Parallelisierung von Innere-Punkte-Verfahren mittels Cyclic Reduction

An der Fakultät für Mathematik
der Otto-von-Guericke-Universität Magdeburg
zur Erlangung des akademischen Grades
Bachelor of Science
angefertigte

Bachelorarbeit

vorgelegt von
Do Duc Le
geboren am 17.07.1991 in Papenburg,
Studiengang Mathematik,
Studienrichtung Computermathematik.

15. August 2014

Betreut am Institut für Mathematische Optimierung von
Prof. Dr. rer. nat. habil. Sebastian Sager

Inhaltsverzeichnis

1. Einleitung **3**
1.1. Struktur der Arbeit 3

2. Quadratische Programmierung **5**
2.1. Quadratische Programmierung in Model Predictive Control 6

3. Primal-duales Innere-Punkte-Verfahren für QP **8**
3.1. Herleitung 8
3.2. Prädiktor-Korrektor-Methode 11

4. Cyclic Reduction **15**
4.1. Herleitung 15
4.2. Algorithmus im Detail 17
4.3. Analytische Wohldefiniertheit von Cyclic Reduction 19

5. Anwendung auf MPC **24**
5.1. Transformation der Gleichungssysteme 24
5.2. Übertragung auf MPC 25

6. Implementation und Tests **28**

7. Schlusswort **31**

A. Programmcode **32**
A.1. IPM Prädiktor-Korrektor Code 32
A.2. Cyclic Reduction Code 39
A.3. Hilfsfunktionen 42
A.4. Matlab Test-script 44

1. Einleitung

Innere-Punkte-Verfahren (Interior Point Method, IPM) sind eine erfolgreiche Klasse von Algorithmen zur Lösung von konvexen Optimierungsproblemen. Sie haben sich in der Praxis als sehr leistungsfähig erwiesen und werden daher in vielen Anwendungen genutzt. Die Grundidee dieser Algorithmen besteht darin, einen Weg zur optimalen Lösung durch das Innere des zulässigen Bereichs zu suchen. Sie erzeugen Iterierte, die die Ungleichungsnebenbedingungen strikt erfüllen, daher auch der Name. Der aufwändigste Schritt bei Innere-Punkte-Verfahren ist das Lösen eines linearen Gleichungssystems, das sich aus den Karush-Kuhn-Tucker-Bedingungen (*kurz:* KKT-Bedingungen) ergibt [1], in jeder Iteration.
Die Klasse von Problemen, die wir uns im Zusammenhang dieser Arbeit anschauen, sind sogenannte quadratische Optimierungsprobleme mit einer quadratischen Zielfunktion und linearen bzw. affinen Nebenbedingungen. Besonderes Interesse gilt dabei einer Subklasse von quadratischen Problemen, die im Kontext von dynamischen Optimierungsproblemen, insbesondere bei der Optimalen Steuerung und bei der Modellprädiktiven Regelung (Model Predictive Control, MPC) auftreten. Solche Probleme weisen eine Block-Band-Struktur in der Zielfunktion und in den Nebenbedingungen auf, die sich bis in das KKT-System durchzieht. Ausnutzen dieser speziellen Struktur bei der Faktorisierung des linearen Gleichungssystems kann zu erheblichen Laufzeitverbesserungen führen. Viele Verfahren arbeiten mit der wohlbekannten Cholesky-Zerlegung oder einer modifizierten Version von ihr [2]. Auch wenn diese Methode sehr effizient ist, eignet sie sich nur bedingt für Parallelisierung, sodass auf modernen Rechnerarchitekturen mit mehreren Prozessoren sich andere Faktorisierungsverfahren als geeigneter erweisen könnten.
Eine Möglichkeit bietet das Cyclic Reduction (vgl.[3]), das Gegenstand dieser Arbeit sein soll. Es ist ein rekursives Verfahren zur Lösung von linearen Gleichungssystemen mit Bandstruktur. Die Idee besteht darin, in jeder Iteration die Anzahl der Gleichungen durch Elimination jeder zweiten Unbekannten zu halbieren, bis man das reduzierte System mit einem direkten Verfahren für dichtbesetzte Probleme lösen kann. Die Elimination der Unbekannten kann dabei teilweise parallel geschehen.

