Technische Universität Berlin

Masterthesis

Modeling Decentralized Electricity Markets

Solving Multi-Period Optimal Power Flow using Alternating Direction Method of Multipliers

Eric Rockstädt
Industrial Engineering and Management

supervised by
Dr. Richard Weinhold
Prof. Dr. Christian von Hirschhausen

Fakultät VII – Wirtschaft und Management
Fachgebiet Wirtschafts- und Infrastrukturpolitik

Berlin, July 20, 2022

Abstract

Modern electricity systems have undergone an enormous transformation process in the last decades. One primary driver has been climate change and the involved actions to reduce carbon emissions. More and more renewable, non-dispatchable energy resources like photovoltaic and wind generators were integrated into almost every national electricity network, increasing the number of market participants and opposing challenges to the transmission grid operators to sustain a reliable electricity supply. On top of these changes, the availability of high-performance technology at very low costs enables new digital innovations to be on the forerun. One of those innovations is the trend toward decentralized systems. The most famous example is undoubtedly the cryptocurrency Bitcoin which provides an alternative to the centralized banking system and showcases a way to conduct transactions without an intermediary. This thesis investigates whether it is possible to decentralize an optimal power flow calculation that is a prevalent task of every transmission system operator. The optimal power flow considers multi-periods and the integration of energy storage resources. Based on the Alternating Direction Method of Multipliers and a review of current papers related to decentralized electricity markets, a decentralized algorithm is developed that solves an optimal power flow without a central entity knowing all sensitive information about the market participants. All computation is done by the market participants and is exchanged via an information network. The decentralized algorithm is applied to a three node case study system, and the obtained results are compared to a centralized optimal power flow. The comparison yields that the results are nearly identical except for minor differences in the per mille range. Some convergence problems were faced while implementing the mathematical formulations. They were removed by adapting the algorithm. Finally, a decentralized algorithm could be established and published as an open-source package to solve an optimal power flow with multi time periods and energy storage resources. The derivation and implementation of this algorithm are thoroughly documented in this thesis.

Zusammenfassung

In den letzten Jahren haben sich die Übertragungsnetze elektrischer Energiesysteme sehr stark gewandelt. Wesentlicher Treiber dieser Veränderung war der Klimawandel und die damit verbundenen Maßnahmen zur Reduzierung der Treibhausgase. Immer mehr erneuerbare Technologien wie zum Beispiel Solaranlagen oder Windkraftanlagen wurden in fast allen Stromnetzen weltweit integriert. Dies führt dazu, dass sich die Anzahl der Marktteilnehmer vervielfacht und die Übertragungsnetzbetreiber viele Herausforderungen lösen müssen, um weiterhin einen sicheren Netzbetrieb zu gewährleisten. Zusätzlich werden digitale Innovationen immer mehr genutzt, weil die Kosten für leistungsfähige Technik gering sind. Eine dieser Innovationen ist der Trend zu dezentralen Systemen. Das bekanntlich beste Beispiel für dezentrale Systeme ist die Kryptowährung Bitcoin, die eine Alternative zum bisherigen Bankwesen darstellt und eine Möglichkeit bietet, Transaktionen ohne einen Intermediär durchzuführen. Die vorliegende Arbeit untersucht, inwieweit es möglich ist, einen dezentralen Algorithmus zu entwickeln, der einen optimalen Lastfluss für ein elektrisches Netzwerk berechnet. Dies ist eine typische Aufgabe, die Übertragungsnetzbetreiber durchführen müssen. In der Lastflussberechnung werden mehrere Zeitperioden und elektrische Speicher berücksichtigt. Basierend auf dem Alternating Direction Method of Multipliers Algorithmus und einer Recherche des aktuellen Stands über dezentrale Energiesystem wird ein dezentraler Algorithmus entwickelt, der einen optimalen Lastfluss berechnet, ohne dass es eine zentrale Einheit gibt, die jegliche, sensiblen Informationen der Marktteilnehmer kennt. Die Berechnung wird ausschließlich von den Marktteilnehmern ausgeführt. Der dezentrale Algorithmus wird auf eine Fallstudie mit drei Netzwerkknoten angewandt und mit den Ergebnissen einer zentralen Lastflussberechnung verglichen. Der Vergleich zeigt, dass die Ergebnisse bis auf Abweichungen im Promillenbereich identisch sind. Bei der Implementierung der mathematischen Formulierungen sind einige Konvergenzprobleme aufgetreten, die durch kleine Anpassungen des Algorithmus behoben wurden. Der Programmcode der Implementierung wurde als open-source Paket veröffentlicht. Die Herleitungen und die Implementierung des dezentralen Algorithmus zum Lösen einer optimalen Lastflussberechnung sind in der vorliegenden Arbeit genauestens dokumentiert.

Contents

List of Figures

List of Tables

List of Listings

List of Symbols

In the following, several symbols are described that will be later used within this thesis.

Parameters

ϵ	Threshold term for the convergence of the algorithm	10^{-3}
$\Gamma_{l,t}$	Average of slack variable $K_{l,t}$ for the lower limit of the previous iteration	1 MW
$\overline{e_s}$	Maximum energy level of storage s	1 MW h
$\overline{f_l}$	Maximum line capacity of transmission line l	1 MW
$\overline{p_g}$	Maximum power of generator g	1 MW
$\overline{p_s}$	Maximum withdrawal and charging power of storage s	1 MW
$\Phi_{n,t}$	Sum of the previous iteration discharge power values' of all storages located at node n and timestep t	1 MW
$\Psi_{n,t}$	Sum of the previous iteration charge values' of all storages located at node n and timestep t	1 MW
$\Theta_{n,t}$	Sum of the previous iteration power values' of all generators located at node n and timestep t	1 MW
$\Upsilon_{l,t}$	Average of slack variable $U_{l,t}$ for the upper limit of the previous iteration	1 MW
$d_{n,t}$	Demand at node n and timestep t	1 MW
mc_g	Marginal costs of generator g	1 EUR/MW
mc_s	Marginal costs of storage s	1 EUR/MW

Sets

$\mathcal{G}$	Set of generators
$\mathcal{L}$	Set of transmission lines
$\mathcal{N}$	Set of nodes
$\mathcal{S}$	Set of storages

$\mathcal{T}$	Set of hourly timesteps	

Variables

$\hat{\lambda}_t$	Scaled dual variable of the system balance constraint	
$\hat{\mu}_{l,t}$	Scaled dual variable of the upper power flow constraint	
$\hat{\rho}_{l,t}$	Scaled dual variable of the lower power flow constraint	
λ_t	Dual variable of the system balance constraint	
$\mu_{l,t}$	Dual variable of the upper power flow constraint	
$\rho_{l,t}$	Dual variable of the lower power flow constraint	
$C_{s,t}$	Charging power of storage s at timestep t	1 MW
$D_{s,t}$	Withdrawal power of storage s at timestep t	1 MW
$E_{s,t}$	Energy storage level of storage s at timestep t	1 MW h
$K_{l,t}$	Slack variable for the lower capacity of transmission line l at timestep t	1 MW
$P_{g,t}$	Power generation of generator g at timestep t	1 MW
$r_{n,t}$	Nodal price of node n at timestep t	1 EUR
$U_{l,t}$	Slack variable for the upper capacity of transmission line l at timestep t	1 MW

Abbreviations

AC	Alternating Current
ADMM	Alternating Direction Method of Multipliers
ALR	Augmented Lagrangian Relaxation
DC	Direct Current
LR	Lagrangian Relaxation
PTDF	Power Transfer Distribution Factor
TNS	Three Node System
TSO	Transmission System Operator

1 Introduction

The importance of renewable energy resources like photovoltaic and wind generators is undeniable in the context of reducing carbon emissions to tackle climate change. The annual capacity expansion rates of renewable energy generators underpin this. For example, the installed solar energy capacity worldwide increased from 2.7 GW in 2003 to 700 GW in 2020 (Ritchie, Roser, and Rosado 2020). Nearly the same growth rate can be seen in Germany, where the extension of renewable energy resources was accelerated by the "Renewable Energy Law" (Erneuerbare-Energien-Gesetz EEG) and feed-in tariffs for photovoltaic (Pesch, Allelein, and Hake 2014). The installed photovoltaic capacities in Germany increased from 0.4 GW in 2003 to 51.5 GW in 2020 (Bundesnetzagentur 2021). Approximately two million photovoltaic systems had been installed until 2020 (Bundesnetzagentur 2021).

According to Quint et al. (2019), renewable energy resources have a significant impact on electricity transmission system planning and operating. They also state that the expansion of renewables is one of the most substantial changes for transmission systems since the integration of the alternating current. In most countries, the Transmission System Operator (TSO) is responsible for a reliable supply of electricity and for grid extensions that improve the system. The TSO takes care of balancing demand and supply by dispatching all power plants in its control area accordingly. The dispatch process is based on the costs of the power plants. Subsequently, each power plant operator has to share detailed and most often sensitive cost data with the TSO. As stated by Ahlqvist, Holmberg, and Tangerås (2022), this sharing of financial data is one of the main characteristics that define a centralized electricity market.

A centralized market schema has a lot of disadvantages, according to Ahlqvist, Holmberg, and Tangerås (2022). For one, producers have an incentive to overestimate their costs to be included in the dispatch. Second, centralized electricity markets tend to be not flexible enough to incorporate new technologies like energy storage resources or demand response. On the other hand, decentralized electricity markets have the advantage that less coordination is needed. The producer does not need to share its detailed cost information and can choose the best production plan on its own (Ahlqvist, Holmberg, and Tangerås 2022). This could be then referred to as decentralized decision-making. Subsequently, a need to establish decentralized processes in transmission grid operating can be identified.

A decentralized market schema would reduce the coordination work of the TSOs and would smoothen the pathway to incorporate even more renewable energy resources into the transmission grid system. Based on the current state of research, there is a lack of decentralized algorithms that can be used to solve an optimal power flow in a transmission grid system. An optimal power flow optimization is a typical, daily task that a TSO has to conduct to ensure the supply of electricity in its control area. Existing papers that investigate a decentralized optimal power flow often miss a detailed description of the derivation and implementation of the algorithm or ignore new technologies like energy storage resources. Unfortunately, no algorithm's source code has been published. Hence, there is a motivation to establish a decentralized algorithm that solves an optimal power flow and publish the implementation as an open-source package that others can use and extend. This led to the following research questions:

1. Is it possible to implement a decentralized algorithm with the help of the Alternating Direction Method of Multipliers (ADMM) that can optimize an optimal power flow without sharing sensitive cost details?

2. Can the algorithm be extended by energy storage resources that introduce an intertemporal component into the framework?

3. Does the decentralized algorithm yield the same results as a centralized algorithm on a case study system?

The thesis provides with section 2 the necessary knowledge to understand the derivation and implementation of the decentralized algorithm. First, the dispatch process of the German TSOs and the concept of the optimal power flow optimization are explained. Afterward, the fundamentals of the ADMM are described, and a summary of the current state of the research is made to confirm the mentioned gaps. Section 3 constitutes the main work of this thesis. This section applies the fundamentals to derive the mathematical formulations of a centralized optimal power flow. These equations are then further transformed into a decentralized problem formulation. Next, the implementation in Julia of both the centralized and decentralized algorithm is presented to the reader. Code listings are used to illustrate the implementation better. Finally, the centralized and decentralized algorithms are tested on a three node case system, and the retrieved results are evaluated.

2 Theoretical Approach

This chapter describes all necessary fundamentals to follow along with the mathematical derivation and the implementation in Julia of the decentralized algorithm to solve an optimal power flow optimization problem. First, the economic dispatch process is explained, which is then extended by considering transmission lines and their power flows in the modeling framework. This leads to the concept of optimal power flow optimization. Afterward, the reader is introduced to the ADMM by describing the basics with a simple example. Finally, the current state of research is summarized in the context of decentralized algorithms that solve an optimal power flow.

2.1 Centralized Economic Dispatch

According to Chowdhury and Rahman (1990) and Hetzer, Yu, and Bhattarai (2008), economic dispatch is a technique to allocate all available generators in an electrical power system to minimize the total system costs while making sure that the system load is covered. A concept that is often used to explain the economic dispatch is the merit order curve in which the supply bids of all generators are sorted by their marginal costs in ascending order. These supply bids are accepted as long as the system load is not satisfied. The marginal costs of the last accepted generator define the market price. As a result, renewable energy resources are always favored because their marginal costs are insignificant (Hetzer, Yu, and Bhattarai 2008).

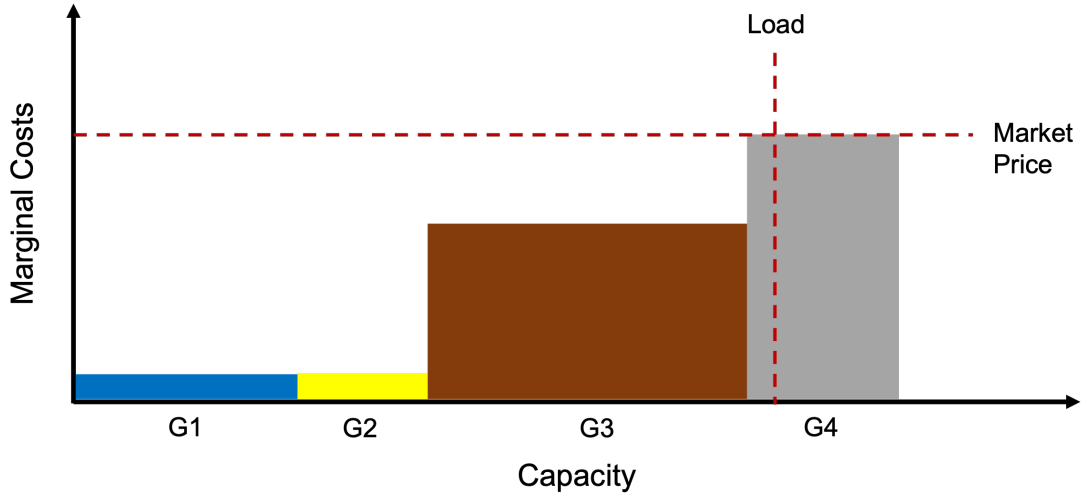

Figure 1: Exemplary merit order curve of four generators

Generators $\mathcal{G}$	mc [EUR/MWh]	$\overline{p}$ [MW]
G1	0	200
G2	0	50
G3	30	300
G4	50	100

Table 1: Exemplary generator parameters for an economic dispatch

Figure (1) shows an exemplary merit order curve of four generators out of which two generators, namely generator one (G1) and generator two (G2), are renewable energy resources like photovoltaic or wind. Generators G3 and G4 are conventional generators like a coal or gas plant. The load is depicted as a vertical, dashed red line. To solve the economic dispatch, one could use the merit order curve. As figure (1) shows, the capacities of generators one to three are not enough to cover the system load. Thus, generator four must be dispatched, setting the market price to its marginal costs. However, generator four is not producing at its capacity limit. This problem could also be solved by formulating the following cost-minimizing optimization problem:

$$\min \quad \sum_{g \in \mathcal{G}} mc_g \cdot P_g \tag{1a}$$

$$\text{s.t.} \quad 0 \leq P_g \leq \overline{p_g} \qquad \forall g \in \mathcal{G} \tag{1b}$$

$$\sum_{g \in \mathcal{G}} P_g = d \tag{1c}$$

The above formulation resembles the graphical solution of the merit order curve. Hereby, mc_g depicts the marginal costs and P_g the power output of the generators. Equation (1a) describes the total system costs that are minimized under two constraints. The power output of the generators has to be within the corresponding capacity limits denoted by $\overline{p_g}$, see constraint (1b). Secondly, the demand d has to be covered by the production of the generators. This is ensured by constraint (1c). In comparison to the graphical solution of the merit order curve, this approach has some advantages. First, it can be extended by multiple periods and incorporate further restrictions like ramping constraints. In addition, a lot of algorithms exist that can quickly solve similar problem formulations, e.g., the simplex algorithm (Dantzig 1963). The solution of the cost-minimization for the parameters in table (1) and a system load of 560 MW yields total costs of 9500 EUR and the following optimal production vector:

$$P^\star = \begin{bmatrix} P^\star_{G1} \\ P^\star_{G2} \\ P^\star_{G3} \\ P^\star_{G4} \end{bmatrix} = \begin{bmatrix} 200 \\ 50 \\ 300 \\ 10 \end{bmatrix} \tag{2}$$

Based on the findings of Ahlqvist, Holmberg, and Tangerås (2022) the economic dispatch is a relatively centralized concept because each market participant, e.g., generators in the example above, has to share its detailed cost data and its capacity limits with the intermediary that coordinates the economic dispatch which is in most cases the TSO.

2.2 Centralized Optimal Power Flow

The problem formulation in equation (1a) does not account for any transportation of electricity between locations or nodes in a network. Thus, it would be desirable to include transmission lines into the modeling framework to better simulate electricity transport and, hence, a real-life system. Most of the power systems worldwide rely on power distribution with Alternating Current (AC) (Mieth 2021). AC is characterized by harmonic functions for voltage and current, resulting in a complex power that is different from the power of a Direct Current (DC) network. A good explanation of the integration of AC into an economic dispatch problem can be found in Weinhold (2022). The subsequent derivation is based on this source.

