Distributed and Large-Scale Optimization

Abdulrahman Kalbat

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

ABSTRACT

Distributed and Large-Scale Optimization

Abdulrahman Kalbat

This dissertation is motivated by the pressing need for solving real-world large-scale optimization problems with the main objective of developing scalable algorithms that are capable of solving such problems efficiently. Large-scale optimization problems naturally appear in complex systems such as power networks and distributed control systems, which are the main systems of interest in this work. This dissertation aims to address four problems with regards to the theory and application of large-scale optimization problems, which are explained below:

Chapter 2: In this chapter, a fast and parallelizable algorithm is developed for an arbitrary decomposable semidefinite program (SDP). Based on the alternating direction method of multipliers, we design a numerical algorithm that has a guaranteed convergence under very mild assumptions. We show that each iteration of this algorithm has a simple closed-form solution, consisting of matrix multiplications and eigenvalue decompositions performed by individual agents as well as information exchanges between neighboring agents. The cheap iterations of the proposed algorithm enable solving a wide spectrum of real-world large-scale conic optimization problems that could be reformulated as SDP.

Chapter 3: Motivated by the application of sparse SDPs to power networks, the objective of this chapter is to design a fast and parallelizable algorithm for solving the SDP relaxation of a large-scale optimal power flow (OPF) problem. OPF is fundamental problem used for the operation and planning of power networks, which is non-convex and NP-hard in the worst case. The proposed algorithm would enable a real-time power network management and improve the system's reliability. In particular, this algorithm helps with the realization of Smart Grid by allowing to make optimal decisions very fast in response to the stochastic nature of renewable energy. The proposed algorithm is evaluated on IEEE benchmark systems.

Chapter 4: The design of an optimal distributed controller using an efficient computational

method is one of the most fundamental problems in the area of control systems, which remains as an open problem due to its NP-hardness in the worst case. In this chapter, we first study the infinite-horizon optimal distributed control (ODC) problem (for deterministic systems) and then generalize the results to a stochastic ODC problem (for stochastic systems). Our approach rests on formulating each of these problems as a rank-constrained optimization from which an SDP relaxation can be derived. We show that both problems admit sparse SDP relaxations with solutions of rank at most 3. Since a rank-1 SDP matrix can be mapped back into a globally-optimal controller, the rank-3 solution may be deployed to retrieve a near-global controller. We also propose computationally cheap SDP relaxation for each problem and then develop effective heuristic methods to recover a near-optimal controller from the low-rank SDP solution. The design of several near-optimal structured controllers with global optimality degrees above 99% will be demonstrated.

Chapter 5: The frequency control problem in power networks aims to control the global frequency of the system within a tight range by adjusting the output of generators in response to the uncertain and stochastic demand. The intermittent nature of distributed power generation in smart grid makes the traditional decentralized frequency controllers less efficient and demands distributed controllers that are able to deal with the uncertainty in the system introduced by non-dispatchable supplies (such as renewable energy), fluctuating loads, and measurement noise. Motivated by this need, we study the frequency control problem using the results developed in Chapter 4. In particular, we formulate the problem and then conduct a case study on the IEEE 39-Bus New England system. The objective is to design a near-global optimal distributed frequency controller for the New England test system by optimally adjusting the mechanical power input to each generator based on the real-time measurement received from neighboring generators through a user-defined communication topology.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my deepest gratitude and appreciation to my PhD advisor, Javad Lavaei, for his continuous support, encouragement and extreme patience throughout my PhD studies. I am honored and privileged to have had the chance to learn from his impressive mathematical talents, his creativity and his generous spirit. I am forever indebted to him for introducing me to a broad range of useful and high-impact research that shaped my academic life and helped me to become a mature and independent researcher. Certainly, this thesis would not have been completed without his generous help and support.

I am also grateful to Xiaodong Wang, Javad Ghaderi, Ali Davoudi and Reza Arghandeh for their time while serving as members on my dissertation committee and for their thoughtful comments and advice that have made my thesis much stronger.

My sincere thanks to my parents for their unconditional love and support. I am grateful to my bothers and sister for their continuous encouragement to pursue my dreams and support my passion for education and science and engineering. I am thankful to all my colleagues and friends at Columbia University who made my academic life exciting.

I would like to thank the United Arab Emirates Embassy and the United Arab Emirates University for their generous financial support during my master and PhD studies in Columbia University.

To my family

Chapter 1

Introduction

This dissertation is motivated by the pressing need for solving real-world large-scale optimization problems with the main objective of developing scalable algorithms that are capable of solving such problems efficiently. Large-scale optimization problems naturally appear in complex systems such as power networks and distributed control systems that are the main systems of interest in this work. This dissertation addresses four problems in Chapters 2-5, which are concerned with the theory and applications of large-scale optimization. In what follows, we will first introduce the problem to be studied in each chapter of this work and then outline the main contributions.

1.0.1 A Fast Distributed Algorithm for Decomposable SDPs

Semidefinite programs (SDP) are attractive due in part to three reasons. First, positive semidefinite constraints appear in many applications [1]. Second, SDPs can be used to study and approximate hard combinatorial optimization problems [2]. Third, this class of convex optimization problems includes linear, quadratic, quadratically-constrained quadratic, and second-order cone programs. It is known that small- to medium-sized SDP problems can be solved efficiently by interior point methods in polynomial time up to any arbitrary precision [3]. However, these methods are less practical for large-scale SDPs due to computation time and memory issues. However, it is possible to somewhat reduce the complexity by exploiting any possible structure in the problem such as sparsity.

Alternating direction method of multipliers (ADMM) is a first-order optimization algorithm proposed in the mid-1970s by [4] and [5]. This method has attracted much attention recently since it can be used for large-scale optimization problems and also be implemented in parallel and distributed computational environments [6; 7]. Compared to second-order methods that are able to achieve a high accuracy via expensive iterations, ADMM relies on low-complex iterations and can achieve a modest accuracy in tens of iterations.

Because of the scalability of ADMM, the main objective of Chapter 2 is to design a distributed ADMM-based parallel algorithm for solving an arbitrary sparse large-scale decomposable SDP with a guaranteed convergence, under very mild assumptions. We consider a canonical form of decomposable SDPs, which is characterized by a graph of agents (nodes) and edges. Each agent needs to find the optimal value of its associated positive semidefintie matrix subject to local equality and inequality constraints as well as overlapping constraints with its neighbors (more precisely, the matrices of two neighboring agents may be subject to consistency constraints). The objective function of the overall SDP is the summation of individual objectives of all agents. At every iteration, each agent performs simple computations (matrix multiplication and eigenvalue decomposition) without having to solve any optimization subproblem, and then communicates some information to its neighbors. By deriving a Lyapunov-type non-increasing function, it is shown that the proposed algorithm converges as long as Slater's conditions hold. Simulations results on large-scale SDP problems with a few million variables are offered to elucidate the efficacy of this work.

1.0.2 A Fast Parallelizable Algorithm for Convex Relaxation of Optimal Power Flow Problem

The optimal power flow (OPF) problem finds an optimal operating point of a power system by minimizing a certain objective function (e.g., transmission loss or generation cost) subject to power flow equations and operational constraints [8], [9]. Motivated by the importance of this fundamental problem for operation and planning as well as the potential monetary savings involved [10], many optimization techniques have been explored for the OPF problem. Due to the non-convexity and NP-hardness of OPF, the existing algorithms are not robust, lack performance guarantees and may not find a global optimum. With the goal of designing a polynomial-time algorithm that finds a global solution for OPF, [11] derives an SDP relaxation for OPF, which results in a globally optimal solution if the duality gap is zero. The proposed relaxation can find near-global solutions with global optimality guarantees of at least 99% for IEEE and Polish systems [12], and is theoretically proven to be exact under various assumptions [13], [14], [15], [16], [17], [18]. However, this relaxation is a high-dimensional SDP problem, which imposes some limitations on its practicality for real-world networks.

Motivated by the application of sparse SDPs to power networks, the objective of Chapter 3 is to design a fast and parallelizable algorithm for solving sparse SDPs that could be utilized to solve large-scale SDP relaxations of the OPF problem. To this end, the underling sparsity structure of a given SDP problem is captured using a tree decomposition technique, leading to a decomposed SDP problem. A highly distributed/parallelizable numerical algorithm is developed for solving the decomposed SDP, based on the ADMM method. Each iteration of the designed algorithm has a closed-form solution, which involves multiplications and eigenvalue decompositions over certain submatrices induced by the tree decomposition of the sparsity graph. The proposed algorithm is applied to the classical optimal power flow problem, and also evaluated on IEEE benchmark systems. This algorithm exhibits an outstanding performance for power systems since real-world networks have low treewidth.

1.0.3 Convex Relaxation for Optimal Distributed Control Problem

Real-world systems mostly consist of many interconnected subsystems, and designing an optimal controller for them pose several challenges to the field of control theory. The area of *distributed control* is created to address the challenges arising in the control of these systems. The objective is to design a constrained controller whose structure is specified by a set of permissible interactions between the local controllers with the aim of reducing the computation or communication complexity of the overall controller. If the local controllers are not allowed to exchange information, the problem is often called *decentralized controller* design. It has been long known that the design of an optimal distributed (decentralized) controller is a daunting task because it amounts to an NP-hard optimization problem in general [19; 20]. There is no surprise that the decentralized control problem is computationally hard to solve. This is a consequence of the fact that several classes of optimization problems, including polynomial optimization and quadratically-constrained quadratic program (QCQP) as a special case, are NP-hard in the worst case. Due to the complexity of such problems, various convex relaxation methods based on linear matrix inequality (LMI), semidefinite programming, and second-order cone programming (SOCP) have gained popularity [21;

22].

In Chapter 4, two problems of infinite-horizon optimal distributed control (ODC) and stochastic ODC are studied. Our approach rests on formulating each of these problems as a rank-constrained optimization problem from which an SDP relaxation can be derived. As the first contribution of this chapter, we show that infinite-horizon ODC and stochastic ODC both admit sparse SDP relaxations with solutions of rank at most 3. Since a rank-1 SDP matrix can be mapped back into a globally-optimal controller, the rank-3 solution may be deployed to retrieve a near-global controller. We also propose two computationally cheap SDP relaxations associated with infinite-horizon ODC and stochastic ODC. Afterwards, we develop effective heuristic methods to recover a near-optimal controller from the low-rank SDP solution. The superiority of the proposed technique is demonstrated on several thousand simulations for mass spring and random systems.

1.0.4 Optimal Distributed Frequency Control in Power Systems

The problem of frequency control in power systems is mainly about controlling the frequency of the grid within a tight range in order to keep a balance between the active powers injected and withdrawn by the generators and customers, respectively. The intermittent nature of distributed power generation in smart grid requires controllers that are able to deal with the uncertainty in the system caused by non-dispatchable supplies (such as renewable energy), fluctuating loads and measurement noise. Motivated by this need, the performance of the computationally-cheap SDP relaxation combined with the indirect recovery method for both Infinite-Horizon and Stochastic ODC developed in Chapter 4 is evaluated in Chapter 5 on the problem of designing an optimal distributed frequency controller for IEEE 39-Bus New England Power System. The main objective of the unknown optimal distributed controller is to optimally adjust the mechanical power input to each generator as well as being structurally constrained by a user-defined communication topology. This pre-determined communication topology specifies which generators exchange their rotor angle and frequency measurements with one another. These controllers are designed for four different communication topologies and are proven to be all stabilizing with high near global optimality degrees (as high as 99 % for some topologies).

It is worth mentioning that the materials presented in this dissertation are published in the following journal and conferences:

- G. Fazelnia, R. Madani, A. Kalbat and J. Lavaei, "Convex Relaxation for Optimal Distributed Control Problem," *accepted in IEEE Transactions on Automatic Control*, 2015.

- A. Kalbat, R. Madani, G. Fazelnia, and J. Lavaei, "Efficient Convex Relaxation for Stochastic Optimal Distributed Control Problem," *in Proc. 52nd Annual Allerton Conference on Communication, Control, and Computing*, Monticello, IL, 2014.

- A. Kalbat and J. Lavaei, "A Fast Distributed Algorithm for Decomposable Semidefinite Programs," *in Proc. 54th IEEE Conference on Decision and Control*, Osaka, Japan, 2015.

- R. Madani, A. Kalbat and J. Lavaei, "ADMM for Sparse Semidefinite Programming with Applications to Optimal Power Flow Problem," *in Proc. 54th IEEE Conference on Decision and Control*, Osaka, Japan, 2015.

Chapter 2

A Fast Distributed Algorithm for Decomposable Semidefinite Programs

In this chapter, a fast and parallelizable algorithm is developed for an arbitrary decomposable semidefinite program (SDP). To formulate a decomposable SDP, we consider a multi-agent canonical form represented by a graph, where each agent (node) is in charge of computing its corresponding positive semidefinite matrix subject to local equality and inequality constraints as well as overlapping (consistency) constraints with regards to the agent's neighbors. Based on the alternating direction method of multipliers, we design a numerical algorithm, which has a guaranteed convergence under very mild assumptions. Each iteration of this algorithm has a simple closed-form solution, consisting of matrix multiplications and eigenvalue decompositions performed by individual agents as well as information exchanges between neighboring agents. The cheap iterations of the proposed algorithm enable solving real-world large-scale conic optimization problems.

2.1 Introduction

Alternating direction method of multipliers (ADMM) is a first-order optimization algorithm proposed in the mid-1970s by [4] and [5]. This method has attracted much attention recently since it can be used for large-scale optimization problems and also be implemented in parallel and distributed computational environments [6; 7]. Compared to second-order methods that are able to achieve a high accuracy via expensive iterations, ADMM relies on low-complex iterations and can achieve a modest accuracy in tens of iterations. Inspired by Nesterov's scheme for accelerating gradient methods [23], great effort has been devoted to accelerating ADMM and attaining a high accuracy in a reasonable number of iterations [24]. Since ADMM's performance is affected by the condition number of the problem's data, diagonal rescaling is proposed in [25] for a class of problems to improve the performance and achieve a linear rate of convergence.

The $\mathcal{O}(\frac{1}{n})$ worst-case convergence rate of ADMM is proven in [26; 27] under the assumptions of closed convex sets and convex functions (not necessarily smooth). In [28], the $\mathcal{O}(\frac{1}{n})$ convergence rate is obtained for an asynchronous ADMM algorithm. The recent paper [29] represents ADMM as a dynamical system and then reduces the problem of proving the linear convergence of ADMM to verifying the stability of a dynamical system [29].

Semidefinite programs (SDP) are attractive due in part to three reasons. First, positive semidefinite constraints appear in many applications [1]. Second, SDPs can be used to study and approximate hard combinatorial optimization problems [2]. Third, this class of convex optimization problems includes linear, quadratic, quadratically-constrained quadratic, and second-order cone programs. It is known that small- to medium-sized SDP problems can be solved efficiently by interior point methods in polynomial time up to any arbitrary precision [3]. However, these methods are less practical for large-scale SDPs due to computation time and memory issues. However, it is possible to somewhat reduce the complexity by exploiting any possible structure in the problem such as sparsity.

The pressing need for solving real-world large-scale optimization problems calls for the development of efficient, scalable, and parallel algorithms. Because of the scalability of ADMM, the main objective of this work is to design a distributed ADMM-based parallel algorithm for solving an arbitrary sparse large-scale SDP with a guaranteed convergence, under very mild assumptions. We consider a canonical form of decomposable SDPs, which is characterized by a graph of agents (nodes) and edges. Each agent needs to find the optimal value of its associated positive semidefintie matrix subject to local equality and inequality constraints as well as overlapping constraints with its neighbors (more precisely, the matrices of two neighboring agents may be subject to consistency constraints). The objective function of the overall SDP is the summation of individual objectives of all agents. From the computation perspective, each agent is treated as a processing unit and each edge of the graph specifies what agents can communicate. We propose a distributed algorithm, whose iterations comprise local matrix multiplications and eigenvalue decompositions performed by individual agents as well as information exchanges between neighboring agents.

This chapter is organized as follows. An overview of ADMM is provided in Section 2.2. The distributed multi-agent SDP problem is formalized in Section 2.3. An ADMM-based parallel algorithm is developed in Section 2.4, by first studying the 2-agent case and then investigating the general multi-agent case. Simulation results on randomly-generated large-scale SDPs with a few million variables are provided in Section 2.5. Finally, a summary is given in Section 2.6.

Notations: $\mathbb{R}^n$ and $\mathbb{S}^n$ denote the sets of $n \times 1$ real vectors and $n \times n$ symmetric matrices, respectively. Lower case letters (e.g., x) represent vectors, and upper case letters (e.g., W) represent matrices. $\mathbf{tr}\{W\}$ denotes the trace of a matrix W and the notation $W \succeq 0$ means that W is symmetric and positive semidefinite. Given a matrix W, its (l,m) entry is denoted as $W(l,m)$. The symbols $(\cdot)^T$, $\|\cdot\|_2$ and $\|\cdot\|_F$ denote the transpose, ℓ_2-norm (for vectors) and Frobenius norm (for matrices) operators, respectively. The ordering operator $(a,b)_{\preceq}$ returns (a,b) if $a < b$ and returns (b,a) if $a > b$. The notation $|\mathcal{X}|$ represents the cardinality (or size) of the set $\mathcal{X}$. The finite sequence of variables $x_1,\ldots,x_n$ is denoted by $\{x_i\}_{i=1}^n$. For an $m \times n$ matrix W, the notation $W(\mathcal{X},\mathcal{Y})$ denotes the submatrix of W whose rows and columns are chosen from $\mathcal{X}$ and $\mathcal{Y}$, respectively, for given index sets $\mathcal{X} \subseteq \{1,\ldots,m\}$ and $\mathcal{Y} \subseteq \{1,\ldots,n\}$.

The notation $\mathcal{G} = (\mathcal{V},\mathcal{E})$ defines a graph $\mathcal{G}$ with the vertex (or node) set $\mathcal{V}$ and the edge set $\mathcal{E}$. The set of neighbors of vertex $i \in \mathcal{V}$ is denoted as $N(i)$. To orient the edges of $\mathcal{G}$, we define a new edge set $\mathcal{E}^+ = \{(i,j) \mid (i,j) \in \mathcal{E} \ \text{ and } \ i < j\}$.

2.2 Alternating Direction Method of Multipliers

Consider the optimization problem

$$\min_{x\in\mathbb{R}^n,\ y\in\mathbb{R}^m} \quad f(x) + g(y) \tag{2.1a}$$

$$\text{subject to} \quad Ax + By = c \tag{2.1b}$$

where $f(x)$ and $g(y)$ are convex functions, A, B are known matrices, and c is a given vector of appropriate dimension. The above optimization problem has a separable objective function and linear constraints. Before proceeding with the chapter, three numerical methods for solving this problem will be reviewed.

The first method is *dual decomposition*, which uses the Lagrangian function

$$\begin{aligned}\mathcal{L}(x,y,\lambda) &= f(x)+g(y)+\lambda^T(Ax+By-c)\\ &= \underbrace{f(x)+\lambda^T Ax}_{h_1(x,\lambda)}+\underbrace{g(y)+\lambda^T By}_{h_2(y,\lambda)}-\lambda^T c\end{aligned} \tag{2.2}$$

where λ is the Lagrange multiplier corresponding to the constraint (2.1b). The above Lagrangian function can be separated into two functions $h_1(x,\lambda)$ and $h_2(y,\lambda)$. Inspired by this separation, the dual decomposition method is based on updating x, y and λ separately. This leads to the iterations

$$x^{t+1} := \operatorname*{argmin}_{x}\ h_1(x,\lambda^t) \tag{2.3a}$$

$$y^{t+1} := \operatorname*{argmin}_{y}\ h_2(y,\lambda^t) \tag{2.3b}$$

$$\lambda^{t+1} := \lambda^t + \alpha^t(Ax^{t+1}+By^{t+1}-c) \tag{2.3c}$$

for $t=0,1,2,...$, with an arbitrary initialization (x^0,y^0,λ^0), where α^t is a step size. Note that "argmin" denotes any minimizer of the corresponding function.

Despite its decomposability, the dual decomposition method has robustness and convergence issues. The *method of multipliers* could be used to remedy these difficulties, which is based on the augmented Lagrangian function

$$\mathcal{L}_\mu(x,y,\lambda) = f(x)+g(y)+\lambda^T(Ax+By-c)+\frac{\mu}{2}\|Ax+By-c\|_2^2 \tag{2.4}$$

where μ is a nonnegative constant. Notice that (2.4) is obtained by augmenting the Lagrangian function in (2.2) with a quadratic term in order to increase the smallest eigenvalue of the Hessian of the Lagrangian with respect to (x,y). However, this augmentation creates a coupling between x and y. The iterations corresponding to the method of multipliers are

$$(x^{t+1},y^{t+1}) := \operatorname*{argmin}_{(x,y)}\ \mathcal{L}_\mu(x,y,\lambda^t) \tag{2.5a}$$

$$\lambda^{t+1} := \lambda^t + \mu(Ax^{t+1}+By^{t+1}-c) \tag{2.5b}$$

where $t=0,1,2,....$

In order to avoid solving a joint optimization with respect to x and y at every iteration, the *alternating direction method of multipliers* (ADMM) can be used. The main idea is to first update x by freezing y at its latest value, and then update y based on the most recent value of x. This leads to the 2-block ADMM problem with the iterations [7]:

$$\text{Block 1:} \quad x^{t+1} := \underset{x}{\operatorname{argmin}} \ \mathcal{L}_\mu(x, y^t, \lambda^t) \tag{2.6a}$$

$$\text{Block 2:} \quad y^{t+1} := \underset{y}{\operatorname{argmin}} \ \mathcal{L}_\mu(x^{t+1}, y, \lambda^t) \tag{2.6b}$$

$$\text{Dual:} \quad \lambda^{t+1} := \lambda^t + \mu(Ax^{t+1} + By^{t+1} - c) \tag{2.6c}$$

ADMM offers a distributed computation property, a high degree of robustness, and a guaranteed convergence under very mild assumptions. In the remainder of this chapter, we will use this first-order method to solve large-scale decomposable SDP problems.

2.3 Problem Formulation

Consider an arbitrary simple, connected, and undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the node set $\mathcal{V} := \{1, \ldots, n\}$ and the edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, as illustrated in Figure 2.1. In a physical context, each node could represent an agent (or a machine or a processor or a thread) and each edge represents a communication link between the agents. In the context of this work, each agent is in charge of computing a positive semidefinite matrix variable W_i, and each edge $(i, j) \in \mathcal{E}$ specifies an overlap between the matrix variables W_i and W_j of agents i and j. More precisely, each edge (i, j) is accompanied by two arbitrary integer-valued index sets I_{ij} and I_{ji} to capture the overlap between W_i and W_j through the equation $W_i(I_{ij}, I_{ij}) = W_j(I_{ji}, I_{ji})$. Figure 2.2 illustrates this specification through an example with three overlapping matrices, where every two neighboring submatrices with an identical color must take the same value at optimality. Another way of thinking about this setting is that Figure 2.1 represents the sparsity graph of an arbitrary sparse large-scale SDP with a single global matrix variable W, which is then reformulated in terms of certain matrices of W, named $W_1, ..., W_n$, using the Chordal extension and matrix completion theorems [30]. The objective of this chapter is to solve the decomposable SDP problem (interchangeably referred to as distributed multi-agent SDP) given below.

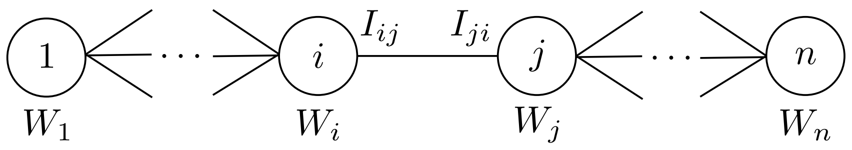


Figure 2.1: A graph representation of the distributed multi-agent SDP.

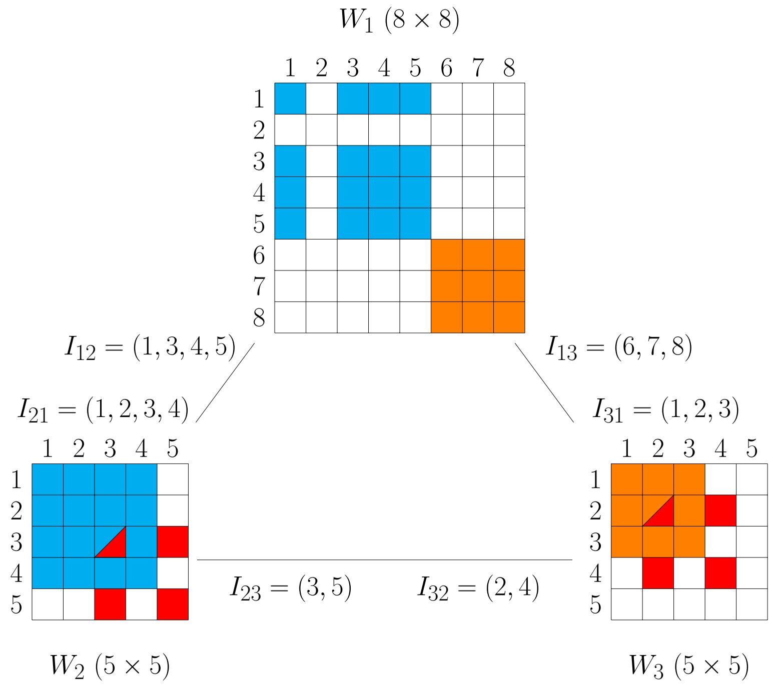


Figure 2.2: An illustration of the definitions of I_{ij} and I_{ji} for three overlapping submatrices W_1, W_2 and W_3

Decomposable SDP:

$$
\begin{aligned}
&\text{minimize} && \sum_{i\in\mathcal{V}} \mathbf{tr}(A_iW_i) && && \text{(2.7a)}\\
&\text{subject to :} && \mathbf{tr}(B_j^{(i)}W_i) = c_j^{(i)} && \forall\, j = 1,\ldots,p_i \quad \text{and} \quad i\in\mathcal{V} && \text{(2.7b)}\\
& && \mathbf{tr}(D_l^{(i)}W_i) \le d_l^{(i)} && \forall\, l = 1,\ldots,q_i \quad \text{and} \quad i\in\mathcal{V} && \text{(2.7c)}\\
& && W_i \succeq 0 && \forall\, i\in\mathcal{V} && \text{(2.7d)}\\
& && W_i(I_{ij},I_{ij}) = W_j(I_{ji},I_{ji}) && \forall\, (i,j)\in\mathcal{E}^+ && \text{(2.7e)}
\end{aligned}
$$

with the variables $W_i \in \mathbb{S}^{n_i}$ for $i = 1,...,n$, where

- the superscript in $(\cdot)^{(i)}$ is not a power but means that the expression corresponds to agent $i \in \mathcal{V}$.
- n_i denotes the size of the submatrix W_i, and p_i and q_i show the numbers of equality and inequality constraints for agent i, respectively.
- $c_j^{(i)}$ and $d_l^{(i)}$ denote the j^{th} and l^{th} elements of the vectors $c_i \in \mathbb{R}^{p_i}$ and $d_i \in \mathbb{R}^{q_i}$ for agent i, as defined below:

$$c_i \triangleq [c_1^{(i)},\ldots,c_{p_i}^{(i)}]^T, \quad d_i \triangleq [d_1^{(i)},\ldots,d_{q_i}^{(i)}]^T$$

- the matrices A_i, $B_j^{(i)}$, and $D_l^{(i)}$ are known and correspond to agent $i \in \mathcal{V}$.

The formulation in (2.7) has three main ingredients:

- **Local objective function:** each agent $i \in \mathcal{V}$ has its own local objective function $\mathbf{tr}(A_iW_i)$ with respect to the local matrix variable W_i. The summation of all local objective functions denotes the global objective function in (2.7a).
- **Local constraints:** each agent $i \in \mathcal{V}$ has local equality and inequality constraints (2.7b) and (2.7c), respectively, as well as a local positive semidefiniteness constraint (2.7d).
- **Overlapping constraints:** constraint (2.7e) states that certain entries of W_i and W_j are identical.

The objective is to design a distributed algorithm for solving (2.7), by allowing each agent $i \in \mathcal{V}$ to collaborate with its neighbors $N(i)$ to find an optimal value for its positive semidefinite submatrix W_i while meeting its own constraints as well as all overlapping constraints. This is accomplished by local computations performed by individual agents and local communication between neighboring agents for information exchange.

There are two scenarios in which (2.7) could be used. In the first scenario, it is assumed that the SDP problem of interest is associated with a multi-agent system and matches the formulation in (2.7) exactly. In the second scenario, we consider an arbitrary sparse SDP problem in the centralized standard form, i.e., an SDP with a single positive semidefinite matrix W, and then convert it into a distributed SDP with multiple but smaller positive semidefinite matrices W_i to match the formulation in (2.7) (note that a dense SDP problem can be put in the form of (2.7) with $n = 1$). The conversion from a standard SDP to a distributed SDP is possible using the idea of chordal decomposition of positive semidefinite cones in [31], which exploits the fact that a matrix W has a positive semidefinite completion if and only if certain submatrices of W, denoted as $W_1, ..., W_n$, are positive semidefinite [32].

In this chapter, we propose an iterative algorithm for solving the decomposable SDP problem (2.7) using the first-order ADMM method. We show that each iteration of this algorithm has a simple closed-form solution, which consists of matrix multiplication and eigenvalue decomposition over matrices of size n_i for agent $i \in \mathcal{V}$.

Our work improves upon some recent papers in this area. [33] is a special case of our work with $n = 1$, which does not offer any parallelizable algorithm for sparse SDPs and may not be applicable to large-scale sparse SDP problems. [31] uses the clique-tree conversion method to decompose sparse SDPs with chordal sparsity pattern into smaller sized SDPs, which can then be solved by interior point methods but this approach is limited by the large number of consistency constraints for the overlapping parts. Recently, [34] solves the decomposed SDP created by [31] using a first-order splitting method, but it requires solving a quadratic program at every iteration, which again imposes some limitations on the scalability of the proposed algorithm. In contrast, the algorithm to be proposed here is parallelizable with low computations at every iteration, without requiring any initial feasible point unlike interior point methods.

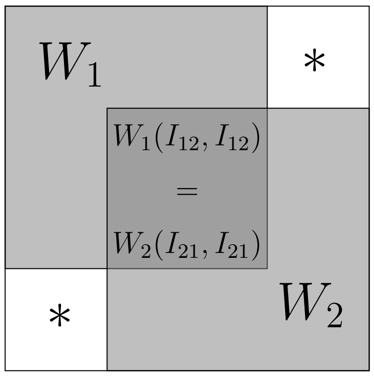


Figure 2.3: Positive semidefinite matrix W (two blocks)

2.4 Distributed Algorithm for Decomposable Semidefinite Programs

In this section, we design an ADMM-based algorithm to solve (2.7). For the convenience of the reader, we first consider the case where there are only two overlapping matrices W_1 and W_2. Later on, we derive the iterations for the general case with an arbitrary graph $\mathcal{G}$.

2.4.1 Two-Agent Case

Assume that there are two overlapping matrices W_1 and W_2 embedded in a global SDP matrix variable W as shown in Figure 2.3, where "*" submatrices of W are redundant (meaning that there is no explicit constraint on the entries of these parts). The SDP problem for this case can be put in the canonical form (2.7), by setting $\mathcal{V} = \{1,2\}$, $\mathcal{E}^+ = \{(1,2)\}$ and $|\mathcal{V}| = 2$:

$$\begin{aligned}
\min_{\substack{W_1 \in \mathbb{S}^{n_1}\\ W_2 \in \mathbb{S}^{n_2}}} \quad & \mathbf{tr}(A_1 W_1) + \mathbf{tr}(A_2 W_2) && && \text{(2.8a)}\\
\text{subject to} \quad & \mathbf{tr}(B_j^{(1)} W_1) = c_j^{(1)} && \forall\, j = 1,\dots,p_1 && \text{(2.8b)}\\
& \mathbf{tr}(B_j^{(2)} W_2) = c_j^{(2)} && \forall\, j = 1,\dots,p_2 && \text{(2.8c)}\\
& \mathbf{tr}(D_l^{(1)} W_1) \le d_l^{(1)} && \forall\, l = 1,\dots,q_1 && \text{(2.8d)}\\
& \mathbf{tr}(D_l^{(2)} W_2) \le d_l^{(2)} && \forall\, l = 1,\dots,q_2 && \text{(2.8e)}\\
& W_1, W_2 \succeq 0 && && \text{(2.8f)}\\
& W_1(I_{12}, I_{12}) = W_2(I_{21}, I_{21}) && && \text{(2.8g)}
\end{aligned}$$

where the data matrices A_1, $B_j^{(1)}$,$D_l^{(1)} \in \mathbb{S}^{n_1}$, the matrix variable $W_1 \in \mathbb{S}^{n_1}$ and the vectors $c_1 \in \mathbb{R}^{p_1}$ and $d_1 \in \mathbb{R}^{q_1}$ correspond to agent 1, whereas the data matrices A_2, $B_j^{(2)}$,$D_l^{(2)} \in \mathbb{S}^{n_2}$, the matrix variable $W_2 \in \mathbb{S}^{n_2}$ and the vectors $c_2 \in \mathbb{R}^{p_2}$ and $d_2 \in \mathbb{R}^{q_2}$ correspond to agent 2. Constraint (2.8g) states that the (I_{12}, I_{12}) submatrix of W_1 overlaps with the (I_{21}, I_{21}) submatrix of W_2. With no loss of generality, assume that the overlapping part occurs at the lower right corner of W_1 and the upper left corner of W_2, as illustrated in Figure 2.3. The dual of the 2-agent SDP problem in (2.8) can be expressed as

$$\begin{aligned}
\text{minimize} \quad & \left(c_1^T z_1 + d_1^T v_1\right) + \left(c_2^T z_2 + d_2^T v_2\right) && \text{(2.9a)}\\
\text{subject to :} \quad & -\sum_{j=1}^{p_1} z_j^{(1)} B_j^{(1)} - \sum_{l=1}^{q_1} v_l^{(1)} D_l^{(1)} + R_1 - \begin{bmatrix} 0 & 0 \\ 0 & H_{1,2} \end{bmatrix} = A_1 && \text{(2.9b)}\\
& -\sum_{j=1}^{p_2} z_j^{(2)} B_j^{(2)} - \sum_{l=1}^{q_2} v_l^{(2)} D_l^{(2)} + R_2 + \begin{bmatrix} H_{2,1} & 0 \\ 0 & 0 \end{bmatrix} = A_2 && \text{(2.9c)}\\
& H_{1,2} = H_{2,1} && \text{(2.9d)}\\
& v_1, v_2 \ge 0 && \text{(2.9e)}\\
& R_1, R_2 \succeq 0 && \text{(2.9f)}
\end{aligned}$$

with the variables $z_1, z_2, v_1, v_2, R_1, R_2, H_{1,2}, H_{2,1}$, where $z_1 \in \mathbb{R}^{p_1}$, $z_2 \in \mathbb{R}^{p_2}$, $v_1 \in \mathbb{R}^{q_1}$ and $v_2 \in \mathbb{R}^{q_2}$ are the Lagrange multipliers corresponding to the equality and inequality constraints in (2.8b)-(2.8e), respectively, and the dual matrix variables $R_1 \in \mathbb{S}^{n_1}$ and $R_2 \in \mathbb{S}^{n_2}$ are the Lagrange multipliers corresponding to the constraint (2.8f). The dual matrix variable $H_{1,2}$ is the Lagrange multiplier corresponding to the submatrix $W_1(I_{12}, I_{12})$ of W_1, whereas $H_{2,1}$ is the Lagrange multiplier corresponding to the submatrix $W_2(I_{21}, I_{21})$ of W_2. Since the overlapping entries between W_1 and W_2 are equal, as reflected in constraint (2.8g), the corresponding Lagrange multipliers should be equal as well, leading to constraint (2.9d).

If we apply ADMM to (2.9), it becomes impossible to split the variables into two blocks of variables associated with agents 1 and 2. The reason is that the augmented Lagrangian function of (2.9) creates a coupling between $H_{1,2}$ and $H_{2,1}$, which then requires updating $H_{1,2}$ and $H_{2,1}$ jointly. This issue can be resolved by introducing a new auxiliary variable $H^{(1,2)}$ in order to decompose the constraint $H_{1,2} = H_{2,1}$ into two constraints $H_{1,2} = H^{(1,2)}$ and $H_{2,1} = H^{(1,2)}$. Similarly, to make the update of v_1 and v_2 easier, we do not impose positivity constraints directly on v_1 and v_2 as in (2.9e). Instead, we impose the positivity on two new vectors $u_1, u_2 \geq 0$ and then add the additional constraints $v_1 = u_1$ and $v_2 = u_2$. By applying the previous modifications, (2.9) could be rewritten in the decomposable form

$$\text{minimize} \quad \sum_{i=1}^{2} \left(c_i^T z_i + d_i^T v_i + I_+(R_i) + I_+(u_i) \right) \tag{2.10a}$$

$$\text{subject to :} \quad -\sum_{j=1}^{p_1} z_j^{(1)} B_j^{(1)} - \sum_{l=1}^{q_1} v_l^{(1)} D_l^{(1)} + R_1 - \begin{bmatrix} 0 & 0 \\ 0 & H_{1,2} \end{bmatrix} = A_1 \tag{2.10b}$$

$$-\sum_{j=1}^{p_2} z_j^{(2)} B_j^{(2)} - \sum_{l=1}^{q_2} v_l^{(2)} D_l^{(2)} + R_2 + \begin{bmatrix} H_{2,1} & 0 \\ 0 & 0 \end{bmatrix} = A_2 \tag{2.10c}$$

$$H_{1,2} = H^{(1,2)} \tag{2.10d}$$

$$H_{2,1} = H^{(1,2)} \tag{2.10e}$$

$$v_1 = u_1 \tag{2.10f}$$

$$v_2 = u_2 \tag{2.10g}$$

with the variables $z_1, z_2, v_1, u_1, v_2, u_2, R_1, R_2, H_{1,2}, H_{2,1}, H^{(1,2)}$, where $I_+(R_i)$ is equal to 0 if $R_i \succeq 0$ and is $+\infty$ otherwise, and $I_+(u_i)$ is equal to 0 if $u_i \geq 0$ and is $+\infty$ otherwise.

To streamline the presentation, define

$$B_i^{\text{sum}} = \sum_{j=1}^{p_i} z_j^{(i)} B_j^{(i)}, \quad D_i^{\text{sum}} = \sum_{l=1}^{q_i} v_l^{(i)} D_l^{(i)}, \quad i = 1, 2 \tag{2.11}$$

and

$$H^{\text{full}}_{1,2} = \begin{bmatrix} 0 & 0 \\ 0 & H_{1,2} \end{bmatrix}, \quad H^{\text{full}}_{2,1} = \begin{bmatrix} -H_{2,1} & 0 \\ 0 & 0 \end{bmatrix} \tag{2.12}$$

Note that B_i^{sum}, D_i^{sum}, $H^{\text{full}}_{1,2}$ and $H^{\text{full}}_{2,1}$ are functions of the variables z_i, v_i, $H_{1,2}$ and $H_{2,1}$, respectively, but the arguments are dropped for notational simplicity. The augmented Lagrangian function for (2.10) can be obtained as

$$\begin{aligned} \mathcal{L}_\mu(\mathcal{F}, \mathcal{M}) = & \sum_{i=1}^{2} \left(c_i^T z_i + d_i^T v_i + I_+(R_i) + I_+(u_i) \right) \\ & + \frac{\mu}{2} \left\| -B_1^{\text{sum}} - D_1^{\text{sum}} + R_1 - H^{\text{full}}_{1,2} - A_1 + \frac{G_1}{\mu} \right\|_F^2 \\ & + \frac{\mu}{2} \left\| -B_2^{\text{sum}} - D_2^{\text{sum}} + R_2 - H^{\text{full}}_{2,1} - A_2 + \frac{G_2}{\mu} \right\|_F^2 \\ & + \frac{\mu}{2} \left\| H_{1,2} - H^{(1,2)} + \frac{G_{1,2}}{\mu} \right\|_F^2 + \frac{\mu}{2} \left\| H_{2,1} - H^{(1,2)} + \frac{G_{2,1}}{\mu} \right\|_F^2 \\ & + \frac{\mu}{2} \left\| v_1 - u_1 + \frac{\lambda_1}{\mu} \right\|_2^2 + \frac{\mu}{2} \left\| v_2 - u_2 + \frac{\lambda_2}{\mu} \right\|_2^2 \end{aligned} \tag{2.13}$$

where $\mathcal{F} = \left(z_1, z_2, v_1, v_2, u_1, u_2, R_1, R_2, H_{1,2}, H_{2,1}, H^{(1,2)}\right)$ is the set of optimization variables and $\mathcal{M} = (G_1, G_2, G_{1,2}, G_{2,1}, \lambda_1, \lambda_2)$ is the set of Lagrange multipliers whose elements correspond to constraints (2.10b) - (2.10g), respectively. Note that the augmented Lagrangian in (2.13) is obtained using the identity

$$\mathbf{tr}\left[X^T(A-B)\right] + \frac{\mu}{2}\|A-B\|_F^2 = \frac{\mu}{2}\left\| A - B + \frac{X}{\mu} \right\|_F^2 + \text{constant} \tag{2.14}$$

In order to proceed, we need to split the set of optimization variables $\mathcal{F}$ into two blocks of variables. To this end, define $\mathcal{X} = \left\{u_1, u_2, R_1, R_2, H^{(1,2)}\right\}$ and $\mathcal{Y} = \{z_1, z_2, v_1, v_2, H_{1,2}, H_{2,1}\}$. Using the method delineated in Section 2.2, the two-block ADMM iterations can be obtained as

$$\text{(Block 1)}\ \ \mathcal{X}^{t+1} = \underset{\mathcal{X}}{\operatorname{argmin}}\ \mathcal{L}_\mu\left(\mathcal{X}, \mathcal{Y}^t, \mathcal{M}^t\right) \tag{2.15a}$$

$$\text{(Block 2)}\ \ \mathcal{Y}^{t+1} = \underset{\mathcal{Y}}{\operatorname{argmin}}\ \mathcal{L}_\mu\left(\mathcal{X}^{t+1}, \mathcal{Y}, \mathcal{M}^t\right) \tag{2.15b}$$

$$G_1^{t+1} = G_1^t + \mu\left(-\overset{t+1}{B_1^{\text{sum}}} - \overset{t+1}{D_1^{\text{sum}}} + R_1^{t+1} - \overset{t+1}{H_{1,2}^{\text{full}}} - A_1\right) \tag{2.15c}$$

$$G_2^{t+1} = G_2^t + \mu\left(-\overset{t+1}{B_2^{\text{sum}}} - \overset{t+1}{D_2^{\text{sum}}} + R_2^{t+1} - \overset{t+1}{H_{2,1}^{\text{full}}} - A_2\right) \tag{2.15d}$$

$$G_{1,2}^{t+1} = G_{1,2}^t + \mu\left(H_{1,2}^{t+1} - \overset{t+1}{H^{(1,2)}}\right) \tag{2.15e}$$

$$G_{2,1}^{t+1} = G_{2,1}^t + \mu\left(H_{2,1}^{t+1} - \overset{t+1}{H^{(1,2)}}\right) \tag{2.15f}$$

$$\lambda_1^{t+1} = \lambda_1^t + \mu\left(v_1^{t+1} - u_1^{t+1}\right) \tag{2.15g}$$

$$\lambda_2^{t+1} = \lambda_2^t + \mu\left(v_2^{t+1} - u_2^{t+1}\right) \tag{2.15h}$$

for $t = 0, 1, 2, ...$

The above updates are derived based on the fact that ADMM aims to find a saddle point of the augmented Lagrangian function by alternatively performing one pass of Gauss Seidel over $\mathcal{X}$ and $\mathcal{Y}$ and then updating the Lagrange multipliers $\mathcal{M}$ through Gradient ascent.

It is straightforward to show that the optimization over $\mathcal{X}$ in Block 1 is fully decomposable and amounts to 5 separate optimization subproblems with respect to the individual variables $u_1, u_2, R_1, R_2, H^{(1,2)}$. In addition, the optimization over $\mathcal{Y}$ in Block 2 is equivalent to 2 separate optimization subproblems with the variables $(z_1, v_1, H_{1,2})$ and $(z_2, v_2, H_{2,1})$, respectively. Interestingly, all these subproblems have closed-form solutions. The corresponding iterations that need to be taken by agents 1 and 2 are provided in (2.16) and (2.17) (given in the next two pages). Note that these agents need to perform local computation in every iteration according to (2.16) and (2.17) and then exchange the updated values of the pairs $(H_{1,2}, G_{1,2})$ and $(H_{2,1}, G_{2,1})$ with one another.

To elaborate on (2.16) and (2.17), the positive semidefinite matrices R_1 and R_2 are updated through the operator $(\cdot)_+$, where X_+ is defined as the projection of an arbitrary symmetric matrix X onto the set of positive semidefinite matrices by replacing its negative eigenvalues with 0 in

Iterations for Agent 1

$$R_1^{t+1} = \left(\overset{t}{B_1^{\text{sum}}} + \overset{t}{D_1^{\text{sum}}} + \overset{t}{H_{1,2}^{\text{full}}} + A_1 - \frac{G_1^t}{\mu} \right)_+ \tag{2.16a}$$

$$u_1^{t+1} = \left(v_1^t + \frac{\lambda_1^t}{\mu} \right)_+ \tag{2.16b}$$

$$\overset{t+1}{H^{(1,2)}} = \frac{1}{2} \left(H_{1,2}^t + H_{2,1}^t + \frac{G_{1,2}^t}{\mu} + \frac{G_{2,1}^t}{\mu} \right) \tag{2.16c}$$

$$(z_1, v_1, H_{1,2})^{t+1} = \text{Lin}\left(u_1^{t+1}, R_1^{t+1}, \overset{t+1}{H^{(1,2)}}, G_1^t, G_{1,2}^t, \lambda_1^t \right) \tag{2.16d}$$

$$G_1^{t+1} = G_1^t + \mu \left(-\overset{t+1}{B_1^{\text{sum}}} - \overset{t+1}{D_1^{\text{sum}}} + R_1^{t+1} - \overset{t+1}{H_{1,2}^{\text{full}}} - A_1 \right) \tag{2.16e}$$

$$G_{1,2}^{t+1} = G_{1,2}^t + \mu \left(H_{1,2}^{t+1} - \overset{t+1}{H^{(1,2)}} \right) \tag{2.16f}$$

$$\lambda_1^{t+1} = \lambda_1^t + \mu \left(v_1^{t+1} - u_1^{t+1} \right) \tag{2.16g}$$

the eigenvalue decomposition[33]. The positive vectors u_1 and u_2 are also updated through the operator $(x)_+$, which replaces any negative entry in an arbitrary vector x with 0 while keeping the nonnegative entries. Using the first-order optimality condition $\nabla_{H^{(1,2)}} \mathcal{L}_\mu(\cdot) = 0$, one could easily find the closed-form solution for $H^{(1,2)}$ as shown in (2.16c) and (2.17c). By combining the conditions $\nabla_{z_1} \mathcal{L}_\mu(\cdot) = 0$, $\nabla_{v_1} \mathcal{L}_\mu(\cdot) = 0$ and $\nabla_{H_{1,2}} \mathcal{L}_\mu(\cdot) = 0$, the updates of $(z_1, v_1, H_{1,2})$ and $(z_2, v_2, H_{2,1})$ reduce to a (not necessarily unique) linear mapping, denoted as Lin$(\cdot)$ in (2.16d) and (2.17d) (due to non-uniqueness, we may have multiple solutions, and any of them can be used in the updates). The Lagrange multipliers in $\mathcal{M}$ are updated through Gradient ascent, as specified in (2.16e)-(2.16g) for agent 1 and in (2.17e)-(2.17g) for agent 2.

