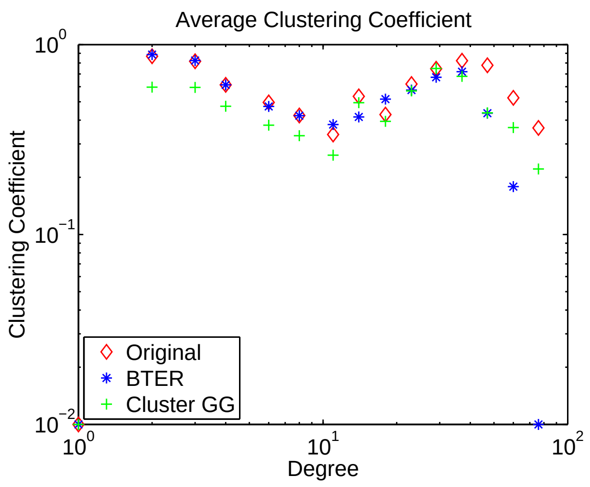
Graph Name	Original	CGG	BTER
ca-GrQc	0.63919	0.57355	0.64715
ca-AstroPh	0.20513	0.12293	0.46496
cit-HepPh	-0.00629	-0.03341	0.28205
as-735	-0.18176	-0.16105	-0.08315
soc-Epinions	-0.04065	-0.09630	0.11137

Table 6.3: Assortativity Coefficient of Each Model.

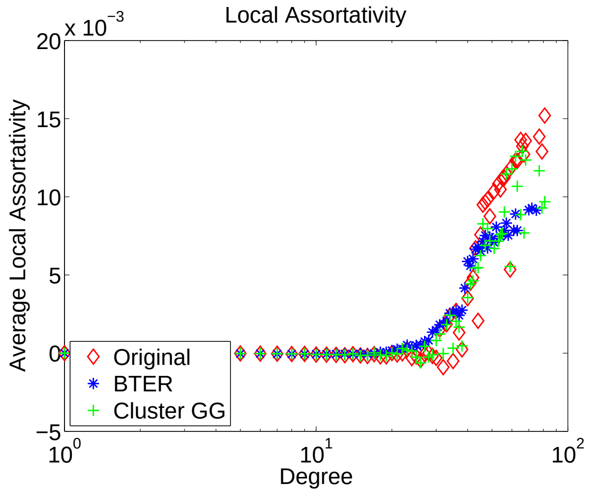
in comparing an instance generated by the CGG to a random graph generated using the edge switching algorithm to examine how restricting the edge switching algorithm to the clusters affects the properties of a generated instance. The matching algorithm when applied to the entire network could also be used as another means of generating a random graph with prescribed degree sequence. Similarly, the Chung-Lu model and the associated $O(n^2)$ algorithm can generate a random graph with an expected degree sequence equal to the prescribed degree sequence. So alternatively, we could use these algorithms to generate an instance to compare with an instance from the CGG. In Section 6.5 we examined



(a) Degree Distribution



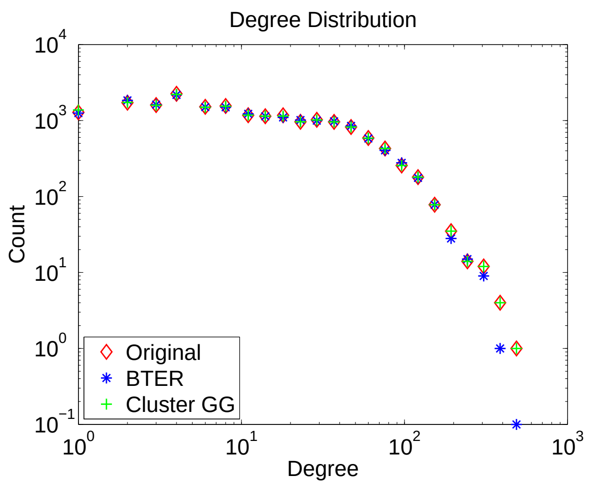
(b) Average Clustering Coefficient



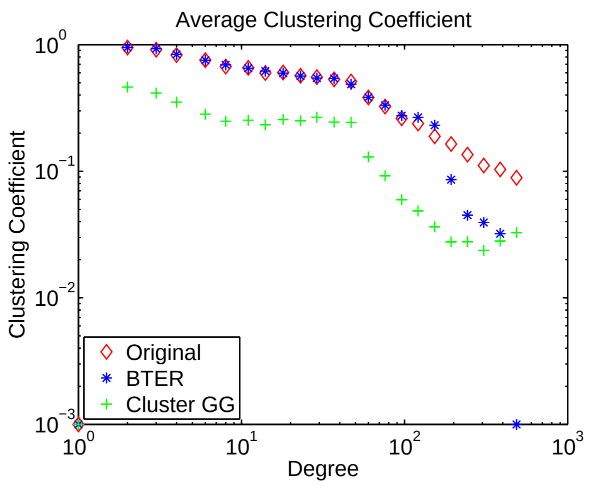
(c) Local Assortativity

Figure 6.3: Comparison of Properties of Various Models for the ca-GrQc Network.

