



universität
wien

DIPLOMARBEIT

Titel der Diplomarbeit

Optimierkern zur Spannungsregelung in regionalen
Stromnetzwerken mit einem hohen Anteil an verteilten
Energiequellen

Verfasser

Daniel Schwabeneder

angestrebter akademischer Grad

Magister der Naturwissenschaften (Mag.rer.nat)

Wien, 2013

Studienkennzahl lt. Studienblatt: A 405

Studienrichtung lt. Studienblatt: Mathematik

Betreuer: Ao. Univ.-Prof. Dipl.-Ing. Dr. Hermann Schichl

Abstract

Ausgangspunkt dieser Arbeit ist ein Projekt des Austrian Institute of Technology (AIT), das sich mit der Spannungsregelung in kleineren regionalen Stromnetzwerken mit einem hohen Anteil an verteilten Energiequellen beschäftigt. Ziel dieses Projektes ist es durch Einführen eines Kommunikationssystems und einer zentralen Steuerungseinheit ein "intelligentes Netzwerk" (*Smart Grid*) zu etablieren, in dem auch die Kraftwerke zur Spannungsregelung beitragen, um somit die hohen Kosten zu vermeiden, die anfallen würden, wenn man das Netz ausbauen und verstärken müsste. Diese Arbeit beschäftigt sich mit dem Optimierkern des zentralen Reglers, in dem berechnet wird, wie viel Blind- und Wirkleistung die einzelnen Energiequellen einspeisen müssen, damit die Spannung im richtigen Bereich liegt.

Im ersten Kapitel wird kurz die Struktur von Stromnetzwerken erklärt und eine genauere Beschreibung des Projekts des AIT gegeben. Außerdem werden die Optimierungsprobleme vorgestellt, mit denen sich die Arbeit im Weiteren befasst.

Im zweiten Kapitel findet sich eine Einführung in die konvexe Optimierung, also in die mathematischen Grundlagen, die nötig sind, um die hier auftretenden Optimierungsprobleme zu behandeln. Zudem wird ein allgemeines Verfahren zum Auffinden einer Lösung von Problemen dieser Form angegeben.

In den Kapiteln 3 und 4 wird beschrieben, wie die zu behandelnden Optimierungsprobleme auf konvexe Form transformiert werden können, und es werden MATLAB-Funktionen zum Auffinden einer Lösung dieser Probleme vorgestellt. Außerdem wird noch erläutert, welche Auswirkungen es hat, wenn man in der Zielfunktion die euklidische Norm statt der 1-Norm wählt. Auch für diese leicht modifizierten Probleme werden Programme zum Auffinden einer Lösung angegeben und erklärt.

Im fünften Kapitel wird demonstriert, wie man bei der Optimierung einen Fehler mitberücksichtigen kann, der bei der Spannungsberechnung entsteht. Hier wird ebenfalls erklärt, wie man die Probleme, die sich dabei ergeben, auf konvexe Form umformulieren kann. Ferner werden MATLAB-Funktionen vorgestellt, mit denen eine Lösung dieser erweiterten Probleme gefunden werden kann.

In Kapitel 6 wird anhand von zwei konkreten Beispielen die Funktionsweise der verschiedenen Programme veranschaulicht und es werden die Unterschiede in den gefundenen Lösungen aufgezeigt und erläutert. Weiters wird versucht, mit Hilfe von zufällig generierten Testdatensätzen einen Eindruck vom Rechen- und Zeitaufwand der einzelnen Programme in Abhängigkeit von verschiedenen Parametern zu gewinnen.

Danksagung

Ich möchte mich an dieser Stelle bei meinem Betreuer Univ.-Prof. Dipl.-Ing. Dr. Hermann Schichl bedanken, der mir das Thema dieser Arbeit vorgeschlagen und mich stets mit kompetenten und hilfreichen Ratschlägen unterstützt hat.

Außerdem gilt mein Dank Dipl.-Ing. Dr. Alexander Viehweider sowie seinen Kollegen Ing. Daniel Burnier de Castro MSc, Dipl.-Ing. Henein Sawsan und Dipl.-Ing. Matthias Stifter, die mich von Seiten des AIT betreut haben und sehr bemüht waren, mich in das hier vorgestellte Projekt einzuführen.

Weiters möchte ich mich bei all meinen Freunden bedanken, die mich während der Schul- und Studienzeit begleitet haben, und dabei im Besonderen Robert Feichtenschlager erwähnen.

Zuletzt danke ich natürlich meiner Familie, vor allem meinen Eltern Helga und Josef Schwabeneder, die mir mein Studium durch ihre moralische und großzügige finanzielle Unterstützung ermöglicht und erleichtert haben, aber auch meinen Geschwistern Magdalena und Benjamin Schwabeneder, die einerseits stets ein offenes Ohr für meine Probleme haben und mich andererseits durch ihre Tätigkeiten inspirieren.

Inhaltsverzeichnis

1. Motivation	9
1.1. Struktur bestehender Stromnetzwerke und bisherige Spannungsregelung	9
1.2. Probleme für die Spannungsregelung bei großem Zuwachs an Erzeugern	11
1.3. Projekt des Austrian Institute of Technology zur Spannungsregelung - Smart Grid . . .	12
1.3.1. Spannungsregelung durch Steuerung der Blindleistung	13
1.3.2. Spannungsregelung durch Steuerung der Blindleistung und der Wirkleistung . .	14
2. Einführung in die konvexe Optimierung	17
2.1. Grundlagen	17
2.2. Dualität	21
2.3. Optimalitätsbedingungen	23
2.3.1. Unrestringierte Probleme	23
2.3.2. Probleme mit Nebenbedingungen	24
2.4. Algorithmen	25
2.4.1. Unrestringierte Probleme	25
2.4.2. Probleme mit Nebenbedingungen - Ein primal-duales Innere-Punkte-Verfahren	27
3. Spannungsregelung durch Steuerung der Blindleistung	33
3.1. Verfahren für ein allgemeines lineares Problem	33
3.2. Transformation auf ein lineares Problem	38
3.3. Wahl der Norm in der Zielfunktion	43
4. Spannungsregelung durch Steuerung der Wirkleistung und der Blindleistung	45
4.1. Transformation auf ein konvexes Problem	45
4.2. Lösung des konvexen Problems	51
4.3. Euklidische Norm in der Zielfunktion	57
5. Berücksichtigung eines Fehlers bei der Linearisierung der Spannungsberechnung	59
5.1. Steuerung der Blindleistung	60
5.2. Steuerung der Wirkleistung und der Blindleistung	65
6. Numerische Tests	71
6.1. Konkretes Beispiel für die Steuerung der Blindleistung	71
6.2. Konkretes Beispiel für die Steuerung der Wirk- und Blindleistung	75
6.3. Numerische Simulationen	80
6.3.1. Wahl des Parameters μ	82
6.3.2. Wahl der Genauigkeit	84
6.3.3. Abhängigkeit von der Größe der Dimensionen	87
6.3.4. Mehraufwand durch Berücksichtigung eines Fehlers	97

7. Conclusio	99
A. Programme	101
A.1. Steuerung der Blindleistung mit linearer Zielfunktion	101
A.2. Steuerung der Blindleistung mit quadratischer Zielfunktion	105
A.3. Steuerung der Wirk- und Blindleistung mit linearer Zielfunktion	110
A.4. Steuerung der Wirk- und Blindleistung mit quadratischer Zielfunktion	115
A.5. Steuerung der Blindleistung unter Berücksichtigung eines Fehlers bei der Spannungs- berechnung	120
A.6. Steuerung der Wirk- und Blindleistung unter Berücksichtigung eines Fehlers	125
Abbildungsverzeichnis	131
Literaturverzeichnis	135
B. Lebenslauf	137

1. Motivation

In den letzten Jahren wird die Nachfrage an elektrischer Energie immer größer, daher werden weltweit viele neue Kraftwerke gebaut. Es kommt jedoch zu Problemen bei der Spannungsregelung, wenn man neue Energiequellen in bestehende Stromnetzwerke einbinden will, da diese Versorgungsnetze nicht für einen derartigen Anstieg an verteilten Erzeugern (*distributed generation (DG) plants*) konstruiert sind. In diesem Kapitel werden kurz die dabei auftretenden Probleme und Schwierigkeiten beschrieben sowie der Lösungsansatz des Teams um Dipl.-Ing. Dr. Alexander Viehweider vom Energy Department des Austrian Institute of Technology (AIT). Genauer zur Elektroenergietechnik allgemein findet sich zum Beispiel in [Sch09] und eine detailliertere Vorstellung des Projekts vom AIT in [Vie+10].

1.1. Struktur bestehender Stromnetzwerke und bisherige Spannungsregelung

Die europäischen Stromnetze sind üblicherweise in folgende Spannungsebenen aufgeteilt¹:

- *Übertragungsnetz* oder *Transportnetz*:
 - *Höchstspannung* (220kV/380kV): In dieses Netz wird die von Großkraftwerken erzeugte Energie eingespeist und es dient vor allem zum Transport der Energie über weite Strecken hin zu den Verbraucherzentren. Außerdem ist es mit dem Europäischen Übertragungsnetz verbunden und ist somit auch für den überregionalen Energieaustausch zuständig.
- *Verteilnetz*:
 - *Hochspannung* (60kV-150kV): In dieser Ebene findet die erste grobe Verteilung in verschiedene Regionen und zu Ballungszentren statt, aber auch große Industriebetriebe sind hier an das Stromnetz angeschlossen. Zudem wird Energie von mittelgroßen Kraftwerken eingespeist.
 - *Mittelspannung* (1kV-30kV): Dieses Netz verteilt den Strom an einzelne Städte, Fabriken oder größere Einrichtungen wie zum Beispiel Krankenhäuser und bezieht wiederum Energie von kleineren städtischen Kraftwerken.
 - *Niederspannung* (230V/400V): In dieser Ebene findet die letzte feinste Verteilung statt. Hier werden private Haushalte mit Energie versorgt und können auch zum Beispiel mit kleineren Photovoltaikanlagen Strom in das Netz einspeisen.

¹Die Informationen über die Struktur des europäischen beziehungsweise des österreichischen Stromnetzwerks wurden entnommen aus [Wik13] und [oes13]

Diese Netze sind durch Umspannwerke und Transformatorstationen miteinander verbunden. Bisher waren Energiequellen in den niedrigeren Spannungsebenen eher selten, daher hatte man es in bestehenden Versorgungsnetzen mit einem unidirektionalen Leistungsfluss vom Umspannwerk hin zu den Lasten zu tun, wie man in Abbildung 1.1 sehen kann. Den vorgegebenen Wert für die jeweilige Span-

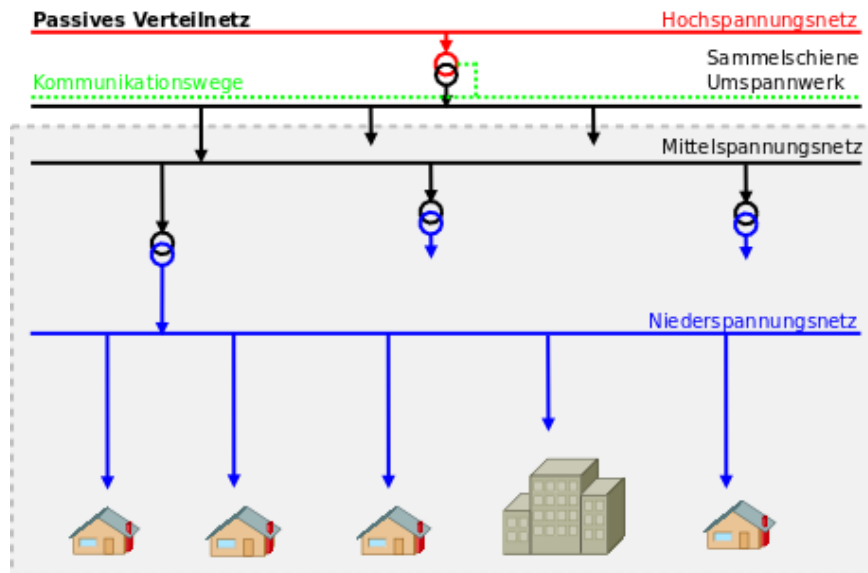


Abbildung 1.1.: Unidirektionaler Leistungsfluss

nungsebene bezeichnet man mit *Nennspannung*. Natürlich kann dieser nicht über das ganze Netz exakt eingehalten werden, aber es gibt ein erlaubtes Spannungsband, dessen Grenzen in der Regel vom Netzbetreiber vorgegeben werden und keinesfalls unter- beziehungsweise überschritten werden dürfen. Für Verteilnetze, in denen Energie nur in eine Richtung fließt, ist die Spannung am Einspeisepunkt am höchsten und fällt dann bis hin zu den Lasten kontinuierlich ab. Damit gestaltet sich die Spannungs-

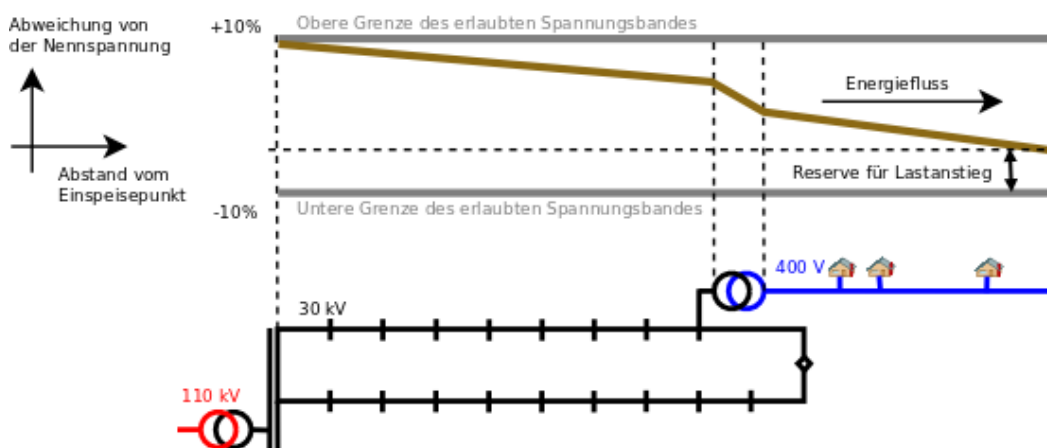


Abbildung 1.2.: Spannungsverlauf an einem exemplarischen Netzausläufer - Nur Last

regelung relativ einfach, indem man den Stufentransformator so einstellt, dass er die Spannung an die obere Grenze des erlaubten Spannungsbandes regelt und somit dessen ganze Breite verwendet wer-

den kann. Abbildung 1.2 zeigt grob skizziert die Abweichung ΔU der tatsächlichen Spannung von der Nennspannung U in Abhängigkeit von der Entfernung zum Umspannwerk für einen exemplarischen Ausläufer eines Netzes mit unidirektionalem Leistungsfluss.

1.2. Probleme für die Spannungsregelung bei großem Zuwachs an Erzeugern

Die Integration von verteilten Energieerzeugern in die niedrigeren Spannungsebenen eines Stromnetzes führt dazu, dass die Leistung nicht mehr nur von den Transformatorstationen hin zu den Verbrauchern fließt, sondern auch in die andere Richtung von den neuen Kraftwerken zu den Umspannwerken. Daher kann es zu Zeiten niedriger Last und maximaler Einspeisung entlang eines Netzausläufers auch zu einem Spannungsanstieg kommen, und es ist nicht mehr möglich, den Stufenschalter auf die obere Grenze des erlaubten Spannungsbandes regeln zu lassen. Die Abweichung von der Nennspannung für dieses Szenario ist in Abbildung 1.3 mit der oberen Linie eingezeichnet. Die untere Linie beschreibt den Verlauf von ΔU für die andere Extremsituation, also für den Fall, dass keine Energie eingespeist, aber sehr viel bezogen wird. Man sieht also, dass das Netz bei einem Zuwachs an Erzeugern bis an

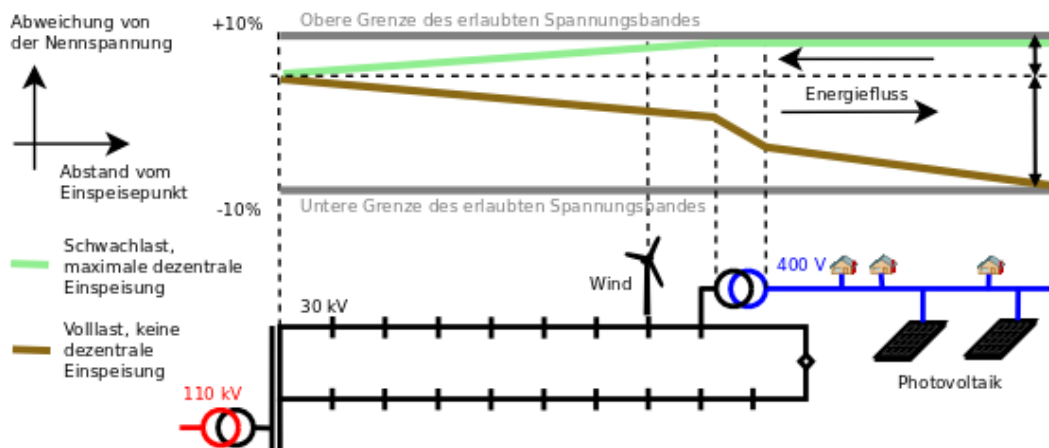


Abbildung 1.3.: Spannungsverlauf an einem exemplarischen Netzausläufer - Last und Erzeugung

seine Grenzen genutzt wird und die Vorgaben an die Spannung beispielsweise bei einem eventuellen Lastanstieg schon verletzt werden können. Nun gibt es einige Möglichkeiten, diesem Problem entgegenzuwirken: Der naheliegendste Ansatz wäre es, das vorhandene Netz auszubauen und zu verstärken, was aber enorme Kosten bedeuten würde. Desweiteren werden in [Vie+10] noch die Verringerung des Widerstands oder die Umgestaltung der Netzwerktopologie als potentielle, aber schwer realisierbare und damit wesentlich teurere Lösungen erwähnt.

1.3. Projekt des Austrian Institute of Technology zur Spannungsregelung - Smart Grid

Das Team vom Energy Department des AIT entschied sich dafür, das aufgezeigte Problem durch die Einführung eines Kommunikationssystems in das Verteilnetz zu lösen. Dafür wird die Infrastruktur des bestehenden Netzes verwendet und eine zentrale Steuerungseinheit bei der Transformatorstation eingebaut. Außerdem werden Messstationen an den sogenannten kritischen Knoten des Netzes angebracht, die den Regler mit den nötigen Informationen versorgen. Unter kritischen Knoten versteht man jene Knoten, an denen die Spannung im Verlauf eines Jahres einmal einen maximalen oder minimalen Wert im Netz angenommen hat, und sie werden durch eine einjährige Simulation ermittelt. Befindet sich nun

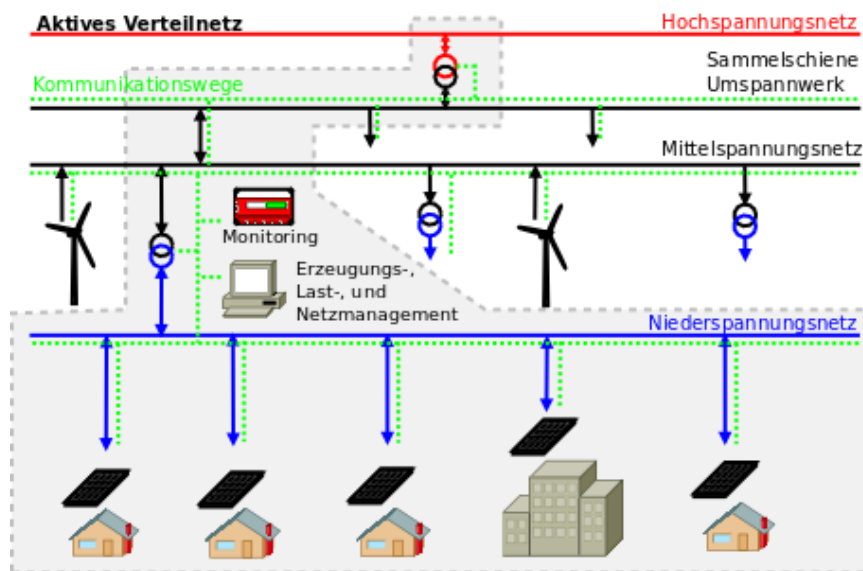


Abbildung 1.4.: Bidirektionaler Leistungsfluss

die Spannung an einem der kritischen Knoten außerhalb des erlaubten Bandes, so erfährt das zentrale Regler sofort und dieser weist daraufhin den Stufenschalter am Transformator an, die Spannung zu erhöhen oder zu senken. Etwas problematischer wird es, wenn die Spannung an zwei Knoten den zugelassenen Wertebereich verlässt, wobei sie einmal über der oberen Grenze und einmal unter der unteren Grenze liegt. In diesem Fall lässt sich das Problem nicht mit der Stufenschaltung allein lösen. Es gibt aber noch eine andere Möglichkeit, wie man die Spannung in einem Stromnetz beeinflussen kann. Denn es trägt auch die Menge an Wirkleistung P und Blindleistung Q , die von den verteilten Energiequellen in das Netz eingespeist wird, einen Teil zu der aktuellen Spannung bei. Daher ist es zusätzlich notwendig, dass der zentrale Regler Zugriff auf die aktuellen Werte von P und Q für die einzelnen Kraftwerke hat, sowie die Möglichkeit, diese gegebenenfalls zu verändern. Wie sich die Änderungen an der Menge der Wirk- beziehungsweise Blindleistung, die von den Erzeugern eingespeist wird, auf die Werte der Spannung an den kritischen Knoten auswirken, wird durch ein linearisiertes

Modell der Form

$$\Delta U = A_Q \Delta Q + A_P \Delta P \quad (1.1)$$

beschrieben. Hier ist ΔU ein m -dimensionaler Vektor, der die Änderungen der Spannung an den kritischen Knoten angibt. Die n -dimensionalen Vektoren ΔQ und ΔP beschreiben die Änderungen der eingespeisten Blind- beziehungsweise Wirkleistung der Kraftwerke und A_Q und A_P sind $(m \times n)$ -Matrizen. Auf den Fehler, der durch die Linearisierung des Modells entsteht, wird später noch eingegangen. Mit Hilfe von (1.1) kann man nun die Spannung an den kritischen Knoten so verändern, dass die Differenz des maximalen und des minimalen Wertes für U die Breite des erlaubten Spannungsbandes nicht übersteigt, und dann kann der Stufenschalter wieder in den richtigen Bereich regeln. Man muss allerdings noch zwei von den Kraftwerken gestellte Bedingungen berücksichtigen. Einerseits muss ein gewisser Leistungsfaktor eingehalten werden:

$$|Q_i + \Delta Q_i| \leq (P_i + \Delta P_i) \tan(\varphi). \quad (1.2)$$

Ein typischer Wert hierfür ist $PF_{min} = \cos(\varphi) = 0.9$. Andererseits darf auch die eingespeiste Scheinleistung S einen gewissen Maximalwert, üblicherweise durch die Erzeugungskapazität des Kraftwerks bestimmt, nicht überschreiten:

$$S_i = \sqrt{(Q_i + \Delta Q_i)^2 + (P_i + \Delta P_i)^2} \leq S_{max,i}. \quad (1.3)$$

1.3.1. Spannungsregelung durch Steuerung der Blindleistung

Aus Kostengründen versucht man im ersten Schritt, nur durch Variation der Werte der Blindleistung die Differenz der Spannungen an den kritischen Knoten auf die vorgegebene Bandbreite zu reduzieren. Zusätzlich versucht man aber auch die von den Kraftwerken in das Netz eingespeiste Blindleistung möglichst gering zu halten. Mit (1.1) - (1.3) führt dies zu folgendem Optimierungsproblem:

$$\begin{aligned} & \min \|Q + \Delta Q\|_1 \\ & \text{s. t. } \max(U + A_Q \Delta Q) - \min(U + A_Q \Delta Q) \leq EVB, \\ & |Q_i + \Delta Q_i| \leq P_i \tan(\arccos(PF_{min})), \quad i = 1, \dots, n, \\ & |Q_i + \Delta Q_i| \leq \sqrt{S_{max,i}^2 - P_i^2}, \quad i = 1, \dots, n. \end{aligned} \quad (1.4)$$

Wie zuvor ist hier U der m -dimensionale Vektor, der die aktuellen Werte der Spannung an den kritischen Knoten enthält. Q und P sind die n -dimensionalen Vektoren mit den aktuellen Werten der Blind- beziehungsweise Wirkleistung der einzelnen Erzeuger, A_Q ist eine $(m \times n)$ -Matrix, EVB die erlaubte Breite des Spannungsbandes, PF_{min} eine Zahl, die den Leistungsfaktor beschreibt, S_{max} der n -dimensionale Vektor mit den maximalen Scheinleistungen der Kraftwerke und ΔQ der gesuchte n -dimensionale Vektor mit den Änderungen an der eingespeisten Blindleistung. Mit $\|\cdot\|_1$ ist die 1-Norm für einen n -dimensionalen Vektor gemeint, also $\|x\|_1 = \sum_{i=1}^n |x_i|$. Wie wir später sehen werden, kann

(1.4) auf ein konvexes, ja sogar auf ein lineares Optimierungsproblem umformuliert werden und ist somit relativ einfach mit Hilfe eines primal-dualen Innere-Punkte-Verfahrens lösbar.

1.3.2. Spannungsregelung durch Steuerung der Blindleistung und der Wirkleistung

Können die Bedingungen in (1.4) nicht erfüllt werden und wird somit keine zulässige Lösung gefunden, so muss zusätzlich zur Blindleistung auch noch die Wirkleistung variieren. Für diesen Fall wird allerdings ebenfalls aus Kostengründen versucht, die Änderungen in den Werten der von den Erzeugern eingespeisten Wirkleistung minimal zu halten. Außerdem wird nur eine Reduktion der Wirkleistung zugelassen. Damit ergibt sich folgendes Optimierungsproblem:

$$\begin{aligned}
 & \min \|\Delta P\|_1 \\
 & \text{s. t. } \max(U + A_Q \Delta Q + A_P \Delta P) - \min(U + A_Q \Delta Q + A_P \Delta P) \leq EVB, \\
 & |Q_i + \Delta Q_i| \leq (P_i + \Delta P_i) \tan(\arccos(PF_{\min})), \quad i = 1, \dots, n, \\
 & \sqrt{(Q_i + \Delta Q_i)^2 + (P_i + \Delta P_i)^2} \leq S_{\max,i}, \quad i = 1, \dots, n, \\
 & -P_i \leq \Delta P_i \leq 0, \quad i = 1, \dots, n.
 \end{aligned} \tag{1.5}$$

Die Notation ist dieselbe wie in (1.4). Zudem ist mit ΔP der n -dimensionale Vektor mit den Änderungen an der Wirkleistung gemeint und A_P ist wieder eine $(m \times n)$ -Matrix. Aufgrund der dritten Zeile in den Nebenbedingungen lässt sich (1.5) zwar nicht in ein lineares Optimierungsproblem umformulieren, sehr wohl aber in ein konvexes. Somit gilt auch hier, dass es sich sehr gut durch ein primal-duales Innere-Punkte-Verfahren lösen lässt.

Zusammenfassend ist festzuhalten, dass für dieses Projekt ein Kommunikationssystem, durch Simulationen ermittelte kritische Knoten und zugehörige Messstationen sowie eine zentrale Steuerungseinheit nötig sind, wobei letztere sowohl die Stufenschaltung am Transformator als auch die von den verteilten Erzeugern eingespeiste Blind- und Wirkleistung kontrollieren kann. Wie in Abbildung 1.5

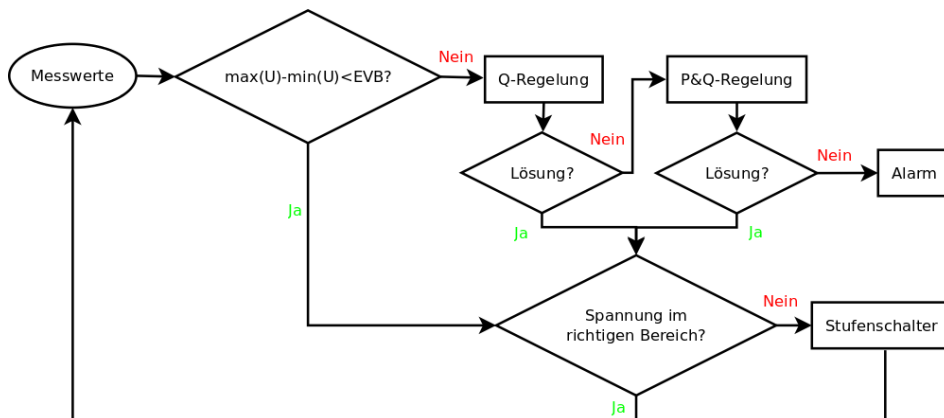


Abbildung 1.5.: Vereinfachter Flowchart für die Funktionsweise der zentralen Steuerungseinheit

dargestellt sieht nun die Vorgehensweise des Reglers folgendermaßen aus: Zuerst wird überprüft, ob die Differenz der Spannungen an den kritischen Knoten größer ist als die erlaubte Spannungsbandbreite. Ist dies der Fall, so wird im nächsten Schritt die Spannungsregelung durch Steuerung der Blindleistung also ein Algorithmus zur Lösung von (1.4) aufgerufen. Werden dafür keine zulässigen Werte gefunden, dann wird ein Programm zum Ermitteln einer Lösung von (1.5) gestartet, welches eigentlich immer sinnvolle Daten zurückgeben sollte. Sobald eine genügend kleine Differenz der Spannungen vorhanden ist, wird überprüft, ob auch alle Werte innerhalb der vorgegebenen Grenzen liegen und gegebenenfalls mit Hilfe der Stufenschaltung auf die richtige Spannungshöhe geregelt.

Ziel dieser Arbeit ist es nun, möglichst schnelle Algorithmen zum Auffinden der Lösungen von (1.4) und (1.5) in Form von MATLAB-Dateien zu verfassen, welche dann mittels eines Interpreters von der zentralen Steuerungseinheit aufgerufen werden können.

2. Einführung in die konvexe Optimierung

Hier wird kurz auf Teile des mathematischen Grundgerüsts eingegangen, das nötig ist, um die Probleme, die im vorigen Kapitel vorgestellt wurden, zu lösen. Es wird also beschrieben, was man unter einem Optimierungsproblem, unter konvexen Mengen und Funktionen und somit letztendlich unter einem konvexen Optimierungsproblem versteht und welche notwendigen und hinreichenden Bedingungen es an eine eventuelle Lösung gibt. Danach wird noch ein Innere-Punkte-Verfahren (*Interior Point Method*) vorgestellt, das im Rahmen dieser Arbeit zum Auffinden der Lösung verwendet wird. Näheres zum Thema konvexe Optimierung sowie den hier benutzten Algorithmus findet man in [BV04] und Genaueres zu (primal-dualen) Innere-Punkte-Verfahren in [Wri97] oder [Ye11].

2.1. Grundlagen

Um ein Optimierungsproblem zu lösen, sollte erst die Terminologie erklärt werden und darum wollen wir mit einigen Definitionen beginnen.

Definition 2.1 (Optimierungsproblem). Seien $f_i, h_j : \mathbb{R}^n \rightarrow \mathbb{R}$ für $i = 0, \dots, m$ und $j = 1, \dots, p$ Funktionen. Dann betrachten wir das *Optimierungsproblem* der Form

$$\begin{aligned} \min f_0(x) \\ \text{s. t. } f_i(x) &\leq 0, \quad i = 1, \dots, m, \\ h_j(x) &= 0, \quad j = 1, \dots, p. \end{aligned} \tag{2.1}$$

Hier werden $f_i(x) \leq 0$, $i = 1, \dots, m$ die *Ungleichungsnebenbedingungen* und $h_j(x) = 0$, $j = 1, \dots, p$ die *Gleichungsnebenbedingungen* genannt. Dabei sei

$$C := \{x \in \mathbb{R}^n \mid f_i(x) \leq 0, h_j(x) = 0 \text{ für } i = 1, \dots, m \text{ und } j = 1, \dots, p\} \tag{2.2}$$

der *zulässige Bereich* unseres Optimierungsproblems. Ein Punkt $x^* \in \mathbb{R}^n$ heißt *lokales Minimum* von (2.1), wenn es eine offene Umgebung $U \subseteq \mathbb{R}^n$ gibt, mit $x^* \in U \cap C$ und $f_0(x^*) \leq f_0(x)$ für alle $x \in U \cap C$. Gilt sogar $f_0(x^*) \leq f_0(x)$ für alle $x \in C$, dann nennt man x^* ein *globales Minimum* von (2.1).

Klarerweise ist jedes globale Minimum auch ein lokales Minimum. Wie sich später noch zeigen wird, gilt für konvexe Optimierungsprobleme die Umkehrung auch. Das ist ein wichtiger Punkt, warum diese unter den Optimierungsproblemen so einen besonderen Stellenwert genießen. Doch bevor wir dazu kommen, müssen wir erst einmal genauer beschreiben, was man unter Konvexität versteht.

Definition 2.2 (Konvexe Mengen und Funktionen). Eine Menge $C \subseteq \mathbb{R}^n$ heißt *konvex*, wenn mit je zwei Punkten in C auch die Strecke zwischen diesen Punkten in C enthalten ist, das heißt, wenn gilt:

$$\overline{xy} := \{ty + (1-t)x \mid t \in [0, 1]\} \subseteq C \quad \forall x, y \in C. \quad (2.3)$$

Sei nun C eine konvexe Menge. Eine Funktion $f : C \rightarrow \mathbb{R}$ nennt man dann *konvex*, wenn sie

$$f(ty + (1-t)x) \leq tf(y) + (1-t)f(x) \quad \forall x, y \in C, t \in [0, 1] \quad (2.4)$$

erfüllt. Ist die Ungleichung in (2.4) strikt für $t \in (0, 1)$ und $x \neq y$, gilt sie für $t \in (0, 1)$ also auch, wenn man $\leq$ durch $<$ ersetzt, so spricht man von einer *strikt konvexen* Funktion. Eine mehrdimensionale Funktion $f : C \rightarrow \mathbb{R}^m$,

$$f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix},$$

heißt (*strikt*) *konvex*, wenn jede Komponentenfunktion $f_i : C \rightarrow \mathbb{R}$ ($i = 1, \dots, m$) (*strikt*) konvex ist. Eine Funktion $f : C \rightarrow \mathbb{R}^m$ heißt (*strikt*) *konkav*, wenn $-f$ (*strikt*) konvex ist.

Beispiel 2.3 (Konvexe Funktionen). Sei $f : \mathbb{R}^n \rightarrow \mathbb{R}$ linear, das heißt $f(x) = a^T x + b$ für $a \in \mathbb{R}^n, b \in \mathbb{R}$. Dann gilt in (2.4) Gleichheit, also ist f konvex.

Mit dem Wissen über konvexe Mengen und Funktionen kann man nun ein konvexes Optimierungsproblem definieren.

Definition 2.4 (Konvexes Optimierungsproblem). Problem (2.1) heißt *konvexes Optimierungsproblem*, wenn sowohl die Zielfunktion f_0 als auch der zulässige Bereich C (siehe (2.2)) konvex sind.

Beispiel 2.5 (Konvexe Mengen).

- (i) Sei $f : \mathbb{R}^n \rightarrow \mathbb{R}$ eine konvexe Funktion. Dann ist $C := \{x \in \mathbb{R}^n \mid f(x) \leq 0\}$ eine konvexe Menge, da mit $x, y \in C$ auch $\overline{xy}$ in C enthalten ist: Sei $t \in [0, 1]$. Dann gilt wegen der Konvexität von f

$$f(tx + (1-t)y) \leq t \underbrace{f(x)}_{\leq 0} + (1-t) \underbrace{f(y)}_{\leq 0} \leq 0.$$

Analog ist leicht zu sehen, dass für eine lineare Funktion $h : \mathbb{R}^n \rightarrow \mathbb{R}$ die Menge $C' := \{x \in \mathbb{R}^n \mid h(x) = 0\}$ konvex ist, da hierfür in obiger Ungleichung überall Gleichheit gilt.

- (ii) Seien $C_1, \dots, C_m$ konvexe Teilmengen des $\mathbb{R}^n$. Dann ist auch der Schnitt dieser Mengen $C := \bigcap_{i=1}^m C_i$ konvex, weil für $x, y \in C$ die Strecke $\overline{xy}$ in jedem C_i ($i = 1, \dots, m$) enthalten ist und somit auch in der Schnittmenge C .

Damit lässt sich nun feststellen, dass (2.1) ein konvexes Optimierungsproblem ist, wenn die Zielfunktion f_0 und die Funktionen $f_1, \dots, f_m$ aus den Ungleichungsnebenbedingungen konvex und die Funktionen $h_1, \dots, h_p$ aus den Gleichungsnebenbedingungen linear sind. Denn dann beschreibt jede Nebenbedingung eine konvexe Menge C_i für $i = 1, \dots, m + p$ und der zulässige Bereich C ist als Schnitt dieser konvexer Mengen auch konvex.

Nun wollen wir uns noch ein wenig mit Eigenschaften konvexer Funktionen beschäftigen: Welche Kriterien gibt es für Konvexität? Welche Operationen erhalten Konvexität?

Lemma 2.6 (Eigenschaften konvexer Funktionen). *Sei $C \subseteq \mathbb{R}^n$ eine konvexe Menge.*

- (i) *Bedingungen erster Ordnung: Eine stetig differenzierbare Funktion $f : C \rightarrow \mathbb{R}$ ist genau dann konvex, wenn*

$$f(x) + \nabla f(x)^T(y - x) \leq f(y) \quad \forall x, y \in C \quad (2.5)$$

gilt. f ist genau dann strikt konvex, wenn die Ungleichung in (2.5) für $x \neq y$ strikt ist.

- (ii) *Bedingungen zweiter Ordnung: Eine zweimal stetig differenzierbare Funktion $f : C \rightarrow \mathbb{R}$ ist genau dann konvex, wenn die Hesse-Matrix von f auf ganz C positiv semidefinit ist, das heißt wenn*

$$\nabla^2 f(x) \succeq 0 \quad \forall x \in C \quad (2.6)$$

gilt. Ist $\nabla^2 f(x)$ sogar positiv definit, dann ist f strikt konvex.

- (iii) *Summe konvexer Funktionen: Sind $f, g : C \rightarrow \mathbb{R}$ konvex $\alpha > 0$ eine positive reelle Zahl, so ist auch $f + \alpha g$ konvex.*

Beweis. (i) Sei f konvex und seien $x, y \in C$. Für $x = y$ ist (2.5) klarerweise erfüllt. Sei also $x \neq y$. Dann gilt

$$\begin{aligned} \nabla f(x)^T(y - x) &= \lim_{t \rightarrow 0} \frac{f(x + t(y - x)) - f(x)}{t} = \lim_{t \rightarrow 0} \frac{f(ty + (1 - t)x) - f(x)}{t} \\ &\stackrel{(2.4)}{\leq} \lim_{t \rightarrow 0} \frac{tf(y) + (1 - t)f(x) - f(x)}{t} = \lim_{t \rightarrow 0} \frac{t(f(y) - f(x))}{t} \\ &= f(y) - f(x) \end{aligned}$$

und das ist genau (2.5). Nehmen wir nun umgekehrt an, dass f (2.5) erfülle und seien wiederum $x, y \in C, x \neq y$. Außerdem sei $t \in (0, 1)$ und $z := tx + (1 - t)y$. Weil C konvex ist, gilt $z \in C$ und so können wir zweimal (2.5) anwenden:

$$\begin{aligned} f(z) + \nabla f(z)^T(x - z) &\leq f(x) \\ f(z) + \nabla f(z)^T(y - z) &\leq f(y) \end{aligned}$$

Multipliziert man jetzt die erste Ungleichung mit t , die zweite mit $(1 - t)$ und addiert die beiden, dann erhält man nach kurzem Umformen

$$f(z) \leq tf(x) + (1 - t)f(y)$$

und somit ist die Konvexität von f gezeigt.

- (ii) Wir wollen zuerst zeigen, dass für konvexe Funktionen die Hesse-Matrix positiv semidefinit ist. Dazu betrachten wir anfangs eindimensionale Funktionen. Sei also $f : \mathbb{R} \supseteq C \rightarrow \mathbb{R}$ konvex. In diesem Fall ist C ein Intervall. Für $x, y \in C$ gilt dann mit (2.5)

$$\begin{aligned} f(x) + f'(x)(y - x) &\leq f(y) \\ f(y) + f'(y)(x - y) &\leq f(x) \end{aligned}$$

und durch Addieren der beiden Ungleichungen erhält man

$$(f'(x) - f'(y))(y - x) \leq 0.$$

f' ist also monoton wachsend. Damit sieht man schnell, dass

$$f''(x) = \lim_{t \rightarrow 0} \frac{f'(x+t) - f'(x)}{t} \geq 0$$

gelten muss. Wenden wir uns nun dem Fall $C \subseteq \mathbb{R}^n$ zu. Angenommen, es gäbe ein $x \in C$ mit $\nabla^2 f(x) \prec 0$, dann wählen wir ein $y \in C$ mit $x \neq y$ und betrachten die eindimensionale Funktion $g(t) := f(ty + (1 - t)x)$. Durch Einsetzen in (2.4) ist leicht zu überprüfen, dass g konvex ist, wenn f konvex ist. Für die zweite Ableitung von g an der Stelle $t = 0$ gilt

$$g''(0) = (y - x)^T \nabla^2 f(x) (y - x) < 0.$$

Also kann g nicht konvex sein und somit auch f nicht, was unseren Annahmen widerspricht. Nun müssen wir noch die Umkehrung zeigen, das heißt es gelte (2.6). Seien $x, y \in C$. Da f zweimal stetig differenzierbar ist, gibt es nach dem Satz von Taylor¹ ein $t \in [0, 1]$, sodass

$$f(y) = f(x) + \nabla f(x)^T (y - x) + \frac{1}{2} (y - x)^T \nabla^2 f(x + t(y - x)) (y - x)$$

gilt. Weil $x + t(y - x) \in C$ ist, ist wegen (2.6) der letzte Summand ≥ 0 und damit folgt (2.5), was die Konvexität von f impliziert.

- (iii) Ist sofort durch Einsetzen in (2.4) ersichtlich.