2.4.2 Multi-Agent Case

In this part, we will study the general distributed multi-agent SDP (2.7). The dual of this problem, after considering all modifications used to convert (2.9) to (2.10), can be expressed in the

Iterations for Agent 2

$$R_2^{t+1} = \left(\overset{t}{B_2^{\text{sum}}} + \overset{t}{D_2^{\text{sum}}} + \overset{t}{H_{2,1}^{\text{full}}} + A_2 - \frac{G_2^t}{\mu}\right)_+ \tag{2.17a}$$

$$u_2^{t+1} = \left(v_2^t + \frac{\lambda_2^t}{\mu}\right)_+ \tag{2.17b}$$

$$\overset{t+1}{H^{(1,2)}} = \frac{1}{2}\left(H_{1,2}^t + H_{2,1}^t + \frac{G_{1,2}^t}{\mu} + \frac{G_{2,1}^t}{\mu}\right) \tag{2.17c}$$

$$(z_2, v_2, H_{2,1})^{t+1} = \operatorname{Lin}\left(u_2^{t+1}, R_2^{t+1}, \overset{t+1}{H^{(1,2)}}, G_2^t, G_{2,1}^t, \lambda_2^t\right) \tag{2.17d}$$

$$G_2^{t+1} = G_2^t + \mu\left(-\overset{t+1}{B_2^{\text{sum}}} - \overset{t+1}{D_2^{\text{sum}}} + R_2^{t+1} - \overset{t+1}{H_{2,1}^{\text{full}}} - A_2\right) \tag{2.17e}$$

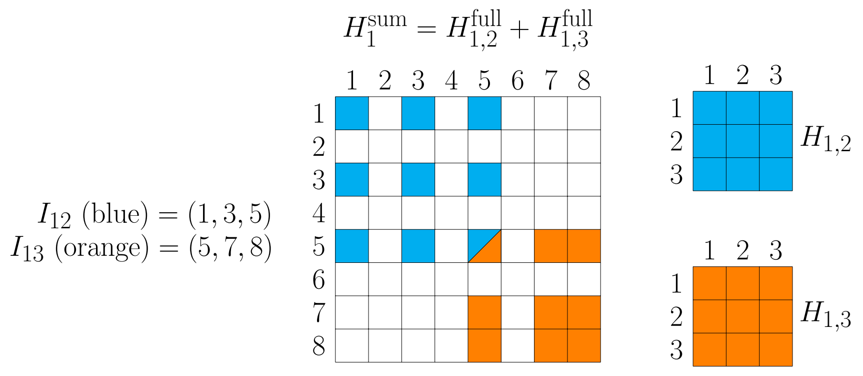
$$G_{2,1}^{t+1} = G_{2,1}^t + \mu\left(H_{2,1}^{t+1} - \overset{t+1}{H^{(1,2)}}\right) \tag{2.17f}$$

$$\lambda_2^{t+1} = \lambda_2^t + \mu\left(v_2^{t+1} - u_2^{t+1}\right) \tag{2.17g}$$

decomposable form

$$\begin{aligned}
\text{minimize} \quad & \sum_{i\in\mathcal{V}} \left(c_i^T z_i + d_i^T v_i + I_+(R_i) + I_+(u_i)\right) && && (2.18\text{a})\\
\text{subject to:} \quad & -B_i^{\text{sum}} - D_i^{\text{sum}} + R_i - \sum_{k\in N(i)} H_{i,k}^{\text{full}} = A_i && \forall\ i \in \mathcal{V} && (2.18\text{b})\\
& H_{i,j} = H^{(i,j)} && \forall\ (i,j)\in\mathcal{E}^+ && (2.18\text{c})\\
& H_{j,i} = H^{(i,j)} && \forall\ (i,j)\in\mathcal{E}^+ && (2.18\text{d})\\
& v_i = u_i && \forall\ i \in \mathcal{V} && (2.18\text{e})
\end{aligned}$$

with the variables $z_i, v_i, u_i, R_i, H_{i,j}, H_{j,i}, H^{(i,j)}$ for every $i \in \mathcal{V}$ and $(i,j) \in \mathcal{E}^+$, where $B_i^{\text{sum}} = \sum_{j=1}^{p_i} z_j^{(i)} B_j^{(i)}$, $D_i^{\text{sum}} = \sum_{l=1}^{q_i} v_l^{(i)} D_l^{(i)}$ and $H_i^{\text{sum}} = \sum_{k\in N(i)} H_{i,k}^{\text{full}}$. Note that $z_i \in \mathbb{R}^{p_i}$ and $v_i \in \mathbb{R}^{q_i}$ are the Lagrange multipliers corresponding to the equality and inequality constraints in (2.7b) and (2.7c), respectively, and that $R_i \in \mathbb{S}^{n_i}$ is the Lagrange multiplier corresponding to the constraint (2.7d). Each element $h_{i,k}^{\text{full}}(a,b)$ of $H_{i,k}^{\text{full}}$ is either zero or equal to the Lagrange multiplier corresponding to an overlapping element $W_i(a,b)$ between W_i and W_k. For a better understanding of the



$H_1^{\text{sum}} = H_{1,2}^{\text{full}} + H_{1,3}^{\text{full}}$
I_{12} (blue) $= (1,3,5)$
I_{13} (orange) $= (5,7,8)$
1 2 3 4 5 6 7 8
$H_{1,2}$
$H_{1,3}$

Figure 2.4: An illustration of the difference between $H_{i,j}^{\text{full}}$, $H_{i,j}$ and H_i^{sum}. Agent 1 is overlapping with agents 2 and agent 3 at the entries specified by I_{12} and I_{13}. The white squares in the left matrix $H_{1,2}^{\text{full}} + H_{1,3}^{\text{full}}$ represent those entries with value 0, and the color squares carry Lagrange multipliers.

difference between $H_{i,j}^{\text{full}}$, $H_{i,j}$ and H_i^{sum}, an example is given in Figure 2.4 for the case where agent 1 is overlapping with agents 2 and 3. The ADMM iterations for the general case can be derived similarly to the 2-agent case, which yields the local computation (2.20) for each agent $i \in \mathcal{V}$.

Consider the parameters defined in (2.21) for every $i \in \mathcal{V}$, $(i,j) \in \mathcal{E}^+$, and time $t \in \{1, 2, 3, ...\}$. Define V^t as

$$\begin{aligned} V^t = & \sum_{i \in \mathcal{V}} \Big(\left(\Delta_{p1}^t\right)_i + \left(\Delta_{p4}^t\right)_i + \left(\Delta_{d1}^t\right)_i + \left(\Delta_{d2}^t\right)_i \Big) \\ & + \sum_{i,j \in \mathcal{E}^+} \Big(\left(\Delta_{p2}^t\right)_{i,j} + \left(\Delta_{p3}^t\right)_{i,j} + \left(\Delta_{d3}^t\right)_{i,j} \Big) \end{aligned} \tag{2.19}$$

Note that $(\Delta_{p1}, \Delta_{p2}, \Delta_{p3}, \Delta_{p4})$, $(\Delta_{d1}, \Delta_{d2}, \Delta_{d3})$, and V are the primal residues, dual residues and aggregate residue for the decomposed problem (2.18). It should be noticed that the dual residues are only considered for the variables in the block $\mathcal{X} = \{u_i, R_i, H^{(i,j)}\}$. Since $H^{(i,j)}$ appears twice in (2.18), the norm in the residue Δ_{d3} is multiplied by 2. The main result of this chapter will be stated below.

Theorem 1. *Assume that Slater's conditions hold for the decomposable SDP problem (2.7). Consider the iterative algorithm given in* (2.20)*. The following statements hold:*

- *The aggregate residue V^t attenuates to 0 in a non-increasing way as t goes to $+\infty$.*

Iterations for Agent $i \in \mathcal{V}$

$$R_i^{t+1} = \left(\overset{t}{B}{}_i^{\mathrm{sum}} + \overset{t}{D}{}_i^{\mathrm{sum}} + \overset{t}{H}{}_i^{\mathrm{sum}} + A_i - \frac{G_i^t}{\mu}\right)_+ \tag{2.20a}$$

$$u_i^{t+1} = \left(v_i^t + \frac{\lambda_i^t}{\mu}\right)_+ \tag{2.20b}$$

$$\overset{t+1}{H}{}^{(i,k)\preceq} = \frac{1}{2}\left(H_{i,k}^t + H_{k,i}^t + \frac{G_{i,k}^t}{\mu} + \frac{G_{k,i}^t}{\mu}\right) \qquad \forall k \in N(i) \tag{2.20c}$$

$$\left(z_i^{t+1}, v_i^{t+1}, \left\{H_{i,k}^{t+1}\right\}_{k\in N(i)}\right) = \mathrm{Lin}\left(u_i^{t+1}, R_i^{t+1}, \left\{\overset{t+1}{H}{}^{(i,k)\preceq}\right\}_{k\in N(i)}, G_i^t, \left\{G_{i,k}^t\right\}_{k\in N(i)}, \lambda_i^t\right) \tag{2.20d}$$

$$G_i^{t+1} = G_i^t + \mu\left(-\overset{t+1}{B}{}_i^{\mathrm{sum}} - \overset{t+1}{D}{}_i^{\mathrm{sum}} + R_i^{t+1} - \overset{t+1}{H}{}_i^{\mathrm{sum}} - A_i\right) \tag{2.20e}$$

$$G_{i,k}^{t+1} = G_{i,k}^t + \mu\left(H_{i,k}^{t+1} - \overset{t+1}{H}{}^{(i,k)\preceq}\right) \qquad \forall k \in N(i) \tag{2.20f}$$

$$\lambda_i^{t+1} = \lambda_i^t + \mu\left(v_i^{t+1} - u_i^{t+1}\right) \tag{2.20g}$$

$$\left(\Delta_{p1}^t\right)_i = \left\|\overset{t}{B}{}_i^{\mathrm{sum}} + \overset{t}{D}{}_i^{\mathrm{sum}} + \overset{t}{H}{}_i^{\mathrm{sum}} + A_i - R_i^t\right\|_F^2 \tag{2.21a}$$

$$\left(\Delta_{p2}^t\right)_{i,j} = \left\|H_{i,j}^t - \overset{t}{H}{}^{(i,j)}\right\|_F^2 \tag{2.21b}$$

$$\left(\Delta_{p3}^t\right)_{i,j} = \left\|H_{j,i}^t - \overset{t}{H}{}^{(i,j)}\right\|_F^2 \tag{2.21c}$$

$$\left(\Delta_{p4}^t\right)_i = \left\|v_i^t - u_i^t\right\|_2^2 \tag{2.21d}$$

$$\left(\Delta_{d1}^t\right)_i = \left\|R_i^t - R_i^{t-1}\right\|_F^2 \tag{2.21e}$$

$$\left(\Delta_{d2}^t\right)_i = \left\|u_i^t - u_i^{t-1}\right\|_2^2 \tag{2.21f}$$

$$\left(\Delta_{d3}^t\right)_{i,j} = 2\left\|\overset{t}{H}{}^{(i,j)} - \overset{t-1}{H}{}^{(i,j)}\right\|_F^2 \tag{2.21g}$$

- *For every $i \in \mathcal{V}$, the limit of $(G_1^t, G_2^t, ..., G_n^t)$ at $t = +\infty$ is an optimal solution for $(W_1, W_2, ..., W_n)$.*

Proof. After realizing that (2.20) is obtained from a two-block ADMM procedure, the theorem follows from [35] that studies the convergence of a standard ADMM problem. The details are omitted for brevity. □

Since the proposed algorithm is iterative with an asymptotic convergence, we need a finite-time stopping rule. Based on [36], we terminate the algorithm as soon as $\max\{\mathrm{P}_1, \mathrm{P}_2, \mathrm{D}_1, \mathrm{D}_2, \mathrm{D}_3, \mathrm{D}_4, \mathrm{Gap}\}$ becomes smaller than a pre-specified tolerance, where

$$(\mathrm{P}_1)_i = \frac{\left\|\overline{B}_i^T \overline{W}_i - c_i\right\|_2 + \left\|\max\left(\overline{D}_i^T \overline{W}_i - d_i, \mathbf{0}\right)\right\|_2}{1 + \|c_i\|_2} \tag{2.22a}$$

$$(\mathrm{P}_2)_{i,j} = \frac{\|W_i(I_{ij}, I_{ij}) - W_j(I_{ji}, I_{ji})\|_F}{1 + \|W_i(I_{ij}, I_{ij})\|_F + \|W_j(I_{ji}, I_{ji})\|_F} \tag{2.22b}$$

$$(\mathrm{D}_1)_i = \frac{\|-B_i^{\mathrm{sum}} - D_i^{\mathrm{sum}} + R_i - H_i^{\mathrm{sum}} - A_i\|_F}{1 + \|A_i\|_1} \tag{2.22c}$$

$$(\mathrm{D}_2)_{i,j} = \frac{\left\|H_{i,j} - H^{(i,j)}\right\|_F}{1 + \|H_{i,j}\|_F + \left\|H^{(i,j)}\right\|_F} \tag{2.22d}$$

$$(\mathrm{D}_3)_{i,j} = \frac{\left\|H_{j,i} - H^{(i,j)}\right\|_F}{1 + \|H_{j,i}\|_F + \left\|H^{(i,j)}\right\|_F} \tag{2.22e}$$

$$(\mathrm{D}_4)_i = \frac{\|v_i - u_i\|_2}{1 + \|v_i\|_2 + \|u_i\|_2} \tag{2.22f}$$

$$\mathrm{Gap} = \frac{\left|\sum_{i\in\mathcal{V}} \left(c_i^T z_i + d_i^T v_i - \mathbf{tr}\left(A_i W_i\right)\right)\right|}{1 + \left|\sum_{i\in\mathcal{V}} \left(c_i^T z_i + d_i^T v_i\right)\right| + \left|\sum_{i\in\mathcal{V}} \mathbf{tr}\left(A_i W_i\right)\right|} \tag{2.22g}$$

for every $i \in \mathcal{V}$ and $(i, j) \in \mathcal{E}^+$, where

- the letters P and D refer to the primal and dual infeasibilities, respectively.
- $\overline{W}_i$ is the vectorized version of W_i obtained by stacking the columns of W_i one under another to create a column vector.
- $\overline{B}_i$ and $\overline{D}_i$ are matrices whose columns are the vectorized versions of $B_j^{(i)}$ and $D_l^{(i)}$ for $j = 1, \ldots, p_i$ and $l = 1, \ldots, q_i$, respectively.

The stopping criteria in (2.22) are based on the primal and dual infeasibilities as well as the duality gap.

2.5 Simulations Results

The objective of this section is to elucidate the results developed earlier on randomly generated large-scale structured SDP problems. The algorithm was implemented in a high-performance C++ code and all of the simulations below were run on a laptop with an Intel Core i7 quad-core 2.5 GHz CPU and 8 GB RAM. For more details about the C++ implementation and for the full code, please check Appendix.

For every $i \in \mathcal{V}$, we generate a random instance of the problem as follows:

- Each matrix A_i is chosen as $\Omega + \Omega^T + n_i I$, where the entries of Ω are uniformly chosen from the integer set $\{1, 2, 3, 4, 5\}$. This creates reasonably well-conditioned matrices A_i.
- Each matrix B_j(or D_l) is chosen as $\Omega + \Omega^T$, where Ω is generated as before.
- Each matrix variable W_i is assumed to be 40 by 40.
- The matrices $W_1, ..., W_n$ are assumed to overlap with each other in a banded structure, associated with a path graph $\mathcal{G}$ with the edges $(1, 2), (2, 3), ..., (n-1, n)$. One can regard W_i's as submatrices of a full-scale matrix variable W in the form of Figure 2.3 but with n overlapping blocks, where 25% of the entries of every two neighboring matrices W_i and W_{i+1} (leading to a 10×10 submatrix) overlaps.

In order to demonstrate the proposed algorithm on large-scale SDPs, three different values will be considered for the total number of overlapping blocks (or agents): 1000, 2000 and 4000. To give the reader a sense of how large the simulated SDPs are, the total number of entries of W_i's in the decomposed SDP problem (N_{Decomp}) and the total number of entries of W in the corresponding full-SDP problem (N_{Full}) are listed below:

- 1000 agents: $N_{\text{Full}} = 0.9$ billion, $N_{\text{Decomp}} = 1.6$ million
- 2000 agents: $N_{\text{Full}} = 3.6$ billion, $N_{\text{Decomp}} = 3.2$ million
- 4000 agents: $N_{\text{Full}} = 14.4$ billion, $N_{\text{Decomp}} = 6.4$ million

The simulation results are provided in Table 2.1 with the following entries: P_{obj} and D_{obj} are the primal and dual objective values, "iter" denotes the number of iterations needed to achieve a

Cases		1000	2000	4000
$p_i = 5$ $q_i = 0$	P_{obj}	4.010774e+05	8.004677e+05	1.607917e+06
	D_{obj}	4.010047e+05	8.003433e+05	1.607689e+06
	iter	308	348	368
	t_{CPU} (sec)	66.74	147.09	329.48
	t_{iter} (sec per iter)	0.22	0.42	0.90
	Optimality	99.98%	99.98%	99.98%
$p_i = 0$ $q_i = 5$	P_{obj}	8.119377e+05	1.626216e+06	3.249436e+06
	D_{obj}	8.119114e+05	1.626207e+06	3.249429e+06
	iter	1033	1360	1652
	t_{CPU} (sec)	230.48	579.95	1544.59
	t_{iter} (sec per iter)	0.22	0.43	0.93
	Optimality	99.996%	99.9994%	99.9997%
$p_i = 5$ $q_i = 5$	P_{obj}	1.192407e+06	2.373408e+06	4.741277e+06
	D_{obj}	1.192402e+06	2.373401e+06	4.741266e+06
	iter	2323	2754	2902
	t_{CPU} (sec)	525.312	1295.69	2940.62
	t_{iter} (sec per iter)	0.23	0.47	1.01
	Optimality	99.9995%	99.9997%	99.9997%

Table 2.1: Simulation results for three cases with 1000, 2000 and 4000 agents.

desired tolerance, t_{CPU} and t_{iter} are the total CPU time (in seconds) and the time per iteration (in seconds per iteration), and "Optimality" (in percentage) is calculated as:

$$\text{Optimality Degree } (\%) = 100 - \frac{P_{\text{obj}} - D_{\text{obj}}}{P_{\text{obj}}} \times 100$$

As shown in Table 2.1, the simulations were run for three cases:

- $p_i = 5$ and $q_i = 0$: each agent has 5 equality constraints and no inequality constraints.
- $p_i = 0$ and $q_i = 5$: each agent has no equality constraints and 5 inequality constraints.
- $p_i = 5$ and $q_i = 5$: each agent has 5 equality constraints and 5 inequality constraints.

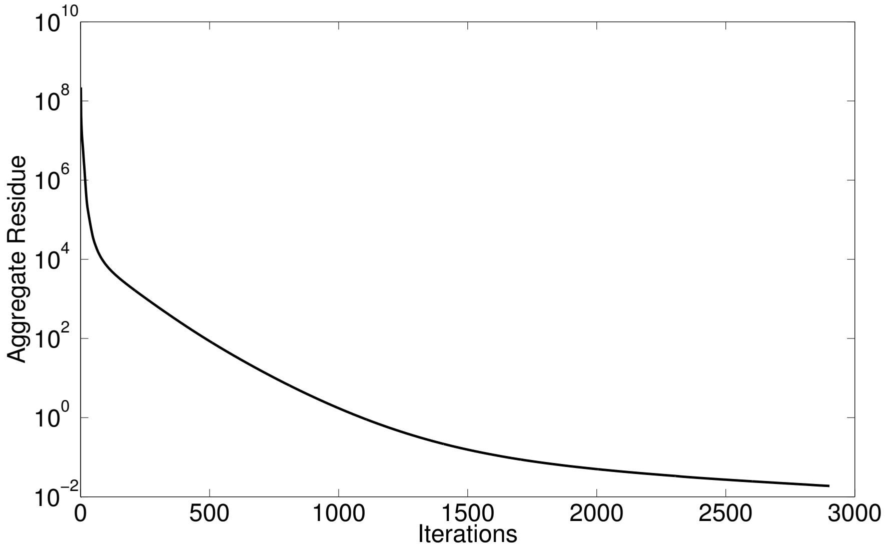


Figure 2.5: Aggregate residue for the case of 4000 agents with $p_i = q_i = 5$.

All solutions reported in Table 2.1 are based on the tolerance of 10^{-3} and an optimality degree of at least 99.9%. The aggregative residue V^t is plotted in Figure 2.5 for the 4000-agent case with $p_i = q_i = 5$, which is a monotonically decreasing function. Note that the time per iteration is between 0.22 and 1.01 in a C++ implementation. Efficient and computationally cheap preconditioning methods could dramatically reduce the number of iterations, but this is outside the scope of this work.

2.6 Summary

In this chapter, a fast and parallelizable algorithm is developed for an arbitrary decomposable semidefinite program (SDP). To formulate a decomposable SDP, we consider a multi-agent canonical form represented by a graph, where each agent (node) is in charge of computing its corresponding positive semidefinite matrix. The main goal of each agent is to ensure that its matrix is optimal with respect to some measure and satisfies local equality and inequality constraints. In addition, the matrices of two neighboring agents may be subject to overlapping constraints. The objective function of the optimization is the sum of all objectives of individual agents. The motivation behind this formulation is that an arbitrary sparse SDP problem can be converted to a decomposable SDP by means of the Chordal extension and matrix completion theorems. Using the alternating direction method of multipliers, we develop a distributed algorithm to solve the underlying SDP problem. At every iteration, each agent performs simple computations (matrix multiplication and eigenvalue decomposition) without having to solve any optimization subproblem, and then communicates some information to its neighbors. By deriving a Lyapunov-type non-increasing function, it is shown that the proposed algorithm converges as long as Slater's conditions hold. Simulations results on large-scale SDP problems with a few million variables are offered to elucidate the efficacy of the proposed technique.

Chapter 3

A Fast Parallelizable Algorithm for Convex Relaxation of Optimal Power Flow Problem

This chapter designs a distributed algorithm for solving the semidefinite programming (SDP) relaxation of the optimal power flow (OPF) problem, based on the alternating direction method of multipliers (ADMM). It is known that exploiting the sparsity of a large-scale SDP problem leads to a decomposed formulation with a lower computational cost. The algorithm proposed in this work deploys the sparsity of power networks and solves the decomposed formulation of the SDP problem using an ADMM scheme whose iterations consist of two subproblems. Both subproblems are highly parallelizable and enjoy closed-form solutions, which make the iterations computationally very cheap. While an arbitrary decomposable multi-agent SDP formulation was solved in the dual domain in Chapter 2, the sparse and large-scale SDP for the OPF problem is solved in the primal domain combined with the tree/chordal/clique decomposition technique in order to better exploit the structure of power systems. The numerical algorithm developed here is also tested on the IEEE benchmark systems.

3.1 Introduction

The optimal power flow (OPF) problem finds an optimal operating point of a power system by minimizing a certain objective function (e.g., transmission loss or generation cost) subject to power flow equations and operational constraints [8], [9]. Motivated by the importance of this fundamental problem for operation and planning as well as the potential monetary savings involved [10], many optimization techniques have been explored for the OPF problem. Due to the non-convexity and NP-hardness of OPF, the existing algorithms are not robust, lack performance guarantees and may not find a global optimum. With the goal of designing a polynomial-time algorithm that finds a global solution for OPF, [11] derives an SDP relaxation for OPF, which results in a globally optimal solution if the duality gap is zero. The proposed relaxation can find near-global solutions with global optimality guarantees of at least 99% for IEEE and Polish systems [12], and is theoretically proven to be exact under various assumptions [13], [14], [15], [16], [17], [18]. However, this relaxation is a high-dimensional SDP problem, which imposes some limitations on its practicality for real-world networks.

The emerging smart grid paradigm and the integration of intermittent and distributed power generation calls for the development of efficient, scalable, and parallel algorithms for solving large-scale OPF problems to enable real-time network management and improve the system's reliability. In response to this need, we aim to design an algorithm that is able to solve large-scale SDP relaxations. Early efforts to solve OPF in a distributed way (without considering non-convexity) can be traced back to [37], [38]. In [39], a fully decentralized ADMM-based algorithm is developed for a convex approximation of dynamic OPF. The papers [40] and [41] exploit primal-dual decomposition and ADMM methods for the SDP relaxation of OPF, but they need to solve an expensive SDP sub-problem at every iteration. The work [42] designs a distributed algorithm for a second-order cone relaxation of OPF over radial (acyclic) networks. In contrast to the existing methods, the algorithm to be proposed here applies to both distribution and transmission networks, and does not require solving any optimization sub-problem at any iteration.

While small- to medium-sized SDPs are efficiently solvable by second-order-based interior point methods in polynomial time up to any arbitrary precision [3], these methods are impractical for solving large-scale SDPs due to computation time and memory issues. A promising approach for solving large-scale SDP problems is ADMM. In light of the scalability of ADMM, the main objective of this work is to design an ADMM-based parallel algorithm for solving sparse large-scale SDPs tailored to the OPF problem with a guaranteed convergence under very mild assumptions. We start by defining a representative graph for the large-scale SDP problem, from which a decomposed SDP formulation is obtained using a tree/chordal/clique decomposition technique. This decomposition replaces the large-scale SDP matrix variable with certain submatrices of this matrix. In order to solve the decomposed SDP problem iteratively, a distributed ADMM-based algorithm is derived, whose iterations comprise entry-wise matrix multiplication/division and eigendecomposition on certain submatrices of the SDP matrix. By finding the optimal solution for the distributed SDP, one could recover the solution to the original large-scale SDP formulation using an explicit formula.

Similar to the work in Chapter 2, the work in this chapter is related to and improves upon the recent papers [33], [43], [34]. In contrast with the above papers, the algorithm proposed in this work is composed of low-complex and parallelizable iterations, which run fast if the treewidth of the representative graph of the SDP problem is small. Since this treewidth is low for real-world power networks, our algorithm is well suited for the SDP relaxation of power optimization problems.

This chapter is organized as follows. Some preliminaries and definitions are provided in Section 3.2. An arbitrary sparse SDP is converted into a decomposed SDP in Section 3.3, for which a numerical algorithm in the primal domain is developed in Section 3.4. The algorithm is used to solve the convex relaxation of the OPF problem in Section 3.5. Numerical examples are given in Section 3.6, followed by a summary in Section 3.7.

Notations: $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{H}^n$ denote the sets of real numbers, complex numbers, and $n \times n$ Hermitian matrices, respectively. The notation $\mathbf{X}_1 \circ \mathbf{X}_2$ refers to the Hadamard (entrywise) multiplication of matrices $\mathbf{X}_1$ and $\mathbf{X}_2$. The symbols $\langle \cdot, \cdot \rangle$ and $\| \cdot \|_F$ denote the Frobenius inner product and norm of matrices, respectively. The notation $\|\mathbf{v}\|_2$ denotes the ℓ_2-norm of a vector $\mathbf{v}$. The $m \times n$ rectangular identity matrix, whose (i, j) entry is equal to the Kronecker delta δ_{ij}, is denoted by $\mathbf{I}_{m\times n}$. The notations $\mathrm{Re}\{\mathbf{W}\}$, $\mathrm{Im}\{\mathbf{W}\}$, $\mathrm{rank}\{\mathbf{W}\}$, and $\mathrm{diag}\{\mathbf{W}\}$ denote the real part, imaginary part, rank, and diagonal of a Hermitian matrix $\mathbf{W}$, respectively. Given a vector $\mathbf{v}$, the notation $\mathrm{diag}\{\mathbf{v}\}$ denotes a diagonal square matrix whose entries are given by $\mathbf{v}$. The notation $\mathbf{W} \succeq 0$ means that $\mathbf{W}$ is Hermitian and positive semidefinite. The notation "$\mathbf{i}$" is reserved for the imaginary unit. The superscripts $(\cdot)^*$ and $(\cdot)^{\mathrm{T}}$ represent the conjugate transpose and transpose operators, respectively. Given a matrix $\mathbf{W}$, its (l, m) entry is denoted as W_{lm}. The subscript $(\cdot)_{\mathrm{opt}}$ is used to show the optimal value of an optimization variable. Given a matrix $\mathbf{W}$, its Moore-Penrose pseudoinverse is denoted as $\text{pinv}\{\mathbf{W}\}$. Given a simple graph $\mathcal{H}$, its vertex and edge sets are denoted by $\mathcal{V}_\mathcal{H}$ and $\mathcal{E}_\mathcal{H}$, respectively. Given two sets $\mathcal{S}_1$ and $\mathcal{S}_2$, the notation $\mathcal{S}_1 \backslash \mathcal{S}_2$ denotes the set of all elements of $\mathcal{S}_1$ that do not exist in $\mathcal{S}_2$. Given a Hermitian matrix $\mathbf{W}$ and two sets of positive integer numbers $\mathcal{S}_1$ and $\mathcal{S}_2$, define $\mathbf{W}\{\boldsymbol{\mathcal{S}_1}, \boldsymbol{\mathcal{S}_2}\}$ as a submatrix of $\mathbf{W}$ obtained through two operations: (i) removing all rows of $\mathbf{W}$ whose indices do not belong to $\mathcal{S}_1$, and (ii) removing all columns of $\mathbf{W}$ whose indices do not belong to $\mathcal{S}_2$. For instance, $\mathbf{W}\,\{\{\mathbf{1},\mathbf{2}\},\{\mathbf{2},\mathbf{3}\}\}$ is a 2×2 matrix with the entries $W_{12}, W_{13}, W_{22}, W_{23}$.

3.2 Preliminaries

Consider the semidefinite program

$$\begin{aligned}
&\underset{\mathbf{X}\in\mathbb{H}^n}{\text{minimize}} && \langle \mathbf{X}, \mathbf{M}_0 \rangle && && \text{(3.1a)}\\
&\text{subject to} && l_s \leq \langle \mathbf{X}, \mathbf{M}_s \rangle \leq u_s, && s = 1,\ldots,p, && \text{(3.1b)}\\
& && \mathbf{X} \succeq 0. && && \text{(3.1c)}
\end{aligned}$$

where $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_p \in \mathbb{H}^n$, and

$$(l_s, u_s) \in (\{-\infty\} \cup \mathbb{R}) \times (\mathbb{R} \cup \{+\infty\})$$

for every $s = 1, \ldots, p$. Notice that the constraint (3.1b) reduces to an equality constraint if $l_s = u_s$.

Problem (3.1) is computationally expensive for a large n due to the presence of the positive semidefinite constraint (3.1c). However, if $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_p$ are sparse, this expensive constraint can be decomposed and expressed in terms of some principal submatrices of $\mathbf{X}$ with smaller dimensions. This will be explained next.

3.2.1 Representative Graph and Tree Decomposition

In order to leverage any possible sparsity of problem (3.1), a simple graph shall be defined to capture the zero-nonzero patterns of $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_p$.

Definition 1. *Define $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ as the representative graph of the SDP problem* (3.1)*, which is a simple graph with n vertices whose edges are specified by the nonzero off-diagonal entries of*

$\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_p$. *In other words, two arbitrary vertices i and j are connected if the (i, j) entry of at least one of the matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_p$ is nonzero.*

Using a tree decomposition algorithm (also known as chordal or clique decomposition), we can obtain a *decomposed* formulation for problem (3.1), in which the positive semidefinite requirement is imposed on certain principal submatrices of $\mathbf{X}$ as opposed to $\mathbf{X}$ itself.

Definition 2 (Tree decomposition)**.** *A tree graph $\mathcal{T}$ is called a tree decomposition of $\mathcal{G}$ if it satisfies the following properties:*

1. *Every node of $\mathcal{T}$ corresponds to and is identified by a subset of $\mathcal{V}_\mathcal{G}$.*

2. *Every vertex of $\mathcal{G}$ is a member of at least one node of $\mathcal{T}$.*

3. *$\mathcal{T}_k$ is a connected graph for every $k \in \mathcal{V}_\mathcal{G}$, where $\mathcal{T}_k$ denotes the subgraph of $\mathcal{T}$ induced by all nodes of $\mathcal{T}$ containing the vertex k of $\mathcal{G}$.*

4. *The subgraphs $\mathcal{T}_i$ and $\mathcal{T}_j$ have a node in common for every $(i, j) \in \mathcal{E}_\mathcal{G}$.*

Each node of $\mathcal{T}$ is a bag (collection) of vertices of $\mathcal{G}$ and hence it is referred to as a **bag**.

Let $\mathcal{T} = (\mathcal{V}_\mathcal{T}, \mathcal{E}_\mathcal{T})$ be an arbitrary tree decomposition of $\mathcal{G}$, with the set of bags $\mathcal{V}_\mathcal{T} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_q\}$. As discussed in the next section, it is possible to cast problem (3.1) in terms of those entries of $\mathbf{X}$ that appear in at least one of the submatrices $\mathbf{X}\{\mathcal{C}_1, \mathcal{C}_1\}, \mathbf{X}\{\mathcal{C}_2, \mathcal{C}_2\}, \ldots, \mathbf{X}\{\mathcal{C}_q, \mathcal{C}_q\}$. These entries of X are referred to as *important entries.* Once the optimal values of the important entries of X are found using an arbitrary algorithm, the remaining entries can be obtained from an explicit (recursive) formula to be stated later.

Among the factors that may contribute to the computational complexity of the decomposed problem are: the size of the largest bag, the number of bags, and the total number of important entries. Finding a tree decomposition that leads to the minimum number of important entries (minimum fill-in problem) or possesses the minimum size for its largest bag (treewidth problem) is known to be NP-hard. Nevertheless, there are many efficient algorithms in the literature that find near-optimal tree decompositions (specially for power networks due to their near planarity) [44; 45].

3.2.2 Sparsity Pattern of Matrices

Let $\mathbb{F}^n$ denote the set of symmetric $n \times n$ matrices with entries belonging to the set $\{0, 1\}$. The distributed optimization scheme to be proposed in this work uses a group of sparse slack matrices. We identify the locations of nonzero entries of such matrix variables using descriptive matrices in $\mathbb{F}^n$.

Definition 3. *Given an arbitrary matrix $\mathbf{X} \in \mathbb{H}^n$, define its sparsity pattern as a matrix $\mathbf{N} \in \mathbb{F}^n$ such that $N_{ij} = 1$ if and only if $X_{ij} \neq 0$ for every $i, j \in \{1, ..., n\}$. Let $|\mathbf{N}|$ denote the number of nonzero entries of $\mathbf{N}$. Define the set*

$$\mathcal{S}(\mathbf{N}) \triangleq \{\mathbf{X} \in \mathbb{H}^n \mid \mathbf{X} \circ \mathbf{N} = \mathbf{X}\}.$$

Due to the Hermitian property of $\mathbf{X}$, if d denotes the number of nonzero diagonal entries of $\mathbf{N}$, then every $\mathbf{X} \in \mathcal{S}(\mathbf{N})$ can be specified by $(|\mathbf{N}| + d)/2$ real-valued scalars corresponding to $\mathrm{Re}\{\mathbf{X}\}$ and $(|\mathbf{N}| - d)/2$ real scalars corresponding to $\mathrm{Im}\{\mathbf{X}\}$. Therefore, $\mathcal{S}(\mathbf{N})$ is $|\mathbf{N}|$-dimensional over $\mathbb{R}$.

Definition 4. *Suppose that $\mathcal{T} = (\mathcal{V}_{\mathcal{T}}, \mathcal{E}_{\mathcal{T}})$ is a tree decomposition of the representative graph $\mathcal{G}$ with the bags $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_q$.*

- *For $r = 1, \ldots, q$, define $\mathbf{C}_r \in \mathbb{F}^n$ as a sparsity pattern whose (i, j) entry is equal to 1 if $\{i, j\} \subseteq \mathcal{C}_r$ and is 0 otherwise for every $i, j \in \{1, ..., n\}$.*
- *Define $\mathbf{C} \in \mathbb{F}^n$ as an aggregate sparsity pattern whose (i, j) entry is equal to 1 if and only if $\{i, j\} \subseteq \mathcal{C}_r$ for at least one index $r \in \{1, \ldots, p\}$.*
- *For $s = 0, 1, \ldots, p$, define $\mathbf{N}_s \in \mathbb{F}^n$ as the sparsity pattern of $\mathbf{M}_s$.*

The sparsity pattern $\mathbf{C}$, which can also be interpreted as the adjacency matrix of a chordal extension of $\mathcal{G}$ induced by $\mathcal{T}$, captures the locations of the important entries of $\mathbf{X}$. The matrix $\mathbf{C}$ will later be used to describe the domain of definition for the variable of decomposed SDP problem.

3.2.3 Indicator Functions

To streamline the formulation, we will replace any positivity or positive semidefiniteness constraints in the decomposed SDP problem by the indicator functions introduced below.

Definition 5. *For every $l \in \{-\infty\} \cup \mathbb{R}$ and $u \in \mathbb{R} \cup \{+\infty\}$, define the convex indicator function $\mathcal{I}_{l,u} : \mathbb{R} \rightarrow \{0, +\infty\}$ as*

$$\mathcal{I}_{l,u}(x) \triangleq \begin{cases} 0 & \text{if } \ l \leq x \leq u \\ +\infty & \text{otherwise} \end{cases}$$

Definition 6. *For every $r \in \{1, 2, \dots, q\}$, define the convex indicator function $\mathcal{J}_r : \mathbb{H}^n \rightarrow \{0, +\infty\}$ as*

$$\mathcal{J}_r(\mathbf{X}) \triangleq \begin{cases} 0 & \text{if } \ \mathbf{X}\{\mathcal{C}_r, \mathcal{C}_r\} \succeq 0 \\ +\infty & \text{otherwise} \end{cases}$$

3.3 Decomposed SDP

Consider the problem

$$\begin{aligned} & \underset{\mathbf{X} \in \mathcal{S}(\mathbf{C})}{\text{minimize}} && \langle \mathbf{X}, \mathbf{M}_0 \rangle && && (3.2a) \\ & \text{subject to} && l_s \leq \langle \mathbf{X}, \mathbf{M}_s \rangle \leq u_s, && s = 1, \dots, p, && (3.2b) \\ & && \mathbf{X}\{\mathcal{C}_r, \mathcal{C}_r\} \succeq 0, && r = 1, \dots, q && (3.2c) \end{aligned}$$

which is referred to as *decomposed SDP* throughout this chapter. Due to the chordal theorem [32], problems (3.1) and (3.2) lead to the same optimal objective value. Furthermore, if $\mathbf{X}_{\text{ref}} \in \mathcal{S}(\mathbf{C})$ denotes an arbitrary solution of the decomposed SDP problem (3.2), then there exists a solution $\mathbf{X}_{\text{opt}}$ to the SDP problem (3.1) such that $\mathbf{X}_{\text{opt}} \circ \mathbf{C} = \mathbf{X}_{\text{ref}}$.

To understand how $\mathbf{X}_{\text{opt}}$ can be constructed from $\mathbf{X}_{\text{ref}}$, observe that those entries of $\mathbf{X}$ corresponding to the zeros of $\mathbf{C}$ are 0 due to the relation $\mathbf{X}_{\text{ref}} \in \mathcal{S}(\mathbf{C})$. These entries of the matrix variable $\mathbf{X}$ that are needed for SDP but have not been found by decomposed SDP are referred to as *missing entries.* Several completion approaches can be adopted in order to recover these missing entries. An algorithm is proposed in [43; 46] that obtains a completion for $\mathbf{X}_{\text{ref}}$ within the set $\{\mathbf{X} \in \mathbb{H}^n \,|\, \mathbf{X} \circ \mathbf{C} = \mathbf{X}_{\text{ref}},\ \mathbf{X} \succeq 0\}$whose determinant is maximum. However such a solution may not be favorable for applications that require a low-rank solution such as an SDP relaxation. It is also known that there exists a polynomial-time algorithm to fill a partially-known real-valued matrix in such a way that the rank of the resulting matrix becomes equal to the highest rank among all bags [47; 48]. In [49], this result was extended to the complex domain by proposing a recursive algorithm that transforms $\mathbf{X}_{\text{ref}} \in \mathcal{S}(\mathbf{C})$ into a solution $\mathbf{X}_{\text{opt}}$ for the original SDP problem (3.1) whose rank is upper bounded by the maximum rank among the matrices $\mathbf{X}_{\text{ref}}\{\mathcal{C}_1, \mathcal{C}_1\}, \mathbf{X}_{\text{ref}}\{\mathcal{C}_2, \mathcal{C}_2\}, \ldots, \mathbf{X}_{\text{ref}}\{\mathcal{C}_q, \mathcal{C}_q\}$. This algorithm is stated below for completeness.

Matrix completion algorithm:

1. Set $\mathcal{T}' := \mathcal{T}$ and $\mathbf{X} := \mathbf{X}_{\text{ref}}$.
2. If $\mathcal{T}'$ has a single node, then consider $\mathbf{X}_{\text{opt}}$ as $\mathbf{X}$ and terminate; otherwise continue to the next step.
3. Choose a pair of bags $\mathcal{C}_x, \mathcal{C}_y$ of $\mathcal{T}'$ such that $\mathcal{C}_x$ is a leaf of $\mathcal{T}'$ and $\mathcal{C}_y$ is its unique neighbor.
4. Define

$$\mathbf{K} \triangleq \text{pinv}\{\mathbf{X}\{\mathcal{C}_x \cap \mathcal{C}_y, \mathcal{C}_x \cap \mathcal{C}_y\}\} \tag{3.3a}$$

$$\mathbf{G}_x \triangleq \mathbf{X}\{\mathcal{C}_x \setminus \mathcal{C}_y, \mathcal{C}_x \cap \mathcal{C}_y\} \tag{3.3b}$$

$$\mathbf{G}_y \triangleq \mathbf{X}\{\mathcal{C}_y \setminus \mathcal{C}_x, \mathcal{C}_x \cap \mathcal{C}_y\} \tag{3.3c}$$

$$\mathbf{E}_x \triangleq \mathbf{X}\{\mathcal{C}_x \setminus \mathcal{C}_y, \mathcal{C}_x \setminus \mathcal{C}_y\} \in \mathbb{C}^{d_x \times d_x} \tag{3.3d}$$

$$\mathbf{E}_y \triangleq \mathbf{X}\{\mathcal{C}_y \setminus \mathcal{C}_x, \mathcal{C}_y \setminus \mathcal{C}_x\} \in \mathbb{C}^{d_y \times d_y} \tag{3.3e}$$

$$\mathbf{S}_x \triangleq \mathbf{E}_x - \mathbf{G}_x \mathbf{K} \mathbf{G}_x^* = \mathbf{Q}_x \mathbf{D}_x \mathbf{Q}_x^* \tag{3.3f}$$

$$\mathbf{S}_y \triangleq \mathbf{E}_y - \mathbf{G}_y \mathbf{K} \mathbf{G}_y^* = \mathbf{Q}_y \mathbf{D}_y \mathbf{Q}_y^* \tag{3.3g}$$

   where $\mathbf{Q}_x \mathbf{D}_x \mathbf{Q}_x^*$ and $\mathbf{Q}_y \mathbf{D}_y \mathbf{Q}_y^*$ denote the eigenvalue decompositions of $\mathbf{S}_x$ and $\mathbf{S}_y$ with the diagonals of $\mathbf{D}_x$ and $\mathbf{D}_y$ arranged in descending order. Then, update a part of $\mathbf{X}$ as follows:

$$\mathbf{X}\{\mathcal{C}_y \setminus \mathcal{C}_x, \mathcal{C}_x \setminus \mathcal{C}_y\} := \mathbf{G}_y \mathbf{K} \mathbf{G}_x^* + \mathbf{Q}_y \sqrt{\mathbf{D}_y}\ \mathbf{I}_{d_y \times d_x} \sqrt{\mathbf{D}_x}\ \mathbf{Q}_x^* \tag{3.4}$$

   and update $\mathbf{X}\{\mathcal{C}_x \setminus \mathcal{C}_y, \mathcal{C}_y \setminus \mathcal{C}_x\}$ accordingly to preserve the Hermitian property of $\mathbf{X}$.
5. Update $\mathcal{T}'$ by merging $\mathcal{C}_x$ into $\mathcal{C}_y$, i.e., replace $\mathcal{C}_y$ with $\mathcal{C}_x \cup \mathcal{C}_y$ and then remove $\mathcal{C}_x$ from $\mathcal{T}'$.
6. Go back to step 2.

Theorem 2. *Consider an arbitrary solution* $\mathbf{X}_{\text{ref}}$ *of the decomposed SDP problem* (3.2). *The output of the matrix completion algorithm, denoted as* $\mathbf{X}_{\text{opt}}$, *is a solution of the original SDP problem* (3.1). *Moreover, the rank of* $\mathbf{X}_{\text{opt}}$ *is smaller than or equal to:*

$$\max\left\{\operatorname{rank}\{\mathbf{X}_{\text{ref}}\{\mathcal{C}_r,\mathcal{C}_r\}\}\;\middle|\; r=1,\ldots,q\right\}.$$

Proof. See [49; 50] for the proof. □

3.4 Alternating Direction Method of Multipliers

For the convenience of the reader, the ADMM algorithm is restated in this section. Consider the optimization problem

$$\begin{aligned} &\underset{\substack{\mathbf{x}\in\mathbb{R}^{n_x}\\ \mathbf{y}\in\mathbb{R}^{n_y}}}{\text{minimize}} && f(\mathbf{x})+g(\mathbf{y}) && \text{(3.5a)}\\ &\text{subject to} && \mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{y}=\mathbf{c}. && \text{(3.5b)}\end{aligned}$$

where $\mathbf{c}\in\mathbb{R}^{n_c}$, $\mathbf{A}\in\mathbb{R}^{n_c\times n_x}$ and $\mathbf{B}\in\mathbb{R}^{n_c\times n_y}$ are given matrices. Also $f:\mathbb{R}^{n_x}\to\mathbb{R}\cup\{+\infty\}$ and $g:\mathbb{R}^{n_y}\to\mathbb{R}\cup\{+\infty\}$ are given convex functions. Notice that the variables $\mathbf{x}$ and $\mathbf{y}$ are coupled through the linear constraint (3.5b) while the objective function is separable.

The augmented Lagrangian function for problem (3.5) is equal to

$$\begin{aligned}\mathcal{L}_\mu(\mathbf{x},\mathbf{y},\lambda) &= f(\mathbf{x})+g(\mathbf{y}) && \text{(3.6a)}\\ &+\lambda^{\mathrm{T}}(\mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{y}-\mathbf{c}) && \text{(3.6b)}\\ &+(\mu/2)\|\mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{y}-\mathbf{c}\|_2^2, && \text{(3.6c)}\end{aligned}$$

where $\lambda\in\mathbb{R}^{n_c}$ is the Lagrange multiplier associated with the constraint (3.5b), and $\mu\in\mathbb{R}$ is a fixed parameter. ADMM is one approach for solving problem (3.5), which performs the following procedure at each iteration [7]:

$$\begin{aligned}\mathbf{x}^{k+1} &= \underset{\mathbf{x}\in\mathbb{R}^{n_x}}{\arg\min}\quad \mathcal{L}_\mu(\mathbf{x},\mathbf{y}^k,\lambda^k), && \text{(3.7a)}\\ \mathbf{y}^{k+1} &= \underset{\mathbf{y}\in\mathbb{R}^{n_y}}{\arg\min}\quad \mathcal{L}_\mu(\mathbf{x}^{k+1},\mathbf{y},\lambda^k), && \text{(3.7b)}\\ \lambda^{k+1} &= \lambda^k+\mu(\mathbf{A}\mathbf{x}^{k+1}+\mathbf{B}\mathbf{y}^{k+1}-\mathbf{c}). && \text{(3.7c)}\end{aligned}$$

where $k = 0, 1, 2, \ldots$, for an arbitrary initialization $(\mathbf{x}^0, \mathbf{y}^0, \lambda^0)$. In these equations, "argmin" means an arbitrary minimizer of a convex function and does not need any uniqueness assumption. Notice that each of the updates (3.7a) and (3.7b) is an optimization sub-problem with respect to either $\mathbf{x}$ and $\mathbf{y}$, by freezing the other variable at its latest value. We employ the energy sequence $\{\varepsilon^k\}_{k=1}^{\infty}$ proposed in [24] as measure for convergence:

$$\varepsilon^{k+1} = (1/\mu)\|\lambda^{k+1} - \lambda^k\|_2^2 + \mu\|\mathbf{B}(y^{k+1} - y^k)\|_2^2 \tag{3.8}$$

ADMM is particularly interesting for the cases where (3.7a) and (3.7b) can be performed efficiently through an explicit formula. Under such circumstances, it would be possible to execute a large number of iterations in a short amount of time. In this section, we first cast the decomposed SDP problem (3.2) in the form of (3.5) and then regroup the variables into two blocks $\mathcal{P}_1$ and $\mathcal{P}_2$ playing the roles of $\mathbf{x}$ and $\mathbf{y}$ in the ADMM algorithm.

3.4.1 Projection Into Positive Semidefinite Cone

The algorithm to be proposed in this work requires the projection of q matrices belonging to $\mathbb{H}^{|\mathcal{C}_1|}, \mathbb{H}^{|\mathcal{C}_2|}, \ldots, \mathbb{H}^{|\mathcal{C}_q|}$ onto the positive semidefinite cone. This is probably the most computationally expensive part of each iteration.

Definition 7. *For a given Hermitian matrix $\widehat{\mathbf{Z}}$, define the unique solution to the optimization problem*

$$\begin{aligned} &\underset{\mathbf{Z}\in\mathbb{H}^m}{\text{minimize}} && \|\mathbf{Z} - \widehat{\mathbf{Z}}\|_F^2 && (3.9\text{a}) \\ &\text{subject to} && \mathbf{Z} \succeq 0 && (3.9\text{b}) \end{aligned}$$

as the projection of $\widehat{\mathbf{Z}}$ onto the cone of positive semidefinite matrices, and denote it as $\widehat{\mathbf{Z}}^+$.