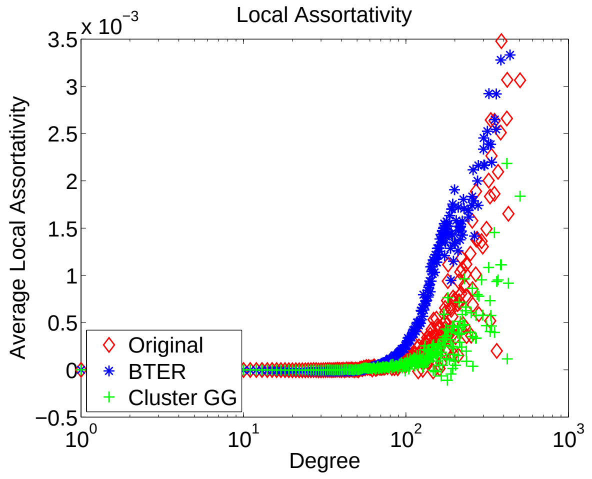
why these two algorithms may not perform as well as the switching algorithm when the constraint $k_i^2 \leq 2m \; \forall \; i$ does not hold. As well, if we wish to draw uniformly from the set of all networks with a prescribed degree sequence then as noted in [69] the switching algorithm performs better than the matching algorithm. Given these reasons we use the edge switching algorithm to generate instances with a prescribed degree sequence to compare with instances from the CGG. Since the switching algorithm is a Markov chain method, we



(a) Degree Distribution



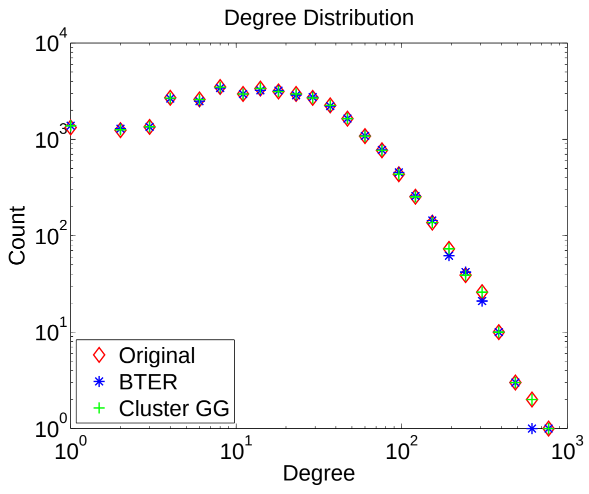
(b) Average Clustering Coefficient



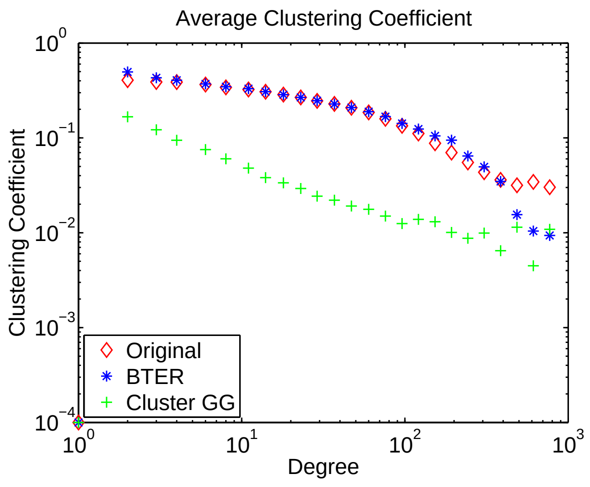
(c) Local Assortativity

Figure 6.4: Comparison of Properties of Various Models for the ca-AstroPh Network.

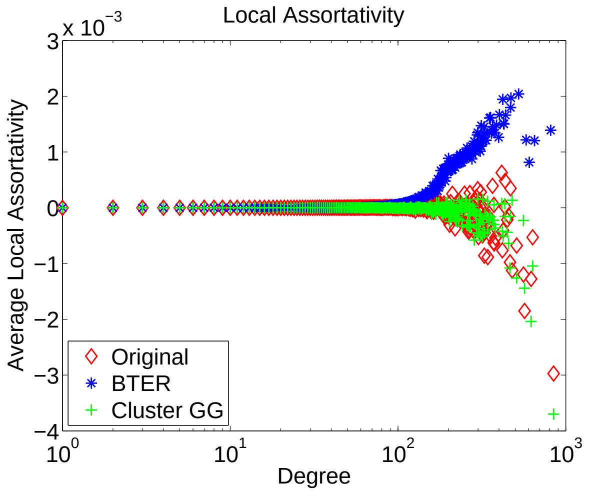
need to ensure that we perform enough switches to get good mixing. For all the graphs, we found empirically that $\alpha = 100$ worked well. In other words, if we attempt $100 \cdot$(Number of edges) switches then the properties of interest (i.e. clustering coefficient and assortativity coefficient) have converged to steady-state values for the graphs we considered. For each graph, Table 6.4 lists the global clustering coefficient and the global assortativity coefficient for the instance generated by the edge switching algorithm as well as the values for the