□

¹Siehe zum Beispiel [Heu04], Kapitel XX.168

Mit Hilfe dieser Kriterien können wir nun noch ein für unsere Probleme relevantes Beispiel für konvexe Funktionen angeben.

Beispiel 2.7 (Konvexe Funktionen). Sei $Q \in \mathbb{R}^{n \times n}$ eine positiv semidefinite, quadratische $(n \times n)$ -Matrix, $a \in \mathbb{R}^n$ und $b \in \mathbb{R}$. Dann ist die quadratische Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $f(x) := x^T Q x + a^T x + b$ konvex, denn es gilt:

$$\nabla^2 f(x) = Q \succeq 0.$$

2.2. Dualität

Wir haben in Abschnitt 2.1 gesehen, dass ein Optimierungsproblem nur dann konvex sein kann, wenn die Funktionen h_j in den Gleichungsnebenbedingungen $h_j(x) = 0$ linear sind. Diese Nebenbedingungen lassen sich aber umformen auf die dazu äquivalenten Ungleichungen

$$\begin{aligned} -h_j(x) &\leq 0 & j = 1, \dots, p, \\ h_j(x) &\leq 0 & j = 1, \dots, p, \end{aligned}$$

wobei ja h_j und $-h_j$ konvexe Funktionen sind. Da sich die im ersten Kapitel vorgestellten Probleme in konvexe Form umschreiben lassen, wie wir noch sehen werden, und sie zudem nicht einmal Gleichungsnebenbedingungen beinhalten, betrachten wir von nun an nur noch das konvexe Optimierungsproblem mit Ungleichungsnebenbedingungen

$$\begin{aligned} \min & f_0(x) \\ \text{s. t. } & f_i(x) \leq 0, \quad i = 1, \dots, m, \end{aligned} \tag{2.7}$$

mit $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$ konvex für $i = 0, \dots, m$.

Um in weiterer Folge Optimalitätskriterien für Lösungen von (2.7) zu erhalten, beschäftigen wir uns zunächst mit dem sogenannten dualen Problem und beginnen dafür wieder mit einer Definition.

Definition 2.8 (Lagrange-Funktion, Lagrange-duale Funktion, Duales Problem).

(i) Die *Lagrange-Funktion* $L : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}$ zu (2.7) ist definiert als

$$L(x, \lambda) := f_0(x) + \sum_{i=1}^m \lambda_i f_i(x). \tag{2.8}$$

Der Vektor λ heißt *Lagrange-Multiplikator* oder *duale Variable*.

(ii) Das punktweise Infimum der Lagrange-Funktion über alle $x \in \mathbb{R}^n$, also die Funktion $g : \mathbb{R}^m \rightarrow \mathbb{R}$,

$$g(\lambda) := \inf_{x \in \mathbb{R}^n} L(x, \lambda) \tag{2.9}$$

nennt man die *Lagrange-duale Funktion* zu (2.7).

(iii) Das Optimierungsproblem

$$\begin{aligned} \max g(\lambda) \\ \text{s. t. } \lambda \geq 0 \end{aligned} \tag{2.10}$$

heißt das zu (2.7) *duale Problem*.

Dazu ist anzumerken, dass die Lagrange-duale Funktion $g(\lambda)$ immer konkav ist, selbst wenn das ursprüngliche Problem nicht konvex ist. Um das zu sehen, bemerke man erst, dass die Lagrange-Funktion $L(x, \lambda)$ linear in λ ist, und betrachte dann

$$\begin{aligned} g(t\lambda + (1-t)\tilde{\lambda}) &= \inf_{x \in \mathbb{R}^n} L(x, t\lambda + (1-t)\tilde{\lambda}) \\ &= \inf_{x \in \mathbb{R}^n} (tL(x, \lambda) + (1-t)L(x, \tilde{\lambda})) \\ &\geq t \inf_{x \in \mathbb{R}^n} L(x, \lambda) + (1-t) \inf_{x \in \mathbb{R}^n} L(x, \tilde{\lambda}) \\ &= tg(\lambda) + (1-t)g(\tilde{\lambda}). \end{aligned}$$

Also ist das duale Problem (2.10) immer ein konvexes Optimierungsproblem, da es äquivalent ist zu

$$\begin{aligned} \min -g(\lambda) \\ \text{s. t. } \lambda \geq 0. \end{aligned}$$

Bezeichnen wir nun mit x^* die Lösung von (2.7) und mit λ^* die Lösung von (2.10) und nehmen an, dass x *primal zulässig* und λ *dual zulässig* sei, was bedeutet, dass

$$\begin{aligned} f_i(x) &\leq 0 \quad i = 1, \dots, m \\ \lambda &\geq 0 \\ g(\lambda) &> -\infty \end{aligned}$$

gelte, dann folgt:

$$g(\lambda) \leq g(\lambda^*) =: d^* \leq p^* := f(x^*) \leq f(x). \tag{2.11}$$

Diese Eigenschaft heißt *schwache Dualität* und $\Delta := p^* - d^*$ nennt man die *Dualitätslücke*. Gilt $\Delta = 0$, so spricht man von *starker Dualität*. Dies ist zum Beispiel der Fall, wenn das Optimierungsproblem konvex ist und gewisse *Constraint Qualifications (CQ)* erfüllt sind. Eines der wichtigsten Beispiele für Constraint Qualifications ist die sogenannte *Slater-Bedingung*², die fordert, dass ein $x \in \mathbb{R}^n$ existiert

²Einen Beweis dafür, dass die Slater-Bedingung starke Dualität impliziert, findet man zum Beispiel in [BV04], Kapitel 5.3.2

mit

$$f_i(x) < 0$$

für alle $i = 1, \dots, m$ mit f_i nichtlinear.

2.3. Optimalitätsbedingungen

In diesem Kapitel wollen wir notwendige Bedingungen für Lösungen von Optimierungsproblemen finden. Für konvexe Probleme werden wir auch hinreichende Kriterien angeben.

2.3.1. Unrestringierte Probleme

Zuerst betrachten wir Probleme ohne Nebenbedingungen, also

$$\min f(x), \tag{2.12}$$

wobei $f : \mathbb{R}^n \rightarrow \mathbb{R}$ eine differenzierbare Funktion ist.

Satz 2.9 (Notwendige Bedingung für unrestringierte Probleme). *Sei $f : \mathbb{R}^n \rightarrow \mathbb{R}$ differenzierbar und $x^* \in \mathbb{R}$ ein lokales Minimum von (2.12). Dann gilt*

$$\nabla f(x^*) = 0. \tag{2.13}$$

Beweis. Wir zeigen zunächst wieder den eindimensionalen Fall. Sei also $x^* \in \mathbb{R}$ ein lokales Minimum von $f : \mathbb{R} \rightarrow \mathbb{R}$. Dann gibt es ein Intervall $I = (x^* - \alpha, x^* + \alpha)$ mit $\alpha > 0$, auf dem

$$f(y) \geq f(x^*) \quad \forall y \in I$$

gilt. Da f differenzierbar ist, müssen der linksseitige und der rechtsseitige Grenzwert des Differenzenquotienten übereinstimmen.

$$\lim_{t \nearrow 0} \frac{f(x^* + t) - f(x^*)}{t} = \lim_{t \searrow 0} \frac{f(x^* + t) - f(x^*)}{t}.$$

Da aber für t klein genug der linke Quotient immer nichtpositiv und der rechte immer nichtnegativ ist, gilt das auch für deren Grenzwert und somit folgt $f'(x^*) = 0$. Sei nun allgemeiner $x^* \in \mathbb{R}^n$ ein lokales Minimum von (2.12) und $y \in \mathbb{R}^n$ beliebig. Dann ist 0 ein lokales Minimum der eindimensionalen Funktion $g : \mathbb{R} \rightarrow \mathbb{R}$, $g(t) := f(x^* + ty)$. Es gilt also

$$0 = g'(0) = y^T \nabla f(x^*)$$

und da y beliebig gewählt war, folgt somit (2.13). □

Bemerkung 2.10. Für konvexe f gilt auch die Umkehrung. Es folgt sogar globale Optimalität, denn für $x^* \in \mathbb{R}^n$ mit $\nabla f(x^*) = 0$ haben wir wegen (2.5)

$$f(x^*) \leq f(y) \quad \forall y \in \mathbb{R}^n.$$

2.3.2. Probleme mit Nebenbedingungen

Unser nächster Schritt wird es sein, notwendige und hinreichende Kriterien für eine Lösung unseres konvexen Standardproblems mit Nebenbedingungen (2.7) zu finden. Diese heißen *Karush-Kuhn-Tucker-Bedingungen* oder kurz *KKT-Bedingungen* und sind im folgenden Satz formuliert.

Satz 2.11 (KKT-Bedingungen für konvexe Optimierungsprobleme).

Seien $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$ für $i = 0, \dots, m$ konvexe differenzierbare Funktionen, $x^* \in \mathbb{R}^n$ und $\lambda^* \in \mathbb{R}^m$. Die folgenden Aussagen sind äquivalent:

- (i) Für (2.7) und (2.10) gilt starke Dualität und x^* und λ^* sind die entsprechenden Lösungen.
- (ii) Es gelten die KKT-Bedingungen:

$$\begin{aligned} f_i(x^*) &\leq 0, & i &= 1, \dots, m, \\ \lambda_i^* &\geq 0, & i &= 1, \dots, m, \\ \lambda_i^* f_i(x^*) &= 0, & i &= 1, \dots, m, \\ \nabla f_0(x^*) + \sum_{i=1}^m \lambda_i^* \nabla f_i(x^*) &= 0. \end{aligned} \tag{2.14}$$

Beweis.

- (i) $\implies$ (ii) Zuerst stellen wir fest, dass x^* und λ^* als Lösungen von (2.7) und (2.10) natürlich auch primal und dual zulässig sein müssen und somit sind die ersten beiden Zeilen in (2.14) trivialerweise erfüllt. Außerdem impliziert die starke Dualität folgende Identität:

$$f_0(x^*) = g(\lambda^*) = \inf_{x \in \mathbb{R}^n} L(x, \lambda^*) \leq L(x^*, \lambda^*) = f_0(x^*) + \sum_{i=1}^m \lambda_i^* f_i(x^*) \leq f_0(x^*).$$

Es gilt also überall Gleichheit. Das führt zu zwei Erkenntnissen: Erstens muss

$$\sum_{i=1}^m \lambda_i^* f_i(x^*) = 0$$

sein und wegen $\lambda_i^* \geq 0$ und $f_i(x^*) \leq 0$ haben wir somit die dritte Zeile in (2.14). Zweitens ist x^* ein Minimum von $L(x, \lambda^*)$, also folgt mit Satz 2.9, dass der Gradient an dieser Stelle

verschwinden muss:

$$0 = \nabla_x L(x^*, \lambda^*) = \nabla f_0(x^*) + \sum_{i=1}^m \lambda_i^* \nabla f_i(x^*).$$

(ii) $\implies$ (i) Zuerst bemerke man wieder, dass x^* wegen der ersten beiden Zeilen in (2.14) primal zulässig ist. Außerdem ist $L(x, \lambda^*)$ konvex, da $\lambda_i^* \geq 0$ gilt, und hat daher aufgrund der letzten Zeile in (2.14) und mit Bemerkung 2.10 bei x^* ein globales Minimum. Somit haben wir

$$g(\lambda^*) = \inf_{x \in \mathbb{R}^n} L(x, \lambda^*) = L(x^*, \lambda^*) = f_0(x^*) + \sum_{i=1}^m \underbrace{\lambda_i^* f_i(x^*)}_{=0} = f_0(x^*).$$

Daraus folgen $\Delta = 0$ und die Optimalität von x^* und λ^* .

□

Bemerkung 2.12. Im ersten Teil des Beweises von Satz 2.11 wurde die Konvexität von f_i nicht verwendet. Das heißt, dass die Richtung (i) $\implies$ (ii) auch für nichtkonvexe Optimierungsprobleme der Form (2.7) gilt. Die KKT-Bedingungen (2.14) sind für allgemeine Probleme also notwendig, aber nicht hinreichend.

2.4. Algorithmen

In manchen Fällen lassen sich die KKT-Bedingungen analytisch lösen. Doch wir beschäftigen uns in diesem Kapitel damit, wie man mit Hilfe von iterativen Verfahren eine Lösung von (2.7) findet, wobei wir von nun an immer von einem konvexen Problem ausgehen.

2.4.1. Unrestringierte Probleme

Vorerst betrachten wir konvexe Optimierungsprobleme ohne Nebenbedingungen, also

$$\min f(x)$$

mit $f : \mathbb{R}^n \rightarrow \mathbb{R}$ konvex und differenzierbar. Für solch ein Problem gilt: x^* ist genau dann eine Lösung, wenn $\nabla f(x^*) = 0$. Die Grundidee ist es, von einem Startpunkt $x^{(0)}$ aus eine Folge $(x^{(k)})_k$ zu konstruieren, die gegen das Minimum x^* konvergiert. Von dieser Folge fordern wir, dass sie eine *Abstiegsfolge* ist, das heißt

$$f(x^{(k+1)}) < f(x^{(k)}) \tag{2.15}$$

für alle k , bis das Minimum erreicht wird. Doch wie kommt man zu solch einer Folge? Der erste Schritt ist es, einen Vektor Δx zu finden, in dessen Richtung $f(x)$ kleiner wird. Da f konvex ist, wissen wir

$f(x^{(k)} + \Delta x) - f(x^{(k)}) \geq \nabla f(x^{(k)})^T \Delta x^{(k)}$, also muss

$$\nabla f(x^{(k)})^T \Delta x^{(k)} < 0 \quad (2.16)$$

gelten. Ein Vektor Δx , der (2.16) erfüllt, heißt *Abstiegsrichtung*.

Beispiel 2.13 (Newton-Schritt). Da es für unrestringierte konvexe Probleme genügt, ein $x^* \in \mathbb{R}$ mit $\nabla f(x^*) = 0$ zu finden, ist eine naheliegende Idee, das aus der Numerik bekannte *Newton-Verfahren*³ zum Auffinden von Nullstellen auf ∇f anzuwenden. Ist also f zweimal differenzierbar und $\nabla^2 f(x) \succ 0$, dann ist der *Newton-Schritt*

$$\Delta x_{\text{nt}} := -\nabla^2 f(x)^{-1} \nabla f(x)$$

eine Abstiegsrichtung, da

$$\nabla f(x)^T \Delta x_{\text{nt}} = -\nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x) < 0$$

gilt, solange $\nabla f(x) \neq 0$ ist. Für $\nabla f(x) = 0$ hat man ohnehin bereits ein Minimum erreicht.

Im zweiten Schritt, bei der sogenannten *Liniensuche*, bestimmen wir entlang der Abstiegsrichtung $\{x^{(k)} + t\Delta x \mid t \in \mathbb{R}_+\}$ eine geeignete Schrittweite t für $x^{(k+1)} := x^{(k)} + t\Delta x$. Die grobe Struktur eines Verfahrens zur Lösung eines unrestringierten konvexen Problems ist in Algorithmus 2.1 dargestellt.

Sei ein Startpunkt $x \in \text{dom } f$ gegeben.

```
while Abbruchkriterium nicht erfüllt do
  Bestimme Abstiegsrichtung  $\Delta x$ 
  Liniensuche: Bestimme Schrittweite  $t$ 
   $x := x + t\Delta x$ 
end while
```

Algorithmus 2.1: Allgemeines Abstiegsverfahren

Eine Möglichkeit für die Liniensuche wäre die exakte Liniensuche, das heißt entlang der Richtung $x + t\Delta x$, $t > 0$, wirklich das Minimum zu suchen. Dies ist jedoch meist sehr aufwendig und nur sinnvoll, wenn das Bestimmen einer neuen Suchrichtung Δx verhältnismäßig kostspieliger ist. Eine andere Methode ist zum Beispiel die *Backtracking Liniensuche*, bei der man mit $t = 1$ startet und t immer weiter verkleinert, bis ein bestimmtes Kriterium, zum Beispiel eine Mindestreduktion des Zielfunktionswerts, erfüllt ist.

³Siehe zum Beispiel [Deu04]

Sei eine Abstiegsrichtung Δx für f bei $x \in \text{dom } f$ und $\alpha \in (0, 0.5)$, $\beta \in (0, 1)$ gegeben.

$t := 1$

while $f(x + t\Delta x) > f(x) + \alpha t \nabla f(x)^T \Delta x$ **do**

$t := \beta t$

end while

Algorithmus 2.2: Backtracking Liniensuche

2.4.2. Probleme mit Nebenbedingungen - Ein primal-duales Innere-Punkte-Verfahren

Wir versuchen nun einen Algorithmus zur Lösung des Problems

$$\begin{aligned} \min f_0(x) \\ \text{s. t. } f_i(x) \leq 0, \quad i = 1, \dots, m, \end{aligned} \tag{2.7}$$

zu finden, wobei $f_0, \dots, f_m : \mathbb{R}^n \rightarrow \mathbb{R}$ konvex und zweimal differenzierbar seien. Außerdem schreiben wir von nun an:

$$f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix}, \quad Df(x) = \begin{pmatrix} \nabla f_1(x)^T \\ \vdots \\ \nabla f_m(x)^T \end{pmatrix}.$$

Des Weiteren sei für $a \in \mathbb{R}^n$ mit $\text{diag}(a)$ die $(n \times n)$ -Diagonalmatrix gemeint, in deren Hauptdiagonale die Einträge von a stehen. Wir wissen, dass x^* das Problem genau dann löst, wenn es ein $\lambda^* \in \mathbb{R}^m$ gibt, sodass (x^*, λ^*) die KKT-Bedingungen

$$\begin{aligned} f(x^*) &\leq 0, \\ \lambda^* &\geq 0, \\ \text{diag}(\lambda^*) f(x^*) &= 0, \\ \nabla f_0(x^*) + Df(x^*)^T \lambda^* &= 0 \end{aligned}$$

erfüllt. So, wie wir in Beispiel 2.13 das Newton-Verfahren zur Lösung nichtlinearer Gleichungen auf $\nabla f(x)$ angewandt haben, ist nun die Idee, die Gleichungen in den KKT-Bedingungen als Funktion $\hat{r} : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^{n+m}$

$$\hat{r}(x, \lambda) = \begin{pmatrix} \nabla f_0(x) + Df(x)^T \lambda \\ -\text{diag}(\lambda) f(x) \end{pmatrix}$$

zu interpretieren und davon eine Nullstelle zu suchen. Ein Newton-Schritt für $\hat{r}$ an der Stelle (x, λ, ν) ist dann:

$$\begin{pmatrix} \Delta x_{\text{nt}} \\ \Delta \lambda_{\text{nt}} \end{pmatrix} = - \begin{pmatrix} \nabla^2 f_0(x) + \sum_{i=1}^m \lambda_i \nabla^2 f_i(x) & Df(x)^T \\ -\text{diag}(\lambda) Df(x) & -\text{diag}(f(x)) \end{pmatrix}^{-1} \begin{pmatrix} \nabla f_0(x) + Df(x)\lambda \\ -\text{diag}(\lambda)f(x) \end{pmatrix}.$$

Um die Ungleichungen in den KKT-Bedingungen kümmern wir uns, indem wir einen Startpunkt $(x^{(0)}, \lambda^{(0)}) \in \{(x, \lambda) \in \mathbb{R}^{n+m} \mid f(x) < 0, \lambda > 0\}$ wählen und mit Hilfe der Liniensuche nicht zulassen, dass dieser Bereich verlassen wird. Da man dabei aber trotzdem bemüht ist, möglichst große Schrittweiten zu erlangen, wird man sich sehr bald dem Rand des zulässigen Bereichs nähern, wo nur noch sehr kleine Schrittweiten möglich sind. Beziehungsweise kann es auch soweit kommen, dass kein weiterer Schritt mehr möglich ist, obwohl man sich noch nicht am Minimum befindet. Dieser Fall tritt dann ein, wenn die Suchrichtung nach außen zeigt und man sich schon so nahe am Rand befindet, dass die Liniensuche keinen weiteren Schritt mehr zulässt. Um das zu vermeiden, versuchen wir anfangs weit genug im Inneren des zulässigen Bereichs zu bleiben. Dazu benötigen wir eine weitere Definition.

Definition 2.14 (Der zentrale Pfad). Sei $t > 0$ eine positive reelle Zahl. Ein *zentraler Punkt* zu dem Optimierungsproblem (2.7) ist ein Punkt $x^*(t) \in \mathbb{R}^n$ für den ein $\lambda^*(t) \in \mathbb{R}^m$ existiert, sodass $(x^*(t), \lambda^*(t))$ die *modifizierten KKT-Bedingungen*

$$\begin{aligned} f(x^*(t)) &\leq 0, \\ \lambda^*(t) &\geq 0, \\ -\lambda_i^*(t) f_i(x^*(t)) &= \frac{1}{t}, \quad i = 1, \dots, m, \\ \nabla f_0(x^*(t)) + \sum_{i=1}^m f_i(x^*(t))^T \lambda_i^*(t) &= 0 \end{aligned} \tag{2.17}$$

erfüllt. Die Menge aller zentralen Punkte $x^*(t)$ mit $t > 0$ heißt der *zentrale Pfad* zu (2.7)

Die einzige Änderung in den modifizierten KKT-Bedingungen ist das Ersetzen von 0 durch $\frac{1}{t}$ in der dritten Zeile. Man beachte, dass $x^*(t)$ wegen der letzten Zeile von (2.17) die Funktion $L(x, \lambda^*(t))$ minimiert. Deshalb ist $g(\lambda^*(t)) = L(x^*(t), \lambda^*(t)) > -\infty$ und somit $\lambda^*(t)$ dual zulässig. Darüber hinaus gilt für die Dualitätslücke bei $x^*(t)$ wegen der dritten Zeile in (2.17)

$$f_0(x^*(t)) - g(\lambda^*(t)) = \sum_{i=1}^m f_i(x^*(t)) \frac{1}{t f_i(x^*(t))} = \frac{m}{t}. \tag{2.18}$$

Daher ist es auch leicht zu sehen, dass $x^*(t)$ für $t \rightarrow \infty$ gegen die Lösung x^* von (2.7) konvergiert. Wenn man also als Suchrichtung einen Newton-Schritt für die modifizierten KKT-Bedingungen wählt und t in jeder Iteration erhöht, so ist man anfangs weit genug vom Rand des zulässigen Bereichs entfernt und nähert sich immer mehr dem zentralen Pfad und somit letztendlich für t groß genug auch der Lösung

x^* . Wir betrachten also für festes $t > 0$ die Funktion $r(x, \lambda) : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^{n+m}$,

$$r(x, \lambda) = \begin{pmatrix} r_{\text{dual}} \\ r_{\text{cent}} \end{pmatrix} := \begin{pmatrix} \nabla f_0(x) + Df(x)\lambda \\ -\text{diag}(\lambda)f(x) - \frac{1}{t}\mathbf{1} \end{pmatrix},$$

und wenden darauf das Newton-Verfahren an, wobei hier mit $\mathbf{1} \in \mathbb{R}^m$ der Vektor gemeint ist, der an jeder Stelle den Eintrag 1 hat. Die Suchrichtung $(\Delta x, \Delta \lambda)$ für unseren Algorithmus erhalten wir also durch Lösen des linearen Gleichungssystems

$$\begin{pmatrix} \nabla^2 f_0(x) + \sum_{i=1}^m \lambda_i \nabla^2 f_i(x) & Df(x)^T \\ -\text{diag}(\lambda)Df(x) & -\text{diag}(f(x)) \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta \lambda \end{pmatrix} = - \begin{pmatrix} r_{\text{dual}} \\ r_{\text{cent}} \end{pmatrix}. \quad (2.19)$$

Da die Wahl des Startpunktes $x^{(0)}$ und die Liniensuche garantieren, dass x in jedem Schritt des Verfahrens strikt zulässig ist, ist $-\text{diag}(f(x))$ strikt positiv definit und wir können $\Delta \lambda$ eliminieren durch

$$\Delta \lambda = -\text{diag}(f(x))^{-1} \text{diag}(\lambda)Df(x)\Delta x + \text{diag}(f(x))^{-1}r_{\text{cent}}$$

und erhalten weiters mit

$$\tilde{H} := \nabla^2 f_0(x) + \sum_{i=1}^m \lambda_i \nabla^2 f_i(x) + \sum_{i=1}^m \frac{\lambda_i}{-f_i(x)} \nabla f_i(x) \nabla f_i(x)^T \quad (2.20)$$

das lineare Gleichungssystem

$$\tilde{H}\Delta x = -r_{\text{dual}} - Df(x)^T \text{diag}(f(x))^{-1}r_{\text{cent}}. \quad (2.21)$$

(2.19) ist also genau dann lösbar, wenn es (2.21) ist, das heißt, wenn $\tilde{H}$ invertierbar ist.

Ein Maß für die Optimalität im k -ten Iterationsschritt eines Verfahrens wäre die zugehörige Dualitätslücke $\eta^{(k)}(x^{(k)}, \lambda^{(k)}) := f_0(x^{(k)}) - g(\lambda^{(k)})$, die immer kleiner wird, je näher man dem Minimum kommt. Da aber die eben beschriebene Suchrichtung nicht notwendigerweise dual zulässige $\lambda^{(k)}$ liefert, lässt sich $\eta^{(k)}$ hier nicht berechnen. Das motiviert folgende Definition.

Definition 2.15 (Surrogat-Dualitätslücke). Sei $x \in \mathbb{R}^n$ mit $f(x) < 0$ und $\lambda \in \mathbb{R}^m, \lambda > 0$. Die *Surrogat-Dualitätslücke* $\hat{\eta}$ zu (2.7) ist definiert als

$$\hat{\eta}(x, \lambda) := -f(x)^T \lambda.$$

Man beachte, dass $\hat{\eta}(x, \lambda) = \eta(x, \lambda)$ gilt, falls λ dual zulässig ist. Nun können wir einen Algorithmus zum Auffinden einer Lösung von (2.7) formulieren.

Wären x und λ zentral, das heißt würden sie (2.17) erfüllen, dann wären die Dualitätslücke $\eta = \hat{\eta} = \frac{m}{t}$ und μ der Faktor, um den man diese verringern würde, wenn x und λ im nächsten Schritt wieder zentral wären. Demnach wäre es gut, μ so groß wie möglich zu wählen. Da das aber nicht der Fall sein muss, besteht die Gefahr, bei zu großem μ zu schnell an den Rand zu kommen, wo dann nur noch

Sei $x \in \mathbb{R}^n$ mit $f(x) < 0$, $\lambda \in \mathbb{R}^m$ mit $\lambda > 0$, $\mu > 1$, $\varepsilon > 0$ und $\varepsilon_{feas} > 0$.

while $\|r_{\text{dual}}\|_2 > \varepsilon_{feas} > 0$ **or** $\hat{\eta} > \varepsilon$ **do**

Setze $t := \frac{\mu m}{\hat{\eta}}$.

Berechne Suchrichtung $(\Delta x, \Delta \lambda)$ aus (2.19).

Linienuche: Bestimme Schrittweite $0 < s < 1$ mit $f(x + s\Delta x) < 0$ und $\lambda + s\Delta \lambda > 0$.

Update: Setze $x := x + s\Delta x$ und $\lambda := \lambda + s\Delta \lambda$.

end while

Algorithmus 2.3: Primal-duales Innere Punkte-Verfahren

winzige Schrittweiten möglich sind. Umgekehrt bleibt man bei einer zu kleinen Wahl von μ zwar länger im Inneren des zulässigen Gebiets, erlaubt dort aber nur kürzere Schritte. Laut [BV04], dem dieses Verfahren entnommen wurde, sind Werte der Größenordnung 10 gut geeignet

Nun müssen wir uns noch um den dritten Schritt in Algorithmus 2.3 kümmern. Es ist also eine möglichst große Schrittweite $s \in (0, 1)$ zu bestimmen, die sowohl $f(x + s\Delta x) < 0$ als auch $\lambda + s\Delta \lambda > 0$ garantiert. Hierfür wird eine modifizierte Version der Backtracking-Liniensuche aus Algorithmus 2.2 verwendet. Damit die zweite an s geforderte Bedingung erfüllt ist, muss $\lambda_i + s\Delta \lambda_i > 0$ für $i = 1, \dots, m$ gelten. Wegen $\lambda > 0$ trifft dies für $\Delta \lambda_i \geq 0$ ohnehin zu, und daher wählen wir als maximalen Wert für s

$$s_{\max} := 0.99 \min\{1, \min\{-\frac{\lambda_i}{\Delta \lambda_i} \mid i = 1, \dots, m, \Delta \lambda_i < 0\}\}.$$

Dieser Wert wird nun so lange mit einem $\beta \in (0, 1)$ multipliziert, bis sowohl $f(x + s\Delta x) < 0$ als auch eine Backtracking-Bedingung für die Norm von r ,

$$\|r(x + s\Delta x, \lambda + s\Delta \lambda)\|_2 \leq (1 - \alpha s)\|r(x, \lambda)\|_2,$$

erfüllt sind. In [BV04] werden $\alpha \in (0.01, 0.1)$ und $\beta \in (0.3, 0.8)$ als typische Werte genannt. Zusammengefasst sieht dieses Verfahren nun folgendermaßen aus:

Seien (x, λ) sowie die Suchrichtung $(\Delta x, \Delta \lambda)$, $\alpha \in (0, 1)$, und $\beta \in (0, 1)$ gegeben.

Setze $s = s_{\max}$

while $\|r(x + s\Delta x, \lambda + s\Delta \lambda)\|_2 > (1 - \alpha s)\|r(x, \lambda)\|_2$ **or** $f(x + s\Delta x) \geq 0$ **do**

Setze $s := \beta s$.

end while

Algorithmus 2.4: Backtracking-Liniensuche für Algorithmus 2.3

Was jetzt noch fehlt, ist die sogenannte *Phase I* also das Auffinden eines Startwerts $x^{(0)}$, der $f(x^{(0)}) < 0$ erfüllt. Dafür lassen sich aber die bereits beschriebenen Werkzeuge folgendermaßen verwenden: Man

führt in (2.7) eine neue Variable $z \in \mathbb{R}$ ein und betrachtet das konvexe Optimierungsproblem:

$$\begin{aligned} \min z \\ \text{s. t. } f_i(x) - z \leq 0, \quad i = 1, \dots, m. \end{aligned} \tag{2.22}$$

Für beliebiges $x^{(0)} \in \mathbb{R}^n$ lässt sich nun ein $z^{(0)} \in \mathbb{R}$ so wählen, dass

$$f_i(x^{(0)}) < z^{(0)}, \quad i = 1, \dots, m$$

gilt und man hat somit mit $(x^{(0)}, z^{(0)})$ einen strikt zulässigen Startwert für das Problem (2.22) gefunden, dessen Lösung sich dann mit Algorithmus 2.3 bestimmen lässt. Erhält man hierfür ein Ergebnis (x^*, z^*) mit $z^* > 0$, so sind die Nebenbedingungen in (2.7) inkonsistent, das heißt es existiert keine Lösung. Gilt jedoch $z^* < 0$, kann man Algorithmus 2.3 mit x^* als Startwert auf das ursprüngliche Problem anwenden. Dazu sei noch erwähnt, dass es nicht nötig ist, in der Phase I das tatsächliche Minimum zu suchen. Es genügt, das Verfahren abubrechen, sobald $z^{(k)} < 0$ erreicht ist.

3. Spannungsregelung durch Steuerung der Blindleistung

In diesem Kapitel wird versucht, mit Hilfe der bisherigen Resultate ein MATLAB-Programm zum Auffinden einer Lösung des Optimierungsproblems

$$\begin{aligned}
 & \min \|Q + \Delta Q\|_1 \\
 & \text{s. t. } \max(U + A_Q \Delta Q) - \min(U + A_Q \Delta Q) \leq EVB, \\
 & |Q_i + \Delta Q_i| \leq P_i \tan(\arccos(PF_{\min})), \quad i = 1, \dots, n, \\
 & |Q_i + \Delta Q_i| \leq \sqrt{S_{\max,i}^2 - P_i^2}, \quad i = 1, \dots, n
 \end{aligned} \tag{1.4}$$

zu schreiben.

3.1. Verfahren für ein allgemeines lineares Problem

Um später darauf zurückgreifen zu können, beschäftigen wir uns vorerst mit der Lösung des linearen Problems

$$\begin{aligned}
 & \min c^T x \\
 & \text{s. t. } Ax - b \leq 0.
 \end{aligned} \tag{3.1}$$

Hier ist $x \in \mathbb{R}^n$ gesucht und $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ und $A \in \mathbb{R}^{m \times n}$ gegeben, wobei wir die Forderungen

$$\begin{aligned}
 & m \geq n, \\
 & \text{rang}(A) = n
 \end{aligned} \tag{3.2}$$

an die Matrix A stellen. Der Grund für diese Voraussetzungen ist sofort ersichtlich, wenn man $\tilde{H}$ aus (2.20) berechnet. Für (3.1) ergibt sich dafür

$$\tilde{H} = A^T \text{diag}(\gamma) A, \quad \gamma_i = \frac{\lambda_i}{b_i - a_i^T x} > 0, \quad i = 1, \dots, m. \tag{3.3}$$

Hier ist mit $a_i \in \mathbb{R}^n$ der Vektor gemeint, der transponiert die i -te Zeile von A gibt. Wegen $\text{rang}(A) = n$ muss für $x \in \mathbb{R}^n$ mit $x \neq 0$

$$y := Ax \neq 0$$

gelten und wir haben somit:

$$x^T \tilde{H} x = y^T \text{diag}(\gamma) y > 0, \quad \forall x \neq 0.$$

Also ist $\tilde{H}$ positiv definit und somit das Gleichungssystem (2.19) eindeutig lösbar. Außerdem gilt für dieses lineare Problem

$$\begin{aligned} r_{\text{dual}} &= c + A^T \lambda, \\ r_{\text{cent}} &= -\text{diag}(\lambda)(Ax - b) - \frac{1}{t} \mathbf{1}, \end{aligned}$$

und für die Suchrichtung ergibt sich

$$\begin{aligned} \Delta x &= -\tilde{H}^{-1}(r_{\text{dual}} + A^T \text{diag}(\delta) r_{\text{cent}}), \\ \Delta \lambda &= -\text{diag}(\delta) \text{diag}(\lambda) A \Delta x + \text{diag}(\delta) r_{\text{cent}} \end{aligned}$$

mit $\delta \in \mathbb{R}^m$ und

$$\delta_i = \frac{1}{a_i^T x - b}, \quad i = 1, \dots, m.$$

Unsere Eingabeparameter für das Programm sind also die Vektoren $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, die Matrix $A \in \mathbb{R}^{m \times n}$, sowie ein Startpunkt $x \in \mathbb{R}^n$ und eine Zahl $\text{acc} \in \mathbb{R}$ für das Abbruchkriterium. Zudem verwenden wir noch einen booleschen Wert $\text{phaseI} \in \{0, 1\}$, der angibt, ob man nur einen zulässigen Punkt sucht ($\text{phaseI} = 1$) oder ob das tatsächliche Minimum zu finden ist ($\text{phaseI} = 0$). Im ersten Fall wird das Verfahren abgebrochen, sobald die Zielfunktion einen negativen Wert hat. Der MATLAB-Code für den Algorithmus 2.3 und das Problem (3.1) sieht dann folgendermaßen aus:

Programm 3.1: solve_lp.m

```
1 function [x, count]=solve_lp(c, A, b, x, acc, phaseI)
2 %-----
3 % The function solve_lp solves the linear optimization problem
4 %
5 %     min   c'*x
6 %     s.t.  A*x ≤ b
7 %
8 % with an interior point algorithm.
9 %-----
10
11 %-----
12 % Input Parameters
```

```

13 %
14 % c..... n-dimensional column vector describing the
15 %          objective funktion
16 % A..... mxn-matrix describing the constraints
17 % b..... m-dimensional column vector describing the
18 %          constraints
19 % x..... n-dimensional column vector, feasible starting
20 %          point for the optimization problem
21 % acc..... accuracy, real number (10^-6, 10^-8, ...)
22 % phaseI..... boolean number to decide, if the actual minimum
23 %              should be found (phaseI=0) or if the program
24 %              should be stopped, as soon as feasible point for a
25 %              reduced problem is found, i.e. as soon as the
26 %              objective function  $c'x < 0$  (phaseI=1)
27 %-----
28
29 %-----
30 % Output Parameters
31 %
32 % x..... n-dimensional column vector, either the solution of
33 %          of the problem or a feasible point for a reduced
34 %          problem
35 %-----
36
37 count=0;
38
39 % set parameter mu
40 mu=20;
41 % determine the dimensions of the problem
42 [m,n]=size(A);
43 % set starting dual variable
44 lambda=ones(m,1);
45 % compute the value of the constraints
46 f=A*x-b;
47 % compute the surrogate duality gap eta and starting value for t
48 eta=-f'*lambda;
49 t=mu*m/eta;
50 % compute rdual and rcent in the KKT-conditions
51 rdual=c+A'*lambda;
52 rcent=-diag(lambda)*f-ones(m,1)/t;
53 while eta>acc || norm(rdual)>acc
54     count=count+1;
55     % determine the matrix describing the linear system for the
56     % newton step, solve that and determine  $\Delta x$  and  $\Delta \lambda$ 
57     H=A'*diag(-lambda./f)*A;
58      $\Delta x = -H \backslash (rdual + A' * diag(1./f) * rcent)$ ;
59      $\Delta \lambda = -diag(lambda./f) * A * \Delta x + diag(1./f) * rcent$ ;
60     % perform a line search and set the new values for x and lambda
61     % and compute the new values for rdual and constraints
62     [x,lambda,rdual,f]=linesearch_lp(x,lambda, $\Delta x$ , $\Delta \lambda$ ,...
63         c,A,b,t,rdual,rcent);
64     % compute the new surrogate duality gap, t, and rcent
65     eta=-f'*lambda;
66     t=mu*m/eta;
67     rcent=-diag(lambda)*f-ones(m,1)/t;
68     % if in phase I: check if objective value  $z < 0$  and if so: stop
69     if phaseI &&  $c'x < -acc$ 
70         break;
71     end
72 end

```

Um eine Lösung für das Problem (3.1) zu finden, wählt man also anfangs einen beliebigen Startwert $x \in \mathbb{R}^n$ und überprüft ob

$$Ax - b < 0$$

erfüllt ist. Ist das nicht der Fall, so bestimmt man nun für die Phase I ein $z \in \mathbb{R}$ mit

$$z > \max(Ax - b),$$

ruft daraufhin das Programm `solve_lp` mit den Eingabeparametern

$$\hat{c} = \begin{pmatrix} \mathbf{0} \\ 1 \end{pmatrix}, \hat{A} = \begin{pmatrix} A & -\mathbf{1} \end{pmatrix}, b, \hat{x} = \begin{pmatrix} x \\ z \end{pmatrix}, \text{acc} > 0 \text{ und } \text{phaseI} = 1$$

auf und erhält als Rückgabewert einen Vektor $\hat{x} \in \mathbb{R}^{n+1}$. Natürlich muss man hierbei auch sicherstellen, dass die Matrix $\hat{A}$ noch die Rangbedingung (3.2) erfüllt. Gilt nun

$$z = \hat{x}_{n+1} < 0,$$

dann hat man einen strikt zulässigen Startpunkt $x \in \mathbb{R}^n$ gefunden, wobei x die ersten n Komponenten von $\hat{x}$ enthält, und man kann mit nochmaligem Aufrufen von `solve_lp` mit dem Input

$$c, A, b, x, \text{acc} > 0 \text{ und } \text{phaseI} = 0$$

das Minimum von (3.1) bestimmen. Andernfalls sind die Nebenbedingungen inkonsistent und es gibt keinen zulässigen Punkt.

Um das Verfahren vollständig zu machen, muss noch die Funktion `linesearch_lp` angegeben werden, die im Programm 3.1 für die Liniensuche verwendet wird. Dazu wird die Backtracking Liniensuche aus Algorithmus 2.4 verwendet. Die Eingabeparameter sind natürlich die primalen und dualen Variablen x und λ sowie die zugehörige Suchrichtung $(\Delta x, \Delta \lambda)$, aber auch die das Problem beschreibenden Größen c, b, A und der aktuelle Wert für t , damit r_{dual} und r_{cent} für $(x + s\Delta x, \lambda + s\Delta \lambda)$ berechnet werden können, was für das Abbruchkriterium in der Liniensuche benötigt wird. Außerdem werden noch die gegenwärtigen Werte für r_{dual} und r_{cent} übergeben, sodass diese nicht doppelt berechnet werden. Aus demselben Grund werden als Output neben den neuen Punkten $x^+ := x + s\Delta x$ und $\lambda^+ := \lambda + s\Delta \lambda$ auch r_{dual} und $Ax^+ - b$ zurückgegeben. Für r_{cent} ist das nicht möglich, da dazu erst der neue Wert für t bestimmt werden muss. Programm 3.2 zeigt die MATLAB-Funktion für dieses Verfahren.