Equations (3a) to (3c) describe the complex power S_n, the active power P_n and the reactive power Q_n injected at node n for an AC system consisting of nodes that are connected by transmission lines.

$$S_n = \sum_{k \in \mathcal{N}} V_n V_k (\cos(\phi_n - \phi_k) + j \sin(\phi_n - \phi_k))(g_{nk} - j b_{nk}) \tag{3a}$$

$$P_n = \sum_{k \in \mathcal{N}} V_n V_k (g_{nk} \cos(\phi_n - \phi_k) + b_{nk} \sin(\phi_n - \phi_k)) \tag{3b}$$

$$Q_n = \sum_{k \in \mathcal{N}} V_n V_k (g_{nk} \sin(\phi_n - \phi_k) + b_{nk} \cos(\phi_n - \phi_k)) \tag{3c}$$

Due to the non-linear relation between the variables voltage angle and voltage magnitude, as well as active and reactive power injections at a network node, it is very complicated to integrate these equations in an optimization problem like (1a). Fortunately, there are some simplifications for techno-economic studies. Van Hertem

et al. (2006) suggests a linear approximation for the AC power flow equations above by applying the following assumptions:

1. The voltage angle differences $\phi_n - \phi_k$ are very small resulting in $\sin(\phi_n - \phi_k) = \phi_n - \phi_k$ and $\cos(\phi_n - \phi_k) = 1$
2. The resistance of high voltage transmission lines is usually lower than the reactance ($r \ll x$) and can be ignored.
3. A flat voltage profile ($V_n = 1p.u.$) exists meaning that all voltage magnitudes are close to their nominal value, e.g. 380 kV.

With these simplifications, it is possible to neglect the reactive power term and simplify the equation for the active power at node n to:

$$P_n = \sum_{k \in \mathcal{N}} b_{nk}(\phi_n - \phi_k) \tag{4}$$

The power flow on a certain transmission line l can be calculated as:

$$P_l = b_{nk}(\phi_n - \phi_k) \tag{5}$$

As explained by Weinhold (2022), these equations are usually written in matrix notation where $\mathbf{B_d}$ is a diagonal matrix with the line susceptances of each transmission line, Φ a column vector of nodal voltage angles, and $\mathbf{A}$ the incidence matrix with dimensions $\mathcal{N} \times \mathcal{L}$ describing the network architecture. In the incidence matrix, a starting node of a transmission line is marked as 1, whereas the end node of a transmission line is characterized by -1. Everything else is zero. The matrix notation then yields:

$$\mathbf{P_l} = \mathbf{B_d} \cdot \mathbf{A} \cdot \Phi = \mathbf{B_d} \cdot \Phi \tag{6a}$$

$$\mathbf{P_n} = \mathbf{A}^T\mathbf{B_d} \cdot \mathbf{A} \cdot \Phi = \mathbf{B_n} \cdot \Phi \tag{6b}$$

With the help of the above equations, one can derive a node-to-line sensitivity ($\frac{P_l}{P_n}$) that is also known as the Power Transfer Distribution Factor (PTDF).

$$PTDF = (\mathbf{B_d} \cdot \mathbf{A})(\mathbf{A}^T\mathbf{B_d} \cdot \mathbf{A})^{-1} \tag{7}$$

Due to the fact that equation (6b) is linear, matrix $\mathbf{A}^T\mathbf{B_d} \cdot \mathbf{A}$ is singular and no inverse can be formulated as required in equation (7). A reference node called slack node is introduced to solve this problem that balances all other nodal injections in

the network. The slack node is removed from the power flow equations and replaced by zero. The equation for the PTDF can be used to calculate the actual power flow on a transmission line l based on the nodal injections I_n. The slack node balances all other injections, and the direction of the flow is relative to the signs in the incidence matrix $\mathbf{A}$.

$$f = PTDF \cdot I_n \tag{8}$$

Equation (8) can be integrated in an economic dispatch while constraining the power flow by the maximum transmission line capacity $\overline{f_l}$. To conclude this section, the example mentioned in section 2.1 is extended by an optimal power flow. The generators are allocated at three nodes connected by three transmission lines. The system load is distributed between the nodes. A schema of this network can be found in figure (2). The parameters of the generators are the same as in table (1) and the transmission lines are parameterized in table (2).

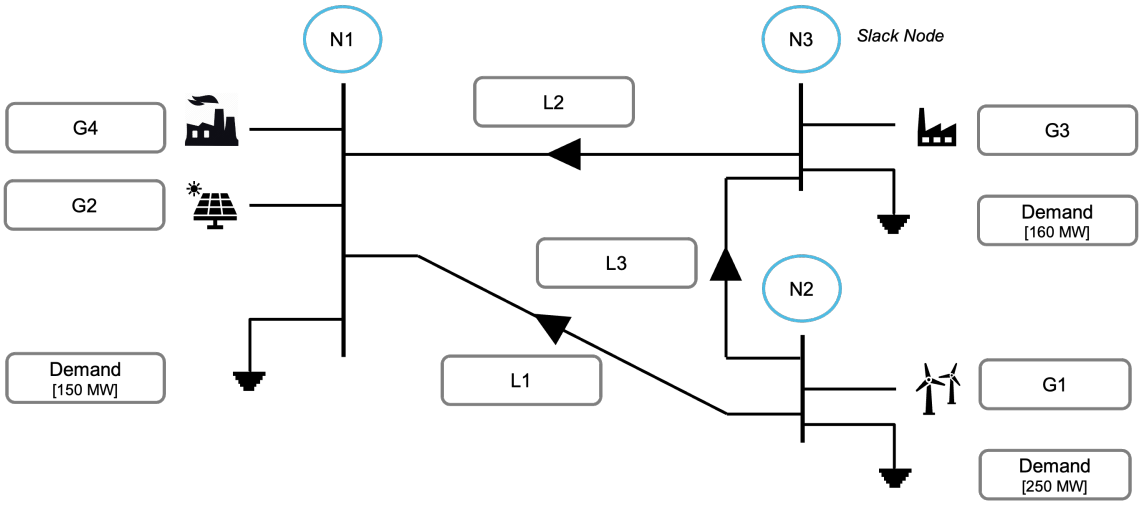

Figure 2: Network layout for an exemplary optimal power flow

Transmission Lines $\mathcal{L}$	Start Node	End Node	S [1/Ω]	$\overline{f}$ [MW]
L1	N2	N1	1	40
L2	N3	N1	1	50
L3	N2	N3	1	100

Table 2: Exemplary transmission line parameters for an optimal power flow

The optimization problem in (1a) has to be extended by an additional constraint to account for the constrained power flows on the transmission lines. The problem formulation for an optimal power flow evolves to:

$$\min \quad \sum_{g \in \mathcal{G}} mc_g \cdot P_g \tag{9a}$$

$$\text{s.t.} \quad 0 \leq P_g \leq \overline{p_g} \qquad \forall g \in \mathcal{G} \tag{9b}$$

$$\sum_{n \in \mathcal{N}} I_n = 0 \tag{9c}$$

$$I_n = \sum_{g \in \mathcal{G}_n} P_g - d_n \tag{9d}$$

$$-\overline{f_l} \leq PTDF_{l,n} \cdot I_n \leq \overline{f_l} \qquad \forall l \in \mathcal{L}, n \in \mathcal{N} \tag{9e}$$

Hereby, equation (9e) accounts for the constrained power flows on each transmission line. All power flows have to be within a lower and upper limit that is indicated by the maximum transmission line capacity $\overline{f}$. To calculate the $PTDF$, one needs the susceptance matrix $\mathbf{B_d}$ and the incidence matrix $\mathbf{A}$. Based on the example above, these matrices yield:

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \qquad \mathbf{B_d} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

Subsequently, the $PTDF$ can be calculated as:

$$PTDF = \begin{bmatrix} -0.33 & 0.33 & 0 \\ -0.66 & -0.33 & 0 \\ 0.33 & 0.66 & 0 \end{bmatrix} \tag{11}$$

In addition, constraint (9c) makes sure that each nodal demand is still satisfied. The nodal injection I_n used in this constraint is calculated by subtracting the nodal demand from the sum of all of the generators located at that node, see equation (9d) for further reference.

Generators $\mathcal{G}$	Economic Dispatch	Optimal Power Flow
G1	200 MW	200 MW
G2	50 MW	50 MW
G3	300 MW	260 MW
G4	10 MW	50 MW

Table 3: Comparison between generator results of economic dispatch and optimal power flow

Transmission Lines $\mathcal{L}$	$\overline{f}$	Economic Dispatch	Optimal Power Flow
L1	40 MW	13.33 MW	0 MW
L2	50 MW	76.66 MW	50 MW
L3	100 MW	-63.33 MW	50 MW

Table 4: Comparison between transmission line results of economic dispatch and optimal power flow

The solution of the optimal power flow problem from equation (9a) yields a different generation decision, see table (3). In comparison to the economic dispatch from section 2.1, generator G4 increases its production from 10 MW to 50 MW while at the same time generator G3 decreases its production from 300 MW to 260 MW. The power output of generators G1 and G2 remains the same. The total costs are 10 300 EUR and thus 800 EUR higher than the economic dispatch due to the higher output of generator G4. The constrained transmission lines explain the difference. As one can see in table (4), the optimal generation vector of the economic dispatch does not consider the maximum capacity of transmission line L2. Since generator G1 already produces at its maximum production capacity, the only way to satisfy the demand at node N1 is to increase the production of the most expensive generator G4. Consequently, generator G3 has to decrease its production, resulting in a less utilized transmission line L2. Subsequently, its maximum capacity is ensured.

As already written in section 2.1, the economic dispatch is a centralized concept. The same applies to the optimal power flow because there is also one coordinator who has to know the cost details of every market participant. The presented example only accounts for a spatial dimension although multiple time steps were considered. The reason here is the fact that no energy exchange is allowed between time steps. Modern electricity systems usually contain energy storage resources that enable energy exchange between time steps increasing the coordination work for the central entity by another dimension. An energy storage resource was not integrated into the optimal power flow above since the only aim of this chapter was to introduce the basics of an optimal power flow to the reader.

2.3 Alternating Direction Method of Multipliers

As written in the introduction of this thesis, there is a need to establish new algorithms and procedures to cope with the ongoing transformation of power systems worldwide. Due to the increased number of market participants, e.g., installation of even more photovoltaic system, the utilized datasets have become larger and

more sophisticated. Subsequently, centralized algorithm schemes will take longer to derive a solution. In this context, the ADMM provides a possibility to create algorithms that are well suited for these large-scale problems incorporating a lot of data points (Boyd 2010). The ADMM decomposes the main problem into multiple subproblems. The solutions to the subproblems are then coordinated and combined to derive a global optimum. It combines the methods of Dual Decomposition and Augmented Lagrangian Relaxation (ALR). Boyd (2010) provides a thorough explanation and derivation of the ADMM algorithm that was established in the mid-1970-s by Gabay, Mercier, Glowinski, and Marrocco. Another significant advantage of the ADMM is the fact that the algorithm enables decentralization and parallelization by moving from a central problem formulation to several smaller subproblems that can be solved independently. The following section equips the reader with the necessary knowledge about the ADMM to follow along with the thesis.

Typically, ADMM solves problems in the form:

$$\min\,(x, z) \quad f(x) + g(z) \tag{12a}$$

$$\text{s.t.} \quad \mathbf{A}x + \mathbf{B}z = c \tag{12b}$$

Here, $f(x)$ and $g(z)$ are two different subproblems that cannot be solved independently because of the constraint (12b) in which both decision variables are integrated. These constraints are called complicating constraints because they prevent splitting the main problem into multiple, smaller problems. With the help of ALR, it is possible to remove the complicating constraint and integrate it into the objective function. In comparison to Lagrangian Relaxation (LR), ALR adds a penalty term to the objective function. ALR is used in the first place because it works for linear as well as quadratic objective functions. See Conejo et al. (2006) for further references on LR and ALR. The corresponding Augmented Lagrangian yields

$$L_p(x, y, \lambda) = f(x) + g(z) + \lambda^T(\mathbf{A}x + \mathbf{B}z - c) + \frac{\gamma}{2}\left\|\mathbf{A}x + \mathbf{B}z - c\right\|_2^2 \tag{13}$$

where variable λ is the dual variable of the constraint that was relaxed and $\frac{\gamma}{2}\left\|\mathbf{A}x + \mathbf{B}z - c\right\|_2^2$ is the penalty term introduced by ALR. Hereby, γ is a damping parameter and the only parameter needed for the ADMM. As for the optimal solution, the penalty term becomes zero. Since the complicating constraint is removed, the function in equation (13) could be split. However, the penalty term prevents this due to the second-order norm and the product of the decision variables within.

At this point, the ADMM comes into action. It provides a possibility to solve an Augmented Lagrangian by fixing the decision variables to values obtained in the previous iteration of the algorithm. Equation (13) can then be transformed and decomposed into two subproblems in which variable v indicates the iteration.

$$x^{v+1} := \min(x) \quad f(x) + g(z^v) + (\lambda^v)^T(\mathbf{A}x + \mathbf{B}z^v - c) + \frac{\gamma}{2}\left\|\mathbf{A}x + \mathbf{B}z^v - c\right\|_2^2 \quad (14)$$

$$z^{v+1} := \min(z) \quad f(x^v) + g(z) + (\lambda^v)^T(\mathbf{A}x^v + \mathbf{B}z - c) + \frac{\gamma}{2}\left\|\mathbf{A}x^v + \mathbf{B}z - c\right\|_2^2 \quad (15)$$

In equation (14), decision variable z is fixed and becomes a normal parameter. Thus, the optimal value of x of the next iteration can be obtained by simply solving a minimization problem with one decision variable. The same applies for equation (15) and decision variable z. After solving each subproblem and obtaining the decision variables for the next iteration, the dual variable is updated with the following equation:

$$\lambda^{v+1} := \lambda^v + \gamma(\mathbf{A}x^{v+1} + \mathbf{B}z^{v+1} - c) \quad (16)$$

As long as the dual variable does not converge, the next iteration is started. The dual variable is converged if the difference between the current and previous value is smaller than a defined threshold. Boyd (2010) provide a detailed proof of convergence that will be not covered in this thesis.

2.4 Current State of Research

To analyze the current state of research, different literature databases, e.g., IEEE, were searched for studies that investigate a decentralized economic dispatch or a decentralized optimal power flow based on the ADMM algorithm. It was found that there were only a few studies that utilize the ADMM to solve problems in the context of decentralized electricity markets. Out of those, only two papers that follow a similar approach and methodology as this thesis could be identified. These studies are presented in the following section.

Xing et al. (2017) study a dynamic economic dispatch problem with energy storage resources in the context of smart grids. They implement an optimization model that includes physical constraints for generators and storages. In addition, they also

consider a system spinning reserve. The energy storage resources are included to account for inter-temporal energy arbitrage, reduce the costs of the generators, and improve the spinning reserve. The presented modeling framework does not deal with constrained transmission lines and the corresponding power flows. Xing et al. (2017) implement a distributed algorithm based on a consensus-like iterative algorithm and the ADMM. A communication network is established to connect the single elements within the network and enable a decentralized schema. According to the authors of this study, the implemented algorithm is distributed and thus decentralized because no coordinator is needed, and all computation is done locally by the network nodes. The authors evaluate the capability and effectiveness of their algorithm on the IEEE 14-bus network and could determine a convergence after around 15 iterations. Although the mathematical derivation of the distributed algorithm is described in the paper, the study lacks the implementation in any programming language. There is no reference to a published open-source package or a reference to a public repository containing the implementation.

The second paper establishes a mathematical framework for a decentralized distribution market that is based on the centralized nodal pricing model (Yang et al. 2019). The established market is fully decentralized and does not need any coordination from a central entity. The authors use the ADMM to formulate a decentralized algorithm and guarantee the convergence when choosing the correct parameter. Based on the formulation of a centralized market schema, the formulations for a decentralized market are derived. The corresponding mathematical transformations sometimes lack a detailed description, so it is difficult for the reader to follow along. The modeling framework of the study accounts for producers and consumers and power flows on transmission lines. Subsequently, the decentralized distribution market includes an optimal power flow calculation. Since the ADMM can only cope with equality constraints, they transform the inequality power flow constraints into equality constraints by introducing slack variables for the upper and lower power flow capacity limit. However, Yang et al. (2019) do not integrate any energy storage resource. Thus, their modeling framework does not reflect current technologies and developments in power systems worldwide. The developed algorithm is tested on a 33-bus and 69-bus system to demonstrate feasibility and efficiency. Yang et al. (2019) provide no references to any software implementation of the decentralized distribution market and the related algorithm.

Both studies use the ADMM to establish and develop a decentralized algorithm

and present a good starting point for decentralized electricity markets. However, none of the studies meets the desired demands. Neither of the studies accounts for both a transmission network and new technologies like energy storage resources. Secondly, the transformation process from a centralized to a decentralized schema is often not explained in great detail. Finally, no study published their software implementation as an open-source package or as a link to a public repository.

3 Application

As already mentioned in the previous chapter, the majority of the analyzed papers utilizing ADMM to solve economic dispatch problems or optimal power flows in a decentralized manner do so without a thorough description of neither the mathematical formulations nor the implementation of the latter. The following chapter describes in detail the derivation of the decentralized optimal power flow problem to tackle this issue. First, the reader is introduced to the modeling framework. Then, the transformation of the centralized problem to the decentralized problem formulation is explained. Lastly, the implementation of the optimization problem in Julia is presented.

3.1 Modeling Framework

This section introduces the reader to all relevant information about the modeling framework used to model a real-life application. The modeling framework has to cover specific requirements. Firstly, there should be generators that produce electrical power. The maximum capacity of the generator limits the power output. There exist no ramping constraints or minimum up- and downtimes. Each generator has marginal costs and is connected to a specific node. To depict these requirements in Julia, a generator structure was implemented, as one can see in the following code snippet. In addition to the stated requirements, the structure also contains a color and a name. These attributes are mainly used for evaluation reasons and are not crucial for modeling purposes.