(a) Degree Distribution



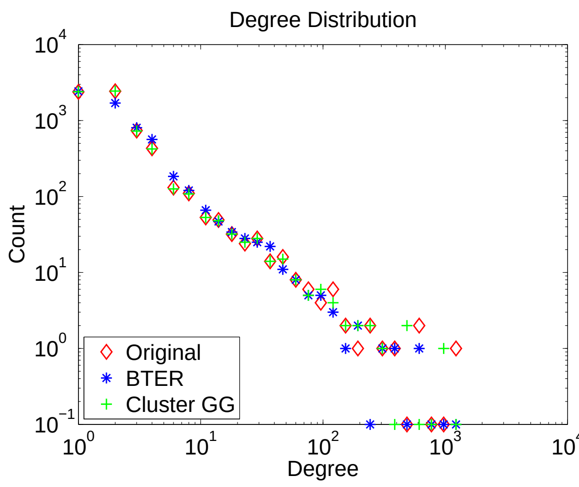
(b) Average Clustering Coefficient



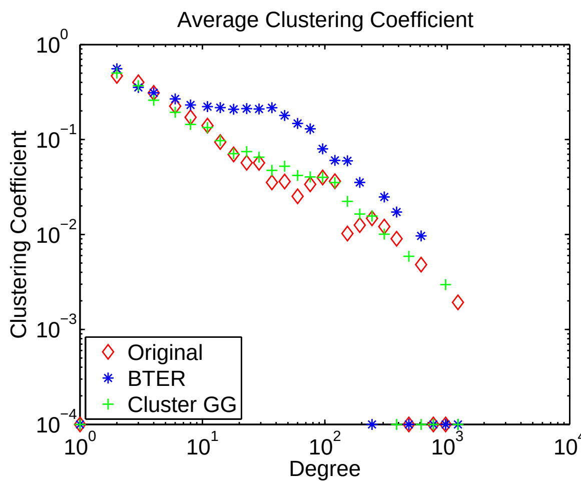
(c) Local Assortativity

Figure 6.5: Comparison of Properties of Various Models for the cit-HepPh Network.

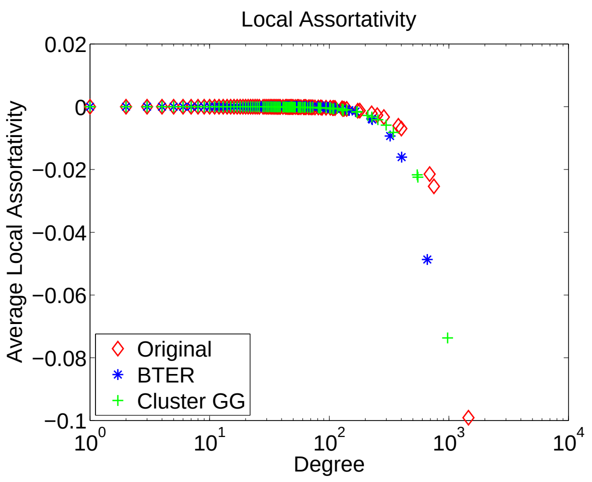
CGG instance and the real-world network. For all the the graphs, the global clustering coefficient in the edge-switching instance is near zero. However, in the CGG instance the global clustering coefficient is generally lower than in the real network but is still relatively close. As well, in the edge-switching instance all the assortativity coefficients are negative and in general further from the value in the real network than in the CGG instance. This suggests that applying the edge switching algorithm to the clusters as opposed to the entire