1.1. Struktur der Arbeit

In Kapitel 2 werden zunächst die quadratischen Programme (QPs) eingeführt, mit denen wir uns in dieser Arbeit beschäftigen wollen. Dabei werden wir auch die MPC-Probleme als solche QPs identifizieren. Die nachfolgenden Kapitel beschreiben die Methoden, das Innere-Punkte-verfahren und das Cyclic Reduction, die zur Lösung der quadratischen Probleme verwendet werden. Diese Verfahren werden hier allgemein erklärt, die Übertragung auf das MPC-Problem geschieht in Kapitel 5. Schließlich werden

die Methoden implementiert und numerische Ergebnisse in Kapitel 6 vorgestellt.

2. Quadratische Programmierung

Die Problemklasse, mit der wir uns hier beschäftigen wollen, sind quadratische Optimierungsprobleme (QPs). Sie sind als Problemklasse an sich schon relevant, spielen aber auch als Subprobleme bei vielen Methoden der allgemeinen, nichtlinearen Optimierung eine wichtige Rolle. Es gibt mehrere verschiedene Arten und Weisen, ein quadratisches Problem zu spezifizieren, wir nutzen hier die allgemeine Form:

$$\begin{aligned} \min_z \ & \frac{1}{2} z^T Q z + q^T z \\ \text{s.t. } & Hz = h \\ & Gz \leq g \end{aligned} \tag{2.1}$$

mit Vektoren $q \in \mathbb{R}^{n_z}, h \in \mathbb{R}^{n_h}, g \in \mathbb{R}^{n_g}$ und symmetrischer Matrix $Q \in \mathbb{R}^{n_z \times n_z}$ und Matrizen $H \in \mathbb{R}^{n_h \times n_z}, G \in \mathbb{R}^{n_g \times n_z}$. Wir minimieren also eine quadratische Zielfunktion unter affinen Nebenbedingungen.
Der Aufwand, mit dem ein solches QP gelöst werden kann, hängt stark von der Zielfunktion und der Anzahl der Ungleichungsnebenbedingungen ab [1]. Ist Q positiv semidefinit, so ist (2.1) ein konvexes Optimierungsproblem. Ein nichtkonvexes QP ist im Allgemeinen signifikant schwieriger zu lösen, da mehrere stationäre Punkte und lokale Minima existieren können (nichtkonvexe QPs sind $\mathcal{NP}$-schwer [4]). Wir gehen daher in dieser Arbeit stets davon aus, dass die Matrix Q positiv semidefinit ist, d.h. es handelt sich beim Problem (2.1) um ein konvexes Optimierugsproblem.

Es existieren unterschiedliche Verfahren zur Lösung des konvexen quadratischen Problems (2.1), die an dieser Stelle nur kurz genannt werden sollen. Eine ausführliche Einführung dieser Verfahren findet man in [1].

1. Treten nur gleichungsbeschränkte Nebenbedingungen auf, so können *direkte Verfahren* zur Lösung verwendet werden. Gemeinsam ist diesen Verfahren, dass sie zunächst die KKT-Bedingungen für das Probem aufstellen und versuchen, dass sich ergebende KKT-System mit Faktorisierungen zu lösen. Beispiele hierfür sind das *Nullraum-Verfahren* und die *Schur-Komplement-Methode*.
   Iterative Verfahren hingegen erzeugen sukzessive eine Folge von Vektoren, die immer bessere Approximationen zur Lösung liefern. Sie eignen sich besonders für große Probleme. Hierzu gehören z.B. bestimmte Klassen von *Konjgierte-Gradienten-Methoden*.

2. Für die Lösung des allgemeinen quadratischen Problems (2.1) sind zwei große Klassen *iterativer Verfahren* zu nennen: die *Active-Set-Methoden* und die *Innere-Punkte-Verfahren*. *Innere-Punkte-Verfahren* sind Gegenstand dieser Arbeit und werden später in Kapitel 3 näher betrachtet.