The next Lemma reveals the interesting fact that problem (3.9) can be solved through an eigenvalue decomposition of $\widehat{\mathbf{Z}}$.

Lemma 1. *Let $\widehat{\mathbf{Z}} = \mathbf{Q} \times \operatorname{diag}\{(\nu_1 \ldots, \nu_m)\} \times \mathbf{Q}^*$ denote the eigenvalue decomposition of $\widehat{\mathbf{Z}}$. The solution of the projection problem* (3.9) *is given by*

$$\widehat{\mathbf{Z}}^+ = \mathbf{Q} \times \operatorname{diag}\{(\max\{\nu_1, 0\}, \ldots, \max\{\nu_m, 0\})\} \times \mathbf{Q}^*$$

Proof. See [51] for the proof. □

3.4.2 ADMM for Decomposed SDP

We apply ADMM to the following reformulation of the decomposed SDP problem (3.2):

$$\begin{aligned}
\underset{\substack{\mathbf{X}\in\mathcal{S}(\mathbf{C})\\ \{\mathbf{X}_{N;s}\in\mathcal{S}(\mathbf{N}_s)\}_{s=0}^{p}\\ \{\mathbf{X}_{C;r}\in\mathcal{S}(\mathbf{C}_r)\}_{r=1}^{q}\\ \{z_s\in\mathbb{R}\}_{s=0}^{p}}}{\text{minimize}} \quad & z_0 + \sum_{s=1}^{p}\mathcal{I}_{l_s,u_s}(z_s) + \sum_{r=1}^{q}\mathcal{J}_r(\mathbf{X}_{C;r}) \\
\text{subject to} \quad & \mathbf{X}\circ\mathbf{C}_r = \mathbf{X}_{C;r}, \qquad r = 1,2,\ldots,q, && \text{(3.10a)}\\
& \mathbf{X}\circ\mathbf{N}_s = \mathbf{X}_{N;s}, \qquad s = 0,1,\ldots,p, && \text{(3.10b)}\\
& z_s = \langle \mathbf{M}_s, \mathbf{X}_{N;s}\rangle, \qquad s = 0,1,\ldots,p. && \text{(3.10c)}
\end{aligned}$$

If $\mathbf{X}$ is a feasible solution of (3.10) with a finite objective value, then

$$\mathcal{J}_r(\mathbf{X}) = \mathcal{J}_r(\mathbf{X}\circ\mathbf{C}_r) \overset{(3.10a)}{=} \mathcal{J}_r(\mathbf{X}_{C;r}) = 0$$

which concludes that $\mathbf{X}\{\mathcal{C}_r,\mathcal{C}_r\}\succeq 0$. Also,

$$\begin{aligned}
\mathcal{I}_{l_s,u_s}(\langle\mathbf{X},\mathbf{M}_s\rangle) &= \mathcal{I}_{l_s,u_s}(\langle\mathbf{X}\circ\mathbf{N}_s,\mathbf{M}_s\rangle)\\
&\overset{(3.10b)}{=} \mathcal{I}_{l_s,u_s}(\langle\mathbf{X}_{N;s},\mathbf{M}_s\rangle)\\
&\overset{(3.10c)}{=} \mathcal{I}_{l_s,u_s}(z_s) = 0
\end{aligned}$$

which yields that $l_s \le \langle\mathbf{X},\mathbf{M}_s\rangle \le u_s$. Therefore, $\mathbf{X}$ is a feasible point for problem (3.2) as well, with the same objective value. Define

1. $\mathbf{\Lambda}_{C;r}\in\mathcal{S}(\mathbf{C}_r)$ as the Lagrange multiplier associated with the constraint (3.10a) for $r = 1,2,\ldots,q$,
2. $\mathbf{\Lambda}_{N;s}\in\mathcal{S}(\mathbf{N}_s)$ as the Lagrange multiplier associated with the constraint (3.10b) for $s = 0,1,\ldots,p$,
3. $\lambda_{z;s}\in\mathbb{R}$ as the Lagrange multiplier associated with the constraint (3.10c) for $s = 0,1,\ldots,p$.

We regroup the primal and dual variables as

$$\begin{aligned}
&\text{(Block 1)} \quad \mathcal{P}_1 = (\mathbf{X},\{z_s\}_{s=0}^{p})\\
&\text{(Block 2)} \quad \mathcal{P}_2 = (\{\mathbf{X}_{C;r}\}_{r=1}^{q},\{\mathbf{X}_{N;s}\}_{s=0}^{p})\\
&\text{(Dual)} \quad \mathcal{D} = (\{\mathbf{\Lambda}_{C;r}\}_{r=1}^{q},\{\mathbf{\Lambda}_{N;s}\}_{s=0}^{p},\{\lambda_{z;s}\}_{s=0}^{p}).
\end{aligned}$$

Note that "block 1", "block 2" and "$\mathcal{D}$" play the roles of $\mathbf{x}$, $\mathbf{y}$ and λ in the standard formulation of ADMM, respectively. The augmented Lagrangian can be calculated as

$$\begin{aligned}(2/\mu)\mathcal{L}_\mu(\mathcal{P}_1,\mathcal{P}_2,\mathcal{D}) = \mathcal{L}_D(\mathcal{D})/\mu^2 &+ \|z_0 - \langle \mathbf{M}_0, \mathbf{X}_{N;0}\rangle + (1+\lambda_{z;0})/\mu\|_F^2 \\ &+ \sum_{s=1}^{p} \|z_s - \langle \mathbf{M}_s, \mathbf{X}_{N;s}\rangle + \lambda_{z;s}/\mu\|_F^2 + (2/\mu)\mathcal{I}_{l_s,u_s}(z_s) \\ &+ \sum_{r=1}^{q} \|\mathbf{X}\circ\mathbf{C}_r - \mathbf{X}_{C;r} + (1/\mu)\mathbf{\Lambda}_{C;r}\|_F^2 + (2/\mu)\mathcal{J}_r(\mathbf{X}_{C;r}) \\ &+ \sum_{s=1}^{p} \|\mathbf{X}\circ\mathbf{N}_s - \mathbf{X}_{N;s} + (1/\mu)\mathbf{\Lambda}_{N;s}\|_F^2 \end{aligned} \tag{3.12}$$

where

$$\mathcal{L}_D(\mathcal{D}) = -(1+\lambda_{z;0})^2 - \sum_{s=1}^{p}\lambda_{z;s}^2 - \sum_{r=1}^{q}\|\mathbf{\Lambda}_{C;r}\|_F^2 - \sum_{s=1}^{p}\|\mathbf{\Lambda}_{N;s}\|_F^2$$

Using the blocks $\mathcal{P}_1$ and $\mathcal{P}_2$, the ADMM iterations for problem (3.10) can be expressed as follows:

1. The subproblem (3.7a) in terms of $\mathcal{P}_1$ consists of two parallel steps:
   (a) *Minimization in terms of* $\mathbf{X}$: This step consists of $|\mathbf{C}|$ scalar quadratic and unconstrained programs. It possesses an explicit formula that involves $|\mathbf{C}|$ parallel multiplication operations.
   (b) *Minimization in terms of* $\{z_s\}_{s=0}^{p}$: This step consists of $p+1$ scalar quadratic programs each with a box constraint. It possesses an explicit formula that involves $p+1$ parallel multiplication operations.
2. The subproblem (3.7b) in terms of $\mathcal{P}_2$ also consists of two parallel steps:
   (a) *Minimization in terms of* $\{\mathbf{X}_{C;r}\}_{r=1}^{q}$: This step consists of q projection problems of the form (3.9). According to Lemma 1, this reduces to q parallel eigenvalue decomposition operations on matrices of sizes $|\mathcal{C}_r| \times |\mathcal{C}_r|$ for $r = 1,\ldots,q$.
   (b) *Minimization in terms of* $\{\mathbf{X}_{N;s}\}_{s=0}^{p}$: This step consists of p unconstrained quadratic programs of sizes $|\mathbf{N}_s|$ for $s = 0,1,\ldots,p$. The quadratic programs are parallel and each of them possesses an explicit formula that involves $2|\mathbf{N}_s|$ multiplications.

3. Computation of the dual variables at each iteration, in equation (3.7c), consists of three parallel steps:
    (a) *Updating* $\{\mathbf{\Lambda}_{C;r}\}_{r=1}^{q}$: Computational costs for this step involves no multiplications and is negligible.
    (b) *Updating* $\{\mathbf{\Lambda}_{N;s}\}_{s=0}^{p}$: Computational costs for this step involves no multiplications and is negligible.
    (c) *Updating* $\{\lambda_{z;s}\}_{s=0}^{p}$: This step is composed of $p+1$ parallel inner product computations, each involving $|\mathbf{N}_s|$ multiplications for $s = 0, 1, \ldots, p$.

The fact that every step of the above algorithm has an explicit easy-to-compute formula makes the algorithm very appealing for large-scale SDPs.

Notation 1. *For every* $\mathbf{D}, \mathbf{E} \in \mathbb{H}^n$*, the notation* $\mathbf{D} \oslash_{\mathbf{C}} \mathbf{E}$ *refers to the entrywise division of those entries of* $\mathbf{D}$ *and* $\mathbf{E}$ *that correspond to the ones of* $\mathbf{C}$ *i.e.,*

$$(\mathbf{D} \oslash_{\mathbf{C}} \mathbf{E})_{ij} \triangleq \begin{cases} D_{ij}/E_{ij} & \text{if } C_{ij} = 1 \\ 0 & \text{if } C_{ij} = 0. \end{cases}$$

Theorem 3. *Assume that Slater's conditions hold for the decomposable SDP problem* (3.2) *and consider the iterative algorithm given in* (3.18)*. The limit of* $\mathbf{X}^k$ *at* $k = +\infty$ *is an optimal solution for* (3.2)*.*

Proof. The convergence of both primal and dual variables is guaranteed for a standard ADMM problem if the matrix $\mathbf{B}$ in (3.5b) has full column rank [35]. After realizing that (3.18) is obtained from a two-block ADMM procedure, the theorem can be concluded form the fact that the equivalent of $\mathbf{B}$ for the algorithm (3.18) is a mapping from the variables $\{\mathbf{X}_{C;r}\}_{r=1}^{q}$ and $\{\mathbf{X}_{N;s}\}_{s=0}^{p}$ to

$$\{\mathbf{X}_{C;r}\}_{r=1}^{q}, \{\mathbf{X}_{N;s}\}_{s=0}^{p} \quad \text{and} \quad \{\langle \mathbf{M}_s, \mathbf{X}_{N;s} \rangle\}_{s=0}^{p}$$

which is not singular, i.e., it has full column rank. The details are omitted for brevity. □

In what follows, we elaborate on every step of the ADMM iterations:

Block 1: The first step of the algorithm that corresponds to (3.7a) consists of the operation

$$\mathcal{P}_1^{k+1} := \arg\min \quad \mathcal{L}_\mu(\mathcal{P}_1, \mathcal{P}_2^k, \mathcal{D}^k).$$

Notice that the minimization of $\mathcal{L}_\mu(\mathcal{P}_1, \mathcal{P}_2^k, \mathcal{D}^k)$ with respect to $\mathcal{P}_1$ is decomposable in terms of the real scalars

$$\text{Re}\{X_{ij}\} \quad \text{for} \quad i = 1, \ldots, n; \quad j = i, \ldots, n \tag{3.14a}$$

$$\text{Im}\{X_{ij}\} \quad \text{for} \quad i = 1, \ldots, n; \quad j = i+1, \ldots, n \tag{3.14b}$$

$$z_s \quad \text{for} \quad s = 1, \ldots, p \tag{3.14c}$$

which leads to the explicit formulas (3.18a), (3.18b) and (3.18c).

Block 2: The second step of the algorithm that corresponds to (3.7b) consists of the operation

$$\mathcal{P}_2^{k+1} = \arg\min \quad \mathcal{L}_\mu(\mathcal{P}_1^{k+1}, \mathcal{P}_2, \mathcal{D}^k)$$

Notice that the minimization of $\mathcal{L}_\mu(\mathcal{P}_1^{k+1}, \mathcal{P}_2, \mathcal{D}^k)$ with respect to $\mathcal{P}_2$ is decomposable in terms of the matrix variables $\{\mathbf{X}_{C;r}\}_{r=1}^q$ and $\{\mathbf{X}_{N;s}\}_{s=0}^p$. Hence, the update of $\mathbf{X}_{C;r}$ reduces to the problem (3.9) for $\widehat{\mathbf{Z}} = \mathbf{X}_{C;r}\{\mathcal{C}_r, \mathcal{C}_r\}$. As shown in Lemma 1, this can be performed via the eigenvalue decomposition of a $|\mathcal{C}_r| \times |\mathcal{C}_r|$ matrix. In addition, the updated value of $\mathbf{X}_{N;s}$ is a minimizer of the function

$$\mathcal{L}_{N;s}(\mathbf{Z}) = \|z_s - \langle \mathbf{M}_s, \mathbf{Z}\rangle + \lambda_{z;s}/\mu\|_F^2 + \|\mathbf{X} \circ \mathbf{N}_s - \mathbf{Z} + (1/\mu)\mathbf{\Lambda}_{N;s}\|_F^2 \tag{3.16}$$

By taking the derivatives of this function, it is possible to find an explicit formula for $\mathbf{Z}_{\text{opt}}$. Define $\mathcal{L}'_{N;s}(\mathbf{Z}) \in \mathcal{S}(\mathbf{N}_s)$ as the gradient of $\mathcal{L}_{N;s}(\mathbf{Z})$ with the following structure:

$$\mathcal{L}'_{N;s}(\mathbf{Z}) \triangleq \left[\frac{\partial \mathcal{L}_{N;s}}{\partial \text{Re}\{Z_{ij}\}} + \mathbf{i}\frac{\partial \mathcal{L}_{N;s}}{\partial \text{Im}\{Z_{ij}\}}\right]_{i,j=1,\ldots,n}$$

Then, we have

$$\begin{aligned}\mathcal{L}'_{N;s}(\mathbf{Z})/2 = \mathbf{Z} - \mathbf{X} \circ \mathbf{N}_s - (1/\mu)\mathbf{\Lambda}_{N,s}& \\ + (-z_s + \langle \mathbf{M}_s, \mathbf{Z}\rangle - \lambda_{z;s}/\mu)\mathbf{M}_s&.\end{aligned}$$

Therefore,

$$\mathbf{Z}_{\text{opt}} = \mathbf{X} \circ \mathbf{N}_s + (1/\mu)\mathbf{\Lambda}_{N,s} + y_s\mathbf{M}_s, \tag{3.17}$$

where $y_s \triangleq z_s - \langle \mathbf{M}_s, \mathbf{Z}^{\text{opt}}\rangle + \lambda_{z;s}/\mu$. Hence, it only remains to derive the scalar y_s, which can be done by inner multiplying $\mathbf{M}_s$ to the both sides of the equation (3.17). This leads to the equations (3.18e) and (3.18f).

ADMM for Decomposed SDP:

Block 1 :

$$\mathbf{X}^{k+1} := \left[\sum_{r=1}^{q} \mathbf{C}_r \circ (\mathbf{X}^k_{C;r} - \mathbf{\Lambda}^k_{C;r}/\mu) + \sum_{s=1}^{p} \mathbf{N}_s \circ (\mathbf{X}^k_{N;s} - \mathbf{\Lambda}^k_{N;s}/\mu)\right] \oslash_{\mathbf{C}} \left[\sum_{r=1}^{q} \mathbf{C}_r + \sum_{s=1}^{p} \mathbf{N}_s\right] \tag{3.18a}$$

$$z_0^{k+1} := \langle \mathbf{M}_0, \mathbf{X}^k_{N;0}\rangle - (\lambda^k_{z;0} + 1)/\mu \tag{3.18b}$$

$$z_s^{k+1} := \max\{\min\{\langle \mathbf{M}_s, \mathbf{X}^k_{N;s}\rangle - \lambda^k_{z;s}/\mu, u_s\}, l_s\} \qquad \text{for} \quad s = 1, 2, \ldots, p \tag{3.18c}$$

Block 2 :

$$\mathbf{X}^{k+1}_{C;r} := (\mathbf{X}^{k+1} \circ \mathbf{C}_r + \mathbf{\Lambda}^k_{C;r}/\mu)^+ \qquad \text{for} \quad r = 1, 2, \ldots, q \tag{3.18d}$$

$$y_s^{k+1} := \frac{z_s^{k+1} + \lambda^k_{z;s}/\mu - \langle \mathbf{M}_s, \mathbf{N}_s \circ \mathbf{X}^{k+1} + \mathbf{\Lambda}^k_{N;s}/\mu\rangle}{1 + \|\mathbf{M}_s\|_F^2} \qquad \text{for} \quad s = 0, 1, \ldots, p \tag{3.18e}$$

$$\mathbf{X}^{k+1}_{N;s} := \mathbf{N}_s \circ \mathbf{X}^{k+1} + \mathbf{\Lambda}^k_{N,s}/\mu + y_s^{k+1}\mathbf{M}_s \qquad \text{for} \quad s = 0, 1, \ldots, p \tag{3.18f}$$

Dual :

$$\mathbf{\Lambda}^{k+1}_{C;r} := \mathbf{\Lambda}^k_{C;r} + \mu(\mathbf{X}^{k+1} \circ \mathbf{C}_r - \mathbf{X}^{k+1}_{C;r}) \qquad \text{for} \quad r = 1, 2, \ldots, q \tag{3.18g}$$

$$\mathbf{\Lambda}^{k+1}_{N;s} := \mathbf{\Lambda}^k_{N;s} + \mu(\mathbf{X}^{k+1} \circ \mathbf{N}_s - \mathbf{X}^{k+1}_{N;s}) \qquad \text{for} \quad s = 0, 1, \ldots, p \tag{3.18h}$$

$$\lambda^{k+1}_{z;s} := \lambda^k_{z;s} + \mu(z_s^{k+1} - \langle \mathbf{M}_s, \mathbf{X}^{k+1}_{N;s}\rangle) \qquad \text{for} \quad s = 0, 1, \ldots, p \tag{3.18i}$$

3.5 Optimal Power Flow

Consider an n-bus electrical power network with the topology described by a simple graph $\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$, meaning that each vertex belonging to $\mathcal{V}_\mathcal{H} = \{1, \ldots, n\}$ represents a node of the network and each edge belonging to $\mathcal{E}_\mathcal{G}$ represents a transmission line. Let $\mathbf{Y} \in \mathbb{C}^{n\times n}$ denote the admittance matrix of the network. Define $\mathbf{V} \in \mathbb{C}^n$ as the voltage phasor vector, i.e., V_k is the voltage phasor for node $k \in \mathcal{V}_\mathcal{H}$. Let $\mathbf{P} + \mathbf{Q}\ \mathbf{i}$ represent the nodal complex power vector, where $\mathbf{P} \in \mathbb{R}^n$ and $\mathbf{Q} \in \mathbb{R}^n$ are the vectors of active and reactive powers injected at all buses. $\mathbf{P} + \mathbf{Q}\ \mathbf{i}$ can be interpreted as the complex-power supply minus the complex-power demand at node k of the network. The classical OPF problem can be described as follows:

$$
\begin{aligned}
\underset{\substack{\mathbf{V}\in\mathbb{C}^n\\ \mathbf{Q}\in\mathbb{R}^n\\ \mathbf{P}\in\mathbb{R}^n}}{\text{minimize}} \quad & \sum_{k\,\in\,\mathcal{V}_\mathcal{G}} f_k(P_k) && && \text{(3.19a)}\\
\text{subject to} \quad & V_k^{\min} \leq |V_k| \leq V_k^{\max}, && k\in\mathcal{N} && \text{(3.19b)}\\
& Q_k^{\min} \leq\; Q_k \;\leq Q_k^{\max}, && k\in\mathcal{N} && \text{(3.19c)}\\
& P_k^{\min} \leq\; P_k \;\leq P_k^{\max} && k\in\mathcal{N} && \text{(3.19d)}\\
& \mathbf{P}+\mathbf{iQ} = \operatorname{diag}\{\mathbf{V}\mathbf{V}^*\mathbf{Y}^*\} && && \text{(3.19e)}
\end{aligned}
$$

where $V_k^{\min}$, $V_k^{\max}$, $P_k^{\min}$, $P_k^{\max}$, $Q_k^{\min}$ and $Q_k^{\max}$ are given network limitations, and $f_k(P_k)$ is a convex function accounting for the power generation cost at node k. This problem may include additional constraints (such as thermal limits over the lines) that are ignored here only for the sake of simplicity in the presentation. For the same reason, assume that the objective function is the total active power loss $\sum_{k\,\in\,\mathcal{V}_\mathcal{G}} P_k$. More details on a general formulation may be found in [11].

OPF is a highly non-convex problem, which is known to be difficult to solve in general. However, the constraints of problem (3.19) can all be expressed as linear functions of the entries of the quadratic matrix $\mathbf{V}\mathbf{V}^*$. This implies that the constraints of OPF are linear in terms of a matrix variable $\mathbf{W} \triangleq \mathbf{V}\mathbf{V}^*$. One can reformulate OPF by replacing each $V_iV_j^*$ by W_{ij} and represent the constraints in the form of problem (3.1) with a representative graph that is isomorphic to the network topology graph $\mathcal{H}$. In order to preserve the equivalence of the two formulations, two additional constraints must be added to the problem: (i) $\mathbf{W} \succeq 0$, (ii) $\operatorname{rank}\{\mathbf{W}\} = 1$. If we drop the rank condition as the only non-convex constraint of the reformulated OPF problem, we attain the SDP relaxation of OPF that is convex:

$$
\begin{aligned}
\underset{\mathbf{W}\in\mathbb{H}^n}{\text{minimize}} \quad & \langle \mathbf{W}, (\mathbf{Y}+\mathbf{Y}^*)/2\rangle && && \text{(3.20a)}\\
\text{subject to} \quad & (V_k^{\min})^2 \leq \langle \mathbf{W}, e_k e_k^*\rangle \leq (V_k^{\max})^2, && k\in\mathcal{V}_\mathcal{H} && \text{(3.20b)}\\
& Q_k^{\min} \leq \langle \mathbf{W}, \mathbf{Y}_{Q;k}\rangle \leq Q_k^{\max}, && k\in\mathcal{V}_\mathcal{H} && \text{(3.20c)}\\
& P_k^{\min} \leq \langle \mathbf{W}, \mathbf{Y}_{P;k}\rangle \leq P_k^{\max}, && k\in\mathcal{V}_\mathcal{H} && \text{(3.20d)}\\
& \mathbf{W} \succeq 0 && && \text{(3.20e)}
\end{aligned}
$$

Test cases	p	q	**Maximum size of bags**	**Running time of 1000 iterations (sec)**
Chow's 9 bus	27	7	3	6.18
IEEE 14 bus	42	12	3	9.96
IEEE 30 bus	90	18	4	14.66
IEEE 57 bus	171	26	6	21.25
IEEE 118 bus	354	66	5	53.13
IEEE 300 bus	900	111	7	98.95

Table 3.1: Running time of the proposed algorithm for solving the SDP relaxation of OPF problem on IEEE test cases.

where $e_1, \ldots, e_n$ denote the standard basis vectors in $\mathbb{R}^n$ and

$$\begin{aligned}\mathbf{Y}_{Q;k} &\triangleq \frac{1}{2\mathbf{i}}(\mathbf{Y}_k^* e_k e_k^* - e_k e_k^* \mathbf{Y}) \\ \mathbf{Y}_{P;k} &\triangleq \frac{1}{2}(\mathbf{Y}^* e_k e_k^* + e_k e_k^* \mathbf{Y})\end{aligned}$$

for every $k \in \mathcal{V}_{\mathcal{H}}$.

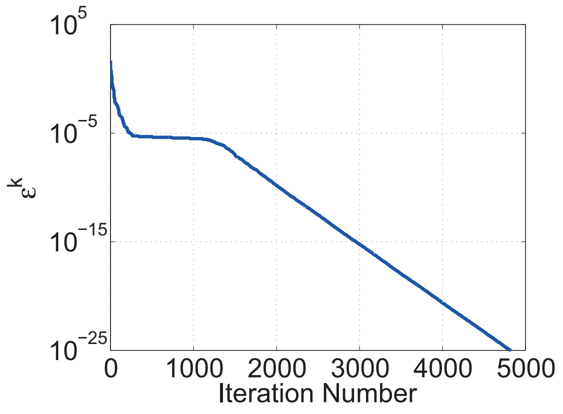
As stated in the introduction, several papers in the literature have shown great promises for finding global or near-global solutions of OPF using the above relaxation. The major drawback of relaxing the OPF problem to an SDP is the requirement of defining a matrix variable, which makes the number of scalar variables of the problem quadratic with respect to the number of network buses. However, we have shown in [50] that real-world grids would have a low treewidth, e.g., at most 26 for the Polish test system with over 3000 buses. This makes our proposed numerical algorithm scalable and highly parallelizable for the above SDP relaxation. As an example, the SDP relaxation of OPF for the Polish Grid amounts to simple operations over matrices of size 27 by 27 or smaller.

3.6 Simulation Results

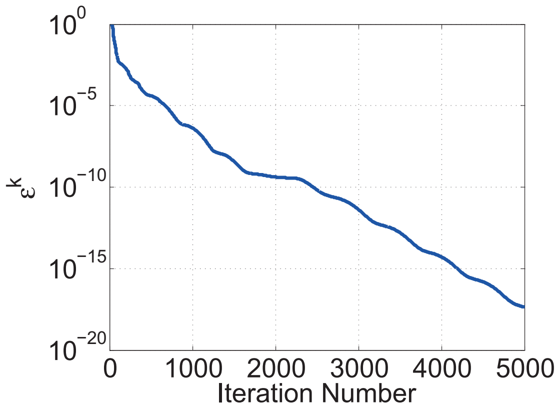
In this section, we evaluate the performance of the proposed algorithm for solving the SDP relaxation of OPF over IEEE test cases. All simulations are run in MATLAB using a laptop with an Intel Core i7 quad-core 2.5 GHz CPU and 12 GB RAM. As shown in Figure 3.1, the energy function ε^k (as defined in (3.8)) is monotonically decreasing for all simulated cases. In addition, the utmost accuracy of 10^{-25} is ultimately achievable for all these systems. The time per 1000 iteration is between 6.18 and 100 seconds in a MATLAB implementation, which can be reduced significantly in C++ and by parallel computing. We have verified that these numbers diminish by at least a factor of 3 if certain small-sized bags are combined to obtain a modest number of bags. This shows a trade-off between the chosen granularity for the algorithm and its computation time for a serial implementation (as opposed to a parallel implementation). To elaborate on the algorithm, note that every iteration amounts to a basic matrix operation or an eigendecomposition over matrices of size at most 7×7 for the IEEE 300-bus system. Efficient preconditioning methods could dramatically reduce the number of iterations (as OPF is often very ill-conditioned due to high inductance-to-resistance ratios), and this is left for future work.

3.7 Summary

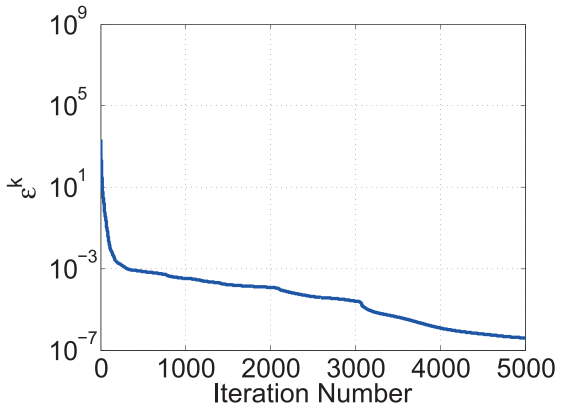
The main objective of this chapter is to design a fast and parallelizable algorithm for solving sparse SDPs corresponding to the convex relaxation of power optimization problems. To this end, the underling sparsity structure of a given SDP problem is captured using a tree decomposition technique, leading to a decomposed SDP problem. A highly distributed/parallelizable numerical algorithm is developed for solving the decomposed SDP, based on the alternating direction method of multipliers (ADMM). Each iteration of the designed algorithm has a closed-form solution, which involves multiplications and eigenvalue decompositions over certain submatrices induced by the tree decomposition of the sparsity graph. The proposed algorithm is applied to the classical optimal power flow problem, and also evaluated on IEEE benchmark systems. This algorithm is well suited for power optimization problems since it exploits the fact that real-world power networks have a low treewidth.



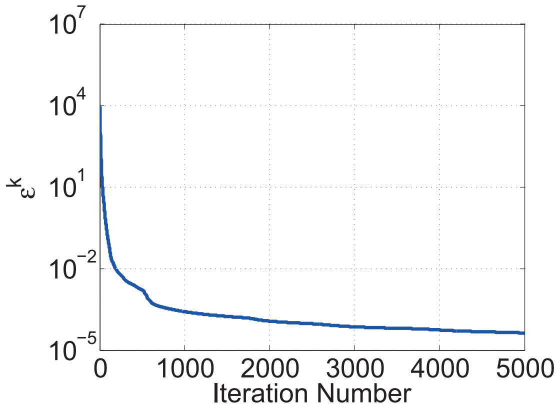
(a)



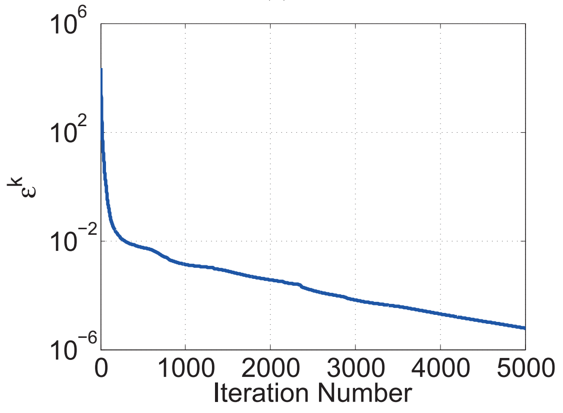
(b)



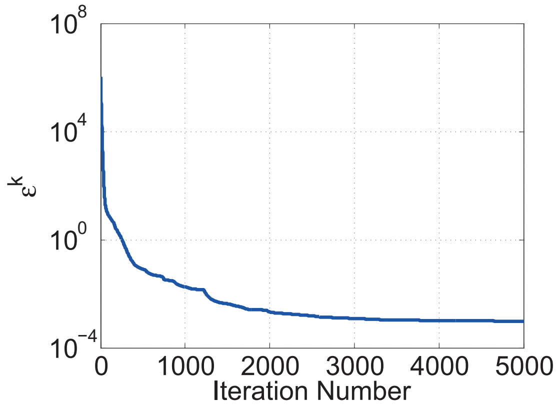
(c)



(d)



(e)



(f)

Figure 3.1: These plots show the convergence behavior of the energy function ε^k for IEEE test cases. (a): Chow's 9 bus, (b): IEEE 14 bus, (c): IEEE 30 bus, (d): IEEE 57 bus, (e): IEEE 118 bus, (f): IEEE 300 bus.

Chapter 4

Convex Relaxation for Optimal Distributed Control Problem

This chapter is concerned with the optimal distributed control (ODC) problem. We first study the infinite-horizon ODC problem (for deterministic systems) and then generalize the results to a stochastic ODC problem (for stochastic systems). By adopting a Lyapunov approach, we show that each of these non-convex controller design problems admits a rank-constrained formulation, which can be relaxed to a semidefinite program (SDP). The notion of treewidth is then utilized to prove that the SDP relaxation has a matrix solution with rank at most 3. If the SDP relaxation has a rank-1 solution, a globally optimal solution can be recovered from it; otherwise, a near-optimal controller together with a bound on its optimality degree may be attained. Since the proposed SDP relaxation is not computationally attractive, a computationally-cheap SDP relaxation is also developed. It is shown that this relaxation works as well as Riccati equations in the extreme case of designing a centralized controller. The superiority of the proposed technique is demonstrated on several thousand simulations for mass spring and random systems.

4.1 Introduction

Real-world systems mostly consist of many interconnected subsystems, and designing an optimal controller for them pose several challenges to the field of control. The area of *distributed control* is created to address the challenges arising in the control of these systems. The objective is to design a constrained controller whose structure is specified by a set of permissible interactions between the local controllers with the aim of reducing the computation or communication complexity of the overall controller. If the local controllers are not allowed to exchange information, the problem is often called *decentralized controller* design. It has been long known that the design of an optimal distributed (decentralized) controller is a daunting task because it amounts to an NP-hard optimization problem in general [19; 20]. Great effort has been devoted to investigating this highly complex problem for special types of systems, including spatially distributed systems [52; 53; 54; 55; 56], dynamically decoupled systems [57; 58], weakly coupled systems [59], and strongly connected systems [60].

There is no surprise that the decentralized control problem is computationally hard to solve. This is a consequence of the fact that several classes of optimization problems, including polynomial optimization and quadratically-constrained quadratic program (QCQP) as a special case, are NP-hard in the worst case. Due to the complexity of such problems, various convex relaxation methods based on linear matrix inequality (LMI), semidefinite programming (SDP), and second-order cone programming (SOCP) have gained popularity [21; 22]. These techniques enlarge the possibly non-convex feasible set into a convex set characterizable via convex functions, and then provide the exact or a lower bound on the optimal objective value. The SDP relaxation usually converts an optimization with a vector variable to a convex optimization with a matrix variable, via a lifting technique. The exactness of the relaxation can then be interpreted as the existence of a low-rank (e.g., rank-1) solution for the SDP relaxation. Several papers have studied the existence of a low-rank solution to matrix optimizations with linear or nonlinear (e.g., LMI) constraints. For instance, the papers [61; 62; 63] provide an upper bound on the lowest rank among all solutions of a feasible LMI problem. A rank-1 matrix decomposition technique is developed in [64] to find a rank-1 solution whenever the number of constraints is small. It was shown in [11] and [65] that the SDP relaxation is able to solve a large class of non-convex energy-related optimization problems performed over power networks. The success of the relaxation was related to the hidden structure of those optimizations induced by the physics of a power grid. Inspired by this positive result, the notion of "nonlinear optimization over graph" was developed in [66] and [67]. This technique maps the structure of an abstract nonlinear optimization into a graph from which the exactness of the SDP relaxation may be concluded. By adopting the graph technique developed in [66] and [67], the objective of this chapter is to study the potential of the SDP relaxation for the optimal distributed control problem.

In this chapter, two problems of infinite-horizon ODC (for deterministic systems) and stochastic ODC (for stochastic systems) are studied. Our approach rests on formulating each of these problems as a rank-constrained optimization from which an SDP relaxation can be derived. With no loss of generality, this chapter focuses on the design of a static controller. As the first contribution of this chapter, we show that infinite-horizon ODC and stochastic ODC both admit sparse SDP relaxations with solutions of rank at most 3. Since a rank-1 SDP matrix can be mapped back into a globally-optimal controller, the rank-3 solution may be deployed to retrieve a near-global controller.

Since the proposed relaxations are computationally expensive, we propose two computationally cheap SDP relaxations associated with infinite-horizon ODC and stochastic ODC. Afterwards, we develop effective heuristic methods to recover a near-optimal controller from the low-rank SDP solution. Note that the computationally-cheap SDP relaxations associated with infinite-horizon ODC and stochastic ODC are both exact for the classical (centralized) LQR and H_2 problems. This implies that the relaxations indirectly solve Riccati equations in the extreme case where the controller under design is unstructured. In this chapter, we conduct thousands of simulations on a mass-spring system and 100 random systems to elucidate the efficacy of the proposed relaxations. In particular, the design of several near-optimal structured controllers with global optimality degrees above 99% will be demonstrated.

This chapter is organized as follows. The ODC problem is formulated in Section 4.2. The SDP relaxation of an arbitrary QCQP is thoroughly studied via a graph- theoretic approach in Section 4.3. The infinite-horizon ODC problem is studied in Section 4.4. The results are generalized to a stochastic ODC problem in Section 4.5. Various experiments on mass spring systems and random simulations are provided in Section 4.6. A summary is given in Section 4.7.

Notations: $\mathbb{R}$, $\mathbb{S}^n$ and $\mathbb{S}^n_+$ denote the sets of real numbers, $n \times n$ symmetric matrices and $n \times n$ positive semidefinite matrices, respectively. $\text{rank}\{W\}$ and $\text{trace}\{W\}$ denote the rank and trace of a matrix W. The notation $W \succeq 0$ means that W is symmetric and positive semidefinite. Given a matrix W, its (l, m) entry is denoted as W_{lm}. Given a block matrix $\mathbf{W}$, its (l, m) block is shown as $\mathbf{W}_{lm}$. The superscript $(\cdot)^{\text{opt}}$ is used to show the globally optimal value of an optimization parameter. The symbols $(\cdot)^T$ and $\|\cdot\|$ denote the transpose and 2-norm operators, respectively. The notation $|x|$ shows the size of a vector x. The expected value of a random variable x is shown as $\mathcal{E}\{x\}$.

4.2 Problem Formulation

Consider the discrete-time system

$$\begin{cases} x[\tau+1] = Ax[\tau] + Bu[\tau] \\ \quad y[\tau] = Cx[\tau] \end{cases} \qquad \tau = 0, 1, 2, ... \tag{4.1}$$

with the known matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{r\times n}$, and $x[0] \in \mathbb{R}^n$. With no loss of generality, assume that C has full row rank. The goal is to design a distributed controller minimizing a quadratic cost function. We focus on the static case where the objective is to design a static controller of the form $u[\tau] = Ky[\tau]$ under the constraint that the controller gain K must belong to a given linear subspace $\mathcal{K} \subseteq \mathbb{R}^{m\times r}$. The set $\mathcal{K}$ captures the sparsity structure of the unknown constrained controller $u[\tau] = Ky[\tau]$ and, more specifically, it contains all $m \times r$ real-valued matrices with forced zeros in certain entries. This problem will be formalized below.

Optimal Distributed Control (ODC) problem: Design a stabilizing static controller $u[\tau] = Ky[\tau]$ to minimize the cost function

$$\sum_{\tau=0}^{p} \left(x[\tau]^T Q x[\tau] + u[\tau]^T R u[\tau]\right) + \alpha \operatorname{trace}\{KK^T\} \tag{4.2}$$

subject to the system dynamics (4.1) and the controller requirement $K \in \mathcal{K}$, for a terminal time p, a nonnegative scalar α, and positive-definite matrices Q and R.

Remark 1. *The third term in the objective function of the ODC problem is a soft penalty term aimed at avoiding a high-gain controller. Instead of this soft penalty, we could impose a hard constraint* $\operatorname{trace}\{KK^T\} \leq \beta$*, for a given number* β*. The method to be developed later can readily be adopted for the modified case.*

In this chapter of the thesis, we first deal with the **infinite-horizon ODC** problem in Section 4.4, corresponding to the case $p = +\infty$, and then generalize the results to a **stochastic ODC** problem in Section 4.5 This problem will be studied based on the following steps:

- First, the infinite-horizon ODC problem is cast as an optimization with linear matrix inequality constraints as well as quadratic constraints.
- Second, the resulting non-convex problem is formulated as a rank-constrained optimization.
- Third, an SDP relaxation of the problem is derived by dropping the non-convex rank constraint.
- Last, the rank of the minimum-rank solution of the SDP relaxation is analyzed.

In the next section, a sparse QCQP formulation of the ODC problem with a guaranteed low-rank SDP solution will be designed. To achieve this goal, a graph is associated to each QCQP formulation, which is then sparsified to contrive a sparse QCQP problem with a low-rank SDP solution. Please note that neither the infinite-horizon ODC nor the stochastic ODC problems could directly be formulated as a QCQP. The main objective of the next section is to understand Theorem 4, which will later be used in the Lyapunov approach for infinite-horizon ODC and the stochastic ODC problems.

4.3 SDP Relaxation for Quadratic Optimization

The objective of this section is to study the SDP relaxation of a QCQP problem using a graph-theoretic approach. Before proceeding with this part, some notions in graph theory will be reviewed.

4.3.1 Graph Theory Preliminaries

Notation 2. *The notation $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denotes as a graph $\mathcal{G}$ with the vertex set $\mathcal{V}$ and the edge set $\mathcal{E}$.*

Definition 8. *For two simple graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, the notation $\mathcal{G}_1 \subseteq \mathcal{G}_2$ means that $\mathcal{V}_1 \subseteq \mathcal{V}_2$ and $\mathcal{E}_1 \subseteq \mathcal{E}_2$. $\mathcal{G}_1$ is called a subgraph of $\mathcal{G}_2$ and $\mathcal{G}_2$ is called a supergraph of $\mathcal{G}_1$. A subgraph $\mathcal{G}_1$ of $\mathcal{G}_2$ is said to be an induced subgraph if for every pair of vertices $v_l, v_m \in \mathcal{V}_1$, the relation $(v_l, v_m) \in \mathcal{E}_1$ holds if and only if $(v_l, v_m) \in \mathcal{E}_2$. In this case, $\mathcal{G}_1$ is said to be induced by the vertex subset $\mathcal{V}_1$.*

Definition 9. *For two simple graphs $\mathcal{G}_1 = (\mathcal{V}, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}, \mathcal{E}_2)$ with the same set of vertices, their union is defined as $\mathcal{G}_1 \cup \mathcal{G}_2 = (\mathcal{V}, \mathcal{E}_1 \cup \mathcal{E}_2)$.*

Definition 10. *The representative graph of an $n \times n$ symmetric matrix W, denoted by $\mathcal{G}(W)$, is a simple graph with n vertices whose edges are specified by the locations of the nonzero off-diagonal entries of W. In other words, two arbitrary vertices i and j are connected if W_{ij} is nonzero.*

Consider a graph $\mathcal{G}$ identified by a set of "vertices" and a set of edges. This graph may have cycles in which case it cannot be a tree. Using the notion to be explained below, we can map $\mathcal{G}$ into a tree $\mathcal{T}$ identified by a set of "nodes" and a set of edges where each node of $\mathcal{T}$ contains a group of vertices of $\mathcal{G}$.

Definition 11 (Treewidth)**.** *Given a graph $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}})$, a tree $\mathcal{T}$ is called a tree decomposition of $\mathcal{G}$ if it satisfies the following properties:*

1. *Every node of $\mathcal{T}$ corresponds to and is identified by a subset of $\mathcal{V}_{\mathcal{G}}$. Alternatively, each node of $\mathcal{T}$ is regarded as a group of vertices of $\mathcal{G}$.*
2. *Every vertex of $\mathcal{G}$ is a member of at least one node of $\mathcal{T}$.*
3. *For every edge (i, j) of $\mathcal{G}$, there should be a node in $\mathcal{T}$ containing vertices i and j simultaneously.*
4. *Given an arbitrary vertex k of $\mathcal{G}$, the subgraph induced by all nodes of $\mathcal{T}$ containing vertex k must be connected (more precisely, a tree).*

The width of a tree decomposition is the cardinality of its biggest node minus one (recall that each node of $\mathcal{T}$ is indeed a set containing a number of vertices of $\mathcal{G}$). The treewidth of $\mathcal{G}$ is the minimum width over all possible tree decompositions of $\mathcal{G}$ and is denoted by $\text{tw}(\mathcal{G})$.

Note that the treewidth of a tree is equal to 1. Figure 4.1 shows a graph $\mathcal{G}$ with 6 vertices named a, b, c, d, e, f, together with its minimal tree decomposition $\mathcal{T}$. Every node of $\mathcal{T}$ is a set containing three members of $\mathcal{V}_{\mathcal{G}}$. The width of this decomposition is therefore equal to 2.

Definition 12 (Enriched Supergraph)**.** *Given a graph $\mathcal{G}$ accompanied by a tree decomposition $\mathcal{T}$ of width t, $\overline{\mathcal{G}}$ is called an enriched supergraph of $\mathcal{G}$ derived by $\mathcal{T}$ if it is obtained according to the following procedure:*

1. *Add a sufficient number of (redundant) vertices to the nodes of $\mathcal{T}$, if necessary, in such a way that every node includes exactly $t + 1$ vertices. Also, add the same vertices to $\mathcal{G}$ (without*

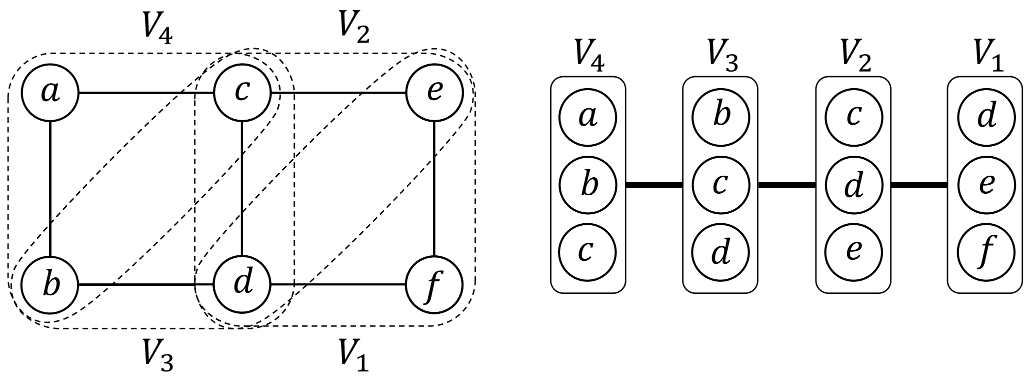


Figure 4.1: A minimal tree decomposition for a ladder

incorporating new edges). Denote the new graphs associated with $\mathcal{T}$ and $\mathcal{G}$ as $\tilde{\mathcal{T}}$ and $\tilde{\mathcal{G}}$, respectively.

2. *Index the nodes of the tree $\tilde{\mathcal{T}}$ as $V_1, V_2, \ldots, V_{|\mathcal{T}|}$ in such a way that for every $r \in \{1, ..., |\mathcal{T}|\}$, the node V_r becomes a leaf of $\mathcal{T}^r$ defined as the subgraph of $\tilde{\mathcal{T}}$ induced by $\{V_1, \ldots, V_r\}$. Denote the neighbor of V_r in $\mathcal{T}^r$ as $V_{r'}$ (note that $V_r \subseteq \mathcal{V}_{\mathcal{G}}$).*

3. *Define $\mathcal{G}^{|\mathcal{T}|} := \tilde{\mathcal{G}}$ and $\mathcal{O}^{|\mathcal{T}|}$ as the empty sequence. Define also $k = |\mathcal{T}|$.*

4. *Let $V_k \setminus V_{k'} = \{o_1, \ldots, o_s\}$ and $V_{k'} \setminus V_k = \{w_1, \ldots, w_s\}$. Define*

$$\mathcal{G}^{k-1} := \big(\mathcal{V}_{\mathcal{G}^k}, \mathcal{E}_{\mathcal{G}^k} \cup \{(o_1, w_1), \ldots, (o_s, w_s)\}\big) \tag{4.3}$$

$$\mathcal{O}^{k-1} := \mathcal{O}^k \cup (o_1, \ldots, o_s) \tag{4.4}$$

$$k := k - 1 \tag{4.5}$$

5. *If $k = 1$, set $\overline{\mathcal{G}} := \mathcal{G}^1$, $\mathcal{O} := \mathcal{O}^1$ and terminate; otherwise go to step 4. $\overline{\mathcal{G}}$ is referred to as an enriched supergraph of $\mathcal{G}$ derived by $\mathcal{T}$.*

Step 4 of the above definition is illustrated in Figure 4.2. Figure 4.3 delineates the process of obtaining an enriched supergraph $\overline{\mathcal{G}}$ of the graph $\mathcal{G}$ depicted in Figure 4.1. Bold lines show the edges added at each step of the algorithm.

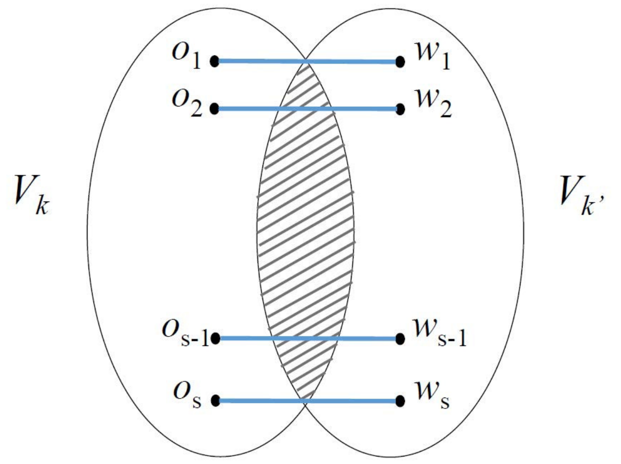


Figure 4.2: This figure illustrates Step 4 of Definition 12 for designing an enriched supergraph. The shaded area includes the common vertices of the nodes V_k and $V_{k'}$.