(a) Degree Distribution



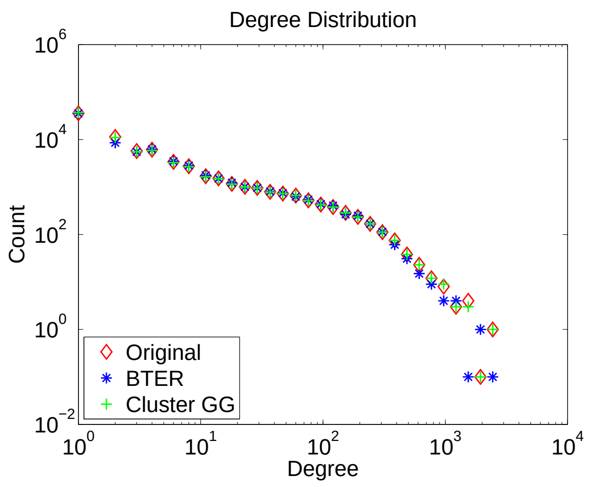
(b) Average Clustering Coefficient



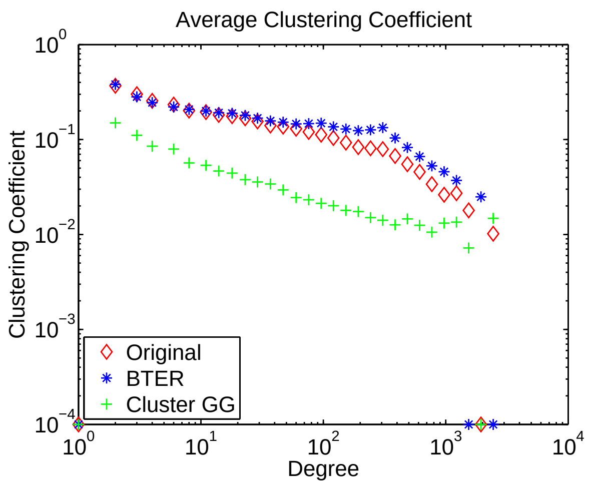
(c) Local Assortativity

Figure 6.6: Comparison of Properties of Various Models for the as-735 Network.

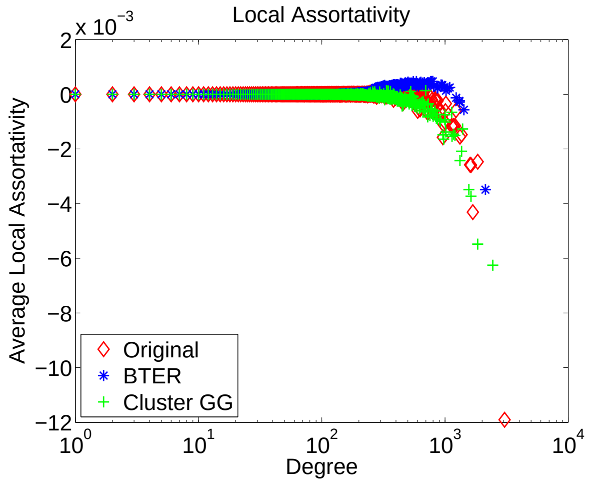
graph preserves important aspects of the community structure of the real network.



(a) Degree Distribution



(b) Average Clustering Coefficient



(c) Local Assortativity

Figure 6.7: Comparison of Properties of Various Models for the soc-Epinions Network.

6.7 Conclusion

In this chapter, we presented an algorithm for generating a network based on clusters generated using a clustering algorithm on the original network. We used numerical experiments to compare our graph generator to the original network and another popular graph generator, BTER, designed to match the clustering coefficient distribution. We looked at

Graph Name	Global Clustering Coefficient			Assortativity Coefficient		
	Original	CGG	Edge-Switching	Original	CGG	Edge-Switching
ca-GrQc	0.629	0.514	0.000	0.63919	0.57355	-0.01619
ca-AstroPh	0.318	0.108	0.000	0.20513	0.12293	-0.01918
cit-HepPh	0.146	0.016	0.000	-0.00629	-0.03341	-0.01417
as-735	0.01	0.016	0.001	-0.18176	-0.16105	-0.18817
soc-Epinions	0.066	0.016	0.001	-0.04065	-0.09630	-0.19121

Table 6.4: Comparing the Edge Switching Algorithm with the Clustering Graph Generator

two different measures of community structure, the clustering coefficient and assortative mixing, both at the global and local level. We found the our graph generator performed well. While the global clustering coefficient was generally lower for our graph generator than in the original graph, our graph generator was much better able to match the assortativity coefficient and unlike BTER was always able to match the sign of the assortativity coefficient. As well, when we compared our graph generator to the edge switching algorithm applied to the entire network we were able to show that by building a synthetic graph based on clusters helps preserve the community structure in the network.

One possible area for future research, is to extend the graph generator to bipartite graphs. In a bipartite graph, nodes can either be classified as belonging to class 1 (black) or class 2 (red) where edges only exist between a node of class 1 and a node of class 2. The clustering graph generator should work they same in the bipartite case where we note that each cluster on the first level can include both black and red nodes. However, we will have to ensure that black-black and red-red connections do not occur. In the first step of the algorithm when we use the matching model to create edges between the clusters this just involves adding a check to make sure that when we choose two nodes they do not belong to the same class. If a trial edge does contain two nodes of the same class then we simply do not accept the trial edge as an actual edge. In the second part of the algorithm where we switch edges within a cluster, again we can check to ensure we are not forming an edge that has nodes from the same class, when we perform a switch. Alternatively, we can list all the edges with nodes of class one (red) first and nodes of class 2 (black) second. Then when we perform any switch we cannot form edges with two nodes of the same class. This would generate bipartite networks with the same community structure as the original network.