Listing 1: Structure for a node element

```
mutable struct Generator
    name::String
    marginal_costs::Int
    max_generation::Int
    plot_color::String
    node::Node
end
```

Another part of the modeling framework is a storage unit. Storages should be able to store a surplus of energy and deliver electrical power by discharging their capacity. The power input and output are also limited. In addition, a storage unit should only be able to store a certain amount of energy. The energy storage level depicts this. Like generators, storages have specific marginal costs for charging and discharging and are connected to a particular node. A Julia structure was implemented respectively. As written before, generators and storages are located at

network nodes. These nodes have a specific demand that the output of the generating units should match. It is also crucial to define a slack node to compensate for the power flow balance for modeling purposes. The node structure was implemented as follows:

Listing 2: Structure for a generator element

```
mutable struct Node
    name::String
    demand::Vector{Int}
    slack::Bool
end
```

The last element of the framework is a transmission line. This element connects nodes and enables the exchange of energy. Transmission lines are characterized by their maximum capacity and their susceptance value. A Julia structure was also implemented for this network element. Lastly, the modeling framework has to consider multi-periods. This is crucial since it would not make sense to include storages in the system without multiple time steps. All structures are defined in `src\structures\network_elements.jl` and can be found in A.1.1.

3.2 Mathematical Formulations

This thesis aims to develop a framework to minimize the total system costs of an electrical network in a decentralized manner. This means that the costs are optimized without a central entity. Thus, the generating units can derive their own optimal decisions without releasing sensitive information like their marginal costs. This section presents the mathematical formulations of the optimization problem. First, the centralized optimization model is introduced since it forms the basis for setting up the decentralized optimization problem.

3.2.1 Centralized Optimization Problem

The total system costs of an electrical network are the sum of generation costs of all participating units over all time steps. Concerning the modeling framework, the system costs consist of the generator costs and the storage costs. The centralized optimization problem can be formulated as follows:

$$\min \quad \sum_{t\in\mathcal{T}} \sum_{g\in\mathcal{G}} mc_g \cdot P_{g,t} + \sum_{s\in\mathcal{S}} mc_s \cdot (D_{s,t} + C_{s,t}) \tag{17a}$$

$$\text{s.t.} \quad 0 \leq P_{g,t} \leq \overline{p_g} \qquad \forall g \in \mathcal{G}, t \in \mathcal{T} \tag{17b}$$

$$0 \leq D_{s,t} \leq \overline{p_s} \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{17c}$$

$$0 \leq C_{s,t} \leq \overline{p_s} \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{17d}$$

$$0 \leq E_{s,t} \leq \overline{e_s} \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{17e}$$

$$E_{s,t} - E_{s,t-1} - C_{s,t} + D_{s,t} = 0 \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{17f}$$

$$\sum_{n\in\mathcal{N}} I_{n,t} = 0 \qquad \forall t \in \mathcal{T} \tag{17g}$$

$$I_{n,t} = \sum_{g\in\mathcal{G}_n} P_{g,t} + \sum_{s\in\mathcal{S}_n} (D_{s,t} - C_{s,t}) - d_{n,t} \tag{17h}$$

$$-\overline{f_l} \leq PTDF_{l,n} \cdot I_{n,t} \leq \overline{f_l} \qquad \forall l \in \mathcal{L}, t \in \mathcal{T}, n \in \mathcal{N} \tag{17i}$$

Hereby, $\mathcal{G}$ represents the set of generators and $\mathcal{S}$ the set of storage respectively. $\mathcal{T}$ includes all considered time steps. The optimization problem is constrained to meet the requirements described in section (3.1). First of all, equation (17b) limits the maximum output of all generators where $\overline{p_g}$ is the maximum power generation of the considered generator. Equations (17c) and (17d) set the power boundaries of all storages respectively. Equation (17e) ensures that the energy storage level of all storages remains between zero and the maximum energy storage level $\overline{e_s}$. In addition, this constraint prevents the storage from discharging more power than energy is stored. The energy level of a storage s at a specific time step t can be calculated as follows:

$$E_{s,t} = E_{s,t-1} + C_{s,t} - D_{s,t} \tag{18}$$

To get a consistent form of all constraints, equation (18) is rearranged to equal zero and integrated in the optimization problem as equation (17f). $C_{s,t}$ denotes the charging power of a storage s at a time step t and $D_{s,t}$ the discharging power. Another requirement of the previous section was that the demand at every node and every time step $d_{n,t}$ is met by the generating units. This is the so-called system balance. Because the system allows line flows on the transmission lines, the production at a node may exceed the nodal demand. Thus, to ensure that all nodal demands are met, the sum of all nodal injections has to be zero, see equation (17g). The injection at a node n and at time step t yields equation (17h). Here, $\mathcal{G}_n$ ($\mathcal{S}_n$)

represents a subset of all generators (storages) that are located at node n. Hence, the injection is the sum of the power values of all generators and storages at that node subtracted by the nodal demand. This injection term is also used in the last constraint (17i) that establishes the network constraints. Here, the line flows on each transmission line are limited by the maximum line capacity $\overline{f_l}$. The line flow of a transmission line l is calculated by multiplying the power distribution factors of that line with the injection term $I_{n,t}$ of a specific node n.

To solve the optimization problem, all relevant information must be shared with the central entity. In the example above, this includes marginal costs and the maximal capacities of the generating units. The central entity can then derive the optimal dispatch plan that minimizes the total system costs and informs the generating units about their generation schedule. Thus, the generating units cannot optimize their decision on their own because the central entity is needed for coordination.

3.2.2 Decentralized Optimization Problem

The generating units in the centralized optimization problem are not able to optimize on their own because a central entity is needed to coordinate the system constraints (17g) and (17i). These are complicating constraints and prevent that the solution is obtained by optimizing per generating unit because both include all decision variables of the optimization problem. If resolved, it would be possible to derive the optimal dispatch plan per generating unit without sharing sensitive information like marginal costs. As written in 2.3, the ADMM provides a technique to resolve these constraints, making the problem decomposable and thus enabling the path to a decentralized mechanism. ADMM is only working for equality constraints. Thus, the power flow inequality constraint (17i) is transformed into two equality constraints by introducing two slack variables $U_{l,t}$ and $K_{l,t}$ for each transmission line l and time step t. The optimization problem evolves to the following:

$$\min \quad \sum_{t\in\mathcal{T}} \sum_{g\in\mathcal{G}} mc_g \cdot P_{g,t} + \sum_{s\in\mathcal{S}} mc_s \cdot (D_{s,t} + C_{s,t}) \tag{19a}$$

$$\text{s.t.} \quad 0 \leq P_{g,t} \leq \overline{p_g} \qquad \forall g \in \mathcal{G}, t \in \mathcal{T} \tag{19b}$$

$$0 \leq D_{s,t} \leq \overline{p_s} \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{19c}$$

$$0 \leq C_{s,t} \leq \overline{p_s} \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{19d}$$

$$0 \leq E_{s,t} \leq \overline{e_s} \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{19e}$$

$$E_{s,t} - E_{s,t-1} - C_{s,t} + D_{s,t} = 0 \qquad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{19f}$$

$$\sum_{n\in\mathcal{N}} I_{n,t} = 0 \qquad \forall t \in \mathcal{T} \tag{19g}$$

$$\sum_{n\in\mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U_{l,t} - \overline{f_l} = 0 \qquad \forall l \in \mathcal{L}, t \in \mathcal{T} \tag{19h}$$

$$\sum_{n\in\mathcal{N}} K_{l,t} - PTDF_{l,n} \cdot I_{n,t} - \overline{f_l} = 0 \qquad \forall l \in \mathcal{L}, t \in \mathcal{T} \tag{19i}$$

The slack variable $U_{l,t}$ ensures that the upper bound of the transmission line capacity is kept, while the slack variable $K_{l,t}$ ensures the lower limit. Both variables are added as additional decision variables. Instead of two complicating constraints, one has to handle three complicating constraints. If these constraints are relaxed, the main problem decomposes into a generator and a storage subproblem.

Relaxing the complicating variables is done by implementing a max-min problem using the dual variables of the complicated constraints. Hereby, λ is the dual of the energy balance constraint, and μ and ρ are the duals of the upper and lower power flow constraint, respectively. Since the objective function is a linear cost function, LR is not applicable. Thus, the complicated constraints are relaxed by applying ALR. A penalty term for each dual variable is introduced. The penalty terms equal zero in the optimality point and contribute to faster convergence. The optimization problem of (19a) derives to:

$$\max(\lambda, \mu, \rho) \tag{20a}$$

$$\begin{aligned}\min \quad & \sum_{t\in\mathcal{T}} \sum_{g\in\mathcal{G}} mc_g \cdot P_{g,t} + \sum_{s\in\mathcal{S}} mc_s \cdot (D_{s,t} + C_{s,t}) \\ & + \lambda_t \cdot \sum_{n\in\mathcal{N}} I_{n,t} \\ & + \frac{\gamma}{2} \cdot \Big\| \sum_{n\in\mathcal{N}} I_{n,t} \Big\|_2^2 \\ & + \sum_{l\in\mathcal{L}} \mu_{l,t} \cdot (\sum_{n\in\mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U_{l,t} - \overline{f_l}) \\ & \qquad + \frac{\gamma}{2} \cdot \Big\| \sum_{n\in\mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U_{l,t} - \overline{f_l} \Big\|_2^2 \\ & \qquad + \rho_{l,t} \cdot (\sum_{n\in\mathcal{N}} K_{l,t} - PTDF_{l,n} \cdot I_{n,t} - \overline{f_l}) \\ & \qquad + \frac{\gamma}{2} \cdot \Big\| \sum_{n\in\mathcal{N}} K_{l,t} - PTDF_{l,n} \cdot I_{n,t} - \overline{f_l} \Big\|_2^2\end{aligned}$$

$$\begin{aligned}\text{s.t.} \quad & 0 \le P_{g,t} \le \overline{p_g} && \forall g \in \mathcal{G}, t \in \mathcal{T} && \text{(20b)} \\ & 0 \le D_{s,t} \le \overline{p_s} && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(20c)} \\ & 0 \le C_{s,t} \le \overline{p_s} && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(20d)} \\ & 0 \le E_{s,t} \le \overline{e_s} && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(20e)} \\ & E_{s,t} - E_{s,t-1} - C_{s,t} + D_{s,t} = 0 && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(20f)}\end{aligned}$$

To be consistent with other literature that covers algorithms based on ADMM, the penalty terms for the complicating constraints are written in a second-order norm[1]. This formulation is unnecessary for the equation above because only scalars are used. Thus, the second-order norm could be ignored and replaced with a simple quadratic term. However, due to the second-order norm of the penalty terms of the relaxed complicating constraints, the first derivative of the objective function is not fixed anymore but continuous. This is also a necessary condition for the convergence of the algorithm. Still, the optimization problem in equation (20a) is not decomposable because of the introduced dual variables. The problem is further relaxed by fixing the dual variable to given values $\overline{\lambda}$, $\overline{\mu}$, $\overline{\rho}$. The problem formulation becomes:

[1] The norm formulation $\|.\|_2^2$ can be understood as the quadratic sum of the entries of the penalty term vector.

$$
\begin{aligned}
\min \quad & \sum_{t\in\mathcal{T}} \quad \sum_{g\in\mathcal{G}} mc_g \cdot P_{g,t} + \sum_{s\in\mathcal{S}} mc_s \cdot (D_{s,t} + C_{s,t}) && \text{(21a)}\\
& \quad + \overline{\lambda}_t \cdot \sum_{n\in\mathcal{N}} I_{n,t} \\
& \quad + \frac{\gamma}{2} \cdot \Big\| \sum_{n\in\mathcal{N}} I_{n,t} \Big\|_2^2 \\
& \quad + \sum_{l\in\mathcal{L}} \quad \overline{\mu_{l,t}} \cdot (\sum_{n\in\mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U_{l,t} - \overline{f_l}) \\
& \qquad\quad + \frac{\gamma}{2} \cdot \Big\| \sum_{n\in\mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U_{l,t} - \overline{f_l} \Big\|_2^2 \\
& \qquad\quad + \overline{\rho_{l,t}} \cdot (\sum_{n\in\mathcal{N}} K_{l,t} - PTDF_{l,n} \cdot I_{n,t} - \overline{f_l}) \\
& \qquad\quad + \frac{\gamma}{2} \cdot \Big\| \sum_{n\in\mathcal{N}} K_{l,t} - PTDF_{l,n} \cdot I_{n,t} - \overline{f_l} \Big\|_2^2
\end{aligned}
$$

$$
\begin{aligned}
\text{s.t.} \quad & 0 \le P_{g,t} \le \overline{p_g} && \forall g \in \mathcal{G}, t \in \mathcal{T} && \text{(21b)}\\
& 0 \le D_{s,t} \le \overline{p_s} && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(21c)}\\
& 0 \le C_{s,t} \le \overline{p_s} && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(21d)}\\
& 0 \le E_{s,t} \le \overline{e_s} && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(21e)}\\
& E_{s,t} - E_{s,t-1} - C_{s,t} + D_{s,t} = 0 && \forall s \in \mathcal{S}, t \in \mathcal{T} && \text{(21f)}
\end{aligned}
$$

Because of the product of the decision variable $P_{\mathcal{G},\mathcal{T}}$, $D_{\mathcal{S},\mathcal{T}}$ and $C_{\mathcal{S},\mathcal{T}}$ in all penalty terms, the optimization model is still not decomposable. At this step, the ADMM comes into action and provides a possibility to solve the problem in a decomposable way. As written in 2.3, ADMM fixes each decision variable to the previous iteration value. Thus, one is finally able to derive the generator and storage subproblem. Out of simplicity, the over-line symbol on the dual variables is omitted in the next equations.

Generator Subproblem

The optimization model for a specific generator g is derived in the following. The problem is only optimized by the decision variable $P_{g,\mathcal{T}}$ of this generator and the corresponding slack variables of the power flow constraint. With the help of ADMM, decision variables $D_{\mathcal{S},\mathcal{T}}$ and $C_{\mathcal{S},\mathcal{T}}$ are fixed to the values of the previous iteration. The power output of all other generators $P_{g'\in\mathcal{G}\setminus g,\mathcal{T}}$ is fixed too. Any constraint related to these variables can be removed. Another parameter, v, is added to the problem

formulation, indicating the iteration number. Obviously, for decision variable $P_{g,\mathcal{T}}$ one takes the current iteration number v while for the storages and other generators, one takes the previous iteration $v-1$. The first derivative of the objective function shows that the objective function can be further simplified by removing terms related to $P_{g' \in \mathcal{G} \setminus g, \mathcal{T}}$, $D_{\mathcal{S},\mathcal{T}}$ and $C_{\mathcal{S},\mathcal{T}}$. The penalty terms can not be further simplified because of the second-order norm. The optimization model for a single generator yields:

$$
\begin{aligned}
\min \quad & \sum_{t \in \mathcal{T}} \quad mc_g \cdot P_{g,t}^{v} + \lambda_t^{v} \cdot P_{g,t}^{v} && \text{(22a)} \\
& \quad + \frac{\gamma}{2} \cdot \Big\| \sum_{n \in \mathcal{N}} I_{n,t} \Big\|_2^2 \\
& \quad + \sum_{l \in \mathcal{L}} \quad \mu_{l,t}^{v} \cdot \sum_{n \in \mathcal{N}} PTDF_{l,n} \cdot P_{g,t}^{v} \\
& \qquad\qquad + \frac{\gamma}{2} \cdot \Big\| \sum_{n \in \mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U_{l,t}^{v} - \overline{f_l} \Big\|_2^2 \\
& \qquad\qquad - \rho_{l,t}^{v} \cdot \sum_{n \in \mathcal{N}} PTDF_{l,n} \cdot P_{g,t}^{v} \\
& \qquad\qquad + \frac{\gamma}{2} \cdot \Big\| \sum_{n \in \mathcal{N}} K_{l,t}^{v} - PTDF_{l,n} \cdot I_{n,t} - \overline{f_l} \Big\|_2^2 \\
\text{s.t.} \quad & 0 \leq P_{g,t}^{v} \leq \overline{p_g} && \forall t \in \mathcal{T} \qquad \text{(22b)}
\end{aligned}
$$

The term inside of the brackets in the second line of equation (22a) is once again the system balance constraint. However, the power output of the other generators and storages is not a variable anymore but a parameter that is fixed to the previous iteration values. Hence, the power output of the specific generator $P_{g,t}$ can be excluded from the sum. The injection of node $\check{n}$ at which the generator is located can be then formulated as:

$$
I_{\check{n},t} = P_{g,t}^{v} + \sum_{g' \in \mathcal{G}_{\check{n}} \setminus g} p_{g',t}^{v-1} + \sum_{s \in \mathcal{S}_{\check{n}}} (d_{s,t}^{v-1} - c_{s,t}^{v-1}) - d_{\check{n},t} \tag{23}
$$

The power generation terms of the other units are written in lower case to match the notation of parameters. Be aware that $d_{s,t}^{v-1}$ and $d_{\check{n},t}$ are not the same. The first term depicts the discharge power output of a storage at node $\check{n}$. The latter is the nodal demand of node $\check{n}$. Equation (23) can be then inserted in the sum of all nodal injections for a specific time step t:

$$
\sum_{n \in \mathcal{N}} I_{n,t} = I_{\check{n},t} + \sum_{n \in \mathcal{N} \setminus \check{n}} \quad \sum_{g \in \mathcal{G}_n} p_{g',t}^{v-1} + \sum_{s \in \mathcal{S}_n} (d_{s,t}^{v-1} - c_{s,t}^{v-1}) - d_{n,t} \tag{24}
$$