4.3.2 SDP Relaxation

Consider the standard nonconvex QCQP problem

$$\min_{x\in\mathbb{R}^n} \quad f_0(x) \tag{4.6a}$$

$$\text{s.t.} \quad f_k(x) \leq 0 \quad \text{for} \quad k=1,\ldots,p \tag{4.6b}$$

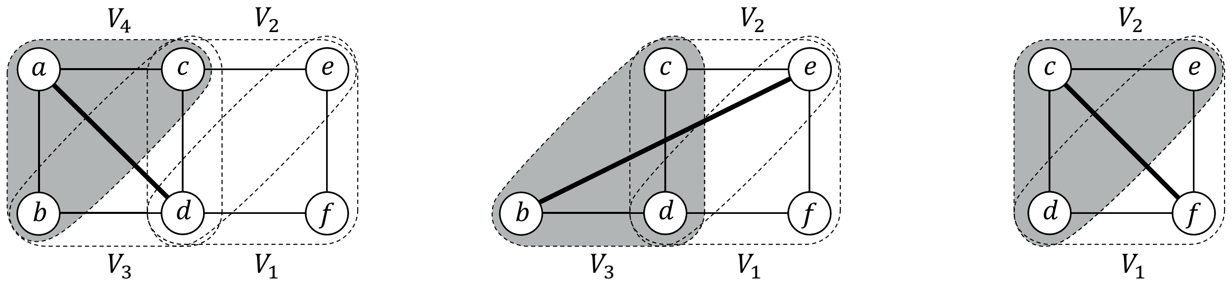
where $f_k(x) = x^T A_k x + 2b_k^T x + c_k$ for $k=0,\ldots,p$. Define

$$F_k \triangleq \begin{bmatrix} c_k & b_k^T \\ b_k & A_k \end{bmatrix} \quad \text{and} \quad w \triangleq [x_0 \quad x^T]^T, \tag{4.7}$$

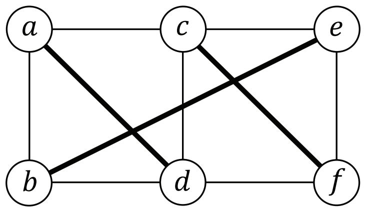
where x_0=1. Given $k \in \{0,1,...,p\}$, the function $f_k(x)$ is a homogeneous polynomial of degree 2 with respect to w. Hence, $f_k(x)$ has a linear representation as $f_k(x) = \text{trace}\{F_k W\}$, where

$$W \triangleq ww^T \tag{4.8}$$

Conversely, an arbitrary matrix $W \in \mathbb{S}^{n+1}$ can be factorized as (4.8) with $w_1 = 1$ if and only if it satisfies the three properties: $W_{11} = 1$, $W \succeq 0$, and $\text{rank}\{W\} = 1$. Therefore, the general QCQP



(a)



(b)

Figure 4.3: An enriched supergraph $\overline{\mathcal{G}}$ of the graph $\mathcal{G}$ given in Figure 4.1: (a) the steps of the algorithm (b) the resulting enriched supergraph.

(4.6) can be reformulated as below:

$$
\begin{aligned}
\min_{W\in\mathbb{S}^{n+1}} \quad & \operatorname{trace}\{F_0 W\} && (4.9a)\\
\text{s.t.} \quad & \operatorname{trace}\{F_k W\} \le 0 \qquad \text{for} \quad k=1,\dots,p && (4.9b)\\
& W_{11} = 1 && (4.9c)\\
& W \succeq 0 && (4.9d)\\
& \operatorname{rank}\{W\} = 1. && (4.9e)
\end{aligned}
$$

This optimization is called a **rank-constrained formulation** of the QCQP (4.6). In the above representation of QCQP, the constraint (4.9e) carries all the nonconvexity. Neglecting this constraint yields the convex problem

$$
\begin{aligned}
\min_{W\in\mathbb{S}^{n+1}} \quad & \operatorname{trace}\{F_0 W\} && (4.10a)\\
\text{s.t.} \quad & \operatorname{trace}\{F_k W\} \le 0 \qquad \text{for} \quad k=1,\dots,p && (4.10b)\\
& W_{11} = 1 && (4.10c)\\
& W \succeq 0, && (4.10d)
\end{aligned}
$$

which is called an **SDP relaxation** of the QCQP (4.6). The existence of a rank-1 solution for the SDP relaxation guarantees the equivalence between the original QCQP and its relaxed problem.

4.3.3 Connection Between Rank and Sparsity

To explore the rank of the minimum-rank solution of the SDP relaxation, define $\mathcal{G} = \mathcal{G}(F_0) \cup \cdots \cup \mathcal{G}(F_p)$ as the **sparsity graph** associated with the rank-constrained problem (4.9). The graph $\mathcal{G}$ describes the zero-nonzero pattern of the matrices $F_0, \dots, F_p$, or alternatively captures the sparsity level of the QCQP problem (4.6). The graph $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}})$ has the following properties:

1. Each vertex of $\mathcal{V}_{\mathcal{G}}$ corresponds to one of the entries of w or equivalently one of the elements of the set $\{x_0, x_1, ..., x_n\}$ (note that $x_0 = 1$). Let the vertex associated with the variable x_i be denoted as v_{x_i} for $i = 0, 1, ..., n$.

2. Given two distinct indices $i, j \in \{0, 1, \dots, n\}$, the pair (v_{x_i}, v_{x_j}) is an edge of $\mathcal{G}$ if and only if the monomial $x_i x_j$ has a nonzero coefficient in at least one of the polynomials $f_0(x), f_1(x), \dots, f_p(x)$.

Let $\bar{\mathcal{G}} = (\mathcal{V}_{\bar{\mathcal{G}}}, \mathcal{E}_{\bar{\mathcal{G}}})$ be an enriched supergraph of $\mathcal{G}$, obtained from a tree decomposition of width t. Let m denote the number of vertices of $\bar{\mathcal{G}}$.

Theorem 4. *Consider an arbitrary solution $\widehat{W} \in \mathbb{S}_+^{n+1}$ of the SDP relaxation problem* (4.10) *and let $Z \in \mathbb{S}^m$ be a matrix with the property that $\mathcal{G}(Z) = \bar{\mathcal{G}}$. Let $\overline{W}^{\text{opt}}$ denote an arbitrary solution of the optimization*

$$\begin{aligned}
\min_{\overline{W} \in \mathbb{S}^m} \quad & \text{trace}\{Z\overline{W}\} && (4.11\text{a})\\
\text{s.t.} \quad & \overline{W}_{kk} = \widehat{W}_{kk} \quad \text{for} \quad k \in \mathcal{V}_{\mathcal{G}}, && (4.11\text{b})\\
& \overline{W}_{kk} = 1 \quad \text{for} \quad k \in \mathcal{V}_{\bar{\mathcal{G}}} \setminus \mathcal{V}_{\mathcal{G}}, && (4.11\text{c})\\
& \overline{W}_{ij} = \widehat{W}_{ij} \quad \text{for} \quad (i,j) \in \mathcal{E}_{\mathcal{G}}, && (4.11\text{d})\\
& \overline{W} \succeq 0. && (4.11\text{e})
\end{aligned}$$

Define W^{opt} as the $(n+1)$-th principal minor of $\overline{W}^{\text{opt}}$. Then, W^{opt} satisfies the following two properties:

a) W^{opt} is an optimal solution to the SDP relaxation (4.10).

b) $\text{rank}\{W^{\text{opt}}\} \leq t + 1$.

Proof. See [63] for the proof. □

Assume that a tree decomposition of $\mathcal{G}$ with a small width is known. Theorem 4 states that an arbitrary (high-rank) solution to the SDP relaxation problem can be transformed into a low-rank solution by solving the convex program (4.11).

4.4 Deterministic Control Systems

The primary objective of the ODC problem is to design a structurally constrained gain K. Assume that the matrix K has l free entries to be designed. Denote these parameters as $h_1, h_2, ..., h_l$. To formulate the ODC problem, the space of permissible controllers can be characterized as

$$\mathcal{K} \triangleq \left\{ \sum_{i=1}^{l} h_i M_i \;\middle|\; h \in \mathbb{R}^l \right\}, \tag{4.12}$$

for some (fixed) 0-1 matrices $M_1, ..., M_l \in \mathbb{R}^{m \times r}$. Now, the ODC problem can be stated as follows.

Optimal Distributed Control (ODC) problem: Minimize

$$\sum_{\tau=0}^{p} \left(x[\tau]^T Q x[\tau] + u[\tau]^T R u[\tau] \right) + \alpha \, \text{trace}\{KK^T\} \tag{4.13a}$$

subject to

$$x[\tau+1] = Ax[\tau] + Bu[\tau] \qquad \text{for} \quad \tau = 0, 1, \ldots, p \tag{4.13b}$$

$$y[\tau] = Cx[\tau] \qquad \text{for} \quad \tau = 0, 1, \ldots, p \tag{4.13c}$$

$$u[\tau] = Ky[\tau] \qquad \text{for} \quad \tau = 0, 1, \ldots, p \tag{4.13d}$$

$$K = h_1 M_1 + \ldots + h_l M_l \tag{4.13e}$$

$$x[0] = \text{given} \tag{4.13f}$$

over the variables

$$x[0], x[1], \ldots, x[p] \in \mathbb{R}^n \tag{4.13g}$$

$$y[0], y[1], \ldots, y[p] \in \mathbb{R}^r \tag{4.13h}$$

$$u[0], u[1], \ldots, u[p] \in \mathbb{R}^m \tag{4.13i}$$

$$h \in \mathbb{R}^l. \tag{4.13j}$$

In this section, we deal with the **infinite-horizon ODC** problem, corresponding to the case $p = +\infty$.

4.4.1 Lyapunov Formulation

To deal with the infinite dimension of the infinite-horizon ODC and its hard stability constraint, a Lyapunov approach will be taken below.

Theorem 5. *The infinite-horizon ODC problem is equivalent to finding a controller $K \in \mathcal{K}$, a symmetric Lyapunov matrix $P \in \mathbb{S}^n$, an auxiliary symmetric matrix $G \in \mathbb{S}^n$ and an auxiliary* *matrix $L \in \mathbb{R}^{m\times n}$ to satisfy the following optimization problem:*

$$\min_{K,L,P,G} \quad x[0]^T P x[0] + \alpha\,\mathrm{trace}\{KK^T\} \tag{4.14a}$$

subject to:

$$\begin{bmatrix} G & G & (AG+BL)^T & L^T \\ G & Q^{-1} & 0 & 0 \\ AG+BL & 0 & G & 0 \\ L & 0 & 0 & R^{-1} \end{bmatrix} \succeq 0, \tag{4.14b}$$

$$\begin{bmatrix} P & I \\ I & G \end{bmatrix} \succeq 0, \tag{4.14c}$$

$$L = KCG \tag{4.14d}$$

$$K \in \mathcal{K} \tag{4.14e}$$

Proof. Given an arbitrary control gain K, consider the system (4.1) under the controller $u[\tau] = Ky[\tau]$. It is evident that

$$x[\tau] = (A+BKC)^\tau x[0], \qquad \tau = 0,1,...,\infty \tag{4.15}$$

Hence, the cost function (4.2) can be written as:

$$\sum_{\tau=0}^{\infty} \left(x[\tau]^T Q x[\tau] + u[\tau]^T R u[\tau]\right) + \alpha\,\mathrm{trace}\{KK^T\} = x[0]^T P x[0] + \alpha\,\mathrm{trace}\{KK^T\} \tag{4.16}$$

where

$$P = \sum_{\tau=0}^{\infty} ((A+BKC)^\tau)^T (Q + C^T K^T R K C)(A+BKC)^\tau \tag{4.17}$$

or equivalently

$$(A+BKC)^T P (A+BKC) - P + Q + (KC)^T R (KC) = 0 \tag{4.18a}$$

$$P \succeq 0 \tag{4.18b}$$

On the other hand, it is well-known that replacing the equality sign "=" in (4.18a) with the inequality sign "$\preceq$" does not affect the solution of the optimization problem [22]. After pre- and post-multiplying the Lyapunov inequality obtained from (4.18a) with P^{-1} and using the Schur complement formula, the constraints (4.18a) and (4.18b) can be combined as

$$\begin{bmatrix} P^{-1} & P^{-1} & S^T & P^{-1}(KC)^T \\ P^{-1} & Q^{-1} & 0 & 0 \\ S & 0 & P^{-1} & 0 \\ (KC)P^{-1} & 0 & 0 & R^{-1} \end{bmatrix} \succeq 0 \tag{4.19}$$

where $S = (A + BKC)P^{-1}$ and 0's in the above matrix are zero matrices of appropriate dimensions. By replacing P^{-1} with a new variable G in the above matrix and defining L as KCG, the constraints (4.14b) and (4.14d) will be obtained. The minimization of $x[0]^T Px[0]$ subject to the constraint (4.14c) ensures that $P = G^{-1}$ is satisfied for at least one optimal solution of the optimization problem. □

Theorem 6. *Consider the special case where $C = I$, $\alpha = 0$ and $\mathcal{K}$ contains the set of all unstructured controllers. Then, the infinite-horizon ODC problem has the same solution as the convex optimization problem obtained from the nonlinear optimization* (4.14) *by removing its non-convex constraint* (4.14d).

Proof. It is easy to verify that a solution $(K^{\text{opt}}, P^{\text{opt}}, G^{\text{opt}}, L^{\text{opt}})$ of the convex problem stated in the theorem can be mapped to the solution $(L^{\text{opt}}(G^{\text{opt}})^{-1}, P^{\text{opt}}, G^{\text{opt}}, L^{\text{opt}})$ of the non-convex problem (4.14) and vice versa (recall that $C = I$ by assumption). This completes the proof. □

4.4.2 SDP Relaxation

Theorem 6 states that a classical optimal control problem can be precisely solved via a convex relaxation of the nonlinear optimization (4.14) by eliminating its constraint (4.14d). However, this simple convex relaxation does not work satisfactorily for a general control structure $\mathcal{K}$. To design a better relaxation, define

$$w := \begin{bmatrix} 1 & h^T & \text{vec}\{CG\}^T \end{bmatrix}^T \tag{4.20}$$

where h is a column vector containing the variables (free parameters) of K, and vec$\{CG\}$ is a column vector containing all scalar entries of CG. It is possible to write every entry of the bilinear matrix term KCG as a linear function of the entries of the parametric matrix ww^T. Hence, by introducing a new matrix variable W playing the role of ww^T, the nonlinear constraint (4.14d) can be rewritten as a linear constraint in term of W. In addition, the term α trace$\{KK^T\}$ in the objective function of the ODC problem is also linear in W. Now, one can relax the non-convex mapping constraint $W = ww^T$ to $W \succeq 0$ and another constraint stating that the first column of W is equal to w. This convex problem is referred to as **SDP relaxation of ODC** in this work. In the case where the relaxation has the same solution as ODC, the relaxation is said to be exact.

Theorem 7. *Consider the case where $\mathcal{K}$ contains only diagonal matrices. The following statements hold regarding the SDP relaxation of the infinite-horizon ODC problem:*

i) The relaxation is exact if it has a solution $(K^{opt}, P^{opt}, G^{opt}, L^{opt}, W^{opt})$ such that rank$\{W^{opt}\} = 1$.

ii) The relaxation always has a solution $(K^{opt}, P^{opt}, G^{opt}, L^{opt}, W^{opt})$ such that rank$\{W^{opt}\} \leq 3$.

Proof. To study the SDP relaxation of the aforementioned control problem, we need to define a sparsity graph $\mathcal{G}$. Let η denote the number of rows of W. The graph $\mathcal{G}$ has η vertices with the property that two arbitrary disparate vertices $i, j \in \{1, 2, ..., \eta\}$ are connected in the graph if W_{ij} appears in at least one of the constraints of the SDP relaxation excluding the global constraint $W \succeq 0$. For example, vertex 1 is connected to all remaining vertices of the graph. The graph $\mathcal{G}$ with its vertex 1 removed is depicted in Figure 4.4. This graph is acyclic and therefore the treewidth of the graph $\mathcal{G}$ is at most 2. Hence, It follows from Theorem 4 that the SDP relaxation has a matrix solution with rank at most 2+1. $\square$

Theorem 7 states that the SDP relaxation of the infinite-horizon ODC problem has a low-rank solution. However, it does not imply that every solution of the relaxation is low-rank. Theorem 4 provides a procedure for converting a high-rank solution of the SDP relaxation into a matrix solution with rank at most 3. The above theorem will be generalized below.

Proposition 1. *The infinite-horizon ODC problem has a convex relaxation with the property that its exactness amounts to the existence of a rank-1 matrix solution W^{opt}. Moreover, it is always guaranteed that this relaxation has a solution such that* rank$\{W^{opt}\} \leq 3$.

Proof. The procedure of designing an SDP relaxation with a guaranteed low-rank solution will be only sketched here. There are two binary matrices Φ_1 and Φ_2 such that $K = \Phi_1 \text{diag}\{k\} \Phi_2$ for

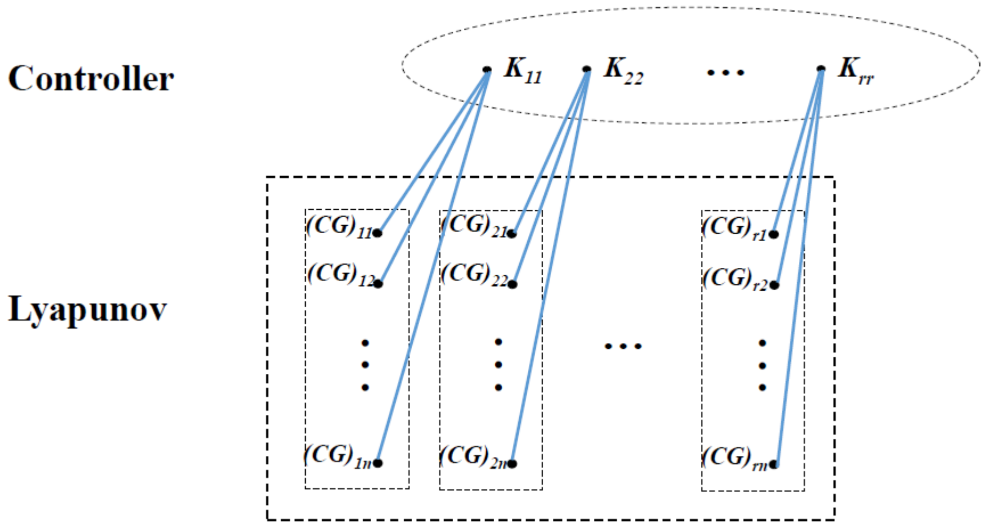


Figure 4.4: The sparsity graph for the infinite-horizon ODC problem in the case where $\mathcal{K}$ consists of diagonal matrices (the central vertex 1 is removed for simplicity).

every $K \in \mathcal{K}$, where diag$\{k\}$ denotes a diagonal matrix whose diagonal contains the free (variable) entries of K. Hence, the design of a structured control gain K for the system (A, B, C) amounts to the design of a diagonal control gain diag$\{k\}$ for the system $(A, B\Phi_1, \Phi_2 C)$ (after updating the matrices Q and R accordingly). It follows from Theorem 7 that the SDP relaxation of the ODC problem equivalently formulated for the new system satisfies the properties of this theorem. $\square$

In this section, it has been shown that the infinite-horizon ODC problem has an SDP relaxation with a low-rank solution. Nevertheless, there are many SDP relaxations with this property and it is desirable to find the one offering the highest lower bound on the optimal solution of the ODC problem. To this end, the abovementioned SDP relaxation should be reformulated in such a way that the diagonal entries of the matrix W are incorporated into as many constraints of the problem as possible in order to indirectly penalize the rank of the matrix W. This idea will be flourished next, but for a computationally-cheap relaxation of the ODC problem.

4.4.3 Computationally-Cheap SDP Relaxation

The aforementioned SDP relaxation has a high dimension for a large-scale system, which makes it less interesting for computational purposes. Moreover, the quality of its optimal objective value can be improved using some indirect penalty technique. The objective of this subsection is to offer a computationally-cheap SDP relaxation for the ODC problem, whose solution outperforms that of the previous SDP relaxation. For this purpose, Consider an invertible matrix $\Phi \in \mathbb{R}^{n\times n}$ such that

$$C\Phi = \begin{bmatrix} I & 0 \end{bmatrix} \tag{4.21}$$

where I the is identity matrix and "0" is an $r \times (n-r)$ zero matrix. Define also

$$\mathcal{K}^2 = \{KK^T \mid K \in \mathcal{K}\} \tag{4.22}$$

Indeed, $\mathcal{K}^2$ captures the sparsity pattern of the matrix KK^T. For example, if $\mathcal{K}$ consists of block-diagonal (rectangular) matrix, $\mathcal{K}^2$ will also include block-diagonal (square) matrices. Let $\mu \in \mathbb{R}$ be a positive number such that

$$Q \succ \mu \times \Phi^{-T}\Phi^{-1} \tag{4.23}$$

where Φ^{-T} denotes the transpose of the inverse of Φ. Define $\widehat{Q} := Q - \mu \times \Phi^{-T}\Phi^{-1}$.

Computationally-Cheap SDP Relaxation of ODC: This optimization problem is defined as the minimization of

$$\text{trace}\{x[0]^T P x[0] + \alpha \mathbf{W}_{33}\} \tag{4.24}$$

subject to the constraints

$$\begin{bmatrix} G - \mu\mathbf{W}_{22} & G & (AG+BL)^T & L^T \\ G & \widehat{Q}^{-1} & 0 & 0 \\ AG+BL & 0 & G & 0 \\ L & 0 & 0 & R^{-1} \end{bmatrix} \succeq 0, \tag{4.25a}$$

$$\begin{bmatrix} P & I \\ I & G \end{bmatrix} \succeq 0, \tag{4.25b}$$

$$\mathbf{W} := \left[\begin{array}{c:c:c} I_n & \Phi^{-1}G & \begin{bmatrix} K^T \\ 0 \end{bmatrix} \\ \hdashline G\Phi^{-T} & \mathbf{W}_{22} & L^T \\ \hdashline \begin{bmatrix} K & 0 \end{bmatrix} & L & \mathbf{W}_{33} \end{array}\right] \succeq 0, \tag{4.25c}$$

$$K \in \mathcal{K}, \tag{4.25d}$$

$$\mathbf{W}_{33} \in \mathcal{K}^2, \tag{4.25e}$$

with the parameter set $\{K, L, G, P, \mathbf{W}\}$, where the dependent variables $\mathbf{W}_{22}$ and $\mathbf{W}_{33}$ represent two blocks of $\mathbf{W}$.

The following remarks can be made regarding the computationally-cheap SDP relaxation:

- The constraint (4.25a) corresponds to the Lyapunov inequality associated with (4.18a), where $\mathbf{W}_{22}$ in its first block aims to play the role of $P^{-1}\Phi^{-T}\Phi^{-1}P^{-1}$.
- The constraint (4.25b) ensures that the relation $P = G^{-1}$ occurs at optimality (at least for one of the solution of the problem).
- The constraint (4.25c) is a surrogate for the only complicating constraint of the ODC problem, i.e., $L = KCG$.
- Since no non-convex rank constraint is imposed on the problem to maintain the convexity of the relaxation, the rank constraint is compensated in various ways. More precisely, the entries of $\mathbf{W}$ are constrained in the objective function (4.24) through the term trace$\{\alpha\mathbf{W}_{33}\}$, in the first block of the constraint (4.25a) through the term $G - \mu\mathbf{W}_{22}$, and also via the constraints (4.25d) and (4.25e). These terms aim to automatically penalize the rank of $\mathbf{W}$ indirectly.
- The proposed relaxation takes advantage of the sparsity of not only K, but also KK^T (through the constraint (4.25e)).

Theorem 8. *The computationally-cheap SDP relaxation is a convex relaxation of the infinite-horizon ODC problem. Furthermore, the relaxation is exact if and only if it possesses a solution* $(K^{opt}, L^{opt}, P^{opt}, G^{opt}, \mathbf{W}^{opt})$ *such that rank*$\{\mathbf{W}^{opt}\} = n$.

Proof. The objective function and constraints of the computationally-cheap SDP relaxation are all linear functions of the tuple $(K, L, P, G, \mathbf{W})$. Hence, this relaxation is indeed convex. To study the relationship between this optimization problem and the infinite-horizon ODC, consider a feasible point (K, L, P, G) of the ODC formulation (4.14). It can be deduced from the relation $L = KCG$ that $(K, L, P, G, \mathbf{W})$ is a feasible solution of the computationally-cheap SDP relaxation if the free blocks of $\mathbf{W}$ are considered as

$$\mathbf{W}_{22} = G\Phi^{-T}\Phi^{-1}G, \qquad \mathbf{W}_{33} = KK^T \tag{4.26}$$

(note that (4.14b) and (4.25a) are equivalent for this choice of $\mathbf{W}$). This implies that computationally-cheap SDP problem is a convex relaxation of the infinite-horizon ODC problem.

Consider now a solution $(K^{\text{opt}}, L^{\text{opt}}, P^{\text{opt}}, G^{\text{opt}}, W^{\text{opt}})$ of the computationally-cheap SDP relaxation such that $\text{rank}\{\mathbf{W}^{\text{opt}}\} = n$. Since the rank of the first block of $\mathbf{W}^{\text{opt}}$ (i.e., I_n) is already n, a Schur complement argument on the blocks $(1,1)$, $(1,3)$, $(2,1)$ and $(2,3)$ of $\mathbf{W}^{\text{opt}}$ yields that

$$0 = L^{\text{opt}} - \left[\begin{array}{cc} K^{\text{opt}} & 0 \end{array}\right] (I_n)^{-1} \Phi^{-1} G^{\text{opt}} \tag{4.27}$$

or equivalently $L^{\text{opt}} = K^{\text{opt}} C G^{\text{opt}}$, which is tantamount to the constraint (4.14d). This implies that $(K^{\text{opt}}, L^{\text{opt}}, P^{\text{opt}}, G^{\text{opt}})$ is a solution of the ODC problem and hence the relaxation is exact. So far, we have shown that the existence of a rank-n solution $\mathbf{W}^{\text{opt}}$ guarantees the exactness of the relaxation. The converse of this statement can also be proved similarly. $\square$

The matrix variable W in the first SDP relaxation of the infinite-horizon ODC problem had $O(n^2)$ rows. In contrast, this number reduces to $O(n)$ for the matrix $\mathbf{W}$ in the computationally-cheap SDP relaxation, which significantly reduces the computation time of the relaxation.

Corollary 1. *Consider the special case where $C = I$, $\alpha = 0$ and $\mathcal{K}$ contains the set of all unstructured controllers. Then, the computationally-cheap SDP relaxation is exact for the infinite-horizon ODC problem.*

Proof. The proof follows from that of Theorem 6. $\square$

4.4.4 Controller Recovery

Once the computationally-cheap SDP relaxation is solved, a controller K must be recovered. This can be achieved in two ways as explained below.

Direct Recovery Method for ODC: A near-optimal controller $\hat{K}$ for the infinite-horizon ODC problem is chosen to be equal to the optimal matrix K^{opt} obtained from the computationally-cheap SDP relaxation.

Indirect Recovery Method for ODC: Let $(K^{\text{opt}}, L^{\text{opt}}, P^{\text{opt}}, G^{\text{opt}}, \mathbf{W}^{\text{opt}})$ denote a solution of the computationally-cheap SDP relaxation. A near-optimal controller $\hat{K}$ for the infinite-horizon ODC problem is recovered by solving a convex program with the variables $K \in \mathcal{K}$ and $\gamma \in \mathbb{R}$ to minimize the cost function

$$\varepsilon \times \gamma + \alpha \operatorname{trace}\{KK^T\} \tag{4.28}$$

subject to the constraint

$$\begin{bmatrix} (G^{\text{opt}})^{-1} - Q + \gamma I_n & (A+BKC)^T & (KC)^T \\ (A+BKC) & G^{\text{opt}} & 0 \\ (KC) & 0 & R^{-1} \end{bmatrix} \succ 0 \tag{4.29}$$

where ε is a pre-specified nonnegative number.

The direct recovery method assumes that the controller K^{opt} obtained from the computationally-cheap SDP relaxation is near-optimal, whereas the indirect method assumes that the controller K^{opt} might be unacceptably imprecise while the inverse of the Lyapunov matrix is near-optimal. The indirect method is built on the SDP relaxation by fixing G at its optimal value and then perturbing Q as $Q - \gamma I_n$ to facilitate the recovery of a stabilizing controller. It may rarely happen that a stabilizing controller can be recovered from a solution G^{opt} if γ is set to zero. In other words, since the solution of the computationally-cheap SDP relaxation is not exact in general, there may not exist any controller $\hat{K}$ satisfying the Lyapunov equation jointly with G^{opt}. Nonetheless, perturbing the diagonal entries of Q with γ boosts the degree of the freedom of the problem and helps with the existence of a controller $\hat{K}$. Although none of the proposed recovery methods is universally better than the other one, we have verified in numerous simulations that the indirect recovery method significantly outperforms the direct recovery method with a high probability.

4.5 Stochastic Control Systems

The ODC problem was investigated for a deterministic system in the preceding section. The objective of this section is to generalize the results derived earlier to stochastic systems. To this end, consider the discrete-time system

$$\begin{cases} x[\tau+1] = Ax[\tau] + Bu[\tau] + Ed[\tau] \\ y[\tau] = Cx[\tau] + Fv[\tau] \end{cases} \qquad \tau = 0, 1, 2, ... \tag{4.30}$$

with the known matrices A, B, C, E, and F, where

- $x[\tau] \in \mathbb{R}^n$, $u[\tau] \in \mathbb{R}^m$ and $y[\tau] \in \mathbb{R}^r$ denote the state, input and output of the system.

- $d[\tau]$ and $v[\tau]$ denote the input disturbance and measurement noise, which are assumed to be zero-mean white-noise random processes.

The goal is to design an optimal distributed controller. In order to simplify the presentation, we focus on the static case where the objective is to design a static controller of the form $u[\tau] = Ky[\tau]$ under the structural constraint $K \in \mathcal{K}$. This section of this chapter is mainly concerned with the following problem.

Stochastic Optimal Distributed Control (SODC) problem: Design a stabilizing static controller $u[\tau] = Ky[\tau]$ to minimize the cost function

$$\lim_{\tau \to +\infty} \mathcal{E}\left(x[\tau]^T Q x[\tau] + u[\tau]^T R u[\tau]\right) + \alpha \operatorname{trace}\{KK^T\} \tag{4.31}$$

subject to the system dynamics (4.30) and the controller requirement $K \in \mathcal{K}$, for a nonnegative scalar α and positive-definite matrices Q and R.

Define two covariance matrices as below:

$$\Sigma_d = \mathcal{E}\{Ed[0]d[0]^T E^T\}, \quad \Sigma_v = \mathcal{E}\{Fv[0]v[0]^T F^T\} \tag{4.32}$$

In what follows, the SODC problem will be formulated as a nonlinear optimization program.

Theorem 9. *The SODC problem is equivalent to finding a controller $K \in \mathcal{K}$, a symmetric Lyapunov matrix $P \in \mathbb{S}^n$, and auxiliary matrices $G \in \mathbb{S}^n$, $L \in \mathbb{R}^{m \times n}$ and $M \in \mathbb{S}^r$ to minimize the objective function*

$$\operatorname{trace}\{P\Sigma_d + M\Sigma_v + K^T R K \Sigma_v\} + \alpha \operatorname{trace}\{KK^T\} \tag{4.33}$$

subject to the constraints

$$\begin{bmatrix} G & G & (AG+BL)^T & L^T \\ G & Q^{-1} & 0 & 0 \\ AG+BL & 0 & G & 0 \\ L & 0 & 0 & R^{-1} \end{bmatrix} \succeq 0, \tag{4.34a}$$

$$\begin{bmatrix} P & I \\ I & G \end{bmatrix} \succeq 0, \tag{4.34b}$$

$$\begin{bmatrix} M & (BK)^T \\ BK & G \end{bmatrix} \succeq 0, \tag{4.34c}$$

$$L = KCG \tag{4.34d}$$

$$K \in \mathcal{K} \tag{4.34e}$$

Proof. It is straightforward to verify that

$$\begin{aligned} x[\tau] = (A+BKC)^\tau x[0] &+ \sum_{t=0}^{\tau-1}(A+BKC)^t Ed[\tau-t-1] \\ &+ \sum_{t=0}^{\tau-1}(A+BKC)^t BKFv[\tau-t-1] \end{aligned} \tag{4.35}$$

for $\tau = 1, 2, ...$ On the other hand, since the controller under design must be stabilizing, $(A+BKC)^\tau$ approaches zero as τ goes to $+\infty$. In light of the above equation, it can be verified that

$$\begin{aligned} &\mathcal{E}\left\{\lim_{\tau\to+\infty}\left(x[\tau]^T Qx[\tau] + u[\tau]^T Ru[\tau]\right) + \alpha\,\text{trace}\{KK^T\}\right\} = \\ &= \mathcal{E}\left\{\lim_{\tau\to+\infty} x[\tau]^T\left(Q + C^T K^T RKC\right)x[\tau]\right\} \\ &+ \mathcal{E}\left\{\lim_{\tau\to+\infty} v[\tau]^T F^T K^T RKFv[\tau]\right\} + \alpha\,\text{trace}\{KK^T\} \\ &= \text{trace}\{P\Sigma_d + (BK)^T P(BK)\Sigma_v + K^T RK\Sigma_v + \alpha KK^T\} \end{aligned} \tag{4.36}$$

where

$$P = \sum_{t=0}^{\infty}\left((A+BKC)^t\right)^T (Q + C^T K^T RKC)(A+BKC)^t \tag{4.37}$$

Similar to the proof of Theorem 5, the above infinite series can be replaced by the following expanded

Lyapunov inequality:

$$\begin{bmatrix} P^{-1} & P^{-1} & S^T & P^{-1}(KC)^T \\ P^{-1} & Q^{-1} & 0 & 0 \\ S & 0 & P^{-1} & 0 \\ (KC)P^{-1} & 0 & 0 & R^{-1} \end{bmatrix} \succeq 0 \tag{4.38}$$

where $S = (A + BKC)P^{-1}$. After replacing P^{-1} and KCP^{-1} with new variables G and L, it can be concluded that:

- The condition (4.38) is identical to the set of constraints (4.34a) and (4.34d).
- The cost function (4.36) can be expressed as

$$\text{trace}\{P\Sigma_d + (BK)^T G^{-1}(BK)\Sigma_v + K^T RK\Sigma_v + \alpha KK^T\} \tag{4.39}$$

- Since P appears only once in the constraints of the optimization problem (4.33)-(4.34) (i.e., the condition (4.34b)) and the objective function of this optimization includes the term trace$\{P\Sigma_d\}$, the optimal value of P is equal to G^{-1}.
- Similarly, the optimal value of M is equal to $(BK)^T G^{-1}(BK)$.

The proof follows from the above observations. □

The SODC problem is cast as a (deterministic) nonlinear program in Theorem 9. This optimization problem is non-convex due only to the complicating constraint (4.34d) . More precisely, the removal of this nonlinear constraint makes the optimization problem a semidefinite program (note that the term $K^T RK$ in the objective function is convex due to the assumption $R \succ 0$).

The traditional H_2 optimal control problem (i.e., in the centralized case) can be solved using Riccati equations. It will be shown in the next proposition that the abovementioned semidefinite program correctly solves the centralized H_2 optimal control problem.

Proposition 2. *Consider the special case where $C = I$, $\alpha = 0$, $\Sigma_v = 0$, and $\mathcal{K}$ contains the set of all unstructured controllers. Then, the SODC problem has the same solution as the convex optimization problem obtained from the nonlinear optimization* (4.33)-(4.34) *by removing its non-convex constraint* (4.34d).

Proof. It is similar to the proof of Theorem 6. $\square$

Proposition 2 states that a classical optimal control problem can be precisely solved via a convex relaxation of the nonlinear optimization (4.33)-(4.34) by eliminating its constraint (4.34d). However, this simple convex relaxation does not work satisfactorily for a general control structure $\mathcal{K}$. To design a better relaxation, consider the vector w defined in (4.20). Similar to infinite-horizon ODC, the bilinear matrix term KCG can be represented as a linear function of the entries of the parametric matrix $\mathbf{W}$ defined as ww^T. Now, relaxing the constraint $\mathbf{W} = ww^T$ to $\mathbf{W} \succeq 0$ and adding another constraint stating that the first column of $\mathbf{W}$ is equal to w leads to an SDP relaxation. This convex problem is referred to as **SDP relaxation of SODC**. In the case where the relaxation has the same solution as SODC, the relaxation is said to be exact.

Proposition 3. *Consider the case where $\mathcal{K}$ contains only diagonal matrices. The following statements hold regarding the SDP relaxation of the SODC problem:*

i) The relaxation is exact if it has a solution $(K^{opt}, P^{opt}, G^{opt}, L^{opt}, M^{opt}, W^{opt})$ such that $\text{rank}\{W^{opt}\} = 1$.

ii) The relaxation always has a solution $(K^{opt}, P^{opt}, G^{opt}, L^{opt}, M^{opt}, W^{opt})$ such that $\text{rank}\{W^{opt}\} \leq 3$.

Proof. The proof is omitted (see Theorems 7 and 9). $\square$

As before, it can be deduced from Proposition 3 that the SODC problem has a convex relaxation with the property that its exactness amounts to the existence of a rank-1 matrix solution W^{opt}. Moreover, it is always guaranteed that this relaxation has a solution such that $\text{rank}\{W^{\text{opt}}\} \leq 3$.

A computationally-cheap SDP relaxation will be derived below. Let μ_1 and μ_2 be two nonnegative numbers such that

$$Q \succ \mu_1 \times \Phi^{-T}\Phi^{-1}, \quad \Sigma_v \succeq \mu_2 \times I \tag{4.40}$$

Define $\widehat{Q} := Q - \mu_1 \times \Phi^{-T}\Phi^{-1}$ and $\widehat{\Sigma}_v := \Sigma_v - \mu_2 \times I$.

Computationally-Cheap SDP Relaxation of SODC: This optimization problem is defined as the minimization of

$$\text{trace}\{P\Sigma_d + M\Sigma_v + \mu_2 R\mathbf{W}_{33} + \alpha\mathbf{W}_{33} + K^T RK\widehat{\Sigma}_v\} \tag{4.41}$$

subject to the constraints

$$\begin{bmatrix} G-\mu_1\mathbf{W}_{22} & G & (AG+BL)^T & L^T \\ G & \widehat{Q}^{-1} & 0 & 0 \\ AG+BL & 0 & G & 0 \\ L & 0 & 0 & R^{-1} \end{bmatrix} \succeq 0, \tag{4.42a}$$

$$\begin{bmatrix} P & I \\ I & G \end{bmatrix} \succeq 0, \tag{4.42b}$$

$$\begin{bmatrix} M & (BK)^T \\ BK & G \end{bmatrix} \succeq 0, \tag{4.42c}$$

$$\mathbf{W} := \left[\begin{array}{c:c:c} I_n & \Phi^{-1}G & \begin{bmatrix} K^T \\ 0 \end{bmatrix} \\ \hdashline G\Phi^{-T} & \mathbf{W}_{22} & L^T \\ \hdashline \begin{bmatrix} K & 0 \end{bmatrix} & L & \mathbf{W}_{33} \end{array}\right] \succeq 0, \tag{4.42d}$$

$$K \in \mathcal{K}, \tag{4.42e}$$

$$\mathbf{W}_{33} \in \mathcal{K}^2, \tag{4.42f}$$

with the parameter set $\{K, L, G, P, M, \mathbf{W}\}$.

It should be noted that the constraint (4.42c) ensures that the relation $M = (BK)^T G^{-1}(BK)$ occurs at optimality.

Theorem 10. *The computationally-cheap SDP relaxation is a convex relaxation of the SODC problem. Furthermore, the relaxation is exact if and only if possesses a solution* $(K^{opt}, L^{opt}, P^{opt}, G^{opt}, M^{opt}, \mathbf{W}^{opt})$ *such that rank*$\{\mathbf{W}^{opt}\} = n$.

Proof. Since the proof is similar to that of the infinite-horizon case presented earlier, it is omitted here. □

For the retrieval of a near-optimal controller, the Direct Recovery Method delineated for the infinite-horizon ODC problem can be readily deployed. However, the Indirect Recovery Method explained earlier should be modified.

Indirect Recovery Method for SODC: Let $(K^{\text{opt}}, L^{\text{opt}}, P^{\text{opt}}, G^{\text{opt}}, M^{\text{opt}}, \mathbf{W}^{\text{opt}})$ denote a solution of the computationally-cheap SDP relaxation of SODC. A near-optimal controller $\hat{K}$ for the SODC problem is recovered by solving a convex program with the variables $K \in \mathcal{K}$ and $\gamma \in \mathbb{R}$ to minimize the cost function

$$\varepsilon \times \gamma + \text{trace}\{(BK)^T (G^{\text{opt}})^{-1} (BK)\Sigma_v + K^T R K \Sigma_v + \alpha\, K K^T\} \tag{4.43}$$

subject to the constraint

$$\begin{bmatrix} (G^{\text{opt}})^{-1} - Q + \gamma I_n & (A+BKC)^T & (KC)^T \\ (A+BKC) & G^{\text{opt}} & 0 \\ (KC) & 0 & R^{-1} \end{bmatrix} \succ 0 \tag{4.44}$$

where ε is a pre-specified nonnegative number.

The above recovery method is obtained by assuming that G^{opt} is the optimal value of the inverse Lyapunov matrix for the ODC problem.

4.6 Mass-Spring and Random Systems

In this section, we elucidate the results of this chapter on a mass-spring system and 100 random system. We will solve thousands of SDP relaxations for these systems and evaluate their performance for different control topologies and a wide range of values for $(\alpha, \Sigma_d, \Sigma_v)$. Note that the computation time for each SDP relaxation is from a fraction of a second to 4 seconds on a desktop computer with an Intel Core i7 quad-core 3.4 GHz CPU and 16 GB RAM.

4.6.1 Mass-Spring Systems

In this subsection, the aim is to evaluate the performance of the developed controller design techniques on the *Mass-Spring* system, as a classical physical system. Consider a mass-spring system consisting of N masses. This system is exemplified in Figure 4.5 for $N = 2$. The system can be modeled in the continuous-time domain as

$$\dot{x}_c(t) = A_c x_c(t) + B_c u_c(t) \tag{4.45}$$

where the state vector $x_c(t)$ can be partitioned as $[o_1(t)^T \ o_2(t)^T]$ with $o_1(t) \in \mathbb{R}^n$ equal to the vector of positions and $o_2(t) \in \mathbb{R}^n$ equal to the vector of velocities of the N masses. We assume

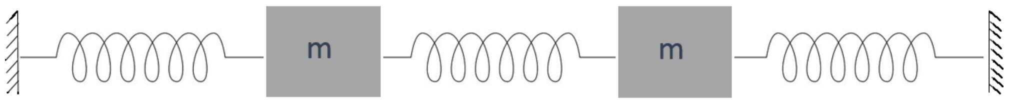
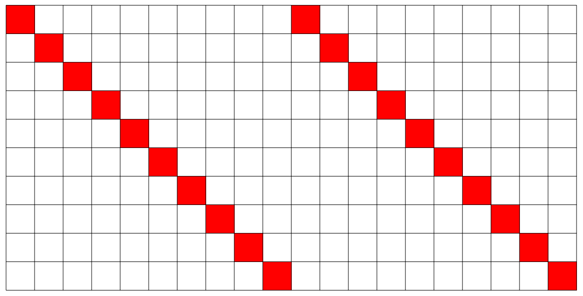
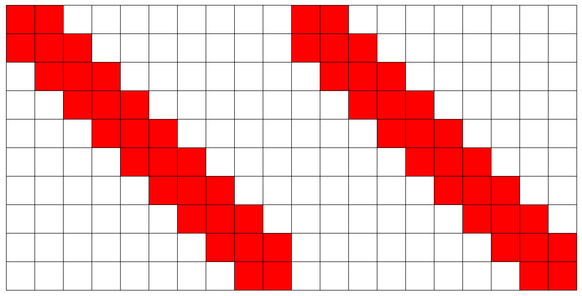


Figure 4.5: Mass-spring system with two masses



(a) Decentralized control structure



(b) Distributed control structure

Figure 4.6: Two different structures for the controller K: (a) Decentralized control structure, (b) Distributed control structure. The free parameters are colored in red (uncolored entries are set to zero).

that $N = 10$ and adopt the values of A_c and B_c from [68]. The goal is to design a static sampled-data controller with a pre-specified structure (i.e., the controller is composed of a sampler, a static discrete-time structured controller and a zero-order holder). Two ODC problems will be solved below.

Infinite-Horizon ODC: In this experiment, we first discretize the system with the sampling time of 0.1 second and denote the obtained system as

$$x[\tau+1] = Ax[\tau] + Bu[\tau], \qquad \tau = 0, 1, ... \tag{4.46}$$

It is aimed to design a constrained controller $u[\tau] = Kx[\tau]$ to minimize the infinite sum cost function

$$\sum_{\tau=0}^{\infty} \left(x[\tau]^T x[\tau] + u[\tau]^T u[\tau] \right) \tag{4.47}$$

with $x[0]$'s entries being drawn from a normal distribution. To study the effects of the initial state on the designed near-optimal controller, we generated 100 random initial states. We then solved the computationally-cheap SDP relaxation combined with the Direct Recovery Method to design a decentralized controller (shown in Figure 4.6 (a)) minimizing the cost function (4.47). The free parameters of each controller are colored in red in this figure. Structure (a) corresponds to a fully decentralized controller, where each local controller has access to the position and velocity of its associated mass. The values of controllers' parameters are depicted in Figure 4.7, where the 20 points on the x-axis represent 20 different entries of the designed decentralized controller. As can be seen, the parameters of the controller vary over the 100 trials. This contrasts with the fact that the optimal controller associated with a centralized (classical) LQR problem is universally optimal and its parameters are independent of the initial state. Define a measure of near-global optimality as follows:

$$\text{Optimality degree (\%)} = 100 - \frac{\text{upper bound - lower bound}}{\text{upper bound}} \times 100$$

where

- *Lower bound:* is equal to the optimal objective value of the SDP relaxation, which serves as a lower bound on the minimum value of the cost function (4.47).
- *Upper bound:* corresponds to the cost function (4.47) at a near-optimal controller $\hat{K}$ retrieved using the Direct Recovery Method. This number serves as an upper bound on the minimum value of the cost function (4.47).

The optimality degrees of the controllers designed for these 100 random trials are depicted in Figure 4.8. As can be seen, the optimality degree is better than 95% for more than 98 trials. It should be mentioned that all of these controllers stabilize the closed-loop system.

Stochastic ODC: In this experiment, two control structures of "decentralized" and "distributed" (shown in Figures 4.6(a) and (b)) will be studied for the matrix $K \in \mathbb{R}^{10\times 20}$. Structure (b) corresponds to a distributed controller, in which limited communications between neighboring local controllers is allowed. We assume that the system is subject to both input disturbance and measurement noise. Consider the case $\Sigma_d = I$ and $\Sigma_v = \sigma I$, where σ varies from 0 to 5. Using the computationally-cheap SDP relaxation in conjunction with the indirect recovery method, a near-optimal controller is designed for each of the aforementioned control structures under various noise levels. The results are reported in Figure 4.9. The structured controllers designed using the SDP relaxation are all stable with optimality degrees higher than 95% in the worst case and close to 99% in many cases.

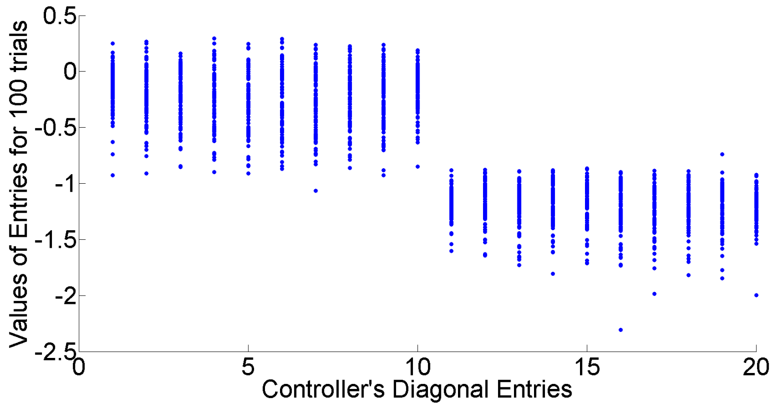


Figure 4.7: The near-optimal values of the free parameters of the decentralized controller $\hat{K}$ for a mass-spring system under 100 random initial states. Corresponding to each free parameter $i \in \{1, 2, ..., 20\}$, the 100 values of this parameter (associated with different trials) are shown as 100 points on a vertical line.