Chapter 7

Conclusion

In this thesis we explored two different topics within the field of data science. In the first part, in Chapters 2 and 3, we use a nonlinearly preconditioned nonlinear conjugate gradient (PNCG) algorithm to find the solution to two different problems: finding the rank-R canonical tensor decomposition and finding the solution to a latent factor model. The PNCG algorithm presented in Chapter 2 is very general and allows for any nonlinear preconditioner, however, for both of the problems we considered we used the ALS algorithm as a nonlinear preconditioner. For both problems the PNCG algorithm was successful in increasing the convergence speed relative to ALS (and NCG for the rank-R canonical tensor decomposition). Given the success of the PNCG algorithm it would be interesting to investigate the effectiveness of PNCG for other nonlinear optimization problems noting that although the PNCG algorithm allows for any nonlinear preconditioner, effective preconditioners for more general nonlinear optimization problems will be highly problem-dependent. As a fist step, we could use the simple latent factor model from Chapter 3 to investigate the convergence speed of the PNCG algorithm using the stochastic gradient descent method as a nonlinear preconditioner. As well, we could consider some of the extensions to the basic factor model that are outlined by Koren, Bell and Volinsky [59] and apply the PNCG algorithm to find a solution to these models, both with ALS as a preconditioner and other possible nonlinear preconditioners.

In the second part of this thesis we explored network models and began in Chapter 4 by examining in detail the Chung-Lu model [4, 22, 23]. The Chung-Lu model is a very simple model that is designed to match the expected degree sequence in the model to the input degree sequence. We showed how changes to the model (i.e. excluding self-edges) and violations of the degree constraint ($k_i^2 \leq 2m \; \forall \; i$) imply that this property, $E(\mathbf{D}_i) = k_i \; \forall \; i$, is no longer true. We also examined in-depth an approximate Chung-Lu model, the MCL model, which arose from an algorithm designed to increase the speed of the $\mathcal{O}(n^2)$ algorithm traditionally used to create instances of the Chung-Lu model. For the approximate model, we showed how large the difference between the expected degree and the actual degree can be for some nodes, especially when the degree constraint is violated. In Chapter 5, we then examined some simple ways of modifying the MCL model without self-edges to better match the expected degree sequence to the actual degree sequence. However, we found that in the case where the constraint on the input degree sequence is violated the modifications to the MCL model still leave large differences between the expected degree and the actual degree for some nodes. This suggested that when the degree constraint is violated, the Chung-Lu model is not a good choice for a null model (i.e. we should not compare other graph generators to the Chung-Lu model for networks where the degree constraint is violated). Finally, we introduced an algorithm for generating a network designed to preserve the community structure of the original network being modeled. Our algorithm was based on using clusters generated using a clustering algorithm on the original network. Using numerical experiments we compared our graph generator to the original network and another popular graph generator, BTER, designed to match the clustering coefficient distribution. Using the clustering coefficient and the assortativity coefficient to measure the community structure we found the our graph generator performed well in preserving these properties. While the global clustering coefficient was generally lower for our graph generator than in the original graph, our graph generator was much better able to match the assortativity coefficient and unlike BTER was always able to match the sign of the assortativity coefficient. As well, when we compared our graph generator to the edge switching algorithm applied to the entire network we were able to show that building a synthetic graph based on clusters helps preserve the community structure in the network. As mentioned in Chapter 6 one possible area for future research is to extend our graph generator to bipartite graphs. As well, it would be interesting to examine ways to increase the clustering coefficient within the clusters to better match the global clustering coefficient.

References

[1] Evrim Acar, Daniel M. Dunlavy, and Tamara G. Kolda. A scalable optimization approach for fitting canonical tensor decompositions. *Journal of Chemometrics*, 25:67–86, 2011.

[2] Evrim Acar, Tamara G. Kolda, and Daniel M. Dunlavy. All-at-once optimization for coupled matrix and tensor factorizations. In *MLG'11: Proceedings of Mining and Learning with Graphs*, August 2011.

[3] Evrim Acar and Bulent Yener. Unsupervised multiway data analysis: A literature survey. *IEEE Transactions on Knowledge and Data Engineering*, 21:6–20, 2008.

[4] William Aiello, Fan Chung, and Linyuan Lu. A random graph model for power law graphs. *Experimental Mathematics*, 10(1):53–66, 2001.

[5] Réka Albert and Albert-László Barabási. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[6] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, 2002.