As for now, the specific generator needs to know the previous dispatch of all other units to compute the injection. This contradicts the aim of this thesis to create a decentralized framework in which sensitive information is not shared with other participants. That includes the power dispatch of the previous iteration. The coordinating algorithm provides the sum of nodal power outputs for the other generators and storages in the network to overcome the problem. The sum of the previous iteration power values' of all generators at a node n and a time step t is denoted as $\Theta_{n,t}$ and can be expressed as:

$$\Theta_{n,t} = \sum_{g \in \mathcal{G}_n} p_{g,t}^{v-1} \tag{25}$$

The same applies to the discharge and charge power values of the storages in the network. $\Phi_{n,t}$ depicts the sum of the previous discharge power values of all storages at node n and time step t and $\Psi_{n,t}$ depicts the charging sum respectively. The corresponding equations are as follows:

$$\Phi_{n,t} = \sum_{s \in \mathcal{S}_n} d_{s,t}^{v-1} \tag{26}$$

$$\Psi_{n,t} = \sum_{s \in \mathcal{S}_n} c_{s,t}^{v-1} \tag{27}$$

Using these values prevents the fact that the specific generators need to know the other participants' actual power dispatch. Since the power output of the particular generator g is also included in $\Theta_{\check{n},t}$, one has to subtract the power output from the previous iteration of the specific generator from the sum. Thus, the sum of all nodal injections for the particular generator g at node $\check{n}$ becomes:

$$\begin{aligned} \sum_{n \in \mathcal{N}} I_{n,t} = P_{g,t}^{v} + \Theta_{\check{n},t} - P_{g,t}^{v-1} + \Phi_{\check{n},t} - \Psi_{\check{n},t} - d_{\check{n},t} \\ + \sum_{n \in \mathcal{N} \setminus \check{n}} \Theta_{n,t} + \Phi_{n,t} - \Psi_{n,t} - d_{n,t} \end{aligned} \tag{28}$$

The optimization model in (22a) is further rearranged by summarizing the terms including the dual variables $\rho_{l,t}$ and $\mu_{l,t}$. In addition, penalty terms for the decision variables are introduced that become zero in the optimality point and enable a faster convergence. The penalty term for the decision variable $P_{g,t}$ calculates the difference between the current iteration value and the previous iteration value. Since the slack variables of the power flow constraints have to be the same for all subproblems

at the global optimum, the penalty terms for $U_{l,t}$ and $K_{l,t}$ consider the average of all subproblems of the previous iteration. The average value for the upper flow constraint is denoted as $\Upsilon_{l,t}$ and $\Gamma_{l,t}$ respectively. Finally, the generator subproblem becomes:

$$
\begin{aligned}
\min \quad & \sum_{t\in\mathcal{T}} \quad mc_g \cdot P^v_{g,t} && && \text{(29a)}\\
& \quad + P^v_{g,t} \cdot (\lambda^v_t + \sum_{l\in\mathcal{L}}\sum_{n\in\mathcal{N}} (\mu^v_{l,t} - \rho^v_{l,t}) \cdot PTDF_{l,n}) \\
& \quad + \frac{\gamma}{2} \cdot \big\| P^v_{g,t} + \Theta_{\check{n},t} - P^{v-1}_{g,t} + \Phi_{\check{n},t} - \Psi_{\check{n},t} - d_{\check{n},t} \\
& \qquad\quad + \sum_{n\in\mathcal{N}\setminus\check{n}} \Theta_{n,t} + \Phi_{n,t} - \Psi_{n,t} - d_{n,t} \big\|_2^2 \\
& \quad + \sum_{l\in\mathcal{L}} \quad \frac{\gamma}{2} \cdot \big\| \sum_{n\in\mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U^v_{l,t} - \overline{f_l} \big\|_2^2 \\
& \qquad\qquad + \frac{\gamma}{2} \cdot \big\| \sum_{n\in\mathcal{N}} K^v_{l,t} - PTDF_{l,n} \cdot I_{n,t} - \overline{f_l} \big\|_2^2 \\
& \qquad\qquad + \frac{\gamma}{2} \cdot \big\| U^v_{l,t} - \Upsilon_{l,t} \big\|_2^2 \\
& \qquad\qquad + \frac{\gamma}{2} \cdot \big\| K^v_{l,t} - \Gamma_{l,t} \big\|_2^2 \\
& \quad + \frac{\gamma}{2} \cdot (P^v_{g,t} - P^{v-1}_{g,t})^2 \\
\text{s.t.} \quad & 0 \le P^v_{g,t} \le \overline{p_g} && \forall t \in \mathcal{T} && \text{(29b)}
\end{aligned}
$$

Storage Subproblem

This section derives the optimization model for a specific storage s. The problem only includes decision variables $D_{s,t}$, $C_{s,t}$ of the storage s and the corresponding slack variables of the power flow constraints. As for the generator subproblem, all other decision variables from the main optimization model are fixed to the values of the previous iteration. Thus, constraints related to other generators or storages can be removed.

The derivation of the optimization model for the storage subproblem is nearly the same as for the generator. Therefore, a description of the derivation is omitted, and the reader is referred to the generator subproblem for further detail. If the previous iteration values for the generators, other storages and the penalty terms are added, the storage subproblem yields:

$$
\begin{aligned}
\min \quad \sum_{t\in\mathcal{T}} \quad & mc_s \cdot (D^v_{s,t} + C^v_{s,t}) && \text{(30a)}\\
& + (D^v_{s,t} - C^v_{s,t}) \cdot (\lambda^v_t + \sum_{l\in\mathcal{L}} \sum_{n\in\mathcal{N}} (\mu^v_{l,t} - \rho^v_{l,t}) \cdot PTDF_{l,n})\\
& + \frac{\gamma}{2} \cdot \big\| \Theta_{\check{n},t} + D^v_{s,t} + \Phi_{\check{n},t} - D^{v-1}_{s,t}\\
& \qquad - C^v_{s,t} - (\Psi_{\check{n},t} - C^{v-1}_{s,t}) - d_{\check{n},t}\\
& \qquad + \sum_{n\in\mathcal{N}\setminus\check{n}} \Theta_{n,t} + \Phi_{n,t} - \Psi_{n,t} - d_{n,t} \big\|^2_2\\
& + \sum_{l\in\mathcal{L}} \quad \frac{\gamma}{2} \cdot \big\| \sum_{n\in\mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + U^v_{l,t} - \overline{f_l} \big\|^2_2\\
& \qquad + \frac{\gamma}{2} \cdot \big\| \sum_{n\in\mathcal{N}} K^v_{l,t} - PTDF \cdot I_{n,t} - \overline{K_l} \big\|^2_2\\
& \qquad + \frac{\gamma}{2} \cdot \big\| U^v_{l,t} - \Upsilon_{l,t} \big\|^2_2\\
& \qquad + \frac{\gamma}{2} \cdot \big\| K^v_{l,t} - \Gamma_{l,t} \big\|^2_2\\
& + \frac{\gamma}{2} \cdot (D^v_{s,t} - D^{v-1}_{s,t})^2\\
& + \frac{\gamma}{2} \cdot (C^v_{s,t} - C^{v-1}_{s,t})^2
\end{aligned}
$$

$$
\begin{aligned}
\text{s.t.} \quad & 0 \le D^v_{s,t} \le \overline{p_s} && \forall t \in \mathcal{T} \quad \text{(30b)}\\
& 0 \le C^v_{s,t} \le \overline{p_s} && \forall t \in \mathcal{T} \quad \text{(30c)}\\
& 0 \le E^v_{s,t} \le \overline{e_s} && \forall t \in \mathcal{T} \quad \text{(30d)}\\
& E^v_{s,t} - E^v_{s,t-1} - C^v_{s,t} + D^v_{s,t} = 0 && \forall t \in \mathcal{T} \quad \text{(30e)}
\end{aligned}
$$

Update Dual Variables

After solving every subproblem, the dual variables of the complicating constraints have to be updated as well. The following equations describe the update process of each dual variable in the problem for one time step. In the case of the dual variables for the power flow constraints, the equations are based on a single time step and a single transmission line.

$$\lambda_t^{v+1} = \lambda_t^v + \gamma \cdot (\sum_{g \in \mathcal{G}} P_{g,t}^v + \sum_{s \in \mathcal{S}} D_{s,t}^v - \sum_{s \in \mathcal{S}} D_{s,t}^v - d_{n,t}) \tag{31a}$$

$$\mu_{l,t}^{v+1} = \mu_{l,t}^v + \gamma \cdot (\overline{\Gamma_{l,t}^v} - \sum_{n \in \mathcal{N}} PTDF_{l,n} \cdot I_{n,t} - \overline{f_l}) \tag{31b}$$

$$\rho_{l,t}^{v+1} = \rho_{l,t}^v + \gamma \cdot (\sum_{n \in \mathcal{N}} PTDF_{l,n} \cdot I_{n,t} + \overline{\Upsilon_{l,t}^v} - \overline{f_l}) \tag{31c}$$

In a decentralized framework, this task would be orchestrated by the coordinator. After updating the duals, the coordinator examines the convergence status of the dual variables. A dual variable is converged if the difference between the next and the current iteration is smaller than a predefined threshold ϵ. The overall algorithm converges if all dual variables are converged. Then, an optimal solution for each subproblem could be found and thus an optimal solutions for the global problem. If a single, dual variable is not converged, the next iteration is started utilizing the updated dual variables. The coordinator communicates the updated duals and the last iteration's nodal information to all generators and storages.

3.2.3 Matrix Notation

It is common to formulate optimization problems in a matrix notation. With this notation, it is much easier to understand multidimensional optimization problems. As for this thesis, several dimensions like time, nodes, and lines are included. Thus, it would be beneficial for the reader to refer to the matrix notation if anything is unclear. In addition, Boyd (2010) also uses matrix notation to explain the ADMM. As already presented, ADMM solves optimization problems in the form of:

$$\min (x, z) \quad f(x) + g(z) \tag{32a}$$

$$\text{s.t.} \quad \mathbf{A}x + \mathbf{B}z = c \tag{32b}$$

If applied to the formulations of the previous sections, $f(x)$ would be the generator costs and $g(z)$ the storage costs. The problem in (32a) would be extended by the corresponding constraints for generators and storages. In addition, the power flow constraints would be added as another system constraint like equation (32b). The matrix notation can also be decomposed by applying the steps described in section (3.2.2).

According to Boyd (2010), ADMM is often written in a shorter, so-called scaled form. In this form, the linear and quadratic terms of the objective function are combined, and the dual variables are scaled. This yields a much shorter formulation. As an example, the scaled dual variable of the energy balance constraint is derived. First, one defines a residual term of the energy balance constraint.

$$\mathbf{r} = \mathbf{N}_{\mathcal{G}} * \mathbf{P} + \mathbf{N}_{\mathcal{S}} * (\mathbf{D} - \mathbf{C}) - \mathbf{d} = \mathbf{I} \tag{33}$$

Here, $\mathbf{N}_{\mathcal{G}}$ and $\mathbf{N}_{\mathcal{S}}$ are the identity matrices indicating the node location of every generator and storage in the network. Inserting the residual term r into the corresponding part of the optimization problem yields:

$$\lambda * \mathbf{r} + \frac{\gamma}{2} * ||\mathbf{r}||_2^2 = \frac{\gamma}{2}||\mathbf{r} + \frac{1}{\gamma}\lambda||_2^2 - \frac{1}{2\gamma}||\lambda||_2^2 = \frac{\gamma}{2}||\mathbf{r} + \hat{\lambda}||_2^2 - \frac{\gamma}{2}||\hat{\lambda}||_2^2 \tag{34}$$

The transformation is not very straightforward but allows to simplify the optimization problem further. With $\hat{\lambda} = \frac{1}{\gamma} * \lambda$, one gets the scaled dual variable of λ. Using the scaled dual variable makes the problem formulation much shorter because the last term $-\frac{\gamma}{2}||\hat{\lambda}||_2^2$ of equation (34) does not contain any optimization variable. Hence, this term can be removed. The other dual variables can be replaced, respectively. In the following, the generator subproblem and the storage subproblem are formulated using the scaled form of the ADMM.

Generator Subproblem

The generator costs in matrix notation are formulated as follows:

$$f(x) = f(\mathbf{P}) = \mathbf{P}^T \cdot \mathbf{mc} \tag{35a}$$

$$= \begin{bmatrix} P_{g_1,t_1} & \dots & P_{g_n,t_1} \\ \vdots & \ddots & \vdots \\ P_{g_1,t_n} & \dots & P_{g_n,t_n} \end{bmatrix} \cdot \begin{bmatrix} mc_{g_1} \\ \vdots \\ mc_{g_n} \end{bmatrix} \tag{35b}$$

The matrix of the power generation values is transposed to allow matrix multiplication. If this formulation is applied to the generator optimization problem of equation (29a), the matrix notation yields:

$$\min \quad \mathbf{P}^v \cdot mc \tag{36a}$$

$$+ \frac{\gamma}{2} \cdot \left\| \mathbf{P}^v + \mathbf{\Theta} - \mathbf{P}^{v-1} + \mathbf{\Phi} - \mathbf{\Psi} - \mathbf{d} + \frac{\lambda}{\gamma} \right\|_2^2 \tag{36b}$$

$$+ \frac{\gamma}{2} \cdot \left\| PTDF \cdot \mathbf{I} + \mathbf{U} - \overline{\mathbf{f}} + \frac{\mu}{\gamma} \right\|_2^2$$

$$+ \frac{\gamma}{2} \cdot \left\| \mathbf{K} - PTDF \cdot \mathbf{I} - \overline{\mathbf{f}} + \frac{\rho}{\gamma} \right\|_2^2$$

$$+ \frac{\gamma}{2} \cdot \left\| \mathbf{U} - \mathbf{\Upsilon} \right\|_2^2$$

$$+ \frac{\gamma}{2} \cdot \left\| \mathbf{K} - \mathbf{\Gamma} \right\|_2^2$$

$$+ \frac{\gamma}{2} \cdot \left\| \mathbf{P}^v - \mathbf{P}^{v-1} \right\|_2^2$$

$$\text{s.t.} \quad \mathbf{0} \leq \mathbf{P} \leq \overline{\mathbf{p}} \tag{36c}$$

Storage Subproblem

The storage costs in matrix notation are formulated as follows:

$$g(z) = g(\mathbf{D}, \mathbf{C}) \tag{37a}$$

$$= (\mathbf{D} + \mathbf{C})^T \cdot \mathbf{mc} \tag{37b}$$

$$= \left(\begin{bmatrix} D_{s_1,t_1} & \dots & D_{s_n,t_1} \\ \vdots & \ddots & \vdots \\ D_{s_1,t_n} & \dots & D_{s_n,t_n} \end{bmatrix} + \begin{bmatrix} C_{s_1,t_1} & \dots & C_{s_n,t_1} \\ \vdots & \ddots & \vdots \\ C_{s_1,t_n} & \dots & C_{s_n,t_n} \end{bmatrix} \right) \cdot \begin{bmatrix} mc_{s_1} \\ \vdots \\ mc_{s_n} \end{bmatrix} \tag{37c}$$

If this formulation is applied to the storage optimization problem of equation (30a), the matrix notation yields:

$$
\begin{aligned}
\min \quad & (\mathbf{D}^v + \mathbf{C}^v)^T \cdot mc && \text{(38a)}\\
& + \frac{\gamma}{2} \cdot \Big\| \mathbf{\Theta} + \mathbf{D}^v + \mathbf{\Phi} - \mathbf{D}^{v-1} \\
& \qquad\qquad - \mathbf{C}^v - (\mathbf{\Psi} - \mathbf{C}^{v-1}) - \mathbf{d} + \frac{\lambda}{\gamma} \Big\|_2^2 \\
& + \frac{\gamma}{2} \cdot \Big\| PTDF \cdot \mathbf{I} + \mathbf{U} - \overline{\mathbf{f}} + \frac{\mu}{\gamma} \Big\|_2^2 \\
& + \frac{\gamma}{2} \cdot \Big\| \mathbf{K} - PTDF \cdot \mathbf{I} - \overline{\mathbf{f}} + \frac{\rho}{\gamma} \Big\|_2^2 \\
& + \frac{\gamma}{2} \cdot \Big\| \mathbf{U} - \mathbf{\Upsilon} \Big\|_2^2 \\
& + \frac{\gamma}{2} \cdot \Big\| \mathbf{K} - \mathbf{\Gamma} \Big\|_2^2 \\
& + \frac{\gamma}{2} \cdot \Big\| \mathbf{D}^v - \mathbf{D}^{v-1} \Big\|_2^2 \\
& + \frac{\gamma}{2} \cdot \Big\| \mathbf{C}^v - \mathbf{C}^{v-1} \Big\|_2^2 \\
\text{s.t.} \quad & \mathbf{0} \leq \mathbf{D} \leq \overline{\mathbf{p}} && \text{(38b)}\\
& \mathbf{0} \leq \mathbf{C} \leq \overline{\mathbf{p}} && \text{(38c)}\\
& \mathbf{0} \leq \mathbf{E} \leq \overline{\mathbf{e}} && \text{(38d)}\\
& \mathbf{0} = \mathbf{E} - \mathbf{E}_{t-1} - \mathbf{C} + \mathbf{D} && \text{(38e)}
\end{aligned}
$$

Update Dual Variables

The equations to update the dual variables in matrix notation are as follows:

$$
\begin{aligned}
\lambda^{v+1} &= \lambda^v + \gamma \cdot (\mathbf{P}^v + \mathbf{D}^v - \mathbf{C}^v - \mathbf{d}) && \text{(39a)}\\
\mu^{v+1} &= \mu^v + \gamma \cdot (\overline{\mathbf{\Gamma^v}} - PTDF \cdot \mathbf{I} - \overline{\mathbf{f}}) && \text{(39b)}\\
\rho^{v+1} &= \rho^v + \gamma \cdot (PTDF \cdot \mathbf{I} + \overline{\mathbf{\Upsilon^v}} - \overline{\mathbf{f}}) && \text{(39c)}
\end{aligned}
$$

3.3 Documentation of the Implementation in Julia

This section provides the reader with a detailed description of how the previously stated optimization models are implemented in the programming language Julia[2]. Julia is used in this thesis because of the scripts' fast execution times. Speed was one of the core design patterns when Julia was developed. Execution times are more

[2] www.julialang.org

rapid in comparison to other scripting languages like Python[3] because Julia code is compiled to efficient native code with the help of the Low-Level Virtual Machine. In addition, Julia is dynamically typed. This means that the programmer can define types, accelerating the execution time even more. The language's syntax is easy to understand, making it possible for beginners to get started quickly. With over a thousand contributors, Julia has a large open-source community from which one can expect even more improvements in future times. Lastly, Julia provides a lot of interfaces to open-source and commercial optimization solvers like Gurobi[4]. These features supported the decision to implement the optimization models in Julia.