2.1. Quadratische Programmierung in Model Predictive Control

In Kapitel 4 werden wir die Innere-Punkte-Verfahren, genauer das Lösen des perturbierten KKT-Systems, speziell für die Probleme betrachten, die im Zusammenhang der Model Predictive Control (MPC) auftauchen. Wir verzichten hier auf eine detaillierte Einführung in MPC und verweisen auf [5] für eine allgemeine Einführung und Motivation/Herkunft dieser Problemklasse.
Die in den dynamischen Systemen auftretenden Optimierungsvariablen können aufgeteilt werden in abhängige Zustandsvariablen $x \in \mathbb{R}^{n_x}$, die das System beschreiben (z.B. Position eines Fahrzeugs) und unabhängigen Parametern, den Steuerungen $u \in \mathbb{R}^{n_u}$, die als Steuergröße verstanden werden und von außen direkt beeinflusst werden können (z.B. Beschleunigung bei einem Fahrzeug). Viele dieser Probleme lassen sich als quadratisches Optimierungsproblem formulieren mit der Form [6]:

$$\min_z \quad \sum_{k=0}^{N} (\frac{1}{2} z_k^T B_k z_k + b_k^T z_k) \tag{2.2a}$$

$$\text{s.t.} \quad E z_{k+1} = C_k z_k + c_k \qquad \forall k \in \{0, 1, ..., N-1\}, \tag{2.2b}$$

$$D_k z_k \leq d_k \qquad \forall k \in \{0, 1, ..., N\}. \tag{2.2c}$$

Die Optimierungsvariable $z_k \in \mathbb{R}^{n_z}$ kann als Zusammenfassung der Zustandsvariablen und Steuerungen verstanden werden, d.h. $z_k = \begin{bmatrix} x_k^T & u_k^T \end{bmatrix}^T$. Die Kostenfunktion ist eine Summe aus quadratischen Funktionen mit symmetrischen, positiv definiten Matrizen $B_k \in \mathbb{R}^{n_z \times n_z}$ und Vektoren $g_k \in \mathbb{R}^{n_z}$. Zwei aufeinanderfolgende Vektoren z_k und z_{k+1} sind gekoppelt durch die Gleichungsnebenbedingung (2.2b), wobei $C_k \in \mathbb{R}^{n_x \times n_z}$, $E = \begin{bmatrix} I & 0 \end{bmatrix} \in \mathbb{R}^{n_x \times n_z}$ mit Einheitsmatrix $I \in \mathbb{R}^{n_x \times n_x}$ und $c_k \in \mathbb{R}^{n_x}$. Weitere Beschränkungen an die Variablen z_k sind gegeben durch (2.2c) mit $D_k \in \mathbb{R}^{n_d \times n_z}$, $d_k \in \mathbb{R}^{n_d}$. Das Problem (2.2) kann in die allgemeine Form (2.1) gebracht werden mit

$$z = \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_{N-1} \\ z_N \end{bmatrix}, \qquad Q = \begin{bmatrix} B_0 & & & & \\ & B_1 & & & \\ & & \ddots & & \\ & & & B_{N-1} & \\ & & & & B_N \end{bmatrix}, \qquad q = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \\ b_N \end{bmatrix},$$

$$H = \begin{bmatrix} C_0 & -E & & & & \\ & C_1 & -E & & & \\ & & \ddots & \ddots & & \\ & & & C_{N-2} & -E & \\ & & & & C_{N-1} & -E \end{bmatrix}, \qquad h = \begin{bmatrix} -c_0 \\ -c_1 \\ \vdots \\ -c_{N-2} \\ -c_{N-1} \end{bmatrix}, \tag{2.3}$$

$$G = \begin{bmatrix} D_0 & & & & \\ & D_1 & & & \\ & & \ddots & & \\ & & & D_{N-1} & \\ & & & & D_N \end{bmatrix}, \qquad g = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{N-1} \\ d_N \end{bmatrix}.$$

Im nächsten Kapitel stellen wir ein mögliches Lösungsverfahren für quadratische Probleme (2.1) vor: das *Innere-Punkte-Verfahren*. Wir werden sehen, dass der Spezialfall (2.2) eine besondere Struktur besitzt, die wir uns zu Nutze machen werden, um das in jeder Iteration auftretende Gleichungssystem effizient zu lösen.