[7] D.G. Anderson. Iterative procedures for nonlinear integral equations. *Journal of the Association for Computing Machinery*, 12:547–560, 1965.

[8] Brett W. Bader and Tamara G. Kolda. MATLAB tensor toolbox version 2.5. Available online, January 2012.

[9] Richard Bartels and James W. Daniel. A conjugate gradient approach to nonlinear elliptic boundary value problems in irregular regions. In G.A. Watson, editor, *Conference on the Numerical Solution of Differential Equations*, volume 363 of *Lecture Notes in Mathematics*, pages 1–11. Springer, Berlin/Heidelberg, 1974.

[10] Robert M. Bell and Yehuda Koren. Lessons from the Netflix prize challenge. *SIGKDD Explorations Newsletter*, 9(2):75–79, 2007.

[11] Robert M. Bell and Yehuda Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, pages 43–52, 2007.

[12] Edward A. Bender and E. Rodney Canfield. The asymptotic number of labeled graphs with given degree sequences. *Journal of Combinatorial Theory, Series A*, 24:296–307, 1978.

[13] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008:P10008, 2008.

[14] J. Bobadilla, F. Ortega, A. Hernando, and A. GutiéRrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 2013.

[15] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Grke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On finding graph clusterings with maximum modularity. In Andreas Brandstdt, Dieter Kratsch, and Haiko Mller, editors, *Graph-Theoretic Concepts in Computer Science*, volume 4769 of *Lecture Notes in Computer Science*, pages 121–132. Springer, Berlin/Heidelberg, 2007.

[16] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking*, pages 309–320, 2000.

[17] Peter Brune, Matthew G. Knepley, Barry F. Smith, and Xuemin Tu. Composing scalable nonlinear algebraic solvers. *SIAM Review*, forthcoming.

[18] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35:283–319, 1970.

[19] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38(1), June 2006.

[20] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Fourth SIAM International Conference on Data Mining*, 2004.

[21] Tianqi Chen, Zhao Zheng, Qiuxia Lu, Xiao Jiang, Yuqiang Chen, Weinan Zhang, Kailong Chen, Yong Yu, Nathan N Liu, Bin Cao, et al. Informative ensemble of multi-resolution dynamic factorization models. In *KDD-Cup'11 Workshop*, 2011.

[22] Fan Chung and Linyuan Lu. The average distance in random graphs with given expected degrees. *Proceedings of National Academy of Science*, 99:15879–15882, 2002.

[23] Fan Chung and Linyuan Lu. Connected components in a random graph with given degree sequences. *Annals of Combinatorics*, 6:125–145, 2002.

[24] P. Concus, G. H. Golub, and D. P. O'Leary. Numerical solution of nonlinear elliptical partial differential equations by a generalized conjugate gradient method. *Computing*, 19:321–339, 1977.

[25] Hans De Sterck. A nonlinear GMRES optimization algorithm for canonical tensor decomposition. *SIAM Journal on Scientific Computing*, 34:A1351–A1379, 2012.

[26] Hans De Sterck. Steepest descent preconditioning for nonlinear GMRES optimization. *Numerical Linear Algebra with Applications*, 20:453–471, 2013.

[27] Hans De Sterck and Manda Winlaw. A nonlinearly preconditioned conjugate gradient algorithm for rank-R canonical tensor approximation. *Numerical Linear Algebra with Applications*, 22:410–432, 2015.

[28] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[29] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. The Yahoo! music dataset and KDD-Cup '11. *JMLR: Workshop and Conference Proceedings*, 18:3–18, 2012.

[30] Daniel M. Dunlavy, Tamara G. Kolda, and Evrim Acar. Poblano v1.0: A MATLAB toolbox for gradient-based optimization. Technical Report SAND2010-1422, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, March 2010.

[31] Nurcan Durak, Tamara G. Kolda, Ali Pinar, and C. Seshadhri. A scalable null model for directed graphs matching all degree distributions: In, out, and reciprocal. In *NSW*

2013: Proceedings of IEEE 2013 2nd International Network Science Workshop, pages 23–30, April 2013.

[32] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1:211–218, 1936.

[33] P. Erdős and A. Rényi. On the evolution of random graphs. In *Publication of The Mathematic Institute of The Hungarian Academy of Sciences*, pages 17–61, 1960.

[34] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 28–36, 2003.

[35] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 251–262, 1999.

[36] Haw-ren Fang and Yousef Saad. Two classes of multisecant methods for nonlinear acceleration. *Numerical Linear Algebra with Application*, 16:197–221, 2009.

[37] Illes J. Farkas, Imre Derenyi, Albert-Laszlo Barabasi, and Tamas Vicsek. Spectra of "real-world" graphs: Beyond the semi-circle law. *Physical Review E*, 64(026704), 2001.