The section starts with a general overview of the code structure, including information about subdirectories and the used open-source packages. Afterward, the optimization model of the centralized approach is explained. Finally, the implementations for the decentralized optimization model are described.

3.3.1 General Information

Rather than defining everything in one Julia script, the approach was to divide the source code into separate units. This is a best practice in modern software development that allows a better understanding of the code and contributes to a more maintainable project. However, Julia is more a scripting language than an object-oriented language. Thus, one had to make adjustments to provide a structured source code.

All Julia files are placed into the directory `src` that contains several subdirectories. Each of them has a different purpose. There is the `src/cases` directory that contains the Julia scripts for setting up a case study system. This directory can contain different case studies, making it easy to switch between different scenarios. Inside of `src/helpers` several helper scripts are defined that are used while running the decentralized optimization model. This includes methods to print results, calculating the $PTDF$, or several plotting functions to visualize the optimization results. All Julia scripts related to the decentralized optimization model are located in `src/optimization`. This compromises, for example, the modeling of the subproblems or functions to check the convergence of the problem. Lastly, all defined structures that are utilized in the source code are placed in `src/structures`. The Julia structure scripts are also separated into corresponding categories, e.g., network

[3] www.python.org
[4] www.gurobi.com

elements like generators or penalty terms. There is one main script for running the centralized optimization model `src/opf_central_reference.jl` and one for the decentral optimization model using the ADMM named `src/opf_admm_decentral.jl`. Both scripts include a case study from the before mentioned directory and the Julia script `src/imports.jl` that manages all imports from the subdirectories and the needed Julia packages.

All open-source packages that are used within the project are listed in the file `Project.toml`. This file is also used to create a reproducible environment for every user. To activate the environment, open the Julia package manager in the root directory of the project and run the command `activate`. All necessary packages are then installed.

Gurobi is utilized as a solver as it is the fastest solver for optimization problems worldwide and provides with the Julia package Gurobi[5] an easy-to-use interface. The solver is used in combination with the Julia package JuMP that presents a modeling language for mathematical optimization problems and is introduced by Dunning, Huchette, and Lubin (2017). The package equips the user with several helper methods to formulate and solve optimization models. Another advantage of JuMP is that one can easily switch between solvers. Thus, it is possible to replace Gurobi with an open-source solver like Clp[6]. Lastly, the Julia package PlotlyJS is used to create interactive plots.

3.3.2 Centralized Optimization Problem

This section illustrates the implementation of the centralized optimization model used as a reference model. It presents the current state of the energy dispatch framework utilized in energy systems worldwide. The corresponding code is shown in the listing below. The line numbers on the left-hand side are referenced in the following explanation.

[5] www.juliapackages.com/p/gurobi
[6] www.github.com/coin-or/Clp

Listing 3: Implementation of the centralized optimization model

```
include("imports.jl")
include("cases/three_node.jl")

# Set up optimization model.
begin
    # Create necessary indexes for all equations.
    N = 1:length(nodes)
    S = 1:length(storages)
    G = 1:length(generators)
    L = 1:length(lines)
    T = 1:length(nodes[1].demand)

    ptdf = calculate_ptdf(nodes, lines)

    # Create model instance.
    m = Model(() ->Gurobi.Optimizer(gurobi_env))

    # Define all optimization variables.

    # Generation power of generators
    @variable(m, 0 <= P[g=G, t=T] <= generators[g].max_generation)
    # Discharge power of storages
    @variable(m, 0 <= D[s=S, t=T] <= storages[s].max_power)
    # Charge power of storages.
    @variable(m, 0 <= C[s=S, t=T] <= storages[s].max_power)
    # Energy level of storages.
    @variable(m, 0 <= E[s=S, t=T] <= storages[s].max_level)
    # Slack variable for upper flow limit.
    @variable(m, 0 <= U[l=L, t=T])
    # Slack variable for lower flow limit.
    @variable(m, 0 <= K[l=L, t=T])

    # Define expression for injection term.
    @expression(m, I[n=N, t=T],
        sum((generators[g].node == nodes[n] ? P[g,t] : 0) for g in G)
        + sum((storages[s].node == nodes[n] ? D[s,t] - C[s,t] : 0) for s in S)
        - nodes[n].demand[t]
    )

    # Set objective function.
    @objective(m, Min,
        sum(P[g, t] * generators[g].marginal_costs for g in G, t in T)
        + sum((D[s,t] + C[s,t]) * storages[s].marginal_costs for s in S, t in
            T)
    )

    # Set energy balance constraint.
    @constraint(m, EB[t=T], sum(I[:,t]) == 0)
    # Set upper flow constraint.
    @constraint(m, FlowUpper[l=L, t=T], sum(ptdf[l,n] * I[n,t] for n in N) + U
        [l,t] == lines[l].max_capacity)
    # Set lower flow constraint.
    @constraint(m, FlowLower[l=L, t=T], K[l,t] - sum(ptdf[l,n] * I[n,t] for n
```

```
        in N) == lines[l].max_capacity)
52      # Set constraint for energy level of storage.
53      @constraint(m, StorageBalance[s=S, t=T], E[s,t] == (t > 1 ? E[s,t-1] : 0)
            - D[s,t] + C[s,t])
54  end
55
56  # Solve optimization problem.
57  optimize!(m)
58
59  # Get objective value.
60  println("Objective value: $(objective_value(m))\n")
61
62  # Print generation results.
63  println("Generator results:\n$(value.(P).data)\n")
64  println("Discharge results:\n$(value.(D).data)\n")
65  println("Charge results:\n$(value.(C).data)\n")
66
67  # Print line flows.
68  println("Line utilization:\n$(ptdf * value.(I).data)\n")
69
70  # Print system price.
71  lambda = dual.(EB).data
72  println("System price:\n$(lambda)\n")
73
74  # Calculate nodal price.
75  mu = dual.(FlowUpper).data + dual.(FlowLower).data
76  nodal_price = zeros(length(N), length(T))
77  for t in T
78      nodal_price[:, t] = lambda[t] .+ sum(mu[l,t] * ptdf[l,:] for l in L)
79  end
80  # Print nodal price.
81  println("Nodal price:\n$(nodal_price)\n")
```

First, the necessary packages and Julia scripts are imported in line 1. Then the case study is included, providing all elements of the network with the relevant parameters. The code block between line 5 and 54 creates the optimization model. At the beginning of the block, helper indexes for all network elements are created. These are used in the later model equations. Afterwards, the $PTDF$ of the network is calculated using the helper function `calculate_ptdf()` that is defined in `src/helpers/ptdf.jl` and is listed in A.2.1 for further reference. As a next step, the JuMP model object `m` is initiated, and the solver is set to Gurobi in line 16. The variable `gurobi_env` is defined in the import script `src/imports.jl` and suppresses multiple Gurobi license printings. Afterward, the optimization variables are attached to the JuMP model. The created indexes are used to access the network elements stored in an array. The first dimension of all variables always represents the network element, as shown in the matrix formulation of section 3.2.3. The second dimension represents the time index. The naming convention of the optimization variables reflect the variable names in the optimization equations from section 3.2.

While declaring the variables, the corresponding bounds are set too. This reduces the number of constraints, making the optimization model more simple and faster. At this moment, one can make use of the implemented structures to access the corresponding parameters, e.g., setting the upper bound of the generator power output with the help of attribute `max_generation`. In lines `34` to `38` an expression is added to the model that calculates the injection at every timestep and node according to equation (17h) from section 3.2. The generation from either a generator or storage is included in the injection formulation of a specific time step and node, if the node is equal to the node of the generating unit. This was implemented by using the ternary operator `?:` and the attribute `node` of the network element. The ternary operator serves as a short if-else-condition. Next, the objective function of the problem is defined in line `41` to `44`. This function sums up the product of the generation decision variables and marginal costs for all network elements and time steps. Both discharging power and charging power are considered in the costs for storage network elements. The last part of the model setup introduces all constraints. Line `17` attaches the system balance constraint to the model. This constraint uses the before-defined expression for the injection term and makes sure that the sum of all node injections at a specific time step equals zero. Afterward, the upper and lower line constraint limit is implemented. As explained in the previous section, this constraint was transformed into two equality constraints by introducing two slack variables. Due to a Julia type error, the dot product between the $PTDF$ of one line and the injection term for one time step had to be calculated by summing up the single values of the dot product. Lastly, the energy level constraint of the storages is added to the model. This constraint makes sure that the current energy level consists of the energy level of the previous time step and the charging power minus the power that is withdrawn. The ternary operator is utilized again to set the initial energy level to zero for every storage in the network. In line `57`, the command is run to solve the optimization model `m`. The rest of the implementation covers the output of the optimization variables and additional information like the system price or the line utilization.

3.3.3 Decentralized Optimization Problem

In the following, the implementations for the decentralized optimization model are explained. In comparison to implementing the central problem, this involves multiple methods and files. Thus, the implementation is not described on a single file basis but instead explains the established algorithm's procedure. An example that runs a decentralized optimal power flow can be found in the file `src/opf_admm_`

`decentral.jl` and is shown in the listing below.

Listing 4: Implementation of the decentralized optimization model

```
1 include("imports.jl")
2
3 include("cases/three_node.jl")
4
5 admm = ADMM(0.3, nodes, generators, storages, lines)
6
7 run!(admm)
8
9 np = get_nodal_price(admm.iteration)
```

This is the top-level file that imports all scripts and the desired case study. Then, the `admm` object is created using the structure defined in `src/helpers/admm.jl` that is also listed in A.1.2. A custom constructor was written for this structure because the majority of the attributes are initialized with default values. The constructur also calculates the $PTDF$ that is then accessible through the corresponding attribute. See line `23` to `63` in A.1.2 for further reference. Only the network elements and the damping parameter γ are necessary for the initialization of the `admm` object. The whole optimization algorithm is started in line `7` with a single command called `run!(admm)`.

All files related to the optimization implementations are located in the directory `src/optimization`. Thus, the implementation of the `run!(...)` method is defined in the file `src/optimization/run.jl` and represents the entry point into the decentralized algorithm framework. The method is shown in listing (5). It orchestrates the whole optimization algorithm that runs as long as the algorithm is not converged. Hence, in line `2` a while loop is established that checks the convergence of the algorithm after each iteration.

Listing 5: Top-level method to orchestrate the decentralized algorithm

```
1 function run!(admm::ADMM)
2     while (!admm.convergence.all)
3         calculate_iteration!(admm)
4     end
5 end
```

The attribute `convergence` is of type `Convergence` that is defined in `src/structures/convergence.jl` and shown in A.1.3. The structure includes the current convergence status of all dual variables and their corresponding residual values. A dual variable is converged if the residual term is smaller than an error term ϵ. The attribute `all` of the `Convergence` structure is used to determine the convergence of the whole algorithm. If all duals are converged, then the algorithm is converged, and the attribute

is set to true. In line 3 of listing (5), the iteration calculation is started, which consists of three steps. First, it optimizes all subproblems and prints the results of the corresponding generation to the console. Afterwards, the dual variables λ_t^v, $\mu_{l,t}^v$ and $\rho_{l,t}^v$ are updated and the convergence of the algorithm is checked. If not converged, the algorithm continues with the next iteration. The implementation of the iteration calculation can be found in A.3.1. In the following, the before-mentioned steps are explained in more detail.

Optimize all Subproblems

The method `optimize_all_subproblems!(...)` is defined in the file `src/optimization/subproblems.jl` and can be found in A.3.2. It does two things. First, it optimizes all storage and generator subproblems and concatenates the results into a dictionary. Then, this dictionary is used to create the `result` object that is attached to the `admm` object. Both optimization methods for the subproblems are defined in `src/optimization/subproblems.jl`. Be aware that both methods can have the same name since the arguments of the methods are different. This is also a design pattern to shorten method names in Julia. The optimization method to solve a generator subproblem is covered first. The corresponding implementation is listed in (6).

Listing 6: Method to optimize a single generator subproblem

```
1  function optimize_subproblem(generator::Generator)
2      # Create model instance.
3      sub = Model(() ->Gurobi.Optimizer(gurobi_env))
4      # Omit output in console.
5      set_silent(sub)
6
7      # Generation variable with maximum bound.
8      @variable(sub, 0 <= P[t=admm.T] <= generator.max_generation)
9
10     # Slack variable for upper flow limit.
11     @variable(sub, 0 <= U[line=admm.L, t=admm.T])
12     # Slack variable for lower flow limit.
13     @variable(sub, 0 <= K[line=admm.L, t=admm.T])
14
15     # Get generator results from previous iteration.
16     prev_P = get_unit_results(generator, admm.iteration - 1)
17
18     # Get general node results from previous iteration.
19     prev_node_results = get_node_id_to_result(admm.iteration - 1)
20
21     # Define expression for injection term.
22     @expression(
23         sub,
24         injection[n=admm.N, t=admm.T],
```

```
        n == admm.node_to_id[generator.node] ? (
            P[t] + (prev_node_results[n].generation[t] - prev_P[t])
            + prev_node_results[n].discharge[t]
            - prev_node_results[n].charge[t]
            - admm.node_id_to_demand[n][t]
        ) : (
            prev_node_results[n].generation[t]
            + prev_node_results[n].discharge[t]
            - prev_node_results[n].charge[t]
            - admm.node_id_to_demand[n][t]
        )
    )

    # Add all penalty terms to the model.
    add_penalty_terms!(sub)

    # Get current node index.
    node_index = admm.node_to_id[generator.node]

    # Set objective function.
    @objective(
        sub,
        Min,
        sum(P[t] * generator.marginal_costs
            + P[t] * (
                admm.lambdas[admm.iteration][t]
                + sum(
                    admm.ptdf[l, node_index] * (
                        admm.mues[admm.iteration][l, t]
                        - admm.rhos[admm.iteration][l, t]
                    ) for l in admm.L
                )
            )
            + admm.gamma/2 * sub[:penalty_term_eb][t]
            + 10*sub[:penalty_term_upper_flow][t]
            + 10*sub[:penalty_term_lower_flow][t]
            + admm.gamma/2 * sub[:penalty_term_U][t]
            + admm.gamma/2 * sub[:penalty_term_K][t]
            + 1/2 * (P[t] - prev_P[t])^2
            for t in admm.T)
    )

    # Solve optimization problem.
    optimize!(sub)

    # Create penalty term structure.
    penalty_terms = PenaltyTerm(
        value.(sub[:penalty_term_eb]).data,
        value.(sub[:penalty_term_upper_flow]).data,
        value.(sub[:penalty_term_lower_flow]).data
    )

    # Create result structure.
    result = ResultGenerator(
```

```
79          generator,
80          value.(P).data,
81          penalty_terms,
82          value.(U).data,
83          value.(K).data
84      )
85
86      return result
87  end
```

The layout of the function is similar to the implementation of the centralized problem in listing (3). First, the JuMP model instance `sub` is initiated. The name refers to the optimization problem being a subproblem of the decentralized optimal power flow. Next, the optimization variables are created. The variable names resemble once again the terms in the equations of section 3.2. Compared to implementing the centralized algorithm, the decision variable P only depends on the time index. Before the injection expression is added to the model instance, the results from the previous iteration are retrieved from the `admm` object in line `16` and `19`. This is done with the help of two functions that are defined in the file `src/helpers/results.jl`. Both functions ensure that in the first iteration, in which no previous results exist, a zero vector for the corresponding variables is returned. Otherwise, one could easily access the previous iteration results from the `admm` object itself. The created variables are then used in the injection expression. As for the centralized problem, the ternary operator is used to determine if the node of the injection term equals the node at which the generator is located. If so, the generation decision variable is added to the injection term, and the previous generation result is subtracted from the generation node result as written in section 3.2.2. The `admm` object includes a dictionary attribute `node_id_to_demand` that is utilized in the injection expression to access the nodal demand by the id of the node. As a next step, the penalty terms for the complicated constraints and the slack variables are added to the JuMP model in line `39`. Since both the generator and storage subproblem have the same penalty terms, a method called `add_penalty_terms!(...)` was created that can be found in the file `src/optimization/penalty_terms.jl` and is listed in A.3.3. The method attaches the penalty terms as expressions to the model instance `sub`. Hereby, the second-order norm formulation from section 3.2.2 is applied. Thus, the penalty term expression only depends on the time index. As for the slack decision variables of the subproblems, the average value from all subproblems is included in the penalty term. A function is used to get the average value for the slack variables from the previous iteration that is defined in the result helper file mentioned before and returns a zero vector in the first iteration. The objective function of the generator problem is

attached to the model instance in line `45` of listing (6). This equation reflects equation (29a) of section 3.2.2 with one exception. The factors before the penalty terms are different, see for example line `59`. This is due to the fact that equation (29a) represents a general problem, whereas the objective function of the model instance `sub` stands for a specific implementation. See section 4.2 for further reference. Once again, the dot product between the $PTDF$ and the dual variables of the power flow constraint is implemented by summing up the vector entries due to a Julia type error. Adding the objective function to the model instance concludes the model setup, and the model can be optimized in line `68`. After solving the problem, the values for the penalty terms are clustered in a corresponding structure that is defined in the file `src/structures/penalty_terms.jl`. Lastly, the result object of the generator is created. The advantage here is that one can easily access the optimization results from the generator with the help of the structure `ResultGenerator` that is defined in the file `src/structures/results.jl`. See A.1.4 for further reference. Returning the result object is the last step of the method. Then, the optimization of a single generator subproblem is finished.