3. Primal-duales Innere-Punkte-Verfahren für QP

Innere-Punkte-Verfahren werden zur Lösung linearer und quadratischer Probleme verwendet, finden aber auch in der nichtlinearen Programmierung Anwendung. Primal-duale Methoden sind eine Unterklasse dieser Verfahren und haben sich seit den frühen 1990er Jahren als äußerst effizient erwiesen [1, p. 392].
In diesem Kapitel entwickeln wir ein primal-duales Innere-Punkte-Verfahren, um das quadratische Problem

$$\begin{aligned} \min_z \frac{1}{2} z^T Q z + q^T z \\ \text{s.t. } Hz = h \\ Gz \leq g \end{aligned} \tag{3.1}$$

zu lösen. Wir leiten das allgemeine Framework für einen Algorithmus her und stellen eine in der Praxis häufig genutzte Variante des Verfahrens, die *Prädiktor-Korrektor-Methode*, vor. Wir werden uns hierbei haupsächlich an [1] orientieren.

3.1. Herleitung

Zunächst stellen wir die notwendigen Optimalitätsbedingungen erster Ordnung, auch bekannt als die Karush-Kuhn-Tucker-Bedingungen, für das Problem (2.1) auf. Mit Hilfe der Lagrange-Funktion

$$L(z, \mu, \lambda) = \frac{1}{2} z^T Q z + q^T z + \mu^T (Hz - h) + \lambda^T (Gz - g) \tag{3.2}$$

und den Lagrange-Multiplikatoren $\lambda \in \mathbb{R}^{n_g}, \mu \in \mathbb{R}^{n_h}$ lauten die KKT-Bedingungen:

$$Qz + q + H^T \mu + G^T \lambda = 0, \tag{3.3a}$$
$$Hz - h = 0, \tag{3.3b}$$
$$Gz - g \leq 0, \tag{3.3c}$$
$$(Gz - g)_i \lambda_i = 0, \quad i = 1, 2, ..., n_g, \tag{3.3d}$$
$$\lambda \geq 0 \tag{3.3e}$$

Wie in [1] gezeigt, können die KKT-Bedingungen (3.3) durch Einführung von Slack-

Variablen $t \geq 0$ umgeschrieben werden:

$$\begin{aligned} Qz + q + H^T\mu + G^T\lambda &= 0, && \text{(3.4a)} \\ Hz - h &= 0, && \text{(3.4b)} \\ Gz - g + t &= 0, && \text{(3.4c)} \\ t_i\lambda_i &= 0, \quad i = 1, 2, ..., n_g, && \text{(3.4d)} \\ (\lambda, t) &\geq 0 && \text{(3.4e)} \end{aligned}$$

Unter der Annahme, dass Q positiv semidefinit ist, ist das Problem (3.1) konvex und jedes lokale Minimum ist gleichzeitig ein globales Minimum. Gilt zudem eine Constraint Qualification (CQ), so sind die Optimalitätsbedingungen erster Ordnung (3.3) nicht nur notwendig, sondern auch hinreichend für ein globales Minimum [7]. Es reicht also, eine Lösung des Systems (3.4) zu finden.
Primal-duale Innere-Punkte-Methoden nutzen nun das Newton-Verfahren mit modfizierter Schrittrichtung und Schrittweiten, um das System der vier Gleichungen (3.4a), (3.4b), (3.4c) und (3.4d) zu lösen. Die Schrittweiten werden derart angepasst, dass die Bedingung (3.4e) strikt eingehalten wird, d.h. $(\lambda, t) > 0$ gilt.

Zur Herleitung primal-dualer Innere-Punkte-Methoden betrachten wir die Gleichungsnebenbedingungen in (3.4) als nichtlineares System $F = 0$ mit der Abbildung

$$F : \mathbb{R}^{n_z+n_h+2n_g} \to \mathbb{R}^{n_z+n_h+2n_g}, \quad F(z, \mu, \lambda, t) = \begin{bmatrix} Qz + q + H^T\mu + G^T\lambda \\ Hz - h \\ Gz - g + t \\ T\Lambda e \end{bmatrix}, \tag{3.5}$$

wobei $T = \text{diag}(t_1, t_2, ..., t_{n_g})$, $\Lambda = \text{diag}(\lambda_1, \lambda_2, ..., \lambda_{n_g})$. Der Vektor e sei nun immer der Vektor aus Einsen mit passender Dimension (hier $e = (1, 1, ..., 1)^T \in \mathbb{R}^{n_g}$). Wir wenden das Newton-Verfahren auf das System