[38] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *Computer Journal*, 7:149–154, 1964.

[39] Simon Funk. Netflix update: Try this at home. http://sifter.org/~simon/journal/20061211.html, 2006.

[40] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[41] Graph500. Graph 500 benchmark. http://www.graph500.org/specifications.html.

[42] Lars Grasedyck, Daniel Kressner, and Christine Tobler. A literature survey of low-rank tensor approximation techniques. http://arxiv.org/abs/1302.7121, 2013.

[43] Alexander Gutfraind, Lauren Ancel Meyers, and Ilya Safro. Multiscale network generation. *CoRR*, abs/1207.4266, 2012.

[44] William W. Hager and Hongchao Zhang. A survey of nonlinear conjugate gradient methods. *Pacific Journal of Optimization*, 2:35–58, 2006.

[45] Richard A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970.

[46] J. Håstad. Tensor rank is NP-complete. *Journal of Algorithms*, 11:644–654, 1990.

[47] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 230–237, 1999.

[48] Magnus R. Hestenes and Eduard Stiefel. Method of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[49] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, pages 263–272, 2008.

[50] Christopher C. Johnson. Logistic matrix factorization for implicit feedback data. In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014.

[51] John M. Roberts Jr. Simple methods for simulating sociomatrices with given marginal totals. *Social Networks*, 22:273–283, 2000.

[52] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[53] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. The web as a graph: measurements, models, and methods. In *Proceedings of the 5th Annual International Conference on Computing and Combinatorics*, COCOON '99, pages 1–17, 1999.

[54] Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. Efficient retrieval of recommendations in a matrix factorization framework. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, pages 535–544, 2012.

[55] Tamara G. Kolda. Orthogonal tensor decompositions. *SIAM Journal on Matrix Analysis and Applications*, 23:243–255, 2001.

[56] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51:455–500, 2009.

[57] Tamara G. Kolda, Ali Pinar, et al. FEASTPACK v1.1. http://www.sandia.gov/~tgkolda/feastpack/, January 2014.

[58] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, September 2014.

[59] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[60] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. Stochastic models for the web graph. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 57–65, Washington, DC, USA, 2000. IEEE Computer Society.

[61] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

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

[63] Jurij Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[64] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[65] David G. Luenberger. *Introduction to Linear and Nonlinear Programming*. Addison Wesley, 2nd edition, 1984.

[66] Stanley Milgram. The small world problem. *Psychology Today*, pages 60–67, 1967.

[67] Joel C. Miller and Aric A. Hagberg. Efficient generation of networks with given expected degrees. In Alan M. Frieze, Paul Horn, and Pawel Pralat, editors, *WAW*, volume 6732 of *Lecture Notes in Computer Science*, pages 115–126. Springer, 2011.

[68] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon. On the uniform generation of random graphs with prescribed degree sequences. *Arxiv preprint cond-mat/0312028*, 2004.

[69] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298:824–827, 2002.

[70] Hans Detlef Mittelmann. On the efficient solution of nonlinear finite element equations I. *Numerische Mathematik*, 35:277–291, 1980.

[71] Michael Molloy and Bruce Reed. A critical point for random graphs with a given degree sequence. *Random Struct. Algorithms*, 6(2-3):161–180, March 1995.

[72] Michael Molloy and Bruce Reed. The size of the giant component of a random graph with a given degree sequence. *Combinatorics, Probababilty and Computing*, 7:295–305, 1998.

[73] Michael Molloy and Bruce Reed. The size of the giant component of a random graph with a given degree sequence. *Combinatorics, Probababilty and Computing*, 7:295–305, 2000.

[74] Jorge J. Moré and David J. Thuente. Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software*, 20(3):286–307, 1994.

[75] MovieLens. http://grouplens.org/datasets/movielens/.

[76] Steven Mussman, John Moore, Joseph J. Pfeiffer III, and Jennifer Neville. Assortativity in chung lu random graph models. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis (SNAKDD 2014)*, 2014.

[77] Larry Nazareth and Jorge Nocedal. Conjugate direction methods with variable storage. *Mathematical Programming*, 23:326–340, 1982.

[78] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64:026118, 2001.

[79] Mark E. Newman. Assortative mixing in networks. *Physical Review Letters*, 89:208701, 2002.

[80] Mark E. J. Newman. *Networks: An Introduction*. Oxford University Press, New York, NY, USA, 2010.

[81] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[82] University of Oregon Route Views Project. Online data and reports. http://www.routeviews.org.

[83] C.W. Oosterlee and T. Washio. Krylov subspace acceleration of nonlinear multigrid with application to recirculating flows. *SIAM Journal on Scientific Computing*, 21:1670–1690, 2000.