A detailed description about the storage implementation is omitted since the storage subproblem is implemented in the same fashion as the generator subproblem. The only differences between the two implementations are the decision variables and the fact that the storage problem includes another constraint to ensure the consistency of the energy level. The implementation of the `optimize_subproblem(storage::Storage)` method can be found in A.3.4 for further detail.

The last step of the iteration calculation is the concatenation of all subproblem results. The returned generator or storage result objects are stored in a dictionary that is then used to create the result object of the whole algorithm. See line `6` to `14` in A.3.1. The corresponding structures `Result` can be found in `src/structures/results.jl`. For the algorithm's result structure, a custom constructor is used whose only parameter is a dictionary containing all subproblem results indexed by the network element. The constructor then calculates all relevant information for the iteration, like the total costs of the iteration, the nodal injection values, or the line utilization. It also sets up a dictionary in which the results for each node are stored that are represented by the structure `ResultNode` that is also defined in the file mentioned above. These node results are used to access the information about the last iteration values to calculate the penalty term of the system balance. Another important piece of information is the average values for the upper and lower flow

slack variables that are needed to set up the power flow penalty terms and can be accessed by the attributes `avg_U` and `avg_K` respectively.

Update Dual Variables

The next step of the iteration calculation involves the updating process of the dual variables λ_t^v, $\mu_{l,t}^v$ and $\rho_{l,t}^v$, see line 7 in A.3.1. The implementation of the update process is defined in `src/optimization/update_duals.jl` and consists of a main function `update_duals!(...)` that handles the update of each dual variable. Since Julia does not provide the functionality of private functions, the update functions of each dual variable are marked with a double underscore. This is a regularly used convention to show that these functions should not be called anywhere else in the project. Updating a single dual variable does not make sense and should be avoided. Listing (7) shows the implementation of all updating methods for the dual variables.

Listing 7: Implementation of the dual updates

```
1  function update_duals!(admm::ADMM)
2      __update_lambda!(admm)
3      __update_mue!(admm)
4      __update_rho!(admm)
5  end
6
7  function __update_lambda!(admm::ADMM)
8      lambdas = admm.lambdas[admm.iteration] + admm.gamma * (
9          admm.results[admm.iteration].generation
10         + admm.results[admm.iteration].discharge
11         - admm.results[admm.iteration].charge
12         - admm.total_demand
13     )
14     push!(admm.lambdas, lambdas)
15 end
16
17 function __update_mue!(admm::ADMM)
18     injection = admm.results[admm.iteration].injection
19     avg_U, _ = get_average_slack_results(admm.iteration)
20     mues = (
21         admm.mues[admm.iteration]
22         + admm.gamma * (admm.ptdf * injection + avg_U .- admm.f_max)
23     )
24     condition = avg_U .<= 1e-2
25     mues .*= condition
26     push!(admm.mues, mues)
27 end
28
29 function __update_rho!(admm::ADMM)
30     injection = admm.results[admm.iteration].injection
31     _, avg_K = get_average_slack_results(admm.iteration)
32     rhos = (
33         admm.rhos[admm.iteration]
```

```
34          + admm.gamma * (avg_K- admm.ptdf * injection .- admm.f_max)
35      )
36      condition = avg_K .<= 1e-2
37      rhos .*= condition
38      push!(admm.rhos, rhos)
39  end
```

Each update method for the dual variable resembles equations (31a) to (31b). As for λ_t^v, the overall system balance of the current iteration is calculated utilizing the result object that was created after solving all subproblems. The updated lambda (λ_t^{v+1}) is then added to the attribute `lambdas` of the `admm` object in line `14`. Thus, the next iteration can access the updated lambda as the last element in the array. The dual variables for the power flow constraint are updated similarly. Only the injection term of the current iteration and the corresponding average slack variable are needed to calculate the dual power flow variables of the next iteration. The injection term is retrieved from the result object of the current iteration. As for the average slack variables of all subproblems, the helper function from the penalty terms is utilized again, see for example line `19`. Both update functions for the dual power flow variables include an adaption in line `24`/`25` and `36`/`37`. The reason for this is explained in more detail in section 4.2. Finally, the calculated and adjusted dual power flow variables are added to the corresponding attribute of the `admm` object to utilize the duals in the next iteration.

Check Convergence Status

The last step of the iteration calculation implies the convergence check. The algorithm converges if the difference between the next and the current iteration values of all dual variables is smaller than a predefined threshold ϵ. The algorithm converges only if all residuals of the dual variables are within that threshold. Since all dual variables are stored in an array, calculating the residual between the next and the current iteration is straightforward. The implementation of the convergence check can be found in `src/optimization/convergence.jl` and is listed below.

Listing 8: Implementation of the convergence check

```
1  function check_convergence!(admm::ADMM)
2      eps = 10^(-3)
3      if admm.iteration != 1
4
5          lambda_res = abs.(admm.lambdas[end] - admm.lambdas[end-1])
6          push!(admm.convergence.lambda_res, lambda_res)
7
8          mue_res = abs.(admm.mues[end] - admm.mues[end-1])
9          push!(admm.convergence.mue_res, mue_res)
10
```

```
11          rho_res = abs.(admm.rhos[end] - admm.rhos[end-1])
12          push!(admm.convergence.rho_res, rho_res)
13
14          admm.convergence.lambda = all(lambda_res .< eps)
15          admm.convergence.mue = all(mue_res .< eps)
16          admm.convergence.rho = all(rho_res .< eps)
17
18          admm.convergence.all = all([
19              admm.convergence.lambda
20              admm.convergence.mue
21              admm.convergence.rho
22          ])
23      end
24
25      if admm.convergence.all
26          println("Converged")
27      else
28          println("Not converged")
29          admm.iteration += 1
30      end
31  end
```

The already mentioned threshold ϵ is defined in line 2. It does not make sense to calculate the residuals of the duals in the first iteration of the algorithm. Thus, in line 3 the method checks the current iteration number against the initialization value of one. This is sufficient because the iteration number can only take values greater than 1. As a consequence, the convergence structure of the `admm` object is not updated in the first iteration. In any other iteration cycle, the residuals of each dual variable are calculated as written above, see for example line 5. The convergence structure provides an attribute to store the residual of each iteration for further analysis. Thus, the calculated residuals are added to the corresponding array of the convergence structure. Next, each entry of the residual matrix of the duals is compared against the defined threshold in lines 14 to 16. This is implemented by using the dot syntax of Julia in front of the less operator. The function `all(...)` tests whether all entries of the residual matrix are less than the threshold. As soon as one entry is bigger than the threshold, the function returns `false`. Finally, the convergence of the whole algorithm can be determined using the `all(...)` function again to check if all duals are converged. The last part of the method checks the convergence status and increases the iteration number by one if the algorithm is not converged. In addition, the convergence status is printed on the console.

4 Results

The following chapter presents the results of this thesis. First, a case study system is presented as a reference example to calculate both the centralized and decentralized problems. Next, the convergence of the decentralized algorithm is analyzed, and several issues are presented that influence the algorithm's convergence. Following this, the results of both approaches are compared against each other.

4.1 Case Study System

This section describes a Three Node System (TNS) that is used as a case study system to examine the decentralized optimal power flow implementation. The system is rather simple to grasp the findings better. The implementation of the case study system can be found in the file `src/cases/three_node.jl`. As mentioned in section 3.3, the source code can be easily extended by a more detailed case study system. The TNS consists of three nodes. A different set of generators and storages is located at each of them. Each node has a different two-period demand vector. A total of three transmission lines connect the nodes. Thus, every node has two neighbors. All network elements must meet the requirements described in section 3.1. The network layout and the corresponding parameters of the elements are shown in figure (3).

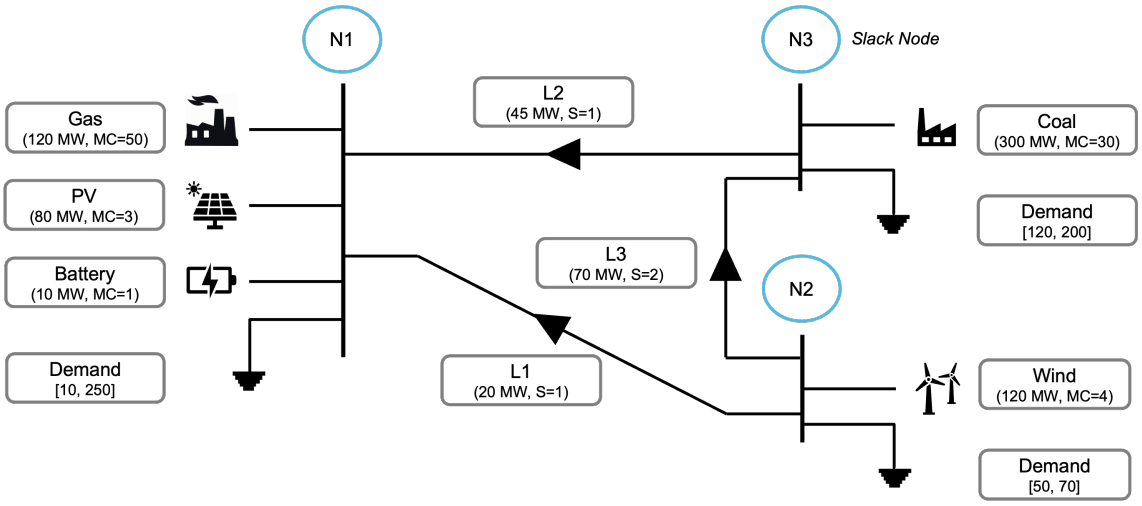

Figure 3: Network layout of case study system

There is a total of four generators in the system. The name of the generators refers to real-life power plants. For example, at node 3 (N3), the coal generator has the largest production capacity and is more expensive than the photovoltaic and wind generator. On the other hand, the gas generator has the highest marginal

costs in the system to account for a generator that is only used to cover peak loads. The marginal costs for the renewable generators are not set to zero because this would violate the penalty terms of the decentralized formulation. Out of this reasons, the marginal costs of these generators are set to 3 and 4, respectively. Both renewable generators act like a dispatchable unit in the TNS. This contradicts the nature of renewable energy resources. However, compared to the other generators, the marginal costs of the photovoltaic and wind generator are minimal. Thus, these generators will always be dispatched.

Generators $\mathcal{G}$	mc [EUR/MWh]	$\overline{p}$ [MW]
PV	3	80
Wind	4	120
Coal	30	300
Gas	50	120

Table 5: Generator parameters for the case study system

In addition to the generators, there is one storage in the case study system that is located at node 1. This network element has the cheapest marginal costs to ensure that the storage is favored in all dispatch decisions. Also, the total system demand of time step one is set to 180 MW and thus smaller than the capacity of the photovoltaic and wind generator together. Subsequently, the storage should be charged to reduce the power output of either the coal or gas generator in time step two. The parameters of the storage are shown in table (6).

Storages $\mathcal{S}$	mc [EUR/MWh]	$\overline{p}$ [MW]	$\overline{e}$ [MWh]
Battery	1	10	20

Table 6: Storage parameters for the case study system

Lastly, the capacities of the transmission lines are set to values so that a congestion on either one line is almost certain. All transmission line specific parameters can be found in table 7.

Transmission Lines $\mathcal{L}$	f [MWh]	S [1/Ω]
Line 1	20	1
Line 2	45	1
Line 3	70	2

Table 7: Transmission line parameters for the case study system

4.2 Convergence Analysis of the Decentralized Algorithm

The following section studies the convergence of the decentralized algorithm. As already mentioned, several convergence problems were faced while implementing the mathematical formulations from section 3.2.2 with Julia.

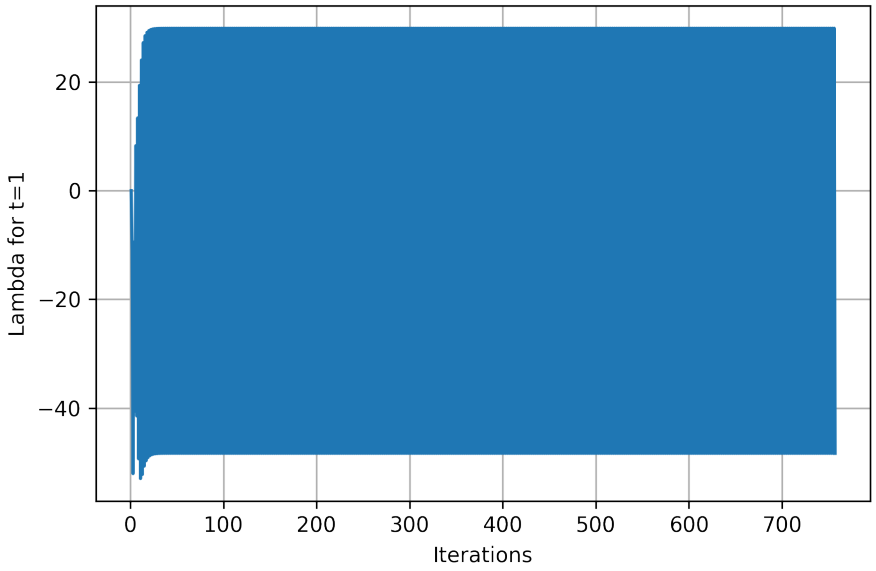

Figure 4: Convergence Problem I: Damping parameter too large

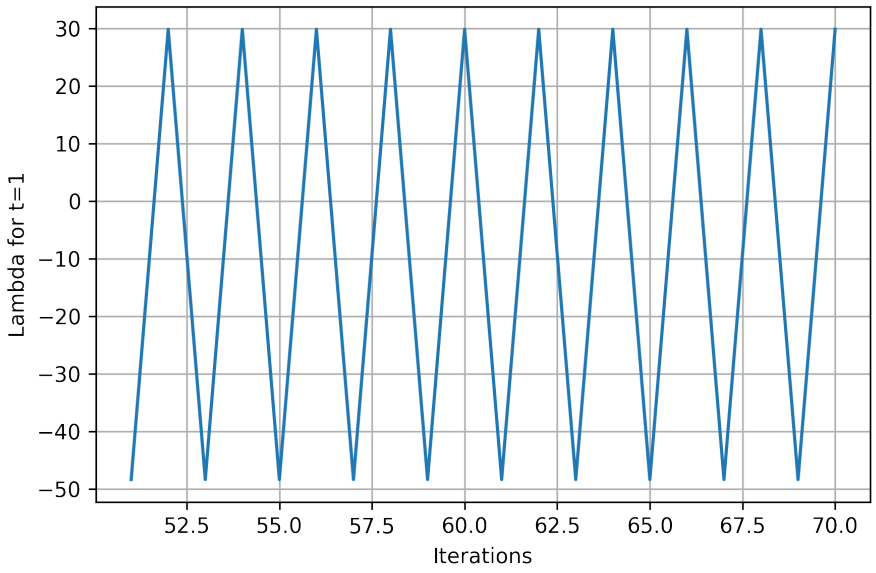

Figure 5: Convergence Problem I: Zoomed version of figure (4)

First of all, it was found that the algorithm is susceptible to the damping param-

eter γ. Although stated as one of the main advantages of the ADMM that one has to choose only one parameter, finding the correct value for the damping parameter represented a challenge. In figure (4) one can see the iteration values for the dual variable λ_1 for time step one. The value of λ_1 is plotted for every iteration. The damping parameter was set to $\gamma = 0.5$. As one can see, the decentralized algorithm did not converge and was interrupted after 750 iterations. The values for λ_1 seemed to be jumping between 30 and -48. See also figure (5) for a better visualization of the divergence. Different plotting scripts were defined in `src/helpers/plotting.jl` to better zoom and pan the plots. After analyzing the plots, it could be identified that the damping parameter was too large. Decreasing the damping parameter to $\gamma = 0.3$ solved the divergence problem of the dual variable λ_t. Figure (6) shows the convergence plot of λ_1.

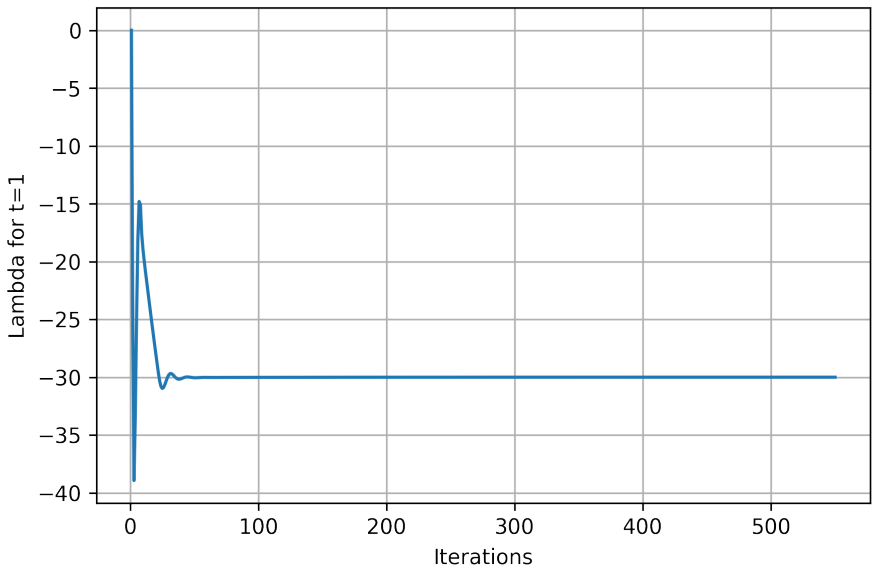

Figure 6: Convergence Dual Variable λ_1

After solving the convergence problem of the system balance dual variable, another divergence problem was faced. Although the correct value for λ_t was found, the dual variables for the upper and lower power flow constraint did not converge. Figure (7) shows the convergence problem for $\rho_{2,1}$ at time step one for transmission line two. Remember that $\rho_{l,t}$ is the dual variable of the lower power flow constraint. If the demand is adapted so that no congestions occur, the dual variable converges. The same could be identified for the dual variable of the upper flow constraint $\mu_{l,t}$. Thus, it seemed that the algorithm could not handle transmission lines that were at their maximum line capacity. This led to the assumption that the penalty terms

for the power flow constraints are not sensitive enough.