Programm 3.2: `linesearch_lp.m`

```
1 function [x_new, lambda_new, rdual, f] = ...
2     linesearch_lp(x, lambda, dx, dlambda, c, A, b, t, rdual, rcent)
3
```

```

4 %-----
5 % linesearch_lp performs a backtracking linesearch for the search
6 % direction ( $\Delta x, \Delta \lambda$ ) calculated in solve_lp.
7 %-----
8
9 %-----
10 % Input Parameters:
11 %
12 % x.....current primal variable, n-dimensional vector
13 % lambda.....current dual variable, m-dimensional vector
14 %  $\Delta x$ .....search direction for x, n-dimensional vector
15 %  $\Delta \lambda$ .....search direction for lambda, m-dimensional vector
16 % c.....n-dimensional vector describing the objective function
17 % A.....(m x n)-Matrix describing the linear constraints
18 % b.....m-dimensional vector describing the linear constraints
19 % t.....real number needed to calculate rcent
20 % rdual.....current value for rdual in the KKT-conditions,
21 %           n-dimensional vector
22 % rcent.....current value for rcent in the KKT-conditions,
23 %           m-dimensional vector
24 %-----
25
26 %-----
27 % Output Parameters:
28 %
29 % x_new.....new value for x, n-dimensional vector
30 % lambda_new..new value for lambda, m-dimensional vector
31 % rdual.....value for rdual at x_new, n-dimensional vector
32 % f.....value for the constraints at x_new, m-dimensional
33 %           vector
34 %-----
35
36 % set some parameters
37 beta=0.5;
38 alpha=0.05;
39 % determine the dimensions
40 m=size(A,1);
41 % determine the maximal value for the step length
42 y=ones(m+1,1);
43 for i=1:m
44     if  $\Delta \lambda(i) < 0$ 
45          $y(i) = -\lambda(i) / \Delta \lambda(i)$ ;
46     end
47 end
48 s=0.99*min(y);
49 % compute current norm of (rdual,rcent)
50 oldnorm=norm([rdual;rcent]);
51 % set new x and lambda
52 x_new=x+s* $\Delta x$ ;
53 lambda_new=lambda+s* $\Delta \lambda$ ;
54 % compute the value of the constraints
55 f=A*x_new-b;
56 % compute the new values of rdual and rcent
57 rdual=c+A'*lambda_new;
58 rcent=-diag(lambda_new)*f-ones(m,1)/t;
59
60 while max(f)  $\geq 0$  || norm([rdual;rcent])>oldnorm*(1-alpha*s)
61     % reduce step length s
62     s=beta*s;
63     % set new x and lambda

```

```

64     x_new=x+s*Δx;
65     lambda_new=lambda+s*Δlambda;
66     % compute the value of the constraints
67     f=A*x_new-b;
68     % compute the new values of rdual and rcent
69     rdual=c+A'*lambda_new;
70     rcent=-diag(lambda_new)*f-ones(m,1)/t;
71 end

```

3.2. Transformation auf ein lineares Problem

Hier beschäftigen wir uns damit, wie wir (1.4) auf ein lineares Problem der Form (3.1) transformieren können und folglich die eben beschriebenen Programme zum Auffinden der Lösung verwenden können. Dazu erinnern wir uns vorerst noch einmal an die auftretenden Größen und deren Dimensionen:

$$\begin{aligned}
 A_Q &\in \mathbb{R}^{m \times n}, \\
 P, Q, \Delta Q, S_{\max} &\in \mathbb{R}^n, \\
 U &\in \mathbb{R}^m, \\
 PF_{\min}, EVB &\in \mathbb{R}.
 \end{aligned}$$

Zur Umformulierung führen wir die neuen Variablen

$$\begin{aligned}
 \overline{Q}_i &:= |Q_i + \Delta Q_i|, \quad i = 1, \dots, n, \\
 \overline{U} &:= \max(U + A_Q \Delta Q), \\
 \underline{U} &:= \min(U + A_Q \Delta Q),
 \end{aligned}$$

sowie den Vektor $v \in \mathbb{R}^n$, gegeben durch

$$v_i := \min(P_i \tan(\arccos(PF_{\min})), \sqrt{S_{\max,i}^2 - P_i^2}), \quad i = 1, \dots, n,$$

ein. Damit wird (1.4) nun zu

$$\begin{aligned}
 &\min \sum_{i=1}^n \overline{Q}_i \\
 &\text{s. t. } \overline{Q}_i - v_i \leq 0, \quad i = 1, \dots, n, \\
 &\quad -\overline{Q}_i - \Delta Q_i - Q_i \leq 0, \quad i = 1, \dots, n, \\
 &\quad -\overline{Q}_i + \Delta Q_i + Q_i \leq 0, \quad i = 1, \dots, n, \\
 &\quad \overline{U} - \underline{U} - EVB \leq 0, \\
 &\quad a_{Q_i}^T \Delta Q - \overline{U} + U_i \leq 0, \quad i = 1, \dots, m, \\
 &\quad -a_{Q_i}^T \Delta Q + \underline{U} - U_i \leq 0, \quad i = 1, \dots, m.
 \end{aligned} \tag{3.4}$$

Hier ist $a_{Qi} \in \mathbb{R}^n$ wie vorhin wieder der Vektor, der transponiert die i -te Zeile von A_Q ergibt. Definiert man nun

$$c := \begin{pmatrix} \mathbf{1}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix}, \quad A := \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \\ \mathbf{0}_{m,n} & A_Q & -\mathbf{1}_{m,1} & \mathbf{0}_{m,1} \\ \mathbf{0}_{m,n} & -A_Q & \mathbf{0}_{m,1} & \mathbf{1}_{m,1} \end{pmatrix}, \quad x := \begin{pmatrix} \overline{Q} \\ \Delta Q \\ \overline{U} \\ \underline{U} \end{pmatrix}, \quad b := \begin{pmatrix} v \\ Q \\ -Q \\ EVB \\ -U \\ U \end{pmatrix}, \quad (3.5)$$

dann lässt sich (3.4) in der Form von (3.1) schreiben mit $c, x \in \mathbb{R}^{2n+2}$, $A \in \mathbb{R}^{(3n+2m+1) \times (2n+2)}$ und $b \in \mathbb{R}^{2n+2}$.

Bemerkung 3.1 (Notation). In (3.5) verstehen wir unter $\mathbb{I}_n$ die $(n \times n)$ -Einheitsmatrix. Mit $\mathbf{0}_n$ bezeichnen wir die quadratische $(n \times n)$ -Nullmatrix und mit $\mathbf{0}_{m,n}$ die $(m \times n)$ -Nullmatrix. Analog sind natürlich auch $\mathbf{1}_n$ und $\mathbf{1}_{m,n}$ die Matrizen, die jeweils an jeder Stelle den Eintrag 1 haben.

Bevor wir nun die Funktion `solve_lp` auf dieses Problem anwenden können, müssen wir noch verifizieren, dass A die Forderungen (3.2) erfüllt. Dazu stelle man zuerst fest, dass für $m, n \in \mathbb{N}$ mit $m \geq 1$ und $n \geq 1$ immer

$$3n + 2m + 1 \geq 2n + 2$$

gilt. Um die Rangbedingung an A zu überprüfen, betrachte man die quadratische $((2n+2) \times (2n+2))$ -Teilmatrix

$$A^\diamond := \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \\ \mathbf{0}_{1,n} & a_{Q1}^T & -1 & 0 \end{pmatrix}$$

und wende darauf zweimal jeweils auf die letzte Spalte den Laplaceschen Entwicklungssatz¹ zur Berechnung der Determinante an. Dann erhält man

$$\det(A^\diamond) = -\det \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{I}_n \end{pmatrix} = -(-2)^n \neq 0. \quad (3.6)$$

Die Identität

$$\det \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{I}_n \end{pmatrix} = (-2)^n$$

lässt sich mittels vollständiger Induktion nach n zeigen. Für $n = 1$ kann man die Determinante leicht

¹Siehe zum Beispiel [RR02], S.57 ff.

ausrechnen und für den Induktionsschritt $n \mapsto n + 1$ erhält wiederum durch zweimaliges Anwenden des Laplaceschen Entwicklungssatzes

$$\det \begin{pmatrix} -\mathbb{I}_{n+1} & -\mathbb{I}_{n+1} \\ -\mathbb{I}_{n+1} & \mathbb{I}_{n+1} \end{pmatrix} = -2 \det \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{I}_n \end{pmatrix}.$$

$A^\diamond$ und folglich auch A haben wegen (3.6) $2n + 2$ linear unabhängige Zeilen und somit gilt $\text{rang}(A) = 2n + 2$. Dasselbe muss man jetzt auch noch für die Matrix $\hat{A} := (A, -\mathbf{1}) \in \mathbb{R}^{(3m+2n+1) \times (2n+3)}$ überprüfen, damit auch die Phase I problemlos durchführbar ist. Klarerweise gilt für $m, n \in \mathbb{N}$ mit $m \geq 1$ und $n \geq 1$ wieder

$$3n + 2m + 1 \geq 2n + 3.$$

Um den Rang von $\hat{A}$ zu bestimmen, betrachte man nun die quadratische $(2n+3) \times (2n+3)$ -Teilmatrix

$$\hat{A}^\diamond := \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 & -1 \\ \mathbf{0}_{1,n} & a_{Q1}^T & -1 & 0 & -1 \\ \mathbf{0}_{1,n} & -a_{Q1}^T & 0 & 1 & -1 \end{pmatrix}.$$

Zweimaliges Entwickeln nach der jeweils vorletzten Spalte liefert

$$\det(\hat{A}^\diamond) = -\det \begin{pmatrix} \mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ -\mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ \mathbf{0}_{1,n} & -a_{Q1}^T & -1 \end{pmatrix} - \det \begin{pmatrix} \mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ -\mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ \mathbf{0}_{1,n} & a_{Q1}^T & -1 \end{pmatrix} - \det \begin{pmatrix} \mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ -\mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & -1 \end{pmatrix}.$$

Entwickelt man dies nun nach der letzten Zeile, so heben sich die Terme mit $\pm a_{Q1,i}$ gegenseitig auf und man erhält

$$\det(\hat{A}^\diamond) = 3 \det \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{I}_n \end{pmatrix} = 3(-2)^n \neq 0.$$

Also hat auch $\hat{A}$ vollen Rang, und wir können nun mit der vorhin beschriebenen Vorgehensweise eine Lösung von (1.4) suchen. Die Transformation des Problems und die Aufrufe von `solve_lp` werden vom Programm 3.3 durchgeführt. Die Eingabewerte hierfür sind A_Q , Q , P , U , EVB , $PF_{\min}$, $S_{\max}$ und ein Genauigkeitsparameter $\text{acc} > 0$. Als Output erhält man den Vektor ΔQ sowie eine boolesche Zahl `exitflag`, die angibt, ob ΔQ die Lösung von (1.4) ist (`exitflag` = 0) oder ob in der Phase I kein zulässiger Punkt gefunden wurde (`exitflag` = 1).

Programm 3.3: optimize_Q_linear.m

```

1 function [ΔQ,exitflag,count,time]= ...
2     optimize_Q_linear(AQ,Q,P,U,EVB,PFmin,Smax,acc)
3
4 %-----
5 % optimize_Q_linear is a function to solve the optimization problem
6 %
7 %     min ||Q+ΔQ||_1
8 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
9 %           |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
10 %           |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
11 %
12 % with respect to the unknown ΔQ, where ||.|| is the 1-norm:
13 % ||x||_1:=sum(|x(i)|)
14 %-----
15
16
17 %-----
18 % Input parameters:
19 %
20 % AQ ..... mxn-matrix to compute an estimate for the change of
21 %           voltage
22 % Q ..... current reactive power, n-dimensional vector
23 % P ..... current active power, n-dimensional vector
24 % U ..... current voltage, m-dimensional vector
25 % EVB ..... bound on diffusion of voltage vector U, real number
26 % PFmin ..... real number
27 % Smax ..... nominal apparent power, n-dimensional vector
28 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
29 %-----
30
31 %-----
32 % Output parameters:
33 %
34 % ΔQ ..... solution of the problem
35 % exitflag .... exitflag:
36 %               exitflag==1 --> no solution could be found, ΔQ is
37 %               not the solution
38 %               exitflag==0 --> ΔQ is the solution of the problem
39 %-----
40
41 tic
42 exitflag=0;
43
44 %-----
45 % transform the problem into a linear optimization problem with of
46 % the form
47 %
48 %     min c'*x
49 %     s.t. A*x≤b
50
51 [m,n]=size(AQ);
52 v=min(P*tan(acos(PFmin)),sqrt(Smax.^2-P.^2));
53
54 A=[ eye(n)      , zeros(n)      , zeros(n,1) , zeros(n,1) ;
55     -eye(n)     , -eye(n)      , zeros(n,1) , zeros(n,1) ;
56     -eye(n)     , eye(n)       , zeros(n,1) , zeros(n,1) ;
57     zeros(1,n) , zeros(1,n)    , 1         , -1        ;
58     zeros(m,n) , AQ           , -ones(m,1) , zeros(m,1) ;
59     zeros(m,n) , -AQ          , zeros(m,1) , ones(m,1)  ];

```

3. Spannungsregelung durch Steuerung der Blindleistung

```
60
61 b=[ v; Q; -Q; EVB; -U; U];
62 %-----
63
64 [M,N]=size(A); % determine the dimensions of the problem
65 x=zeros(N,1); % choose a starting point
66
67 %-----
68 % Phase I:
69 % find a strictly feasible point x for the constraints  $A*x \leq b$  by
70 % solving a linear optimization problem
71 %
72 %      min z                      (phase1)
73 %      s.t.  $A*x-b \leq z$ 
74 %
75 % starting with  $z > \max(A*x-b)$  and stopping as soon as  $z < 0$  is achieved
76
77 % determine the maximal violation of the constraints
78 fmax=max(A*x-b);
79 if fmax ≥ 0
80     % transform to prepare for (phase1)
81     x(N+1)=fmax+0.1;
82     A(:,N+1)=-ones(M,1);
83     c=(1:N+1==N+1)';
84     % run linear optimization algorithm for (phase1)
85     [x,count(1)]=solve_lp(c,A,b,x,acc,1);
86     if x(N+1) ≥ 0
87         % if no point (x,z) with  $z < 0$  could be found, no feasible
88         % point for the original problem can be found, so stop and
89         % set exitflag=1
90         ΔQ=x(n+1:2*n);
91         exitflag=1;
92         time=toc;
93         count(2)=0;
94         count(3)=count(1);
95         return
96     end
97     % (phase1) was successful, retransform the problem
98     x=x(1:N);
99     A=A(:,1:N);
100 end
101
102 %-----
103 % Phase II:
104 % If we have found a feasible point we can start the actual
105 % optimization
106 c=(1:N ≤ n)';
107 [x,count(2)]=solve_lp(c,A,b,x,acc,0);
108 count(3)=count(1)+count(2);
109 ΔQ=x(n+1:2*n);
110 time=toc;
```

3.3. Wahl der Norm in der Zielfunktion

Im Lauf des Projekts wurde von Seiten des AIT der Wunsch nach der Möglichkeit geäußert, in den Zielfunktionen in (1.4) und (1.5) die 1-Norm

$$\|x\|_1 = \sum_{i=1}^n |x_i|$$

durch die euklidische Norm

$$\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

zu ersetzen. Der Unterschied, den die Wahl der Norm bewirkt, liegt in der Gewichtung der einzelnen Komponenten der Lösung beziehungsweise für (1.4) des Vektors, dessen Norm minimiert wird. Für die 1-Norm wird üblicherweise ein Eintrag relativ groß und die restlichen eher klein sein, während sich für die euklidische Norm die Werte gleichmäßiger verteilen werden. Diese Eigenschaft lässt sich im folgenden Beispiel gut sehen.

Beispiel 3.2. Seien $x, y \in \mathbb{R}$. Man betrachte das Optimierungsproblem

$$\begin{aligned} \min & \|(x, y)\| \\ \text{s. t. } & x + 2y \geq 2 \end{aligned}$$

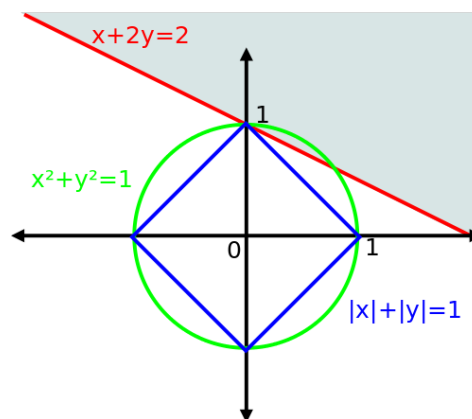


Abbildung 3.1.: Zulässiger Bereich und Norm-Einheitskugeln

Nun kann man schon aus Abbildung 3.1 ablesen, dass für die 1-Norm $(x_1, y_1) = (0, 1)$ das Problem löst. Wählt man hingegen die euklidische Norm, so ergibt sich als Lösung $(x_2, y_2) = (0.4, 0.8)$.

Da die Funktion $q : \mathbb{R} \rightarrow \mathbb{R}$, $q(x) := x^2$, für $x \geq 0$ streng monoton wachsend ist, kann man äquivalent statt der Norm auch das Quadrat der Norm minimieren und damit wird (1.4) mit der euklidischen

Norm zu dem konvexen quadratischen Optimierungsproblem

$$\begin{aligned}
 \min \|Q + \Delta Q\|_2^2 &= (Q + \Delta Q)^T (Q + \Delta Q) \\
 \text{s. t. } \max(U + A_Q \Delta Q) - \min(U + A_Q \Delta Q) &\leq EVB, \\
 |Q_i + \Delta Q_i| &\leq P_i \tan(\arccos(PF_{min})), \quad i = 1, \dots, n, \\
 |Q_i + \Delta Q_i| &\leq \sqrt{S_{\max,i}^2 - P_i^2}, \quad i = 1, \dots, n.
 \end{aligned} \tag{3.7}$$

In (1.4) versucht man also die Spannung in den richtigen Bereich zu bringen, wobei möglichst viele Kraftwerke möglichst wenig Blindleistung in das Netz einspeisen sollten und eines oder einige wenige verhältnismäßig mehr. Demgegenüber wird in (3.7) nach einer Lösung gesucht, bei der alle Kraftwerke einen gleichmäßiger verteilten Anteil an Blindleistung beitragen. Das Verfahren zum Auffinden dieser Lösung ist dem vorhin beschrieben sehr ähnlich. Die Transformation der Nebenbedingungen ist genau dieselbe wie in (3.5). Nur für die Beschreibung der Zielfunktion führen wir statt des Vektors c die Matrix

$$J := \text{diag} \begin{pmatrix} \mathbf{1}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix}$$

ein und erhalten dann das Optimierungsproblem

$$\begin{aligned}
 \min x^T J x \\
 \text{s. t. } Ax - b \leq 0.
 \end{aligned} \tag{3.8}$$

Da sich an den Nebenbedingungen nichts geändert hat, kann man für die Phase I die Algorithmen für das lineare Problem verwenden. Zur Bestimmung des Minimums muss man einige kleine Änderungen in den Programmen 3.1 und 3.2 vornehmen. Es gilt

$$\begin{aligned}
 r_{\text{dual}} &= 2Jx + A^T \lambda, \\
 \tilde{H} &= 2J + A^T \text{diag } \gamma A,
 \end{aligned}$$

mit γ wie in (3.3), also ist auch hierfür $\tilde{H}$ positiv definit. Die MATLAB-Funktionen zur Lösung von (3.7) `solve_qlp`, `linesearch_qlp` und `optimize_Q_quad` finden sich in Abschnitt A.2.

4. Spannungsregelung durch Steuerung der Wirkleistung und der Blindleistung

Sollte es für gewisse Daten nicht gelingen, mit den Methoden, die im vorigen Kapitel beschrieben wurden, eine Lösung für (1.4) zu finden, ist es also für einen bestimmten Zustand im Netz nicht möglich, die Spannungswerte an den kritischen Knoten nur durch Variation der Blindleistung, die von den Kraftwerken eingespeist wird, in die vorgegebene Bandbreite zu bringen, dann wird versucht, dieses Problem durch möglichst kleine Änderungen an der Wirkleistung zu beheben. Die zentrale Steuerungseinheit ruft also ein Programm zum Auffinden einer Lösung von

$$\begin{aligned}
 & \min \|\Delta P\|_1 \\
 & \text{s. t. } \max(U + A_Q \Delta Q + A_P \Delta P) - \min(U + A_Q \Delta Q + A_P \Delta P) \leq EVB, \\
 & |Q_i + \Delta Q_i| \leq (P_i + \Delta P_i) \tan(\arccos(PF_{\min})), \quad i = 1, \dots, n, \\
 & \sqrt{(Q_i + \Delta Q_i)^2 + (P_i + \Delta P_i)^2} \leq S_{\max,i}, \quad i = 1, \dots, n, \\
 & -P_i \leq \Delta P_i \leq 0, \quad i = 1, \dots, n
 \end{aligned} \tag{1.5}$$

auf. In diesem Kapitel wird eine MATLAB-Funktion dafür angegeben und erklärt.

4.1. Transformation auf ein konvexes Problem

Beginnen wir zunächst damit, das gegenständliche Optimierungsproblem zu transformieren. Wegen der vierten Zeile in (1.5) ist es zwar nicht mehr möglich, ein lineares Problem der Form (3.1) zu erreichen, aber durch Quadrieren dieser Zeile erhält man ein konvexes Problem mit linearen und quadratischen Nebenbedingungen. Zuerst führt man also wieder neue Variable ein:

$$\begin{aligned}
 \overline{U} &:= \max(U + A_Q \Delta Q + A_P \Delta P), \\
 \underline{U} &:= \min(U + A_Q \Delta Q + A_P \Delta P).
 \end{aligned}$$

Aufgrund der letzten Zeile von (1.5) muss

$$\begin{aligned}
 |\Delta P_i| &= -\Delta P_i, \quad i = 1, \dots, n, \\
 |P_i + \Delta P_i| &= P_i + \Delta P_i, \quad i = 1, \dots, n,
 \end{aligned}$$

gelten und somit lässt sich das Problem mit $\varphi := \arccos(PF_{\min})$ nun schreiben als:

$$\begin{aligned}
 & \min - \sum_{i=1}^n \Delta P_i \\
 & \text{s. t. } -\Delta P_i - P_i \leq 0, \quad i = 1, \dots, n, \\
 & \quad \Delta P_i \leq 0, \quad i = 1, \dots, n, \\
 & \quad -\tan(\varphi)\Delta P_i + \Delta Q_i - \tan(\varphi)P_i + Q_i \leq 0, \quad i = 1, \dots, n, \\
 & \quad -\tan(\varphi)\Delta P_i - \Delta Q_i - \tan(\varphi)P_i - Q_i \leq 0, \quad i = 1, \dots, n, \\
 & \quad \bar{U} - \underline{U} - EVB \leq 0, \\
 & \quad a_{P_i}^T \Delta P + a_{Q_i}^T \Delta Q - \bar{U} + U_i \leq 0, \quad i = 1, \dots, m, \\
 & \quad -a_{P_i}^T \Delta P - a_{Q_i}^T \Delta Q + \underline{U} - U_i \leq 0, \quad i = 1, \dots, m, \\
 & \quad \Delta P_i^2 + 2P_i \Delta P_i + \Delta Q_i^2 + 2Q_i \Delta Q_i + P_i^2 + Q_i^2 - S_{\max,i}^2 \leq 0, \quad i = 1, \dots, n,
 \end{aligned} \tag{4.1}$$

wobei wieder die Schreibweise

$$A_P = \begin{pmatrix} a_{P1}^T \\ \vdots \\ a_{Pm}^T \end{pmatrix}, \quad A_Q = \begin{pmatrix} a_{Q1}^T \\ \vdots \\ a_{Qm}^T \end{pmatrix}$$

mit $a_{P_i}, a_{Q_i} \in \mathbb{R}^n$ für $i = 1, \dots, m$ verwendet wurde. Nun definieren wir ähnlich wie im vorigen Kapitel die Vektoren $c, x \in \mathbb{R}^{2n+2}$ und $b \in \mathbb{R}^{4n+2m+1}$ durch

$$c := \begin{pmatrix} -\mathbf{1}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix}, \quad x := \begin{pmatrix} \Delta P \\ \Delta Q \\ \bar{U} \\ \underline{U} \end{pmatrix}, \quad b := \begin{pmatrix} P \\ \mathbf{0}_{n,1} \\ \tan(\varphi)P - Q \\ \tan(\varphi)P + Q \\ EVB \\ -U \\ U \end{pmatrix}, \tag{4.2}$$

und die $(4n + 2m + 1) \times (2n + 2)$ -Matrix A als

$$A := \begin{pmatrix} -\mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \\ A_P & A_Q & -\mathbf{1}_{m,1} & \mathbf{0}_{m,1} \\ -A_P & -A_Q & \mathbf{0}_{m,1} & \mathbf{1}_{m,1} \end{pmatrix}. \tag{4.3}$$

Desweiteren führen wir den Vektor $d \in \mathbb{R}^n$ sowie die Matrizen $K, L \in \mathbb{R}^{(2n+2) \times n}$ ein, mit

$$\begin{aligned} d_i &:= S_{\max,i}^2 - P_i^2 - Q_i^2, \quad i = 1, \dots, n, \\ K &:= \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \end{pmatrix}, \\ L &:= \begin{pmatrix} \mathbf{0}_n & \mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \end{pmatrix}, \end{aligned} \tag{4.4}$$

und können dann (4.1) schreiben als

$$\begin{aligned} \min \quad & c^T x \\ \text{s. t.} \quad & Ax - b \leq 0, \\ & \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d \leq 0. \end{aligned} \tag{4.5}$$

Zuerst verifiziere man wieder

$$4n + 2m + 1 \geq 2n + 3 > 2n + 2, \quad \forall m, n \in \mathbb{N}$$

mit $m \geq 1$ und $n \geq 1$, und betrachte daraufhin abermals die quadratischen Teilmatrizen $A^\diamond \in \mathbb{R}^{(2n+2) \times (2n+2)}$ und $\hat{A}^\diamond \in \mathbb{R}^{(2n+3) \times (2n+3)}$ von A beziehungsweise $\hat{A}$, gegeben durch

$$A^\diamond = \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ a_{P1}^T & a_{Q1}^T & -1 & 0 \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \end{pmatrix}$$

und

$$\hat{A}^\diamond = \begin{pmatrix} \mathbb{I}_{1,n} & \mathbf{0}_{1,n} & 0 & 0 & -1 \\ \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ a_{P1}^T & a_{Q1}^T & -1 & 0 & -1 \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 & -1 \end{pmatrix},$$

wobei mit $\mathbb{I}_{1,n}$ der erste n -dimensionale Zeilen-Einheitsvektor gemeint ist. Nun sieht man sofort, dass $A^\diamond$ eine untere Diagonalmatrix ist, deren Eintragungen auf der Hauptdiagonale alle ungleich 0 sind. Daher hat $A^\diamond$ und somit auch A vollen Rang. Um dieses Resultat auch für $\hat{A}^\diamond$ zu zeigen, berechnen wir wieder die Determinante. Für diese gilt nach zweimaligem Entwickeln nach der vorletzten Spalte

$$\det(\hat{A}^\diamond) = \det \begin{pmatrix} \mathbb{I}_{1,n} & \mathbf{0}_{1,n} & -1 \\ \mathbb{I}_n & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & -\mathbf{1}_{n,1} \end{pmatrix}.$$

Führt man nun fort, indem man den Laplaceschen Entwicklungssatz zweimal auf die erste Zeile anwendet, so erhält man

$$\begin{aligned} \det(\hat{A}^\diamond) &= -\det \begin{pmatrix} \mathbf{0}_{1,n-1} & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ \mathbb{I}_{n-1} & & \\ \mathbf{0}_{1,n-1} & \mathbb{I}_n & -\mathbf{1}_{n,1} \\ -\tan(\varphi)\mathbb{I}_{n-1} & & \end{pmatrix} = \underbrace{-\det \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n \end{pmatrix}}_{=1} \\ &= -\det \underbrace{\begin{pmatrix} \mathbb{I}_{n-1} & \mathbf{0}_{n-1,n} \\ \mathbf{0}_{1,n-1} & \mathbb{I}_n \\ -\tan(\varphi)\mathbb{I}_{n-1} & \end{pmatrix}}_{=1} - 1 = -2 \neq 0, \end{aligned}$$

und kann somit folgern, dass auch $\hat{A}$ vollen Rang hat. Analog zum Programm 3.3 ergibt sich zum Auffinden einer Lösung von (1.5) nun folgende Methode: Zuerst transformiert man das Problem wie eben beschrieben, sodass es die Gestalt von (4.5) erhält und wählt dafür einen beliebigen Startwert $x^{(0)}$. Dann überprüft man, ob $x^{(0)}$ strikt zulässig ist, und startet gegebenenfalls in der Phase I ein Verfahren, das einen Punkt x sucht, der die Nebenbedingungen erfüllt. Wird kein solcher Punkt gefunden, dann sind die Nebenbedingungen inkonsistent und man setzt `exitflag = 1` und bricht das Programm ab. Andernfalls ruft man mit Startwert x eine Funktion zur Lösung von (4.5) auf. Das MATLAB-Programm, welches diese Vorgehensweise durchführt, ist mit Programm `optimize_P_Q_linear.m` gegeben. Die Eingabeparameter hierfür sind die Vektoren $P, Q, S_{\max} \in \mathbb{R}^n, U \in \mathbb{R}^m$, die $(m \times n)$ -Matrizen A_P, A_Q , und die Zahlen $EVB, PF_{\min}, acc \in \mathbb{R}$, wobei `acc` die Genauigkeit bestimmt, mit der das Minimum berechnet werden soll. Als Output erhält man die Vektoren $\Delta P, \Delta Q \in \mathbb{R}^n$, die angeben, wie die Kraftwerke ihre eingespeiste Wirk- und Blindleistung verändern müssen, damit die Spannungswerte an den kritischen Knoten wieder in der vorgegebenen Bandbreite liegen, und einen booleschen Wert `exitflag`, der mitteilt, ob ΔP und ΔQ (1.5) wirklich lösen (`exitflag = 0`) oder ob keine Werte gefunden wurden, die die Nebenbedingungen erfüllen (`exitflag = 1`).

Programm 4.1: `optimize_P_Q_linear.m`

```
1 function [ΔP, ΔQ, exitflag, count, time] = ...
2     optimize_P_Q_linear(AP, AQ, P, Q, U, EVB, PFmin, Smax, acc)
3
4 %-----
5 % optimize_P_Q_linear is a function to solve the optimization
6 % problem
7 %
8 %     min ||Q+ΔQ||_1
9 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %         |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %         |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %
13 % with respect to the unknown ΔQ, where ||.||_1 is the 1-norm:
14 % ||x||_1:=sum(|x(i)|)
15 %-----
```

```

16
17 %-----
18 % Input parameters:
19 %
20 % AP ..... mxn-matrix to compute an estimate for the change of
21 %           voltage
22 % AQ ..... mxn-matrix to compute an estimate for the change of
23 %           voltage
24 % P ..... current active power, n-dimensional vector
25 % Q ..... current reactive power, n-dimensional vector
26 % U ..... current voltage, m-dimensional vector
27 % EVB ..... bound on diffusion of voltage vector U, real number
28 % PFmin ..... real number
29 % Smax ..... nominal apparent power, n-dimensional vector
30 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
31 %-----
32
33 %-----
34 % Output parameters:
35 %
36 % ΔQ ..... solution of the problem
37 % exitflag .... exitflag:
38 %           exitflag==1 --> no solution could be found, ΔQ is
39 %           not the solution
40 %           exitflag==0 --> ΔQ is the solution of the problem
41 %-----
42
43 tic
44 exitflag=0;
45
46 %-----
47 % transform the problem into a optimization problem of the form
48 %   min c'*x
49 %   s.t. A*x ≤ b
50 %       diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
51
52 [m,n]=size(AQ);
53 tp=tan(acos(PFmin));
54
55 A=[-eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
56     eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
57     -tp*eye(n), eye(n) , zeros(n,1), zeros(n,1);
58     -tp*eye(n), -eye(n) , zeros(n,1), zeros(n,1);
59     zeros(1,n), zeros(1,n), 1 , -1 ;
60     AP , AQ , -ones(m,1), zeros(m,1);
61     -AP , -AQ , zeros(m,1), ones(m,1) ];
62
63 b=[P; zeros(n,1); tp*P-Q; tp*P+Q; EVB; -U; U];
64
65 K=[eye(n) , zeros(n) , zeros(n,1), zeros(n,1)];
66 L=[zeros(n) , eye(n) , zeros(n,1), zeros(n,1)];
67 d=Smax.^2-P.^2-Q.^2;
68
69 %-----
70
71 [M,N]=size(A); % determine the dimensions of the problem
72 x=zeros(N,1); % choose a starting point
73
74 %-----
75 % Phase I:

```

```

76 % find a strictly feasible point x for the constraints  $A*x \leq b$  by
77 % solving a linear optimization problem
78 %
79 %      min  z                                (phase1)
80 %      s.t.  $A*x - b \leq z$ 
81 %            $\text{diag}(K*x + 2*P)*K*x + \text{diag}(L*x + 2*Q)*L*x - d \leq z$ 
82 %
83 % starting with  $z > \max(A*x - b, \text{diag}(K*x + 2*P)*K*x + \text{diag}(L*x + 2*Q)*L*x - d)$ 
84 % and stopping as soon as  $z < 0$  is achieved
85
86 % determine the maximal violation of the constraints
87 fmax = max([-b; -d]);
88 if fmax ≥ 0
89     % transform to prepare for (phase1)
90     x(N+1) = fmax + 0.1;
91     A(:, N+1) = -ones(M, 1);
92     K(:, N+1) = zeros(n, 1);
93     L(:, N+1) = zeros(n, 1);
94     c = (1:N+1 == N+1)';
95     % run linear optimization algorithm for (phase1)
96     [x, count(1)] = solve_lqp(c, A, b, K, L, P, Q, d, x, acc, 1);
97     if x(N+1) ≥ 0
98         % if no point (x,z) with  $z < 0$  could be found, no feasible
99         % point for the original problem can be found, so stop and
100         % set exitflag=1
101         ΔP = x(1:n);
102         ΔQ = x(n+1:2*n);
103         exitflag = 1;
104         time = toc;
105         return
106     end
107     % (phase1) was successful, retransform the problem
108     x = x(1:N);
109     A = A(:, 1:N);
110     K = K(:, 1:N);
111     L = L(:, 1:N);
112 end
113
114 % If we have found a feasible point we can start the actual
115 % optimization
116 c = -(1:N ≤ n)';
117 [x, count(2)] = solve_lqp(c, A, b, K, L, P, Q, d, x, acc, 0);
118 count(3) = count(1) + count(2);
119 ΔP = x(1:n);
120 ΔQ = x(n+1:2*n);
121 time = toc;

```

Die Funktion `solve_lqp`, die in Programm 4.1 aufgerufen wird, wird im nächsten Abschnitt angegeben und erklärt.

4.2. Lösung des konvexen Problems

Hier wollen wir nun ein MATLAB-Programm zur Lösung des vorhin eingeführten konvexen Optimierungsproblems

$$\begin{aligned} \min c^T x \\ \text{s. t. } Ax - b \leq 0, \\ \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d \leq 0. \end{aligned} \quad (4.5)$$

präsentieren. Es implementiert wieder das primal-duale Innere-Punkte-Verfahren aus Algorithmus 2.3. Als Eingabewerte haben wir die Matrizen $A \in \mathbb{R}^{(m \times n)}$, $K, L \in \mathbb{R}^{(p \times n)}$, die Vektoren $c, x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $P, Q, d \in \mathbb{R}^p$, einen Genauigkeitsparameter $\text{acc} \in \mathbb{R}$ und eine boolesche Zahl phaseI , wobei x strikt zulässig sei, $m \geq n > p$ gelte und A vollen Rang habe, also:

$$\begin{aligned} Ax - b &< 0, \\ \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d &< 0, \\ \text{rang}(A) &= n. \end{aligned}$$

Als Rückgabewert erhält man einen Vektor $x \in \mathbb{R}^n$, der entweder für $\text{phaseI} = 0$ das Problem (4.5) löst, oder dessen ersten $n - 1$ Komponenten für $\text{phaseI} = 1$ einen Vektor bilden, der die um die letzte Dimension reduzierten Nebenbedingungen strikt erfüllt, falls dies möglich ist. Da dasselbe Innere-Punkte-Verfahren zugrundeliegt, wird die gesuchte Funktion dem Programm 3.1 sehr ähnlich sein. Nur die Unterscheidung zwischen $\text{phaseI} = 0$ und $\text{phaseI} = 1$ wird etwas komplizierter. Aufgrund der nichtlinearen Nebenbedingungen in (4.5) reicht es nicht mehr aus, für die Phase I einfach die erweiterte Matrix $\hat{A} := \begin{pmatrix} A & -\mathbf{1}_{m,1} \end{pmatrix}$ zu verwenden. Man muss auch die letzten p Nebenbedingungen beachten, kann aber an die Matrizen K und L nicht einfach den Vektor $-\mathbf{1}_{p,1}$ hängen. Wir erinnern uns also an das Optimierungsproblem für die Phase I. Für (4.5) hat es folgende Gestalt:

$$\begin{aligned} \min z \\ \text{s. t. } Ax - b - z \leq 0, \\ \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d - z \leq 0. \end{aligned} \quad (4.6)$$

Setzt man nun

$$\hat{c} := \begin{pmatrix} \mathbf{0}_{n,1} \\ 1 \end{pmatrix}, \quad \hat{x} := \begin{pmatrix} x \\ z \end{pmatrix}, \quad \hat{K} = \begin{pmatrix} K & \mathbf{0}_{p,1} \end{pmatrix} \text{ und } \hat{L} = \begin{pmatrix} L & \mathbf{0}_{p,1} \end{pmatrix},$$

dann wird (4.6) zu

$$\begin{aligned} \min \hat{c}^T \hat{x} \\ \text{s. t. } \hat{A}\hat{x} - b \leq 0, \\ \text{diag}(\hat{K}\hat{x})\hat{K}\hat{x} + \text{diag}(\hat{L}\hat{x})\hat{L}\hat{x} - \hat{x}_{n+1}\mathbf{1}_{p,1} - d \leq 0. \end{aligned} \quad (4.7)$$

Bezeichnet man mit $f : \mathbb{R}^n \rightarrow \mathbb{R}^{m+p}$ die Funktion, die alle Nebenbedingungen beinhaltet, so ergibt sich in dem gesuchten Programm für die Phase I:

$$\begin{aligned} f(x) &= \begin{pmatrix} Ax - b \\ \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - x_n\mathbf{1}_{p,1} - d \end{pmatrix}, \\ Df(x) &= \begin{pmatrix} A \\ 2\text{diag}(Kx + P)K + 2\text{diag}(Lx + Q)L - \begin{pmatrix} \mathbf{0}_{p,n-1} & \mathbf{1}_{p,1} \end{pmatrix} \end{pmatrix}. \end{aligned}$$

Für $\text{phaseI} = 0$ hingegen gilt:

$$\begin{aligned} f(x) &= \begin{pmatrix} Ax - b \\ \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d \end{pmatrix}, \\ Df(x) &= \begin{pmatrix} A \\ 2\text{diag}(Kx + P)K + 2\text{diag}(Lx + Q)L \end{pmatrix}. \end{aligned}$$

Das Programm muss also jedesmal, wenn $f(x)$ oder $Df(x)$ berechnet werden soll, den Wert von phaseI überprüfen. Alle anderen Größen lassen sich damit unabhängig von phaseI berechnen. Um das zu tun, definieren wir wieder $\gamma \in \mathbb{R}^{m+p}$ durch

$$\gamma_i = -\frac{\lambda_i}{f_i(x)} > 0, \quad i = 1, \dots, m+p.$$

Außerdem setzen wir

$$\lambda_A := \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_m \end{pmatrix}, \quad \lambda_{KL} := \begin{pmatrix} \lambda_{m+1} \\ \vdots \\ \lambda_{m+p} \end{pmatrix}, \quad \gamma_A := \begin{pmatrix} \gamma_1 \\ \vdots \\ \gamma_m \end{pmatrix}, \quad \gamma_{KL} := \begin{pmatrix} \gamma_{m+1} \\ \vdots \\ \gamma_{m+p} \end{pmatrix}$$

und

$$Df(x)_{KL} := \begin{pmatrix} Df(x)_{m+1,:} \\ \vdots \\ Df(x)_{m+p,:} \end{pmatrix}$$

und erhalten damit

$$r_{\text{dual}} = c + Df(x)^T \lambda,$$

$$r_{\text{cent}} = -\text{diag}(\lambda)f(x) - \frac{1}{t}\mathbf{1}_{m+p,1},$$

$$\begin{aligned}\tilde{H} &= 2K^T \text{diag}(\lambda_{KL})K + 2L^T \text{diag}(\lambda_{KL})L + Df(x)^T \text{diag}(\gamma)Df(x) \\ &= 2K^T \text{diag}(\lambda_{KL})K + 2L^T \text{diag}(\lambda_{KL})L + Df(x)_{KL}^T \text{diag}(\gamma_{KL})Df(x)_{KL} + A^T \text{diag}(\gamma_A)A.\end{aligned}$$

Wegen $\lambda_i > 0$ und $\gamma_i > 0$ für $i = 1, \dots, m+p$ und weil A vollen Rang hat, ist $\tilde{H}$ strikt positiv definit und wir können die MATLAB-Funktion `solve_lqp` zur Lösung von (4.5) angeben:

Programm 4.2: `solve_lqp.m`

```
1 function [x, count]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,phaseI)
2
3 %-----
4 % The function solve_lp solves the linear optimization problem
5 %
6 %     min   c'*x
7 %     s.t.  A*x ≤ b
8 %           diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
9 %
10 % with an interior point algorithm.
11 %-----
12
13 %-----
14 % Input Parameters
15 %
16 % c..... n-dimensional column vector describing the
17 %          objective funktion
18 % A..... mxn-matrix describing the constraints
19 % b..... m-dimensional column vector describing the
20 %          constraints
21 % K..... pxn-matrix describing the constraints
22 % L..... pxn-matrix describing the constraints
23 % P..... p-dimensional column vector describing the
24 %          constraints, current active power in optimize_P_Q*
25 % Q..... p-dimensional column vector describing the
26 %          constraints, current reactive power in optimize_P_Q*
27 % d..... p-dimensional column vector describing the
28 %          constraints
29 % x..... n-dimensional column vector, feasible starting
30 %          point for the optimization problem
31 % acc..... accuracy, real number (10^-6, 10^-8, ...)
32 % phaseI..... boolean number to decide, if the actual minimum
33 %              should be found (phaseI=0) or if the program
34 %              should be stopped, as soon as feasible point for a
35 %              reduced problem is found, i.e. as soon as the
36 %              objective function c'*x<0
37 %-----
38
39 %-----
40 % Output Parameters
41 %
```

```

42 % x..... n-dimensional column vector, either the solution of
43 %           of the problem or a feasible point for a reduced
44 %           problem
45 %-----
46
47 count=0;
48 % set parameter mu:
49 mu=20;
50 % determine the dimensions of the problem:
51 [m,n]=size(A);
52 p=size(P,1);
53 % set starting dual variable:
54 lambda=ones(m+p,1);
55 % compute the value of the constraints:
56 if phaseI
57     f=[A*x-b;diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d-x(n)*ones(p,1)];
58     df=[A;2*diag(K*x+P)*K+2*diag(L*x+Q)*L-[zeros(p,n-1),ones(p,1)]];
59 else
60     f=[A*x-b;diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d];
61     df=[A;2*diag(K*x+P)*K+2*diag(L*x+Q)*L];
62 end
63 % compute surrogate duality gap eta and starting value for t:
64 eta=-f'*lambda;
65 t=mu*(m+p)/eta;
66 % compute rdual and rcent in the KKT-conditions:
67 rdual=c+df'*lambda;
68 rcent=-diag(lambda)*f-ones(m+p,1)/t;
69 while eta>acc || norm(rdual)>acc
70     count=count+1;
71     % determine the matrix describing the linear system for the
72     % newton step, solve that and determine Δx and Δtλ
73     H=2*K'*diag(lambda(m+1:m+p))*K+2*L'*diag(lambda(m+1:m+p))*L+...
74     df'*diag(-lambda./f)*df;
75     Δx=-H\ (rdual+df'*diag(1./f)*rcent);
76     Δlambda=-diag(lambda./f)*df*Δx+diag(1./f)*rcent;
77     % perform a line search and set the new values for x and lambda
78     % and compute the new values for rdual and constraints
79     [x,lambda,rdual,f,df]=linesearch_lqp(x,lambda,...
80     Δx,Δlambda,c,A,b,K,L,P,Q,d,t,rdual,rcent,phaseI);
81     % if in phase I: check if objective value z<0 and if so: stop
82     if phaseI && c'*x<-acc
83         break;
84     end
85     % compute the new surrogate duality gap eta, t, and rcent
86     eta=-f'*lambda;
87     t=mu*(m+p)/eta;
88     rcent=-diag(lambda)*f-ones(m+p,1)/t;
89 end