[84] Pentti Paatero. The multilinear engine: A table-driven, least squares program for solving multilinear problems, including the n-way parallel factor analysis model. *Journal of Computational and Graphical Statistics*, 8:854–888, 1999.

[85] Joseph J. Pfeiffer III, Timothy La Fond, Sebastian Moreno, and Jennifer Neville. Fast generation of large scale social networks while incorporating transitive closures. In *Fourth ASE/IEEE International Conference on Social Computing (SocialCom)*, 2012.

[86] Ali Pinar, C. Seshadhri, and Tamara G. Kolda. The similarity between stochastic Kronecker and Chung-Lu graph models. In *SDM12: Proceedings of the Twelfth SIAM International Conference on Data Mining*, pages 1071–1082, April 2012.

[87] M. Piraveenan, M. Prokopenko, and A. Y. Zomaya. Local assortativeness in scale-free networks. *Europhysics Letters*, 84:28002, 2008.

[88] E. Polak and G. Ribière. Note sur la convergence de méthodes de directions conjugées. *Revue Françasie d'Informatique et de Recherche Opérationnelle*, 16:35–43, 1969.

[89] Péter Pulay. Convergence acceleration of iterative sequences: The case of SCF iteration. *Chemical Physics Letters*, 73:393–398, 1980.

[90] A. Ramachandra Rao, Rabindranath Jana, and Suraj Bandyopadhyay. A Markov chain Monte Carlo method for generating random (0,1)-matrices with given marginals. *Sankhya: The Indian Journal of Statistics, Series A (1961-2002)*, 58:225–242, 1996.

[91] S. Redner. How Popular is Your Paper? An Empirical Study of the Citation Distribution. *The European Physical Journal B*, 4:131–134, 1998.

[92] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.

[93] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, pages 285–295, 2001.

[94] C. Seshadhri, Tamara G. Kolda, and Ali Pinar. Community structure and scale-free collections of Erdős-Rényi graphs. *Physical Review E*, 85, May 2012.

[95] David S.G. Stirling, editor. *Mathematical Analysis and Proof.* Woodhead Publishing, 2nd edition, 2010.

[96] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Major components of the gravity recommendation system. *SIGKDD Explorations*, 9(2):80–83, 2007.

[97] A. N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-posed problems.* John Wiley, New York, 1977.

[98] Giorgio Tomasi and Rasmus Bro. A comparison of algorithms for fitting the PARAFAC model. *Computational Statistics and Data Analysis*, 50:1700–1734, 2006.

[99] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra.* SIAM, 1997.

[100] André. Uschmajew. Local convergence of the alternating least squares algorithm for canonical tensor approximation. *SIAM Journal on Matrix Analysis and Applications*, 33:639–652, 2012.

[101] Homer F. Walker and Peng Ni. Anderson acceleration for fixed-point iterations. *SIAM Journal on Numerical Analysis*, 49:1715–1735, 2011.

[102] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[103] Manda Winlaw, Michael Hynes, Anthony Caterini, and Hans De Sterck. Algorithmic acceleration of parallel als for collaborative filtering: Speeding up distributed big data recommendation in spark. In *Proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems*, 2015. forthcoming.

[104] Nicholas C. Wormald. The asymptotic connectivity of labelled regular graphs. *Journal of Combinatorial Theory, Series B*, 31(2):156 – 167, 1981.

[105] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*, pages 337–348, 2008.

[106] Michael Zibulevsky and Michael Elad. L1-L2 optimization in signal and image processing. *IEEE Signal Processing Magazine*, 27:76–88, 2010.

APPENDICES

Appendix A

$\mathcal{O}(m)$ Chung-Lu Model Proofs

A.1 Prove $0 < \frac{k_i k_j}{2m^2} < 1 \; \forall \; i, j$ where $k_i, k_j > 0$.

Proof. For any i, j,

$$\begin{aligned}
k_i k_j &< \frac{1}{2}k_i^2 + k_i k_j + \frac{1}{2}k_j^2 \\
&= \frac{1}{2}(k_i + k_j)^2 \\
&\leq \frac{1}{2}\left[(k_i + k_j)^2 + 2(k_i + k_j)(2m - k_i - k_j) + (2m - k_i - k_j)^2\right] \\
&= \frac{1}{2}(k_i + k_j + 2m - k_i - k_j)^2 \\
&= \frac{1}{2}(2m)^2 \\
&\Rightarrow \frac{k_i k_j}{(2m)^2} < \frac{1}{2}
\end{aligned}$$

where $2m - k_i - k_j \geq 0$ since $2m = \sum_n k_i$. This also implies that $\frac{k_i k_j}{2m^2} < 1$.

Since $k_i, k_j > 0$ this implies $\frac{k_i k_j}{2m^2} > 0$. □

Corollary. $0 < \frac{k_i^2}{4m^2} < 1 \; \forall \; i$