$$F(z, \mu, \lambda, t) = 0 \tag{3.6}$$

an: Das Newton-Verfahren bildet ein lineares Modell der Funktion in einem Ausgangspunkt und verwendet dessen Nullstelle als Näherung für die Nullstelle der eigentlichen Funktion. Die so erhaltene Nullstelle dient wiederum als Ausgangspunkt für einen weiteren Verbesserungsschritt. Um ausgehend von einer Iterierten $(z_k, \mu_k, \lambda_k, t_k)$ zur nächsten zu gelangen, wird das lineare Gleichungssystem

$$J(z, \mu, \lambda, t) \begin{bmatrix} \Delta z \\ \Delta\mu \\ \Delta\lambda \\ \Delta t \end{bmatrix} = -F(z, \mu, \lambda, t) \tag{3.7}$$

gelöst, wobei $J(z, \mu, \lambda, t)$ die Jakobi-Matrix der Funktion F ist. Im Sinne eines *gedämpften*

Newtonverfahrens erhält man die neue Iterierte durch

$$(z^{k+1}, \mu^{k+1}, \lambda^{k+1}, t^{k+1}) = (z^k, \mu^k, \lambda^k, t^k) + \alpha(\Delta z, \Delta\mu, \Delta\lambda, \Delta t), \quad \alpha \in (0, 1]. \tag{3.8}$$

Mit der Notation

$$r_q := Qz + q + H^T\mu + G^T\lambda, \quad r_h := Hz - h, \quad r_g := Gz - g + t$$

lautet das zu lösende System für den Fall (3.7)

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T & \Lambda \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\mu \\ \Delta\lambda \\ \Delta t \end{bmatrix} = - \begin{bmatrix} r_q \\ r_h \\ r_g \\ T\Lambda e \end{bmatrix}. \tag{3.9}$$

Oftmals können wir jedoch nur einen sehr kleinen Schritt ($\alpha \ll 1$) entlang der Newtonrichtung $(\Delta z, \Delta\mu, \Delta\lambda, \Delta t)$ gehen, bevor die Positivitätsbedingung $(\lambda, t) > 0$ verletzt wird, sodass wir nur langsam gegen die gewünschte Lösung konvergieren.
Primal-duale Innere-Punkte-Methoden lösen daher nicht das Newton-System (3.9), sondern ein leicht verändertes System mit modifizierter Komplementaritätsbedingung

$$t_i\lambda_i = \sigma\xi, \quad i = 1, 2, ..., n_g \quad \text{mit} \quad \sigma \in (0, 1] \quad \text{und} \quad \xi := \frac{1}{n_g}\sum_{i=1}^{n_g} t_i\lambda_i = \frac{t^T\lambda}{n_g} \tag{3.10}$$

anstatt (3.4d). Man nennt σ den sogenannten Centering-Parameter und ξ das Dualitätsmaß. Man betrachtet also nicht die exakten KKT-Bedingungen, sondern eine gestörte Form und lässt diese Störung gegen Null streben.
Die modifizierte Schrittrichtung erhalten wir daher als Lösung des Systems

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T & \Lambda \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\mu \\ \Delta\lambda \\ \Delta t \end{bmatrix} = - \begin{bmatrix} r_q \\ r_h \\ r_g \\ T\Lambda e - \sigma\xi e \end{bmatrix}. \tag{3.11}$$

Typischerweise erlaubt ein Wert $\sigma > 0$ einen größeren Schritt entlang der Suchrichtung, bevor die Positivitätsbedingung verletzt wird.
Damit haben wir alles zusammen, um ein erstes allgemeines Framework für den Algorithmus zu formulieren:

Algorithm 1: Allgemeines Framework eines Innere-Punkte-Verfahrens für QP

Input: Startpunkt $(z^0, \mu^0, \lambda^0, t^0)$ mit $(\lambda^0, t^0) > 0$.
Output: Lösung $(z, \mu, \lambda, t) = (z^k, \mu^k, \lambda^k, t^k)$ der KKT-Bedingungen (3.4).