```

Zuletzt muss jetzt noch die Funktion zur Liniensuche `linesearch_lqp` die im Programm 4.2 aufgerufen wird, angegeben werden. Auch sie ist der Liniensuche `linesearch_lp` aus Abschnitt 3.1 sehr ähnlich. Der Unterschied besteht lediglich in der Berechnung von f und Df . Daher muss hier auch `phaseI` als zusätzlicher Eingabeparameter übergeben werden. Der gesamte Input besteht somit aus den aktuellen Werten für x und λ , der Suchrichtung $(\Delta x, \Delta \lambda)$ sowie allen Größen, die zur Bestimmung von r_{dual} und r_{cent} notwendig sind: $c, A, b, K, L, P, Q, d, t$ und `phaseI`. Außerdem werden wieder die aktuellen Werte für r_{dual} und r_{cent} übergeben, um Doppelberechnungen zu vermei-

den. Aus genau demselben Grund finden sich in den Rückgabewerten neben den x^+ und λ^+ auch die entsprechenden, schon ermittelten r_{dual} , $f(x^+)$ und $Df(x^+)$. Das MATLAB-Programm hierzu sieht folgendermaßen aus:

Programm 4.3: linesearch_lqp.m

```

1 function [x_new, lambda_new, rdual, f, df]=linesearch_lqp(x, lambda, ...
2     Δx, Δlambda, c, A, b, K, L, P, Q, d, t, rdual, rcent, phaseI)
3
4 %-----
5 % linesearch_lqp performs a backtracking linesearch for the search
6 % direction (Δx, Δlambda) calculated in solve_lqp.
7 %-----
8
9 %-----
10 % Input Parameters:
11 %
12 % x.....current primal variable, n-dimensional vector
13 % lambda.....current dual variable, (m+p)-dimensional vector
14 % Δx.....search direction for x, n-dimensional vector
15 % Δlambda.....search direction for lambda, (m+p)-dimensional vector
16 % c.....n-dimensional vector describing the objective function
17 % A.....(m×n)-Matrix describing the linear constraints
18 % b.....m-dimensional vector describing the linear constraints
19 % K.....(p×n)-Matrix describing the quadratic constraints
20 % L.....(p×n)-Matrix describing the quadratic constraints
21 % P.....p-dimensional column vector describing the
22 %      constraints, current active power in optimize_P_Q*
23 % Q.....p-dimensional column vector describing the
24 %      constraints, current reactive power in optimize_P_Q*
25 % d.....p-dimensional vector describing the quadratic
26 %      constraints
27 % t.....real number needed to calculate rcent
28 % rdual.....current value for rdual in the KKT-conditions,
29 %      n-dimensional vector
30 % rcent.....current value for rcent in the KKT-conditions,
31 %      (m+p)-dimensional vector
32 % phaseI.....boolean number describing, if the algorithm is in phase
33 %      I
34 %-----
35
36 %-----
37 % Output Parameters:
38 %
39 % x_new.....new value for x, n-dimensional vector
40 % lambda_new..new value for lambda, (m+p)-dimensional vector
41 % rdual.....value for rdual at x_new, n-dimensional vector
42 % f.....value for the constraints at x_new, (m+p)-dimensional
43 %      vector
44 % df.....value for the derivative of the constraints at x_new,
45 %      (m+p)-dimensional vector
46 %-----
47
48 % set some parameters
49 beta=0.5;
50 alpha=0.05;
51 % determine the dimensions
52 [m,n]=size(A);

```

```

53 p=size(K,1);
54 % determine the maximal value for the step length
55 y=ones(m+p+1,1);
56 for i=1:m+p
57     if Δlambda(i)<0
58         y(i)=-lambda(i)/Δlambda(i);
59     end
60 end
61 s=0.99*min(y);
62 % compute current norm of (rdual,rcent)
63 oldnorm=norm([rdual;rcent]);
64 % set new x and lambda
65 x_new=x+s*Δx;
66 lambda_new=lambda+s*Δlambda;
67 % compute the value of the constraints
68 if phaseI
69     f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
70         diag(L*x_new+2*Q)*L*x_new-d-x_new(n)*ones(p,1)];
71     df=[A;2*diag(K*x_new+P)*K+...
72         2*diag(L*x_new+Q)*L-[zeros(p,n-1),ones(p,1)]];
73 else
74     f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
75         diag(L*x_new+2*Q)*L*x_new-d];
76     df=[A;2*diag(K*x_new+P)*K+2*diag(L*x_new+Q)*L];
77 end
78 % compute the new values of rdual and rcent
79 rdual=c+df'*lambda_new;
80 rcent=-diag(lambda_new)*f-ones(m+p,1)/t;
81
82 while max(f)≥0 || norm([rdual;rcent])>oldnorm*(1-alpha*s)
83     % reduce step length s
84     s=beta*s;
85     % set new x and lambda
86     x_new=x+s*Δx;
87     lambda_new=lambda+s*Δlambda;
88     % compute the value of the constraints
89     if phaseI
90         f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
91             diag(L*x_new+2*Q)*L*x_new-d-x_new(n)*ones(p,1)];
92         df=[A;2*diag(K*x_new+P)*K+...
93             2*diag(L*x_new+Q)*L-[zeros(p,n-1),ones(p,1)]];
94     else
95         f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
96             diag(L*x_new+2*Q)*L*x_new-d];
97         df=[A;2*diag(K*x_new+P)*K+2*diag(L*x_new+Q)*L];
98     end
99     % compute the new values of rdual and rcent
100     rdual=c+df'*lambda_new;
101     rcent=-diag(lambda_new)*f-ones(m+p,1)/t;
102 end

```

4.3. Euklidische Norm in der Zielfunktion

Wie schon in Abschnitt 3.3 erwähnt, möchte man auch für das Problem (1.5) die Möglichkeit haben, in der Zielfunktion die 1-Norm durch die euklidische Norm zu ersetzen. Damit wird (1.5) zu

$$\begin{aligned}
 & \min \|\Delta P\|_2^2 \\
 & \text{s. t. } \max(U + A_Q \Delta Q + A_P \Delta P) - \min(U + A_Q \Delta Q + A_P \Delta P) \leq EVB, \\
 & |Q_i + \Delta Q_i| \leq (P_i + \Delta P_i) \tan(\arccos(PF_{min})), \quad i = 1, \dots, n, \\
 & \sqrt{(Q_i + \Delta Q_i)^2 + (P_i + \Delta P_i)^2} \leq S_{\max,i}, \quad i = 1, \dots, n, \\
 & -P_i < \Delta P_i < 0, \quad i = 1, \dots, n.
 \end{aligned} \tag{4.8}$$

Der Unterschied besteht wieder darin, dass man für (1.5) an einer Lösung interessiert ist, bei der möglichst wenig Kraftwerke eine Änderung an der Einspeisung der Wirkleistung vornehmen müssen, während man für (4.8) ein Ergebnis sucht, bei dem jedes einzelne Kraftwerk die eingespeiste Wirkleistung möglichst wenig reduzieren muss. Da sich in den Nebenbedingungen nichts ändert kann man wieder die Transformation aus (4.1) - (4.4) durchführen und mit $J \in \mathbb{R}^{(2n+2) \times (2n+2)}$,

$$J := \text{diag} \begin{pmatrix} \mathbf{1}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix},$$

erhält man das konvexe Optimierungsproblem mit quadratischer Zielfunktion und linearen und quadratischen Nebenbedingungen

$$\begin{aligned}
 & \min x^T J x \\
 & \text{s. t. } Ax - b \leq 0, \\
 & \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d \leq 0.
 \end{aligned} \tag{4.9}$$

Wie auch schon für (3.8) lassen sich hier für die Phase I die Programme 4.2 und 4.3 verwenden, die wir zum Auffinden einer Lösung von (1.5) verfasst haben. Um dann mit einem zulässigen Startwert das tatsächliche Minimum zu suchen, ändern sich in diesen Programmen folgende Größen:

$$\begin{aligned}
 r_{\text{dual}} &= 2Jx + Df(x)^T \lambda, \\
 \tilde{H} &= 2J + 2K^T \text{diag}(\lambda_{KL})K + 2L^T \text{diag}(\lambda_{KL})L + Df(x)^T \text{diag}(\gamma)Df(x) \\
 &= 2J + 2K^T \text{diag}(\lambda_{KL})K + 2L^T \text{diag}(\lambda_{KL})L \\
 &\quad + Df(x)_{KL}^T \text{diag}(\gamma_{KL})Df(x)_{KL} + A^T \text{diag}(\gamma_A)A,
 \end{aligned}$$

mit $\gamma, \lambda_{KL}, \gamma_{KL}, \gamma_A$ und $Df(x)_{KL}$ wie im vorigen Abschnitt. Daher ist $\tilde{H}$ auch für dieses Problem strikt positiv definit, und wir können die MATLAB-Funktion `optimize_P_Q_quad, solve_qqp`

und `linsearch_qqp` zum Auffinden einer Lösung formulieren. Die Programme hierzu sind im Abschnitt A.4 zu finden.

5. Berücksichtigung eines Fehlers bei der Linearisierung der Spannungsberechnung

In Abschnitt 1.3 wurde bereits erwähnt, dass durch die Linearisierung des Modells zur Spannungsberechnung an den kritischen Knoten

$$\Delta U = A_Q \Delta Q + A_P \Delta P, \quad (1.1)$$

mit $\Delta U \in \mathbb{R}^m, \Delta Q, \Delta P \in \mathbb{R}^n$ und $A_Q, A_P \in \mathbb{R}^{m \times n}$, dieses nicht mehr ganz exakt ist. Wenn man aber gewisse Informationen über die Größe des dabei auftretenden Fehlers hat, so kann man das in den Programmen zur Lösung von (1.4) und (1.5) mitberücksichtigen. Nehmen wir also an, dass $\tilde{A}_Q \in \mathbb{R}^{m \times n}$ und $\tilde{A}_P \in \mathbb{R}^{m \times n}$ jene Matrizen seien, für die (1.1) exakt wäre. Seien weiters $a_{Qij}, a_{Pij}, \tilde{a}_{Qij}, \tilde{a}_{Pij} \in \mathbb{R}$ die Eintragungen von $A_Q, A_P, \tilde{A}_Q$ und $\tilde{A}_P$. Wenn man nun weiß, dass

$$\begin{aligned} |\tilde{a}_{Qij} - a_{Qij}| &\leq e_{Qij} \quad i = 1, \dots, m, j = 1, \dots, n, \\ |\tilde{a}_{Pij} - a_{Pij}| &\leq e_{Pij} \quad i = 1, \dots, m, j = 1, \dots, n \end{aligned} \quad (5.1)$$

gilt für $e_{Qij} \geq 0$ und $e_{Pij} \geq 0$, dann kann man folgern:

$$\begin{aligned} \tilde{A}_Q \Delta Q + \tilde{A}_P \Delta P &= (\tilde{A}_Q - A_Q + A_Q) \Delta Q + (\tilde{A}_P - A_P + A_P) \Delta P \\ &= (\tilde{A}_Q - A_Q) \Delta Q + A_Q \Delta Q + (\tilde{A}_P - A_P) \Delta P + A_P \Delta P \\ &\leq E_Q \overline{\Delta Q} + A_Q \Delta Q + E_P \overline{\Delta P} + A_P \Delta P, \end{aligned} \quad (5.2)$$

wobei hier $E_Q, E_P \in \mathbb{R}^{m \times n}$ die Matrizen bezeichnen, deren Komponenten e_{Qij} beziehungsweise e_{Pij} sind, und $\overline{\Delta Q}, \overline{\Delta P} \in \mathbb{R}^n$ definiert sind durch

$$\begin{aligned} \overline{\Delta Q}_i &= |\Delta Q_i| \quad i = 1, \dots, n \\ \overline{\Delta P}_i &= |\Delta P_i| \quad i = 1, \dots, n. \end{aligned} \quad (5.3)$$

Analog erhält man auch

$$-\tilde{A}_Q \Delta Q - \tilde{A}_P \Delta P \leq E_Q \overline{\Delta Q} - A_Q \Delta Q + E_P \overline{\Delta P} - A_P \Delta P, \quad (5.4)$$

5.1. Steuerung der Blindleistung

Zum Auffinden einer Lösung von (1.4) haben wir die Nebenbedingung, die A_Q enthält,

$$\max(U + A_Q \Delta Q) - \min(U + A_Q \Delta Q) \leq EVB, \quad (5.5)$$

transformiert auf die $2m + 1$ dazu äquivalenten Bedingungen:

$$\begin{aligned} U + A_Q \Delta Q &\leq \bar{U} \mathbf{1}_{m,1}, \\ -U - A_Q \Delta Q &\leq -\underline{U} \mathbf{1}_{m,1}, \\ \bar{U} - \underline{U} &\leq EVB. \end{aligned} \quad (5.6)$$

Will man also, dass (5.5) für alle Matrizen $\tilde{A}_Q$ gilt, die sich maximal um eine Matrix E_Q von A_Q unterscheiden, dann führt man die neuen Variablen $\overline{\Delta Q}$ wie in (5.3) ein und erhält mit (5.2) und (5.4) die Nebenbedingungen

$$\begin{aligned} U + A_Q \Delta Q + E_Q \overline{\Delta Q} &\leq \bar{U} \mathbf{1}_{m,1}, \\ -U - A_Q \Delta Q + E_Q \overline{\Delta Q} &\leq -\underline{U} \mathbf{1}_{m,1}, \\ \bar{U} - \underline{U} &\leq EVB, \\ \Delta Q &\leq \overline{\Delta Q}, \\ -\Delta Q &\leq \overline{\Delta Q}. \end{aligned} \quad (5.7)$$

Zusammen mit der restlichen Transformation von (1.4), die in Kapitel 3.2 beschrieben wurde, erhält man also für die 1-Norm wieder ein lineares Problem der Form

$$\begin{aligned} \min c^T x \\ \text{s. t. } Ax - b &\leq 0 \end{aligned} \quad (3.1)$$

beziehungsweise für die euklidische Norm ein quadratisches Problem der Form

$$\begin{aligned} \min x^T J x \\ \text{s. t. } Ax - b &\leq 0, \end{aligned} \quad (3.8)$$

wobei $x, c \in \mathbb{R}^{3n+2}$, $b \in \mathbb{R}^{5n+2m+1}$, $A \in \mathbb{R}^{(5n+2m+1) \times (3n+2)}$ und $J \in \mathbb{R}^{(3n+2) \times (3n+2)}$ gegeben sind durch

$$x := \begin{pmatrix} \overline{Q} \\ \Delta Q \\ \overline{\Delta Q} \\ \bar{U} \\ \underline{U} \end{pmatrix}, \quad c := \begin{pmatrix} \mathbf{1}_{n,1} \\ \mathbf{0}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix}, \quad J := \text{diag} \begin{pmatrix} \mathbf{1}_{n,1} \\ \mathbf{0}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix} \quad (5.8)$$

und

$$A := \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_n & -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_n & \mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \\ \mathbf{0}_{m,n} & A_Q & E_Q & -\mathbf{1}_{m,1} & \mathbf{0}_{m,1} \\ \mathbf{0}_{m,n} & -A_Q & E_Q & \mathbf{0}_{m,1} & \mathbf{1}_{m,1} \end{pmatrix}, \quad b := \begin{pmatrix} v \\ Q \\ -Q \\ \mathbf{0}_{n,1} \\ \mathbf{0}_{n,1} \\ EVB \\ -U \\ U \end{pmatrix}, \quad (5.9)$$

mit $v_i := \min(P_i \tan(\arccos(PF_{min})), \sqrt{S_{\max,i}^2 - P_i^2})$ für $i = 1, \dots, n$. Da für $m, n \in \mathbb{N}$, mit $m \geq 1$ und $n \geq 1$,

$$5n + 2m + 1 \geq 3n + 2$$

gilt, lassen sich zur Lösung dieser Probleme die Programme aus Kapitel 3 verwenden, wenn wir zeigen können, dass A und die erweiterte Matrix $\hat{A} := \begin{pmatrix} A & -\mathbf{1}_{5n+2m+1,1} \end{pmatrix}$ für die Phase I vollen Rang haben. Dazu betrachten wir zuerst wieder eine quadratische $(3n + 2) \times (3n + 2)$ -Teilmatrix $A^\diamond$ von A :

$$A^\diamond := \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_n & -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \\ \mathbf{0}_{1,n} & a_{Q1}^T & e_{Q1}^T & -1 & 0 \end{pmatrix}.$$

Wenn man die Determinante dieser Matrix berechnen will, erhält man durch wiederholtes Anwenden des Laplaceschen Entwicklungssatzes auf die letzte Spalte

$$\det(A^\diamond) = -\det \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_n \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n \\ \mathbf{0}_n & -\mathbb{I}_n & -\mathbb{I}_n \end{pmatrix} = (-1)^{n+1} \det \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{I}_n \end{pmatrix} = -2^n \neq 0.$$

Für die Matrix $\hat{A}$ bezeichnen wir mit $\mathbb{I}_{1,n}$ den transponierten ersten n -dimensionalen Einheitsvektor und betrachten die $(3n+3) \times (3n+3)$ -Teilmatrix

$$\hat{A}^\diamond := \begin{pmatrix} \mathbb{I}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 0 & 0 & -1 \\ -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ \mathbf{0}_n & -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 & -1 \\ \mathbf{0}_{1,n} & a_{Q1}^T & e_{Q1}^T & -1 & 0 & -1 \end{pmatrix}.$$

Zur Berechnung der Determinante wird zuerst $n+2$ mal nach der vorletzten Spalte entwickelt danach wird die Entwicklung in jedem Schritt angegeben:

$$\begin{aligned} \det(\hat{A}^\diamond) &= -\det \begin{pmatrix} \mathbb{I}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & -1 \\ -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ \mathbf{0}_n & -\mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \end{pmatrix} = -\det \begin{pmatrix} \mathbb{I}_{1,n} & \mathbf{0}_{1,n} & -1 \\ -\mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ -\mathbb{I}_n & \mathbb{I}_n & -\mathbf{1}_{n,1} \end{pmatrix} \\ &\stackrel{1. \text{Zeile}}{=} -\det \begin{pmatrix} \mathbf{0}_{1,n-1} & -\mathbb{I}_n & -\mathbf{1}_{n,1} \\ -\mathbb{I}_{n-1} & & \\ \mathbf{0}_{1,n-1} & \mathbb{I}_n & -\mathbf{1}_{n,1} \\ -\mathbb{I}_{n-1} & & \end{pmatrix} + \underbrace{\det \begin{pmatrix} -\mathbb{I}_n & -\mathbb{I}_n \\ -\mathbb{I}_n & \mathbb{I}_n \end{pmatrix}}_{=(-2)^n} \\ &\stackrel{n. \text{Spalte}}{=} (-1)^{n+1} \det \begin{pmatrix} -\mathbb{I}_{n-1} & -\mathbb{I}_{n-1} & -\mathbf{1}_{n-1,1} \\ \mathbf{0}_{1,n-1} & \mathbf{0}_{1,n-1} & -1 \\ -\mathbb{I}_{n-1} & \mathbb{I}_{n-1} & -\mathbf{1}_{n-1,1} \end{pmatrix} + \det \begin{pmatrix} \mathbf{0}_{1,n-1} & \mathbf{0}_{1,n-1} & -1 \\ -\mathbb{I}_{n-1} & -\mathbb{I}_{n-1} & -\mathbf{1}_{n-1,1} \\ -\mathbb{I}_{n-1} & \mathbb{I}_{n-1} & -\mathbf{1}_{n-1,1} \end{pmatrix} + (-2)^n \\ &\stackrel{n./1. \text{Zeile}}{=} \underbrace{-\det \begin{pmatrix} -\mathbb{I}_{n-1} & -\mathbb{I}_{n-1} \\ -\mathbb{I}_{n-1} & \mathbb{I}_{n-1} \end{pmatrix}}_{=(-2)^{n-1}} - \underbrace{\det \begin{pmatrix} -\mathbb{I}_{n-1} & -\mathbb{I}_{n-1} \\ -\mathbb{I}_{n-1} & \mathbb{I}_{n-1} \end{pmatrix}}_{=(-2)^{n-1}} + (-2)^n = -(-2)^{n+1} \neq 0. \end{aligned}$$

Somit haben sowohl A als auch $\hat{A}$ vollen Rang und wir können die Funktionen `solve_lp` und `line-search_lp` beziehungsweise `solve_qlp` und `linesearch_qlp` verwenden, um eine Lösung für (1.4) zu suchen, welche die Nebenbedingungen für beliebige Matrizen $\tilde{A}_Q \in \mathbb{R}^{m \times n}$ erfüllt, die sich maximal um eine Matrix $E_Q \in \mathbb{R}^{m \times n}$ von A_Q unterscheiden. Programm 5.1 ist die MATLAB-Funktion, die die Transformation auf ein lineares Problem mit Einbeziehung der Fehlermatrix E_Q durchführt und dann die entsprechenden Algorithmen zur Lösung aufruft. Als zusätzlichen Eingabeparameter haben wir $E_Q > 0$. Für den Fall, dass man den Fehler in der Linearisierung nicht für jede Komponente abschätzen kann, sondern nur für jede Spalte, jede Zeile oder auch die ganze Matrix einen maximalen Wert kennt, wollen wir auch zulassen, dass E_Q ein n -dimensionaler Zeilenvektor, ein

m -dimensionaler Spaltenvektor oder eine reelle Zahl ist, und setzen dann im Programm

$$E_Q := \mathbf{1}_{m,n} \text{diag}(E_Q), \text{diag}(E_Q)\mathbf{1}_{m,n} \text{ oder } E_Q := E_Q \mathbf{1}_{m,n}.$$

Programm 5.1: optimize_Q_linear_error.m

```

1 function [ΔQ,exitflag,count,time]= ...
2     optimize_Q_linear_error(AQ,Q,P,U,EVB,PFmin,Smax,acc,EQ)
3
4 %-----
5 % optimize_Q_linear_error is a function to solve the optimization
6 % problem
7 %
8 %     min ||Q+ΔQ||_1
9 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %          |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %          |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %          |AQ(i,j)-AQexact(i,j)| ≤ EQ(i,j)
13 %
14 % with respect to the unknown ΔQ, where ||.|| is the 1-norm:
15 % ||x||_1:=sum(|x(i)|)
16 %-----
17
18 %-----
19 % Input parameters:
20 %
21 % AQ ..... mxn-matrix to compute an estimate for the change of
22 %           voltage
23 % Q ..... current reactive power, n-dimensional vector
24 % P ..... current active power, n-dimensional vector
25 % U ..... current voltage, m-dimensional vector
26 % EVB ..... bound on diffusion of voltage vector U, real number
27 % PFmin ..... real number
28 % Smax ..... nominal apparent power, n-dimensional vector
29 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
30 % EQ ..... mxn Matrix, m-dimensional column vector,
31 %           n-dimensional row vector or real number giving a
32 %           bound to the error arising in the calculation of AQ
33 %-----
34
35 %-----
36 % Output parameters:
37 %
38 % ΔQ ..... solution of the problem
39 % exitflag .... exitflag:
40 %               exitflag==1 --> no solution could be found, ΔQ is
41 %               not the solution
42 %               exitflag==0 --> ΔQ is the solution of the problem
43 %-----
44
45 tic
46 exitflag=0;
47
48 %-----
49 % transform the problem into a linear optimization problem with of
50 % the form
51 %

```

```

52 %      min  c'*x
53 %      s.t. A*x ≤ b
54
55 [m,n]=size(AQ);
56 v=min(P*tan(acos(PFmin)),sqrt(Smax.^2-P.^2));
57 % determine the size of EQ and transform it into a mxn-matrix
58 if size(EQ,1)==1
59     if size(EQ,2)==1
60         EQ=EQ*ones(m,n);
61     else
62         EQ=ones(m,n)*diag(EQ);
63     end
64 elseif size(EQ,2)==1
65     EQ=diag(EQ)*ones(m,n);
66 end
67
68 A=[ eye(n)      , zeros(n)      , zeros(n)      , zeros(n,1) , zeros(n,1) ;
69     -eye(n)     , -eye(n)      , zeros(n)      , zeros(n,1) , zeros(n,1) ;
70     -eye(n)     , eye(n)       , zeros(n)      , zeros(n,1) , zeros(n,1) ;
71     zeros(n)    , eye(n)       , -eye(n)       , zeros(n,1) , zeros(n,1) ;
72     zeros(n)    , -eye(n)      , -eye(n)       , zeros(n,1) , zeros(n,1) ;
73     zeros(1,n) , zeros(1,n)    , zeros(1,n)    , 1          , -1         ;
74     zeros(m,n) , AQ           , EQ            , -ones(m,1) , zeros(m,1) ;
75     zeros(m,n) , -AQ          , EQ            , zeros(m,1) , ones(m,1)  ];
76
77 b=[ v; Q; -Q; zeros(n,1); zeros(n,1); EVB; -U; U];
78 %-----
79
80 [M,N]=size(A); % determine the dimensions of the problem
81 x=zeros(N,1); % choose a starting point
82
83 %-----
84 % Phase I:
85 % find a strictly feasible point x for the constraints A*x ≤ b by
86 % solving a linear optimization problem
87 %
88 %      min  z                (phase1)
89 %      s.t. A*x-b ≤ z
90 %
91 % starting with z>max(A*x-b) and stopping as soon as z<0 is achieved
92
93 % determine the maximal violation of the constraints
94 fmax=max(A*x-b);
95 if fmax ≥ 0
96     % transform to prepare for (phase1)
97     x(N+1)=fmax+0.1;
98     A(:,N+1)=-ones(M,1);
99     c=(1:N+1==N+1)';
100    % run linear optimization algorithm for (phase1)
101    [x,count(1)]=solve_lp(c,A,b,x,acc,1);
102    if x(N+1) ≥ 0
103        % if no point (x,z) with z<0 could be found, no feasible
104        % point for the original problem can be found, so stop and
105        % set exitflag=1
106        ΔQ=x(n+1:2*n);
107        exitflag=1;
108        time=toc;
109        return
110    end
111    % (phase1) was successful, retransform the problem

```

```

112     x=x(1:N);
113     A=A(:,1:N);
114 end
115
116 %-----
117 % Phase II:
118 % If we have found a feasible point we can start the actual
119 % optimization
120 c=(1:N≤n)';
121 [x,count(2)]=solve_lp(c,A,b,x,acc,0);
122 count(3)=count(1)+count(2);
123 ΔQ=x(n+1:2*n);
124 time=toc;

```

Das Programm, das dies für (1.4) mit der euklidischen Norm in der Zielfunktion tut, `optimize_Q_quad_error`, ist im Kapitel A.5 zu finden.

5.2. Steuerung der Wirkleistung und der Blindleistung

Um auch für (1.5) einen möglichen Fehler in der Berechnung von A zu berücksichtigen, gehen wir analog zum vorigen Abschnitt vor und schreiben zuerst die Nebenbedingung

$$\max(U + A_P \Delta P + A_Q \Delta Q) - \min(U + A_P \Delta P + A_Q \Delta Q) \leq EVB$$

um auf die $2m + 1$ dazu äquivalenten Ungleichungen:

$$\begin{aligned}
 U + A_P \Delta P + A_Q \Delta Q &\leq \bar{U} \mathbf{1}_{m,1}, \\
 -U - A_P \Delta P - A_Q \Delta Q &\leq -\underline{U} \mathbf{1}_{m,1}, \\
 \bar{U} - \underline{U} &\leq EVB.
 \end{aligned}$$

Nun seien wieder E_P und E_Q die Matrizen mit nichtnegativen Eintragungen, um die sich A_P und A_Q maximal von den exakten Matrizen $\tilde{A}_P$ und $\tilde{A}_Q$ unterscheiden. Wenn man die neuen Variablen $\overline{\Delta Q}$ wie in (5.3) einführt und bemerkt, dass $\Delta P_i < 0$ gilt für $i = 1, \dots, m$, dann erhält man mit (5.2) und (5.4) die Nebenbedingungen

$$\begin{aligned}
 U + A_P \Delta P - E_P \Delta P + A_Q \Delta Q + E_Q \overline{\Delta Q} &\leq \bar{U} \mathbf{1}_{m,1}, \\
 -U - A_P \Delta P - E_P \Delta P - A_Q \Delta Q + E_Q \overline{\Delta Q} &\leq -\underline{U} \mathbf{1}_{m,1}, \\
 \bar{U} - \underline{U} &\leq EVB, \\
 \Delta Q &\leq \overline{\Delta Q}, \\
 -\Delta Q &\leq \overline{\Delta Q}.
 \end{aligned}$$

Damit und mit der Transformation der übrigen Ungleichungen in (1.5) lässt sich das Problem jetzt wieder auf die Gestalt

$$\begin{aligned} & \min c^T x \\ & \text{s. t. } Ax - b \leq 0, \\ & \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d \leq 0, \end{aligned} \quad (4.5)$$

beziehungsweise

$$\begin{aligned} & \min x^T Jx \\ & \text{s. t. } Ax - b \leq 0, \\ & \text{diag}(Kx + 2P)Kx + \text{diag}(Lx + 2Q)Lx - d \leq 0, \end{aligned} \quad (4.9)$$

für die euklidische Norm in der Zielfunktion, bringen, wobei $d \in \mathbb{R}^n$ wie in (4.4) sei und $x, c \in \mathbb{R}^{3n+2}$, $b \in \mathbb{R}^{6n+2m+1}$, $A \in \mathbb{R}^{(6n+2m+1) \times (3n+2)}$, $K, L \in \mathbb{R}^{n \times (3n+2)}$ sowie $J \in \mathbb{R}^{(3n+2) \times (3n+2)}$ gegeben sind durch

$$x := \begin{pmatrix} \Delta P \\ \Delta Q \\ \overline{\Delta Q} \\ \overline{U} \\ \underline{U} \end{pmatrix}, \quad c := \begin{pmatrix} -\mathbf{1}_n \\ \mathbf{0}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix}, \quad J := \text{diag} \begin{pmatrix} \mathbf{1}_n \\ \mathbf{0}_{n,1} \\ \mathbf{0}_{n,1} \\ 0 \\ 0 \end{pmatrix},$$

$$A := \begin{pmatrix} -\mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_n & \mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_n & -\mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \\ A_P - E_P & A_Q & E_Q & -\mathbf{1}_{m,1} & \mathbf{0}_{m,1} \\ -A_P - E_P & -A_Q & E_Q & \mathbf{0}_{m,1} & \mathbf{1}_{m,1} \end{pmatrix}, \quad b := \begin{pmatrix} P \\ \mathbf{0}_{n,1} \\ \tan(\varphi)P - Q \\ \tan(\varphi)P + Q \\ \mathbf{0}_{n,1} \\ \mathbf{0}_{n,1} \\ EVB \\ -U \\ U \end{pmatrix},$$

und

$$\begin{aligned} K &:= \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \end{pmatrix}, \\ L &:= \begin{pmatrix} \mathbf{0}_n & \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \end{pmatrix}. \end{aligned}$$

Betrachtet man nun wieder die quadratischen Teilmatrizen $A^\diamond$ und $\hat{A}^\diamond$ von A beziehungsweise von $\hat{A} := \begin{pmatrix} A & -\mathbf{1}_{6n+2m+1,1} \end{pmatrix}$, gegeben durch

$$A^\diamond := \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ \mathbf{0}_n & \mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ a_{P1}^T - e_{P1}^T & a_{Q1}^T & e_{Q1}^T & -1 & 0 \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 \end{pmatrix},$$

und

$$\hat{A}^\diamond := \begin{pmatrix} -\mathbb{I}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 0 & 0 & -1 \\ \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ \mathbf{0}_n & \mathbb{I}_n & -\mathbb{I}_n & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} & -\mathbf{1}_{n,1} \\ a_{P1}^T - e_{P1}^T & a_{Q1}^T & e_{Q1}^T & -1 & 0 & -1 \\ \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & 1 & -1 & -1 \end{pmatrix}$$

so stellt man zuerst wieder fest, dass $A^\diamond$ eine untere Dreiecksmatrix ist, die vollen Rang hat, da all ihre Diagonaleinträge verschieden von 0 sind. Um den Rang von $\hat{A}^\diamond$ zu bestimmen, berechnet man analog zur Vorgehensweise in Abschnitt 4.1 die Determinante, indem man zuerst zweimal nach der vorletzten Spalte entwickelt:

$$\det(\hat{A}^\diamond) = \det \begin{pmatrix} -\mathbb{I}_{1,n} & \mathbf{0}_{1,n} & \mathbf{0}_{1,n} & -1 \\ \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ \mathbf{0}_n & \mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \end{pmatrix}.$$

Wendet man nun weiters zweimal den Laplaceschen Entwicklungssatz auf die erste Zeile an, so ergibt sich:

$$\begin{aligned} \det(\hat{A}^\diamond) &= -\det \begin{pmatrix} \mathbf{0}_{1,n-1} & \mathbf{0}_n & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ \mathbb{I}_{n-1} & & & \\ \mathbf{0}_{1,n-1} & \mathbb{I}_n & \mathbf{0}_n & -\mathbf{1}_{n,1} \\ -\tan(\varphi)\mathbb{I}_{n-1} & & & \\ \mathbf{0}_n & \mathbb{I}_n & -\mathbb{I}_n & -\mathbf{1}_{n,1} \end{pmatrix} + (-1)^{n+1} \det \begin{pmatrix} \mathbb{I}_n & \mathbf{0}_n & \mathbf{0}_n \\ -\tan(\varphi)\mathbb{I}_n & \mathbb{I}_n & \mathbf{0}_n \\ \mathbf{0}_n & \mathbb{I}_n & -\mathbb{I}_n \end{pmatrix} \\ &= (-1)^{n+1} \det \begin{pmatrix} \mathbb{I}_{n-1} & \mathbf{0}_{n-1,n} & \mathbf{0}_{n-1,n} \\ \mathbf{0}_{1,n-1} & \mathbb{I}_n & \mathbf{0}_n \\ -\tan(\varphi)\mathbb{I}_{n-1} & & \\ \mathbf{0}_{n,n-1} & \mathbb{I}_n & -\mathbb{I}_n \end{pmatrix} -1 = -2 \neq 0 \end{aligned}$$

$\underbrace{\hspace{10em}}_{=(-1)^n}$

Also haben A und $\hat{A}$ vollen Rang und man kann die MATLAB-Programme `solve_lqp`, `line-search_lqp`, `solve_qqp` und `linesearch_qqp`, die in den Abschnitten 4.2 und 4.3 beschrieben wurden, zum Auffinden einer Lösung verwenden. Analog zum vorigen Abschnitt geben wir eine MATLAB-Funktion mit den zusätzlichen Eingabeparametern E_P und E_Q an, die das Problem (1.5) unter Berücksichtigung eines Fehlers in den Matrizen A_P und A_Q auf konvexe Form transformiert und die entsprechenden Algorithmen zur Lösung davon aufruft:

Programm 5.2: `optimize_P_Q_linear_error.m`

```

1 function [ΔP, ΔQ, exitflag, count, time]=...
2     optimize_P_Q_linear_error(AP, AQ, P, Q, U, EVB, PFmin, Smax, acc, EP, EQ)
3
4 %-----
5 % optimize_P_Q_linear_error is a function to solve the optimization
6 % problem
7 %
8 %     min ||Q+ΔQ||_1
9 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %         |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %         |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %         |AP(i,j)-APexact(i,j)| ≤ EP(i,j)
13 %         |AQ(i,j)-AQexact(i,j)| ≤ EQ(i,j)
14 %
15 % with respect to the unknown ΔQ, where ||.||_1 is the 1-norm:
16 % ||x||_1:=sum(|x(i)|)
17 %-----
18
19 %-----
20 % Input parameters:
21 %
22 % AP ..... mxn-matrix to compute an estimate for the change of
23 %           voltage
24 % AQ ..... mxn-matrix to compute an estimate for the change of
25 %           voltage
26 % P ..... current active power, n-dimensional vector
27 % Q ..... current reactive power, n-dimensional vector
28 % U ..... current voltage, m-dimensional vector
29 % EVB ..... bound on diffusion of voltage vector U, real number
30 % PFmin ..... real number
31 % Smax ..... nominal apparent power, n-dimensional vector
32 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
33 % EP ..... mxn Matrix, m-dimensional column vector,
34 %           n-dimensional row vector or real number giving a
35 %           bound to the error arising in the calculation of AP
36 % EQ ..... mxn Matrix, m-dimensional column vector,
37 %           n-dimensional row vector or real number giving a
38 %           bound to the error arising in the calculation of AQ
39 %-----
40
41 %-----
42 % Output parameters:
43 %
44 % ΔQ ..... solution of the problem
45 % exitflag .... exitflag:
46 %               exitflag==1 --> no solution could be found, ΔQ is
47 %               not the solution

```

```

48 %                               exitflag==0 --> ΔQ is the solution of the problem
49 %-----
50
51 tic
52 exitflag=0;
53
54 %-----
55 % transform the problem into a optimization problem of the form
56 %      min   c'*x
57 %      s.t.  A*x ≤ b
58 %      diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
59
60 [m,n]=size(AQ);
61 tp=tan(acos(PFmin));
62 % determine the size of EP and transform it into a mxn-matrix
63 if size(EP,1)==1
64     if size(EP,2)==1
65         EP=EP*ones(m,n);
66     else
67         EP=ones(m,n)*diag(EP);
68     end
69 elseif size(EP,2)==1
70     EP=diag(EP)*ones(m,n);
71 end
72 % determine the size of EQ and transform it into a mxn-matrix
73 if size(EQ,1)==1
74     if size(EQ,2)==1
75         EQ=EQ*ones(m,n);
76     else
77         EQ=ones(m,n)*diag(EQ);
78     end
79 elseif size(EQ,2)==1
80     EQ=diag(EQ)*ones(m,n);
81 end
82
83 A=[-eye(n) , zeros(n) , zeros(n) , zeros(n,1), zeros(n,1);
84     eye(n) , zeros(n) , zeros(n) , zeros(n,1), zeros(n,1);
85     -tp*eye(n), eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
86     -tp*eye(n), -eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
87     zeros(n) , eye(n) , -eye(n) , zeros(n,1), zeros(n,1);
88     zeros(n) , -eye(n) , -eye(n) , zeros(n,1), zeros(n,1);
89     zeros(1,n), zeros(1,n), zeros(1,n), 1 , -1 ;
90     AP-EP , AQ , EQ , -ones(m,1), zeros(m,1);
91     -AP-EP , -AQ , EQ , zeros(m,1), ones(m,1) ];
92
93 b=[P; zeros(n,1); tp*P-Q; tp*P+Q; zeros(n,1); zeros(n,1); EVB; -U; U];
94
95 K=[eye(n), zeros(n), zeros(n), zeros(n,1), zeros(n,1)];
96 L=[zeros(n), eye(n), zeros(n), zeros(n,1), zeros(n,1)];
97 d=Smax.^2-P.^2-Q.^2;
98
99 %-----
100
101 [M,N]=size(A); % determine the dimensions of the problem
102 x=zeros(N,1); % choose a starting point
103
104 %-----
105 % Phase I:
106 % find a strictly feasible point x for the constraints A*x≤b by
107 % solving a linear optimization problem

```

```

108 %
109 %      min      z                      (phase1)
110 %      s.t.  A*x-b ≤ z
111 %            diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d ≤ z
112 %
113 % starting with z>max(A*x-b,diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d)
114 % and stopping as soon as z<0 is achieved
115
116 % determine the maximal violation of the constraints
117 fmax=max([-b;-d]);
118 if fmax≥0
119     % transform to prepare for (phase1)
120     x(N+1)=fmax+0.1;
121     A(:,N+1)=-ones(M,1);
122     K(:,N+1)=zeros(n,1);
123     L(:,N+1)=zeros(n,1);
124     c=(1:N+1==N+1)';
125     % run linear optimization algorithm for (phase1)
126     [x,count(1)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,1);
127     if x(N+1)≥0
128         % if no point (x,z) with z<0 could be found, no feasible
129         % point for the original problem can be found, so stop and
130         % set exitflag=1
131         ΔP=x(1:n);
132         ΔQ=x(n+1:2*n);
133         exitflag=1;
134         time=toc;
135         count(2)=0;
136         count(3)=count(1);
137         return
138     end
139     % (phase1) was successful, retransform the problem
140     x=x(1:N);
141     A=A(:,1:N);
142     K=K(:,1:N);
143     L=L(:,1:N);
144 end
145
146 % If we have found a feasible point we can start the actual
147 % optimization
148 c=-(1:N≤n)';
149 [x,count(2)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,0);
150 count(3)=count(1)+count(2);
151 ΔP=x(1:n);
152 ΔQ=x(n+1:2*n);
153 time=toc;

```

6. Numerische Tests

In diesem Kapitel werden zuerst anhand konkreter Beispiele die Funktionsweise und die Unterschiede der einzelnen Programme veranschaulicht und daraufhin versucht, anhand mehrerer zufällig generierter Beispieldaten einen Eindruck vom Aufwand in Abhängigkeit von der Dimension des Problems und von der gewählten Genauigkeit zu gewinnen.