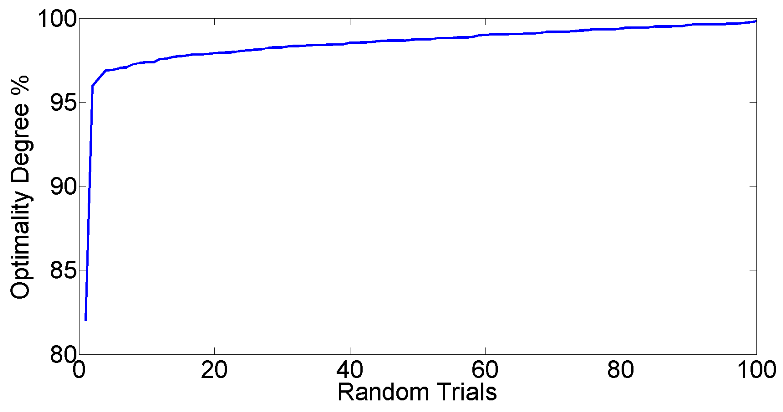
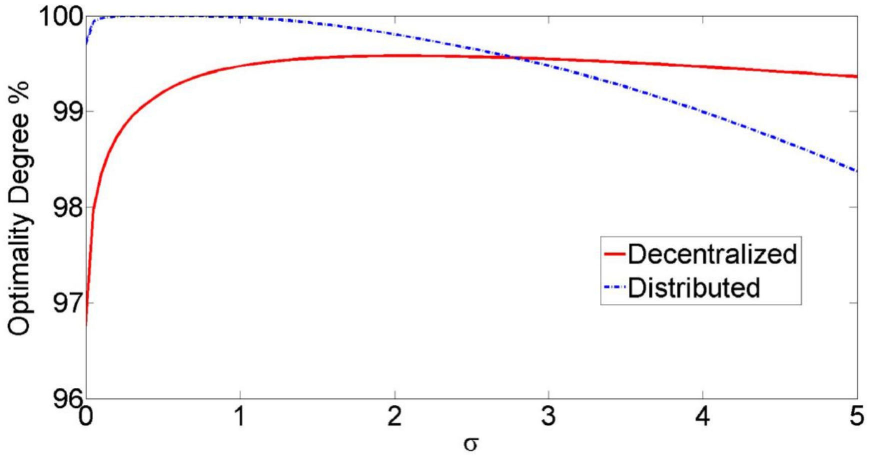
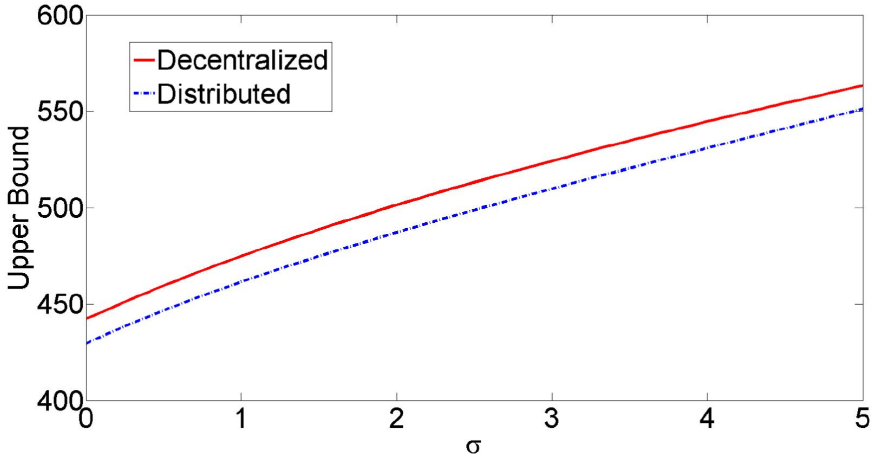


Figure 4.8: Optimality degree (%) of the decentralized controller $\hat{K}$ for a mass-spring system under 100 random initial states.



(a) Optimality degree of the near-optimal controller for a stochastic mass spring system.



(b) Cost of the near-optimal controller for a stochastic mass spring system.

Figure 4.9: The optimality degree and the optimal cost of the near-optimal controller designed for the mass-spring system for two different control structures. The noise covariance matrix Σ_v is assumed to be equal to σI, where σ varies over a wide range.

4.6.2 Random Systems

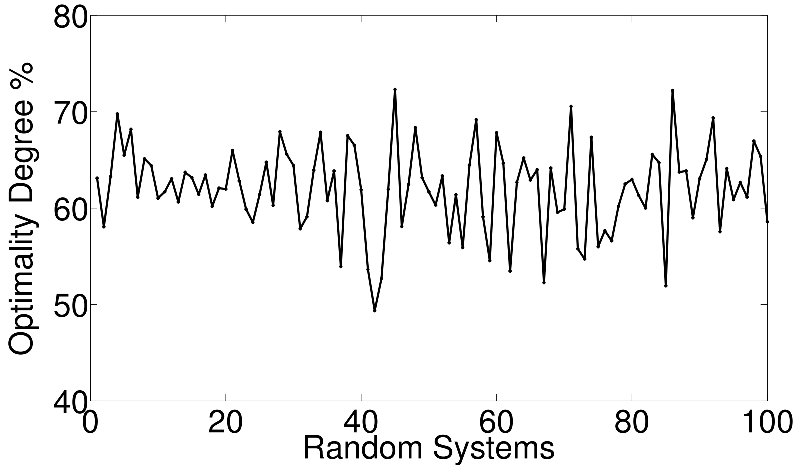
The goal of this example is to test the efficiency of the computationally-cheap SDP relaxation combined with the indirect recovery method on 100 highly-unstable random systems. Assume that $n = m = r = 25$, and that C, Q, R are identity matrices of appropriate dimensions. Suppose that $\Sigma_d = I$ and $\Sigma_v = 0$. To make the problem harder, assume that the controller under design must satisfy the hard constraint trace$\{KK^T\} \leq 2$ (to avoid a high gain K). We generated hundred random tuples $(A, B, \mathcal{K})$ according to the following rules:

- The entries of A were uniformly chosen from the interval $[0, 0.5]$ at random.
- The entries of B were uniformly chosen from the interval $[0, 1]$ at random.
- Each entry of the matrix K was enforced to be zero with the probability of 70%.

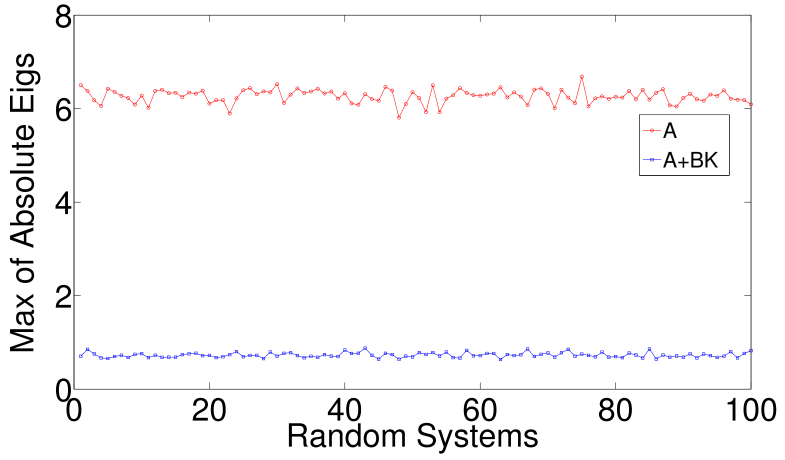
Note that although the matrices A and B are nonnegative, the matrix K under design can have both positive and negative entries. The randomly generated systems are highly unstable with the maximum absolute eigenvalue as high as 6 (instability for discrete-time systems requires a maximum magnitude less than 1). Although the control of such systems was not easy and the control structure was enforced to be 70% sparse with an enforced sparsity pattern, the proposed technique was always able to design a “stabilizing” near-optimal controller with an optimality degree between 50% and 75%. The results are reported in Figure 4.10.

4.7 Summary

This chapter studies the infinite-horizon ODC problem as well as the stochastic ODC problem. The objective is to design a fixed-order distributed controller with a pre-determined structure to minimize a quadratic cost functional for either a deterministic or a stochastic system. For both infinite-horizon ODC and stochastic ODC, the problem is cast as a rank-constrained optimization with only one non-convex constraint requiring the rank of a variable matrix to be 1. This chapter proposes a semidefinite program (SDP) as a convex relaxation, which is obtained by dropping the rank constraint. The notion of treewidth is exploited to study the rank of the minimum-rank solution of the SDP relaxation. This method is applied to the static distributed control case and it is shown that the SDP relaxation has a matrix solution with rank at most 3. Moreover, multiple



(a) Optimality degree



(b) Stability level of open-loop and closed-loop systems

Figure 4.10: The optimality degree and the stability level (maximum of the absolute eigenvalues) associated with 100 near-optimal sparse controllers designed for 100 highly-unstable random systems.

recovery methods are proposed to round the rank-3 solution to rank 1, from which a near-global controller may be retrieved. Computationally-cheap SDP relaxations are also developed for infinite-horizon ODC and stochastic ODC. These relaxations are guaranteed to exactly solve the LQR and H_2 problems for the classical centralized control problem. The results of this work are tested through thousands of simulations.

Chapter 5

Optimal Distributed Frequency Control in Power Systems

In this chapter, the results developed in Chapter 4 for Infinite-Horizon and Stochastic Optimal Distributed Control (ODC) are used to design an optimal distributed frequency controller for power systems. In general, the problem of frequency control in power systems accounts for keeping the balance between the real powers injected and demanded by the generators and the customers, respectively. There are mainly two reasons why the previous results are promising for designing such a controller. First, the integration of distributed power generation in the era of smart grid calls for efficient methods to design distributed controllers that allow certain generators to exchange real-time information with one another. Second, the intermittent nature of distributed power generation needs robust controllers that are able to deal with the uncertainty in the system introduced by non-dispatchable supplies (such as renewable energy), fluctuating loads and measurement noise. In the context of this chapter, the main objective of the unknown optimal distributed controller is to optimally adjust the mechanical power input to each generator as well as being structurally constrained by a user-defined communication topology. This pre-determined communication topology specifies which generators exchange their rotor angle and frequency measurements with one another. In this chapter, we first derive the state-space model of the power system. Then, the performance of the computationally-cheap SDP relaxation combined with the indirect recovery method for both Infinite-Horizon and Stochastic ODC is evaluated on the problem of designing an optimal distributed frequency control for IEEE 39-Bus New England Power System. These controllers are designed for four different communication topologies and we show that they are all stabilizing and with high global optimality degrees (as high as 99 % for some topologies).

5.1 Introduction

The installed capacity and energy production levels for electric generation from non-traditional renewable resources, such as solar and wind, are growing rapidly in the United States and throughout many parts of the world. The high penetration of renewable energy in the next-generation grid will reduce the greenhouse gas emission and the carbon footprint. A challenge, however, of solar and wind generation is their intermittency, making it hard to match supply and demand that result in a challenge for frequency control of power systems. This is due to the fact that most frequency/active power control actions are continuous, in contrast to the discrete switching action inherent in switched capacitor banks and tap changing transformers used for voltage/VAR control.

Frequency control in power systems usually involves three different stages that work at different timescales. As generation or load fluctuates, the primary frequency control, also known as droop control, operates continuously to stop frequency deviation through a speed governor that adjusts the generation power based on local frequency feedback. The secondary frequency control, also known as automatic generation control (AGC), operates at time steps of several seconds and adjusts the setpoints of governors in a control area in a centralized fashion to bring the frequency back to the reference value and the inter-area power flows to their scheduled values. Economic dispatch, also known as the tertiary control, operates at time steps of several minutes or up and schedules the output levels of online generators and the power flows [69], [70], [71], [72].

Early efforts of demonstrating the potential performance improvement obtained by applying optimal control theory concepts to frequency control are represented in the works [73], [74], [75], [76]. However, these efforts were impractical at the time due to the lack of wide area measurements that were needed for state estimation which is a fundamental element in optimal control. With the rapidly increasing penetration of Phasor Measurement Units (PMU) at the bulk transmission scale in the US and many other parts of the world, we could overcome the previous limitations. When coupled with tremendous advances in computational power to implement advanced control and estimation algorithms, it is believed that it is the time to revisit optimal control applications in frequency control of power systems [77].

Motivated by the idea that optimal control theory becomes a viable and promising option, the objective of this chapter is to design an optimal distributed frequency controller using the results developed in Chapter 4 for Infinite-Horizon and Stochastic Optimal Distributed Control (ODC). The main objective of the unknown optimal distributed controller is to optimally adjust the mechanical power input to each generator as well as being structurally constrained by a user-defined communication topology. This pre-determined communication topology specifies which generators exchange their rotor angle and frequency measurements with one another. In this chapter, we first derive the state-space model of the power system. Then, the performance of the computationally-cheap SDP relaxation combined with the indirect recovery method for both Infinite-Horizon and Stochastic ODC is evaluated on the problem of designing an optimal distributed frequency control for IEEE 39-Bus New England Power System.

This chapter is organized as follows. A power system dynamic model is derived in Section 5.2. The computationally-cheap SDP relaxation combined with the indirect recovery method for both Infinite-Horizon and Stochastic ODC is used to design an optimal distributed frequency controller through a case study for IEEE 39-Bus New England Power System in Section 5.3. A summary is given in Section 5.4.

5.2 Power System Dynamic Model

In this section, we derive a simple classical model of the power system. However, our result can be deployed for a complicated high-order model with nonlinear terms (our SDP relaxation may be revised to handle possible nonlinear terms in the dynamics). To derive a simple state-space model of the power system, we start with the widely-used per-unit swing equation

$$M_i\ddot{\theta}_i + D_i\dot{\theta}_i = P_{Mi} - P_{Ei} \tag{5.1}$$

where θ_i denotes the voltage (or rotor) angle at bus i (in rad), P_{Mi} is the mechanical power input to the generator at bus i (in per unit), P_{Ei} is the electrical active power injection at bus i (in per unit), M_i is the inertia coefficient of the generator at bus i (in pu-sec^2/rad), and D_i is the damping coefficient of the generator at bus i (in pu-sec/rad)[78]. The electrical real power P_{Ei} in (5.1) comes from the nonlinear AC power flow equation:

$$P_{Ei} = \sum_{j=1}^{n} |V_i||V_j| \; [\; G_{ij} \; cos(\theta_i - \theta_j) + B_{ij} \; sin(\theta_i - \theta_j) \;] \tag{5.2}$$

where n denotes the number of buses in the system, V_i is the voltage phasor at bus i, G_{ij} is the line conductance, and B_{ij} is the line susceptance. To simplify the formulation, a commonly-used technique is to approximate equation (5.2) by its corresponding DC power flow equation stated below:

$$P_{Ei} = \sum_{j=1}^{n} B_{ij}(\theta_i - \theta_j) \tag{5.3}$$

The approximation error is often small in practice due to the common practice of power engineering, which rests upon the following assumptions:

- For most networks, $G \ll B \longrightarrow G = 0$
- For most neighbouring buses, $|\theta_i - \theta_j| \leq (10^o \text{ to } 15^o)$
  $\longrightarrow sin(\theta_i - \theta_j) \approx \theta_i - \theta_j$
  $\longrightarrow cos(\theta_i - \theta_j) \approx 1$
- In per unit, $|V_i|$ is close to 1 (0.95 to 1.05)
  $\longrightarrow |V_i||V_j| \approx 1$

It is possible to rewrite (5.3) into the matrix format $P_E = L\theta$, where P_E and θ are the vectors of real power injections and voltage (or rotor) angles at only the generator buses (after removing the load buses and the intermediate zero buses). In this equation, L denotes the Laplacian matrix and can be found as follows [79]:

$$\begin{aligned} L_{ii} &= \sum_{j=1, j\neq i}^{\bar{n}} B_{ij}^{\text{Kron}} \quad \text{if } i = j \\ L_{ij} &= -B_{ij}^{\text{Kron}} \qquad\quad \text{if } i \neq j \end{aligned} \tag{5.4}$$

where B^{Kron} is the susceptance of the Kron reduced admittance matrix Y^{Kron} defined as

$$Y_{ij}^{\text{Kron}} = Y_{ij} - \frac{Y_{ik}Y_{kj}}{Y_{kk}} \quad (i, j = 1, 2, \ldots, n \text{ and } i, j \neq k) \tag{5.5}$$

where k is the index of the non-generator bus to be eliminated from the admittance matrix and $\bar{n}$ is the number of generator buses. Note that the Kron reduction method aims to eliminate the static buses of the network because the dynamics and interactions of only the generator buses are of interest [69].

By defining the rotor angle state vector as $\theta = [\theta_1, \ldots, \theta_{\bar{n}}]^T$ and the frequency state vector as $w = [w_1, \ldots, w_{\bar{n}}]^T$ and by substituting the matrix format of P_E into (5.1), the state space model of the swing equation used for frequency control in power systems could be written as

$$\begin{bmatrix} \dot{\theta} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} 0_{\bar{n}\times\bar{n}} & I_{\bar{n}} \\ -M^{-1}L & -M^{-1}D \end{bmatrix} \begin{bmatrix} \theta \\ w \end{bmatrix} + \begin{bmatrix} 0_{\bar{n}\times\bar{n}} \\ M^{-1} \end{bmatrix} P_M \tag{5.6a}$$

$$\tag{5.6b}$$

$$y = \begin{bmatrix} \theta \\ w \end{bmatrix} \tag{5.6c}$$

where $M = \mathrm{diag}(M_1, \ldots, M_{\bar{n}})$ and $D = \mathrm{diag}(D_1, \ldots, D_{\bar{n}})$. It is assumed that both rotor angle and frequency are available for measurement at each generator (implying that $C = I_{2\bar{n}}$). This is a reasonable assumption with the recent advances in Phasor Measurement Unit (PMU) technology [80].

5.3 Case Study: IEEE 39-Bus System

In this section, the performance of the computationally-cheap SDP relaxation combined with the indirect recovery method will be evaluated on the problem of designing an optimal distributed frequency control for IEEE 39-Bus New England Power System. The one-line diagram of this system is shown in Figure 5.1. The main objective of the unknown controller is to optimally adjust the mechanical power input to each generator as well as being structurally constrained by a user-defined communication topology. This pre-determined communication topology specifies which generators exchange their rotor angle and frequency measurements with one another.

By substituting the per-unit inertia (M) and damping (D) coefficients for the 10 generators of IEEE 39-Bus system [81] based on the data in Table 5.1, the continuous-time state space model matrices A_c, B_c and C_c can be found. The system is then discretized to the discrete-time model matrices A, B and C with the sampling time of 0.2 second. The initial values of the rotor angle (θ_0) were calculated by solving power (or load) flow problem for the system using MATPOWER [82].

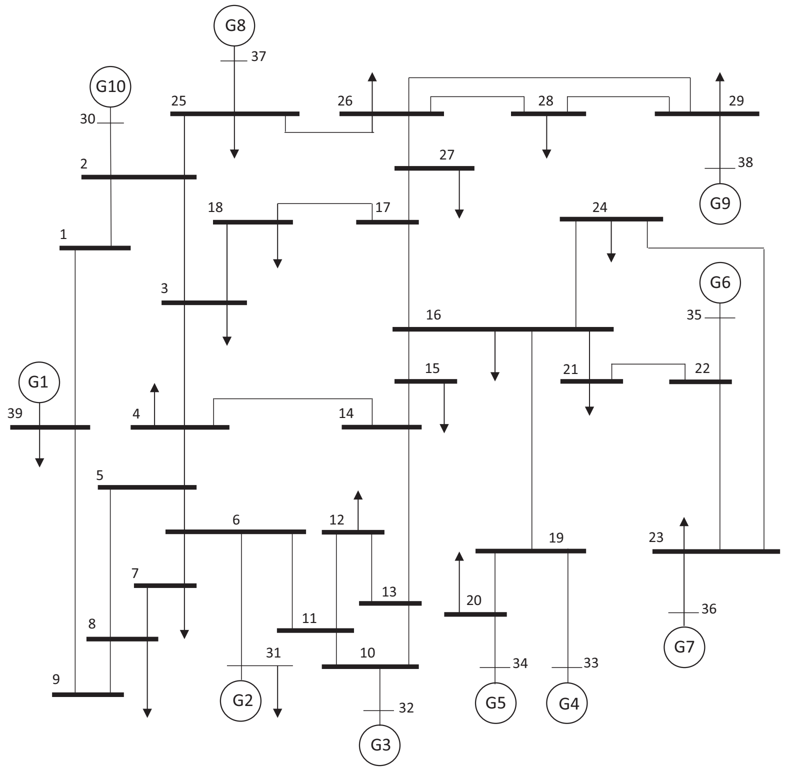


Figure 5.1: Single line diagram of IEEE 39-Bus New England Power System.

Bus	**Gen**	**M**	**D**	θ_0	w_0
30	G10	4	5	-0.0839	1.0
31	G2	3	4	0.0000	1.0
32	G3	2.5	4	0.0325	1.0
33	G4	4	6	0.0451	1.0
34	G5	2	3.5	0.0194	1.0
35	G6	3.5	3	-0.0073	1.0
36	G7	3	7.5	0.1304	1.0
37	G8	2.5	4	0.0211	1.0
38	G9	2	6.5	0.127	1.0
39	G1	6	5	-0.2074	1.0

Table 5.1: The data and initial values of generators (in per unit) for IEEE 39-Bus New England Power System.

In practice, the rotor speed does not vary significantly from synchronous speed and thus the initial frequency (w_0) was assumed to be 1.0 per unit. Both θ_0 and w_0 are reported for each generator in Table 5.1.

The 39-bus system has 10 generators, labeled as $G_1, G_2, ..., G_{10}$. Four communication topologies are considered in this work: decentralized, localized, star, and ring. In order to better understand how the interactions among the 10 generators in the system are related to the communication structures, the Kron reduced network of the system is visualized by the weighted graph shown in Figure 5.2. In a fully decentralized structure, none of the generators communicate with each other. In a localized communication structure, the generators may only communicate with their close neighbors. In a star topology, a single generator is able to communicate with all other generators in the system. The ring communication structure—forming a closed path—aims to provide communications between neighbors. These topologies are visualized in Figure 5.3. The locations of the generators in the figure are based on the exact coordinates of the power plants named in [83]. Note that G_1 represents a group of generators, but it is considered as a single node near the border between New York and Connecticut in this map. G_4 and G_5 are very close in

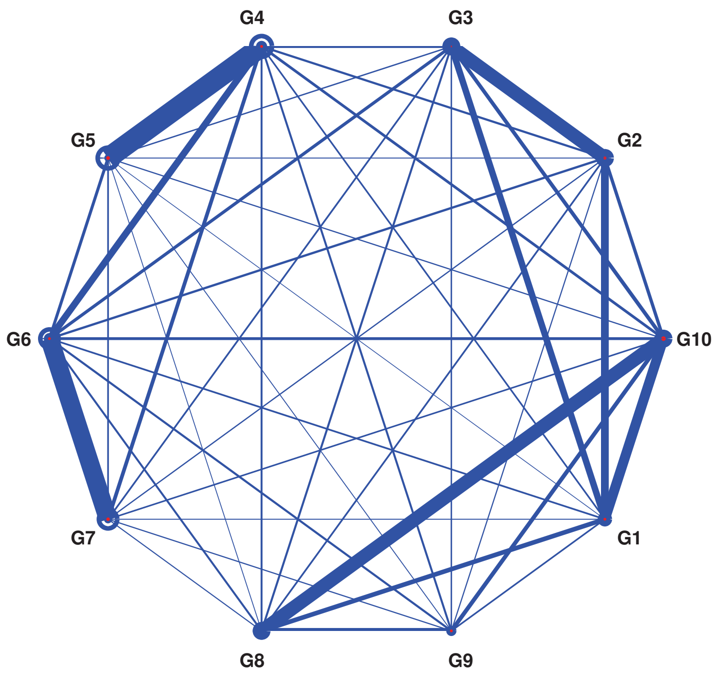


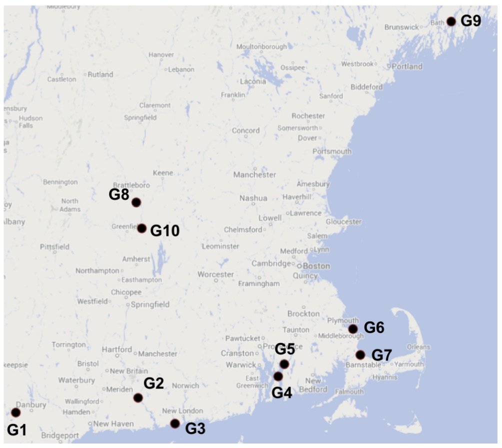
Figure 5.2: Weighted graph of the Kron reduced network of IEEE 39-Bus New England Power System. Weights (thicknesses) of all edges are normalized to the minimum off-diagonal entry of the susceptance B^{Kron}.

distance, but G_4 was somewhat shifted from its real coordinates to make the communication link between them visible in this map.

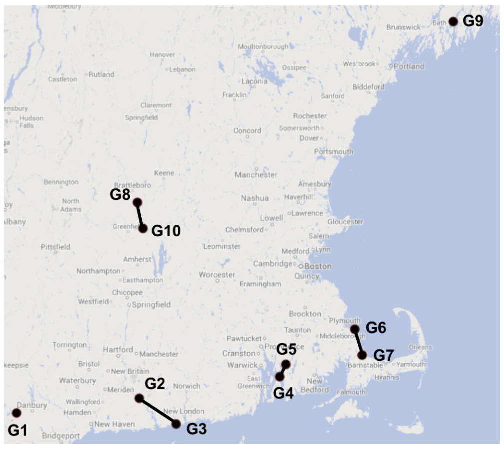
Infinite-Horizon ODC: Assume that $Q = I$ and $R = 0.1I$. Suppose also that α is a parameter between 0 and 15. The goal is to solve a an infinite-horizon ODC problem for each value of α in the interval $[0, 15]$ and for each of the four aforementioned communication topologies. This will be achieved in two steps. First, a computationally-cheap SDP relaxation is solved. Second, a near-optimal controller $\hat{K}$ is designed by choosing the best solution of the direct and indirect recovery methods. The results are reported in Figures 5.4(a)-(c). The following observations can be made:

- The designed controllers are almost 100% optimal for three control topologies of decentralized, localized and ring, and this result holds for all possible values of α. The optimality degree for the star controller is above 77%.

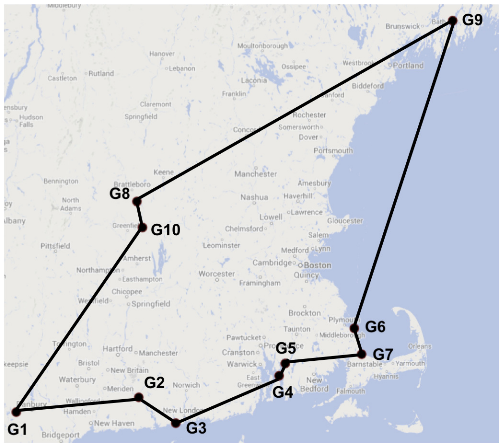
- For every value of $\alpha \in [0, 15]$, the decentralized controller has the lowest performance while



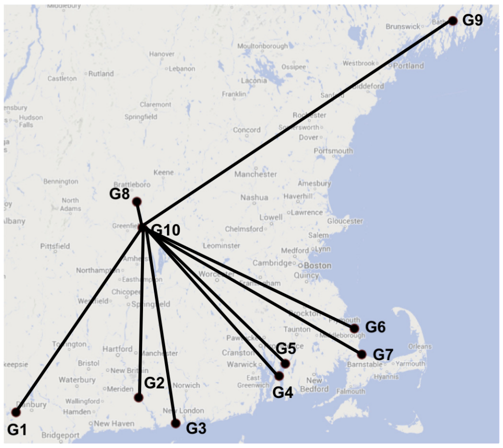
(a) Decentralized



(b) Localized



(c) Ring



(d) Star Topology (G_{10} in center)

Figure 5.3: Four communication topologies studied for IEEE 39-bus system.

the ring controller offers the best performance.

- The closed-loop system is always stable for all 4 control topologies and all possible values of α.

Stochastic ODC: Assume that the power system is under input disturbance and measurement noise. The disturbance can arise from non-dispatchable supplies (such as renewable energy) and fluctuating loads, among others. The measurement noise may account for the inaccuracy of the rotor angle and frequency measurements. Assume that Σ_d is equal to I. We consider two different scenarios:

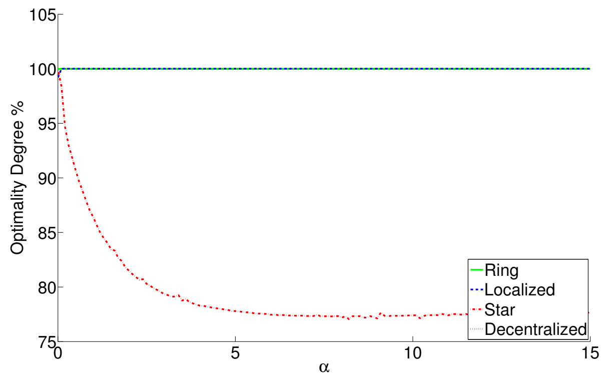
i) Suppose that $\Sigma_v = 0$, while α varies from 0 to 15. For each SODC problem, we solve a computationally-cheap SDP relaxation, from which a near-optimal solution $\hat{K}$ is designed by choosing the best solution of the direct and indirect recovery methods. The outcome is plotted in Figure 5.5.

ii) Suppose that $\alpha = 0$, while Σ_v is equal to σI with σ varying between 0 and 15. As before, we design a near-optimal controller for each SODC problem. The results are reported in Figure 5.6.

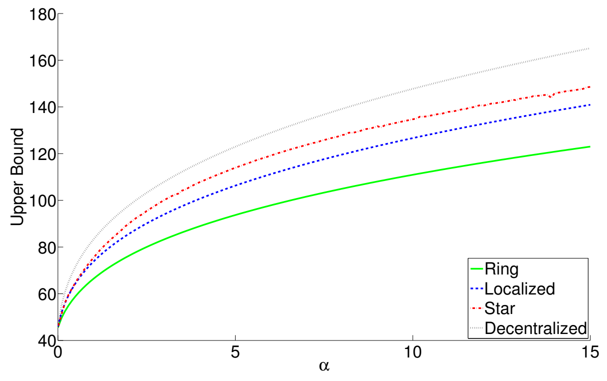
In the above experiments, we designed structured controllers to optimize an infinite-horizon ODC or a stochastic ODC problem. This was achieved by solving their associated computationally-cheap SDP relaxations. Interestingly, the designed controllers were all stabilizing (with no exception), and their optimality degrees were close to 99% in case of decentralized, localized and ring structures. In case of the star structure, the optimality degree was higher than 77% in infinite-horizon ODC and around 94% for various levels of σ and α in stochastic ODC.

5.4 Summary

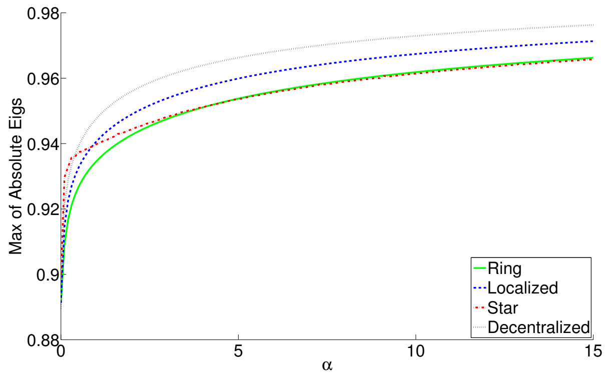
This chapter utilizes the results previously developed for Infinite-Horizon and Stochastic ODC in Chapter 4 to design an optimal distributed frequency controller for power systems. The main objective of the unknown optimal distributed controller is to optimally adjust the mechanical power input



(a) Optimality degree for infinite-horizon ODC

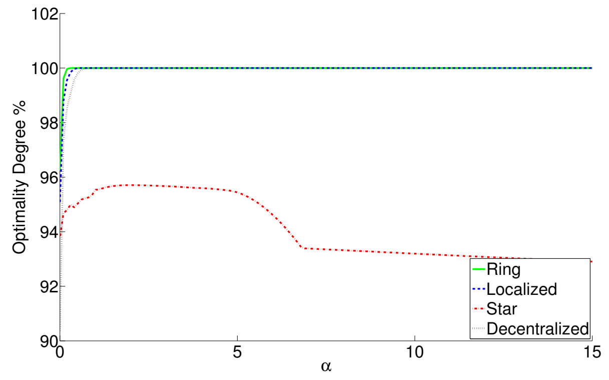


(b) Near-optimal cost for infinite-horizon ODC

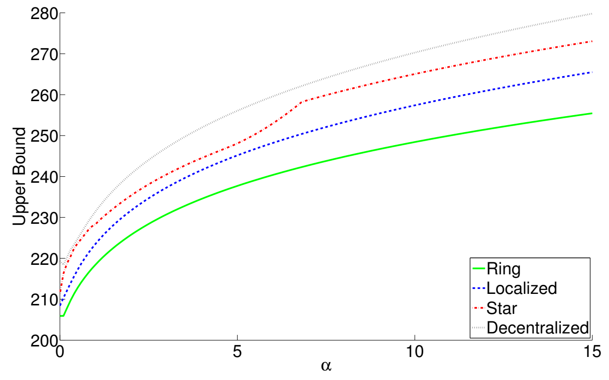


(c) Stability degree for infinite-horizon ODC

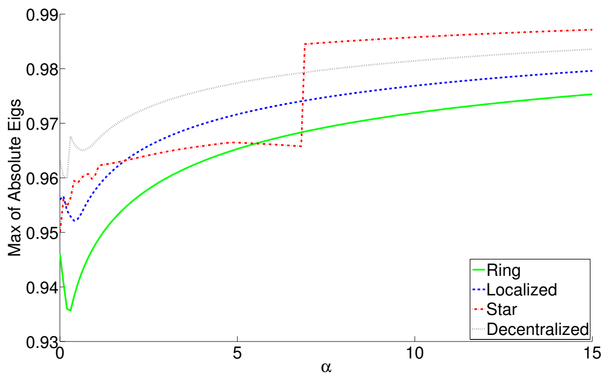
Figure 5.4: A near-optimal controller $\hat{K}$ is designed to solve the infinite-horizon ODC problem for every control topology given in Figure 5.3 and every α between 0 and 15: (a) optimality degree, (b) near-optimal cost, and (c) closed-loop stability (maximum of the absolute eigenvalues of the closed-loop system).



(a) Optimality degree for stochastic ODC

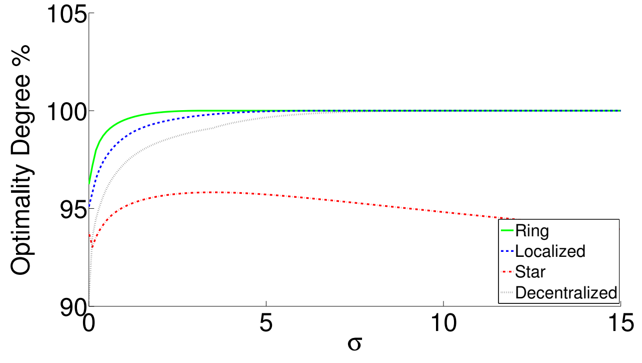


(b) Near-optimal cost for stochastic ODC

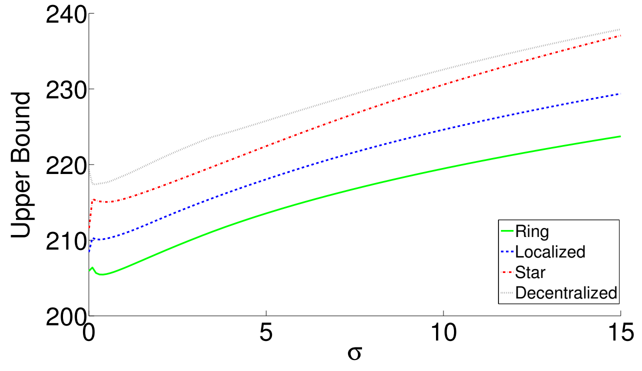


(c) Stability degree for stochastic ODC

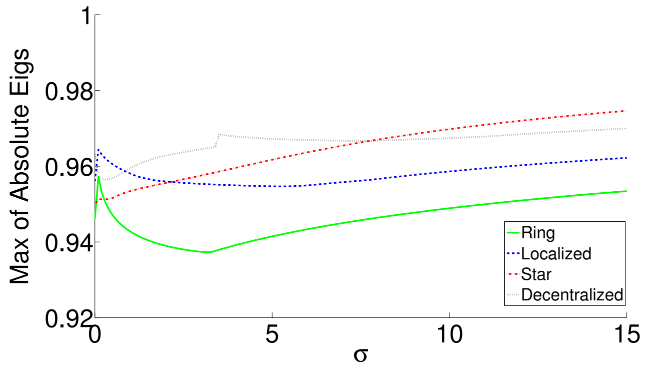
Figure 5.5: A near-optimal controller $\hat{K}$ is designed to solve the stochastic ODC problem for every control topology given in Figure 5.3 and every α between 0 and 15 under the assumptions that $\Sigma_d = I$ and $\Sigma_v = 0$: (a) optimality degree, (b) near-optimal cost, and (c) closed-loop stability



(a) Optimality degree for stochastic ODC



(b) Near-optimal cost for stochastic ODC



(c) Stability degree for stochastic ODC

Figure 5.6: A near-optimal controller $\hat{K}$ is designed to solve the stochastic ODC problem for every control topology given in Figure 5.3 and every σ between 0 and 15 under the assumptions that $\Sigma_d = I$, $\alpha = 0$ and $\Sigma_v = \sigma I$: (a) optimality degree, (b) near-optimal cost, and (c) closed-loop stability

to each generator as well as being structurally constrained by a user-defined communication topology. This pre-determined communication topology specifies which generators exchange their rotor angle and frequency measurements with one another. The performance of the computationally-cheap SDP relaxation combined with the indirect recovery method for both Infinite-Horizon and Stochastic ODC is evaluated on the problem of designing an optimal distributed frequency control for IEEE 39-Bus New England Power System. These controllers are designed for four different communication topologies and it is shown that the controllers are all stabilizing with global optimality degrees close to 99% in case of decentralized, localized and ring structures. In case of the star structure, the optimality degree was higher than 77% in infinite-horizon ODC and around 94% for various levels of σ and α in stochastic ODC.

Chapter 6

Conclusions and Future Work

This dissertation is concerned with developing efficient, scalable and distributed algorithms for solving real-world large-scale optimization problems that arise in complex systems such as power networks and distributed control systems. This dissertation addresses four problems, each involving the development of an efficient optimization algorithm. In what follows, the contributions made for each problem are first summarized and possible future directions are then outlined.

Chapter 2: In this chapter, a fast and parallelizable algorithm is developed for an arbitrary decomposable semidefinite program (SDP). To formulate a decomposable SDP, we consider a multi-agent canonical form represented by a graph, where each agent (node) is in charge of computing its corresponding positive semidefinite matrix. The motivation behind the multi-agent formulation is that an arbitrary sparse SDP problem can be converted to a decomposable SDP by means of the Chordal extension and matrix completion theorems. Using the alternating direction method of multipliers (ADMM), we develop a distributed algorithm to solve the underlying SDP problem. At every iteration, each agent performs simple computations (matrix multiplication and eigenvalue decomposition) without having to solve any optimization subproblem, and then communicates some information to its neighbors. By deriving a Lyapunov-type non-increasing function, it is shown that the proposed algorithm converges as long as Slater's conditions hold. Simulations results on large-scale SDP problems with a few million variables are offered to elucidate the efficacy of this work. Some of the possible future research directions, are as follows:

- To accelerate the proposed first-order method and obtain a faster convergence, it is important to study how this algorithm can be combined with Nesterov method.

- Since ADMM is sensitive to the condition number of the problem's data, it is important to study how efficient and cheap preconditioning techniques could be used to speed up the convergence for ill-conditioned problems.
- The distributed algorithm designed here is a synchronous algorithm in which each agent should wait for the messages from the neighbours before starting the new iteration. An asynchronous version of the previous algorithm should be developed to eliminate the need for a global clock that commands the agents when to start the computations and when to start exchanging data. Such algorithm is easier to be implemented in a multi-machine setting if needed so.

Chapter 3: Motivated by the application of SDPs to power networks, the objective of this chapter is to design a fast and parallelizable algorithm for solving sparse SDPs corresponding to power optimization problems. To this end, the underling sparsity structure of a given SDP problem is captured using a tree decomposition technique, leading to a decomposed SDP problem. A highly distributed/parallelizable numerical algorithm is developed for solving the decomposed SDP, based on the ADMM method in the primal domain. Each iteration of the designed algorithm has a closed-form solution, which involves multiplications and eigenvalue decompositions over certain submatrices induced by the tree decomposition of the sparsity graph. The proposed algorithm is applied to the classical optimal power flow problem, and also evaluated on IEEE benchmark systems. The proposed algorithm has a very low computational complexity for power systems because real-world power networks have low treewidth. All of the future research directions previously discussed for Chapter 2 are valid here to improve the convergence of the proposed algorithm in the primal domain. Another direction is to study other power optimization problems such as state estimation.

Chapter 4: This chapter studies the infinite-horizon optimal distributed control (ODC) problem as well as the stochastic ODC problem. The objective is to design a fixed-order distributed controller with a pre-determined structure to minimize a quadratic cost functional for either a deterministic or a stochastic system. Both problems are cast as a rank-constrained optimization problem with only one non-convex constraint requiring the rank of a variable matrix to be 1. This chapter proposes an SDP problem as a convex relaxation, which is obtained by dropping the rank constraint. The notion of treewidth is exploited to study the rank of the minimum-rank solution of the SDP relaxation. This method is applied to the static distributed control case and it is shown that the SDP relaxation has a matrix solution with rank at most 3. Moreover, multiple recovery methods are proposed to round the rank-3 solution to rank 1, from which a near-global controller may be retrieved. Computationally-cheap SDP relaxations are also developed for infinite-horizon ODC and stochastic ODC. The results of this work are tested on thousands of simulations. Some of the possible extensions as future work, are as follows:

- One direction is to study the design of a robust distributed controller with a known structure to minimize a quadratic cost function either in the worst case or in expectation with respect to the random variable Δ. This corresponds to the case when the system matrices $A(\Delta)$, $B(\Delta)$, $C(\Delta)$ and $D(\Delta)$ depend on some uncertainty vector Δ belonging to some uncertainty region.

- Another possibility is to generalize the results obtained for linear systems to certain nonlinear systems with the aim of representing a sufficiently detailed (approximate) model of a real-world system.

Chapter 5: This chapter utilizes the results previously developed for Infinite-Horizon and Stochastic ODC in Chapter 4 to design an optimal distributed frequency control in power systems. The performance of the computationally-cheap SDP relaxation combined with the indirect recovery method for both Infinite-Horizon and Stochastic ODC is evaluated on the problem of designing an optimal distributed frequency control for IEEE 39-Bus New England Power System. These controllers are designed for four different communication topologies and shown to be all stabilizing with optimality degrees close to 99% in some cases. A simple classical model of the power system was used. A necessary future work is to consider a more realistic high-order model for the power system.

Bibliography

[1] J. Lavaei and S. H. Low, "Zero duality gap in optimal power flow problem," *IEEE Transactions on Power Systems*, vol. 27, no. 1, pp. 92–107, 2012.

[2] M. X. Goemans and D. P. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming," *Journal of the ACM (JACM)*, vol. 42, no. 6, pp. 1115–1145, 1995.

[3] L. Vandenberghe and S. Boyd, "Semidefinite programming," *SIAM Review*, vol. 38, pp. 49–95, 1994.

[4] D. Gabay and B. Mercier, "A dual algorithm for the solution of nonlinear variational problems via finite element approximation," *Computers and Mathematics with Applications*, vol. 2, no. 1, pp. 17 – 40, 1976.

[5] R. Glowinski and A. Marroco, "Sur l'approximation, par lments finis d'ordre un, et la rsolution, par pnalisation-dualit d'une classe de problmes de dirichlet non linaires," *ESAIM: Mathematical Modelling and Numerical Analysis - Modlisation Mathmatique et Analyse Numrique*, vol. 9, no. R2, pp. 41–76, 1975. [Online]. Available: http://eudml.org/doc/193269

[6] J. Eckstein and W. Yao, "Augmented lagrangian and alternating direction methods for convex optimization: A tutorial and some illustrative computational results," *RUTCOR Research Reports*, vol. 32, 2012.

[7] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[8] J. Momoh, R. Adapa, and M. El-Hawary, "A review of selected optimal power flow literature to 1993. I. nonlinear and quadratic programming approaches," *IEEE Trans. Power Syst.,*, vol. 14, no. 1, pp. 96–104, Feb 1999.

[9] J. Carpentier, "Contribution a l etude du dispatching economique," *Bulletin Society Francaise Electricians*, vol. 3, no. 8, pp. 431–447, 1962.

[10] M. B. Cain, R. P. O'Neill, and A. Castillo, "History of optimal power flow and formulations," Federal Energy Regulatory Commission FERC, Tech. Rep., December 2012.

[11] J. Lavaei and S. H. Low, "Zero duality gap in optimal power flow problem," *IEEE Transactions on Power Systems*, vol. 27, no. 1, pp. 92–107, 2012.

[12] R. Madani, M. Ashraphijuo, and J. Lavaei, "Promises of conic relaxation for contingency-constrained optimal power flow problem," in *52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton),*, Sept 2014, pp. 1064–1071.

[13] S. Sojoudi and J. Lavaei, "Physics of power networks makes hard optimization problems easy to solve," in *IEEE Power and Energy Society General Meeting*, 2012.

[14] S. H. Low, "Convex relaxation of optimal power flow (part I, II)," *IEEE Trans. Control of Network Systems*, vol. 1, no. Part I: 1; Part II: 2, pp. Part–I, 2014.

[15] J. Lavaei, D. Tse, and B. Zhang, "Geometry of power flows and optimization in distribution networks," *IEEE Trans. Power Syst.*, vol. 29, no. 2, pp. 572–583, 2014.

[16] S. Sojoudi and J. Lavaei, "Exactness of semidefinite relaxations for nonlinear optimization problems with underlying graph structure," *SIAM J. Optimiz.*, vol. 24, no. 4, pp. 1746–1778, 2014.

[17] L. Gan, N. Li, U. Topcu, and S. H. Low, "Optimal power flow in distribution networks," *Proc. 52nd IEEE Conference on Decision and Control*, 2013.

[18] R. Madani, S. Sojoudi, and J. Lavaei, "Convex relaxation for optimal power flow problem: Mesh networks," *IEEE Trans. Power Syst.*, vol. 30, no. 1, pp. 199–211, 2015.

[19] H. S. Witsenhausen, "A counterexample in stochastic optimum control," *SIAM Journal of Control*, vol. 6, no. 1, 1968.

[20] J. N. Tsitsiklis and M. Athans, "On the complexity of decentralized decision making and detection problems," *Conference on Decision and Control*, 1984.

[21] L. Vandenberghe and S. Boyd, "Semidefinite programming," *SIAM Review*, 1996.

[22] S. Boyd and L. Vandenberghe, *Convex Optimization.* Cambridge, 2004.

[23] Y. Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$," *Soviet Mathematics Doklady*, vol. 27, no. 2, pp. 372–376, 1983.

[24] T. Goldstein, B. O'Donoghue, S. Setzer, and R. Baraniuk, "Fast alternating direction optimization methods," *SIAM Journal on Imaging Sciences*, vol. 7, no. 3, pp. 1588–1623, 2014.

[25] P. Giselsson and S. Boyd, "Diagonal scaling in Douglas–Rachford splitting and ADMM," *53rd IEEE Conference on Decision and Control*, 2014.

[26] B. He and X. Yuan, "On the $O(1/n)$ convergence rate of the Douglas–Rachford alternating direction method," *SIAM Journal on Numerical Analysis*, vol. 50, no. 2, pp. 700–709, 2012.

[27] R. D. C. Monteiro and B. F. Svaiter, "Iteration-complexity of block-decomposition algorithms and the alternating direction method of multipliers," *SIAM Journal on Optimization*, vol. 23, no. 1, pp. 475–507, 2013.

[28] E. Wei and A. Ozdaglar, "On the $O(1/k)$ Convergence of Asynchronous Distributed Alternating Direction Method of Multipliers," *ArXiv e-prints*, Jul. 2013.

[29] R. Nishihara, L. Lessard, B. Recht, A. Packard, and M. I. Jordan, "A general analysis of the convergence of ADMM," *arXiv preprint arXiv:1502.02009*, 2015.

[30] R. Madani, G. Fazelnia, S. Sojoudi, and J. Lavaei, "Low-rank solutions of matrix inequalities with applications to polynomial optimization and matrix completion problems," *IEEE Conference on Decision and Control*, 2014.