1 Setze $k = 0$.
2 **while** *KKT-Bedingungen sind nicht hinreichend genau erfüllt* **do**
3 Wähle $\sigma^k \in [0, 1]$ und löse das Gleichungssystem

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T^k & \Lambda^k \end{bmatrix} \begin{bmatrix} \Delta z^k \\ \Delta \mu^k \\ \Delta \lambda^k \\ \Delta t^k \end{bmatrix} = - \begin{bmatrix} r_q^k \\ r_h^k \\ r_g^k \\ T^k \Lambda^k e - \sigma^k \xi^k e \end{bmatrix}.$$

mit $\xi^k = (t^k)^T \lambda^k / n_g$.
4 Setze

$$(z^{k+1}, \mu^{k+1}, \lambda^{k+1}, t^{k+1}) = (z^k, \mu^k, \lambda^k, t^k) + \alpha^k (\Delta z^k, \Delta \mu^k, \Delta \lambda^k, \Delta t^k)$$

mit $\alpha^k \in (0, 1]$, sodass $(\lambda^{k+1}, t^{k+1}) > 0$ erfüllt ist.
5 Setze $k = k + 1$.

3.2. Prädiktor-Korrektor-Methode

Wir stellen eine in praktischen Algorithmen oft angewendete Variante der Innere-Punkte-Verfahren vor: die *Prädiktor-Korrektor-Methode*. Der Algorithmus wurde von Mehrotra [8] ursprünglich für lineare Programme entwickelt, lässt sich aber leicht für konvexe QPs erweitern. Die Idee ist es, den Linearisierungsfehler durch das Newton-Verfahren mit Hilfe eines sogenannten *corrector steps* zu kompensieren.
Zur Herleitung betrachten wir die Newtonrichtung (eng. *affine-scaling direction*), definiert durch die Lösung des Systems

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T & \Lambda \end{bmatrix} \begin{bmatrix} \Delta z_{\text{aff}} \\ \Delta \mu_{\text{aff}} \\ \Delta \lambda_{\text{aff}} \\ \Delta t_{\text{aff}} \end{bmatrix} = - \begin{bmatrix} r_q \\ r_h \\ r_g \\ T \Lambda e \end{bmatrix}. \tag{3.12}$$

Mit einem Vollschritt in diese Richtung erhalten wir:

$$\begin{aligned} (t_i + \Delta t_{\text{aff},i})(\lambda_i + \Delta \lambda_{\text{aff},i}) &= \underbrace{t_i \lambda_i + t_i \Delta \lambda_{\text{aff},i} + \Delta t_{\text{aff},i} \lambda_i}_{= 0, \text{ folgt aus 4. Zeile von (3.12)}} + \Delta t_{\text{aff},i} \Delta \lambda_{\text{aff},i} \\ &= \Delta t_{\text{aff},i} \Delta \lambda_{\text{aff},i}, \end{aligned}$$

d.h. der aktualisierte Wert von $t_i\lambda_i$ verletzt die Komplementaritätsbedingung um $\Delta t_i^{\text{aff}}\Delta\lambda_i^{\text{aff}}$. Um diese Abweichung zu korrigieren, löst man das System

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T & \Lambda \end{bmatrix} \begin{bmatrix} \Delta z_{\text{cor}} \\ \Delta\mu_{\text{cor}} \\ \Delta\lambda_{\text{cor}} \\ \Delta t_{\text{cor}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\Delta T_{\text{aff}}\Delta\Lambda_{\text{aff}}e \end{bmatrix}, \tag{3.13}$$

die kombinierte Suchrichtung $(\Delta z_{\text{aff}}, \Delta\mu_{\text{aff}}, \Delta\lambda_{\text{aff}}, \Delta t_{\text{aff}}) + (\Delta z_{\text{cor}}, \Delta\mu_{\text{cor}}, \Delta\lambda_{\text{cor}}, \Delta t_{\text{cor}})$ reduziert oftmals das Dualitätsmaß $\xi = t^T\lambda/n_g$ stärker als die *affine-scaling*-Richtung allein.
Die Prädiktor-Korrektor-Methode für Innere-Punkte-Verfahren verwendet nun obige Idee und geht wie folgt vor: Zunächst wird das System (3.12) gelöst, um die *affine-scaling*-Richtung zu erhalten. Anschließend berechnen wir die maximal möglichen Schrittweiten hinsichtlich der Nicht-Negativität jeweils für die Variablen t und λ, d.h.