6.1. Konkretes Beispiel für die Steuerung der Blindleistung

Da das Projekt des AIT auch in einem kleinen Netz in Vorarlberg getestet wird, stehen konkrete Beispieldaten zur Verfügung. In diesem Netz wurden durch eine Simulation elf kritische Knoten ermittelt und es speisen vier regelbare Anlagen ein. Hier werden nun die vier Programme zur Spannungsregelung durch Steuerung der Blindleistung `optimize_Q_linear`, `optimize_Q_linear_error`, `optimize_Q_quad` und `optimize_Q_quad_error` auf einen konkreten Beispieldatensatz angewandt und die jeweiligen Ergebnisse untereinander verglichen. Die Eingabewerte sind gegeben durch

$$A_Q := \begin{pmatrix} 0.0061 & 0.0084 & 0.0091 & 0.0091 \\ 0.0038 & 0.0038 & 0.0038 & 0.0038 \\ 0.0038 & 0.0038 & 0.0038 & 0.0038 \\ 0.006 & 0.0098 & 0.0084 & 0.0084 \\ 0.0038 & 0.0037 & 0.0037 & 0.0037 \\ 0.0033 & 0.0033 & 0.0033 & 0.0033 \\ 0.0033 & 0.0033 & 0.0033 & 0.0033 \\ 0.0033 & 0.0033 & 0.0033 & 0.0033 \\ 0.0034 & 0.0033 & 0.0033 & 0.0033 \\ 0.0074 & 0.006 & 0.0061 & 0.0061 \\ 0.0038 & 0.0037 & 0.0037 & 0.0037 \end{pmatrix}, \quad U := \begin{pmatrix} 1.0108 \\ 0.9385 \\ 0.9384 \\ 1.0119 \\ 0.9678 \\ 0.966 \\ 0.9649 \\ 0.9759 \\ 0.9737 \\ 0.9897 \\ 0.969 \end{pmatrix},$$

$$P := \begin{pmatrix} 0.6907 \\ 1.381 \\ 1.9156 \\ 1.9156 \end{pmatrix}, \quad Q := \begin{pmatrix} 0 \\ 0 \\ 0.0001 \\ 0.0001 \end{pmatrix}, \quad S_{\max} := \begin{pmatrix} 1.1344 \\ 2.2689 \\ 2.1956 \\ 2.1956 \end{pmatrix},$$

$EVB := 0.0698$ und $PF_{\min} := 0.9$. Für den Vektor U , der die Spannungswerte an den kritischen Knoten enthält, haben wir

$$\max(U) - \min(U) = 0.0735 > EVB.$$

Also lässt sich die Spannung nicht durch den Stufenschalter allein in den richtigen Bereich bringen. Wählt man nun als Genauigkeitsparameter $\text{acc} := 10^{-8}$ und ruft mit diesen Eingabewerten das Programm `optimize_Q_linear` auf, so findet dieses nach 11 Iterationsschritten in der Phase I den Punkt

$$\Delta Q_I := \begin{pmatrix} -0.0297936 \\ -0.328137 \\ -0.497131 \\ -0.497131 \end{pmatrix},$$

der die Nebenbedingungen

$$\max(U + A_Q \Delta Q_I) - \min(U + A_Q \Delta Q_I) = 0.0669 \leq 0.0698 = EVB,$$

$$|Q + \Delta Q_I| = \begin{pmatrix} 0.0297936 \\ 0.328137 \\ 0.497031 \\ 0.497031 \end{pmatrix} \leq \begin{pmatrix} 0.334521 \\ 0.668849 \\ 0.927767 \\ 0.927767 \end{pmatrix} = P \tan(\arccos(PF_{\min})),$$

$$|Q + \Delta Q_I| = \begin{pmatrix} 0.0297936 \\ 0.328137 \\ 0.497031 \\ 0.497031 \end{pmatrix} \leq \begin{pmatrix} 0.8999 \\ 1.8002 \\ 1.0729 \\ 1.0729 \end{pmatrix} = \sqrt{S_{\max}^2 - P^2}$$

erfüllt, wobei hier der Betrag, das Wurzelziehen und das Quadrieren komponentenweise zu verstehen ist. Lässt man das Programm nun weiterlaufen, dann gibt es nach 14 weiteren Iterationsschritten den Vektor

$$\Delta Q_{\text{lin}} := \begin{pmatrix} -9.58336 \cdot 10^{-11} \\ -0.616513 \\ -0.0001 \\ -0.0001 \end{pmatrix}$$

zurück, der natürlich auch die Nebenbedingungen

$$\max(U + A_Q \Delta Q_{\text{lin}}) - \min(U + A_Q \Delta Q_{\text{lin}}) = 0.0698 \leq 0.0698 = EVB$$

und

$$|Q + \Delta Q_{\text{lin}}| = \begin{pmatrix} 9.58336 \cdot 10^{-11} \\ 0.616513 \\ 4.14317 \cdot 10^{-10} \\ 4.14317 \cdot 10^{-10} \end{pmatrix} \leq \begin{pmatrix} 0.334521 \\ 0.668849 \\ 0.927767 \\ 0.927767 \end{pmatrix} = \min(P \tan(\arccos(PF_{\min})), \sqrt{S_{\max}^2 - P^2})$$

erfüllt. Für die Werte der Zielfunktion gilt

$$\|Q + \Delta Q_{\text{lin}}\|_1 = 0.6165 < 1.3520 = \|Q + \Delta Q_I\|_1.$$

An diesem Ergebnis ist auch schön zu sehen, dass für die Minimierung der 1-Norm die ganze Gewichtung von $Q + \Delta Q_{\text{lin}}$ auf einer Komponente liegt, in diesem Fall auf der zweiten. Ruft man nun mit demselben Input `optimize_Q_quad` auf, so wird in der Phase I natürlich in genau gleich vielen Schritten derselbe Wert für ΔQ_I wie in `optimize_Q_linear` gefunden, da ja dieselben Funktionen `solve_lp` und `linesearch_lp` verwendet werden. Als Lösung des Optimierungsproblems erhält man nach 14 weiteren Iterationsschritten den Vektor

$$\Delta Q_{\text{quad}} := \begin{pmatrix} -0.0978593 \\ -0.266889 \\ -0.204715 \\ -0.204715 \end{pmatrix}.$$

Für die Nebenbedingungen gilt wieder

$$\max(U + A_Q \Delta Q_{\text{quad}}) - \min(U + A_Q \Delta Q_{\text{quad}}) = 0.0698 \leq 0.0698 = EVB$$

und

$$|Q + \Delta Q_{\text{quad}}| = \begin{pmatrix} 0.0978593 \\ 0.266889 \\ 0.204615 \\ 0.204615 \end{pmatrix} \leq \begin{pmatrix} 0.334521 \\ 0.668849 \\ 0.927767 \\ 0.927767 \end{pmatrix} = \min(P \tan(\arccos(PF_{\min})), \sqrt{S_{\max}^2 - P^2}).$$

Außerdem haben wir

$$\|Q + \Delta Q_{\text{quad}}\|_2 = 0.4056 < 0.7763 = \|Q + \Delta Q_I\|_2$$

und natürlich gilt auch

$$\begin{aligned} \|Q + \Delta Q_{\text{quad}}\|_2 &= 0.4056 < 0.6165 = \|Q + \Delta Q_{\text{lin}}\|_2, \\ \|Q + \Delta Q_{\text{lin}}\|_1 &= 0.6165 < 0.7740 = \|Q + \Delta Q_{\text{quad}}\|_1. \end{aligned}$$

Es ist auch schön zu sehen, dass sich bei der Wahl der euklidischen Norm in der Zielfunktion die Gewichtung der Komponenten von $Q + \Delta Q_{\text{quad}}$ viel gleichmäßiger verteilt als bei $Q + \Delta Q_{\text{lin}}$. Geht man nun zum Beispiel davon aus, dass bei der Berechnung von A_Q in jeder Komponente ein Fehler von maximal $\varepsilon = 10^{-4}$ auftritt, so kann man das mitberücksichtigen, indem man `optimize_Q_linear_error` mit ε als zusätzlichen Eingabeparameter aufruft, und erhält dann nach 11 Phase I-Schritten und 20 weiteren Iterationen die Lösung

$$\Delta Q_{\text{lin}}^\varepsilon := \begin{pmatrix} -1.74612 \cdot 10^{-10} \\ -0.637779 \\ -0.000100001 \\ -0.000100001 \end{pmatrix}.$$

Da die Nebenbedingungen strenger sind, wenn man einen Fehler mitberücksichtigt, das heißt, da der zulässige Bereich für das Optimierungsproblem kleiner wird, gilt erwartungsgemäß

$$\|Q + \Delta Q_{\text{lin}}^\varepsilon\|_1 = 0.6378 > 0.6165 = \|Q + \Delta Q_{\text{lin}}\|_1.$$

Wenn man aber in A_Q zu jeder Komponente der dritten Zeile ε addiert und von jeder Komponente der vierten Zeile ε subtrahiert und die so erhaltene Matrix mit A_Q^ε bezeichnet, dann gilt

$$\begin{aligned} \max(U + A_Q^\varepsilon \Delta Q_{\text{lin}}) - \min(U + A_Q^\varepsilon \Delta Q_{\text{lin}}) &= 0.0699 > 0.0698 = \text{EVB}, \\ \max(U + A_Q^\varepsilon \Delta Q_{\text{lin}}^\varepsilon) - \min(U + A_Q^\varepsilon \Delta Q_{\text{lin}}^\varepsilon) &= 0.0698 \leq 0.0698 = \text{EVB}. \end{aligned}$$

Analoges gilt natürlich auch für die euklidische Norm: `optimize_Q_quad_error` liefert mit $\varepsilon = 10^{-4}$ nach 11 + 20 Iterationsschritten die Lösung

$$\Delta Q_{\text{quad}}^\varepsilon := \begin{pmatrix} -0.0968863 \\ -0.28097 \\ -0.21325 \\ -0.21325 \end{pmatrix},$$

mit

$$\|Q + \Delta Q_{\text{quad}}^\varepsilon\|_2 = 0.4233 > 0.4056 = \|Q + \Delta Q_{\text{quad}}\|_2.$$

Allerdings haben wir für dieselbe Matrix A_Q^ε wie oben

$$\begin{aligned} \max(U + A_Q^\varepsilon \Delta Q_{\text{quad}}) - \min(U + A_Q^\varepsilon \Delta Q_{\text{quad}}) &= 0.06995 > 0.0698 = \text{EVB}, \\ \max(U + A_Q^\varepsilon \Delta Q_{\text{quad}}^\varepsilon) - \min(U + A_Q^\varepsilon \Delta Q_{\text{quad}}^\varepsilon) &= 0.0698 \leq 0.0698 = \text{EVB}. \end{aligned}$$

Dazu sei noch erwähnt, dass für $\varepsilon = 10^{-2}$ zum Beispiel keine Lösung mehr gefunden werden kann, also die Nebenbedingungen inkonsistent sind. Ruft man die Programme `optimize_Q_linear_error`

und `optimize_Q_quad_error` mit $E_Q = 0$ auf so erhält man die Ergebnisse

$$\Delta Q_{\text{lin}}^0 = \begin{pmatrix} -4.13259 \cdot 10^{-11} \\ -0.616513 \\ -0.0001 \\ -0.0001 \end{pmatrix}, \quad \Delta Q_{\text{quad}}^0 = \begin{pmatrix} -0.0978593 \\ -0.266889 \\ -0.204715 \\ -0.204715 \end{pmatrix}.$$

Diese Werte stimmen zwar nicht exakt mit denen von ΔQ_{lin} und ΔQ_{quad} überein, da unterschiedliche Suchrichtungen verwendet werden, die Differenz ist aber für `optimize_Q_linear_error` jedenfalls kleiner als die gewählte Genauigkeit. Für `optimize_Q_quad_error` lässt sich diese Abweichung durch die Wurzel der Genauigkeit beschränken, da in der Zielfunktion ein Quadrat vorkommt:

$$\Delta Q_{\text{lin}}^0 - \Delta Q_{\text{lin}} = \begin{pmatrix} 5.45077 \cdot 10^{-11} \\ -1.9684 \cdot 10^{-10} \\ 2.41967 \cdot 10^{-10} \\ 2.41967 \cdot 10^{-10} \end{pmatrix}, \quad \Delta Q_{\text{quad}}^0 - \Delta Q_{\text{quad}} = \begin{pmatrix} -1.65137 \cdot 10^{-11} \\ 1.04418 \cdot 10^{-10} \\ 8.55456 \cdot 10^{-11} \\ 8.55457 \cdot 10^{-11} \end{pmatrix}.$$

6.2. Konkretes Beispiel für die Steuerung der Wirk- und Blindleistung

Man betrachte nun ebenfalls wieder ein Stromnetz mit elf kritischen Knoten und vier regelbaren Anlagen, in dem die Matrizen A_P und A_Q zur Berechnung der Spannungsänderung an den kritischen Knoten gegeben seien durch

$$A_P := \begin{pmatrix} 0.4648 & 0.9052 & 0.1341 & 0.3074 \\ 0.764 & 0.6754 & 0.2126 & 0.4561 \\ 0.8182 & 0.4685 & 0.8949 & 0.1017 \\ 0.1002 & 0.9121 & 0.0715 & 0.9954 \\ 0.1781 & 0.104 & 0.2425 & 0.3321 \\ 0.3596 & 0.7455 & 0.0538 & 0.2973 \\ 0.0567 & 0.7363 & 0.4417 & 0.062 \\ 0.5219 & 0.5619 & 0.0133 & 0.2982 \\ 0.3358 & 0.1842 & 0.8972 & 0.0464 \\ 0.1757 & 0.5972 & 0.1967 & 0.5054 \\ 0.2089 & 0.2999 & 0.0934 & 0.7614 \end{pmatrix}, \quad A_Q := \begin{pmatrix} 0.1057 & 0.9344 & 0.2955 & 0.1771 \\ 0.142 & 0.9844 & 0.3329 & 0.6628 \\ 0.1665 & 0.8589 & 0.4671 & 0.3308 \\ 0.621 & 0.7856 & 0.6482 & 0.8985 \\ 0.5737 & 0.5134 & 0.0252 & 0.1182 \\ 0.0521 & 0.1776 & 0.8422 & 0.9884 \\ 0.9312 & 0.3986 & 0.559 & 0.54 \\ 0.7287 & 0.1339 & 0.8541 & 0.7069 \\ 0.7378 & 0.0309 & 0.3479 & 0.9995 \\ 0.0634 & 0.9391 & 0.446 & 0.2878 \\ 0.8604 & 0.3013 & 0.0542 & 0.4145 \end{pmatrix}.$$

Die aktuellen Werte für den Spannungsvektor U , den Wirkleistungsvektor P und den Blindleistungsvektor Q sowie der Vektor $S_{\max}$, der die maximale Scheinleistung der jeweiligen Kraftwerke enthält, seien

$$U := \begin{pmatrix} 1.0303 \\ 1.0753 \\ 1.0882 \\ 1.0197 \\ 1.0091 \\ 1.0147 \\ 0.9982 \\ 1.0378 \\ 1.048 \\ 1.003 \\ 1.0301 \end{pmatrix}, \quad P := \begin{pmatrix} 0.3345 \\ 0.0163 \\ 0.1055 \\ 0.1014 \end{pmatrix}, \quad Q := \begin{pmatrix} 0.0425 \\ -0.0065 \\ -0.0428 \\ 0.0272 \end{pmatrix}, \quad S_{\max} := \begin{pmatrix} 2.2423 \\ 1.5513 \\ 1.223 \\ 1.9308 \end{pmatrix},$$

und wie im vorigen Beispiel seien

$$EVB := 0.0698, \quad PF_{\min} := 0.9 \text{ und } acc := 10^{-8}.$$

Für den aktuellen Spannungsvektor gilt auch hier wieder

$$\max(U) - \min(U) = 0.09 > 0.0698 = EVB.$$

Daher lässt sich die Spannung nicht alleine mit der Stufenschaltung in den richtigen Bereich bringen. Also wird die zentrale Steuerungseinheit ein Programm zur Spannungsregelung durch Steuerung der Blindleistung aufrufen. Für diesen Input aber gibt `optimize_Q_linear` den Wert `exitflag = 1` zurück. Es kann also kein zulässiger Punkt gefunden werden. Folglich lässt sich das Problem auch nicht mit `optimize_Q_quad`, `optimize_Q_linear_error` oder `optimize_Q_quad_error` lösen, da diese Funktionen dieselben oder strengere Nebenbedingungen betrachten. Das heißt, der zentrale Regler wird im nächsten Schritt eine der `optimize_P_Q_*`-Funktionen aufrufen. Mit `optimize_P_Q_linear` erhält man nach 9 Iterationsschritten in der Phase I einen zulässigen Punkt $(\Delta P_I, \Delta Q_I)$ mit

$$\Delta P_I := \begin{pmatrix} -0.112331 \\ -0.00502838 \\ -0.0295195 \\ -0.0274098 \end{pmatrix}, \quad \Delta Q_I := \begin{pmatrix} -0.0111301 \\ 0.00682284 \\ 0.0356947 \\ -0.0232929 \end{pmatrix}.$$

Für die Nebenbedingungen an dieser Stelle gilt

$$\begin{aligned} \max(U + \Delta U_I) - \min(U + \Delta U_I) &= 0.015 \leq 0.0698 = EVB, \\ |Q + \Delta Q_I| - \tan(\arccos(PF_{\min}))|P + \Delta P_I| &= \begin{pmatrix} -0.0762315 \\ -0.00513626 \\ -0.0296937 \\ -0.0319281 \end{pmatrix} \leq 0, \\ (Q + \Delta Q_I)^2 + (P + \Delta P_I)^2 &= \begin{pmatrix} 0.0503431 \\ 0.000127154 \\ 0.00582352 \\ 0.00548982 \end{pmatrix} \leq \begin{pmatrix} 5.02791 \\ 2.40653 \\ 1.49573 \\ 3.72799 \end{pmatrix} = S_{\max}^2, \\ -P &\leq \Delta P_I \leq 0, \end{aligned}$$

wobei $\Delta U_I := A_P \Delta P_I + A_Q \Delta Q_I$ gesetzt wurde und der Betrag und das Quadrieren hier wieder komponentenweise zu verstehen ist. Nach 19 weiteren Iterationsschritten erhält man die Lösung von (1.5), gegeben durch

$$\Delta P_{\text{lin}} := \begin{pmatrix} -8.6468 \cdot 10^{-10} \\ -1.77316 \cdot 10^{-11} \\ -0.0234218 \\ -1.42184 \cdot 10^{-11} \end{pmatrix}, \quad \Delta Q_{\text{lin}} := \begin{pmatrix} 0.0229626 \\ 0.0143944 \\ 0.00780082 \\ -0.0098727 \end{pmatrix}.$$

Natürlich erfüllt auch dieser Punkt die Nebenbedingungen:

$$\begin{aligned} \max(U + \Delta U_{\text{lin}}) - \min(U + \Delta U_{\text{lin}}) &= 0.0698 \leq 0.0698 = EVB, \\ |Q + \Delta Q_{\text{lin}}| - \tan(\arccos(PF_{\min}))|P + \Delta P_{\text{lin}}| &= \begin{pmatrix} -0.0965432 \\ -1.13168 \cdot 10^{-8} \\ -0.00475312 \\ -0.031783 \end{pmatrix} \leq 0, \\ (Q + \Delta Q_{\text{lin}})^2 + (P + \Delta P_{\text{lin}})^2 &= \begin{pmatrix} 0.116176 \\ 0.000328012 \\ 0.00796178 \\ 0.0105822 \end{pmatrix} \leq \begin{pmatrix} 5.02791 \\ 2.40653 \\ 1.49573 \\ 3.72799 \end{pmatrix} = S_{\max}^2, \\ -P &\leq \Delta P_{\text{lin}} \leq 0. \end{aligned}$$

Außerdem gilt erwartungsgemäß für den Wert der Zielfunktion

$$\|\Delta P_{\text{lin}}\|_1 = 0.0234218 < 0.174289 = \|\Delta P_I\|_1.$$

Hier sei noch bemerkt, dass man schon an der Lösung ΔP_{lin} ablesen kann, dass es zu diesem Beispiel keine Lösung für die Steuerung der Blindleistung geben kann, da sich sonst die einzelnen Komponenten von ΔP_{lin} maximal durch die gewählte Genauigkeit $\text{acc} = 10^{-8}$ von 0 unterscheiden dürften. Will man einen zulässigen Punkt finden, der die euklidische Norm in der Zielfunktion minimiert, so ruft man mit demselben Input wie vorher die Funktion `optimize_P_Q_quad` auf und erhält nach 9 Schritten in der Phase I natürlich denselben Punkt $(\Delta P_I, \Delta Q_I)$ wie für das lineare Problem und nach 21 weiteren Iterationen die Lösung

$$\Delta P_{\text{quad}} := \begin{pmatrix} -0.0117053 \\ -7.99042 \cdot 10^{-9} \\ -0.0120327 \\ -5.56551 \cdot 10^{-9} \end{pmatrix}, \quad \Delta Q_{\text{quad}} := \begin{pmatrix} 0.0162548 \\ 0.014394 \\ -0.00246758 \\ 0.00122131 \end{pmatrix}.$$

Auch dieser Punkt erfüllt wieder die Nebenbedingungen:

$$\begin{aligned} \max(U + \Delta U_{\text{quad}}) - \min(U + \Delta U_{\text{quad}}) &= 0.0698 \leq 0.0698 = \text{EVB}, \\ |Q + \Delta Q_{\text{quad}}| - \tan(\arccos(PF_{\min}))|P + \Delta P_{\text{quad}}| &= \begin{pmatrix} -0.0975818 \\ -4.19672 \cdot 10^{-7} \\ -7.00073 \cdot 10^{-7} \\ -0.0206889 \end{pmatrix} \leq 0, \\ (Q + \Delta Q_{\text{quad}})^2 + (P + \Delta P_{\text{quad}})^2 &= \begin{pmatrix} 0.107649 \\ 0.000328005 \\ 0.0107853 \\ 0.0110897 \end{pmatrix} \leq \begin{pmatrix} 5.02791 \\ 2.40653 \\ 1.49573 \\ 3.72799 \end{pmatrix} = S_{\max}^2, \\ -P &\leq \Delta P_{\text{quad}} \leq 0. \end{aligned}$$

Vergleicht man die Werte der der Zielfunktionen für die jeweilige Norm miteinander, dann gilt:

$$\begin{aligned} \|\Delta P_{\text{quad}}\|_2 &= 0.0167869 < 0.119441 = \|\Delta P_I\|_2, \\ \|\Delta P_{\text{quad}}\|_2 &= 0.0167869 < 0.0234218 = \|\Delta P_{\text{lin}}\|_2, \\ \|\Delta P_{\text{quad}}\|_1 &= 0.0237381 > 0.0234218 = \|\Delta P_{\text{lin}}\|_1. \end{aligned}$$

Betrachtet man nun ΔP_{lin} , so sieht man, dass die ganze Gewichtung auf der dritten Komponente liegt, während diese sich für ΔP_{quad} gleichmäßiger verteilt, soweit es die Nebenbedingungen zulassen. Zuletzt wollen wir für dieses Beispiel noch Lösungen suchen, die einen Fehler in der Berechnung von A_P und A_Q mitberücksichtigen. Dazu nehmen wir wieder an, dass dieser Fehler für beide Matrizen in jeder Komponente maximal $\varepsilon = 10^{-4}$ und rufen dann für die 1-Norm in der Zielfunktion das Programm `optimize_P_Q_linear_error` mit den zusätzlichen Eingabeparametern $E_P = E_Q = \varepsilon$ auf.

Dieses gibt nach 9 + 21 Iterationsschritten die Lösung

$$\Delta P_{\text{lin}}^\varepsilon := \begin{pmatrix} -1.19082 \cdot 10^{-9} \\ -2.37195 \cdot 10^{-11} \\ -0.0234312 \\ -1.89483 \cdot 10^{-11} \end{pmatrix}, \quad \Delta Q_{\text{lin}}^\varepsilon := \begin{pmatrix} 0.0229664 \\ 0.0143944 \\ 0.00781855 \\ -0.0098846 \end{pmatrix}.$$

zurück. Vergleicht man nun den Zielfunktionswert davon mit jenem der Lösung von `optimize_Q_linear`, dann ergibt sich

$$\|\Delta P_{\text{lin}}\|_1 = 0.0234218 < 0.0234312 = \|\Delta P_{\text{lin}}^\varepsilon\|_1.$$

Wählt man allerdings A_P^ε als die Matrix, die man erhält wenn man in A_P von der dritten Komponente in der dritten Zeile ε subtrahiert, dann gilt:

$$\begin{aligned} \max(U + A_P^\varepsilon \Delta P_{\text{lin}} + A_Q \Delta Q_{\text{lin}}) - \min(U + A_P^\varepsilon \Delta P_{\text{lin}} + A_Q \Delta Q_{\text{lin}}) &= 0.0698023 > 0.0698, \\ \max(U + A_P^\varepsilon \Delta P_{\text{lin}}^\varepsilon + A_Q \Delta Q_{\text{lin}}^\varepsilon) - \min(U + A_P^\varepsilon \Delta P_{\text{lin}}^\varepsilon + A_Q \Delta Q_{\text{lin}}^\varepsilon) &= 0.069796 \leq 0.0698. \end{aligned}$$

Mit derselben Vorgehensweise für die euklidische Norm in der Zielfunktion, also für `optimize_P_Q_quad_error` mit den zusätzlichen Eingabeparametern $E_P = E_Q = \varepsilon$, erhält man nach 9 + 26 Iterationen die Lösung

$$\Delta P_{\text{quad}}^\varepsilon := \begin{pmatrix} -0.0117058 \\ -6.8126 \cdot 10^{-9} \\ -0.0120354 \\ -5.32024 \cdot 10^{-9} \end{pmatrix}, \quad \Delta Q_{\text{quad}}^\varepsilon := \begin{pmatrix} 0.0162825 \\ 0.0143941 \\ -0.0016766 \\ 0.000774135 \end{pmatrix}$$

und stellt wieder fest, dass zwar für den Wert der Zielfunktion

$$\|\Delta P_{\text{quad}}\|_2 = 0.0167869 < 0.0167892 = \|\Delta P_{\text{quad}}^\varepsilon\|_2$$

gilt, dafür aber $(\Delta P_{\text{quad}}^\varepsilon, \Delta Q_{\text{quad}}^\varepsilon)$ im Gegensatz zu $(\Delta P_{\text{quad}}, \Delta Q_{\text{quad}})$ für dieselbe Matrix A_P^ε wie oben immer noch zulässig ist:

$$\begin{aligned} \max(U + A_P^\varepsilon \Delta P_{\text{quad}} + A_Q \Delta Q_{\text{quad}}) - \min(U + A_P^\varepsilon \Delta P_{\text{quad}} + A_Q \Delta Q_{\text{quad}}) &= 0.0698012 > 0.0698, \\ \max(U + A_P^\varepsilon \Delta P_{\text{quad}}^\varepsilon + A_Q \Delta Q_{\text{quad}}^\varepsilon) - \min(U + A_P^\varepsilon \Delta P_{\text{quad}}^\varepsilon + A_Q \Delta Q_{\text{quad}}^\varepsilon) &= 0.0697993 \leq 0.0698. \end{aligned}$$

Wie schon bei dem Beispiel zur Steuerung der Blindleistung gilt auch hier, dass die gefundenen Lösungen von `optimize_P_Q_linear_error` und `optimize_P_Q_quad_error` für $E_P =$

$$E_Q = 0,$$

$$\begin{aligned} \Delta P_{\text{lin}}^0 &= \begin{pmatrix} -7.61095 \cdot 10^{-10} \\ -1.49815 \cdot 10^{-11} \\ -0.0234218 \\ -1.18202 \cdot 10^{-11} \end{pmatrix}, \quad \Delta Q_{\text{lin}}^0 = \begin{pmatrix} 0.0229626 \\ 0.0143944 \\ 0.00780081 \\ -0.00987269 \end{pmatrix}, \\ \Delta P_{\text{quad}}^0 &= \begin{pmatrix} -0.0117053 \\ -6.22351 \cdot 10^{-9} \\ -0.0120327 \\ -4.15215 \cdot 10^{-9} \end{pmatrix}, \quad \Delta Q_{\text{quad}}^0 = \begin{pmatrix} 0.0162548 \\ 0.0143942 \\ -0.00246797 \\ 0.00122166 \end{pmatrix}, \end{aligned}$$

zwar nicht exakt mit $(\Delta P_{\text{lin}}, \Delta Q_{\text{lin}})$ beziehungsweise $(\Delta P_{\text{quad}}, \Delta Q_{\text{quad}})$ übereinstimmen, ihre Differenz aber durch die gewählte Genauigkeit, beziehungsweise für `optimize_P_Q_quad_error` durch die Wurzel davon, beschränkt ist:

$$\begin{aligned} \Delta P_{\text{lin}}^0 - \Delta P_{\text{lin}} &= \begin{pmatrix} 1.03585 \cdot 10^{-10} \\ 2.75017 \cdot 10^{-12} \\ -7.4222 \cdot 10^{-11} \\ 2.39812 \cdot 10^{-12} \end{pmatrix}, \quad \Delta Q_{\text{lin}}^0 - \Delta Q_{\text{lin}} = \begin{pmatrix} 5.07055 \cdot 10^{-10} \\ 1.95377 \cdot 10^{-9} \\ -4.66111 \cdot 10^{-9} \\ 4.58066 \cdot 10^{-9} \end{pmatrix}, \\ \Delta P_{\text{quad}}^0 - \Delta P_{\text{quad}} &= \begin{pmatrix} 1.95018 \cdot 10^{-9} \\ 1.76691 \cdot 10^{-9} \\ 1.18298 \cdot 10^{-9} \\ 1.41336 \cdot 10^{-9} \end{pmatrix}, \quad \Delta Q_{\text{quad}}^0 - \Delta Q_{\text{quad}} = \begin{pmatrix} 3.38856 \cdot 10^{-8} \\ 1.34828 \cdot 10^{-7} \\ -3.92198 \cdot 10^{-7} \\ 3.43756 \cdot 10^{-7} \end{pmatrix}, \end{aligned}$$

6.3. Numerische Simulationen

In diesem Kapitel wollen wir anhand von Simulationen mit zufällig generierten Testdatensätzen einen Eindruck davon gewinnen, wie sich der Aufwand der einzelnen Programme in Abhängigkeit von unterschiedlichen Parametern verhält. Um die Wahrscheinlichkeit zu erhöhen, dass für die Testdaten eine Lösung existiert, wählen wir die Größen ähnlich wie in den konkreten Beispieldaten aus den Abschnitten 6.1 und 6.2. Dazu gehen wir folgendermaßen vor: Zuerst generieren wir für vorgegebene $m, n \in \mathbb{N}$ zwei $(m \times n)$ -Matrizen A_P und A_Q , deren Komponenten zufällig aus dem Intervall $(0, 1)$ gewählt werden, und einen Vektor $P \in \mathbb{R}^n$ mit Eintragungen aus dem Intervall $(0, 2)$. $PF_{\min}$ und EVB setzen wir wie in den vorigen Abschnitten auf:

$$\begin{aligned} PF_{\min} &:= 0.9, \\ EVB &:= 0.0698. \end{aligned}$$

Damit die Nebenbedingungen von den Kraftwerken erfüllt sind, wählen wir $Q, S_{\max} \in \mathbb{R}^n$ mit

$$|Q_i| < \tan(\arccos(PF_{\min}))P_i, \quad i = 1, \dots, n,$$

$$S_{\max,i} > \sqrt{P_i^2 + Q_i^2}, \quad i = 1, \dots, n.$$

Zuletzt setzen wir $U^* \in \mathbb{R}^m$ erst

$$U^* := A_P P + A_Q Q$$

und manipulieren diesen Vektor dann so auf $U \in \mathbb{R}^m$, dass sich die einzelnen Komponenten mit einer Bandbreite von $\alpha > EVB$ um 1 verteilen:

$$U := 1 + \frac{\alpha(U^* - 1)}{\max(U^*) - \min(U^*)}.$$

Zusammengefasst ergibt das die MATLAB-Funktion `generatetestdata.m`:

Programm 6.1: `generatetestdata.m`

```
1 function [AP, AQ, P, Q, U, EVB, PFmin, Smax]=generatetestdata(m,n,alpha)
2 AQ=rand(m,n);
3 AP=rand(m,n);
4 P=2*rand(n,1);
5 PFmin=0.9;
6 EVB=0.0698;
7 Q=tan(acos(PFmin))*(2*rand(n,1)-1).*P;
8 Smax=sqrt(P.^2+Q.^2)+rand(n,1);
9 U=AP*P+AQ*Q;
10 U=1+alpha*(U-1)/(max(U)-min(U));
```

Ein Testlauf mit $m = 11$, $n = 4$, $\alpha = 0.1 > EVB$ und $\text{acc} = 10^{-8}$ ergab für 10000 auf diese Weise generierte Datensätze, dass die Funktion zur Spannungsregelung durch Steuerung der Blindleistung `optimize_Q_linear` in 62,36% der Fälle eine Lösung gefunden hat, während die Funktion zur Steuerung der Wirk- und Blindleistung `optimize_P_Q_linear` immer zu einem Ergebnis mit `exitflag = 0` gelangte:

```
>> for i=1:10000
[AP,AQ,P,Q,U,EVB,PFmin,Smax]=generatetestdata(11,4,0.1);
[ΔQ_Q(:,i),exitflag_Q(i),count_Q(i,:),time_Q(i)]=...
    optimize_Q_linear(AQ,Q,P,U,EVB,PFmin,Smax,10^-8);
[ΔP(:,i),ΔQ(:,i),exitflag(i),count(i,:),time(i)]=...
    optimize_P_Q_linear(AP,AQ,P,Q,U,EVB,PFmin,Smax,10^-8);
end
>> 1-mean(exitflag_Q)
ans =
    0.6236
>> 1-mean(exitflag)
ans =
    1
```

Dieses Ergebnis ist sehr stark abhängig von der Wahl der Verletzung der Spannungsbandbreitenbedingung. Setzt man zum Beispiel $\alpha = 0.07 > EVB$, so findet auch `optimize_Q_linear` für jeden Testdatensatz eine zulässige Lösung:

```
>> for i=1:10000
    [AP,AQ,P,Q,U,EVB,PFmin,Smax]=generatetestdata(11,4,0.07);
    [ΔQ_Q(:,i),exitflag_Q(i),count_Q(i,:),time_Q(i)]=...
        optimize_Q_linear(AQ,Q,P,U,EVB,PFmin,Smax,10^-8);
    [ΔP(:,i),ΔQ(:,i),exitflag(i),count(i,:),time(i)]=...
        optimize_P_Q_linear(AP,AQ,P,Q,U,EVB,PFmin,Smax,10^-8);
end
>> 1-mean(exitflag_Q)
ans =
    1
>> 1-mean(exitflag)
ans =
    1
```

Wählt man hingegen für α den Wert $0.14 > 2EVB$, dann wird durch die Steuerung der Blindleistung nur noch in 15,54% aller Fälle ein zulässiger Punkt gefunden:

```
>> for i=1:10000
    [AP,AQ,P,Q,U,EVB,PFmin,Smax]=generatetestdata(11,4,0.14);
    [ΔQ_Q(:,i),exitflag_Q(i),count_Q(i,:),time_Q(i)]=...
        optimize_Q_linear(AQ,Q,P,U,EVB,PFmin,Smax,10^-8);
    [ΔP(:,i),ΔQ(:,i),exitflag(i),count(i,:),time(i)]=...
        optimize_P_Q_linear(AP,AQ,P,Q,U,EVB,PFmin,Smax,10^-8);
end
>> 1-mean(exitflag_Q)
ans =
    0.1554
>> 1-mean(exitflag)
ans =
    1
```

Doch nun wollen wir uns damit beschäftigen wie sich die Veränderung einzelner Parameter auf das Verhalten der Programme auswirkt:

6.3.1. Wahl des Parameters μ

In Kapitel 2.4.2 wurde schon erwähnt, dass laut [BV04] Werte der Ordnung 10 eine geeignete Wahl für den Parameter μ seien. Wir wollen jetzt für $\mu = 10, 20, \dots, 100$ jeweils 100 Testdatensätze mit

$$m = 20, n = 10, \alpha = 0.07$$

generieren und dazu mit `optimize_Q_linear` beziehungsweise `optimize_Q_quad` und der Genauigkeit $\text{acc} = 10^{-8}$ nach Lösungen suchen. Anschließend vergleichen wir die Mittelwerte der dafür benötigten Iterationsschritte miteinander. Das Resultat ist in Abbildung 6.1 dargestellt. Wenn wir nun mit derselben Vorgehensweise `optimize_Q_linear_error` und `optimize_Q_quad_`

error mit dem zusätzlichem Eingabeparameter $E_Q = 10^{-4}$ nach Lösungen suchen lassen, so ergibt sich ein ähnliches Verhalten, wie man in Abbildung 6.2 sehen kann.

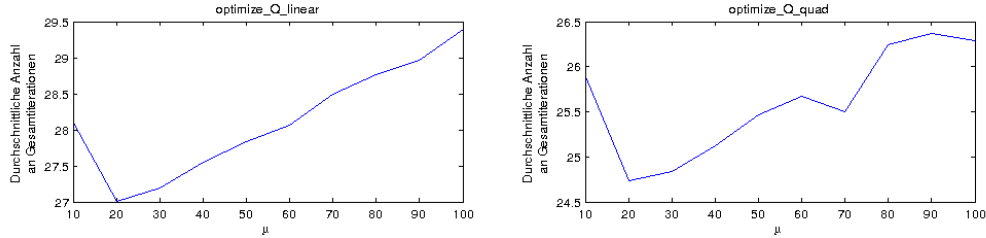


Abbildung 6.1.: Durchschnittliche Anzahl an Iterationen in `optimize_Q_linear` und `optimize_Q_quad` in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$ und $\alpha = 0.07$

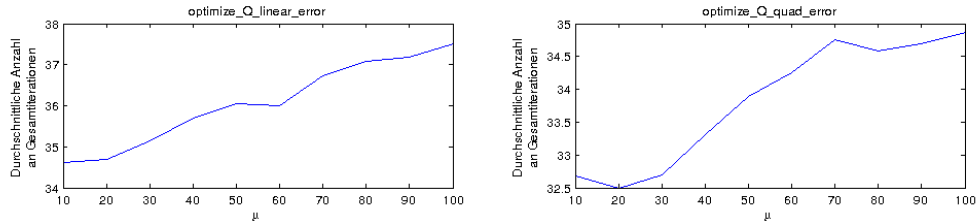


Abbildung 6.2.: Durchschnittliche Anzahl an Iterationen in `optimize_Q_linear_error` und `optimize_Q_quad_error` in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

Um dasselbe jetzt auch noch für die Spannungsregelung durch Steuerung der Wirk- und Blindleistung zu testen, wählen wir wie vorher $m = 20$ und $n = 10$. Allerdings setzen wir nun $\alpha = 0.14$, damit nicht jedesmal eine Lösung mit $\Delta P \approx 0$ gefunden wird. Für die Genauigkeit und den Fehler verwenden wir wieder die Werte $\text{acc} = 10^{-8}$ und $E_P = E_Q = 10^{-4}$ und betrachten diesmal die Programme `optimize_P_Q_linear`, `optimize_P_Q_quad`, `optimize_P_Q_linear_error` und `optimize_P_Q_quad_error`. Die Abbildungen 6.3 und 6.4 zeigen die Resultate.

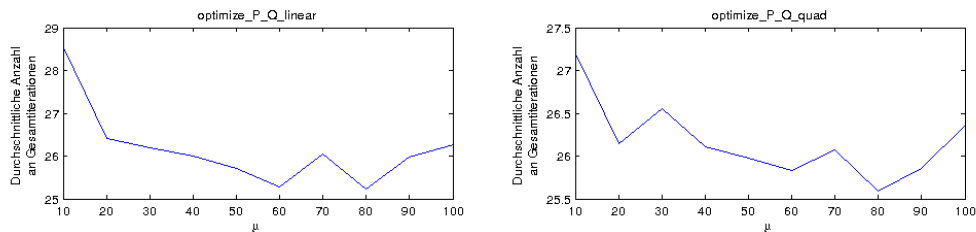


Abbildung 6.3.: Durchschnittliche Anzahl an Iterationen in `optimize_P_Q_linear` und `optimize_P_Q_quad` in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$ und $\alpha = 0.14$

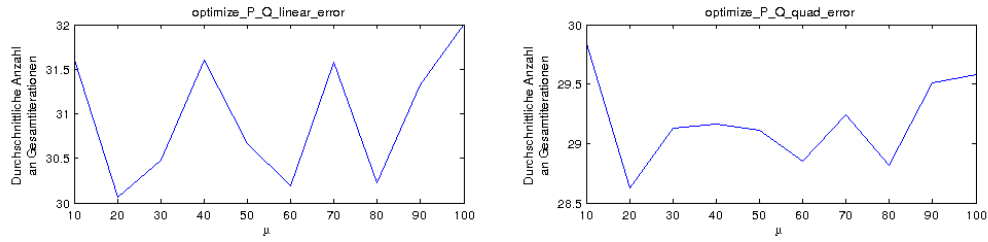


Abbildung 6.4.: Durchschnittliche Anzahl an Iterationen in `optimize_P_Q_linear_error` und `optimize_P_Q_quad_error` in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

Zusammenfassend lässt sich vermuten, dass die Wahl des Parameters μ aus dem Intervall $[10, 100]$ wohl keine große Auswirkung auf die Komplexität der einzelnen Funktionen hat. Für die Spannungsregelung durch Steuerung der Blindleistung scheint $\mu = 20$ und für die Steuerung der Wirk- und Blindleistung $\mu = 60$ oder $\mu = 80$ ein geeigneter Wert zu sein.

6.3.2. Wahl der Genauigkeit

In diesem Kapitel wollen wir beobachten, wie sich die Wahl der Genauigkeit auf die Anzahl an benötigten Iterationen auswirkt. Dazu erstellen wir für $\text{acc} = 10^{-3}, 10^{-4}, \dots, 10^{-10}$ jeweils 100 Testdatensätze mit den Eingabeparametern

$$m = 20, n = 10, \alpha = 0.07$$

und betrachten dann die Mittelwerte der Schritte, die `optimize_Q_linear`, `optimize_Q_quad` beziehungsweise `optimize_Q_linear_error` und `optimize_Q_quad_error` mit $E_Q = 10^{-4}$ zum Auffinden einer Lösung benötigen. Das Ergebnis ist in den Abbildungen 6.5 bis 6.8 zu sehen.