[31] M. Fukuda, M. Kojima, K. Murota, and K. Nakata, "Exploiting sparsity in semidefinite programming via matrix completion I: General framework," *SIAM Journal on Optimization*, vol. 11, no. 3, pp. 647–674, 2001.

[32] R. Grone, C. R. Johnson, E. M. Sá, and H. Wolkowicz, "Positive definite completions of partial hermitian matrices," *Linear algebra and its applications*, vol. 58, pp. 109–124, 1984.

[33] Z. Wen, D. Goldfarb, and W. Yin, "Alternating direction augmented lagrangian methods for semidefinite programming," *Mathematical Programming Computation*, vol. 2, no. 3-4, pp. 203–230, 2010.

[34] Y. Sun, M. S. Andersen, and L. Vandenberghe, "Decomposition in conic optimization with partially separable structure," *SIAM Journal on Optimization*, vol. 24, no. 2, pp. 873–897, 2014.

[35] B. He and X. Yuan, "On non-ergodic convergence rate of Douglas–Rachford alternating direction method of multipliers," *Numerische Mathematik*, pp. 1–11, 2012.

[36] H. Mittelmann, "An independent benchmarking of SDP and SOCP solvers," *Mathematical Programming*, vol. 95, no. 2, pp. 407–430, 2003. [Online]. Available: http://dx.doi.org/10.1007/s10107-002-0355-5

[37] B. Kim and R. Baldick, "Coarse-grained distributed optimal power flow," *IEEE Trans. Power Syst.,*, vol. 12, no. 2, pp. 932–939, May 1997.

[38] R. Baldick, B. Kim, C. Chase, and Y. Luo, "A fast distributed implementation of optimal power flow," *IEEE T. Power Syst.,*, vol. 14, no. 3, pp. 858–864, Aug 1999.

[39] M. Kraning, E. Chu, J. Lavaei, and S. Boyd, "Dynamic network energy management via proximal message passing," *Foundations and Trends in Optimization*, vol. 1, no. 2, pp. 73–126, 2014.

[40] A. Lam, B. Zhang, and D. Tse, "Distributed algorithms for optimal power flow problem," in *51st Annual Conference on Decision and Control (CDC)*, Dec 2012, pp. 430–437.

[41] E. Dall'Anese, H. Zhu, and G. Giannakis, "Distributed optimal power flow for smart microgrids," *IEEE Trans. Smart Grid,*, vol. 4, no. 3, pp. 1464–1475, Sept 2013.

[42] Q. Peng and S. Low, "Distributed algorithm for optimal power flow on a radial network," in *53rd Annual Conference on Decision and Control (CDC)*, Dec 2014, pp. 167–172.

[43] M. Fukuda, M. Kojima, K. Murota, and K. Nakata, "Exploiting sparsity in semidefinite programming via matrix completion I: General framework," *SIAM J. Optimiz.*, vol. 11, no. 3, pp. 647–674, 2001.

[44] H. L. Bodlaender and A. M. Koster, "Treewidth computations I. upper bounds," *Inform. Comput.*, vol. 208, no. 3, pp. 259–275, 2010.

[45] ——, "Treewidth computations II. lower bounds," *Inform. Comput.*, vol. 209, no. 7, pp. 1103–1119, 2011.

[46] K. Nakata, K. Fujisawa, M. Fukuda, M. Kojima, and K. Murota, "Exploiting sparsity in semidefinite programming via matrix completion II: Implementation and numerical results," *Math. Program.*, vol. 95, no. 2, pp. 303–327, 2003.

[47] M. Laurent, "Polynomial instances of the positive semidefinite and Euclidean distance matrix completion problems," *SIAM J. Matrix Aanl. A.*, vol. 22, no. 3, pp. 874–894, 2001.

[48] M. Laurent and A. Varvitsiotis, "A new graph parameter related to bounded rank positive semidefinite matrix completions," *Math. Program.*, vol. 145, no. 1-2, pp. 291–325, 2014.

[49] R. Madani, G. Fazelnia, S. Sojoudi, and J. Lavaei, "Low-rank solutions of matrix inequalities with applications to polynomial optimization and matrix completion problems," in *53rd Annual Conference on Decision and Control (CDC)*, Dec 2014, pp. 4328–4335.

[50] R. Madani, M. Ashraphijuo, and J. Lavaei, "Promises of conic relaxation for contingency-constrained optimal power flow problem," to appear in *IEEE T. Power Syst.*, 2015, http://www.ieor.berkeley.edu/~lavaei/SCOPF_2014.pdf.

[51] N. J. Higham, "Computing a nearest symmetric positive semidefinite matrix," *Linear Algebra Appl.*, vol. 103, pp. 103–118, May. 1988.

[52] R. D'Andrea and G. Dullerud, "Distributed control design for spatially interconnected systems," *IEEE Transactions on Automatic Control*, vol. 48, no. 9, pp. 1478–1495, 2003.

[53] B. Bamieh, F. Paganini, and M. A. Dahleh, "Distributed control of spatially invariant systems," *IEEE Transactions on Automatic Control*, vol. 47, no. 7, pp. 1091–1107, 2002.

[54] C. Langbort, R. Chandra, and R. D'Andrea, "Distributed control design for systems interconnected over an arbitrary graph," *IEEE Transactions on Automatic Control*, vol. 49, no. 9, pp. 1502–1519, 2004.

[55] N. Motee and A. Jadbabaie, "Optimal control of spatially distributed systems," *IEEE Transactions on Automatic Control*, vol. 53, no. 7, pp. 1616–1629, 2008.

[56] G. Dullerud and R. D'Andrea, "Distributed control of heterogeneous systems," *IEEE Transactions on Automatic Control*, vol. 49, no. 12, pp. 2113–2128, 2004.

[57] T. Keviczky, F. Borrelli, and G. J. Balas, "Decentralized receding horizon control for large scale dynamically decoupled systems," *Automatica*, vol. 42, no. 12, pp. 2105–2115, 2006.

[58] F. Borrelli and T. Keviczky, "Distributed LQR design for identical dynamically decoupled systems," *IEEE Transactions on Automatic Control*, vol. 53, no. 8, pp. 1901–1912, 2008.

[59] D. D. Siljak, "Decentralized control and computations: status and prospects," *Annual Reviews in Control*, vol. 20, pp. 131–141, 1996.

[60] J. Lavaei, "Decentralized implementation of centralized controllers for interconnected systems," *IEEE Transactions on Automatic Control*, vol. 57, no. 7, pp. 1860–1865, 2012.

[61] A. Barvinok, "Problems of distance geometry and convex properties of quadartic maps," *Discrete and Computational Geometry*, vol. 12, pp. 189–202, 1995.

[62] G. Pataki, "On the rank of extreme matrices in semidefinite programs and the multiplicity of optimal eigenvalues," *Mathematics of Operations Research*, vol. 23, pp. 339–358, 1998.

[63] R. Madani, G. Fazelnia, S. Sojoudi, and J. Lavaei, "Low-rank solutions of matrix inequalities with applications to polynomial optimization and matrix completion problems," *Conference on Decision and Control*, 2014.

[64] J. F. Sturm and S. Zhang, "On cones of nonnegative quadratic functions," *Mathematics of Operations Research*, vol. 28, pp. 246–267, 2003.

[65] S. Sojoudi and J. Lavaei, "Physics of power networks makes hard optimization problems easy to solve," *IEEE Power & Energy Society General Meeting*, 2012.

[66] ——, "On the exactness of semidefinite relaxation for nonlinear optimization over graphs: Part I," *IEEE Conference on Decision and Control*, 2013.

[67] ——, "On the exactness of semidefinite relaxation for nonlinear optimization over graphs: Part II," *IEEE Conference on Decision and Control*, 2013.

[68] F. Lin, M. Fardad, and M. R. Jovanovi, "Design of optimal sparse feedback gains via the alternating direction method of multipliers," *IEEE Transactions on Automatic Control*, vol. 58, no. 9, 2013.

[69] A. R. Bergen and V. Vittal, *Power Systems Analysis.* Prentice Hall, 1999, vol. 2.

[70] M. Ilic, "From hierarchical to open access electric power systems," *Proceedings of the IEEE*, vol. 95, no. 5, pp. 1060–1084, May 2007.

[71] A. Kiani and A. Annaswamy, "A hierarchical transactive control architecture for renewables integration in smart grids," in *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, Dec 2012, pp. 4985–4990.

[72] C. Zhao and S. Low, "Optimal decentralized primary frequency control in power networks," in *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, Dec 2014, pp. 2467–2473.

[73] C. Fosha and O. I. Elgerd, "The megawatt-frequency control problem: A new approach via optimal control theory," *Power Apparatus and Systems, IEEE Transactions on*, vol. PAS-89, no. 4, pp. 563–577, April 1970.

[74] E. Tacker, C. Lee, T. Reddoch, and P. Julich, "Optimal control of interconnected, electric energy systems: A new formulation," *Proceedings of the IEEE*, vol. 60, no. 10, pp. 1239–1241, Oct 1972.

[75] E. Bohn and S. Miniesy, "Optimum load-frequency sampled-data control with randomly varying system disturbances," *Power Apparatus and Systems, IEEE Transactions on*, vol. PAS-91, no. 5, pp. 1916–1923, Sept 1972.

[76] P. Kambale, H. Mukai, J. Spare, and J. Zaborszky, ""a reevaluation of the normal operating state control (agc) of the power system using computer control and system theory part iii. tracking the dispatch targets with unit control"," *Power Apparatus and Systems, IEEE Transactions on*, vol. PAS-102, no. 6, pp. 1903–1912, June 1983.

[77] C. DeMarco, C. Baone, Y. Han, and B. Lesieutre, "Primary and secondary control for high penetration renewables," Power Systems Engineering Research Center PSERC, Tech. Rep., May 2012.

[78] M. A. Pai, *Energy Function Analysis for Power System Stability.* Kluwer Academic Publishers, Boston, 1989.

[79] F. Dorfler and F. Bullo, "Novel insights into lossless AC and DC power flow," *IEEE Power and Energy Society General Meeting*, 2013.

[80] M. Andreasson, D. Dimarogonas, H. Sandberg, and K. Johansson, "Distributed control of networked dynamical systems: Static feedback, integral action and consensus," *IEEE Transactions on Automatic Control*, vol. PP, no. 99, pp. 1–1, 2014.

[81] I. E. Atawi, "An advance distributed control design for wide-area power system stability," Ph.D. dissertation, Swanson School Of Engineering,, University of Pittsburgh, Pittsburgh,Pennsylvania, 2013.

[82] R. Zimmerman, C. Murillo-Sanchez, and R. Thomas, "Matpower: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Transactions on Power Systems*, vol. 26, no. 1, pp. 12–19, Feb 2011.

[83] M. Ilic, J. Lacalle-Melero, F. Nishimura, W. Schenler, D. Shirmohammadi, A. Crough, and A. Catelli, "Short-term economic energy management in a competitive utility environment," *IEEE Transactions on Power Systems*, vol. 8, no. 1, pp. 198–206, Feb 1993.

Appendix: High-performance C++ Implementation

This appendix shows the full C++ code that implements the multi-agent SDP algorithm developed in Chapter 2. The code consists of the main implementation file admm-sdp.cpp and also a header file admm-sdp.hpp.

admm-sdp.cpp

```
/*
 ADMM for Solving SDPs in Parallel

 -- This code implements a fast, parallelizable algorithm for an arbitrary decomposable
 ↪  semidefinite program (SDP).
 This code solves the the decomposable SDP problem defined below:

 ------------------------------------------------------------------------------
 min   sum_(over all agents i in V) [ tr(A_i * W_i)]

 subject to
 tr(B_j^(i) * W_i) =  c_j^(i)       for all j = 1,...., p_i  and  i  in V
 tr(D_l^(i) * W_i) <= d_l^(i)       for all l = 1,...., q_i  and  i  in V
 W_i >= 0 (PSD)
 W_i(I_ij, I_ij) = W_j(I_ji, I_ji)  for all (i, j) in E

 over the variables W_i in S^ni  for i = 1, ...., n
 ------------------------------------------------------------------------------

 -- Please check the following reference paper on which this code is based:
 - Abdulrahman Kalbat and Javad Lavaei, A Fast Distributed Algorithm for Decomposable
 ↪  Semidefinite Programs,
 Proc. 54th IEEE Conference on Decision and Control, 2015.

 -- Variables definitions and correspondance between the code and the reference paper:

 ------------------------------------------------------------------------
 | Code           |  Paper     |  Type                                   |
 ------------------------------------------------------------------------
 randAdj          |  g=(V,E)   |  Input                                  |
 edges_Set        |  E         |  Found from randAdj                     |
 mu_mult          |  mu        |  Input                                  |
 delta_less       |            |  Found from randAdj                     |
 delta_greater    |            |  Found from randAdj                     |
 n                |  |v|       |  Input or from randAdj                  |
 w_size_i         |  n_i       |  Input                                  |
 p_i              |  p_i       |  Input                                  |
 q_i              |  q_i       |  Input                                  |
 A                |  A         |  Input                                  |
 B                |  B         |  Input                                  |
 D                |  D         |  Input                                  |
 c_i              |  c_i       |  Input                                  |
 d_i              |  d_i       |  Input                              |
 I_ij             |  I_ij      |  Input                              |
 I_ji             |  I_ji      |  Input                              |
 z                |  z_i       |  Variable                           |
 v                |  v_i       |  Variable                           |
 u                |  u_i       |  Variable                           |
 R_lower          |  R_i       |  Variable                           |
 G_i_lower        |  G_i       |  Variable                           |
 Lambda_i         |  Lambda_i  |  Variable                           |
 H_ij_lower       |  H_ij      |  Variable                           |
 H_ji_lower       |  H_ji      |  Variable                           |
 H_ij_coup_lower  |  H^(ij)    |  Variable                           |
 G_ij_lower       |  G_ij      |  Variable                           |
 G_ji_lower       |  G_ji      |  Variable                           |
 H_ij_basis_map   |            |  Found from I_ij                    |
 H_ji_basis_map   |            |  Found from I_ji                    |
 H_ij_sum_tr      |  H_i_sum   |  Found from H_*_basis_map and H_*_lower |
 B_sum            |  B_i_sum   |  Found from B_lower and z           |
 D_sum            |  D_i_sum   |  Found from D_lower and v           |
 p_infeas_i_1     |  P_1       |  DIMACS error measure               |
 p_infeas_i_2     |  P_2       |  DIMACS error measure               |
 d_infeas_i_1     |  D_1       |  DIMACS error measure               |
 d_infeas_i_2[0]  |  D_2       |  DIMACS error measure               |
 d_infeas_i_2[1]  |  D_3       |  DIMACS error measure               |
 d_infeas_i_3     |  D_4       |  DIMACS error measure               |
 gap_iter         |  Gap       |  DIMACS error measure               |
 p_residue_i_1    |  delta_p1  |  primal residue                     |
 p_residue_i_2    |  delta_p4  |  primal residue                     |
 p_residue_i_3    |  delta_p2  |  primal residue                     |
 p_residue_i_4    |  delta_p3  |  primal residue                     |
 d_residue_i_1    |  delta_d1  |  dual residue                       |
 d_residue_i_2    |  delta_d2  |  dual residue                       |
 d_residue_i_3    |  delta_d3  |  dual residue                       |
 residue_sum      |  V^t       |  aggregate residue                  |
 -----------------------------------------------------------------------


 -- In order to start using the code, please open the header file "admm_sdp.h"
 and please read the definitions of the different paramters in the bottom of the file that are
 ↪   needed
 to randomly generate Multiagent SDP problems. The paramterer could be changed in the bottom of
 ↪   the
 header file.

 -- Dependencies: this code has no dependencies. If you want to activate the multi-threaded
 ↪   version
 of the code, you only need a C++ compiler that supports OpenMp. OpenMp 3.1 is supported since
 ↪   gcc and g++ 4.7

 -- Compiling the code:
 -> Single Threaded (32 bit): g++ -O3 -std=c++11 admm_sdp.cpp -o admm_sdp -lstdc++
 ↪   -D__NO_INLINE__ -m32
 -> Single Threaded (64 bit): g++ -O3 -std=c++11 admm_sdp.cpp -o admm_sdp -lstdc++
 ↪   -D__NO_INLINE__ -m64

 -> Multi Threaded (32 bit): g++ -O3 -std=c++11 admm_sdp.cpp -o admm_sdp -lstdc++
 ↪   -D__NO_INLINE__ -m32 -fopenmp
 -> Multi Threaded (64 bit): g++ -O3 -std=c++11 admm_sdp.cpp -o admm_sdp -lstdc++
 ↪   -D__NO_INLINE__ -m64 -fopenmp

 Note: some of the flags in the compilation command are redundant, but they are included so you
 ↪   could use both gcc and g++
 without the need to change anything in the command.

 */



#include <stdlib.h>
#include <cmath>
#include <vector>
#include <ctime>
#include <omp.h>
#include <iostream>
#include <fstream>
#include <chrono>
#include <stdint.h>
#include "admm_sdp.h"

#include <time.h>
#include <sys/timeb.h>

// initialization of static members
```

```
bool RandomFuncs::FirstCall = true;

unsigned long long RandomFuncs::x;

////////////////////////////////////////////////////////////////////////////////
// value = S[row][column]
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::Get(uint64_t row, uint64_t column, double &value)
{
        if (row < m_size)
        {
                value = 0;

                uint64_t row_size = m_Values[row].size();
                for(uint64_t i = 0; i < row_size; i++)
                {
                        if (m_Columns[row][i] > column)
                                break;

                        if (m_Columns[row][i] == column)
                        {
                                value = m_Values[row][i];
                                break;
                        }
                }
        }
        else
                value = GetNAN();
}

////////////////////////////////////////////////////////////////////////////////
// value = S[row][column]
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::GetLastElement(uint64_t row, uint64_t column, double &value)
{
        if (row < m_size)
        {
                value = 0;

                if (column <= m_LastNonZeroElement[row])
                {
                        uint64_t row_size = m_Values[row].size();
                        for(uint64_t i = row_size - 1; i >= 0; i--)
                        {
                                if (m_Columns[row][i] < column)
                                        break;

                                if (m_Columns[row][i] == column)
                                {
                                        value = m_Values[row][i];
                                        break;
                                }
                        }
                }
        }
        else
                value = GetNAN();
}

////////////////////////////////////////////////////////////////////////////////
// S[row][column] = value
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::Set(uint64_t row, uint64_t column, double value)
{
        if (row < m_size)
        {
                uint64_t row_size = m_Values[row].size();
                uint64_t i = 0;
                for(; i < row_size; i++)
                {
                        if (m_Columns[row][i] > column)
                        {
                                if (std::abs(value) > DEF_PRESICE)
                                {
                                        std::vector<double> temp_Values(row_size + 1);
                                        std::vector<uint64_t> temp_Columns(row_size + 1);

                                        for (uint64_t j = 0; j < i; j++)
                                        {
                                                temp_Values[j] = m_Values[row][j];
                                                temp_Columns[j] = m_Columns[row][j];
                                        }
```

```

                            temp_Values[i] = value;
                            temp_Columns[i] = column;

                            for (uint64_t j = i + 1; j <= row_size; j++)
                            {
                                temp_Values[j] = m_Values[row][j - 1];
                                temp_Columns[j] = m_Columns[row][j - 1];
                            }

                            m_Values[row].swap(temp_Values);
                            m_Columns[row].swap(temp_Columns);

                            if (m_LastNonZeroElement[row] < column)
                                m_LastNonZeroElement[row] = column;
                        }

                        break;
                    }

                    if (m_Columns[row][i] == column)
                    {
                        if (std::abs(value) > DEF_PRESICE)
                        {
                            m_Values[row][i] = value;

                            if (m_LastNonZeroElement[row] < column)
                                m_LastNonZeroElement[row] = column;
                        }
                        else
                        {
                            std::vector<double> temp_Values(row_size - 1);
                            std::vector<uint64_t> temp_Columns(row_size - 1);

                            for (uint64_t j = 0; j < i; j++)
                            {
                                temp_Values[j] = m_Values[row][j];
                                temp_Columns[j] = m_Columns[row][j];
                            }

                            for (uint64_t j = i; j < row_size - 1; j++)
                            {
                                temp_Values[j] = m_Values[row][j + 1];
                                temp_Columns[j] = m_Columns[row][j + 1];
                            }

                            m_Values[row].swap(temp_Values);
                            m_Columns[row].swap(temp_Columns);


                            if (m_LastNonZeroElement[row] == column)
                            {
                                int64_t i_last = i - 1;
                                for(; i_last >= 0; i_last--)
                                {
                                    if (m_Values[row][i_last] != 0)
                                    {
                                        m_LastNonZeroElement[row] =
                                         ↪   m_Columns[row][i_last];
                                        break;
                                    }
                                }

                                if (i_last < 0)
                                    m_LastNonZeroElement[row] = 0;
                            }
                        }

                        break;
                    }
                }

                if (i == row_size && std::abs(value) > DEF_PRESICE)
                {
                    std::vector<double> temp_Values(row_size + 1);
                    std::vector<uint64_t> temp_Columns(row_size + 1);

                    for (uint64_t j = 0; j < i; j++)
                    {
                        temp_Values[j] = m_Values[row][j];
                        temp_Columns[j] = m_Columns[row][j];
                    }

```

```
                        temp_Values[i] = value;
                        temp_Columns[i] = column;

                        for (uint64_t j = i + 1; j <= row_size; j++)
                        {
                                temp_Values[j] = m_Values[row][j - 1];
                                temp_Columns[j] = m_Columns[row][j - 1];
                        }

                        m_Values[row].swap(temp_Values);
                        m_Columns[row].swap(temp_Columns);

                        m_LastNonZeroElement[row] = column;
                }
        }
}

////////////////////////////////////////////////////////////////////////////////
// S[row][column] = S[row][column] + value
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::Add(uint64_t row, uint64_t column, double value)
{
        if (row < m_size)
        {
                if (std::abs(value) > DEF_PRESICE)
                {
                        uint64_t row_size = m_Values[row].size();
                        uint64_t i = 0;
                        for(; i < row_size; i++)
                        {
                                if (m_Columns[row][i] > column)
                                {
                                        std::vector<double> temp_Values(row_size + 1);
                                        std::vector<uint64_t> temp_Columns(row_size + 1);

                                        for (uint64_t j = 0; j < i; j++)
                                        {
                                                temp_Values[j] = m_Values[row][j];
                                                temp_Columns[j] = m_Columns[row][j];
                                        }

                                        temp_Values[i] = value;
                                        temp_Columns[i] = column;

                                        for (uint64_t j = i + 1; j <= row_size; j++)
                                        {
                                                temp_Values[j] = m_Values[row][j - 1];
                                                temp_Columns[j] = m_Columns[row][j - 1];
                                        }

                                        m_Values[row].swap(temp_Values);
                                        m_Columns[row].swap(temp_Columns);

                                        if (m_LastNonZeroElement[row] < column)
                                                m_LastNonZeroElement[row] = column;

                                        break;
                                }

                                if (m_Columns[row][i] == column)
                                {
                                        double val = m_Values[row][i] + value;
                                        if (std::abs(val) > DEF_PRESICE)
                                        {
                                                m_Values[row][i] = val;

                                                if (m_LastNonZeroElement[row] < column)
                                                        m_LastNonZeroElement[row] = column;
                                        }
                                        else
                                        {
                                                std::vector<double> temp_Values(row_size - 1);
                                                std::vector<uint64_t> temp_Columns(row_size -
                                                 ↪   1);

                                                for (uint64_t j = 0; j < i; j++)
                                                {
                                                        temp_Values[j] = m_Values[row][j];
                                                        temp_Columns[j] = m_Columns[row][j];
                                                }

                                                for (uint64_t j = i; j < row_size - 1; j++)
```

```cpp
                        {
                            temp_Values[j] = m_Values[row][j + 1];
                            temp_Columns[j] = m_Columns[row][j + 1];
                        }

                        m_Values[row].swap(temp_Values);
                        m_Columns[row].swap(temp_Columns);

                        if (m_LastNonZeroElement[row] == column)
                        {
                            int64_t i_last = i - 1;
                            for(; i_last >= 0; i_last--)
                            {
                                if (m_Values[row][i_last] != 0)
                                {
                                    m_LastNonZeroElement[row] = m_Columns[row][i_last];
                                    break;
                                }
                            }

                            if (i_last < 0)
                                m_LastNonZeroElement[row] = 0;
                        }
                    }

                    break;
                }
            }

            if (i == row_size && std::abs(value) > DEF_PRESICE)
            {
                std::vector<double> temp_Values(row_size + 1);
                std::vector<uint64_t> temp_Columns(row_size + 1);

                for (uint64_t j = 0; j < i; j++)
                {
                    temp_Values[j] = m_Values[row][j];
                    temp_Columns[j] = m_Columns[row][j];
                }

                temp_Values[i] = value;
                temp_Columns[i] = column;

                for (uint64_t j = i + 1; j <= row_size; j++)
                {
                    temp_Values[j] = m_Values[row][j - 1];
                    temp_Columns[j] = m_Columns[row][j - 1];
                }

                m_Values[row].swap(temp_Values);
                m_Columns[row].swap(temp_Columns);

                m_LastNonZeroElement[row] = column;
            }
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
// permutation of rows I and J in the matrix
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::SwapRows(uint64_t row_i, uint64_t row_j)
{
    if (row_i < m_size  && row_j < m_size)
    {
        m_Values[row_i].swap(m_Values[row_j]);
        m_Columns[row_i].swap(m_Columns[row_j]);

        uint64_t temp_last = m_LastNonZeroElement[row_i];
        m_LastNonZeroElement[row_i] = m_LastNonZeroElement[row_j];
        m_LastNonZeroElement[row_j] = temp_last;
    }
}

////////////////////////////////////////////////////////////////////////////////
// addition of row I to row SUM and saving the result in the row SUM
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::AddRow(uint64_t row_i, uint64_t row_sum, double alpha)
{
    if (row_i < m_size && row_sum < m_size && alpha != 0)
    {
        uint64_t i_size = m_Values[row_i].size();
        uint64_t sum_size = m_Values[row_sum].size();

        uint64_t temp_size = i_size + sum_size;

        std::vector<double> sum_Values(temp_size);
        std::vector<uint64_t> sum_Columns(temp_size);

        uint64_t i_index = 0;
        uint64_t sum_index = 0;
        uint64_t k = 0;
        while (i_index != i_size || sum_index != sum_size)
        {
            if (sum_index == sum_size)
            {
                sum_Values[k] = alpha * m_Values[row_i][i_index];
                sum_Columns[k] = m_Columns[row_i][i_index];
                i_index++;
                k++;

                continue;
            }
            else if (i_index == i_size)
            {
                sum_Values[k] = m_Values[row_sum][sum_index];
                sum_Columns[k] = m_Columns[row_sum][sum_index];
                sum_index++;
                k++;
            }
            else if (m_Columns[row_i][i_index] < m_Columns[row_sum][sum_index])
            {
                sum_Values[k] = alpha * m_Values[row_i][i_index];
                sum_Columns[k] = m_Columns[row_i][i_index];
                i_index++;
                k++;
            }
            else if (m_Columns[row_i][i_index] > m_Columns[row_sum][sum_index])
            {
                sum_Values[k] = m_Values[row_sum][sum_index];
                sum_Columns[k] = m_Columns[row_sum][sum_index];
                sum_index++;
                k++;
            }
            else
            {
                double val = alpha * m_Values[row_i][i_index] +
                 ↪   m_Values[row_sum][sum_index];
                if (std::abs(val) > DEF_PRESICE)
                {
                        sum_Values[k] = val;
                        sum_Columns[k] = m_Columns[row_i][i_index];
                        k++;
                }

                i_index++;
                sum_index++;
            }
        }

        sum_Values.resize(k);
        sum_Columns.resize(k);

        m_Values[row_sum].swap(sum_Values);
        m_Columns[row_sum].swap(sum_Columns);

        int64_t i_last = k - 1;
        for(; i_last >= 0; i_last--)
        {
            if (m_Values[row_sum][i_last] != 0)
            {
                m_LastNonZeroElement[row_sum] = m_Columns[row_sum][i_last];

                break;
            }
        }

        if (i_last == -1)
            m_LastNonZeroElement[row_sum] = 0;
    }
}