$$\alpha_{\text{aff}}^t = \min(1, \min_{i:\Delta t_{\text{aff},i}<0} -\frac{t_i}{\Delta t_{\text{aff},i}}), \quad \alpha_{\text{aff}}^\lambda = \min(1, \min_{i:\Delta\lambda_{\text{aff,i}}<0} -\frac{\lambda_i}{\Delta\lambda_{\text{aff},i}}),$$

und berechnen damit das Dualitätsmaß bzgl. der *affine-scaling*-Richtung

$$\xi_{\text{aff}} = (t^T + \alpha_{\text{aff}}^t\Delta t_{\text{aff}})^T(\lambda + \alpha_{\text{aff}}^\lambda\Delta\lambda_{\text{aff}})/n_g.$$

Dieses Dualitätsmaß geht ein in die Berechung des Centering-Parameters σ. Die Idee ist, durch einen Vergleich von ξ_{aff} mit dem aktuellen Dualitätsmaß ξ die Güte des Newtonschritts zu beurteilen. Ist beispielsweise $\xi_{\text{aff}} \ll \xi$, d.h. die *affine-scaling*-Richtung verringert das Dualitätsmaß stark, so wird nur ein kleiner Wert σ gebraucht.
In der Praxis berechnet man den Centering-Parameter heuristisch mit

$$\sigma = \left(\frac{\xi_{\text{aff}}}{\xi}\right)^3.$$

Anschließend lösen wir das Gleichungssystem

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T & \Lambda \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\mu \\ \Delta\lambda \\ \Delta t \end{bmatrix} = - \begin{bmatrix} r_q \\ r_h \\ r_g \\ T\Lambda e + \Delta T_{\text{aff}}\Delta\Lambda_{\text{aff}}e - \sigma\xi e \end{bmatrix} \tag{3.14}$$

und verwenden die Lösung als Suchrichtung für die aktuelle Iterierte. Man beachte, dass das System (3.14) sowohl die Störung des KKT-Systems durch $\sigma\xi$ als auch die Korrektur im Sinne von (3.13) beinhaltet. Da die Systeme (3.12) und (3.14) die gleiche Koeffizientenmatrix besitzen, bietet es sich an, die Matrix einmal zu faktorisieren und diese Faktorisierung zur Lösung beider Systeme zu nutzen. Als letztes bestimmen wir noch die Schrittweite für die gefundene Suchrichtung $(\Delta z, \Delta\mu, \Delta\lambda, \Delta t)$, sodass die Bedingung $(\lambda, t) > 0$ erfüllt ist. Eine mögliche Wahl ist, $\alpha = \min(\alpha_\tau^t, \alpha_\tau^\lambda)$ zu setzen,

wobei

$$\begin{aligned} \alpha_\tau^t &= \max\{\alpha \in (0,1] : t + \alpha\Delta t \geq (1-\tau)t\}, \\ \alpha_\tau^\lambda &= \max\{\alpha \in (0,1] : \lambda + \alpha\Delta\lambda \geq (1-\tau)\lambda\}. \end{aligned} \tag{3.15}$$

Hier ist $\tau \in (0,1)$ (üblicherweise $\tau \approx 0.99$).

Für Mehrotras Prädiktor-Korrektor-Methode gibt es keinen formalen Konvergenzbeweis, zumindest nicht in der Form, wie er oben beschrieben ist [1]. Trotzdem hat er sich in der Praxis als sehr effizient erwiesen und wird in vielen IPM-Codes verwendet [9]. Für andere Varianten von von Innere-Punkte-Verfahren kann eine theoretische Laufzeit von $\mathcal{O}(\sqrt{n}log(1/\epsilon))$ nachgewiesen werden [10].
Wir formulieren nun auch hierfür den Algorithmus:

Algorithm 2: Prädiktor-Korrektor-Methode für QP

Input: Startpunkt $(z^0, \mu^0, \lambda^0, t^0)$ mit $(\lambda^0, t^0) > 0$.

Output: Lösung $(z, \mu, \lambda, t) = (z^k, \mu^k, \lambda^k, t^k)$ der KKT-Bedingungen (3.4).