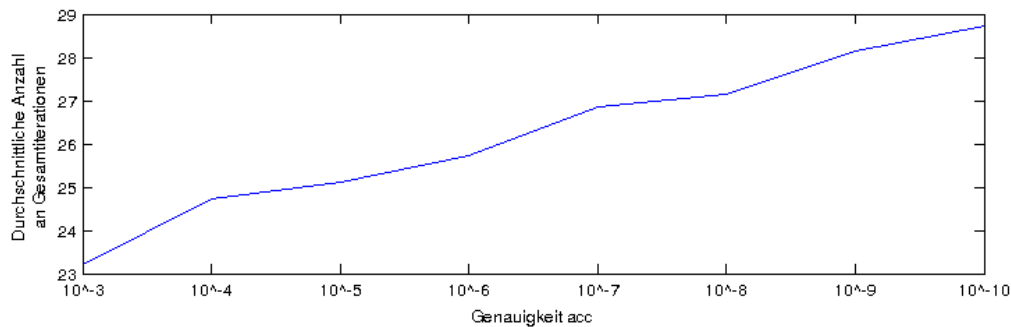


Abbildung 6.5.: Durchschnittliche Anzahl an Iterationen in `optimize_Q_linear` in Abhängigkeit von der gewählten Genauigkeit acc für je 100 Testdatensätze mit $m = 20$, $n = 10$ und $\alpha = 0.07$

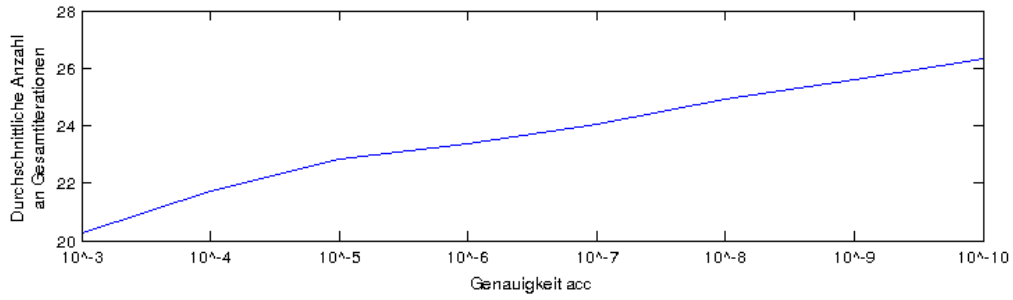


Abbildung 6.6.: Durchschnittliche Anzahl an Iterationen in `optimize_Q_quad` in Abhängigkeit von der gewählten Genauigkeit `acc` für je 100 Testdatensätze mit $m = 20$, $n = 10$ und $\alpha = 0.07$

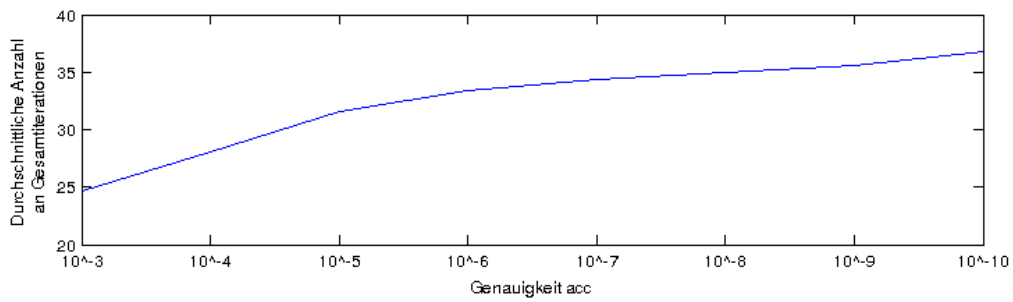


Abbildung 6.7.: Durchschnittliche Anzahl an Iterationen in `optimize_Q_linear_error` in Abhängigkeit von der gewählten Genauigkeit `acc` für je 100 Testdatensätze mit $m = 20$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

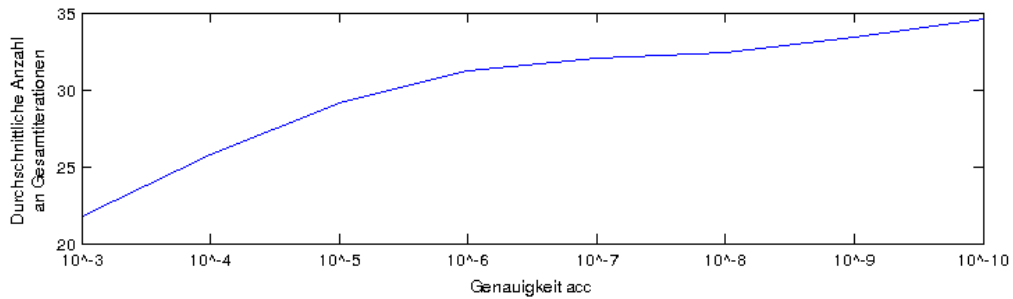


Abbildung 6.8.: Durchschnittliche Anzahl an Iterationen in `optimize_Q_quad_error` in Abhängigkeit von der gewählten Genauigkeit `acc` für je 100 Testdatensätze mit $m = 20$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

Für die Steuerung der Blindleistung und der Wirkleistung wählen wir als Input für `generatetest-data`

$$m = 20, n = 10, \alpha = 0.14$$

und betrachten wiederum für $\text{acc} = 10^{-3}, 10^{-4}, \dots, 10^{-10}$ die durchschnittliche Anzahl an Iterationsschritten, die `optimize_P_Q_linear`, `optimize_P_Q_quad` beziehungsweise `optimize_P_Q_linear_error` und `optimize_P_Q_quad_error` mit $E_P = E_Q = 10^{-4}$ zum Auffinden einer Lösung für je 100 Datensätze benötigen. Die Abbildungen 6.9 bis 6.12 zeigen das

hierbei auftretende Verhalten.

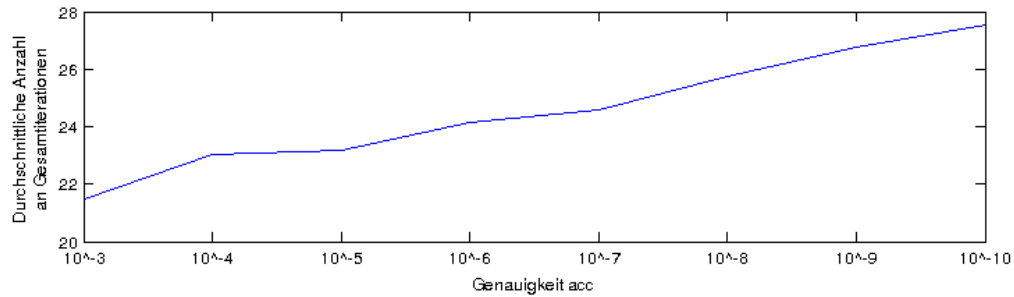


Abbildung 6.9.: Durchschnittliche Anzahl an Iterationen in `optimize_P_Q_linear` in Abhängigkeit von der gewählten Genauigkeit `acc` für je 100 Testdatensätze mit $m = 20$, $n = 10$ und $\alpha = 0.14$

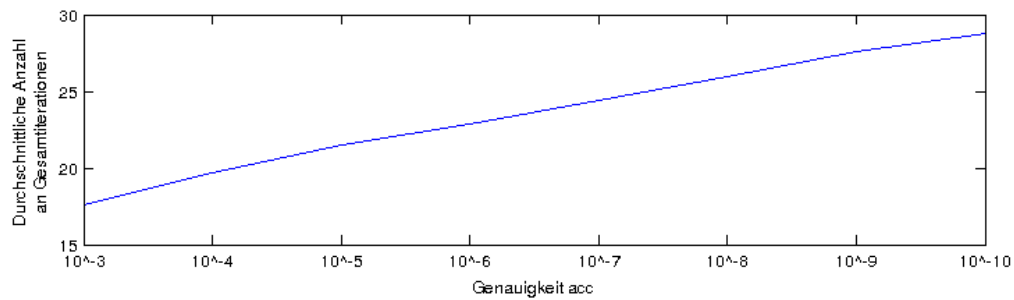


Abbildung 6.10.: Durchschnittliche Anzahl an Iterationen in `optimize_P_Q_quad` in Abhängigkeit von der gewählten Genauigkeit `acc` für je 100 Testdatensätze mit $m = 20$, $n = 10$ und $\alpha = 0.14$

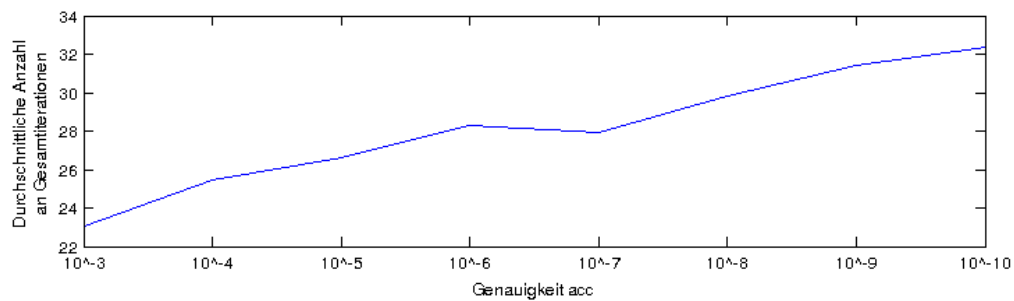


Abbildung 6.11.: Durchschnittliche Anzahl an Iterationen in `optimize_P_Q_linear_error` in Abhängigkeit von der gewählten Genauigkeit `acc` für je 100 Testdatensätze mit $m = 20$, $n = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

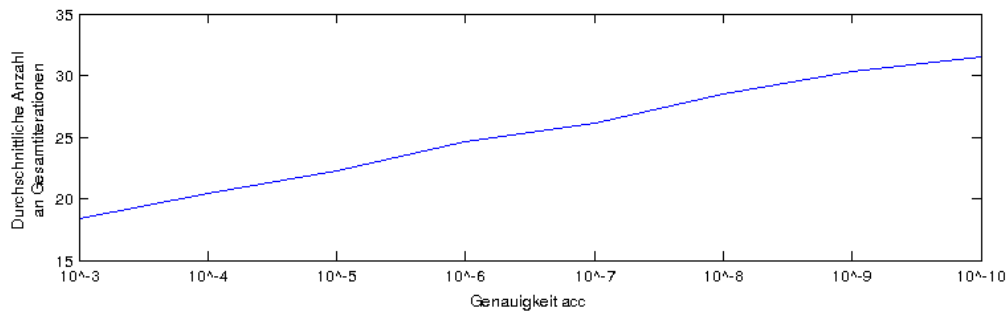


Abbildung 6.12.: Durchschnittliche Anzahl an Iterationen in `optimize_P_Q_quad_error` in Abhängigkeit von der gewählten Genauigkeit `acc` für je 100 Testdatensätze mit $m = 20, n = 10, E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

Anhand dieser Beispiele und Grafiken lässt sich also der Eindruck gewinnen, dass der Aufwand der Programme für $\text{acc} = 10^{-j}$ ungefähr linear von j abhängt. Für Werte mit $\text{acc} < 10^{-10}$ kann es mitunter vorkommen, dass gewisse Programme für bestimmte Daten sehr viele Iterationen benötigen. Das liegt daran, dass die Matrix $\tilde{H}$ aus (2.20), die in jedem Schritt invertiert werden muss, zwar für jeden inneren Punkt positiv definit ist, wie für die gegenständlichen Probleme gezeigt wurde, aber beliebig nahe an eine singuläre Matrix herankommen kann. Das führt wiederum dazu, dass die Berechnung der Suchrichtung sehr ungenau wird. Man könnte sich hier zum Beispiel mit einem modifizierten unvollständigen Cholesky-Verfahren, wie es in [Wri96] vorgestellt wird, Abhilfe verschaffen. Für die Zwecke dieser Arbeit ist aber eine derart hohe Genauigkeit nicht vonnöten, da das Hauptaugenmerk in den Problemen auf der Reduktion der Spannungsbandbreite mit möglichst geringem Aufwand liegt.

6.3.3. Abhängigkeit von der Größe der Dimensionen

Nun wollen wir uns damit befassen, wie sich eine Vergrößerung der Dimensionen $m, n \in \mathbb{N}$ auf die benötigte Anzahl an Iterationen beziehungsweise auf die benötigte Zeit zum Auffinden einer Lösung auswirkt. Natürlich sind die Resultate für die Berechnungsdauer der Programme abhängig von der verwendeten Hardware. Die numerischen Tests, deren Resultate hier vorgestellt werden, wurden auf einem Intel® Core™ i7-950 Prozessor durchgeführt.

Zunächst betrachten wir für festes $m = 20$ und $n = 5, 10, \dots, 50$ die durchschnittliche Iterationsanzahl und Zeit für je 100 zufällig generierte Testdaten mit $\alpha = 0.07$ für die Steuerung der Blindleistung und $\alpha = 0.14$ für die Steuerung der Wirk- und Blindleistung. Die Resultate hierfür sind in den Abbildungen 6.13 bis 6.20 zu sehen.

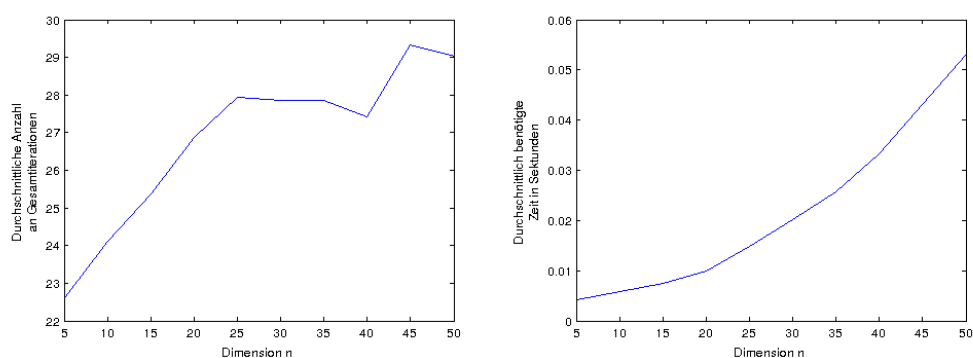


Abbildung 6.13.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_linear` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$ und $\alpha = 0.07$

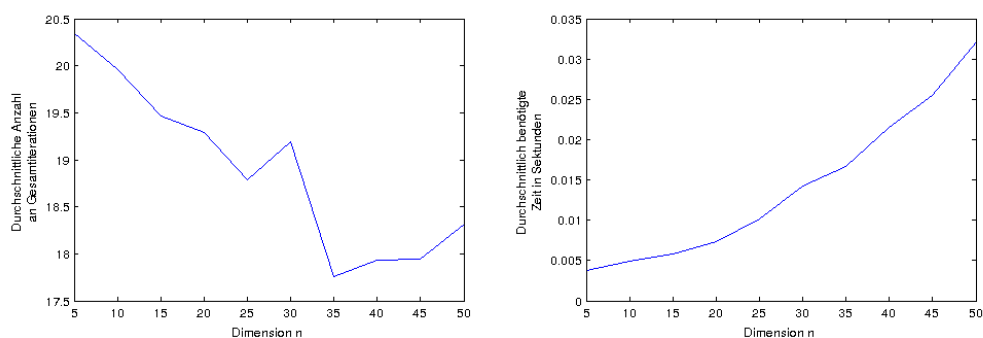


Abbildung 6.14.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_quad` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$ und $\alpha = 0.07$

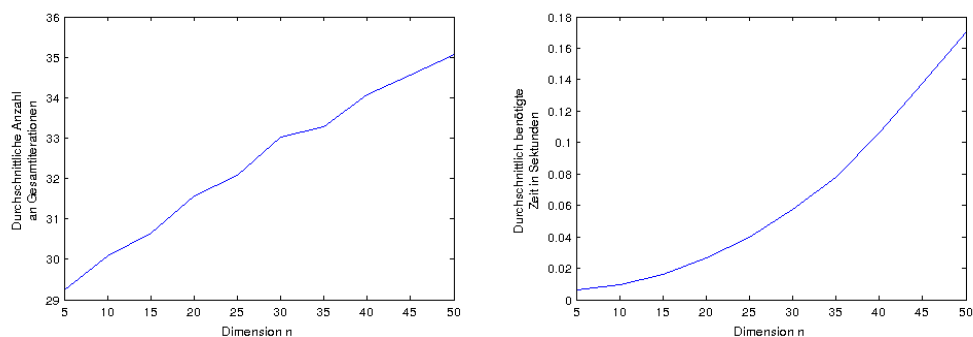


Abbildung 6.15.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_linear_error` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

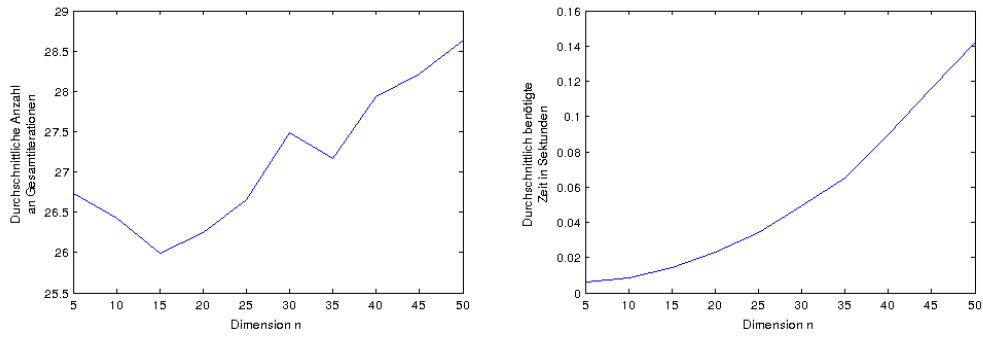


Abbildung 6.16.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_quad_error` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

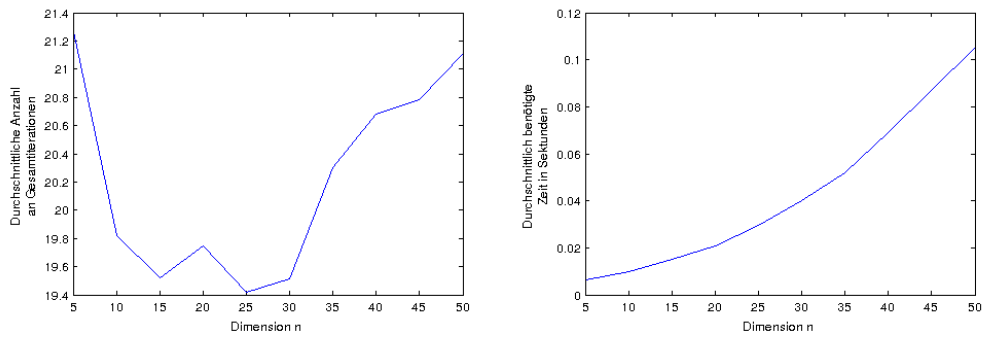


Abbildung 6.17.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_linear` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$ und $\alpha = 0.14$

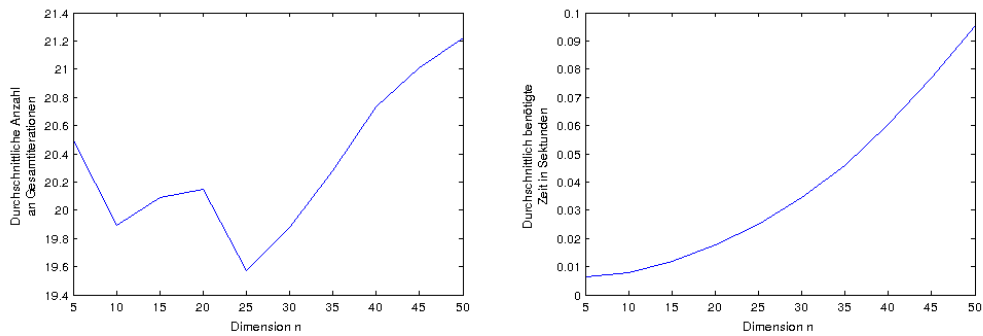


Abbildung 6.18.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_quad` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$ und $\alpha = 0.14$

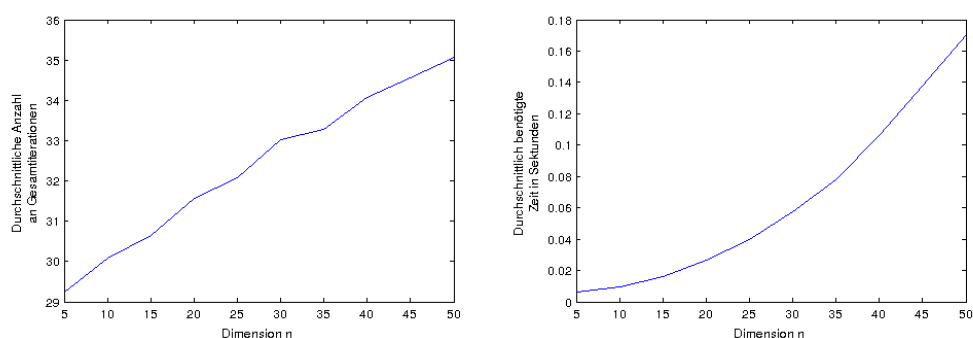


Abbildung 6.19.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_linear_error` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

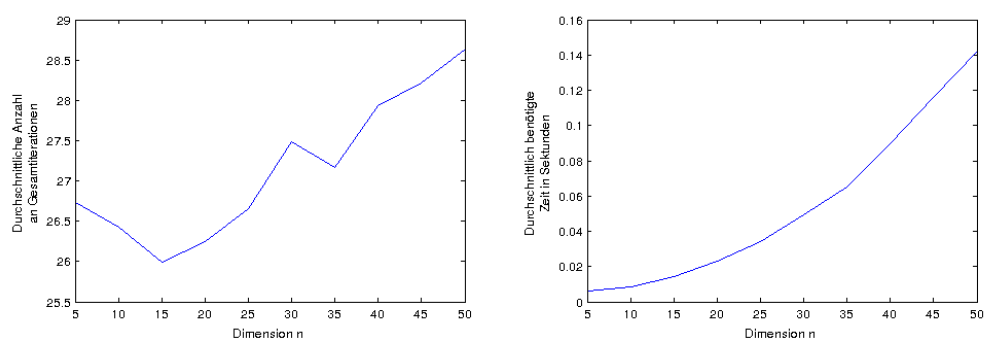


Abbildung 6.20.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_quad_error` in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $m = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

Nun wollen wir für festes $n = 10$ den Einfluss von m auf den Aufwand der Programme beobachten. Dazu rufen wir für $m = 10, 20, \dots, 100$ wieder jeweils 100-mal `generatetestdata` mit $\alpha = 0.07$ für die Steuerung der Blindleistung beziehungsweise $\alpha = 0.14$ für die Steuerung der Wirk- und Blindleistung auf und betrachten die Durchschnittswerte der dafür benötigten Iterationsanzahl und Berechnungszeit. In den Abbildungen 6.21 bis 6.28 sind die Ergebnisse der einzelnen Programme dazu grafisch dargestellt.

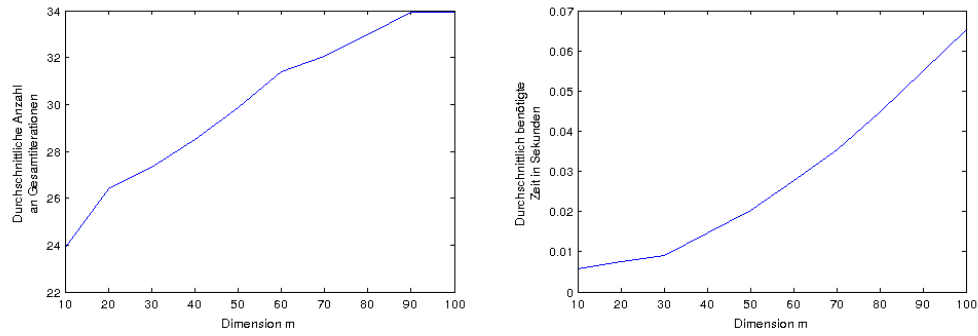


Abbildung 6.21.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_linear` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$ und $\alpha = 0.07$

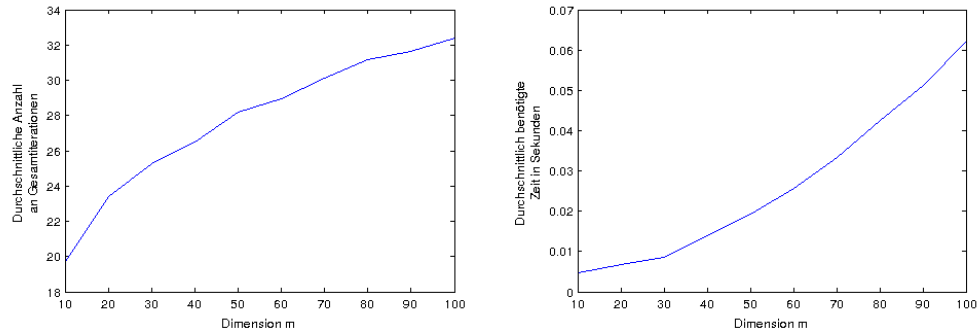


Abbildung 6.22.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_quad` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$ und $\alpha = 0.07$

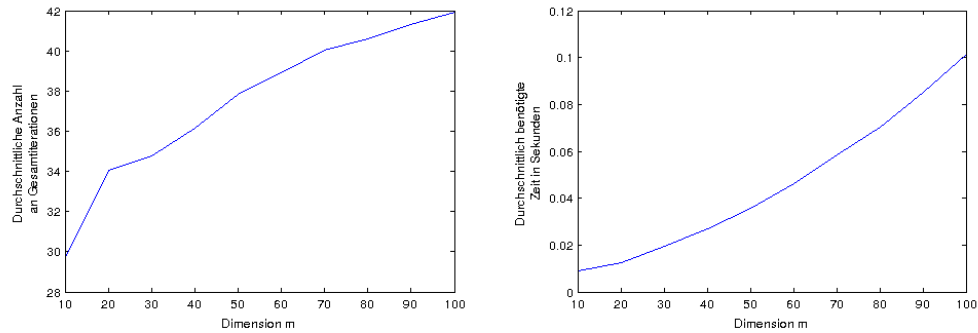


Abbildung 6.23.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_linear_error` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

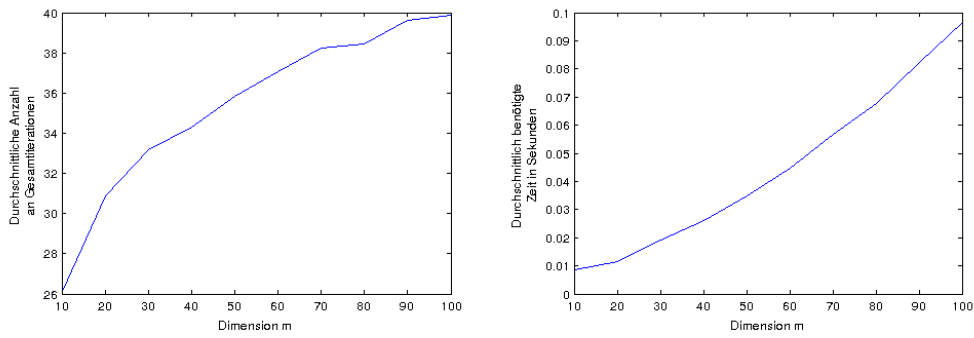


Abbildung 6.24.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_quad_error` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

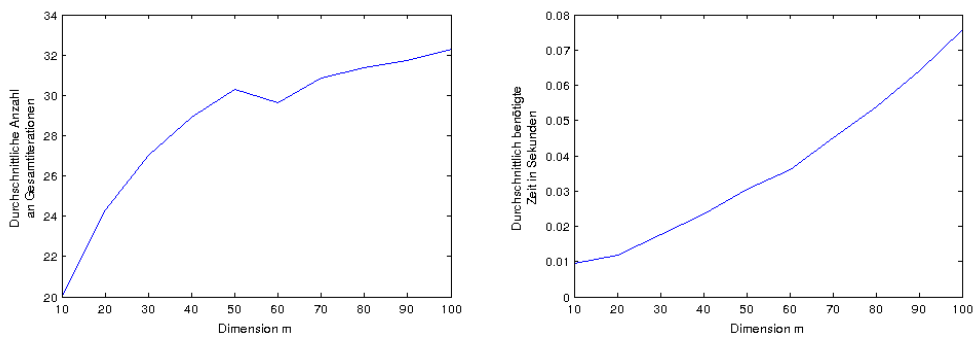


Abbildung 6.25.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_linear` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$ und $\alpha = 0.14$

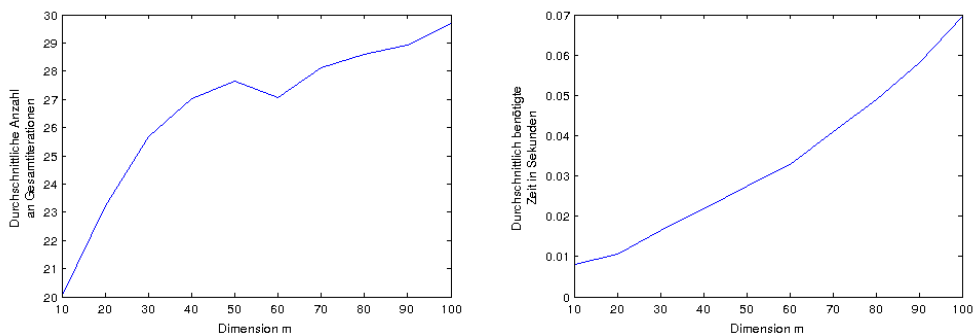


Abbildung 6.26.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_quad` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$ und $\alpha = 0.14$

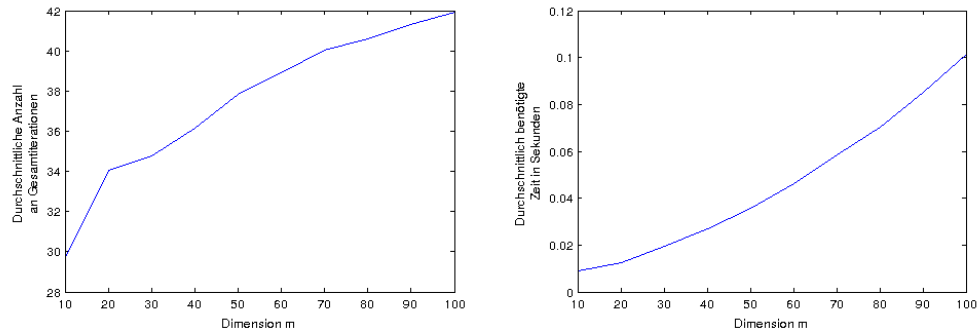


Abbildung 6.27.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_linear_error` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

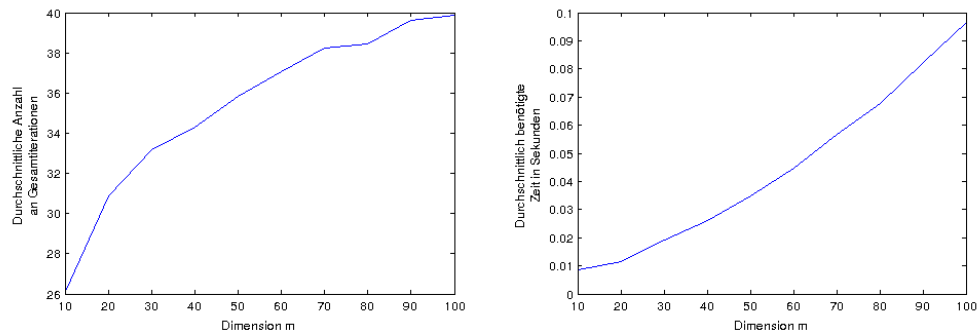


Abbildung 6.28.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_quad_error` in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

Zuletzt wird noch der Anstieg des Aufwands betrachtet, wenn sowohl n als auch m wachsen. Wir generieren also für $j = 1, \dots, 10$ je 100 Testdatensätze mit

$$m = 10j, \quad n = 5j$$

und $\alpha = 0.07$ für die Steuerung der Blindleistung beziehungsweise $\alpha = 0.14$ für die Steuerung der Wirk- und Blindleistung. In den Abbildungen 6.29 bis 6.36 ist wiederum die mittlere Iterationsanzahl und Rechendauer der Programme zu sehen.

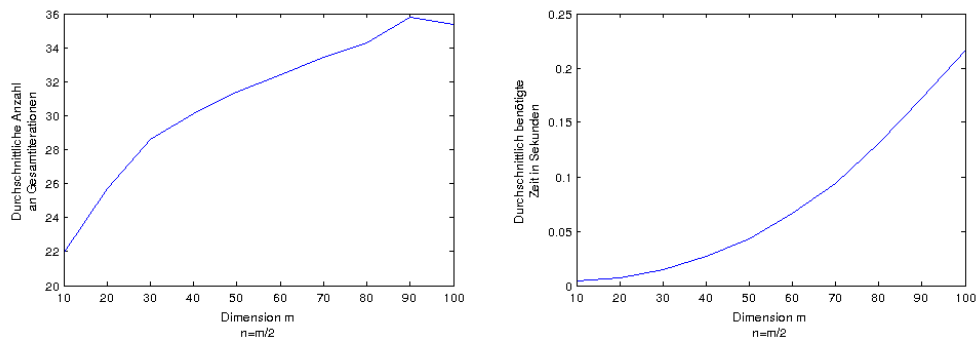


Abbildung 6.29.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_linear` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.07$

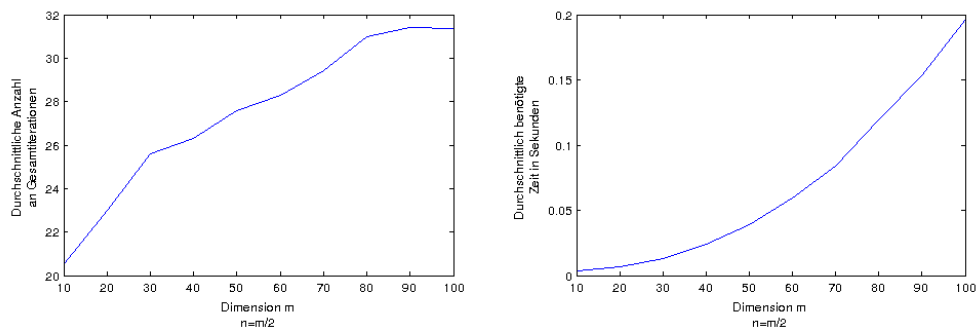


Abbildung 6.30.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_quad` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.07$

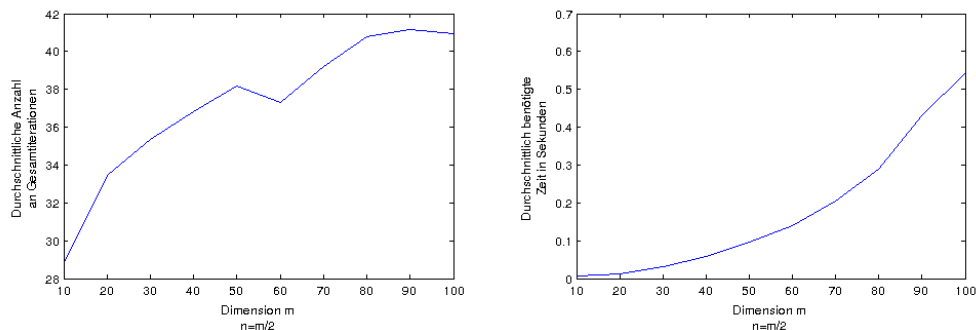


Abbildung 6.31.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_linear_error` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

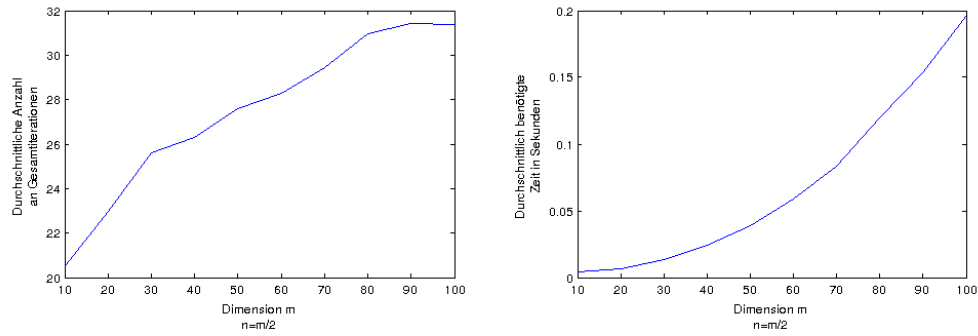


Abbildung 6.32.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_Q_quad_error` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

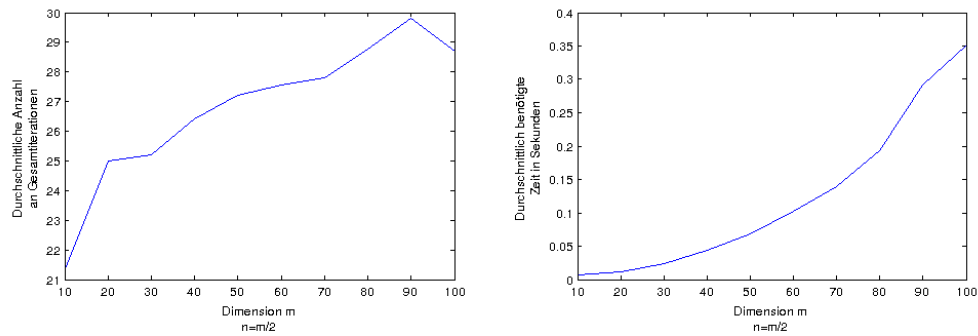


Abbildung 6.33.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_linear` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.14$

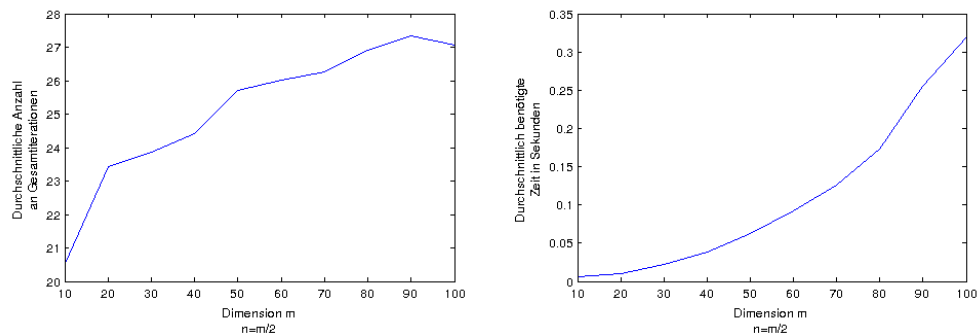


Abbildung 6.34.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_quad` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.14$

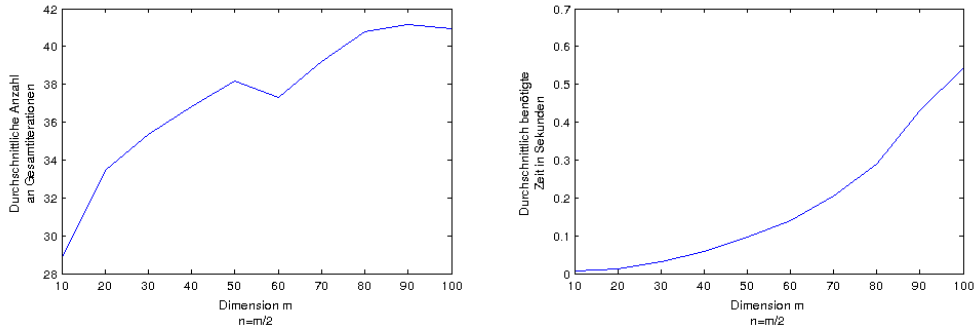


Abbildung 6.35.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_linear_error` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

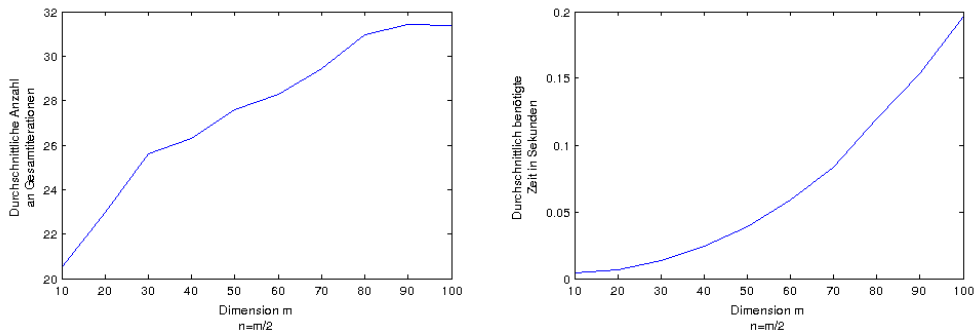


Abbildung 6.36.: Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in `optimize_P_Q_quad_error` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

In diesen Grafiken lässt sich vor allem der Einfluss von m und n auf die benötigte Zeit ablesen. Wenn auch die Iterationsanzahl nicht besonders stark ansteigt, so wächst doch die Größe des in jedem Schritt auftretenden linearen Gleichungssystems

$$\tilde{H}\Delta x = -r_{\text{dual}} - Df(x)^T \text{diag}(f(x))^{-1} r_{\text{cent}}$$

sehr deutlich mit m und n und dessen Lösung beansprucht daher für höhere Dimensionen wesentlich mehr Zeit. Es ist aber in den Grafiken zu sehen, dass die Laufzeit der Programme für

$$m \leq 50, n \leq 25$$

immer unter 0.1 Sekunden beträgt. In den meisten Fällen ist es sogar deutlich weniger. Da die regionalen elektrischen Netzwerke, in denen diese Programme angewendet werden sollen, meist sehr klein sind und ungefähr die Anzahl an kritischen Knoten und regelbaren Kraftwerken haben wie in den Abschnitten 6.1 und 6.2, eignen sich die in dieser Arbeit vorgestellten MATLAB-Funktionen gut zum Bestimmen einer Lösung der zugehörigen Optimierungsprobleme. Möchte man dennoch die Performance der Pro-

gramme auch für höhere Dimensionen verbessern, so könnte man zum Beispiel die Dünnbesetztheit¹ der Matrix $\tilde{H}$ ausnutzen. In den hier relevanten niedrigdimensionalen Problemen würde dies aber keinen Vorteil im Vergleich zum Standardverfahren zum Lösen von linearen Gleichungssystemen von MATLAB liefern.