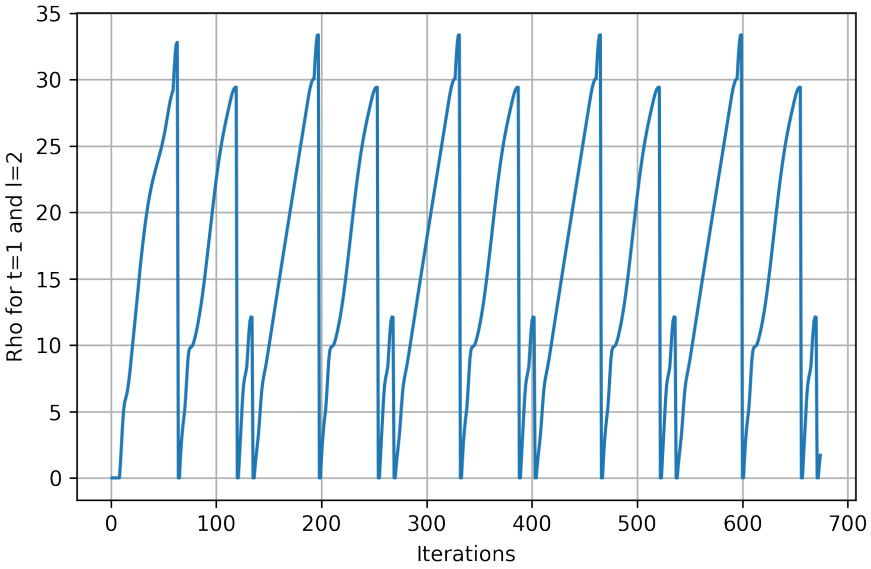

Figure 7: Convergence Problem II: Wrong weight for power flow penalty term

Another observation supported this assumption. It was found that the value of the penalty terms of the power flow constraints is much smaller than the value of the penalty term of the system balance. Consequently, the weight in front of the power flow constraints was adapted to increase the influence of the power flow penalty terms. The general mathematical formulation of the ADMM states that the relaxed complicated constraint is multiplied with the factor $\frac{\gamma}{2}$. This factor was adapted to a higher value to increase the impact of the power flow penalty terms. After several tries, the factor was changed to 10, as one can see in line `59` and `60` of listing (6), and the dual variables of the power flow constraints were not diverging anymore.

However, the algorithm was still not converging because the difference of the power flow dual variables between the current and the previous iteration was always greater than the defined threshold ϵ. It was observed that the average slack variables $\Upsilon_{l,t}$ and $\Gamma_{l,t}$ were only almost zero for time steps and transmission lines that were at their maximum line capacity limit. In terms of the modeling framework, these average values should equal zero because the line capacity is fully utilized, see equations (19h) and (19i) for further reference. This was why the duals of the power flow constraint were not converging. As a solution, the update process of the power flow dual variables was adapted. The single elements of the average slack variables are checked against a certain threshold. If the values are smaller than the threshold, the

corresponding entry of the dual variables is set to zero. Be aware that the mentioned threshold is not the threshold ϵ used for the convergence of the dual variables. One can find the implementation of the before mentioned process in lines `24` and `25` in listing (7) that shows the update method for the upper flow dual variable $\mu_{l,t}$. Since this procedure is not very straightforward, it is explained with an example. At an iteration i after solving all subproblems, the following average slack variable $\Upsilon_{l,t}$ for the upper power flow limit and dual variable $\mu_{l,t}$ was calculated:

$$\Upsilon_{l,t} = \begin{bmatrix} 30.00 & 6.70 \cdot 10^{-10} \\ 90.00 & 25.01 \\ 1.91 \cdot 10^{-8} & 69.98 \end{bmatrix} \tag{40}$$

$$\mu_{l,t} = \begin{bmatrix} 3.16 \cdot 10^{-6} & 129.89 \\ 7.31 \cdot 10^{-6} & 2.47 \cdot 10^{-5} \\ 25.46 & -9.86 \cdot 10^{-5} \end{bmatrix} \tag{41}$$

The updated dual variable is then calculated with the help of equation (31b). Since the entries for transmission line three and time step one and transmission line one and time step two are not zero, the updated dual is constantly increasing by a marginal value. Subsequently, the overall algorithm is not able to converge. Thus, the before-mentioned process is implemented. A boolean matrix is created that incorporates the check against the defined threshold of 10^{-2}.

$$\Upsilon_{l,t} < 10^{-2} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{42}$$

This boolean matrix is then multiplied with the updated dual variable to smooth the values, see the following equation:

$$\left\langle \begin{bmatrix} 3.16 \cdot 10^{-6} & 129.89 \\ 7.31 \cdot 10^{-6} & 2.47 \cdot 10^{-5} \\ 25.46 & -9.86 \cdot 10^{-5} \end{bmatrix} , \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} \right\rangle = \begin{bmatrix} 0 & 129.89 \\ 0 & 0 \\ 25.46 & 0 \end{bmatrix} \tag{43}$$

After implementing this correction, the dual variables of the power flow constraint were finally converging. Figure (8) and (9) show the convergence of the upper und lower power flow dual respectively. In figure (8), the line plot for transmission line one is not visible because it is covered by the line plot for transmission line two. Both transmission lines are not at their maximum line capacity limit. Thus the value of the dual variable is zero. The same applies to the line plots of transmission lines one and three in figure (9).

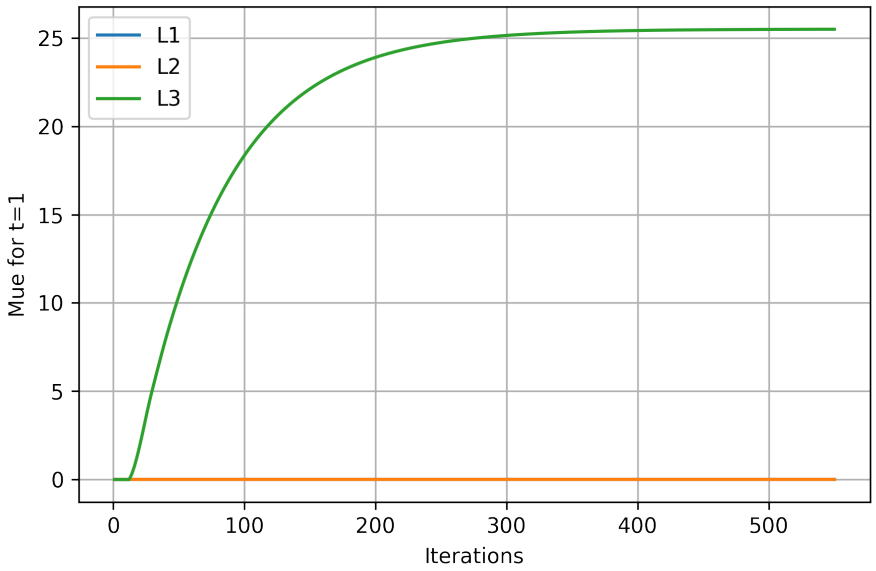

Figure 8: Convergence Dual Variable $\mu_{l,1}$

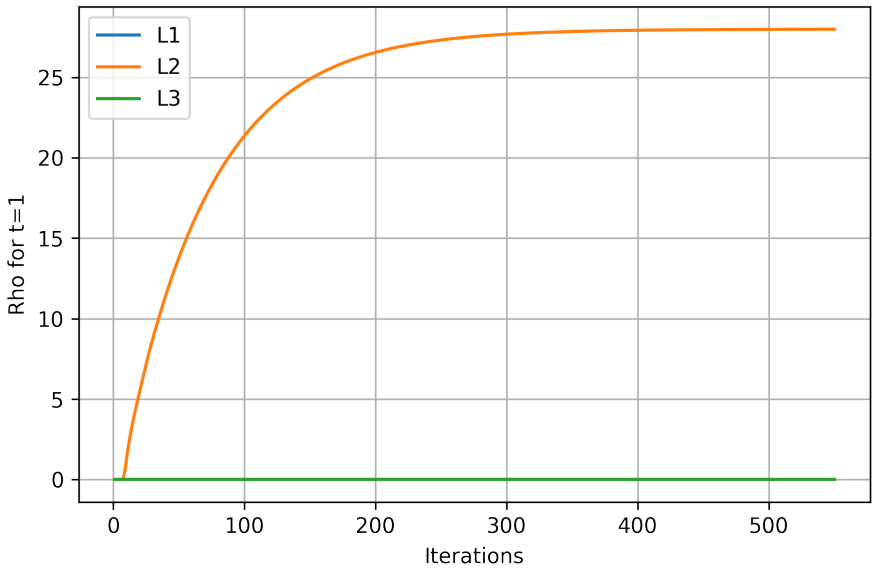

Figure 9: Convergence Dual Variable $\rho_{l,1}$

4.3 Comparison between Centralized and Decentralized Approach

The results for both the centralized optimal power flow and the decentralized optimal power flow were calculated for the case study system. For each network element, the obtained results are compared to each other. Tables are used to allow the reader to spot any differences quickly. First, the results of the generator elements are analyzed.

Generators $\mathcal{G}$	Centralized OPF P [MW]	
	$t = 1$	$t = 2$
PV	75.0000	80.0000
Wind	110.0000	90.0000
Coal	5.0000	220.0000
Gas	0.0000	120.0000

Table 8: Generator results for centralized optimal power flow

Generators $\mathcal{G}$	Decentralized OPF P [MW]	
	$t = 1$	$t = 2$
PV	75.0015	80.0000
Wind	110.0008	90.0166
Coal	4.9976	219.9834
Gas	0.0000	120.0000

Table 9: Generator results for decentralized optimal power flow

As one can see in tables (8) and (9), there are only small differences between the results of both approaches. All of them are in the per mille range. The biggest difference can be observed for the wind generator at time step two. Here, the discrepancy is about 1.84 ‰. Thus, one can state that the generator results of the decentralized optimal power flow resemble the results of the centralized approach. Furthermore, the generation results are as expected. The photovoltaic and wind generator are dispatched in both time steps since they are the cheapest generators in the system. The coal generator accounts for the residual in time step one due to the transmission line congestions. There is a surplus of 10 MW in time step one. This power is used to charge the battery storage, as one can see in the next two tables (10) and (11) that show the storage results for both approaches. This includes the discharge power, the charge power, and the energy level of all-time steps. The results of the centralized and decentralized approaches are the same. There are no differences in the per mille range. As expected, the storage is charged with the surplus of energy from the photovoltaic generator in time step one and discharged in time step two to reduce the total system costs by preventing the coal generator from producing

more electricity. Hence, the battery's energy level is zero at the end of time step two.

Storages $\mathcal{S}$	Centralized OPF					
	D [MW]		C [MW]		E [MWh]	
	$t=1$	$t=2$	$t=1$	$t=2$	$t=1$	$t=2$
Battery	0.0000	10.0000	10.0000	0.0000	10.0000	0.0000

Table 10: Storage results for centralized optimal power flow

Storages $\mathcal{S}$	Decentralized OPF					
	D [MW]		C [MW]		E [MWh]	
	$t=1$	$t=2$	$t=1$	$t=2$	$t=1$	$t=2$
Battery	0.0000	10.0000	10.0000	0.0000	10.0000	0.0000

Table 11: Storage results for decentralized optimal power flow

Tables (12) and (13) list the results of all transmission lines in the system. Similar to the generator results, small differences between the centralized and decentralized approaches can be observed. However, most differences are only in the per mille range, with one exception. The discrepancy for line three at time step two is 0.0133 MW. Although this difference is a lot bigger than the difference from the generator results, it is still reasonable to claim that the transmission line results from the centralized approach equal the results from the decentralized approach. At time step one, transmission lines two and three are at their capacity limits. Transmission line two is at its lower limit of -45 MW, and transmission line three is at its upper limit of 70 MW. These congestions explain the observation that the coal generator is dispatched in time step one, although the cheaper photovoltaic and wind generator do not exploit their production limits. There is only one exploited transmission line at time step two. This is transmission line one at its upper limit with a utilization of 20 MW. At time step two, line three is not utilized at all because the wind and coal generator satisfy the nodal demand, respectively.

Tables (14) and (15) show the injections at each system node. Once again, the differences between the centralized and decentralized approaches are minimal. The biggest discrepancy can be observed at nodes two and three for time step two. The difference between the two approaches is 8.3 ‰. Subsequently, the claim that both

Transmission Lines $\mathcal{L}$	Centralized OPF Capacity [MW]	
	$t = 1$	$t = 2$
Line 1	-10.0000	20.0000
Line 2	-45.0000	20.0000
Line 3	70.0000	0.0000

Table 12: Transmission line results for centralized optimal power flow

Transmission Lines $\mathcal{L}$	Decentralized OPF Capacity [MW]	
	$t = 1$	$t = 2$
Line 1	-10.0005	20.0033
Line 2	-45.0011	19.9967
Line 3	70.0013	0.0133

Table 13: Transmission line results for decentralized optimal power flow

approaches yield the same results is still valid. The results for the nodal injections are as expected. At time step one, node three is the only node that imports electricity from the other nodes. It is the other way around for time step two, where only node one imports electricity from the other nodes. The sum of all nodal injections equals zero for every time step. Thus, the system balance constraint is ensured.

Nodes $\mathcal{N}$	Centralized OPF Injection [MW]	
	$t = 1$	$t = 2$
Node 1	55.0000	-40.0000
Node 2	60.0000	20.0000
Node 3	-115.0000	20.0000

Table 14: Node results for centralized optimal power flow

Next, the system electricity price and the nodal prices are analyzed. The system electricity price is the dual variable of the system balance constraint. According to Yang et al. (2019), the nodal price can be calculated as the sum of the system price and the dual variables of the upper and lower power flow constraint. The corresponding mathematical formulation can be found in equation (44).

Nodes $\mathcal{N}$	Decentralized OPF Injection [MW]	
	$t = 1$	$t = 2$
Node 1	55.0015	-40.0000
Node 2	60.0008	20.0166
Node 3	-115.0020	19.9834

Table 15: Node results for decentralized optimal power flow

	Decentralized OPF Price [EUR]	
	$t = 1$	$t = 2$
System	30.0000	30.0000
Node 1	36.6000	82.0000
Node 2	15.2000	4.0000
Node 3	30.0000	30.0000

Table 16: Prices for the centralized optimal power flow

	Decentralized OPF Price [EUR]	
	$t = 1$	$t = 2$
System	-30.0000	-30.0004
Node 1	-36.5972	-81.9756
Node 2	-15.2162	-4.0127
Node 3	-30.0000	-30.0004

Table 17: Prices for the decentralized optimal power flow

$$r_{n,t} = \lambda_t + \sum_{l \in \mathcal{L}} (\mu_{l,t} + \rho_{l,t}) * PTDF_{l,n} \tag{44}$$

Retrieving the dual variable of the system balance and calculating the nodal prices for each time step with the help of equation (44) yields the results that are shown in tables (16) and (17). The absolute prices are nearly the same. However, the algebraic sign is different. The reason here is the way the system balance is formulated in section 3.2.2 and how the decentralized algorithm is set up. Thus, the algebraic sign is irrelevant and can be ignored for this analysis. The differences between the two approaches are again only marginal. Although the differences are more significant

than the generator results, it is possible to claim that the two approaches yield the same system and nodal prices for every time step. The interpretation of the prices is not straightforward because of the power line congestion. However, the prices are like one would assume. The most significant nodal price (82 EUR) is at node one for time step two, where we have the highest congestion in the network. This leads to the dispatch of the gas generator, which is the most expensive generator.

5 Conclusion

This thesis analyzed if a fully decentralized algorithm that solves an optimal power flow optimization problem with the help of the ADMM. It was shown that this is possible. Hereby, storage resources and multiple time steps could be included in the modeling framework showing that the decentralized coordination of multiple dimensions can be done. The decentralized algorithm was implemented in Julia. During the implementation, several convergence problems were tackled. First, it was found that the only parameter of the ADMM is strongly influencing the convergence of the algorithm. Thus, it is crucial to find the correct damping parameter of the ADMM that allows for a high accuracy as well as a low number of iterations. Next, the weight of the power flow penalty terms prevented the convergence. Increasing the weight from $\frac{\gamma}{2}$ to 10 solved this convergence problem. Lastly, an inaccuracy in the update process of the power flow dual variables was fixed that prevented these dual variables from converging. The decentralized algorithm was compared to a centralized algorithm on a typical three node case study system with different generators and one energy storage resource. It was shown that the decentralized algorithm yields the same results as the centralized one. The identified differences can be neglected since they are only in the per mille range. This demonstrates that the stability of a modern electricity transmission system cannot only be ensured by a central entity like the TSO doing the coordination on its own but also by letting the market participants finding their optimal decision on their own while all network constraints are still secured.

5.1 Contributions

Several contributions were made while writing this thesis. An open-source package DecentralOPF.jl[7] was published under MIT license. It contains the complete source code for implementing the decentralized algorithm in Julia. The source code is very well documented through section 3 of this thesis. Solving the optimal power flow in a decentralized manner is explained in great detail and always refers to the source code. Finally, the project structure was based on best practices in modern software development. Thus, it can be easily used and extended by others.