```

```
////////////////////////////////////////////////////////////////////////////////
// product of two rows like two vectors, the sum of the pairwise products of the elements
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::RowsProduct(uint64_t row_i, uint64_t row_j, double & prod)
{
        if (row_i < m_size && row_j < m_size)
        {
                prod = 0;

                uint64_t i_index = 0;
                uint64_t j_index = 0;

                uint64_t i_size = m_Values[row_i].size();
                uint64_t j_size = m_Values[row_j].size();

                while (i_index != i_size && j_index != j_size)
                {
                        if (m_Columns[row_i][i_index] < m_Columns[row_j][j_index])
                                i_index++;
                        else if (m_Columns[row_i][i_index] == m_Columns[row_j][j_index])
                        {
                                prod += m_Values[row_i][i_index] * m_Values[row_j][j_index];
                                i_index++;
                                j_index++;
                        }
                        else
                                j_index++;
                }
        }
}

////////////////////////////////////////////////////////////////////////////////
// product of row and vector like two vectors, the sum of the pairwise products of the elements
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::RowVectorProduct(const std::vector<double> &x, uint64_t row, double & prod)
{
        if (row < m_size)
        {
                prod = 0;

                uint64_t i_index = 0;
                uint64_t i_size = m_Values[row].size();
                uint64_t x_size = x.size();

                while (i_index != i_size)
                {
                        if (m_Columns[row][i_index] >= x_size)
                        {
                                prod = GetNAN();
                                return;
                        }

                        prod += x[m_Columns[row][i_index]] * m_Values[row][i_index];
                        i_index++;
                }
        }
}

////////////////////////////////////////////////////////////////////////////////
//  filling the sparse matrix row values
////////////////////////////////////////////////////////////////////////////////
void SparseMatrix::PushRow(uint64_t row, const std::vector<double> &values, const
 ↪  std::vector<uint64_t> &columns, uint64_t count)
{
        if (row < m_size && values.size() >= count  && columns.size() >= count)
        {
                m_Values[row].resize(count);
                m_Columns[row].resize(count);

                for (uint64_t i = 0; i < count; i++)
                {
                        m_Values[row][i] = values[i];
                        m_Columns[row][i] = columns[i];
                }

                int64_t i_last = count - 1;
                for(; i_last >= 0; i_last--)
                {
                        if (m_Values[row][i_last] != 0)
                        {
                                m_LastNonZeroElement[row] = m_Columns[row][i_last];

                                break;
```

```cpp
                        }
                }

                if (i_last == -1)
                        m_LastNonZeroElement[row] = 0;
        }
}

////////////////////////////////////////////////////////////////////////////////
// Computes eigenvectors and eigenvalues of a symmetric matrix
////////////////////////////////////////////////////////////////////////////////
MatrixFuncs::ResultCode MatrixFuncs::EigenVectorsSymm(const std::vector<double> &a, std::vector<double> &eigen_values, std::vector<double> &eigen_vectors)
{
        MatrixFuncs::ResultCode result_code = ercNoError;

        int64_t dim = (int64_t)sqrt(a.size());

        if (dim * dim != (int64_t)a.size() || dim == 0)
                return ercInputDataError;

        double tolerance = DEF_TOLERANCE;

    // allocation for a vector of eigenvalues and a matrix of eigenvectors
        eigen_values.resize(dim);
        eigen_vectors.resize(dim * dim);

        // calculating Hessenberg form of A
        std::vector<double> d;
        std::vector<double> e;
        HessenbergFormSymm(a, eigen_vectors, d, e);

        // computing the norm of H
        double norm = 0;
        for (int64_t i = 0; i < dim; i++)
                norm += std::abs(d[i]);
        for (int64_t i = 0; i < dim - 1; i++)
                norm += 2 * std::abs(e[i]);

        // finding the index of the first non-zero subdiagonal element
        int64_t min_index;
        for (int64_t i = 0; i < dim; i++)
        {
                if (i == dim - 1)
                {
                        min_index = i;
                        break;
                }

                double sum = std::abs(d[i]) + std::abs(d[i + 1]);
                if (sum == 0)
                        sum = norm;
                if ((std::abs(e[i]) <= tolerance * sum) && (std::abs(e[i]) <= tolerance))
                        e[i] = 0;
                else
                {
                        min_index = i;
            break;
                }
        }

        // finding the index of the first zero element e[i] starting from min_index
        int64_t max_index;
        for (max_index = min_index + 1; max_index < dim; max_index++)
        {
                if (max_index == dim - 1)
                        break;
                double sum = std::abs(d[max_index]) + std::abs(d[max_index + 1]);
                if (sum == 0)
                        sum = norm;
                if ((std::abs(e[max_index]) < tolerance * sum) && (std::abs(e[max_index]) < tolerance))
                {
                        e[max_index] = 0;
                        break;
                }
        }

        int64_t count = 0;

        // we now proceed with an iterative algorithm. On each step we are making e[i] closer to zero for i = min_index
        // and recalculating max_index and min_index
        while ((min_index < dim - 1) && (count < 10000))
        {
                // performing a step of the QR-algorithm with shifts for the block
                 ↪  [min_index,max_index] of H;
                // for that we compute H = P_k*...*P_1*H*P'_1*...*P'_k, k = max_index -
                 ↪  min_index,
                // each P_i is a plane rotation making a subdiagonal element of H - shift*I
                 ↪  zero

                // the shift is the eigenvalue of an upper left block 2x2 closer to the corner
                 ↪  element d[min_index]
                double g = (d[min_index+1] - d[min_index]) / (2.0 * e[min_index]);
                if (g >= 0)
                        g -= sqrt(g * g + 1);
                else
                        g += sqrt(g * g + 1);

                double shift = d[min_index] + e[min_index] * g;

                g = d[max_index] - shift;

                // performing max_index - min_index plane rotations on the block
                 ↪  [min_index,max_index];
                // this is an implicit computation, done in a way to work faster
                bool zero = false;
                double s = 1, c = 1, p = 0;
                for (int64_t i = max_index - 1; i >= min_index; i--)
                {
                  double f = s * e[i];
                  double b = c * e[i];
                  double r = sqrt(f * f + g * g);
                  e[i + 1] = r;

                  if (r == 0)
                        {
                            // in case zero appeared on the subdiagonal of the block
                             ↪  [min_index,max_index]
                        e[i + 1] = 0;
                        d[i + 1] -= p;
                        zero = true;
                        break;
                        }

                        s = f / r;
                  c = g / r;
                  g = d[i + 1] - p;
                  r = (d[i] - g) * s + 2.0 * c * b;
                  p = s * r;
                  d[i + 1] = g + p;
                  g = c * r - b;

                        // modification of S (which is being saved as eigenVectors)
                        for (int64_t j = 0; j < dim; j++)
                        {
                            f = eigen_vectors[j * dim + i + 1];
                            eigen_vectors[j * dim + i + 1] = s * eigen_vectors[j * dim + i]
                             ↪  + c * f;
                            eigen_vectors[j * dim + i] = c * eigen_vectors[j * dim + i] - s
                             ↪  * f;
                        }
                }

                e[max_index] = 0;
                if (!zero)
                {
                        d[min_index] -= p;
                        e[min_index] = g;
                }

                count++;

                // recalculation of min_index
                for (int64_t i = min_index; i < dim; i++)
                {
                        if (i == dim - 1)
                        {
                            min_index = i;
                            break;
                        }

                        double sum = std::abs(d[i]) + std::abs(d[i + 1]);
                        if (sum == 0)
```

```cpp
                                sum = norm;
                if ((std::abs(e[i]) < tolerance * sum) && (std::abs(e[i]) < tolerance))
                        {
                                e[i] = 0;
                                count = 0;
                        } else
                        {
                                min_index = i;
                                break;
                        }
                }

                // recalculation of max_index
                int64_t indx;
                for (indx = min_index + 1; indx < dim; indx++)
                {
                        if (indx == dim - 1)
                                break;
                        double sum = std::abs(d[indx]) + std::abs(d[indx + 1]);
                        if (sum == 0)
                                sum = norm;
                        if ((std::abs(e[indx]) < tolerance * sum) && (std::abs(e[indx]) < tolerance))
                        {
                                e[indx] = 0;
                                break;
                        }
                }

                if (indx < max_index || min_index >= max_index)
                {
                        max_index = indx;
                        count = 0;
                }
        }

        // eigenvalues of A are the diagonal elements
        for (int64_t i = 0; i < dim; i++)
                eigen_values[i] = d[i];

        return result_code;
}

////////////////////////////////////////////////////////////////////////////////
// Computes the Hessenberg (tridiagonal in this case) form of a symmetric matrix A
//H = SAS', where H is an upper Hessenberg matrix, S - ortogonal matrix and S' is S transposed
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::HessenbergFormSymm(const std::vector<double> &a, std::vector<double> &s, std::vector<double> &d, std::vector<double> &e)
{
        int64_t dim = (int64_t)sqrt(a.size());

        // memory allocation
        s.resize(dim * dim);
        d.resize(dim);
        e.resize(dim);

        std::vector<double> H(a);
        std::vector<double> v(dim);
        std::vector<double> h(dim);

        bool first_modification = true;
        // algorithm based on Householder transformations
        for (int64_t i = 0; i < dim - 2; i++)
        {
        double t = 0;
        for (int64_t j = i + 1; j < dim; j++)
            t += H[j * dim + i] * H[j * dim + i];

                double u = sqrt(t);

                // if all the elements of the i^th column starting from i+2 are zeroes, then we save the diagonal
                // and subdiagonal elements and go to the next iteration
                if (u <= std::abs(H[(i + 1) * dim + i]))
                {
                        d[i] = H[i * dim + i];
                        e[i] = H[(i + 1) * dim + i];
            continue;
                }

        if (H[(i + 1) * dim + i] > 0)
            u *= -1;
```

```

    double w = sqrt(u * u - H[(i + 1) * dim + i] * u);

    v[i + 1] = (H[(i + 1) * dim + i] - u) / w;
    for (int64_t j = i + 2; j < dim; j++)
      v[j] = H[j * dim + i] / w;

        // at this iteration, we compute H -> P*H*P, where P = I - v*v'
        // P*H*P = H - H*v*v' - v*v'*H + v*v'*H*v*v' =
        // = H - h*v' - v*h' + mult*v*v' = H - h*v' - (h*v')' + mult*v*v',
        // where the vector h = H*v and the number mult = h'*v are computed below

        // the vector h has zeros at the first (i-1) coordinates, and the i^th
         ↪   coordinate is irrelevant
//       #pragma omp parallel for schedule(guided)        //SECOND
        for (int64_t j = i + 1; j < dim; j++)
        {
            h[j] = 0;
            // we use only the elements of H below the main diagonal
            for (int64_t k = i + 1; k < dim; k++)
                if (k <= j)
                    h[j] += H[j * dim + k] * v[k];
                else
                    h[j] += H[k * dim + j] * v[k];
        }

        double mult = 0;
        for (int64_t j = i + 1; j < dim; j++)
            mult += h[j] * v[j];

        // final computation of H;
        // we save the next diagonal and subdiagonal elements and compute only the
         ↪   columns starting from i+1
        d[i] = H[i * dim + i];
        e[i] = u;

//       #pragma omp parallel for schedule(guided)        //SECOND
        for (int64_t j = i + 1; j < dim; j++)
            for (int64_t k = i + 1; k <= j; k++)
                H[j * dim + k] += (-v[k] * h[j] - v[j] * h[k] + mult * v[j] *
                 ↪   v[k]);

        if (first_modification)
        {
            // at the first modification, S is initialized by P = I - v*v'
            for (int64_t k = 0; k < dim; k++)
                for (int64_t j = 0; j < dim; j++)
                    if ((j > i) && (k > i))
                        if (j == k)
                            s[k * dim + j] = 1 - v[j] * v[k];
                        else
                            s[k * dim + j] = -v[j] * v[k];
                    else
                        if (j == k)
                            s[k * dim + j] = 1;
                        else
                            s[k * dim + j] = 0;
            first_modification = false;
        }
        else
        {
            // computation of S = P*S = S - v*v'*S = S - v*h', where h' = v'*S
//           #pragma omp parallel for schedule(guided)        //SECOND
            for (int64_t j = 1; j < dim; j++)
            {
                h[j] = 0;
                for (int64_t k = i + 1; k < dim; k++)
                    h[j] += s[j * dim + k] * v[k];
            }
//           #pragma omp parallel for schedule(guided)        //SECOND
            for (int64_t j = i + 1; j < dim; j++)
                for (int64_t k = 1; k < dim; k++)
                    s[k * dim + j] -= (v[j] * h[k]);
        }
    }

    // in case the matrix A was already in the Hessenberg form, we initialize S by identity
    if (first_modification)
        for (int64_t k = 0; k < dim; k++)
            for (int64_t j = 0; j < dim; j++)
                if (j == k)
                    s[k * dim + j] = 1;
```

```
                                else
                                        s[k * dim + j] = 0;

        d[dim - 2] = H[(dim - 2) * dim + dim - 2];
        d[dim - 1] = H[(dim - 1) * dim + dim - 1];
        e[dim - 2] = H[(dim - 1) * dim + dim - 2];

        return;
}


////////////////////////////////////////////////////////////////////////////////
// Multiplication of real matrices written in a 1-dim array row-wise
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::Multiply(        const int64_t &m, const int64_t &dim, const int64_t &n, const std::vector<double> &a, const std::vector<double> &b, bool left_trans, bool right_trans,
                                                        const double &alpha, std::vector<double> &mult)
{
        if ((int64_t)mult.size() != m * n)
                mult.resize(m * n);

        // computation of mult
        if(left_trans && right_trans)
        {
                // Both matrices are transposed
//                #pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t i = 0; i < m; i++)
                        for (int64_t j = 0; j < n; j++)
                        {
                                double sum = 0;
                                int64_t ind = i - m;
                                for ( int64_t k = 0; k < dim; k++)
                                        sum += a[ind += m] * b[j * dim + k];
                                mult[i * n + j] = alpha * sum;
                        }
        }else if(left_trans)
        {
                // First matrix is transposed
//                #pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t i = 0; i < m; i++)
                        for (int64_t j = 0; j < n; j++)
                        {
                                double sum = 0;
                                int64_t ind1 = i - m, ind2 = j - n;
                                for (int64_t k = 0; k < dim; k++)
                                        sum += a[ind1 += m] * b[ind2 += n];
                                mult[i * n + j] = alpha * sum;
                        }
        }else if(right_trans)
        {
                // Second matrix is transposed
//                #pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t i = 0; i < m; i++)
                        for (int64_t j = 0; j < n; j++)
                        {
                                double sum = 0;
                                for (int64_t k = 0; k < dim; k++)
                                        sum += a[i * dim + k] * b[j * dim + k];
                                mult[i * n + j] = alpha * sum;
                        }
        }else
        {
                // Matrices are not transposed
//                #pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t i = 0; i < m; i++)
                        for (int64_t j = 0; j < n; j++)
                        {
                                double sum = 0;
                                int64_t ind = j - n;
                                for (int64_t k = 0; k < dim; k++)
                                        sum += a[i * dim + k] * b[ind += n];
                                mult[i * n + j] = alpha * sum;
                        }
        }

        return;
}

////////////////////////////////////////////////////////////////////////////////
// Multiplication of sparse real matrices
```

```
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::MultiplySparse( const int64_t &m, const int64_t &dim, SparseMatrix &a,
 ↪   SparseMatrix &mult)
{
        mult.clear();

        if ((int64_t)a.size() < m)
                return;

        mult.resize(m);

        std::vector<double> temp_values(m);
        std::vector<uint64_t> temp_columns(m);

        // Second matrix is transposed
#pragma omp parallel for schedule(guided) // StarGraph
        for (int64_t i = 0; i < m; i++)
        {
                int64_t count = 0;
                for (int64_t j = 0; j < m; j++)
                {
                        double sum = 0;

                        a.RowsProduct(i, j, sum);

                        if (std::abs(sum) > DEF_PRESICE)
                        {
                                temp_values[count] = sum;
                                temp_columns[count] = j;

                                count++;
                        }
                }

                mult.PushRow(i, temp_values, temp_columns, count);
        }

        return;
}

////////////////////////////////////////////////////////////////////////////////
// Solves the system of linear equations A*x = B
////////////////////////////////////////////////////////////////////////////////
MatrixFuncs::ResultCode MatrixFuncs::DevideByVectorAnaliticSymm(        const
 ↪   std::vector<double> &a, const std::vector<double> &b, std::vector<double> &x)
{
        MatrixFuncs::ResultCode result_code = ercNoError;

        if (a.size() == 0)
                return ercInputDataError;

        int64_t dim = (int64_t)sqrt(a.size());

        if (dim * dim != (int64_t)a.size())
                return ercInputDataError;

        if (dim != (int64_t)b.size())
                return ercInputDataError;

        x.resize(dim);

        std::vector<double> L(a);

        // computation of L

        for (int64_t i = 0; i < dim; i++)
        {
                for (int64_t j = 0; j < i; j++)
                {
                        double sum2 = 0;
//#pragma omp parallel for schedule(guided)        //SECOND
                        for (int64_t k = 0; k < j; k++)
                                sum2 += L[i * dim + k] * L[j * dim + k];

                        L[i * dim + j] = (L[i * dim + j] - sum2) / L[j * dim + j];
                }

                double sum1 = 0;
//#pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t k = 0; k < i; k++)
                        sum1 += L[i * dim + k] * L[i * dim + k];

                if (L[i * dim + i] - sum1 <= 0)
                {
                        for (int64_t j = 0; j < dim; j++)
                                x[j] = GetNAN();

                        return ercInputDataError; //A must be a positive definite matrix
                }
                else
                        L[i * dim + i] = sqrt(L[i * dim + i] - sum1);
        }

//#pragma omp parallel for schedule(guided)        //SECOND
        for (int64_t i = 0; i < dim; i++)
        {
                x[i] = b[i];
        }

        for (int64_t i = 0; i < dim; i++)
        {
                x[i] /= L[i * dim + i];

//#pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t j = i + 1; j < dim; j++)
                {
                        x[j] -= L[j * dim + i] * x[i];
                }
        }

        for (int64_t i = dim - 1; i >= 0; i--)
        {
                x[i] /= L[i * dim + i];

//#pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t j = 0; j < i; j++)
                {
                        x[j] -= L[i * dim + j] * x[i];
                }
        }

        return result_code;
}

////////////////////////////////////////////////////////////////////////////////
// Solves the system of linear equations A*x = b for symmetric positive definite matrix A by
 ↪  using Gauss method(analitical method).
// The matrices A  and vector b must have the same number of rows.
// Algorithm is divided into two phases
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// Fase_1(preliminary calculations) - reduction matrix A to the lower triangular matrices
////////////////////////////////////////////////////////////////////////////////
MatrixFuncs::ResultCode
 ↪  MatrixFuncs::DevideByVectorAnaliticSymmSparse_Fase_1(        SparseMatrix &a_triang,
 ↪  SparseMatrix &s)
{
        MatrixFuncs::ResultCode result_code = ercNoError;

        int64_t dim = a_triang.size();

        if (dim == 0)
                return ercInputDataError;

        s.resize(dim);

        //reduction of the matrix A to a triangular form

        for (int64_t i = dim - 1; i >= 0; i--)
        {
                double val_ii;
                a_triang.Get(i, i, val_ii);

                // modification of A and S
#pragma omp parallel for schedule(guided) // StarGraph
                for (int64_t j = i - 1; j >= 0; j--)
                {
                        double val_ji;
                        a_triang.GetLastElement(j, i, val_ji);

                        if (val_ji != 0.0)
                        {
                                double temp = val_ji / val_ii;

                                s.Set(j, i, -temp);

                                a_triang.AddRow(i, j, -temp);
                        }
                }
        }

        return result_code;
}

////////////////////////////////////////////////////////////////////////////////
// Solves the system of linear equations A*x = b for symmetric positive definite matrix A by
 ↪  using Gauss method(analitical method).
// The matrices A  and vector b must have the same number of rows.
// Algorithm is divided into two phases
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// Fase_2 - transformation of vector b (using transformation matrix S) and sequential
 ↪  computation of the vector x
////////////////////////////////////////////////////////////////////////////////
MatrixFuncs::ResultCode
 ↪  MatrixFuncs::DevideByVectorAnaliticSymmSparse_Fase_2(        SparseMatrix &a_triang,
 ↪  SparseMatrix &s, const std::vector<double> &b, std::vector<double> &x)
{
        MatrixFuncs::ResultCode result_code = ercNoError;

        int64_t dim = a_triang.size();

        if (dim == 0)
                return ercInputDataError;

        if (dim != s.size())
        {
                std::cout << "return" << std::endl;
                return ercInputDataError;
        }

        if (dim != (int64_t)b.size())
        {
                std::cout << "return" << std::endl;
                return ercInputDataError;
        }

        x.resize(dim);

        std::vector<double> temp_b(b);

        // modification of B
        for (int64_t i = dim - 1; i >= 0; i--)
        {
                double temp = 0;
                s.RowVectorProduct(temp_b, i, temp);
                temp_b[i] += temp;
        }

        // recursive computation of the vector x
        for (int64_t i = 0; i < dim; i++)
        {
                double sum_sq = 0.0;
                a_triang.RowVectorProduct(x, i, sum_sq);

                double val_ii;
                a_triang.Get(i, i, val_ii);

                x[i] = (temp_b[i] - sum_sq) / val_ii;
        }

        return result_code;
}

////////////////////////////////////////////////////////////////////////////////
// Addition of real vectors
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::AddVectors(const std::vector<double> &v_1, const std::vector<double> &v_2,
 ↪  const double &alpha, const double &beta, std::vector<double> &sum)
{
        int64_t dim = v_1.size();
        if ((int64_t)sum.size() != dim)
                sum.resize(dim);

        for (int64_t i = 0; i < dim; i++)
                sum[i] = alpha * v_1[i] + beta * v_2[i];

        return;
```

```
}

////////////////////////////////////////////////////////////////////////////////
// Computes the inverse of matrix a, matrices are written in a 1-dim array row-wise
////////////////////////////////////////////////////////////////////////////////
MatrixFuncs::ResultCode MatrixFuncs::Inverse(const std::vector<double> &a, std::vector<double>
 ↪   &a_inv)
{
        MatrixFuncs::ResultCode result_code = ercNoError;

        if (a.size() == 0)
                return ercInputDataError;

        int64_t dim = (int64_t)sqrt(a.size());

        if (dim * dim != (int64_t)a.size())
                return ercInputDataError;

        a_inv.resize(dim * dim);

        std::vector<double> lu(a);
        std::vector<double> lu_mod(dim*dim);
        std::vector<int64_t> permutation(dim);

        // LU decomposition
        for (int64_t i = 0; i < dim; i++)
        {
                // finding pivot: the row's number of the maximal element among A[i][i],
                 ↪   A[i+1][i], ..., A[n-1][i]
                double permutation_value = 0;
                long permutation_indx = -1;

                for (int64_t k = i; k < dim; k++)
                        if (std::abs(lu[k * dim + i]) - permutation_value > 0)
                        {
                                permutation_value = std::abs(lu[k * dim + i]);
                                permutation_indx = k;
                        }

                if (std::abs(permutation_value) < DEF_TOLERANCE)
                {
                        // error in case matrix a is singular (will be treated as warning
                         ↪   unless pivotValue = 0)
                        result_code = ercSingularMatrixWarning;

                        if (!permutation_value)
                        {
                                for (int64_t j = 0; j < dim * dim; j++)
                                        a_inv[j] = GetNAN();
                                return ercSingularMatrixError;
                        }
                }

                if (i != permutation_indx)
                {
                        permutation[i] = permutation_indx;
                        // switching i and pivot rows in A:
                        for (int64_t j = 0; j < dim; j++)
                        {
                                double temp = lu[i * dim + j];
                                lu[i * dim + j] = lu[permutation_indx * dim + j];
                                lu[permutation_indx * dim + j] = temp;
                        }
                }
                else
                        permutation[i] = -1;

                // modification of A
//#pragma omp parallel for schedule(guided)        //SECOND
                for (int64_t j = i + 1; j < dim; j++)
                {
                        double temp = (lu[j * dim + i] /= lu[i * dim + i]);

                        for (int64_t k = i + 1; k < dim; k++)
                                lu[j * dim + k] -= temp * lu[i * dim + k];
                }
        }

        // recursive computation of the inverse matrix of L (the lower half of LU)
//#pragma omp parallel for schedule(guided)        //SECOND
        for (int64_t i = 0; i < dim - 1; i++)
                for (int64_t j = i + 1; j < dim; j++)
```

```
		{
			double temp = lu[j * dim + i] * -1.0;

			for (int64_t k = i + 1; k < j; k++)
				temp -= lu[j * dim + k] * lu_mod[i * dim + k];

			lu_mod[i * dim + j] = temp;
		}

	// recursive computation of the inverse matrix of U (the upper half of LU)
//#pragma omp parallel for schedule(guided)	//SECOND
	for (int64_t i = dim - 1; i >= 0; i--)
	{
		lu_mod[i * dim + i] = 1.0 / lu[i * dim + i];

		for (int64_t j = i - 1; j >= 0; j--)
		{
			double temp = 0;

			for (int64_t k = i; k > j; k--)
				temp -= lu[j * dim + k] * lu_mod[i * dim + k];

			lu_mod[i * dim + j] = temp / lu[j * dim + j];
		}
	}

	// computation of inv(U)*inv(L)
//#pragma omp parallel for schedule(guided)	//SECOND
	for (int64_t i = 0; i < dim; i++)
	{
		for (int64_t j = 0; j < i; j++)
		{
			double temp = 0;

			for (int64_t k = i; k < dim; k++)
				temp += lu_mod[k * dim + i] * lu_mod[j * dim + k];

			a_inv[i * dim + j] = temp;
		}

		for (int64_t j = i; j < dim; j++)
		{
			double temp = lu_mod[j * dim + i];

			for (int64_t k = j + 1; k < dim; k++)
				temp += lu_mod[k * dim + i] * lu_mod[j * dim + k];

			a_inv[i * dim + j] = temp;
		}
	}

	// computation of the final result
	for (int64_t i = dim - 1; i >= 0; i--)
		if (permutation[i] != -1)
			for (int64_t j = 0; j < dim; j++)
			{
				double temp = a_inv[j * dim + i];
				a_inv[j * dim + i] = a_inv[j * dim + permutation[i]];
				a_inv[j * dim + permutation[i]] = temp;
			}

	return result_code;
}

////////////////////////////////////////////////////////////////////////////////
//Restoring symmetric matrix from lower triangular part
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::SymmMatrixFromLowerMatrix(	const int64_t &m, const std::vector<double>
 ↪  &a_lower, std::vector<double> &a)
{
	uint64_t lower_dim = (int64_t)(0.5 * m * (m + 1));
	if (a_lower.size() != lower_dim)
	{
		a.clear();
		return;
	}

	a.resize(m * m);
	for (int64_t i = 0, k = 0; i < m; i++)
		for (int64_t j = 0; j <= i; j++, k++)
			a[i * m + j] = a_lower[k];

	for (int64_t i = 0, k = 0; i < m; i++, k++)
```

```
		for (int64_t j = 0; j < i; j++, k++)
			a[j * m + i] = a_lower[k];

	return;
}

////////////////////////////////////////////////////////////////////////////////
//Recording lower triangular part of symmetric matrix
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::LowerMatrix(	const int64_t &m, const std::vector<double> &a,
 ↪	std::vector<double> &a_lower)
{
	if ((int64_t)a.size() != m * m)
	{
		a_lower.clear();
		return;
	}

	uint64_t lower_dim = (int64_t)(0.5 * m * (m + 1));

	a_lower.resize(lower_dim);
	for (int64_t i = 0, k = 0; i < m; i++)
		for (int64_t j = 0; j <= i; j++, k++)
			a_lower[k] = a[i * m + j];

	return;
}

////////////////////////////////////////////////////////////////////////////////
//Frobenius matrix norm calculation using lower triangular part of symmetric matrix
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::FrobeniusNormSymmLower(const int64_t &m, const std::vector<double> &a_lower,
 ↪	double &norm)
{
	uint64_t lower_dim = (int64_t)(0.5 * m * (m + 1));
	if (a_lower.size() != lower_dim)
	{
		norm = GetNAN();
		return;
	}

	norm = 0;
	for (int64_t i = 0, count = 0; i < m; i++, count++)
	{
		for (int64_t j = 0; j < i; j++, count++)
			norm += 2 * a_lower[count] * a_lower[count];

		norm += a_lower[count] * a_lower[count];
	}

	norm = sqrt(norm);

	return;
}

////////////////////////////////////////////////////////////////////////////////
//P-norm calculation using lower triangular part of symmetric matrix
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::PNormSymmLower(const int64_t &m, const int64_t &p, const std::vector<double>
 ↪	&a_lower, double &norm)
{
	uint64_t lower_dim = (int64_t)(0.5 * m * (m + 1));
	if (a_lower.size() != lower_dim || m == 0)
	{
		norm = GetNAN();
		return;
	}

	norm = 0;

	std::vector<double> sum(m, 0);
	if (p == 1)
	{
		for (int64_t i = m - 1, count = lower_dim - 1; i >= 0; i--)
		{
			sum[i] += std::abs(a_lower[count]);
			count--;
			for (int64_t j = i - 1; j >= 0; j--, count--)
			{
				sum[i] += std::abs(a_lower[count]);
				sum[j] += std::abs(a_lower[count]);
			}
		}
```

```
                double max = sum[0];
                for (int64_t i = 1; i < m; i++)
                        if (sum[i] > max)
                                max = sum[i];

                norm = max;
        }
        else
        {
                for (int64_t i = m - 1, count = lower_dim - 1; i >= 0; i--)
                {
                        sum[i] += pow(std::abs(a_lower[count]), p);
                        count--;
                        for (int64_t j = i - 1; j >= 0; j--, count--)
                        {
                                sum[i] += pow(std::abs(a_lower[count]), p);
                                sum[j] += pow(std::abs(a_lower[count]), p);
                        }
                }
                double max = sum[0];
                for (int64_t i = 1; i < m; i++)
                        if (sum[i] > max)
                                max = sum[i];

                norm = pow(max, 1.0 / p);
        }

        return;
}

////////////////////////////////////////////////////////////////////////////////
//P-norm calculation for vector
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::PNormVector(const int64_t &p, const std::vector<double> &v, double &norm)
{
        int64_t dim = v.size();
        if (dim == 0)
        {
                norm = GetNAN();
                return;
        }

        norm = 0;
        for (int64_t i = 0; i < dim; i++)
        {
                norm += pow(v[i], p);
        }

        norm = pow(norm, 1.0 / p);

        return;
}

////////////////////////////////////////////////////////////////////////////////
// Multiplication of real vectors
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::MultiplyVectors( const std::vector<double> &v_1, const std::vector<double>
 ↪   &v_2, const double &alpha, double &mult)
{
        int64_t dim = v_1.size();
        if ((int64_t)v_2.size() != dim)
        {
                mult = GetNAN();
                return;
        }

        mult = 0;
        for (int64_t i = 0; i < dim; i++)
                mult += alpha * v_1[i] * v_2[i];

        return;
}

////////////////////////////////////////////////////////////////////////////////
// Multiplication of integer and real vectors
////////////////////////////////////////////////////////////////////////////////
void MatrixFuncs::MultiplyVectors( const std::vector<int64_t> &v_1, const std::vector<double>
 ↪   &v_2, const double &alpha, double &mult)
{
        int64_t dim = v_1.size();
        if ((int64_t)v_2.size() != dim)
        {
                mult = GetNAN();
                return;
        }

        mult = 0;
        for (int64_t i = 0; i < dim; i++)
                mult += alpha * v_1[i] * v_2[i];

        return;
}

////////////////////////////////////////////////////////////////////////////////
// Finding the maximum element in the vector
////////////////////////////////////////////////////////////////////////////////
double MatrixFuncs::Max( const std::vector<double> &a)
{
        int64_t dim = a.size();
        if (dim == 0)
                return GetNAN();

        double max = a[0];
        for (int64_t i = 1; i < dim; i++)
                if (a[i] > max)
                        max = a[i];

        return max;
}

////////////////////////////////////////////////////////////////////////////////
// Calculation of the sum of vector elements
////////////////////////////////////////////////////////////////////////////////
double MatrixFuncs::Sum( const std::vector<double> &a)
{
        int64_t dim = a.size();
        if (dim == 0)
                return GetNAN();

        double sum = a[0];
        for (int64_t i = 1; i < dim; i++)
                sum += a[i];

        return sum;
}

///////////////////////////////////////////////////////////////////////////////
// Generate integer random matrix
///////////////////////////////////////////////////////////////////////////////
RandomFuncs::ResultCode RandomFuncs::MatrixI(int64_t n, int64_t m, std::vector <int64_t>
 ↪  &rand_m, uint64_t min, uint64_t max, bool rand_init, int64_t mult)
{
        if (n <= 0 || m <= 0)
        {
                return ercDimensionError; // error dimension
        }

        rand_m.resize(n * m);

        if (FirstCall)
        {
                InitSeed(rand_init);
                FirstCall = false;
        }

        int64_t d = max - min + 1;
        if (d != 1)
                for (int64_t i = 0; i < n * m; i++)
                        rand_m[i] = (NextInt() % d + min) * mult;
        else
                for (int64_t i = 0; i < n * m; i++)
                                rand_m[i] = min * mult;
        return ercNoError;
}

///////////////////////////////////////////////////////////////////////////////
// Generate real random matrix
///////////////////////////////////////////////////////////////////////////////
RandomFuncs::ResultCode RandomFuncs::Matrix(int64_t n, int64_t m, std::vector <double> &rand_m,
 ↪  uint64_t min, uint64_t max, bool rand_init, double max_add, double mult)
{
        if (n <= 0 || m <= 0)
        {
                return ercDimensionError; // error dimension
        }

	rand_m.resize(n * m);

	if (FirstCall)
	{
		InitSeed(rand_init);
		FirstCall = false;
	}

	int64_t d = max - min + 1;

	double add = 0;

	if (d != 1)
		for (int64_t i = 0; i < n * m; i++)
		{
			if (std::abs(max_add) > 0)
			{
				add = NextDouble() * max_add;
			}
			rand_m[i] = (NextInt() % d + min + add) * mult;
		}
	else
		for (int64_t i = 0; i < n * m; i++)
			rand_m[i] = (min + add) * mult;

	return ercNoError;
}

RandomFuncs::ResultCode RandomFuncs::SparseSymmetricMatrixZeroDiagonalB(const int64_t n, const
 ↪  double density, std::vector <bool> &rand_m, bool rand_init)
{
	if (n <= 0)
	{
		return ercDimensionError; // error dimension
	}

	if (density < 0 || density > 1)
	{
		return ercDensityError; // error density
	}

	if (FirstCall)
	{
		InitSeed(rand_init);
		FirstCall = false;
	}

	rand_m.resize(n * n);
	for (int64_t i = 0; i < n * n; i++)
		rand_m[i] = 0;

	int64_t max_nonzero_count = (int64_t)((n * (n - 1)) * density); // zero diagonal
	max_nonzero_count -= max_nonzero_count%2;
	int64_t nonzero_count = 0;
	int64_t IJ_count = (int64_t)(0.5 * n * (n - 1));
	std::vector <std::pair<int64_t, int64_t> > IJ(IJ_count);
	for (int64_t i = 0, k = 0; i < n; i++)
		for (int64_t j = i + 1; j < n; j++, k++)
			IJ[k] = std::pair<int64_t, int64_t>(i, j);

	while (nonzero_count < max_nonzero_count)
	{
		int64_t indx = NextInt() % IJ_count;

		int64_t i = IJ[indx].first;
		int64_t j = IJ[indx].second;

		rand_m[i * n + j] = 1;
		rand_m[j * n + i] = 1;

		IJ.erase(IJ.begin() + indx);

		IJ_count -= 1;
		nonzero_count += 2;
	}

	return ercNoError;
}

double RandomFuncs::NextDouble()
{
	x = a * x + c;
	x = x % m;

	return (double)x / (m - 1);
}

unsigned long RandomFuncs::NextInt()
{
	x = a * x + c;
	x = x % m;

	return x;
}

void RandomFuncs::InitSeed(bool rand_init)
{
	// initialization of the seed
	if (rand_init)
	{
		x = time(NULL);
		x = x % m;
	}
	else
	{
		x = 5;
	}

	FirstCall = false;
}

////////////////////////////////////////////////////////////////////////////////
// Generate boolean banded matrix
////////////////////////////////////////////////////////////////////////////////
AdjacencyMatrix::ResultCode AdjacencyMatrix::CreateBandedGraph(std::vector <bool>
 ↪  &AdjacencyMatrix, const int64_t n)
{
	if (n <= 0)
	{
		return ercDimensionError; // error dimension
	}

	AdjacencyMatrix.resize(n * n);
	for (int64_t i = 0; i < n * n; i++)
		AdjacencyMatrix[i] = 0;

	for (int64_t i = 0; i < n - 1; i++)
	{
		AdjacencyMatrix[i * n + i + 1] = 1;
		AdjacencyMatrix[(i + 1) * n + i] = 1;
	}

	return ercNoError;
}


////////////////////////////////////////////////////////////////////////////////
// Generate boolean sparse random matrix with zero diagonal elements
////////////////////////////////////////////////////////////////////////////////
AdjacencyMatrix::ResultCode AdjacencyMatrix::CreateRandomGraph(std::vector <bool>
 ↪  &AdjacencyMatrix, const int64_t n, bool rand_init, double density)
{
	if (n <= 0)
	{
		return ercDimensionError; // error dimension
	}

	if (density < 0 || density > 1)
	{
		return ercDensityError; // error density
	}

	RandomFuncs::SparseSymmetricMatrixZeroDiagonalB(n, density, AdjacencyMatrix,
	 ↪  rand_init);

	return ercNoError;
}

////////////////////////////////////////////////////////////////////////////////
// Generate boolean sparse matrix with non-zero elements in center-th row and column excluding
 ↪  the diagonal element
////////////////////////////////////////////////////////////////////////////////
AdjacencyMatrix::ResultCode AdjacencyMatrix::CreateStarGraph(std::vector <bool>
 ↪  &AdjacencyMatrix, const int64_t n, int64_t center)
{
        if (n <= 0)
        {
                return ercDimensionError; // error dimension
        }

        if ((center < 0 && center != -1) || center > (n - 1))
        {
                return ercCenterError; // error central point
        }

        AdjacencyMatrix.resize(n * n);
        for (int64_t i = 0; i < n * n; i++)
                AdjacencyMatrix[i] = 0;

        int64_t k = center;
        if (k == -1)
                k = (int64_t)std::floor(0.5 * n) - 1;

        for (int64_t i = 0; i < n; i++)
        {
                if ( i != k)
                {
                        AdjacencyMatrix[i * n + k] = 1;
                        AdjacencyMatrix[k * n + i] = 1;
                }
        }

        return ercNoError;
}

////////////////////////////////////////////////////////////////////////////////
// Generate boolean user defined matrix
////////////////////////////////////////////////////////////////////////////////
AdjacencyMatrix::ResultCode AdjacencyMatrix::CreateUserDefinedGraph(std::vector <bool>
 ↪   &AdjacencyMatrix, const int64_t n, const std::string& filein)
{
        if (n <= 0)
        {
                return ercDimensionError;                        // error dimension
        }

        std::ifstream file;
        file.open(filein.c_str());
        if (file)
        {
                AdjacencyMatrix.resize(n * n);

                for(int64_t i = 0; i < n * n; i++)
                {
                        if (!file.eof())
                        {
                                double a;
                                file >> a;
                                if (a != 0)
                                        AdjacencyMatrix[i] = 1;
                                else
                                        AdjacencyMatrix[i] = 0;
                        }
                        else
                                return ercDimensionError;        // error input data dimension
                }
                file.close();
        }
        else
        {
                AdjacencyMatrix.clear();
                return ercEmptyInputError;                        // inpur file not found
        }


        return ercNoError;
}


////////////////////////////////////////////////////////////////////////////////
//ADMM iterative algorithm starts here
////////////////////////////////////////////////////////////////////////////////
```

```
int64_t ADMM_SDP_Algo(int64_t n, int64_t W_size_min, int64_t W_size_max, int64_t p_min, int64_t
 ↪   p_max, int64_t q_min, int64_t q_max, int64_t A_i_min, int64_t A_i_max, int64_t B_i_min,
 ↪   int64_t B_i_max, int64_t D_i_min, int64_t D_i_max, int64_t W_i_min, int64_t W_i_max,
 ↪   int64_t c_i_min, int64_t c_i_max, int64_t d_i_min, int64_t d_i_max,
 ↪   AdjacencyMatrix::AdjacencyMatrixType AdjacencyType, double density, int64_t center, const
 ↪   std::string& filein, double mu_mult, double overlap_ratio, double tole, const
 ↪   std::string& fileout, bool rand_init)
{
    // Start the timer for calculating algorithm initialization time
    auto t0 = std::chrono::high_resolution_clock::now();

        double inv_mu_mult = 1.0 / mu_mult;


        std::vector<bool> randAdj;
        AdjacencyMatrix::ResultCode result;
        switch(AdjacencyType)
        {
        case(AdjacencyMatrix::eamtBandedGraph):
                result = AdjacencyMatrix::CreateBandedGraph(randAdj, n);
//#ifdef _DEBUG
                std::cout << "Banded Graph" << std::endl;
                for (int64_t i = 0; i < n; i++)
                {
                        for (int64_t j = 0; j < n; j++)
                                std::cout << randAdj[i * n + j] << " ";
                        std::cout << std::endl;
                }
//#endif
                break;
        case(AdjacencyMatrix::eamtRandomGraph):
                result = AdjacencyMatrix::CreateRandomGraph(randAdj, n, rand_init, density);
//#ifdef _DEBUG
                std::cout << "Random Graph" << std::endl;
                for (int64_t i = 0; i < n; i++)
                {
                        for (int64_t j = 0; j < n; j++)
                                std::cout << randAdj[i * n + j] << " ";
                        std::cout << std::endl;
                }
//#endif
                break;
        case(AdjacencyMatrix::eamtStarGraph):
                result = AdjacencyMatrix::CreateStarGraph(randAdj, n, center);
//#ifdef _DEBUG
                std::cout << "Star Graph" << std::endl;
                for (int64_t i = 0; i < n; i++)
                {
                        for (int64_t j = 0; j < n; j++)
                                std::cout << randAdj[i * n + j] << " ";
                        std::cout << std::endl;
                }
//#endif
                break;
        case(AdjacencyMatrix::eamtUserDefinedGraph):
                result = AdjacencyMatrix::CreateUserDefinedGraph(randAdj, n, filein);
//#ifdef _DEBUG
                std::cout << "User Defined Graph" << std::endl;
                for (int64_t i = 0; i < n; i++)
                {
                        for (int64_t j = 0; j < n; j++)
                                std::cout << randAdj[i * n + j] << " ";
                        std::cout << std::endl;
                }
//#endif
                break;
        default:
                result = AdjacencyMatrix::ercTypeError;
        }

        if (result != AdjacencyMatrix::ercNoError)
                return -2;                // adjacency matrix creation error

    // get the indices of only the non-zero entries to define the set of edges
        std::vector<int64_t> edges_Set;
        for (int64_t i = 0; i < n; i++)
                for (int64_t j = i; j < n; j++)
                        if (randAdj[i * n + j] != 0)
                        {
                                edges_Set.push_back(i);
                                edges_Set.push_back(j);
                        }

        int64_t edges_Num = edges_Set.size();
        edges_Num /= 2;

    // Compute for each agent i:
    // neighb_all_num: the total number of agents connected to agent i
    // neighb_less_num: the total number of agents in the lower part of randAdj connected to
     ↪  agent i
    // neighb_greater_num: The total number of agents in the upper part of randAdj connected to
     ↪  agent i
        std::vector<int64_t> neighb_all_num(n, 0);
        std::vector<int64_t> neighb_less_num(n, 0);
        std::vector<int64_t> neighb_greater_num(n, 0);

        for (int64_t i = 0; i < n; i++)
        {
                for (int64_t j = 0; j < i; j++)
                        if (randAdj[i * n + j] != 0)
                        {
                                neighb_all_num[i]++;
                                neighb_less_num[i]++;
                        }
                for (int64_t j = i + 1; j < n; j++)
                        if (randAdj[i * n + j] != 0)
                        {
                                neighb_all_num[i]++;
                                neighb_greater_num[i]++;
                        }
        }

    // Find the set delta which is the set of agents connected to agent i
        std::vector<std::vector<int64_t> > delta_less(n);
        std::vector<std::vector<int64_t> > delta_greater(n);

        for (int64_t i = 0; i < n; i++)
        {
                delta_less[i].resize(neighb_less_num[i]);
                delta_greater[i].resize(neighb_greater_num[i]);

                for (int64_t j = 0, k = 0; j < i; j++)
                        if (randAdj[i * n + j] != 0)
                        {
                                delta_less[i][k] = j;
                                k++;
                        }

                for (int64_t j = i + 1, k = 0; j < n; j++)
                        if (randAdj[i * n + j] != 0)
                        {
                                delta_greater[i][k] = j;
                                k++;
                        }
        }

        randAdj.clear();
        randAdj.reserve(0);

    // Randomly define the number of data matrices B and D for each agent i.
        std::vector<int64_t> p_i;
        std::vector<int64_t> q_i;
        RandomFuncs::MatrixI(n, 1, p_i, p_min, p_max, rand_init);
        RandomFuncs::MatrixI(n, 1, q_i, q_min, q_max, rand_init);

        std::vector<int64_t> w_size_i;
        std::vector<std::vector<double> > A(n);
        std::vector<std::vector<double> > A_lower(n);
        std::vector<std::vector<std::vector<double> > > B(n);
        std::vector<std::vector<std::vector<double> > > B_lower(n);
        std::vector<std::vector<int64_t> > c_i(n);
        std::vector<std::vector<std::vector<double> > > D(n);
        std::vector<std::vector<std::vector<double> > > D_lower(n);
        std::vector<std::vector<int64_t> > d_i(n);

    // Randomly define the size of the variable W_i for each agent i.
        RandomFuncs::MatrixI(n, 1, w_size_i, W_size_min, W_size_max, rand_init);

    // Randomly create data matrices A, B and D for each agent each with size
        // w_size_i. The total number of data matrices B and D for each agent is p_i and q_i,
         ↪  respectively.
    // Also, randomly create the vectors c_i and d_i with sizes p_i and q_i, respectively.
        for (int64_t i = 0; i < n; i++)
        {
                int64_t w_size = w_size_i[i];
```

```cpp

      RandomFuncs::Matrix(w_size, w_size, A[i], A_i_min, A_i_max, rand_init, 1.0);

      for (int64_t j = 0; j < w_size; j++)
      {
          A[i][j * w_size + j] = 2 * A[i][j * w_size + j] + w_size_i[i];

          for (int64_t k = j + 1; k < w_size; k++)
          {
              A[i][j * w_size + k] = A[i][k * w_size + j] = A[i][j * w_size + k] + A[i][k * w_size + j];
          }
      }

      MatrixFuncs::LowerMatrix( w_size, A[i], A_lower[i]);

      if (p_i[i] != 0)
      {
          B[i].resize(p_i[i]);
          B_lower[i].resize(p_i[i]);
          for (int64_t p = 0; p < p_i[i]; p++)
          {
              RandomFuncs::Matrix(w_size, w_size, B[i][p], B_i_min, B_i_max, rand_init, 0, 0.01);

              for (int64_t j = 0; j < w_size; j++)
              {
                  B[i][p][j * w_size + j] = 2 * B[i][p][j * w_size + j];

                  for (int64_t k = 0; k < j; k++)
                  {
                      B[i][p][j * w_size + k] = B[i][p][k * w_size + j] = B[i][p][j * w_size + k] + B[i][p][k * w_size + j];
                  }
              }

              MatrixFuncs::LowerMatrix( w_size, B[i][p], B_lower[i][p]);

          }

          RandomFuncs::MatrixI(p_i[i], 1, c_i[i], c_i_min, c_i_max, rand_init);
      }

      if (q_i[i] != 0)
      {
          D[i].resize(q_i[i]);
          D_lower[i].resize(q_i[i]);
          for (int64_t q = 0; q < q_i[i]; q++)
          {
              RandomFuncs::Matrix(w_size, w_size, D[i][q], D_i_min, D_i_max, rand_init, 0, 0.01);

              for (int64_t j = 0; j < w_size_i[i]; j++)
              {
                  D[i][q][j * w_size + j] = 2 * D[i][q][j * w_size + j];
                  for (int64_t k = 0; k < j; k++)
                  {
                      D[i][q][j * w_size + k] = D[i][q][k * w_size + j] = D[i][q][j * w_size + k] + D[i][q][k * w_size + j];
                  }
              }

              MatrixFuncs::LowerMatrix( w_size, D[i][q], D_lower[i][q]);

          }

          RandomFuncs::MatrixI(q_i[i], 1, d_i[i], d_i_min, d_i_max, rand_init, -1);
      }
  }

// Define for each edge two sets of indices I_ij and I_ji which
  // specifies the rows and columns where the two agents variables W_i and
  // W_j are overlapping
  int64_t max_ij = 0, max_ji = 0, max_ = 0;
  for (int64_t i = 1; i < edges_Num; i++)
  {
      if (edges_Set[i * 2] > max_ij)
          max_ij = edges_Set[i * 2];
```

```

                        if (edges_Set[i * 2 + 1] > max_ji)
                                max_ji = edges_Set[i * 2 + 1];
                }
                max_ij++;
                max_ji++;
                max_ = std::max(max_ij, max_ji);

                std::vector <std::vector <int64_t> > I_ij(max_ij * max_ji);
                std::vector <std::vector <int64_t> > I_ji(max_ji * max_ij);

                std::vector <int64_t> overlap_size(max_ * max_, 0);

                for (int64_t i = 0; i < edges_Num; i++)
                {
                        int64_t indx_i = edges_Set[i * 2];
                        int64_t indx_j = edges_Set[i * 2 + 1];

                // Pick the minimum size between W_i and W_j. This represents the extreme case when
                // W_i (or W_j) lies completely inside W_j (or W_i)
                        int64_t k_min = std::min(w_size_i[indx_i], w_size_i[indx_j]);

                        int64_t overlap_size_i = (int64_t)(overlap_ratio * k_min + 0.5);
                        overlap_size[indx_i * max_ + indx_j] = overlap_size_i;
                        overlap_size[indx_j * max_ + indx_i] = overlap_size_i;

                // Randomly generate the set of unique indices I_ij at which W_i
                // overlaps with W_j
                        I_ij[indx_i * max_ji + indx_j].resize(overlap_size_i);
                        for (int64_t j = overlap_size_i - 1, count = 1; j >= 0; j--, count++)
                        {
                                I_ij[indx_i * max_ji + indx_j][j] = w_size_i[indx_i] - count;
                        }

                // Randomly generate the set of unique indices I_ji at which W_j
                // overlaps with W_i
                        I_ji[indx_j * max_ij + indx_i].resize(overlap_size_i);
                        for (int64_t j = 0; j < overlap_size_i; j++)
                        {
                                I_ji[indx_j * max_ij + indx_i][j] = j;
                        }
                }

                std::vector<std::vector<int64_t> > size_type_z_v_Hij_Hji(n);

                for (int64_t i = 0; i < n; i++)
                {
                        size_type_z_v_Hij_Hji[i].resize((2 + neighb_less_num[i] +
                         ↪  neighb_greater_num[i]) * 2, 0);
                        size_type_z_v_Hij_Hji[i][0] = p_i[i];
                        size_type_z_v_Hij_Hji[i][1 * 2] = q_i[i];

                        for (int64_t j = 0; j < neighb_less_num[i]; j++)
                        {
                                int64_t n_overlap = overlap_size[i * max_ + delta_less[i][j]];
                                size_type_z_v_Hij_Hji[i][(j + 2) * 2] = n_overlap * n_overlap;
                                size_type_z_v_Hij_Hji[i][(j + 2) * 2 + 1] = (int64_t)(n_overlap *
                                 ↪  (n_overlap + 1) * 0.5);
                        }

                        for (int64_t j = 0; j < neighb_greater_num[i]; j++)
                        {
                                int64_t n_overlap = overlap_size[i * max_ + delta_greater[i][j]];
                                size_type_z_v_Hij_Hji[i][(j + 2 + neighb_less_num[i]) * 2] = n_overlap
                                 ↪  * n_overlap;
                                size_type_z_v_Hij_Hji[i][(j + 2 + neighb_less_num[i]) * 2 + 1] =
                                 ↪  (int64_t)(n_overlap * (n_overlap + 1) * 0.5);
                        }
                }

                std::vector<int64_t> jacobian_size_i_lower(n);

                for (int64_t i = 0; i < n; i++)
                {
                        double jacobian_size_i = 0;
                        jacobian_size_i_lower[i] = p_i[i] + q_i[i];
                        for (int64_t j = 0; j < (2 + neighb_less_num[i] + neighb_greater_num[i]); j++)
                        {
                                jacobian_size_i += size_type_z_v_Hij_Hji[i][j * 2];
                                jacobian_size_i_lower[i] += size_type_z_v_Hij_Hji[i][j * 2 + 1];
                        }

```

```

        }

    // Preallocation and initial values of all variables and multipliers
        std::vector<std::vector<double> > z(n);
        std::vector<std::vector<double> > v(n);
        std::vector<std::vector<double> > u(n);
        std::vector<std::vector<double> > R_lower(n);
        std::vector<std::vector<double> > G_i_lower(n);
        std::vector<std::vector<double> > Lambda_i(n);

        std::vector<std::vector<double> > H_ij_lower(max_ij * max_ji);
        std::vector<std::vector<double> > H_ji_lower(max_ji * max_ij);
        std::vector<std::vector<double> > H_ij_coup_lower(max_ij * max_ji);
        std::vector<std::vector<double> > G_ij_lower(max_ij * max_ji);
        std::vector<std::vector<double> > G_ji_lower(max_ji * max_ij);

        std::vector<std::vector<double> > H_ij_basis(max_ij * max_ji);
        std::vector<std::vector<double> > H_ij_basis_full(max_ij * max_ji);
        std::vector<std::vector<double> > H_ij_basis_vec_tr(max_ij * max_ji);

        std::vector<std::vector<double> > H_ji_basis(max_ji * max_ij);
        std::vector<std::vector<double> > H_ji_basis_full(max_ji * max_ij);
        std::vector<std::vector<double> > H_ji_basis_vec_tr(max_ji * max_ij);


        std::vector<std::vector<int64_t> > H_ij_basis_map(max_ij * max_ji);
        std::vector<std::vector<int64_t> > H_ji_basis_map(max_ji * max_ij);

        std::vector<SparseMatrix> triang_jacobian(n);
        std::vector<SparseMatrix> s_jacobian(n);;

        std::vector<std::vector<double> > H_ij_sum_tr(n);
        std::vector<std::vector<double> > B_sum(n);
        std::vector<std::vector<double> > D_sum(n);

        std::vector<std::vector<double> > H_ij_sum_tr_lower(n);
        std::vector<std::vector<double> > B_sum_lower(n);
        std::vector<std::vector<double> > D_sum_lower(n);

        std::vector<double> p_infeas_i_1(n, 0);
        std::vector<double> p_infeas_i_2(edges_Num, 0);
        std::vector<double> d_infeas_i_1(n, 0);
        std::vector<double> d_infeas_i_3(n, 0);
        std::vector<std::vector<double> > d_infeas_i_2(2);
        d_infeas_i_2[0].resize(edges_Num, 0);
        d_infeas_i_2[1].resize(edges_Num, 0);

        std::vector<double> p_residue_i_1(n, 0);
        std::vector<double> p_residue_i_2(n, 0);
        std::vector<double> p_residue_i_3(edges_Num, 0);
        std::vector<double> p_residue_i_4(edges_Num, 0);

        std::vector<double> d_residue_i_1(n, 0);
        std::vector<double> d_residue_i_2(n, 0);
        std::vector<double> d_residue_i_3(edges_Num, 0);

        for (int64_t i = 0; i < n; i++)
        {
                z[i].resize(p_i[i], 0);
                v[i].resize(q_i[i], 0);
                u[i].resize(q_i[i], 0);
                Lambda_i[i].resize(q_i[i], 0);

                int64_t dim_lower = (int64_t)(0.5 * (w_size_i[i] * ( w_size_i[i] + 1)));

                R_lower[i].resize(dim_lower, 0);
                G_i_lower[i].resize(dim_lower, 0);
        }

        for (int64_t i = 0; i < edges_Num; i++)
        {
                int64_t indx_i = edges_Set[i * 2];
                int64_t indx_j = edges_Set[i * 2 + 1];
                int64_t temp_overlap_size = overlap_size[indx_i * max_ + indx_j];
                int64_t dim_lower = (int64_t)(0.5 * (temp_overlap_size * (temp_overlap_size +
                 ↪  1)));

                H_ij_lower[indx_i * max_ji + indx_j].resize(dim_lower, 0);
                H_ji_lower[indx_j * max_ij + indx_i].resize(dim_lower, 0);
                H_ij_coup_lower[indx_i * max_ji + indx_j].resize(dim_lower, 0);

                G_ij_lower[indx_i * max_ji + indx_j].resize(dim_lower, 0);
                G_ji_lower[indx_j * max_ij + indx_i].resize(dim_lower, 0);
        }

    // Create the basis matrix for H_ij and H_ji
        for (int64_t i = 0; i < edges_Num; i++)
        {
                int64_t indx_i = edges_Set[i * 2];
                int64_t indx_j = edges_Set[i * 2 + 1];
                const std::vector <int64_t> & IJ = I_ij[indx_i * max_ji + indx_j];
                int64_t temp_overlap_size = overlap_size[indx_i * max_ + indx_j];
                int64_t temp_w_size = w_size_i[indx_i];
                int64_t columns_count_vec = (int64_t)(0.5 * temp_overlap_size *
                 ↪   (temp_overlap_size + 1));
                H_ij_basis_full[indx_i * max_ji + indx_j].resize(temp_w_size * temp_w_size, 0);
                H_ij_basis_vec_tr[indx_i * max_ji + indx_j].resize(temp_w_size * temp_w_size *
                 ↪   columns_count_vec, 0);

                for (int64_t k = 0, count = 0; k < temp_overlap_size; k++, count++)
                {
                        for (int64_t j = 0; j < k; j++, count++)
                        {
                                H_ij_basis_full[indx_i * max_ji + indx_j][IJ[j] * temp_w_size +
                                 ↪   IJ[k]] = 1;
                                H_ij_basis_full[indx_i * max_ji + indx_j][IJ[k] * temp_w_size +
                                 ↪   IJ[j]] = 1;
                                H_ij_basis_vec_tr[indx_i * max_ji + indx_j][count * temp_w_size
                                 ↪   * temp_w_size + IJ[j] * temp_w_size + IJ[k]] = 1;
                                H_ij_basis_vec_tr[indx_i * max_ji + indx_j][count * temp_w_size
                                 ↪   * temp_w_size + IJ[k] * temp_w_size + IJ[j]] = 1;
                        }
                        H_ij_basis_full[indx_i * max_ji + indx_j][IJ[k] * temp_w_size + IJ[k]]
                         ↪   = 1;
                        H_ij_basis_vec_tr[indx_i * max_ji + indx_j][count * temp_w_size *
                         ↪   temp_w_size + IJ[k] * temp_w_size + IJ[k]] = 1;
                }
        }

        for (int64_t i = 0; i < edges_Num; i++)
        {
                int64_t indx_i = edges_Set[i * 2];
                int64_t indx_j = edges_Set[i * 2 + 1];
                const std::vector <int64_t> & JI = I_ji[indx_j * max_ij + indx_i];
                int64_t temp_overlap_size = overlap_size[indx_j * max_ + indx_i];
                int64_t temp_w_size = w_size_i[indx_j];
                int64_t columns_count_vec = (int64_t)(0.5 * temp_overlap_size *
                 ↪   (temp_overlap_size + 1));
                H_ji_basis_full[indx_j * max_ij + indx_i].resize(temp_w_size * temp_w_size, 0);
                H_ji_basis_vec_tr[indx_j * max_ij + indx_i].resize(temp_w_size * temp_w_size *
                 ↪   columns_count_vec, 0);

                for (int64_t k = 0, count = 0; k < temp_overlap_size; k++, count++)
                {
                        for (int64_t j = 0; j < k; j++, count++)
                        {
                                H_ji_basis_full[indx_j * max_ij + indx_i][JI[j] * temp_w_size +
                                 ↪   JI[k]] = -1;
                                H_ji_basis_full[indx_j * max_ij + indx_i][JI[k] * temp_w_size +
                                 ↪   JI[j]] = -1;
                                H_ji_basis_vec_tr[indx_j * max_ij + indx_i][(count) *
                                 ↪   temp_w_size * temp_w_size + JI[j] * temp_w_size + JI[k]]
                                 ↪   = -1;
                                H_ji_basis_vec_tr[indx_j * max_ij + indx_i][(count) *
                                 ↪   temp_w_size * temp_w_size + JI[k] * temp_w_size + JI[j]]
                                 ↪   = -1;
                        }
                        H_ji_basis_full[indx_j * max_ij + indx_i][JI[k] * temp_w_size + JI[k]]
                         ↪   = -1;
                        H_ji_basis_vec_tr[indx_j * max_ij + indx_i][(count) * temp_w_size *
                         ↪   temp_w_size + JI[k] * temp_w_size + JI[k]] = -1;
                }
        }

        // Create a mapping between the non-zero elements in H_ij_basis_full and
        // H_ij (the variable format). This will be used to update H_ij_sum for each agent i.
    // Similarly for H_ji_basis_full and H_ji
        for (int64_t i = 0; i < n; i++)
        {
                for (int64_t j = 0; j < neighb_less_num[i]; j++)
```

```
      {
          int64_t temp_delta_less = delta_less[i][j];
          int64_t temp_overlap_size = overlap_size[i * max_ + temp_delta_less];
          H_ji_basis_map[i * max_ij + temp_delta_less].resize( temp_overlap_size
           ↪  * (temp_overlap_size + 1) * 2, 0);

          for (int64_t p = 0, count = 0; p < w_size_i[i]; p++)
              for (int64_t k = 0; k <= p; k++)
                  if (H_ji_basis_full[i * max_ij + temp_delta_less][k *
                   ↪  w_size_i[i] + p] == -1)
                  {
                      H_ji_basis_map[i * max_ij +
                       ↪  temp_delta_less][count * 4] = k;
                      H_ji_basis_map[i * max_ij +
                       ↪  temp_delta_less][count * 4 + 1] = p;
                      count++;
                  }

          for (int64_t p = 0, count = 0; p < temp_overlap_size; p++)
              for (int64_t k = 0; k <= p; k++, count++)
              {
                  H_ji_basis_map[i * max_ij + temp_delta_less][count * 4
                   ↪  + 2] = k;
                  H_ji_basis_map[i * max_ij + temp_delta_less][count * 4
                   ↪  + 3] = p;
              }
      }

      for (int64_t j = 0; j < neighb_greater_num[i]; j++)
      {
          int64_t temp_delta_greater = delta_greater[i][j];
          int64_t temp_overlap_size = overlap_size[i * max_ +
           ↪  temp_delta_greater];
          H_ij_basis_map[i * max_ji + temp_delta_greater].resize(
           ↪  temp_overlap_size * (temp_overlap_size + 1) * 2, 0);

          for (int64_t p = 0, count = 0; p < w_size_i[i]; p++)
              for (int64_t k = 0; k <= p; k++)
                  if (H_ij_basis_full[i * max_ji + temp_delta_greater][k
                   ↪  * w_size_i[i] + p] == 1)
                  {
                      H_ij_basis_map[i * max_ji +
                       ↪  temp_delta_greater][count * 4] = k;
                      H_ij_basis_map[i * max_ji +
                       ↪  temp_delta_greater][count * 4 + 1] = p;
                      count++;
                  }

          for (int64_t p = 0, count = 0; p < temp_overlap_size; p++)
              for (int64_t k = 0; k <= p; k++, count++)
              {
                  H_ij_basis_map[i * max_ji + temp_delta_greater][count *
                   ↪  4 + 2] = k;
                  H_ij_basis_map[i * max_ji + temp_delta_greater][count *
                   ↪  4 + 3] = p;
              }
      }
  }

  bool error = false;

// Find the inverse of the Jacobian matrix for each agent i
#pragma omp parallel for schedule(guided)
  for (int64_t i = 0; i < n; i++)
  {
      int64_t dim2 = w_size_i[i] * w_size_i[i];
      int64_t dim1 = jacobian_size_i_lower[i];

  // Store the different terms for each agent in row and column format
  // which are multiplied later to create the different blocks of the jacobian

      SparseMatrix jacobian_col(dim1);
      std::vector<double> values(dim2);
      std::vector<uint64_t> columns(dim2);

      int64_t count = 0;
      for (int64_t j = 0; j < p_i[i]; j++, count++)
      {
          for (int64_t k = 0; k < dim2; k++)
          {
```

```
                                        values[k] = B[i][j][k];
                                        columns[k] = k;
                                }

                                jacobian_col.PushRow(count, values, columns, dim2);
                        }

                        for (int64_t j = 0; j < q_i[i]; j++, count++)
                        {
                                for (int64_t k = 0; k < dim2; k++)
                                {
                                        values[k] = D[i][j][k];
                                        columns[k] = k;
                                }

                                jacobian_col.PushRow(count, values, columns, dim2);
                        }

                        for (int64_t j = 0; j < neighb_less_num[i]; j++)
                                for (int64_t l = 0; l < size_type_z_v_Hij_Hji[i][(j + 2) * 2 + 1]; l++,
                                 ↪   count++)
                                {
                                        int64_t index = i * max_ij + delta_less[i][j];
                                        for (int64_t k = 0, nz = 0; k < dim2; k++)
                                        {
                                                double val = H_ji_basis_vec_tr[index][l * dim2 + k];
                                                if (val != 0)
                                                {
                                                        values[nz] = val;
                                                        columns[nz] = k;

                                                        nz++;
                                                }

                                                jacobian_col.PushRow(count, values, columns, nz);
                                        }
                                }

                        for (int64_t j = 0; j < neighb_greater_num[i]; j++)
                                for (int64_t l = 0; l < size_type_z_v_Hij_Hji[i][(j + 2 +
                                 ↪   neighb_less_num[i]) * 2 + 1]; l++, count++)
                                {
                                        int64_t index = i * max_ji + delta_greater[i][j];
                                        for (int64_t k = 0, nz = 0; k < dim2; k++)
                                        {
                                                double val = H_ij_basis_vec_tr[index][l * dim2 + k];
                                                if (val != 0)
                                                {
                                                        values[nz] = val;
                                                        columns[nz] = k;

                                                        nz++;
                                                }

                                                jacobian_col.PushRow(count, values, columns, nz);
                                        }
                                }

                        MatrixFuncs::MultiplySparse(dim1, dim2, jacobian_col, triang_jacobian[i]);
                        for (int64_t j = p_i[i]; j < dim1; j++)
                                triang_jacobian[i].Add(j, j, 1.0);

                        jacobian_col.clear();

                        MatrixFuncs::DevideByVectorAnaliticSymmSparse_Fase_1(triang_jacobian[i],
                         ↪   s_jacobian[i]);
                }
                if (error)
                        return -1;

                std::vector<double> gap;
                std::vector<double> max_infeas;
                std::vector<double> residue_sum;
                std::vector<double> residue_sum_primal;
                std::vector<double> residue_sum_dual;

                std::vector<double> p_residue_i_1_plot;
                std::vector<double> p_residue_i_2_plot;
                std::vector<double> p_residue_i_3_plot;
                std::vector<double> d_residue_i_1_plot;
                std::vector<double> d_residue_i_2_plot;
                std::vector<double> d_residue_i_3_plot;