6.3.4. Mehraufwand durch Berücksichtigung eines Fehlers

Nun wollen wir noch kurz beobachten, wie sich die Laufzeit der Programme, die einen Fehler in der Linearisierung der Spannungsberechnung mitberücksichtigen, zur Laufzeit der anderen verhält. Dazu generieren wir wie im vorigen Kapitel für $j = 1, \dots, 10$ je 100 Testdatensätze mit

$$m = 10j, \quad n = 5j$$

und $\alpha = 0.07$ beziehungsweise $\alpha = 0.14$ und betrachten die Mittelwerte von

$$\frac{t_{\text{error}}}{t},$$

wobei t_{error} die Zeit ist, die eine Funktion mit Berücksichtigung eines Fehlers benötigt, und t die Zeit, die die entsprechende Funktion ohne Berücksichtigung des Fehlers braucht. In den Abbildungen 6.37 bis 6.40 ist dies dargestellt.

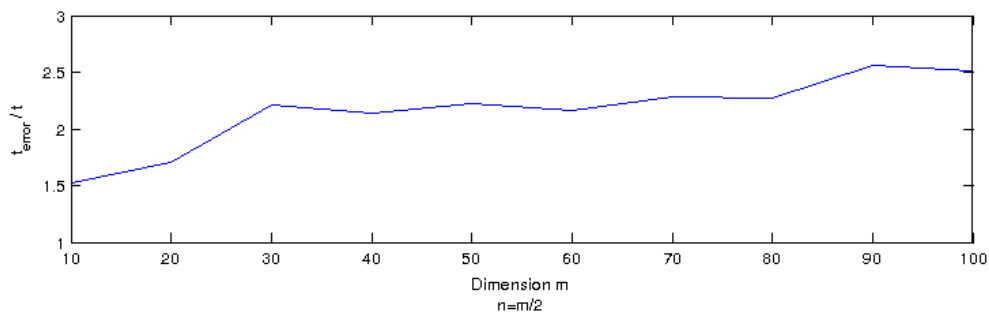


Abbildung 6.37.: Quotient der benötigten Zeit `optimize_Q_linear_error` und jener von `optimize_Q_linear` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

¹Siehe zum Beispiel [Dav06]

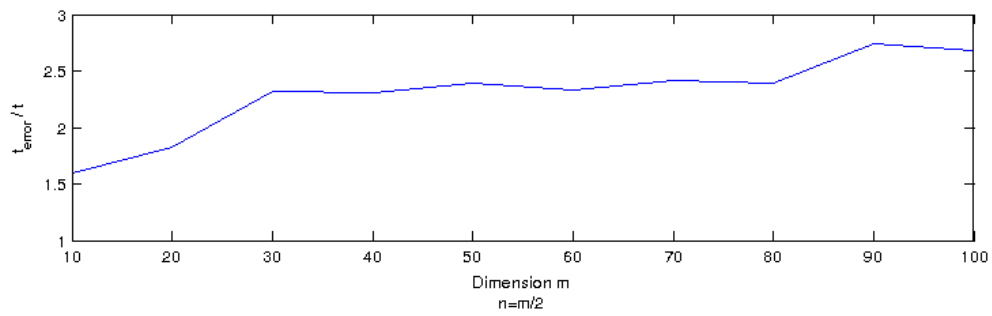


Abbildung 6.38.: Quotient der benötigten Zeit von `optimize_Q_quad_error` und jener von `optimize_Q_quad` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$

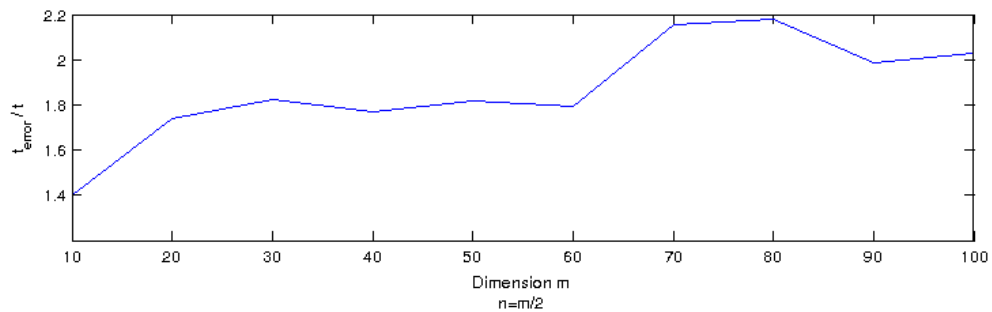


Abbildung 6.39.: Quotient der benötigten Zeit von `optimize_P_Q_linear_error` und jener von `optimize_P_Q_linear` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

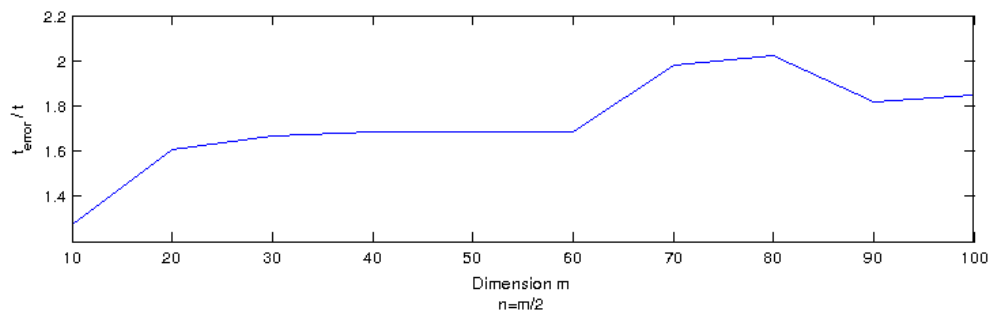


Abbildung 6.40.: Quotient der benötigten Zeit von `optimize_P_Q_quad_error` und jener von `optimize_P_Q_quad` in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$

Wir sehen also, dass man zur Lösung der Probleme (1.4) und (1.5) unter Berücksichtigung eines Fehlers etwa das Doppelte bis 2.5-fache der Zeit benötigt, die das Lösen dieser Probleme ohne Berücksichtigung des Fehlers beansprucht.

7. Conclusio

Abschließend lässt sich festhalten, dass sich die in dieser Arbeit vorgestellten Programme für eine nicht allzu große Anzahl an kritischen Knoten m und regelbaren Kraftwerken n , also in etwa

$$m \leq 50, n \leq 25,$$

und eine Wahl der Genauigkeit `acc` mit

$$10^{-10} \leq \text{acc} \leq 10^{-3}$$

sehr gut eignen, um die Probleme (1.4) und (1.5) zu lösen. Es ist jedoch empfehlenswert, den Genauigkeitsparameter nicht kleiner als benötigt zu wählen, da sich die zu invertierende Matrix $\tilde{H}$ für zu kleine Werte immer mehr einer singulären Matrix annähert, was dann ungenaue Berechnungen zur Folge hat und weiters zu einer verhältnismäßig sehr langen Laufzeit der Programme führen kann. Hier ist noch anzumerken, dass dies hauptsächlich `optimize_P_Q_linear` und `optimize_P_Q_linear_error` betrifft. Die Programme, die die euklidische Norm in der Zielfunktion minimieren, sind deutlich weniger anfällig für dieses Verhalten. Wie schon in Abschnitt 6.3.2 erwähnt könnte man diesem Problem entgegenwirken, indem man zur Lösung des Gleichungssystems

$$\tilde{H}\Delta x = -r_{\text{dual}} - Df(x)^T \text{diag}(f(x))^{-1} r_{\text{cent}}$$

ein modifiziertes unvollständiges Cholesky-Verfahren verwendet, wie es in [Wri96] vorgestellt wird. Allerdings ist es für größere Werte für `acc` effizienter, die Standardmethode von MATLAB zur Lösung von linearen Gleichungssystemen zu verwenden. Da es für die hier betrachteten Optimierungsprobleme wesentlich wichtiger ist, möglichst schnell einen zulässigen Punkt zu finden, als mit sehr hoher Genauigkeit ein Minimum zu bestimmen, fiel die Wahl auf das Standardverfahren von MATLAB.

Die hier vorgestellten Programme sollen in kleinen regionalen Netzwerken Anwendung finden. In diesen haben die zu lösenden Probleme ungefähr die Dimensionen wie in den konkreten Beispielen aus den Abschnitten 6.1 und 6.2, also $m = 11$ und $n = 4$. Dafür eignen sich die vorgestellten MATLAB-Funktionen sehr gut. Möchte man die Verfahren auch für deutlich höherdimensionale Probleme effizienter machen, so könnte man zum Beispiel die Dünnbesetztheit der Matrix $\tilde{H}$ ausnutzen.

Weiters sei noch erwähnt, dass man bei einer Mitberücksichtigung eines Fehlers ungefähr mit dem doppelten Zeitaufwand rechnen muss und dass die Programme zur Minimierung der euklidischen Norm im Allgemeinen etwas schneller sind als jene zur Minimierung der 1-Norm, wie sich anhand der Grafiken in Abschnitt 6.3.3 erkennen lässt.

A. Programme

A.1. Steuerung der Blindleistung mit linearer Zielfunktion

Programm A.1: optimize_Q_linear.m

```
1 function [ΔQ,exitflag,count,time]= ...
2     optimize_Q_linear(AQ,Q,P,U,EVB,PFmin,Smax,acc)
3
4 %-----
5 % optimize_Q_linear is a function to solve the optimization problem
6 %
7 %     min ||Q+ΔQ||_1
8 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
9 %           |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
10 %           |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
11 %
12 % with respect to the unknown ΔQ, where ||.|| is the 1-norm:
13 % ||x||_1:=sum(|x(i)|)
14 %-----
15
16
17 %-----
18 % Input parameters:
19 %
20 % AQ ..... mxn-matrix to compute an estimate for the change of
21 %           voltage
22 % Q ..... current reactive power, n-dimensional vector
23 % P ..... current active power, n-dimensional vector
24 % U ..... current voltage, m-dimensional vector
25 % EVB ..... bound on diffusion of voltage vector U, real number
26 % PFmin ..... real number
27 % Smax ..... nominal apparent power, n-dimensional vector
28 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
29 %-----
30
31 %-----
32 % Output parameters:
33 %
34 % ΔQ ..... solution of the problem
35 % exitflag .... exitflag:
36 %               exitflag==1 --> no solution could be found, ΔQ is
37 %               not the solution
38 %               exitflag==0 --> ΔQ is the solution of the problem
39 %-----
40
41 tic
42 exitflag=0;
43
```

```

44 %-----
45 % transform the problem into a linear optimization problem with of
46 % the form
47 %
48 %      min  c'*x
49 %      s.t. A*x ≤ b
50
51 [m,n]=size(AQ);
52 v=min(P*tan(acos(PFmin)),sqrt(Smax.^2-P.^2));
53
54 A=[ eye(n)      , zeros(n)      , zeros(n,1) , zeros(n,1) ;
55     -eye(n)     , -eye(n) , zeros(n,1) , zeros(n,1) ;
56     -eye(n)     , eye(n)      , zeros(n,1) , zeros(n,1) ;
57     zeros(1,n), zeros(1,n) , 1          , -1          ;
58     zeros(m,n), AQ          , -ones(m,1) , zeros(m,1) ;
59     zeros(m,n), -AQ         , zeros(m,1) , ones(m,1)  ];
60
61 b=[ v; Q; -Q; EVB; -U; U];
62 %-----
63
64 [M,N]=size(A); % determine the dimensions of the problem
65 x=zeros(N,1); % choose a starting point
66
67 %-----
68 % Phase I:
69 % find a strictly feasible point x for the constraints A*x ≤ b by
70 % solving a linear optimization problem
71 %
72 %      min  z                      (phase1)
73 %      s.t. A*x-b ≤ z
74 %
75 % starting with z>max(A*x-b) and stopping as soon as z<0 is achieved
76
77 % determine the maximal violation of the constraints
78 fmax=max(A*x-b);
79 if fmax ≥ 0
80     % transform to prepare for (phase1)
81     x(N+1)=fmax+0.1;
82     A(:,N+1)=-ones(M,1);
83     c=(1:N+1==N+1)';
84     % run linear optimization algorithm for (phase1)
85     [x,count(1)]=solve_lp(c,A,b,x,acc,1);
86     if x(N+1) ≥ 0
87         % if no point (x,z) with z<0 could be found, no feasible
88         % point for the original problem can be found, so stop and
89         % set exitflag=1
90         ΔQ=x(n+1:2*n);
91         exitflag=1;
92         time=toc;
93         count(2)=0;
94         count(3)=count(1);
95         return
96     end
97     % (phase1) was successful, retransform the problem
98     x=x(1:N);
99     A=A(:,1:N);
100 end
101
102 %-----
103 % Phase II:

```

```

104 % If we have found a feasible point we can start the actual
105 % optimization
106 c=(1:N≤n)';
107 [x,count(2)]=solve_lp(c,A,b,x,acc,0);
108 count(3)=count(1)+count(2);
109 ΔQ=x(n+1:2*n);
110 time=toc;

```

Programm A.2: solve_lp.m

```

1 function [x,count]=solve_lp(c,A,b,x,acc,phaseI)
2 %-----
3 % The function solve_lp solves the linear optimization problem
4 %
5 %      min   c'*x
6 %      s.t.  A*x≤b
7 %
8 % with an interior point algorithm.
9 %-----
10
11 %-----
12 % Input Parameters
13 %
14 % c..... n-dimensional column vector describing the
15 %          objective funktion
16 % A..... mxn-matrix describing the constraints
17 % b..... m-dimensional column vector describing the
18 %          constraints
19 % x..... n-dimensional column vector, feasible starting
20 %          point for the optimization problem
21 % acc..... accuracy, real number (10^-6, 10^-8, ...)
22 % phaseI..... boolean number to decide, if the actual minimum
23 %              should be found (phaseI=0) or if the program
24 %              should be stopped, as soon as feasible point for a
25 %              reduced problem is found, i.e. as soon as the
26 %              objective function c'*x<0 (phaseI=1)
27 %-----
28
29 %-----
30 % Output Parameters
31 %
32 % x..... n-dimensional column vector, either the solution of
33 %          of the problem or a feasible point for a reduced
34 %          problem
35 %-----
36
37 count=0;
38
39 % set parameter mu
40 mu=20;
41 % determine the dimensions of the problem
42 [m,n]=size(A);
43 % set starting dual variable
44 lambda=ones(m,1);
45 % compute the value of the constraints
46 f=A*x-b;
47 % compute the surrogate duality gap eta and starting value for t
48 eta=-f'*lambda;

```

```

49 t=mu*m/eta;
50 % compute rdual and rcent in the KKT-conditions
51 rdual=c+A'*lambda;
52 rcent=-diag(lambda)*f-ones(m,1)/t;
53 while eta>acc || norm(rdual)>acc
54     count=count+1;
55     % determine the matrix describing the linear system for the
56     % newton step, solve that and determine Δx and deltlambda
57     H=A'*diag(-lambda./f)*A;
58     Δx=-H\ (rdual+A'*diag(1./f)*rcent);
59     Δlambda=-diag(lambda./f)*A*Δx+diag(1./f)*rcent;
60     % perform a line search and set the new values for x and lambda
61     % and compute the new values for rdual and constraints
62     [x, lambda, rdual, f]=linesearch_lp(x, lambda, Δx, Δlambda, ...
63         c, A, b, t, rdual, rcent);
64     % compute the new surrogate duality gap, t, and rcent
65     eta=-f'*lambda;
66     t=mu*m/eta;
67     rcent=-diag(lambda)*f-ones(m,1)/t;
68     % if in phase I: check if objective value z<0 and if so: stop
69     if phaseI && c'*x<-acc
70         break;
71     end
72 end

```

Programm A.3: linesearch_lp.m

```

1 function [x_new, lambda_new, rdual, f]= ...
2     linesearch_lp(x, lambda, Δx, Δlambda, c, A, b, t, rdual, rcent)
3
4 %-----
5 % linesearch_lp performs a backtracking linesearch for the search
6 % direction (Δx, Δlambda) calculated in solve_lp.
7 %-----
8
9 %-----
10 % Input Parameters:
11 %
12 % x.....current primal variable, n-dimensional vector
13 % lambda.....current dual variable, m-dimensional vector
14 % Δx.....search direction for x, n-dimensional vector
15 % Δlambda.....search direction for lambda, m-dimensional vector
16 % c.....n-dimensional vector describing the objective function
17 % A.....(mxn)-Matrix describing the linear constraints
18 % b.....m-dimensional vector describing the linear constraints
19 % t.....real number needed to calculate rcent
20 % rdual.....current value for rdual in the KKT-conditions,
21 %             n-dimensional vector
22 % rcent.....current value for rcent in the KKT-conditions,
23 %             m-dimensional vector
24 %-----
25
26 %-----
27 % Output Parameters:
28 %
29 % x_new.....new value for x, n-dimensional vector
30 % lambda_new..new value for lambda, m-dimensional vector
31 % rdual.....value for rdual at x_new, n-dimensional vector

```

```

32 % f.....value for the constraints at x_new, m-dimensional
33 %          vector
34 %-----
35
36 % set some parameters
37 beta=0.5;
38 alpha=0.05;
39 % determine the dimensions
40 m=size(A,1);
41 % determine the maximal value for the step length
42 y=ones(m+1,1);
43 for i=1:m
44     if Δlambda(i)<0
45         y(i)=-lambda(i)/Δlambda(i);
46     end
47 end
48 s=0.99*min(y);
49 % compute current norm of (rdual,rcent)
50 oldnorm=norm([rdual;rcent]);
51 % set new x and lambda
52 x_new=x+s*Δx;
53 lambda_new=lambda+s*Δlambda;
54 % compute the value of the constraints
55 f=A*x_new-b;
56 % compute the new values of rdual and rcent
57 rdual=c+A'*lambda_new;
58 rcent=-diag(lambda_new)*f-ones(m,1)/t;
59
60 while max(f)≥0 || norm([rdual;rcent])>oldnorm*(1-alpha*s)
61     % reduce step length s
62     s=beta*s;
63     % set new x and lambda
64     x_new=x+s*Δx;
65     lambda_new=lambda+s*Δlambda;
66     % compute the value of the constraints
67     f=A*x_new-b;
68     % compute the new values of rdual and rcent
69     rdual=c+A'*lambda_new;
70     rcent=-diag(lambda_new)*f-ones(m,1)/t;
71 end

```

A.2. Steuerung der Blindleistung mit quadratischer Zielfunktion

Programm A.4: optimize_Q_quad.m

```

1 function [ΔQ,exitflag,count,time]= ...
2     optimize_Q_quad(AQ,Q,P,U,EVB,PFmin,Smax,acc)
3
4 %-----
5 % optimize_Q is a function to solve the optimization problem
6 %
7 %     min  (||Q+ΔQ||_2)^2
8 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB

```

```

9 %          |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
10 %          |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
11 %
12 % with respect to the unknown ΔQ, where ||.||_2 is the 2-norm:
13 % ||x||_2:=sqrt(sum(|x(i)|^2))
14 %-----
15
16 %-----
17 % Input parameters:
18 %
19 % AQ ..... mxn-matrix to compute an estimate for the change of
20 %          voltage
21 % Q ..... current reactive power, n-dimensional vector
22 % P ..... current active power, n-dimensional vector
23 % U ..... current voltage, m-dimensional vector
24 % EVB ..... bound on diffusion of voltage vector U, real number
25 % PFmin ..... real number
26 % Smax ..... nominal apparent power, n-dimensional vector
27 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
28 %-----
29
30 %-----
31 % Output parameters:
32 %
33 % ΔQ ..... solution of the problem
34 % exitflag .... exitflag:
35 %          exitflag==1 --> no solution could be found, ΔQ is
36 %          not the solution
37 %          exitflag==0 --> ΔQ is the solution of the problem
38 %-----
39
40 tic
41 exitflag=0;
42
43 %-----
44 % transform the problem into a quadratic optimization problem with of
45 % the form
46 %
47 %      min  x'*J*x
48 %      s.t. A*x≤b
49
50 [m,n]=size(AQ);
51 v=min(P*tan(acos(PFmin)),sqrt(Smax.^2-P.^2));
52
53 A=[ eye(n)      , zeros(n)      , zeros(n,1) , zeros(n,1) ;
54     -eye(n)     , -eye(n)      , zeros(n,1) , zeros(n,1) ;
55     -eye(n)     , eye(n)       , zeros(n,1) , zeros(n,1) ;
56     zeros(1,n) , zeros(1,n)    , 1         , -1         ;
57     zeros(m,n) , AQ           , -ones(m,1) , zeros(m,1) ;
58     zeros(m,n) , -AQ          , zeros(m,1) , ones(m,1)  ];
59
60 b=[ v; Q; -Q; EVB; -U; U];
61 %-----
62
63 [M,N]=size(A); % determine the dimensions of the problem
64 x=zeros(N,1); % choose a starting point
65
66 %-----
67 % Phase I:
68 % find a strictly feasible point x for the constraints A*x≤b by

```

```

69 % solving a linear optimization problem
70 %
71 %      min    z                      (phase1)
72 %      s.t.  A*x-b ≤ z
73 %
74 % starting with z>max(A*x-b) and stopping as soon as z<0 is achieved
75
76 % determine the maximal violation of the constraints
77 fmax=max(A*x-b);
78 if fmax≥0
79     % transform to prepare for (phase1)
80     x(N+1)=fmax+0.1;
81     A(:,N+1)=-ones(M,1);
82     c=(1:N+1==N+1)';
83     % run linear optimization algorithm for (phase1)
84     [x,count(1)]=solve_lp(c,A,b,x,acc,1);
85     if x(N+1)≥0
86         % if no point (x,z) with z<0 could be found, no feasible
87         % point for the original problem can be found, so stop and
88         % set exitflag=1
89         ΔQ=x(n+1:2*n);
90         exitflag=1;
91         time=toc;
92         return
93     end
94     % (phase1) was successful, retransform the problem
95     x=x(1:N);
96     A=A(:,1:N);
97 end
98
99 %-----
100 % Phase II:
101 % If we have found a feasible point we can start the actual
102 % optimization
103 J=diag((1:N≤n));
104 [x,count(2)]=solve_qlp(J,A,b,x,acc);
105 count(3)=count(1)+count(2);
106 ΔQ=x(n+1:2*n);
107 time=toc;

```

Programm A.5: solve_qlp.m

```

1  function [x,count]=solve_qlp(J,A,b,x,acc)
2
3  %-----
4  % The function solve_lp solves the linear optimization problem
5  %
6  %      min    x'Jx
7  %      s.t.  Ax≤b
8  %
9  % with an interior point algorithm.
10 %-----
11
12 %-----
13 % Input Parameters
14 %
15 % J..... positive definite (nxn)-matrix describing the
16 %          ojective funktion

```

```

17 % A..... mxn-matrix describing the constraints
18 % b..... m-dimensional column vector describing the
19 %          constraints
20 % x..... n-dimensional column vector, feasible starting
21 %          point for the optimization problem
22 % acc..... accuracy, real number (10^-6, 10^-8, ...)
23 %-----
24
25 %-----
26 % Output Parameters
27 %
28 % x..... n-dimensional column vector, either the solution of
29 %          of the problem or a feasible point for a reduced
30 %          problem
31 %-----
32
33 count=0;
34 % set parameter mu
35 mu=20;
36 % determine the dimensions of the problem
37 [m,n]=size(A);
38 % set starting dual variable
39 lambda=ones(m,1);
40 % compute the value of the constraints
41 f=A*x-b;
42 % compute surrogate duality gap eta and starting value for t:
43 eta=-f'*lambda;
44 t=mu*m/eta;
45 % compute rdual and rcent in the KKT-conditions
46 rdual=2*J*x+A'*lambda;
47 rcent=-diag(lambda)*f-ones(m,1)/t;
48 while eta>acc || norm(rdual)>acc
49     count=count+1;
50     % determine the matrix describing the linear system for the
51     % newton step, solve that and determine Δx and Δtλ
52     H=2*J+A'*diag(-lambda./f)*A;
53     Δx=-H\ (rdual+A'*diag(1./f)*rcent);
54     Δlambda=-diag(lambda./f)*A*Δx+diag(1./f)*rcent;
55     % perform a line search and set the new values for x and lambda
56     % and compute the new values for rdual and constraints
57     [x,lambda,rdual,f]=linesearch_qlp(x,lambda,Δx,Δlambda,...
58     J,A,b,t,rdual,rcent);
59     % compute the new surrogate duality gap, t, and rcent
60     eta=-f'*lambda;
61     t=mu*m/eta;
62     rcent=-diag(lambda)*f-ones(m,1)/t;
63 end

```

Programm A.6: linesearch_qlp.m

```

1 function [x_new,lambda_new,rdual,f]= ...
2     linesearch_qlp(x,lambda,Δx,Δlambda,J,A,b,t,rdual,rcent)
3
4 %-----
5 % linesearch_qlp performs a backtracking linesearch for the search
6 % direction (Δx,Δlambda) calculated in solve_qlp.
7 %-----
8

```

```

9  %-----
10 % Input Parameters:
11 %
12 % x.....current primal variable, n-dimensional vector
13 % lambda.....current dual variable, m-dimensional vector
14 % Δx.....search direction for x, n-dimensional vector
15 % Δlambda.....search direction for lambda, m-dimensional vector
16 % J.....(nxn)-Matrix describing the objective function
17 % A.....(mxn)-Matrix describing the linear constraints
18 % b.....m-dimensional vector describing the linear constraints
19 % t.....real number needed to calculate rcent
20 % rdual.....current value for rdual in the KKT-conditions,
21 %           n-dimensional vector
22 % rcent.....current value for rcent in the KKT-conditions,
23 %           m-dimensional vector
24 %-----
25
26 %-----
27 % Output Parameters:
28 %
29 % x_new.....new value for x, n-dimensional vector
30 % lambda_new..new value for lambda, m-dimensional vector
31 % rdual.....value for rdual at x_new, n-dimensional vector
32 % f.....value for the constraints at x_new, m-dimensional
33 %           vector
34 %-----
35
36 % set some parameters
37 beta=0.5;
38 alpha=0.05;
39 % determine the dimensions
40 m=size(A,1);
41 % determine the maximal value for the step length
42 y=ones(m+1,1);
43 for i=1:m
44     if Δlambda(i)<0
45         y(i)=-lambda(i)/Δlambda(i);
46     end
47 end
48 s=0.99*min(y);
49 % compute current norm of (rdual,rcent)
50 oldnorm=norm([rdual;rcent]);
51 % set new x and lambda
52 x_new=x+s*Δx;
53 lambda_new=lambda+s*Δlambda;
54 % compute the value of the constraints
55 f=A*x_new-b;
56 % compute the new values of rdual and rcent
57 rdual=2*J*x_new+A'*lambda_new;
58 rcent=-diag(lambda_new)*f-ones(m,1)/t;
59
60 while max(f)≥0 || norm([rdual;rcent])>oldnorm*(1-alpha*s)
61     % reduce step length s
62     s=beta*s;
63     % set new x and lambda
64     x_new=x+s*Δx;
65     lambda_new=lambda+s*Δlambda;
66     % compute the value of the constraints
67     f=A*x_new-b;
68     % compute the new values of rdual and rcent

```

```

69     rdual=2*J*x_new+A'*lambda_new;
70     rcent=-diag(lambda_new)*f-ones(m,1)/t;
71 end

```

A.3. Steuerung der Wirk- und Blindleistung mit linearer Zielfunktion

Programm A.7: optimize_P_Q_linear.m

```

1 function [ΔP,ΔQ,exitflag,count,time]=...
2     optimize_P_Q_linear(AP,AQ,P,Q,U,EVB,PFmin,Smax,acc)
3
4 %-----
5 % optimize_P_Q_linear is a function to solve the optimization
6 % problem
7 %
8 %     min ||Q+ΔQ||_1
9 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %         |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %         |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %
13 % with respect to the unknown ΔQ, where ||.||_1 is the 1-norm:
14 % ||x||_1:=sum(|x(i)|)
15 %-----
16
17 %-----
18 % Input parameters:
19 %
20 % AP ..... mxn-matrix to compute an estimate for the change of
21 %          voltage
22 % AQ ..... mxn-matrix to compute an estimate for the change of
23 %          voltage
24 % P ..... current active power, n-dimensional vector
25 % Q ..... current reactive power, n-dimensional vector
26 % U ..... current voltage, m-dimensional vector
27 % EVB ..... bound on diffusion of voltage vector U, real number
28 % PFmin ..... real number
29 % Smax ..... nominal apparent power, n-dimensional vector
30 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
31 %-----
32
33 %-----
34 % Output parameters:
35 %
36 % ΔQ ..... solution of the problem
37 % exitflag .... exitflag:
38 %             exitflag==1 --> no solution could be found, ΔQ is
39 %             not the solution
40 %             exitflag==0 --> ΔQ is the solution of the problem
41 %-----
42
43 tic
44 exitflag=0;
45

```

```

46 %-----
47 % transform the problem into a optimization problem of the form
48 %      min  c'*x
49 %      s.t. A*x ≤ b
50 %      diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
51
52 [m,n]=size(AQ);
53 tp=tan(acos(PFmin));
54
55 A=[-eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
56     eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
57     -tp*eye(n), eye(n) , zeros(n,1), zeros(n,1);
58     -tp*eye(n), -eye(n) , zeros(n,1), zeros(n,1);
59     zeros(1,n), zeros(1,n), 1 , -1 ;
60     AP , AQ , -ones(m,1), zeros(m,1);
61     -AP , -AQ , zeros(m,1), ones(m,1) ];
62
63 b=[P; zeros(n,1); tp*P-Q; tp*P+Q; EVB; -U; U];
64
65 K=[eye(n), zeros(n), zeros(n,1), zeros(n,1)];
66 L=[zeros(n), eye(n), zeros(n,1), zeros(n,1)];
67 d=Smax.^2-P.^2-Q.^2;
68
69 %-----
70
71 [M,N]=size(A); % determine the dimensions of the problem
72 x=zeros(N,1); % choose a starting point
73
74 %-----
75 % Phase I:
76 % find a strictly feasible point x for the constraints A*x≤b by
77 % solving a linear optimization problem
78 %
79 %      min  z                      (phase1)
80 %      s.t. A*x-b ≤ z
81 %      diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d ≤ z
82 %
83 % starting with z>max(A*x-b,diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d)
84 % and stopping as soon as z<0 is achieved
85
86 % determine the maximal violation of the constraints
87 fmax=max([-b;-d]);
88 if fmax≥0
89     % transform to prepare for (phase1)
90     x(N+1)=fmax+0.1;
91     A(:,N+1)=-ones(M,1);
92     K(:,N+1)=zeros(n,1);
93     L(:,N+1)=zeros(n,1);
94     c=(1:N+1==N+1)';
95     % run linear optimization algorithm for (phase1)
96     [x,count(1)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,1);
97     if x(N+1)≥0
98         % if no point (x,z) with z<0 could be found, no feasible
99         % point for the original problem can be found, so stop and
100         % set exitflag=1
101         ΔP=x(1:n);
102         ΔQ=x(n+1:2*n);
103         exitflag=1;
104         time=toc;
105         return

```

```

106     end
107     % (phase1) was successful, retransform the problem
108     x=x(1:N);
109     A=A(:,1:N);
110     K=K(:,1:N);
111     L=L(:,1:N);
112 end
113
114 % If we have found a feasible point we can start the actual
115 % optimization
116 c=-(1:N≤n)';
117 [x,count(2)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,0);
118 count(3)=count(1)+count(2);
119 ΔP=x(1:n);
120 ΔQ=x(n+1:2*n);
121 time=toc;

```

Programm A.8: solve_lqp.m

```

1 function [x,count]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,phaseI)
2
3 %-----
4 % The function solve_lp solves the linear optimization problem
5 %
6 %     min   c'*x
7 %     s.t.  A*x ≤ b
8 %           diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
9 %
10 % with an interior point algorithm.
11 %-----
12
13 %-----
14 % Input Parameters
15 %
16 % c..... n-dimensional column vector describing the
17 %          objective funktion
18 % A..... mxn-matrix describing the constraints
19 % b..... m-dimensional column vector describing the
20 %          constraints
21 % K..... pxn-matrix describing the constraints
22 % L..... pxn-matrix describing the constraints
23 % P..... p-dimensional column vector describing the
24 %          constraints, current active power in optimize_P_Q_*
25 % Q..... p-dimensional column vector describing the
26 %          constraints, current reactive power in optimize_P_Q_*
27 % d..... p-dimensional column vector describing the
28 %          constraints
29 % x..... n-dimensional column vector, feasible starting
30 %          point for the optimization problem
31 % acc..... accuracy, real number (10^-6, 10^-8, ...)
32 % phaseI..... boolean number to decide, if the actual minimum
33 %              should be found (phaseI=0) or if the program
34 %              should be stopped, as soon as feasible point for a
35 %              reduced problem is found, i.e. as soon as the
36 %              objective function c'*x<0
37 %-----
38
39 %-----

```

```

40 % Output Parameters
41 %
42 % x..... n-dimensional column vector, either the solution of
43 %           of the problem or a feasible point for a reduced
44 %           problem
45 %-----
46
47 count=0;
48 % set parameter mu:
49 mu=20;
50 % determine the dimensions of the problem:
51 [m,n]=size(A);
52 p=size(P,1);
53 % set starting dual variable:
54 lambda=ones(m+p,1);
55 % compute the value of the constraints:
56 if phaseI
57     f=[A*x-b;diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d-x(n)*ones(p,1)];
58     df=[A;2*diag(K*x+P)*K+2*diag(L*x+Q)*L-[zeros(p,n-1),ones(p,1)]];
59 else
60     f=[A*x-b;diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d];
61     df=[A;2*diag(K*x+P)*K+2*diag(L*x+Q)*L];
62 end
63 % compute surrogate duality gap eta and starting value for t:
64 eta=-f'*lambda;
65 t=mu*(m+p)/eta;
66 % compute rdual and rcent in the KKT-conditions:
67 rdual=c+df'*lambda;
68 rcent=-diag(lambda)*f-ones(m+p,1)/t;
69 while eta>acc || norm(rdual)>acc
70     count=count+1;
71     % determine the matrix describing the linear system for the
72     % newton step, solve that and determine Δx and deltlambda
73     H=2*K'*diag(lambda(m+1:m+p))*K+2*L'*diag(lambda(m+1:m+p))*L+...
74     df'*diag(-lambda./f)*df;
75     Δx=-H\ (rdual+df'*diag(1./f)*rcent);
76     Δlambda=-diag(lambda./f)*df*Δx+diag(1./f)*rcent;
77     % perform a line search and set the new values for x and lambda
78     % and compute the new values for rdual and constraints
79     [x,lambda,rdual,f,df]=linesearch_lqp(x,lambda,...
80     Δx,Δlambda,c,A,b,K,L,P,Q,d,t,rdual,rcent,phaseI);
81     % if in phase I: check if objective value z<0 and if so: stop
82     if phaseI && c'*x<-acc
83         break;
84     end
85     % compute the new surrogate duality gap eta, t, and rcent
86     eta=-f'*lambda;
87     t=mu*(m+p)/eta;
88     rcent=-diag(lambda)*f-ones(m+p,1)/t;
89 end

```

Programm A.9: linesearch_lqp.m

```

1 function [x_new,lambda_new,rdual,f,df]=linesearch_lqp(x,lambda,...
2     Δx,Δlambda,c,A,b,K,L,P,Q,d,t,rdual,rcent,phaseI)
3
4 %-----
5 % linesearch_lqp performs a backtracking linesearch for the search

```

```
6 % direction ( $\Delta x, \Delta \lambda$ ) calculated in solve_lqp.
7 %-----
8
9 %-----
10 % Input Parameters:
11 %
12 % x.....current primal variable, n-dimensional vector
13 % lambda.....current dual variable, (m+p)-dimensional vector
14 %  $\Delta x$ .....search direction for x, n-dimensional vector
15 %  $\Delta \lambda$ .....search direction for lambda, (m+p)-dimensional vector
16 % c.....n-dimensional vector describing the objective function
17 % A.....(mxn)-Matrix describing the linear constraints
18 % b.....m-dimensional vector describing the linear constraints
19 % K.....(pxn)-Matrix describing the quadratic constraints
20 % L.....(pxn)-Matrix describing the quadratic constraints
21 % P.....p-dimensional column vector describing the
22 %         constraints, current active power in optimize_P_Q_*
23 % Q.....p-dimensional column vector describing the
24 %         constraints, current reactive power in optimize_P_Q_*
25 % d.....p-dimensional vector describing the quadratic
26 %         constraints
27 % t.....real number needed to calculate rcent
28 % rdual.....current value for rdual in the KKT-conditions,
29 %            n-dimensional vector
30 % rcent.....current value for rcent in the KKT-conditions,
31 %            (m+p)-dimensional vector
32 % phaseI.....boolean number describing, if the algorithm is in phase
33 %            I
34 %-----
35
36 %-----
37 % Output Parameters:
38 %
39 % x_new.....new value for x, n-dimensional vector
40 % lambda_new..new value for lambda, (m+p)-dimensional vector
41 % rdual.....value for rdual at x_new, n-dimensional vector
42 % f.....value for the constraints at x_new, (m+p)-dimensional
43 %         vector
44 % df.....value for the derivative of the constraints at x_new,
45 %         (m+p)-dimensional vector
46 %-----
47
48 % set some parameters
49 beta=0.5;
50 alpha=0.05;
51 % determine the dimensions
52 [m,n]=size(A);
53 p=size(K,1);
54 % determine the maximal value for the step length
55 y=ones(m+p+1,1);
56 for i=1:m+p
57     if  $\Delta \lambda(i) < 0$ 
58          $y(i) = -\lambda(i) / \Delta \lambda(i)$ ;
59     end
60 end
61 s=0.99*min(y);
62 % compute current norm of (rdual,rcent)
63 oldnorm=norm([rdual;rcent]);
64 % set new x and lambda
65 x_new=x+s* $\Delta x$ ;
```

```

66 lambda_new=lambda+s*Δlambda;
67 % compute the value of the constraints
68 if phaseI
69     f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
70         diag(L*x_new+2*Q)*L*x_new-d-x_new(n)*ones(p,1)];
71     df=[A;2*diag(K*x_new+P)*K+...
72         2*diag(L*x_new+Q)*L-[zeros(p,n-1),ones(p,1)]];
73 else
74     f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
75         diag(L*x_new+2*Q)*L*x_new-d];
76     df=[A;2*diag(K*x_new+P)*K+2*diag(L*x_new+Q)*L];
77 end
78 % compute the new values of rdual and rcent
79 rdual=c+df'*lambda_new;
80 rcent=-diag(lambda_new)*f-ones(m+p,1)/t;
81
82 while max(f)>0 || norm([rdual;rcent])>oldnorm*(1-alpha*s)
83     % reduce step length s
84     s=beta*s;
85     % set new x and lambda
86     x_new=x+s*Δx;
87     lambda_new=lambda+s*Δlambda;
88     % compute the value of the constraints
89     if phaseI
90         f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
91             diag(L*x_new+2*Q)*L*x_new-d-x_new(n)*ones(p,1)];
92         df=[A;2*diag(K*x_new+P)*K+...
93             2*diag(L*x_new+Q)*L-[zeros(p,n-1),ones(p,1)]];
94     else
95         f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+...
96             diag(L*x_new+2*Q)*L*x_new-d];
97         df=[A;2*diag(K*x_new+P)*K+2*diag(L*x_new+Q)*L];
98     end
99     % compute the new values of rdual and rcent
100    rdual=c+df'*lambda_new;
101    rcent=-diag(lambda_new)*f-ones(m+p,1)/t;
102 end

```

A.4. Steuerung der Wirk- und Blindleistung mit quadratischer Zielfunktion

Programm A.10: optimize_P_Q_quad.m

```

1 function [ΔP, ΔQ, exitflag, count, time]=optimize_P_Q_quad(AP,AQ,...
2     P,Q,U,EVB,PFmin,Smax,acc)
3
4 %-----
5 % optimize_P_Q_quad is a function to solve the optimization
6 % problem
7 %
8 %     min    (||Q+ΔQ||_2)^2
9 %     s.t.   max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %           |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %           |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)

```

```

12 %
13 % with respect to the variable  $\Delta Q$ , where  $||.||_2$  is the 2-norm:
14 %  $||x||_2 := \sqrt{\sum (|x(i)|^2)}$ 
15 %-----
16
17 %-----
18 % Input parameters:
19 %
20 % AP ..... mxn-matrix to compute an estimate for the change of
21 %           voltage
22 % AQ ..... mxn-matrix to compute an estimate for the change of
23 %           voltage
24 % P ..... current active power, n-dimensional vector
25 % Q ..... current reactive power, n-dimensional vector
26 % U ..... current voltage, m-dimensional vector
27 % EVB ..... bound on diffusion of voltage vector U, real number
28 % PFmin ..... real number
29 % Smax ..... nominal apparent power, n-dimensional vector
30 % acc ..... accuracy, real number ( $10^{-6}$ ,  $10^{-8}$ , ...)
31 %-----
32
33 %-----
34 % Output parameters:
35 %
36 %  $\Delta Q$  ..... solution of the problem
37 % exitflag .... exitflag:
38 %           exitflag==1 --> no solution could be found,  $\Delta Q$  is
39 %           not the solution
40 %           exitflag==0 -->  $\Delta Q$  is the solution of the problem
41 %-----
42
43 tic
44 exitflag=0;
45
46 %-----
47 % transform the problem into a optimization problem of the form
48 %   min  $x' * J * x$ 
49 %   s.t.  $A * x \leq b$ 
50 %        $\text{diag}(K * x + 2 * P) * K * x + \text{diag}(L * x + 2 * Q) * L * x \leq d$ 
51
52 [m,n]=size(AQ);
53 tp=tan(acos(PFmin));
54
55 A=[-eye(n)      , zeros(n)      , zeros(n,1), zeros(n,1);
56     eye(n)      , zeros(n)      , zeros(n,1), zeros(n,1);
57     -tp*eye(n)  , eye(n)        , zeros(n,1), zeros(n,1);
58     -tp*eye(n) , -eye(n)        , zeros(n,1), zeros(n,1);
59     zeros(1,n), zeros(1,n), 1      , -1      ;
60     AP          , AQ            , -ones(m,1), zeros(m,1);
61     -AP         , -AQ           , zeros(m,1), ones(m,1) ];
62
63 b=[P; zeros(n,1); tp*P-Q; tp*P+Q; EVB; -U; U];
64
65 K=[eye(n), zeros(n), zeros(n,1), zeros(n,1)];
66 L=[zeros(n), eye(n), zeros(n,1), zeros(n,1)];
67 d=Smax.^2-P.^2-Q.^2;
68
69 %-----
70
71 [M,N]=size(A); % determine the dimensions of the problem