[7] www.github.com/rockstaedt/DecentralOPF.jl

5.2 Limitation and Outlook

A few assumptions limit the impact of this thesis. The implemented modeling framework is rather simple than sophisticated, e.g., the generators do not have any ramping constraints, and there are no efficiency losses for the storages. In addition, the modeling framework does not distinguish between dispatchable and non-dispatchable resources. Thus, renewable energy resources are not fully resembled. Removing these limits would undoubtedly improve the applicability of the presented decentralized algorithm. These limits also represented initial starting points for further research. Future work could also study a method to automatically determine the damping parameter γ based on certain conditions. This would allow the algorithm to be applied to a more extensive set of case study systems. It would also be worth looking into the implementation of regulations. It was found that while providing a decentralized algorithm, specific behavior can be induced, e.g., providing financial incentives to increase the security of the transmission network. Finally, future work should also test the decentralized algorithm on a more sophisticated case study system with more than two time steps and three nodes.

References

Ahlqvist, Victor, Pär Holmberg, and Thomas Tangerås. 2022. "A Survey Comparing Centralized and Decentralized Electricity Markets." *Energy Strategy Reviews* 40 (2022): 100812.

Boyd, Stephen. 2010. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers." *Foundations and Trends® in Machine Learning* 3 (1): 1–122.

Bundesnetzagentur. 2021. "Zahlen, Daten, Informationen Zum EEG." Accessed May 11, 2022. https://www.bundesnetzagentur.de/DE/Fachthemen/ElektrizitaetundGas/ErneuerbareEnergien/ZahlenDatenInformationen/start.html.

Chowdhury, B.H., and S. Rahman. 1990. "A Review of Recent Advances in Economic Dispatch." *IEEE Transactions on Power Systems* 5 (4): 1248–1259.

Conejo, Antonio J, Enrique Castillo, Roberto Minguez, and Raquel Garcia-Bertrand. 2006. *Decomposition Techniques in Mathematical Programming: Engineering and Science Applications.* Springer Science & Business Media.

Dantzig, George. 1963. *Linear Programming and Extensions.* RAND Corporation.

Dunning, Iain, Joey Huchette, and Miles Lubin. 2017. "JuMP: A Modeling Language for Mathematical Optimization." *SIAM Review* 59 (2): 295–320.

Hetzer, John, David C. Yu, and Kalu Bhattarai. 2008. "An Economic Dispatch Model Incorporating Wind Power." *IEEE Transactions on Energy Conversion* 23 (2): 603–611.

Mieth, Robert. 2021. "Risk-Aware Control, Dispatch and Coordination in Sustainable Power Systems." Doctoral thesis, Technische Universität Berlin.

Pesch, T., H.-J. Allelein, and J.-F. Hake. 2014. "Impacts of the Transformation of the German Energy System on the Transmission Grid." *The European Physical Journal Special Topics* 223, no. 12 (2014): 2561–2575.

Quint, Ryan, Lisa Dangelmaier, Irina Green, David Edelson, Vijaya Ganugula, Robert Kaneshiro, James Pigeon, Bill Quaintance, Jenny Riesz, and Naomi Stringer. 2019. "Transformation of the Grid: The Impact of Distributed Energy Resources on Bulk Power Systems." *IEEE Power and Energy Magazine* 17 (6): 35–45.

Ritchie, Hannah, Max Roser, and Pablo Rosado. 2020. "Renewable Energy Statistics." Our World in Data, 2020. Accessed May 11, 2022. https://ourworldindata.org/renewable-energy.

Van Hertem, D., J. Verboomen, K. Purchala, R. Belmans, and W.L. Kling. 2006. "Usefulness of DC Power Flow for Active Power Flow Analysis with Flow Controlling Devices." In *The 8th IEE International Conference on AC and DC Power Transmission,* 58–62.

Weinhold, Richard. 2022. "Open-Source Modeling of Flow Based Market Coupling : Methods, Parametrization, and Analysis for Sustainable Power Systems." Doctoral thesis, Technische Universität Berlin.

Xing, Hao, Zhiyun Lin, Minyue Fu, and Benjamin F. Hobbs. 2017. "Distributed Algorithm for Dynamic Economic Power Dispatch with Energy Storage in Smart Grids." *IET Control Theory & Applications* 11 (11): 1813–1821.

Yang, Jiajia, ZhaoYang Dong, Guo Chen, Fushuan Wen, and Chen Li. 2019. "A Fully Decentralized Distribution Market Mechanism Using ADMM." In *2019 IEEE Power & Energy Society General Meeting (PESGM),* 1–5. Atlanta, GA, USA: IEEE.

A Appendix: Julia Implementations

A.1 Structures

A.1.1 Network Elements

```
mutable struct Node
    name::String
    demand::Vector{Int}
    slack::Bool
end

mutable struct Generator
    name::String
    marginal_costs::Int
    max_generation::Int
    plot_color::String
    node::Node
end

mutable struct Storage
    name::String
    marginal_costs::Int
    max_power::Int
    max_level::Int
    plot_color::String
    node::Node
end

mutable struct Line
    name::String
    from::Node
    to::Node
    max_capacity::Int
    susceptance::Int
end
```

A.1.2 ADMM

```
mutable struct ADMM
    iteration::Int
    gamma::Float64
    lambdas::Vector{Vector{Float64}}
    mues::Vector{Matrix{Float64}}
    rhos::Vector{Matrix{Float64}}
    T::Vector{Int}
    N::Vector{Int}
    L::Vector{Int}
    nodes::Vector{Node}
    generators::Vector{Generator}
    storages::Vector{Storage}
```

```
    lines::Vector{Line}
    results::Vector{Result}
    convergence::Convergence
    ptdf::Matrix{Float64}
    total_demand::Vector{Float64}
    node_id_to_demand::Dict{Int, Vector{Int}}
    node_to_id::Dict{Node, Int}
    node_to_units::Dict{Node, Vector{Union{Generator, Storage}}}
    f_max::Vector{Float64}

    function ADMM(gamma::Float64,
                  nodes::Vector{Node},
                  generators::Vector{Generator},
                  storages::Vector{Storage},
                  lines::Vector{Line})
        admm = new()
        admm.iteration = 1
        admm.gamma = gamma
        admm.T = collect(1:length(nodes[1].demand))
        admm.N = collect(1:length(nodes))
        admm.L = collect(1:length(lines))
        admm.lambdas = [zeros(Float64, length(admm.T))]
        admm.mues = [zeros(Float64, length(admm.L), length(admm.T))]
        admm.rhos = [zeros(Float64, length(admm.L), length(admm.T))]
        admm.nodes = nodes
        admm.generators = generators
        admm.storages = storages
        admm.lines = lines
        admm.f_max = [line.max_capacity for line in admm.lines]
        admm.results = []
        admm.convergence = Convergence()
        admm.ptdf = calculate_ptdf(admm.nodes, admm.lines)
        admm.total_demand = zeros(length(admm.T))
        admm.node_id_to_demand = Dict()
        admm.node_to_id = Dict()
        admm.node_to_units = Dict()
        for (id, node) in enumerate(admm.nodes)
            admm.total_demand += node.demand
            admm.node_id_to_demand[id] = node.demand
            admm.node_to_id[node] = id
        end
        for unit in vcat(admm.generators, admm.storages)
            if haskey(admm.node_to_units, unit.node)
                push!(admm.node_to_units[unit.node], unit)
            else
                admm.node_to_units[unit.node] = [unit]
            end
        end
        return admm
    end
end
```

A.1.3 Convergence

```
mutable struct Convergence
    lambda::Bool
    lambda_res::Vector{Vector{Float64}}
    mue::Bool
    mue_res::Vector{Matrix{Float64}}
    rho::Bool
    rho_res::Vector{Matrix{Float64}}
    all::Bool
    function Convergence()
        convergence = new()
        convergence.lambda = false
        convergence.lambda_res = []
        convergence.mue = false
        convergence.mue_res = []
        convergence.rho = false
        convergence.rho_res = []
        convergence.all = false
        return convergence
    end
end
```

A.1.4 Results

```
struct ResultStorage
    storage::Storage
    discharge::Vector{Float64}
    charge::Vector{Float64}
    level::Vector{Float64}
    penalty_term::PenaltyTerm
    U::Matrix{Float64}
    K::Matrix{Float64}
end

struct ResultGenerator
    generator::Generator
    generation::Vector{Float64}
    penalty_term::PenaltyTerm
    U::Matrix{Float64}
    K::Matrix{Float64}
end

mutable struct ResultNode
    node::Node
    generation::Vector{Float64}
    discharge::Vector{Float64}
    charge::Vector{Float64}
    penalty_term::PenaltyTerm

    function ResultNode(node::Node)
        node_result = new()
```

```
        node_result.node = node
        node_result.generation = zeros(Float64, length(admm.T))
        node_result.discharge = zeros(Float64, length(admm.T))
        node_result.charge = zeros(Float64, length(admm.T))
        node_result.penalty_term = get_empty_penalty_term()
        return node_result
    end
end

mutable struct Result
    unit_to_result::Dict
    node_to_result::Dict{Node, ResultNode}
    generation::Vector{Float64}
    discharge::Vector{Float64}
    charge::Vector{Float64}
    penalty_term::PenaltyTerm
    avg_U::Matrix{Float64}
    avg_K::Matrix{Float64}
    total_costs::Float64
    injection::Matrix{Float64}
    line_utilization::Matrix{Float64}

    function Result(unit_to_result::Dict)
        result = new()

        result.unit_to_result = unit_to_result

        result.generation = zeros(Float64, length(admm.T))
        result.discharge = zeros(Float64, length(admm.T))
        result.charge = zeros(Float64, length(admm.T))
        result.injection = zeros(Float64, length(admm.N), length(admm.T))
        result.avg_U = zeros(Float64, length(admm.L), length(admm.T))
        result.avg_K = zeros(Float64, length(admm.L), length(admm.T))

        result.node_to_result = Dict()
        for (node_id, node) in enumerate(admm.nodes)
            result.injection[node_id, :] -= node.demand
            result.node_to_result[node] = ResultNode(node)
        end

        result.total_costs = 0

        result.penalty_term = get_empty_penalty_term()

        for (unit, result_unit) in unit_to_result
            result.penalty_term = sum_up(
                result.penalty_term,
                result_unit.penalty_term
            )

            result.node_to_result[unit.node] = update(
                result.node_to_result[unit.node],
                result_unit
            )
```

```

            result.avg_U += result_unit.U
            result.avg_K += result_unit.K

            node_id = admm.node_to_id[unit.node]

            if typeof(unit) == Storage
                result.discharge += result_unit.discharge
                result.injection[node_id, :] += result_unit.discharge

                result.charge += result_unit.charge
                result.injection[node_id, :] -= result_unit.charge

                result.total_costs += result_unit.storage.marginal_costs * (
                    sum(result_unit.discharge + result_unit.charge)
                )
            elseif typeof(unit) == Generator
                result.generation += result_unit.generation
                result.injection[node_id, :] += result_unit.generation

                result.total_costs += (
                    sum(result_unit.generation)
                    * result_unit.generator.marginal_costs
                )
            end
        end

        # Calculate average of slack variables
        counter_subproblems = length(admm.generators) + length(admm.storages)
        result.avg_U = 1/(counter_subproblems) .* result.avg_U
        result.avg_K = 1/(counter_subproblems) .* result.avg_K

        result.line_utilization = admm.ptdf * result.injection

        return result
    end
end
```

A.2 Helper Methods

A.2.1 Calculate a $PTDF$

```
function calculate_ptdf(nodes::Vector{Node}, lines::Vector{Line})

    N = collect(1:length(nodes))
    SLACK = nothing

    incidence = zeros(Int, (length(lines), length(nodes)))
    susceptances = []

    for (l, line) in enumerate(lines)
        push!(susceptances, line.susceptance)
```

```
        for (n, node) in enumerate(nodes)
            if isnothing(SLACK)
                if node.slack
                    SLACK = n
                end
            end
            if line.from == node
                incidence[l, n] = 1
            elseif line.to == node
                incidence[l, n] = -1
            else
                incidence[l, n] = 0
            end
        end
    end

    B = Diagonal([susceptances...])
    # Line suceptance Matrix
    Bl = B*incidence
    # Nodes suceptance Matrix
    Bn = (incidence'*B)*incidence


    B_inv = zeros(length(N), length(N))

    B_inv[setdiff(N, SLACK), setdiff(N, SLACK)] = (
        inv(Bn[setdiff(N, SLACK), setdiff(N, SLACK)])
    )
    PTDF = Bl*B_inv
    return PTDF
end
```

A.3 Optimization Methods

A.3.1 Calculate an Iteration

```
function calculate_iteration!(admm::ADMM)
    optimize_all_subproblems!(admm)

    println("Generation Results: ")
    print_results(true, true, false, admm.iteration)

    update_duals!(admm)

    check_convergence!(admm)
end
```

A.3.2 Optimize all Subproblems

```
function optimize_all_subproblems!(admm::ADMM)
    println("\nIteration: $(admm.iteration)")
    println("###############")
    print_duals(admm.iteration)

    unit_to_result = Dict(zip(
        vcat(generators, storages),
        vcat(
            optimize_subproblem.(generators),
            optimize_subproblem.(storages)
        )
    ))

    result = Result(unit_to_result)

    push!(admm.results, result)
end
```

A.3.3 Add Penalty Terms to Subproblem

```
function add_penalty_terms!(sub)
    # Penalty term energy balance.
    @expression(
        sub,
        penalty_term_eb[t=admm.T],
        sum(sub[:injection][:, t].data)^2
    )

    # Penalty term upper flow.
    @expression(
        sub,
        penalty_term_upper_flow[t=admm.T],
        sum(
            (
                sum(admm.ptdf[l, n] * sub[:injection][n, t] for n in admm.N)
                + sub[:U][l, t] - admm.f_max[l]
            ).^2
            for l in admm.L
        )
    )

    # Penalty term lower flow.
    @expression(
        sub,
        penalty_term_lower_flow[t=admm.T],
        sum(
            (
                sub[:K][l, t]
                - sum(
                    admm.ptdf[l, n] * sub[:injection].data[n, t]
```

```
                    for n in admm.N
                )
                - admm.f_max[l]
            ).^2
            for l in admm.L
        )
    )

    # Penalty terms for slack variables.
    avg_U, avg_K = get_average_slack_results(admm.iteration - 1)

    @expression(
        sub,
        penalty_term_U[t=admm.T],
        sum((sub[:U].data[:, t] - avg_U[:, t]).^2)
    )

    @expression(
        sub,
        penalty_term_K[t=admm.T],
        sum((sub[:K].data[:, t] - avg_K[:, t]).^2)
    )

end
```

A.3.4 Optimize a Single Storage Subproblem

```
    penalty_terms = PenaltyTerm(
        value.(sub[:penalty_term_eb]).data,
        value.(sub[:penalty_term_upper_flow]).data,
        value.(sub[:penalty_term_lower_flow]).data
    )

    # Create result structure.
    result = ResultGenerator(
        generator,
        value.(P).data,
        penalty_terms,
        value.(U).data,
        value.(K).data
    )

    return result
end

function optimize_subproblem(storage::Storage)
    # Create model instance.
    sub = Model(() ->Gurobi.Optimizer(gurobi_env))
    # Omit output in console.
    set_silent(sub)

    # Storage variables with maximum bounds.
    @variable(sub, 0 <= D[t=admm.T] <= storage.max_power)
```

```
    @variable(sub, 0 <= C[t=admm.T] <= storage.max_power)
    @variable(sub, 0 <= E[t=admm.T] <= storage.max_level)

    # Slack variable for upper flow limit.
    @variable(sub, 0 <= U[line=admm.L, t=admm.T])
    # Slack variable for lower flow limit.
    @variable(sub, 0 <= K[line=admm.L, t=admm.T])

    # Get storage results from previous iteration.
    prev_D, prev_C = get_unit_results(storage, admm.iteration - 1)

    # Get general node results from previous iteration.
    prev_node_results = get_node_id_to_result(admm.iteration - 1)

    # Define expression for injection term.
    @expression(
        sub,
        injection[n=admm.N, t=admm.T],
        n == admm.node_to_id[storage.node] ? (
            prev_node_results[n].generation[t]
            + D[t] + (prev_node_results[n].discharge[t] - prev_D[t])
            - C[t] - (prev_node_results[n].charge[t] - prev_C[t])
            - admm.node_id_to_demand[n][t]
        ) : (
            prev_node_results[n].generation[t]
            + prev_node_results[n].discharge[t]
            - prev_node_results[n].charge[t]
            - admm.node_id_to_demand[n][t]
        )
    )

    # Add all penalty terms to the model.
    add_penalty_terms!(sub)

    # Define constraint for energy level of storage.
    @constraint(
        sub,
        StorageBalance[t=admm.T],
        E[t] == (
            (t == 1 ? 0 : E[t-1]) + C[t] - D[t]
        )
    )

    # Get current node index.
    node_index = admm.node_to_id[storage.node]

    # Set objective function.
    @objective(
        sub,
        Min,
        sum(storage.marginal_costs * (D[t] + C[t])
            + (D[t] - C[t]) * (
                admm.lambdas[admm.iteration][t]
                + sum(
```

```
                admm.ptdf[l, node_index] * (
                    admm.mues[admm.iteration][l, t]
                    - admm.rhos[admm.iteration][l, t]
                ) for l in admm.L
            )
        )
        + admm.gamma/2 * sub[:penalty_term_eb][t]
        + 10*sub[:penalty_term_upper_flow][t]
        + 10*sub[:penalty_term_lower_flow][t]
        + admm.gamma/2 * sub[:penalty_term_U][t]
        + admm.gamma/2 * sub[:penalty_term_K][t]
        + 1/2 * (D[t] - prev_D[t])^2
        + 1/2 * (C[t] - prev_C[t])^2
        for t in admm.T)
)

# Solve optimization problem.
optimize!(sub)

# Create penalty term structure.
penalty_terms = PenaltyTerm(
```