```

```
        double max_infeas_iter = tole + 1;

    // Stop the timer for calculating algorthm's initialization time
    auto t1 = std::chrono::high_resolution_clock::now();
    auto dt = 1.e-9*std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
    std::cout<<"" << std::endl;
    std::cout<<"Algorithm's Initialization Time= "<< dt << " seconds" << std::endl;
    std::cout<<"" << std::endl;

    std::cout<<"Starting to solve using ADMM... " << std::endl;
    std::cout<<"" << std::endl;
    std::cout<<"ITE   PFEAS    DFEAS    POBJ         DOBJ         TIME" << std::endl;

    double time_s=0;
    int64_t iter = 0;

    // Start ADMM Algorithm main loop here
        while (max_infeas_iter > tole)
        {
        // Start the timer for calculating algorithm's main loop time
        auto t0 = std::chrono::high_resolution_clock::now();
        iter=iter+1;

                // Update (z_i, H_ij, H_ji)
#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < n; i++)
                {
                        std::vector<double> temp_Array;
                        std::vector<double> temp;
                        std::vector<double> vec_R_G_A;
                        std::vector<double> vec_R_G_A_lower;
                        std::vector<double> zi_vi_Hij_Hji_vec;

                        vec_R_G_A.clear();
                        vec_R_G_A_lower.clear();
                        MatrixFuncs::AddVectors(R_lower[i], G_i_lower[i], 1.0, inv_mu_mult,
                         ↪   vec_R_G_A_lower);
                        MatrixFuncs::AddVectors(vec_R_G_A_lower, A_lower[i], 1.0, -1.0,
                         ↪   vec_R_G_A_lower);
                        MatrixFuncs::SymmMatrixFromLowerMatrix(w_size_i[i], vec_R_G_A_lower,
                         ↪   vec_R_G_A);

                        int64_t dim2 = w_size_i[i] * w_size_i[i];
                        int64_t jacobian_size_lower = jacobian_size_i_lower[i];

                        temp_Array.clear();
                        temp_Array.resize(jacobian_size_lower, 0);
                        temp.clear();

                        int64_t count = 0;
                        for (int64_t j = 0; j < p_i[i]; j++, count++)
                        {
                                MatrixFuncs::MultiplyVectors(B[i][j], vec_R_G_A, 1.0,
                                 ↪   temp_Array[count]);
                                temp_Array[count] += -inv_mu_mult * c_i[i][j];
                        }

                        for (int64_t j = 0; j < q_i[i]; j++, count++)
                        {
                                MatrixFuncs::MultiplyVectors(D[i][j], vec_R_G_A, 1.0,
                                 ↪   temp_Array[count]);
                                temp_Array[count] += -inv_mu_mult * d_i[i][j] + u[i][j] -
                                 ↪   inv_mu_mult * Lambda_i[i][j];
                        }

                        for (int64_t j = 0; j < neighb_less_num[i]; j++)
                        {
                                int64_t dim = size_type_z_v_Hij_Hji[i][(j + 2) * 2 + 1];
                                MatrixFuncs::Multiply(dim, dim2, 1, H_ji_basis_vec_tr[i *
                                 ↪   max_ij + delta_less[i][j]], vec_R_G_A, false, false, 1,
                                 ↪   temp);
                                MatrixFuncs::AddVectors(H_ij_coup_lower[delta_less[i][j] *
                                 ↪   max_ji + i], temp, 1.0, 1.0, temp);
                                MatrixFuncs::AddVectors(G_ji_lower[i * max_ij +
                                 ↪   delta_less[i][j]], temp, - inv_mu_mult, 1.0, temp);
                                for (int64_t k = 0; k < dim; k++, count++)
                                        temp_Array[count] = temp[k];
                        }

                        for (int64_t j = 0; j < neighb_greater_num[i]; j++)
                        {
                                int64_t dim = size_type_z_v_Hij_Hji[i][(j + 2 +
                                 ↪   neighb_less_num[i]) * 2 + 1];
                                MatrixFuncs::Multiply(dim, dim2, 1, H_ij_basis_vec_tr[i *
                                 ↪   max_ji + delta_greater[i][j]], vec_R_G_A, false, false,
                                 ↪   1, temp);
                                MatrixFuncs::AddVectors(H_ij_coup_lower[i * max_ji +
                                 ↪   delta_greater[i][j]], temp, 1.0, 1.0, temp);
                                MatrixFuncs::AddVectors(G_ij_lower[i * max_ji +
                                 ↪   delta_greater[i][j]], temp, - inv_mu_mult, 1.0, temp);
                                for (int64_t k = 0; k < dim; k++, count++)
                                        temp_Array[count] = temp[k];
                        }

                        zi_vi_Hij_Hji_vec.clear();
            MatrixFuncs::DevideByVectorAnaliticSymmSparse_Fase_2(triang_jacobian[i],
             ↪   s_jacobian[i], temp_Array, zi_vi_Hij_Hji_vec);


                        int64_t size_sum = 0;

                        for (int64_t l = 0; l < p_i[i]; l++)
                                z[i][l] = zi_vi_Hij_Hji_vec[l];
                        size_sum += p_i[i];

                        for (int64_t l = 0, k = size_sum; l < q_i[i]; l++, k++)
                                v[i][l] = zi_vi_Hij_Hji_vec[k];
                        size_sum += q_i[i];

                        for (int64_t j = 0; j < neighb_less_num[i]; j++)
                        {
                                for (int64_t l = 0, k = size_sum; l <
                                 ↪   size_type_z_v_Hij_Hji[i][(j + 2) * 2 + 1]; l++, k++)
                                        H_ji_lower[i * max_ij + delta_less[i][j]][l] =
                                         ↪   zi_vi_Hij_Hji_vec[k];
                                size_sum += size_type_z_v_Hij_Hji[i][(j + 2) * 2 + 1];
                        }

                        for (int64_t j = 0; j < neighb_greater_num[i]; j++)
                        {
                                for (int64_t l = 0, k = size_sum; l <
                                 ↪   size_type_z_v_Hij_Hji[i][(j + 2 + neighb_less_num[i]) * 2
                                 ↪   + 1]; l++, k++)
                                        H_ij_lower[i  * max_ji + delta_greater[i][j]][l] =
                                         ↪   zi_vi_Hij_Hji_vec[k];
                                size_sum += size_type_z_v_Hij_Hji[i][(j + 2 +
                                 ↪   neighb_less_num[i]) * 2 + 1];
                        }
                }

                // Update H_ij_sum for each agent i
#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < n; i++)
                {
                        int64_t dim = w_size_i[i];

                        if ((int64_t)H_ij_sum_tr[i].size() != dim * dim)
                                H_ij_sum_tr[i].resize(dim * dim);

                        for (int64_t j = 0; j < dim * dim; j++)
                                H_ij_sum_tr[i][j] = 0;

                        for (int64_t j = 0; j < neighb_less_num[i]; j++)
                        {
                                int64_t dim1 = size_type_z_v_Hij_Hji[i][(j + 2) * 2 + 1];
                                int64_t indx = i * max_ij + delta_less[i][j];

                                for (int64_t k = 0; k < dim1; k ++)
                                        H_ij_sum_tr[i][H_ji_basis_map[indx][k * 4 + 1] * dim +
                                         ↪   H_ji_basis_map[ indx][k * 4]] +=
                                         ↪   -H_ji_lower[indx][k];
                        }

                        for (int64_t j = 0; j < neighb_greater_num[i]; j++)
                        {
                                int64_t dim1 = size_type_z_v_Hij_Hji[i][(j + 2 +
                                 ↪   neighb_less_num[i]) * 2 + 1];
                                int64_t indx = i * max_ji + delta_greater[i][j];

                                for (int64_t k = 0; k < dim1; k ++)
```

```
                                        H_ij_sum_tr[i][H_ij_basis_map[ indx][k * 4 + 1] * dim +
                                         ↪  H_ij_basis_map[ indx][k * 4]] +=
                                         ↪  H_ij_lower[indx][k];
                        }
                        MatrixFuncs::LowerMatrix(dim, H_ij_sum_tr[i], H_ij_sum_tr_lower[i]);
                }

        // Update B_sum for each agent i
#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < n; i++)
                {
                        if (p_i[i] != 0)
                        {
                                int64_t dim = (int64_t)(0.5 * w_size_i[i] * (w_size_i[i] + 1));

                                if ((int64_t)B_sum_lower[i].size() != dim)
                                        B_sum_lower[i].resize(dim);

                                for (int64_t k = 0; k < dim; k++)
                                        B_sum_lower[i][k] = 0;

                                for (int64_t j = 0; j < p_i[i]; j++)
                                        MatrixFuncs::AddVectors(B_sum_lower[i], B_lower[i][j],
                                         ↪  1, z[i][j], B_sum_lower[i]);
                        }
                }

        // Update D_sum for each agent i
#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < n; i++)
                {
                        if (q_i[i] != 0)
                        {
                                int64_t dim = (int64_t)(0.5 * w_size_i[i] * (w_size_i[i] + 1));

                                if ((int64_t)D_sum_lower[i].size() != dim)
                                        D_sum_lower[i].resize(dim);

                                for (int64_t k = 0; k < dim; k++)
                                        D_sum_lower[i][k] = 0;

                                for (int64_t j = 0; j < q_i[i]; j++)
                                        MatrixFuncs::AddVectors(D_sum_lower[i], D_lower[i][j],
                                         ↪  1, v[i][j], D_sum_lower[i]);
                        }
                }

        // Update R_i
#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < n; i++)
                {
                        std::vector<double> temp_Array;
                        std::vector<double> temp_mat;
                        std::vector<double> temp_mat_1;
                        std::vector<double> temp_mat_2;
                        std::vector<double> Veig;
                        std::vector<double> Deig_vec;
                        std::vector<double> Deig;
                        std::vector<double> R_lower_old;
                        std::vector<double> R_residue_lower;
                        std::vector<double> u_old;

                        int64_t dim = w_size_i[i];

                        temp_Array.clear();

                        MatrixFuncs::AddVectors(A_lower[i], H_ij_sum_tr_lower[i], 1.0, 1.0,
                         ↪  temp_Array);
                        MatrixFuncs::AddVectors(temp_Array, G_i_lower[i], 1.0, -inv_mu_mult,
                         ↪  temp_Array);
                        if (p_i[i] != 0)
                                MatrixFuncs::AddVectors(temp_Array, B_sum_lower[i], 1.0, 1.0,
                                 ↪  temp_Array);
                        if (q_i[i] != 0)
                                MatrixFuncs::AddVectors(temp_Array, D_sum_lower[i], 1.0, 1.0,
                                 ↪  temp_Array);

                        temp_mat.clear();
                        temp_mat_1.clear();
                        temp_mat_2.clear();

                        MatrixFuncs::SymmMatrixFromLowerMatrix(dim, temp_Array, temp_mat);

                        Veig.clear();
                        Deig_vec.clear();
                        Deig.clear();
                        Deig.resize(dim *dim, 0);

                        MatrixFuncs::EigenVectorsSymm( temp_mat, Deig_vec, Veig);

                        for (int64_t j = 0; j < dim; j++)
                                if (Deig_vec[j] > 0)
                                        Deig[j * dim + j] = Deig_vec[j];

                        MatrixFuncs::Multiply(dim, dim, dim, Deig, Veig, false, true, 1,
                         ↪  temp_mat_1);
                        MatrixFuncs::Multiply(dim, dim, dim, Veig, temp_mat_1, false, false, 1,
                         ↪  temp_mat_2);

                        R_lower_old.clear();
                        R_lower_old = R_lower[i];
                        MatrixFuncs::LowerMatrix(dim, temp_mat_2, R_lower[i]);

                        R_residue_lower.clear();
                        MatrixFuncs::AddVectors(R_lower[i], R_lower_old, 1.0, -1.0,
                         ↪  R_residue_lower);

                        double norm = 0;
                        MatrixFuncs::FrobeniusNormSymmLower(dim, R_residue_lower, norm);
                        d_residue_i_1[i] = norm * norm;

                        if (q_i[i] != 0)
                        {
                                u_old.clear();
                                u_old = u[i];
                                MatrixFuncs::AddVectors(v[i], Lambda_i[i], 1.0, inv_mu_mult,
                                 ↪  u[i]);

                                for (int64_t q = 0; q < q_i[i]; q++)
                                        u[i][q] = std::max(0.0, u[i][q]);

                                d_residue_i_2[i] = 0;
                                for (int64_t k = 0; k < q_i[i]; k++)
                                        d_residue_i_2[i] += (u[i][k] - u_old[k]) * (u[i][k] -
                                         ↪  u_old[k]);

                                MatrixFuncs::AddVectors(Lambda_i[i], v[i], 1.0, mu_mult,
                                 ↪  Lambda_i[i]);
                                MatrixFuncs::AddVectors(Lambda_i[i], u[i], 1.0, -mu_mult,
                                 ↪  Lambda_i[i]);
                        }

                        MatrixFuncs::AddVectors(G_i_lower[i], R_lower[i], 1.0, mu_mult,
                         ↪  G_i_lower[i]);
                        MatrixFuncs::AddVectors(G_i_lower[i], H_ij_sum_tr_lower[i], 1.0,
                         ↪  -mu_mult, G_i_lower[i]);
                        MatrixFuncs::AddVectors(G_i_lower[i], A_lower[i], 1.0, -mu_mult,
                         ↪  G_i_lower[i]);
                        if (p_i[i] != 0)
                                MatrixFuncs::AddVectors(G_i_lower[i], B_sum_lower[i], 1.0,
                                 ↪  -mu_mult, G_i_lower[i]);
                        if (q_i[i] != 0)
                                MatrixFuncs::AddVectors(G_i_lower[i], D_sum_lower[i], 1.0,
                                 ↪  -mu_mult, G_i_lower[i]);
                }

        // Update G_ij, G_ji and H_ij_coup
#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < edges_Num; i++)
                {
                        std::vector<double> H_ij_coup_lower_old;
                        std::vector<double> H_ij_coup_lower_residue;

                        int64_t ind_i = edges_Set[i * 2];
                        int64_t ind_j = edges_Set[i * 2 + 1];
                        int64_t dim = overlap_size[ind_i * max_ + ind_j];

                        H_ij_coup_lower_old.clear();
                        H_ij_coup_lower_old = H_ij_coup_lower[ind_i * max_ji + ind_j];

                        MatrixFuncs::AddVectors(H_ij_lower[ind_i * max_ji + ind_j],
                         ↪  H_ji_lower[ind_j * max_ij + ind_i], 0.5, 0.5,
                         ↪  H_ij_coup_lower[ind_i * max_ji + ind_j]);
                        MatrixFuncs::AddVectors(H_ij_coup_lower[ind_i * max_ji + ind_j],
                         ↪   G_ij_lower[ind_i * max_ji + ind_j], 1.0, 0.5 * inv_mu_mult,
                         ↪   H_ij_coup_lower[ind_i * max_ji + ind_j]);
                        MatrixFuncs::AddVectors(H_ij_coup_lower[ind_i * max_ji + ind_j],
                         ↪   G_ji_lower[ind_j * max_ij + ind_i], 1.0, 0.5 * inv_mu_mult,
                         ↪   H_ij_coup_lower[ind_i * max_ji + ind_j]);

                        H_ij_coup_lower_residue.clear();
                        MatrixFuncs::AddVectors(H_ij_coup_lower[ind_i * max_ji + ind_j],
                         ↪   H_ij_coup_lower_old, 1.0, -1.0, H_ij_coup_lower_residue);
                        double norm = 0;
                        MatrixFuncs::FrobeniusNormSymmLower(dim, H_ij_coup_lower_residue,
                         ↪   norm);
                        d_residue_i_3[i] = 2 * norm * norm;

                        MatrixFuncs::AddVectors(G_ij_lower[ind_i * max_ji + ind_j],
                         ↪   H_ij_lower[ind_i * max_ji + ind_j], 1.0, mu_mult,
                         ↪   G_ij_lower[ind_i * max_ji + ind_j]);
                        MatrixFuncs::AddVectors(G_ij_lower[ind_i * max_ji + ind_j],
                         ↪   H_ij_coup_lower[ind_i * max_ji + ind_j], 1.0, -mu_mult,
                         ↪   G_ij_lower[ind_i * max_ji + ind_j]);

                        MatrixFuncs::AddVectors(G_ji_lower[ind_j * max_ij + ind_i],
                         ↪   H_ji_lower[ind_j * max_ij + ind_i], 1.0, mu_mult,
                         ↪   G_ji_lower[ind_j * max_ij + ind_i]);
                        MatrixFuncs::AddVectors(G_ji_lower[ind_j * max_ij + ind_i],
                         ↪   H_ij_coup_lower[ind_i * max_ji + ind_j], 1.0, -mu_mult,
                         ↪   G_ji_lower[ind_j * max_ij + ind_i]);
                }

        // Calculate the stopping criteria measures
                double gap_primal_dual_pt1 = 0;
                double gap_primal_dual_pt2 = 0;
                double gap_primal_dual_pt3 = 0;
        double obj_primal = 0;
        double obj_dual = 0;

#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < n; i++)
                {
                        std::vector<double> temp_mat_G_i;
                        std::vector<double> temp;

                        int64_t dim = w_size_i[i];

                        temp_mat_G_i.clear();
                        MatrixFuncs::SymmMatrixFromLowerMatrix(dim, G_i_lower[i],
                         ↪   temp_mat_G_i);

                    double norm = 0;
                        temp.clear();

                        for (int64_t p = 0; p < p_i[i]; p++)
                        {
                                double mult = 0;
                                MatrixFuncs::MultiplyVectors(B[i][p], temp_mat_G_i, 1.0, mult);
                                norm += pow(mult - c_i[i][p], 2);
                        }

                        p_infeas_i_1[i] = sqrt(norm);

                        norm = 0;
                        for (int64_t q = 0; q < q_i[i]; q++)
                        {
                                double mult = 0;
                                MatrixFuncs::MultiplyVectors(D[i][q], temp_mat_G_i, 1.0, mult);
                                norm += pow(std::max(mult - d_i[i][q], 0.0), 2);
                        }

                        p_infeas_i_1[i] += sqrt(norm);

                        norm = 0;
                        for (int64_t p = 0; p < p_i[i]; p++)
                                norm += pow(c_i[i][p], 2);

                        if (p_i[i] == 0)
                                for (int64_t q = 0; q < q_i[i]; q++)
                                        norm += pow(d_i[i][q], 2);

                        p_infeas_i_1[i] /= 1 + sqrt(norm);

                        temp.clear();
                        MatrixFuncs::AddVectors(R_lower[i], H_ij_sum_tr_lower[i], 1.0, -1.0,
                         ↪  temp);
                        MatrixFuncs::AddVectors(temp, A_lower[i], 1.0, -1.0, temp);
                        if (p_i[i] != 0)
                                MatrixFuncs::AddVectors(temp, B_sum_lower[i], 1.0, -1.0, temp);
                        if (q_i[i] != 0)
                                MatrixFuncs::AddVectors(temp, D_sum_lower[i], 1.0, -1.0, temp);

                        MatrixFuncs::FrobeniusNormSymmLower(dim, temp, norm);

                        d_infeas_i_1[i] = norm;

                        p_residue_i_1[i] = norm * norm;

                        MatrixFuncs::PNormSymmLower(dim, 1, A_lower[i], norm);

                        d_infeas_i_1[i] /= 1 + norm;

                        norm = 0;
                        double norm1 = 0, norm2 = 0;
                        for (int64_t q = 0; q < q_i[i]; q++)
                        {
                                norm += pow(v[i][q] - u[i][q], 2);
                                norm1 += pow(v[i][q], 2);
                                norm2 += pow(u[i][q], 2);
                        }

                        d_infeas_i_3[i] = sqrt(norm) /(1.0 + sqrt(norm1) + sqrt(norm2));

                        p_residue_i_2[i] = norm;

                        double mult_1 = 0, mult_2 = 0, mult_3 = 0;
                        for (int64_t p = 0; p < p_i[i]; p++)
                                mult_1 += c_i[i][p] * z[i][p];

                        for (int64_t q = 0; q < q_i[i]; q++)
                                mult_2 += d_i[i][q] * v[i][q];

                        MatrixFuncs::MultiplyVectors(A[i], temp_mat_G_i, 1.0, mult_3);

                        #pragma omp critical
                        {
                                gap_primal_dual_pt1 -= mult_1;
                                gap_primal_dual_pt2 += mult_1;
                                gap_primal_dual_pt1 -= mult_2;
                                gap_primal_dual_pt2 += mult_2;
                                gap_primal_dual_pt1 -= mult_3;
                                gap_primal_dual_pt3 += mult_3;
                        }
                }

#pragma omp parallel for schedule(dynamic)
                for (int64_t i = 0; i < edges_Num; i++)
                {
                        std::vector<double> temp;

                        int64_t indx_i = edges_Set[i * 2];
                        int64_t indx_j = edges_Set[i * 2 + 1];

                        int64_t dim = overlap_size[indx_i * max_ + indx_j];

                        temp.clear();
                        MatrixFuncs::AddVectors(G_ij_lower[indx_i * max_ji + indx_j],
                         ↪  G_ji_lower[indx_j * max_ij + indx_i], 1.0, 1.0, temp);

                        double norm_1 = 0, norm_2 = 0;
                        MatrixFuncs::PNormVector(2, temp, norm_1);

                        p_infeas_i_2[i] = norm_1;

                        MatrixFuncs::PNormVector(2, G_ij_lower[indx_i * max_ji + indx_j],
                         ↪  norm_1);
                        MatrixFuncs::PNormVector(2, G_ji_lower[indx_j * max_ij + indx_i],
                         ↪  norm_2);

                        p_infeas_i_2[i] /= 1.0 + norm_1 + norm_2;

                        MatrixFuncs::AddVectors(H_ij_lower[indx_i * max_ji + indx_j],
                         ↪  H_ij_coup_lower[indx_i * max_ji + indx_j], 1.0, -1.0, temp);

                        MatrixFuncs::PNormVector(2, temp, norm_1);

                        d_infeas_i_2[0][i] = norm_1;

                        MatrixFuncs::PNormVector(2, H_ij_lower[indx_i * max_ji + indx_j],
                         ↪  norm_1);
                        MatrixFuncs::PNormVector(2, H_ij_coup_lower[indx_i * max_ji + indx_j],
                         ↪  norm_2);

                        d_infeas_i_2[0][i] /= 1.0 + norm_1 + norm_2;

                        MatrixFuncs::AddVectors(H_ji_lower[indx_j * max_ij + indx_i],
                         ↪  H_ij_coup_lower[indx_i * max_ji + indx_j], 1.0, -1.0, temp);

                        MatrixFuncs::PNormVector(2, temp, norm_1);

                        d_infeas_i_2[1][i] = norm_1;

                        MatrixFuncs::PNormVector(2, H_ji_lower[indx_j * max_ij + indx_i],
                         ↪  norm_1);
                        MatrixFuncs::PNormVector(2, H_ij_coup_lower[indx_i * max_ji + indx_j],
                         ↪  norm_2);

                        d_infeas_i_2[1][i] /= 1.0 + norm_1 + norm_2;

                        //        p_residue_i_3(i_index,1) =
                         ↪  (norm(temp_mat_H_ij-temp_mat_H_ij_coup,'fro'))^2;
                        MatrixFuncs::AddVectors(H_ij_lower[indx_i * max_ji + indx_j],
                         ↪  H_ij_coup_lower[indx_i * max_ji + indx_j], 1.0, -1.0, temp);

                        MatrixFuncs::FrobeniusNormSymmLower(dim, temp, norm_1);

                        p_residue_i_3[i] = norm_1 * norm_1;

                        //        p_residue_i_4(i_index,1) =
                         ↪  (norm(temp_mat_H_ji-temp_mat_H_ij_coup,'fro'))^2;
                        MatrixFuncs::AddVectors(H_ji_lower[indx_j * max_ij + indx_i],
                         ↪  H_ij_coup_lower[indx_i * max_ji + indx_j], 1.0, -1.0, temp);

                        MatrixFuncs::FrobeniusNormSymmLower(dim, temp, norm_1);

                        p_residue_i_4[i] = norm_1 * norm_1;
                }

                double gap_iter = std::abs(gap_primal_dual_pt1) / (1 +
                 ↪  std::abs(gap_primal_dual_pt2) + std::abs(gap_primal_dual_pt3));
                gap.push_back(gap_iter);

                double p_infeas_max_1 =        MatrixFuncs::Max(p_infeas_i_1);
                double p_infeas_max_2 =        MatrixFuncs::Max(p_infeas_i_2);
                double d_infeas_max_1 = MatrixFuncs::Max(d_infeas_i_1);
                double d_infeas_max_2_pt1 = MatrixFuncs::Max(d_infeas_i_2[0]);
                double d_infeas_max_2_pt2 = MatrixFuncs::Max(d_infeas_i_2[1]);
                double d_infeas_max_3 = MatrixFuncs::Max(d_infeas_i_3);

                max_infeas_iter = std::max(gap_iter, std::max(p_infeas_max_1,
                 ↪  std::max(p_infeas_max_2, std::max(d_infeas_max_1,
                 ↪  std::max(d_infeas_max_2_pt1, std::max(d_infeas_max_2_pt2,
                 ↪  d_infeas_max_3))))));
                max_infeas.push_back(max_infeas_iter);

        double p_infeas=std::max(p_infeas_max_1, p_infeas_max_2);
        double d_infeas=std::max(d_infeas_max_1,
         ↪  std::max(d_infeas_max_2_pt1,std::max(d_infeas_max_2_pt2,d_infeas_max_3)));

                double p_residue_i_1_sum = MatrixFuncs::Sum(p_residue_i_1);
                double p_residue_i_2_sum = MatrixFuncs::Sum(p_residue_i_2);
                double p_residue_i_3_sum = MatrixFuncs::Sum(p_residue_i_3);
                double p_residue_i_4_sum = MatrixFuncs::Sum(p_residue_i_4);
                double d_residue_i_1_sum = MatrixFuncs::Sum(d_residue_i_1);
                double d_residue_i_2_sum = MatrixFuncs::Sum(d_residue_i_2);
                double d_residue_i_3_sum = MatrixFuncs::Sum(d_residue_i_3);
                double p_residue_sum = p_residue_i_1_sum + p_residue_i_2_sum +
                 ↪  p_residue_i_3_sum + p_residue_i_4_sum;
                double d_residue_sum = d_residue_i_1_sum + d_residue_i_2_sum +
                 ↪  d_residue_i_3_sum;

                residue_sum.push_back(p_residue_sum + d_residue_sum);
                residue_sum_primal.push_back(p_residue_sum);
                residue_sum_dual.push_back(d_residue_sum);

                p_residue_i_1_plot.push_back(p_residue_i_1_sum);
                p_residue_i_2_plot.push_back(p_residue_i_2_sum);
                p_residue_i_3_plot.push_back(p_residue_i_3_sum);
                d_residue_i_1_plot.push_back(d_residue_i_1_sum);
                d_residue_i_2_plot.push_back(d_residue_i_2_sum);
                d_residue_i_3_plot.push_back(d_residue_i_3_sum);

        // Stop the timer for calculating algorthm's main loop time
        auto t1 = std::chrono::high_resolution_clock::now();
        auto dt = 1.e-9*std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
        time_s = time_s+dt;

        // Uncomment #ifdef _DEBUG and #endif to stop showing the solution in each iteration

        //#ifdef _DEBUG
        std::printf("%-3u   %1.1e  %1.1e  %e %e %2.2f\n", iter, p_infeas, d_infeas,
         ↪   -gap_primal_dual_pt2, gap_primal_dual_pt3, time_s);
        //#endif
        }

        double ADMM_Solution = 0;
        for (int64_t i = 0; i < n; i++)
        {
                double mult = 0;
                MatrixFuncs::MultiplyVectors( c_i[i], z[i], 1.0, mult);
                ADMM_Solution -= mult;
                MatrixFuncs::MultiplyVectors( d_i[i], v[i], 1.0, mult);
                ADMM_Solution -= mult;
        }


        Output(fileout, residue_sum);
        return 0;
}

// Write result to file. Here, residue_sum is written to a .csv file
void Output(const std::string& fileout, const std::vector <double> &data)
{
        std::ofstream file;
        file.open(fileout.c_str());
        int64_t dim = data.size();
        for(int64_t i = 0; i <dim; i++)
        {
                file << data[i] << std::endl;
        }
        file.close();
}

// Main function
int main(int argc, char *argv[])
{
        int64_t Solution = ADMM_SDP_Algo();
        return 0;
}
```

admm-sdp.h

```
const double DEF_TOLERANCE = 1E-13;
const double DEF_PRESICE = 1E-13;

// Square sparse matrix
// Only the non-zero elements are stored
class SparseMatrix
{
        uint64_t m_size;                                                                //
         ↪   matrix size, count of rows(columns)
        std::vector<std::vector<double> > m_Values;                // m_size-vector of real
         ↪   vectors, m_Values[i] - contains all non-zero elements of the i-th row
        std::vector<std::vector<uint64_t> > m_Columns;        // m_size-vector of integer
         ↪   vectors, m_Columns[i] - contains a numbers of columns corresponding to all
         ↪   non-zero elements of the i-th row
        std::vector<uint64_t> m_LastNonZeroElement;                // m_size-vector of integer
         ↪   numbers, vector contains the column indexes of last non-zero elements in each row
         ↪   of the matrix

        static double GetNAN()
        {
                uint64_t nan[2] = { 0xffffffff, 0x7fffffff };
                return *(double*)nan;
        }

public:
        SparseMatrix()
        {
                m_size = 0;
        };

        SparseMatrix(uint64_t size)
        {
                m_size = size;
                m_Values.resize(m_size);
                m_Columns.resize(m_size);
                m_LastNonZeroElement.resize(m_size, 0);
        };

        SparseMatrix(SparseMatrix &SMatrix)
        {
                m_size = SMatrix.m_size;
                m_Values = SMatrix.m_Values;
                m_LastNonZeroElement = SMatrix.m_LastNonZeroElement;
                m_Columns = SMatrix.m_Columns;
        };

        void create(uint64_t size)
        {
                clear();

                m_size = size;
                m_Values.resize(m_size);
                m_Columns.resize(m_size);
                m_LastNonZeroElement.resize(m_size, 0);
        };

        void create(SparseMatrix &SMatrix)
        {
                clear();

                m_size = SMatrix.m_size;
                m_Values = SMatrix.m_Values;
                m_LastNonZeroElement = SMatrix.m_LastNonZeroElement;
                m_Columns = SMatrix.m_Columns;
        };

        uint64_t size()
        {
                return m_size;
        };

        void resize(uint64_t size)
        {
                clear();

                m_size = size;
                m_Values.resize(m_size);
                m_Columns.resize(m_size);
                m_LastNonZeroElement.resize(m_size, 0);
        };

        void clear()
        {
                if (m_size > 0)
                {
                        m_size = 0;
                        m_Values.clear();
                        m_Values.resize(0);
                        m_LastNonZeroElement.clear();
                        m_LastNonZeroElement.resize(0);
                        m_Columns.clear();
                        m_Columns.resize(0);
                }
        };

        ////////////////////////////////////////////////////////////////////////////////
        // value = S[row][column]
        ////////////////////////////////////////////////////////////////////////////////
        void Get(uint64_t row, uint64_t column, double &value); // Search element starts with
         ↪  the biginning of the row

        ////////////////////////////////////////////////////////////////////////////////
        // value = S[row][column]
        ////////////////////////////////////////////////////////////////////////////////
        void GetLastElement(uint64_t row, uint64_t column, double &value); //Search element
         ↪  starts with the end of the row

        ////////////////////////////////////////////////////////////////////////////////
        // S[row][column] = value
        ////////////////////////////////////////////////////////////////////////////////
        void Set(uint64_t row, uint64_t column, double value);

        ////////////////////////////////////////////////////////////////////////////////
        // S[row][column] = S[row][column] + value
        ////////////////////////////////////////////////////////////////////////////////
        void Add(uint64_t row, uint64_t column, double value);

        ////////////////////////////////////////////////////////////////////////////////
        // permutation of rows I and J in the matrix
        ////////////////////////////////////////////////////////////////////////////////
        void SwapRows(uint64_t row_i, uint64_t row_j);

        ////////////////////////////////////////////////////////////////////////////////
        // addition of row I to row SUM and saving the result in the row SUM
        ////////////////////////////////////////////////////////////////////////////////
        void AddRow(uint64_t row_i, uint64_t row_sum, double alpha = 1.0);

        ////////////////////////////////////////////////////////////////////////////////
        // product of two rows like two vectors, the sum of the pairwise products of the
         ↪   elements
        ////////////////////////////////////////////////////////////////////////////////
        void RowsProduct(uint64_t row_i, uint64_t row_j, double & prod);

        ////////////////////////////////////////////////////////////////////////////////
        // product of row and vector like two vectors, the sum of the pairwise products of the
         ↪   elements
        ////////////////////////////////////////////////////////////////////////////////
        void RowVectorProduct(const std::vector<double> &x, uint64_t row, double & prod);

        ////////////////////////////////////////////////////////////////////////////////
        // filling the sparse matrix row values
        ////////////////////////////////////////////////////////////////////////////////
        void PushRow(uint64_t row, const std::vector<double> &values, const
         ↪   std::vector<uint64_t> &columns, uint64_t count);
};

class MatrixFuncs
{
        static double GetNAN()
        {
                uint64_t nan[2] = { 0xffffffff, 0x7fffffff };
                return *(double*)nan;
        }
public:
        enum ResultCode
        {
                ercNoError,
                ercInputDataError,
                ercSingularMatrixWarning,
                ercSingularMatrixError,
                ercNoConvergence
        };
public:
        ////////////////////////////////////////////////////////////////////////////////
        // Computes the inverse of matrix a, matrices are written in a 1-dim array row-wise
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - (dim-by-dim) matrix, row-wise

        // Output
        // a_inv - (dim-by-dim) matrix, inverted matrix a
        static MatrixFuncs::ResultCode Inverse (const std::vector<double> &a,
         ↪   std::vector<double> &a_inv);

        ////////////////////////////////////////////////////////////////////////////////
        // Computes eigenvectors and eigenvalues of a symmetric matrix
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - symmetric (m-by-m) matrix, row-wise

        // Output
        // eigen_values -  m-vector of eigenvalues
        // eigen_vectors -m-vector of m-vectors of eigenvectors, column-wise
        static MatrixFuncs::ResultCode EigenVectorsSymm(const std::vector<double> &a,
         ↪   std::vector<double> &eigen_values, std::vector<double> &eigen_vectors);

        ////////////////////////////////////////////////////////////////////////////////
        // Multiplication of real matrices written in a 1-dim array row-wise
```

```cpp
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - (m-by-dim) matrix (dim-by-m if transposed)
        // b - (dim-by-n) matrix (n-by-dim if transposed)
        // alpha - scalar factor

        // Output
        // mult - (m-by-n) matrix = alpha * a * b
        static void Multiply(        const int64_t &m, const int64_t &dim, const int64_t &n, const std::vector<double> &a, const std::vector<double> &b, bool left_trans, bool right_trans,
                         const double &alpha, std::vector<double> &mult);

        ////////////////////////////////////////////////////////////////////////////////
        // Multiplication of sparse real matrices
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - (m-by-dim) matrix
        // b - (dim-by-m) matrix

        // Output
        // mult - (m-by-m) matrix = alpha * a * b
        static void MultiplySparse( const int64_t &m, const int64_t &dim, SparseMatrix &a, SparseMatrix &mult);

        ////////////////////////////////////////////////////////////////////////////////
        // Solves the system of linear equations A*x = B for symmetric positive definite matrix A by using Cholesky decomposition(analitical method).
        // The matrices A  and B must have the same number of rows
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - (dim-by-dim) matrix
        // b - dim-vector

        // Output
        // x - dim-vector
        static MatrixFuncs::ResultCode DevideByVectorAnaliticSymm(        const std::vector<double> &a, const std::vector<double> &b, std::vector<double> &x);

        ////////////////////////////////////////////////////////////////////////////////
        // Solves the system of linear equations A*x = b for symmetric positive definite matrix A by using Gauss method(analitical method).
        // The matrices A  and vector b must have the same number of rows.
        // Algorithm is divided into two phases
        ////////////////////////////////////////////////////////////////////////////////

        ////////////////////////////////////////////////////////////////////////////////
        // Fase_1(preliminary calculations) - reduction matrix A to the lower triangular matrices
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a_triang - square sparse matrix

        // Output
        // a_triang - lower triangular matrix
        // s - transformation matrix

        /////////////////////////////////////////////////////////////////////////////////
        // Fase_2 - transformation of vector b (using transformation matrix S) and sequential computation of the vector x
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a_triang - lower triangular matrix
        // b - vector of constant terms
        // s - transformation matrix

        // Output
        // x - vector of unknown variables

        static MatrixFuncs::ResultCode DevideByVectorAnaliticSymmSparse_Fase_1( SparseMatrix &a_triang, SparseMatrix &s);
        static MatrixFuncs::ResultCode DevideByVectorAnaliticSymmSparse_Fase_2(        SparseMatrix &a_triang, SparseMatrix &s, const std::vector<double> &b, std::vector<double> &x);

        ////////////////////////////////////////////////////////////////////////////////
        // Addition of real vectors
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // v_1 - dim-vector
        // v_2 - dim-vector
        // alpha - scalar factor
        // beta - scalar factor

        // Output
        // sum - dim-vectors
        static void AddVectors(        const std::vector<double> &v_1, const
         ↪   std::vector<double> &v_2,        const double &alpha, const double &beta,
         ↪   std::vector<double> &sum);

        ////////////////////////////////////////////////////////////////////////////////
        //Restoring symmetric matrix from lower triangular part
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a_lower - 0.5*m*(m+1) vector

        // Output
        // a - (m-by-m) symmetric matrix, row-wise
        static void SymmMatrixFromLowerMatrix(        const int64_t &m, const
         ↪   std::vector<double> &a_lower, std::vector<double> &a);

        ////////////////////////////////////////////////////////////////////////////////
        //Recording lower triangular part of symmetric matrix
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - (m-by-m) symmetric matrix, row-wise

        // Output
        // a_lower - 0.5*m*(m+1) vector
        static void LowerMatrix(        const int64_t &m, const std::vector<double> &a,
         ↪   std::vector<double> &a_lower);

        ////////////////////////////////////////////////////////////////////////////////
        //Frobenius matrix norm calculation using lower triangular part of symmetric matrix
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a_lower - 0.5*m*(m+1) vector

        // Output
        // norm - Frobenius matrix norm
        static void FrobeniusNormSymmLower(        const int64_t &m, const std::vector<double>
         ↪   &a_lower, double &norm);

        ////////////////////////////////////////////////////////////////////////////////
        //P-norm calculation using lower triangular part of symmetric matrix
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a_lower - 0.5*m*(m+1) vector
        // p - order of the norm

        // Output
        // norm - p-norm of matrix
        static void PNormSymmLower(        const int64_t &m, const int64_t &p, const
         ↪   std::vector<double> &a_lower, double &norm);

        ////////////////////////////////////////////////////////////////////////////////
        //P-norm calculation for vector
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // v - dim-vector
        // p - order of the norm

        // Output
        // norm - p-norm of vector
        static void PNormVector( const int64_t &p, const std::vector<double> &v, double &norm);

        ////////////////////////////////////////////////////////////////////////////////
        // Multiplication of real vectors
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // v_1 - real vector
        // v_2 - real vector
        // alpha - scalar factor

        // Output
        // mult - scalar product of vectors
        static void MultiplyVectors( const std::vector<double> &v_1, const std::vector<double>
         ↪  &v_2, const double &alpha, double &mult);

        ////////////////////////////////////////////////////////////////////////////////
        // Multiplication of integer and real vectors
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // v_1 - integer vector
        // v_2 - real vector
        // alpha - scalar factor

        // Output
        // mult - scalar product of vectors
        static void MultiplyVectors( const std::vector<int64_t> &v_1, const std::vector<double>
         ↪  &v_2, const double &alpha, double &mult);

        ////////////////////////////////////////////////////////////////////////////////
        // Finding the maximum element in the vector
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - real vector

        // Return
        // maximum element
        static double Max( const std::vector<double> &a);

        ////////////////////////////////////////////////////////////////////////////////
        // Calculation of the amount of vector elements
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // a - real vector

        // Return
        // sum of the elements
        static double Sum( const std::vector<double> &a);

private:
        ///////////////////////////////////////////////////////////////////////////////
        // Computes the Hessenberg (tridiagonal in this case) form of a symmetric matrix A
        //H = SAS', where H is an upper Hessenberg matrix, S - ortogonal matrix and S' is S
         ↪  transposed
        ///////////////////////////////////////////////////////////////////////////////

        // Input
        // a - symmetric (m-by-m) matrix, row-wise

        // Output
        // s - ortogonal matrix
        // d - m-vector of diagonal elements of the tridiagonal symmetric matrix H
        // e - vector of subdiagonal  of H
        static void HessenbergFormSymm(const std::vector<double> &a, std::vector<double> &s,
         ↪  std::vector<double> &d, std::vector<double> &e);
};

class RandomFuncs
{
public:
        enum ResultCode
        {
                ercNoError,
                ercDimensionError,
                ercDensityError,
        };

        ///////////////////////////////////////////////////////////////////////////////
        // Generate integer random matrix
        ///////////////////////////////////////////////////////////////////////////////

        // Input
        // min - minimum value for the entries of matrix
        // max - maximum value for the entries of matrix
        // mult - multiplier

        // Output
        // rand_m - random integer n*m matrix
        static ResultCode MatrixI(int64_t n, int64_t m, std::vector <int64_t> &rand_m, uint64_t
         ↪  min, uint64_t max, bool rand_init, int64_t mult = 1);

        ////////////////////////////////////////////////////////////////////////////////
        // Generate real random matrix
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // min - minimum value for the entries of matrix
        // max - maximum value for the entries of matrix
        // mult - multiplier
        // max_add - maximum added value

        // Output
        // rand_m - random real n*m matrix
        static ResultCode Matrix(int64_t n, int64_t m, std::vector <double> &rand_m, uint64_t
         ↪  min, uint64_t max, bool rand_init, double max_add = 0, double mult = 1);

        ////////////////////////////////////////////////////////////////////////////////
        // Generate boolean sparse random matrix with zero diagonal elements
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // density - density of sparse matrix, the number of non-zero elements is approximately
         ↪  equal to density*n*n

        // Output
        // rand_m - boolean sparse random n*n matrix
        static ResultCode SparseSymmetricMatrixZeroDiagonalB(const int64_t n, const double
         ↪  density, std::vector <bool> &rand_m, bool rand_init);
private:
        // boolean variable that indicates the first call of a function from CKeStatRandomGens
        static bool FirstCall;
        static unsigned long long x;

        // parameters of the basic random number generator
        static const unsigned long long a = 8121;
        static const unsigned long long c = 28411;
        static const unsigned long long m = 134456;

        // generates next real number on [0,1]
        static double NextDouble();

        // generates next integer from {0,...,m - 1}
        static unsigned long NextInt();

        // initializes the seed
        static void InitSeed(bool rand_init = true);
};

class AdjacencyMatrix
{
public:
        enum ResultCode
        {
                ercNoError,
                ercDimensionError,
                ercDensityError,
                ercCenterError,
                ercEmptyInputError,
                ercTypeError
        };

        enum AdjacencyMatrixType
        {
                eamtBandedGraph,
                eamtRandomGraph,
                eamtStarGraph,
                eamtUserDefinedGraph
        };

        ////////////////////////////////////////////////////////////////////////////////
        // Generate boolean banded matrix
        ////////////////////////////////////////////////////////////////////////////////

        // Output
        // AdjacencyMatrix - boolean tridiagonal n*n matrix
        static ResultCode CreateBandedGraph(std::vector <bool> &AdjacencyMatrix, const int64_t
         ↪  n);

        ////////////////////////////////////////////////////////////////////////////////
        // Generate boolean sparse random matrix with zero diagonal elements
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // density - density of sparse matrix, the number of non-zero nondiagonal elements is
         ↪  approximately equal to density*n*(n-1)

        // Output
        // AdjacencyMatrix - boolean sparse random n*n matrix
        static ResultCode CreateRandomGraph(std::vector <bool> &AdjacencyMatrix, const int64_t
         ↪  n, bool rand_init, double density = 0.1);

        ////////////////////////////////////////////////////////////////////////////////
        // Generate boolean sparse matrix with non-zero elements in center-th row and column
         ↪  excluding the diagonal element
        ////////////////////////////////////////////////////////////////////////////////

        // Output
        // AdjacencyMatrix - boolean sparse n*n matrix
        static ResultCode CreateStarGraph(std::vector <bool> &AdjacencyMatrix, const int64_t n,
         ↪  int64_t center = -1);

        ////////////////////////////////////////////////////////////////////////////////
        // Generate boolean user defined matrix
        ////////////////////////////////////////////////////////////////////////////////

        // Input
        // filein - input file

        // Output
        // AdjacencyMatrix - boolean n*n matrix
        static ResultCode CreateUserDefinedGraph(std::vector <bool> &AdjacencyMatrix, int64_t
         ↪  n, const std::string& filein = "adjacency_matrix.csv");

};

static void Output(const std::string& fileout, const std::vector <double> &data);


/*
 --------------------------------------------------------------------------------
 --Please read the following definitions of the different parameters
 needed to randomly generate multiagent SDP problems:

 n - total number of agents

 W_size_min - minimum possible size of variable W_i
 W_size_max - maximum possible size of variable W_i

 p_min - minimum possible number of data matrices B (equality costraints)
 p_max - maximum possible number of data matrices B (equality costraints)

 q_min - minimum possible number of data matrices D (inequality costraints)
 q_max - maximum possible number of data matrices D (inequality costraints)

 A_i_min - minimum value for the entries of matrices A_i
 A_i_max - maximum value for the entries of matrices A_i

 B_i_min - minimum value for the entries of matrices B_i
 B_i_max - maximum value for the entries of matrices B_i

 D_i_min - minimum value for the entries of matrices D_i
 D_i_max - maximum value for the entries of matrices A_i, B_i, D_i

 c_i_min - minimum value for the entries of vectors c_i
 c_i_max - maximum value for the entries of vectors c_i

 d_i_min - minimum value for the entries of vectors d_i
 d_i_max - maximum value for the entries of vectors d_i

 AdjacencyMatrix::AdjacencyMatrixType AdjacencyType = AdjacencyMatrix::***** - to get different
 ↪  graphs, please change ***** with one of the following options:
 --> eamtBandedGraph: creates a banded (path) graph. Inputs are: n
 --> eamtRandomGraph: creates a random graph. Inputs are: n, desnity
 --> eamtStarGraph: creates a star graph. Inputs are: n, center
 --> eamtUserDefinedGraph: creates a graph that is read from a file called
 ↪  "adjacency_matrix.csv" which should be created by the user. Make sure that the number of
 ↪  agents "n" are matching in "adjacency_matrix.csv" and the one that is defined here.

 density - density of the randomly generated graph when "eamtRandomGraph" is chosen

 center - defines the center node in the star graph when "eamtStarGraph" is chosen

 std::string& filein = "adjacency_matrix.csv" - this file is read to define the graph when
 ↪  "eamtUserDefinedGraph" is chosen

 mu_mult - a constant multiplier for ADMM that the user should specify (usually chosen as 0.1)

 overlap_ratio - this ratio specifies the number of entries of W_i which overlap with other
 ↪   agent's matrices W_j

 tole - this specifies the desired precision of the final solution

 std::string& fileout = "residue_sum.csv" - this file writes the aggregate residue to the file
 ↪   "residue_sum.csv". After the code run, please run "fig_plot.m" so you could plot the
 ↪   aggregate residue from "residue_sum.csv". You can't plot data in "residue_sum.csv"
 ↪   directly from the c++ code

 rand_init - please use this as follows:
 --> rand_init = true - this will create different instances of SDP at every code run
 --> rand_init = false - this will create the same SDP instance at every code run which is also
 ↪   the same as the Matlab code (for verifying the correctness of the final answer). This is
 ↪   achieved by defining the following four parameters exactly the same in the c++ and Matlab
 ↪   (please don't change them since very specific combinations should be chosen):

 a = 8121            (could be found in the file "admm_sdp.h" at line 258)
 c = 28411           (could be found in the file "admm_sdp.h" at line 259)
 m = 134456          (could be found in the file "admm_sdp.h" at line 260)
 RandomFuncs::x = 5  (could be found in the file "admm_sdp.cpp" at line 1110)

 -------------------------------------------------------------------------------

 Addtional Notes:

 -- The data matrices are created as follows:
 -> A = rand + rand' + n_i * eye(n_i) (where integer elements of rand in [A_i_min, A_i_max])
 -> B = rand + rand'                  (where integer elements of rand in [B_i_min, B_i_max])
 -> D = rand + rand'                  (where integer elements of rand in [D_i_min, D_i_max])

 -- The data vectors are created as follows:
 -> c = rand    (where integer elements of rand in [c_i_min, c_i_max])
 -> d = rand    (where integer elements of rand in [d_i_min, d_i_max])
 -------------------------------------------------------------------------------

 Known Bugs:

 - Using "eamtRandomGraph" and "rand_init = false" will create a segmentation error.
 -------------------------------------------------------------------------------
 */



////////////////////////////////////////////////////////////////////////////////
// Please specify these paramters needed to randomly generate Multiagent SDP Problems
////////////////////////////////////////////////////////////////////////////////


static int64_t ADMM_SDP_Algo(int64_t n = 100, int64_t W_size_min = 40, int64_t W_size_max = 40,
 ↪   int64_t p_min = 5, int64_t p_max = 5, int64_t q_min = 5, int64_t q_max = 5, int64_t
 ↪   A_i_min = 1, int64_t A_i_max = 5, int64_t B_i_min = 1, int64_t B_i_max = 5, int64_t
 ↪   D_i_min = 1, int64_t D_i_max = 5, int64_t W_i_min = 1, int64_t W_i_max = 2, int64_t
 ↪   c_i_min = 1, int64_t c_i_max = 3, int64_t d_i_min = 1, int64_t d_i_max = 3,
 ↪   AdjacencyMatrix::AdjacencyMatrixType AdjacencyType = AdjacencyMatrix::eamtBandedGraph,
 ↪   double density = 0.1, int64_t center = 0, const std::string& filein =
 ↪   "adjacency_matrix.csv", double mu_mult = 0.1, double overlap_ratio = 0.25, double tole =
 ↪   1e-3, const std::string& fileout = "residue_sum.csv", bool rand_init = false);
```