```

```

72 x=zeros(N,1); % choose a starting point
73
74 %-----
75 % Phase I:
76 % find a strictly feasible point x for the constraints  $A*x \leq b$  by
77 % solving a linear optimization problem
78 %
79 %      min    z                      (phase1)
80 %      s.t.   $A*x - b \leq z$ 
81 %            $\text{diag}(K*x+2*P)*K*x + \text{diag}(L*x+2*Q)*L*x - d \leq z$ 
82 %
83 % starting with  $z > \max(A*x - b, \text{diag}(K*x+2*P)*K*x + \text{diag}(L*x+2*Q)*L*x - d)$ 
84 % and stopping as soon as  $z < 0$  is achieved
85
86 % determine the maximal violation of the constraints
87 fmax=max([-b;-d]);
88 if fmax ≥ 0
89     % transform to prepare for (phase1)
90     x(N+1)=fmax+0.1;
91     A(:,N+1)=-ones(M,1);
92     K(:,N+1)=zeros(n,1);
93     L(:,N+1)=zeros(n,1);
94     c=(1:N+1==N+1)';
95     % run linear optimization algorithm for (phase1)
96     [x,count(1)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,1);
97     if x(N+1) ≥ 0
98         % if no point (x,z) with  $z < 0$  could be found, no feasible
99         % point for the original problem can be found, so stop and
100         % set exitflag=1
101         ΔP=x(1:n);
102         ΔQ=x(n+1:2*n);
103         exitflag=1;
104         time=toc;
105         count(2)=0;
106         count(3)=count(1);
107         return
108     end
109     % (phase1) was successful, retransform the problem
110     x=x(1:N);
111     A=A(:,1:N);
112     K=K(:,1:N);
113     L=L(:,1:N);
114 end
115
116 % If we have found a feasible point we can start the actual
117 % optimization
118 J=diag((1:N ≤ n));
119 [x,count(2)]=solve_qqp(J,A,b,K,L,P,Q,d,x,acc);
120 count(3)=count(1)+count(2);
121 ΔP=x(1:n);
122 ΔQ=x(n+1:2*n);
123 time=toc;

```

Programm A.11: solve_qqp.m

```

1 function [x,count]=solve_qqp(J,A,b,K,L,P,Q,d,x,acc)
2
3 %-----

```

```

4 % The function solve_qqp solves the convex optimization problem
5 %
6 %      min   x'*J*x
7 %      s.t.  A*x ≤ b
8 %            diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
9 %
10 % with an interior point algorithm.
11 %-----
12
13 %-----
14 % Input Parameters
15 %
16 % J..... nxn-matrix describing the objective function
17 % A..... mxn-matrix describing the constraints
18 % b..... m-dimensional column vector describing the
19 %          constraints
20 % K..... pxn-matrix describing the constraints
21 % L..... pxn-matrix describing the constraints
22 % P..... p-dimensional column vector describing the
23 %          constraints, current active power in optimize_P_Q_*
24 % Q..... p-dimensional column vector describing the
25 %          constraints, current reactive power in optimize_P_Q_*
26 % d..... p-dimensional column vector describing the
27 %          constraints
28 % x..... n-dimensional column vector, feasible starting
29 %          point for the optimization problem
30 % acc..... accuracy, real number (10^-6, 10^-8, ...)
31 %-----
32
33 %-----
34 % Output Parameters
35 %
36 % x..... n-dimensional column vector, either the solution of
37 %          of the problem or a feasible point for a reduced
38 %          problem
39 %-----
40
41 count=0;
42 % set parameter mu:
43 mu=20;
44 % determine the dimensions of the problem:
45 [m,n]=size(A);
46 p=size(K,1);
47 % set starting dual variable:
48 lambda=ones(m+p,1);
49 % compute the value of the constraints:
50 f=[A*x-b;diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x-d];
51 df=[A;2*diag(K*x+P)*K+2*diag(L*x+Q)*L];
52 % compute surrogate duality gap eta and starting value for t:
53 eta=-f'*lambda;
54 t=mu*(m+p)/eta;
55 % compute rdual and rcent in the KKT-conditions:
56 rdual=2*J*x+df'*lambda;
57 rcent=-diag(lambda)*f-ones(m+p,1)/t;
58 while eta>acc || norm(rdual)>acc
59     count=count+1;
60     % determine the matrix describing the linear system for the
61     % newton step, solve that and determine Δx and Δtλ
62     H=2*J+2*K'*diag(lambda(m+1:m+p))*K+...
63     2*L'*diag(lambda(m+1:m+p))*L+df'*diag(-lambda./f)*df;

```

```

64     Δx=-H\ (rdual+df'*diag(1./f)*rcent);
65     Δlambda=-diag(lambda./f)*df*Δx+diag(1./f)*rcent;
66     % perform a line search and set the new values for x and lambda
67     % and compute the new values for rdual and constraints
68     [x,lambda,rdual,f,df]=linesearch_qqp(x,lambda,Δx, ...
69         Δlambda,J,A,b,K,L,P,Q,d,t,rdual,rcent);
70     % compute the new surrogate duality gap eta, t, and rcent
71     eta=-f'*lambda;
72     t=mu*(m+p)/eta;
73     rcent=-diag(lambda)*f-ones(m+p,1)/t;
74 end

```

Programm A.12: linesearch_qqp.m

```

1 function [x_new,lambda_new,rdual,f,df]=linesearch_qqp(x,lambda, ...
2     Δx,Δlambda,J,A,b,K,L,P,Q,d,t,rdual,rcent)
3
4 %-----
5 % linesearch_qqp performs a backtracking linesearch for the search
6 % direction (Δx,Δlambda) calculated in solve_qqp.
7 %-----
8
9 %-----
10 % Input Parameters:
11 %
12 % x.....current primal variable, n-dimensional vector
13 % lambda.....current dual variable, (m+p)-dimensional vector
14 % Δx.....search direction for x, n-dimensional vector
15 % Δlambda.....search direction for lambda, (m+p)-dimensional vector
16 % J.....nxn-Matrix describing the objective function
17 % A.....(mxn)-Matrix describing the linear constraints
18 % b.....m-dimensional vector describing the linear constraints
19 % K.....(pxn)-Matrix describing the quadratic constraints
20 % L.....(pxn)-Matrix describing the quadratic constraints
21 % P.....p-dimensional column vector describing the
22 %     constraints, current active power in optimize_P_Q*
23 % Q.....p-dimensional column vector describing the
24 %     constraints, current reactive power in optimize_P_Q*
25 % d.....p-dimensional vector describing the quadratic
26 %     constraints
27 % t.....real number needed to calculate rcent
28 % rdual.....current value for rdual in the KKT-conditions,
29 %     n-dimensional vector
30 % rcent.....current value for rcent in the KKT-conditions,
31 %     (m+p)-dimensional vector
32 %-----
33
34 %-----
35 % Output Parameters:
36 %
37 % x_new.....new value for x, n-dimensional vector
38 % lambda_new.....new value for lambda, (m+p)-dimensional vector
39 % rdual.....value for rdual at x_new, n-dimensional vector
40 % f.....value for the constraints at x_new, (m+p)-dimensional
41 %     vector
42 % df.....value for the derivative of the constraints at x_new,
43 %     (m+p)-dimensional vector
44 %-----

```

```
45
46 % set some parameters
47 beta=0.5;
48 alpha=0.05;
49 % determine the dimensions
50 [m,n]=size(A);
51 p=size(K,1);
52 % determine the maximal value for the step length
53 y=ones(m+p+1,1);
54 for i=1:m+p
55     if Δlambda(i)<0
56         y(i)=-lambda(i)/Δlambda(i);
57     end
58 end
59 s=0.99*min(y);
60 % compute current norm of (rdual,rcent)
61 oldnorm=norm([rdual;rcent]);
62 % set new x and lambda
63 x_new=x+s*Δx;
64 lambda_new=lambda+s*Δlambda;
65 % compute the value of the constraints
66 f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+diag(L*x_new+2*Q)*L*x_new-d];
67 df=[A;2*diag(K*x_new+P)*K+2*diag(L*x_new+Q)*L];
68 % compute the new values of rdual and rcent
69 rdual=2*J*x_new+df'*lambda_new;
70 rcent=-diag(lambda_new)*f-ones(m+p,1)/t;
71
72 while max(f)≥0 || norm([rdual;rcent])>oldnorm*(1-alpha*s)
73     % reduce step length s
74     s=beta*s;
75     % set new x and lambda
76     x_new=x+s*Δx;
77     lambda_new=lambda+s*Δlambda;
78     % compute the value of the constraints
79     f=[A*x_new-b;diag(K*x_new+2*P)*K*x_new+diag(L*x_new+...
80         2*Q)*L*x_new-d];
81     df=[A;2*diag(K*x_new+P)*K+2*diag(L*x_new+Q)*L];
82     % compute the new values of rdual and rcent
83     rdual=2*J*x_new+df'*lambda_new;
84     rcent=-diag(lambda_new)*f-ones(m+p,1)/t;
85 end
```

A.5. Steuerung der Blindleistung unter Berücksichtigung eines Fehlers bei der Spannungsberechnung

Programm A.13: optimize_Q_linear_error.m

```
1 function [ΔQ,exitflag,count,time]= ...
2     optimize_Q_linear_error(AQ,Q,P,U,EVB,PFmin,Smax,acc,EQ)
3
4 %-----
5 % optimize_Q_linear_error is a function to solve the optimization
6 % problem
7 %
```

```

8 %      min ||Q+ΔQ||_1
9 %      s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %      |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %      |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %      |AQ(i,j)-AQexact(i,j)| ≤ EQ(i,j)
13 %
14 % with respect to the unknown ΔQ, where ||.|| is the 1-norm:
15 % ||x||_1:=sum(|x(i)|)
16 %-----
17
18 %-----
19 % Input parameters:
20 %
21 % AQ ..... mxn-matrix to compute an estimate for the change of
22 %           voltage
23 % Q ..... current reactive power, n-dimensional vector
24 % P ..... current active power, n-dimensional vector
25 % U ..... current voltage, m-dimensional vector
26 % EVB ..... bound on diffusion of voltage vector U, real number
27 % PFmin ..... real number
28 % Smax ..... nominal apparent power, n-dimensional vector
29 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
30 % EQ ..... mxn Matrix, m-dimensional column vector,
31 %           n-dimensional row vector or real number giving a
32 %           bound to the error arising in the calculation of AQ
33 %-----
34
35 %-----
36 % Output parameters:
37 %
38 % ΔQ ..... solution of the problem
39 % exitflag .... exitflag:
40 %               exitflag==1 --> no solution could be found, ΔQ is
41 %               not the solution
42 %               exitflag==0 --> ΔQ is the solution of the problem
43 %-----
44
45 tic
46 exitflag=0;
47
48 %-----
49 % transform the problem into a linear optimization problem with of
50 % the form
51 %
52 %      min c'*x
53 %      s.t. A*x≤b
54
55 [m,n]=size(AQ);
56 v=min(P*tan(acos(PFmin)),sqrt(Smax.^2-P.^2));
57 % determine the size of EQ and transform it into a mxn-matrix
58 if size(EQ,1)==1
59     if size(EQ,2)==1
60         EQ=EQ*ones(m,n);
61     else
62         EQ=ones(m,n)*diag(EQ);
63     end
64 elseif size(EQ,2)==1
65     EQ=diag(EQ)*ones(m,n);
66 end
67

```

```

68 A=[ eye(n) , zeros(n) , zeros(n) , zeros(n,1) , zeros(n,1) ;
69      -eye(n) , -eye(n) , zeros(n) , zeros(n,1) , zeros(n,1) ;
70      -eye(n) , eye(n) , zeros(n) , zeros(n,1) , zeros(n,1) ;
71      zeros(n) , eye(n) , -eye(n) , zeros(n,1) , zeros(n,1) ;
72      zeros(n) , -eye(n) , -eye(n) , zeros(n,1) , zeros(n,1) ;
73      zeros(1,n) , zeros(1,n) , zeros(1,n) , 1 , -1 ;
74      zeros(m,n) , AQ , EQ , -ones(m,1) , zeros(m,1) ;
75      zeros(m,n) , -AQ , EQ , zeros(m,1) , ones(m,1) ] ;
76
77 b=[ v; Q; -Q; zeros(n,1); zeros(n,1); EVB; -U; U];
78 %-----
79
80 [M,N]=size(A); % determine the dimensions of the problem
81 x=zeros(N,1); % choose a starting point
82
83 %-----
84 % Phase I:
85 % find a strictly feasible point x for the constraints  $A^*x \leq b$  by
86 % solving a linear optimization problem
87 %
88 % min z (phase1)
89 % s.t.  $A^*x - b \leq z$ 
90 %
91 % starting with  $z > \max(A^*x - b)$  and stopping as soon as  $z < 0$  is achieved
92
93 % determine the maximal violation of the constraints
94 fmax=max(A*x-b);
95 if fmax ≥ 0
96     % transform to prepare for (phase1)
97     x(N+1)=fmax+0.1;
98     A(:,N+1)=-ones(M,1);
99     c=(1:N+1==N+1)';
100    % run linear optimization algorithm for (phase1)
101    [x,count(1)]=solve_lp(c,A,b,x,acc,1);
102    if x(N+1) ≥ 0
103        % if no point (x,z) with  $z < 0$  could be found, no feasible
104        % point for the original problem can be found, so stop and
105        % set exitflag=1
106        ΔQ=x(n+1:2*n);
107        exitflag=1;
108        time=toc;
109        return
110    end
111    % (phase1) was successful, retransform the problem
112    x=x(1:N);
113    A=A(:,1:N);
114 end
115
116 %-----
117 % Phase II:
118 % If we have found a feasible point we can start the actual
119 % optimization
120 c=(1:N ≤ n)';
121 [x,count(2)]=solve_lp(c,A,b,x,acc,0);
122 count(3)=count(1)+count(2);
123 ΔQ=x(n+1:2*n);
124 time=toc;

```

Program A.14: optimize_Q_quad_error.m

```

1 function [ΔQ,exitflag,count,time]= ...
2     optimize_Q_quad_error(AQ,Q,P,U,EVB,PFmin,Smax,acc,EQ)
3
4 %-----
5 % optimize_Q_linear_error is a function to solve the optimization
6 % problem
7 %
8 %     minimize      (||Q+ΔQ||_2)^2
9 %     subject to    max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %                  |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %                  |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %                  |AQ(i,j)-AQexact(i,j)| ≤ EQ(i,j)
13 %
14 % with respect to the variable ΔQ, where ||.||_2 is the 2-norm:
15 % ||x||_2:=sqrt(sum(|x(i)|^2))
16 %-----
17
18 %-----
19 % Input parameters:
20 %
21 % AQ ..... mxn-matrix to compute an estimate for the change of
22 %           voltage
23 % Q ..... current reactive power, n-dimensional vector
24 % P ..... current active power, n-dimensional vector
25 % U ..... current voltage, m-dimensional vector
26 % EVB ..... bound on diffusion of voltage vector U, real number
27 % PFmin ..... real number
28 % Smax ..... nominal apparent power, n-dimensional vector
29 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
30 % EQ ..... mxn Matrix, m-dimensional column vector,
31 %           n-dimensional row vector or real number giving a
32 %           bound to the error arising in the calculation of AQ
33 %-----
34
35 %-----
36 % Output parameters:
37 %
38 % ΔQ ..... solution of the problem
39 % exitflag .... exitflag:
40 %               exitflag==1 --> no solution could be found, ΔQ is
41 %               not the solution
42 %               exitflag==0 --> ΔQ is the solution of the problem
43 %-----
44
45 tic
46 exitflag=0;
47
48 %-----
49 % transform the problem into a linear optimization problem with of
50 % the form
51 %
52 %     min    x'*J*x
53 %     s.t.   A*x≤b
54
55 [m,n]=size(AQ);
56 v=min(P*tan(acos(PFmin)),sqrt(Smax.^2-P.^2));
57 % determine the size of EQ and transform it into a mxn-matrix
58 if size(EQ,1)==1
59     if size(EQ,2)==1

```

```

60     EQ=EQ*ones(m,n);
61     else
62         EQ=ones(m,n)*diag(EQ);
63     end
64 elseif size(EQ,2)==1
65     EQ=diag(EQ)*ones(m,n);
66 end
67
68 A=[ eye(n)      , zeros(n)      , zeros(n)      , zeros(n,1) , zeros(n,1) ;
69     -eye(n)     , -eye(n)      , zeros(n)     , zeros(n,1) , zeros(n,1) ;
70     -eye(n)     , eye(n)       , zeros(n)     , zeros(n,1) , zeros(n,1) ;
71     zeros(n)    , eye(n)       , -eye(n)      , zeros(n,1) , zeros(n,1) ;
72     zeros(n)    , -eye(n)      , -eye(n)      , zeros(n,1) , zeros(n,1) ;
73     zeros(1,n) , zeros(1,n)    , zeros(1,n)   , 1          , -1         ;
74     zeros(m,n) , AQ           , EQ           , -ones(m,1) , zeros(m,1) ;
75     zeros(m,n) , -AQ          , EQ           , zeros(m,1) , ones(m,1)  ];
76
77 b=[ v; Q; -Q; zeros(n,1); zeros(n,1); EVB; -U; U];
78 %-----
79
80 [M,N]=size(A); % determine the dimensions of the problem
81 x=zeros(N,1); % choose a starting point
82
83 %-----
84 % Phase I:
85 % find a strictly feasible point x for the constraints  $A^*x \leq b$  by
86 % solving a linear optimization problem
87 %
88 %      min z                      (phase1)
89 %      s.t.  $A^*x - b \leq z$ 
90 %
91 % starting with  $z > \max(A^*x - b)$  and stopping as soon as  $z < 0$  is achieved
92
93 % determine the maximal violation of the constraints
94 fmax=max(A*x-b);
95 if fmax ≥ 0
96     % transform to prepare for (phase1)
97     x(N+1)=fmax+0.1;
98     A(:,N+1)=-ones(M,1);
99     c=(1:N+1==N+1)';
100    % run linear optimization algorithm for (phase1)
101    [x,count(1)]=solve_lp(c,A,b,x,acc,1);
102    if x(N+1) ≥ 0
103        % if no point (x,s) with  $s < 0$  could be found, no feasible
104        % point for the original problem can be found, so stop and
105        % set exitflag=1
106        ΔQ=x(n+1:2*n);
107        exitflag=1;
108        time=toc;
109        count(2)=0;
110        count(3)=count(1);
111        return
112    end
113    % (phase1) was successful, retransform the problem
114    x=x(1:N);
115    A=A(:,1:N);
116 end
117
118 %-----
119 % Phase II:

```

```

120 % If we have found a feasible point we can start the actual
121 % optimization
122 J=diag((1:N≤n));
123 [x,count(2)]=solve_qlp(J,A,b,x,acc);
124 count(3)=count(1)+count(2);
125 ΔQ=x(n+1:2*n);
126 time=toc;

```

A.6. Steuerung der Wirk- und Blindleistung unter Berücksichtigung eines Fehlers

Programm A.15: optimize_P_Q_linear_error.m

```

1 function [ΔP,ΔQ,exitflag,count,time]=...
2     optimize_P_Q_linear_error(AP,AQ,P,Q,U,EVB,PFmin,Smax,acc,EP,EQ)
3
4 %-----
5 % optimize_P_Q_linear_error is a function to solve the optimization
6 % problem
7 %
8 %     min ||Q+ΔQ||_1
9 %     s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %         |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %         |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %         |AP(i,j)-APexact(i,j)| ≤ EP(i,j)
13 %         |AQ(i,j)-AQexact(i,j)| ≤ EQ(i,j)
14 %
15 % with respect to the unknown ΔQ, where ||.||_1 is the 1-norm:
16 % ||x||_1:=sum(|x(i)|)
17 %-----
18
19 %-----
20 % Input parameters:
21 %
22 % AP ..... mxn-matrix to compute an estimate for the change of
23 %           voltage
24 % AQ ..... mxn-matrix to compute an estimate for the change of
25 %           voltage
26 % P ..... current active power, n-dimensional vector
27 % Q ..... current reactive power, n-dimensional vector
28 % U ..... current voltage, m-dimensional vector
29 % EVB ..... bound on diffusion of voltage vector U, real number
30 % PFmin ..... real number
31 % Smax ..... nominal apparent power, n-dimensional vector
32 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
33 % EP ..... mxn Matrix, m-dimensional column vector,
34 %           n-dimensional row vector or real number giving a
35 %           bound to the error arising in the calculation of AP
36 % EQ ..... mxn Matrix, m-dimensional column vector,
37 %           n-dimensional row vector or real number giving a
38 %           bound to the error arising in the calculation of AQ
39 %-----
40
41 %-----

```

```

42 % Output parameters:
43 %
44 % ΔQ ..... solution of the problem
45 % exitflag .... exitflag:
46 %             exitflag==1 --> no solution could be found, ΔQ is
47 %             not the solution
48 %             exitflag==0 --> ΔQ is the solution of the problem
49 %-----
50
51 tic
52 exitflag=0;
53
54 %-----
55 % transform the problem into a optimization problem of the form
56 %     min c'*x
57 %     s.t. A*x ≤ b
58 %         diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
59
60 [m,n]=size(AQ);
61 tp=tan(acos(PFmin));
62 % determine the size of EP and transform it into a mxn-matrix
63 if size(EP,1)==1
64     if size(EP,2)==1
65         EP=EP*ones(m,n);
66     else
67         EP=ones(m,n)*diag(EP);
68     end
69 elseif size(EP,2)==1
70     EP=diag(EP)*ones(m,n);
71 end
72 % determine the size of EQ and transform it into a mxn-matrix
73 if size(EQ,1)==1
74     if size(EQ,2)==1
75         EQ=EQ*ones(m,n);
76     else
77         EQ=ones(m,n)*diag(EQ);
78     end
79 elseif size(EQ,2)==1
80     EQ=diag(EQ)*ones(m,n);
81 end
82
83 A=[-eye(n)      ,zeros(n)      ,zeros(n)      ,zeros(n,1),zeros(n,1);
84     eye(n)      ,zeros(n)      ,zeros(n)      ,zeros(n,1),zeros(n,1);
85     -tp*eye(n)  , eye(n)      ,zeros(n)      ,zeros(n,1),zeros(n,1);
86     -tp*eye(n)  ,-eye(n)      ,zeros(n)      ,zeros(n,1),zeros(n,1);
87     zeros(n)    , eye(n)      ,-eye(n)      ,zeros(n,1),zeros(n,1);
88     zeros(n)    ,-eye(n)      ,-eye(n)      ,zeros(n,1),zeros(n,1);
89     zeros(1,n)  ,zeros(1,n)    ,zeros(1,n)    , 1        ,-1        ;
90     AP-EP       , AQ          , EQ          , -ones(m,1),zeros(m,1);
91     -AP-EP      , -AQ         , EQ          , zeros(m,1),ones(m,1) ];
92
93 b=[P;zeros(n,1);tp*P-Q;tp*P+Q;zeros(n,1);zeros(n,1);EVB;-U;U];
94
95 K=[eye(n) , zeros(n) , zeros(n) , zeros(n,1) , zeros(n,1)];
96 L=[zeros(n) , eye(n) , zeros(n) , zeros(n,1) , zeros(n,1)];
97 d=Smax.^2-P.^2-Q.^2;
98
99 %-----
100
101 [M,N]=size(A); % determine the dimensions of the problem

```

```

102 x=zeros(N,1); % choose a starting point
103
104 %-----
105 % Phase I:
106 % find a strictly feasible point x for the constraints  $A*x \leq b$  by
107 % solving a linear optimization problem
108 %
109 %      min    z                      (phase1)
110 %      s.t.   $A*x-b \leq z$ 
111 %            $\text{diag}(K*x+2*P)*K*x+\text{diag}(L*x+2*Q)*L*x-d \leq z$ 
112 %
113 % starting with  $z > \max(A*x-b, \text{diag}(K*x+2*P)*K*x+\text{diag}(L*x+2*Q)*L*x-d)$ 
114 % and stopping as soon as  $z < 0$  is achieved
115
116 % determine the maximal violation of the constraints
117 fmax=max([-b;-d]);
118 if fmax ≥ 0
119     % transform to prepare for (phase1)
120     x(N+1)=fmax+0.1;
121     A(:,N+1)=-ones(M,1);
122     K(:,N+1)=zeros(n,1);
123     L(:,N+1)=zeros(n,1);
124     c=(1:N+1:-N+1)';
125     % run linear optimization algorithm for (phase1)
126     [x,count(1)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,1);
127     if x(N+1) ≥ 0
128         % if no point (x,z) with  $z < 0$  could be found, no feasible
129         % point for the original problem can be found, so stop and
130         % set exitflag=1
131         ΔP=x(1:n);
132         ΔQ=x(n+1:2*n);
133         exitflag=1;
134         time=toc;
135         count(2)=0;
136         count(3)=count(1);
137         return
138     end
139     % (phase1) was successful, retransform the problem
140     x=x(1:N);
141     A=A(:,1:N);
142     K=K(:,1:N);
143     L=L(:,1:N);
144 end
145
146 % If we have found a feasible point we can start the actual
147 % optimization
148 c=-(1:N≤n)';
149 [x,count(2)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,0);
150 count(3)=count(1)+count(2);
151 ΔP=x(1:n);
152 ΔQ=x(n+1:2*n);
153 time=toc;

```

Programm A.16: optimize_P_Q_quad_error.m

```

1 function [ΔP,ΔQ,exitflag,count,time]=...
2     optimize_P_Q_quad_error(AP,AQ,P,Q,U,EVB,PFmin,Smax,acc,EP,EQ)
3

```

```

4 %-----
5 % optimize_P_Q_quad_error is a function to solve the optimization
6 % problem
7 %
8 %      min  (||Q+ΔQ||_2)^2
9 %      s.t. max(U+AQ*ΔQ)-min(U+AQ*ΔQ) ≤ EVB
10 %          |Q(i)+ΔQ(i)| ≤ P(i)*tan(acos(PFmin))
11 %          |Q(i)+ΔQ(i)| ≤ sqrt(Smax(i)^2-P(i)^2)
12 %          |AP(i,j)-APexact(i,j)| ≤ EP(i,j)
13 %          |AQ(i,j)-AQexact(i,j)| ≤ EQ(i,j)
14 %
15 % with respect to the unknown ΔQ, where ||.||_2 is the 2-norm:
16 % ||x||_2:=sqrt(sum(|x(i)|^2))
17 %-----
18
19 %-----
20 % Input parameters:
21 %
22 % AP ..... mxn-matrix to compute an estimate for the change of
23 %           voltage
24 % AQ ..... mxn-matrix to compute an estimate for the change of
25 %           voltage
26 % P ..... current active power, n-dimensional vector
27 % Q ..... current reactive power, n-dimensional vector
28 % U ..... current voltage, m-dimensional vector
29 % EVB ..... bound on diffusion of voltage vector U, real number
30 % PFmin ..... real number
31 % Smax ..... nominal apparent power, n-dimensional vector
32 % acc ..... accuracy, real number (10^-6, 10^-8, ...)
33 % EP ..... mxn Matrix, m-dimensional column vector,
34 %           n-dimensional row vector or real number giving a
35 %           bound to the error arising in the calculation of AP
36 % EQ ..... mxn Matrix, m-dimensional column vector,
37 %           n-dimensional row vector or real number giving a
38 %           bound to the error arising in the calculation of AQ
39 %-----
40
41 %-----
42 % Output parameters:
43 %
44 % ΔQ ..... solution of the problem
45 % exitflag .... exitflag:
46 %               exitflag==1 --> no solution could be found, ΔQ is
47 %               not the solution
48 %               exitflag==0 --> ΔQ is the solution of the problem
49 %-----
50
51 tic
52 exitflag=0;
53
54 %-----
55 % transform the problem into a optimization problem of the form
56 %      min  x'*J*x
57 %      s.t. A*x ≤ b
58 %          diag(K*x+2*P)*K*x+diag(L*x+2*Q)*L*x ≤ d
59
60 [m,n]=size(AQ);
61 tp=tan(acos(PFmin));
62 % determine the size of EP and transform it into a mxn-matrix
63 if size(EP,1)==1

```

```

64     if size(EP,2)==1
65         EP=EP*ones(m,n);
66     else
67         EP=ones(m,n)*diag(EP);
68     end
69 elseif size(EP,2)==1
70     EP=diag(EP)*ones(m,n);
71 end
72 % determine the size of EQ and transform it into a mxn-matrix
73 if size(EQ,1)==1
74     if size(EQ,2)==1
75         EQ=EQ*ones(m,n);
76     else
77         EQ=ones(m,n)*diag(EQ);
78     end
79 elseif size(EQ,2)==1
80     EQ=diag(EQ)*ones(m,n);
81 end
82
83 A=[-eye(n) , zeros(n) , zeros(n) , zeros(n,1), zeros(n,1);
84     eye(n) , zeros(n) , zeros(n) , zeros(n,1), zeros(n,1);
85     -tp*eye(n), eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
86     -tp*eye(n), -eye(n) , zeros(n) , zeros(n,1), zeros(n,1);
87     zeros(n) , eye(n) , -eye(n) , zeros(n,1), zeros(n,1);
88     zeros(n) , -eye(n) , -eye(n) , zeros(n,1), zeros(n,1);
89     zeros(1,n), zeros(1,n), zeros(1,n), 1 , -1 ;
90     AP-EP , AQ , EQ , -ones(m,1), zeros(m,1);
91     -AP-EP , -AQ , EQ , zeros(m,1), ones(m,1) ];
92
93 b=[P; zeros(n,1); tp*P-Q; tp*P+Q; zeros(n,1); zeros(n,1); EVB; -U; U];
94
95 K=[eye(n), zeros(n), zeros(n), zeros(n,1), zeros(n,1)];
96 L=[zeros(n), eye(n), zeros(n), zeros(n,1), zeros(n,1)];
97 d=Smax.^2-P.^2-Q.^2;
98
99 %-----
100
101 [M,N]=size(A); % determine the dimensions of the problem
102 x=zeros(N,1); % choose a starting point
103
104 %-----
105 % Phase I:
106 % find a strictly feasible point x for the constraints  $A*x \leq b$  by
107 % solving a linear optimization problem
108 %
109 % min z (phase1)
110 % s.t.  $A*x - b \leq z$ 
111 %  $\text{diag}(K*x + 2*P) * K*x + \text{diag}(L*x + 2*Q) * L*x - d \leq z$ 
112 %
113 % starting with  $z > \max(A*x - b, \text{diag}(K*x + 2*P) * K*x + \text{diag}(L*x + 2*Q) * L*x - d)$ 
114 % and stopping as soon as  $z < 0$  is achieved
115
116 % determine the maximal violation of the constraints
117 fmax=max([-b;-d]);
118 if fmax ≥ 0
119     % transform to prepare for (phase1)
120     x(N+1)=fmax+0.1;
121     A(:,N+1)=-ones(M,1);
122     K(:,N+1)=zeros(n,1);
123     L(:,N+1)=zeros(n,1);

```

```

124     c=(1:N+1==N+1)';
125     % run linear optimization algorithm for (phase1)
126     [x,count(1)]=solve_lqp(c,A,b,K,L,P,Q,d,x,acc,1);
127     if x(N+1)≥0
128         % if no point (x,z) with z<0 could be found, no feasible
129         % point for the original problem can be found, so stop and
130         % set exitflag=1
131         ΔP=x(1:n);
132         ΔQ=x(n+1:2*n);
133         exitflag=1;
134         time=toc;
135         return
136     end
137     % (phase1) was successful, retransform the problem
138     x=x(1:N);
139     A=A(:,1:N);
140     K=K(:,1:N);
141     L=L(:,1:N);
142 end
143
144 % If we have found a feasible point we can start the actual
145 % optimization
146 J=diag(1:N≤n);
147 [x,count(2)]=solve_qqp(J,A,b,K,L,P,Q,d,x,acc);
148 count(3)=count(1)+count(2);
149 ΔP=x(1:n);
150 ΔQ=x(n+1:2*n);
151 time=toc;

```

Abbildungsverzeichnis

1.1. Unidirektionaler Leistungsfluss	10
1.2. Spannungsverlauf an einem exemplarischen Netzausläufer - Nur Last	10
1.3. Spannungsverlauf an einem exemplarischen Netzausläufer - Last und Erzeugung	11
1.4. Bidirektionaler Leistungsfluss	12
1.5. Vereinfachter Flowchart für die Funktionsweise der zentralen Steuerungseinheit	14
3.1. Zulässiger Bereich und Norm-Einheitskugeln	43
6.1. Durchschnittliche Anzahl an Iterationen in <code>optimize_Q_linear</code> und <code>optimize_Q_quad</code> in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$ und $\alpha = 0.07$	83
6.2. Durchschnittliche Anzahl an Iterationen in <code>optimize_Q_linear_error</code> und <code>optimize_Q_quad_error</code> in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	83
6.3. Durchschnittliche Anzahl an Iterationen in <code>optimize_P_Q_linear</code> und <code>optimize_P_Q_quad</code> in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$ und $\alpha = 0.14$	83
6.4. Durchschnittliche Anzahl an Iterationen in <code>optimize_P_Q_linear_error</code> und <code>optimize_P_Q_quad_error</code> in Abhängigkeit vom Parameter μ für je 100 Testdatensätze mit $\text{acc} = 10^{-8}$, $m = 20$, $n = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	84
6.5. Durchschnittliche Anzahl an Iterationen in <code>optimize_Q_linear</code> in Abhängigkeit von der gewählten Genauigkeit acc für je 100 Testdatensätze mit $m = 20$, $n = 10$ und $\alpha = 0.07$	84
6.6. Durchschnittliche Anzahl an Iterationen in <code>optimize_Q_quad</code> in Abhängigkeit von der gewählten Genauigkeit acc für je 100 Testdatensätze mit $m = 20$, $n = 10$ und $\alpha = 0.07$	85
6.7. Durchschnittliche Anzahl an Iterationen in <code>optimize_Q_linear_error</code> in Abhängigkeit von der gewählten Genauigkeit acc für je 100 Testdatensätze mit $m = 20$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	85
6.8. Durchschnittliche Anzahl an Iterationen in <code>optimize_Q_quad_error</code> in Abhängigkeit von der gewählten Genauigkeit acc für je 100 Testdatensätze mit $m = 20$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	85

6.9. Durchschnittliche Anzahl an Iterationen in <code>optimize_P_Q_linear</code> in Abhängigkeit von der gewählten Genauigkeit <code>acc</code> für je 100 Testdatensätze mit $m = 20, n = 10$ und $\alpha = 0.14$	86
6.10. Durchschnittliche Anzahl an Iterationen in <code>optimize_P_Q_quad</code> in Abhängigkeit von der gewählten Genauigkeit <code>acc</code> für je 100 Testdatensätze mit $m = 20, n = 10$ und $\alpha = 0.14$	86
6.11. Durchschnittliche Anzahl an Iterationen in <code>optimize_P_Q_linear_error</code> in Abhängigkeit von der gewählten Genauigkeit <code>acc</code> für je 100 Testdatensätze mit $m = 20, n = 10, E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	86
6.12. Durchschnittliche Anzahl an Iterationen in <code>optimize_P_Q_quad_error</code> in Abhängigkeit von der gewählten Genauigkeit <code>acc</code> für je 100 Testdatensätze mit $m = 20, n = 10, E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	87
6.13. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_linear</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10$ und $\alpha = 0.07$	88
6.14. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_quad</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10$ und $\alpha = 0.07$	88
6.15. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_linear_error</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10, E_Q = 10^{-4}$ und $\alpha = 0.07$	88
6.16. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_quad_error</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10, E_Q = 10^{-4}$ und $\alpha = 0.07$	89
6.17. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_linear</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10$ und $\alpha = 0.14$	89
6.18. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_quad</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10$ und $\alpha = 0.14$	89
6.19. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_linear_error</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10, E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	90
6.20. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_quad_error</code> in Abhängigkeit von der Dimension n für je 100 Testdatensätze mit $acc = 10^{-6}, m = 10, E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	90
6.21. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_linear</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $acc = 10^{-6}, n = 10$ und $\alpha = 0.07$	91

6.22. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_quad</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$ und $\alpha = 0.07$	91
6.23. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_linear_error</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	91
6.24. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_quad_error</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	92
6.25. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_linear</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$ und $\alpha = 0.14$	92
6.26. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_quad</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$ und $\alpha = 0.14$	92
6.27. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_linear_error</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	93
6.28. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_quad_error</code> in Abhängigkeit von der Dimension m für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $n = 10$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	93
6.29. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_linear</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.07$	94
6.30. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_quad</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.07$	94
6.31. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_linear_error</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	94
6.32. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_Q_quad_error</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	95
6.33. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_linear</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.14$	95
6.34. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_quad</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$ und $\alpha = 0.14$	95

6.35. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_linear_error</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	96
6.36. Durchschnittliche Anzahl an Iterationen und benötigte Zeit in Sekunden in <code>optimize_P_Q_quad_error</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	96
6.37. Quotient der benötigten Zeit <code>optimize_Q_linear_error</code> und jener von <code>optimize_Q_linear</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	97
6.38. Quotient der benötigten Zeit von <code>optimize_Q_quad_error</code> und jener von <code>optimize_Q_quad</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_Q = 10^{-4}$ und $\alpha = 0.07$	98
6.39. Quotient der benötigten Zeit von <code>optimize_P_Q_linear_error</code> und jener von <code>optimize_P_Q_linear</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	98
6.40. Quotient der benötigten Zeit von <code>optimize_P_Q_quad_error</code> und jener von <code>optimize_P_Q_quad</code> in Abhängigkeit von den Dimensionen m und n für je 100 Testdatensätze mit $\text{acc} = 10^{-6}$, $E_P = E_Q = 10^{-4}$ und $\alpha = 0.14$	98

Literatur

- [BV04] S.P. Boyd und L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. ISBN: 9780521833783. URL: <http://books.google.de/books?id=mYm0bLd3fcoC> (siehe S. 17, 22, 30, 82).
- [Dav06] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. CAMBRIDGE - USA, 2006. ISBN: 9780898716139. URL: <http://books.google.de/books?id=TvwiyF8vy3EC> (siehe S. 97).
- [Deu04] Peter Deuffhard. *Newton Methods for Nonlinear Problems - Affine Invariance and Adaptive Algorithms*. Corr. 2nd printing 2005. Berlin, Heidelberg: Springer, 2004. ISBN: 978-3-540-21099-3. URL: http://books.google.de/books?id=120xK__HG_kC (siehe S. 26).
- [Heu04] H. Heuser. *Lehrbuch der Analysis*. 2. Teubner, 2004. ISBN: 9783519622321. URL: <http://books.google.de/books?id=zN7aCuj5t3AC> (siehe S. 20).
- [oes13] oesterreichsenergie.at. *Stromtransport: Vom Kraftwerk in die Steckdose*. [Online; Stand 15. Februar 2013]. 2013. URL: <http://oesterreichsenergie.at/stromtransport-vom-kw-in-die-steckdose.html> (siehe S. 9).
- [RR02] H.E. Rose und A.E. Rose. *Linear Algebra: A Pure Mathematical Approach*. Birkhäuser, 2002. ISBN: 9783764369057. URL: <http://books.google.at/books?id=mTdAj-Yn4L4C> (siehe S. 39).
- [Sch09] A.J. Schwab. *Elektroenergiesysteme: Erzeugung, Transport, Übertragung und Verteilung elektrischer Energie*. Springer, 2009. ISBN: 9783540922261. URL: <http://books.google.at/books?id=93Vp9pNZTzgC> (siehe S. 9).
- [Vie+10] A. Viehweider u. a. “Smart robust voltage control for distribution networks using interval arithmetic and state machine concepts”. In: *Innovative Smart Grid Technologies Conference Europe (ISGT Europe), 2010 IEEE PES*. Okt. 2010, S. 1 –8. doi: 10.1109/ISGTEUROPE.2010.5638932 (siehe S. 9, 11).
- [Wik13] Wikipedia. *Stromnetz — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 15. Februar 2013]. 2013. URL: <http://de.wikipedia.org/w/index.php?title=Stromnetz&oldid=113802170> (siehe S. 9).

- [Wri96] S. J. Wright. “Modified Cholesky factorizations in interior-point algorithms for linear programming”. In: *SIAM Journal on Optimization* 9 (1996), S. 1159–1191. URL: <http://pages.cs.wisc.edu/~swright/papers/P600.pdf> (siehe S. 87, 99).
- [Wri97] S.J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial und Applied Mathematics, 1997. ISBN: 9780898713824. URL: <http://books.google.de/books?id=9TQ5tP8uqeYC> (siehe S. 17).
- [Ye11] Y. Ye. *Interior Point Algorithms: Theory and Analysis*. Wiley, 2011. ISBN: 9781118030950. URL: <http://books.google.de/books?id=9Jh58LImo-UC> (siehe S. 17).

B. Lebenslauf

Persönliche Daten

Daniel Schwabeneder

Österreich

Schulbildung

09/1992–12/1992 Volksschule Ternberg

12/1992–07/1996 Volksschule Krenglbach

09/1996–07/2004 BG/BRG Dr. Schauerstraße Wels

Zivildienst

10/2004–09/2005 Verein Miteinander Wels

Studium

seit 10/2005 Diplomstudium Mathematik, Universität Wien

Berufserfahrung

07/2001 Ferialpraktikum, Prillinger GmbH

07/2003 Ferialpraktikum, SPAR-Zentrallager Wels

07/2004–08/2004 Ferialpraktikum, Prillinger GmbH

07/2007–08/2007 Ferialpraktikum, NIROTEC Aschl GmbH

07/2008–08/2008 Ferialpraktikum, NIROTEC Aschl GmbH

07/2009–08/2009 Ferialpraktikum, Fronius International GmbH,
Standort Thalheim, Abteilung Numerische Simulation

seit 06/2011 Persönlicher Assistent bei der Wiener Assistenzgenossenschaft (WAG)