1 Setze $k = 0$.

2 **while** *KKT-Bedingungen sind nicht hinreichend genau erfüllt* **do**

3 Löse das Gleichungssystem

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T^k & \Lambda^k \end{bmatrix} \begin{bmatrix} \Delta z^k_{\text{aff}} \\ \Delta \mu^k_{\text{aff}} \\ \Delta \lambda^k_{\text{aff}} \\ \Delta t^k_{\text{aff}} \end{bmatrix} = - \begin{bmatrix} r^k_q \\ r^k_h \\ r^k_g \\ T^k \Lambda^k e \end{bmatrix}.$$

zur Bestimmung des *affine-scaling*-Richtung $(\Delta z_{\text{aff}}, \Delta \mu_{\text{aff}}, \Delta \lambda_{\text{aff}}, \Delta t_{\text{aff}})$.

4 Berechne

$$\alpha^t_{\text{aff}} = \min(1, \min_{i:\Delta t_{\text{aff},i}<0} -\frac{t_i}{\Delta t_{\text{aff},i}}), \quad \alpha^\lambda_{\text{aff}} = \min(1, \min_{i:\Delta \lambda_{\text{aff,i}}<0} -\frac{\lambda_i}{\Delta \lambda_{\text{aff},i}}).$$

5 Bereche Dualitätsmaße

$$\xi = (t^k)^T \lambda^k / n_g, \quad \xi_{\text{aff}} = (t^k + \alpha^t_{\text{aff}} \Delta t^k_{\text{aff}})^T (\lambda^k + \alpha^\lambda_{\text{aff}} \Delta \lambda^k_{\text{aff}}) / n_g.$$

6 Berechne den Centering-Parameter mit

$$\sigma^k = \left(\frac{\xi^k_{\text{aff}}}{\xi_k} \right)^3.$$

7 Bestimme die Suchrichtung durch Lösen des Gleichungssystems

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T^k & \Lambda^k \end{bmatrix} \begin{bmatrix} \Delta z^k \\ \Delta \mu^k \\ \Delta \lambda^k \\ \Delta t^k \end{bmatrix} = - \begin{bmatrix} r^k_q \\ r^k_h \\ r^k_g \\ T^k \Lambda^k e + \Delta T^k_{\text{aff}} \Delta \Lambda^k_{\text{aff}} e - \sigma^k \xi^k e \end{bmatrix}.$$

8 Bestimme die Schrittweite $\alpha^k = \min(\alpha^t_\tau, \alpha^\lambda_\tau)$, wobei

$$\alpha^t_\tau = \max\{\alpha \in (0,1] : t^k + \alpha \Delta t^k \geq (1-\tau) t^k\},$$
$$\alpha^\lambda_\tau = \max\{\alpha \in (0,1] : \lambda^k + \alpha \Delta \lambda^k \geq (1-\tau) \lambda^k\}.$$

9 Setze $(z^{k+1}, \mu^{k+1}, \lambda^{k+1}, t^{k+1}) = (z^k, \mu^k, \lambda^k, t^k) + \alpha^k (\Delta z^k, \Delta \mu^k, \Delta \lambda^k, \Delta t^k)$.

10 Setze $k = k + 1$.

4. Cyclic Reduction

Der Hauptaufwand des Algorithmus (2) ist das Lösen der Gleichungssysteme (3.12) und (3.14) in Zeile 3 und 7. Es stellt sich also die Frage, wie diese effizient gelöst werden können. In diesem Kapitel werden wir dieses Problem für eine bestimmte Klasse von Gleichungssystem betrachten, die eine Block-Tridiagonal-Struktur in der Koeffizientenmatrix besitzen. Wir werden zeigen, dass unser MPC-Problem (2.2) bei geeigneter Anordnung der Variablen zu einem KKT-System mit genau einer solchen Struktur führt. Das Verfahren des *Cyclic Reduction* nutzt diese Struktur aus und erlaubt uns hierbei eine Laufzeitreduktion von $\mathcal{O}(N)$ auf $\mathcal{O}(log(N))$ mit paralleler Ausführung.

4.1. Herleitung

Wir betrachten das Block-Tridiagonalsystem

$$\begin{bmatrix} A_1 & C_1 & & & & \\ B_2 & A_2 & C_2 & & & \\ & B_3 & A_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & B_{N-1} & A_{N-1} & C_{N-1} \\ & & & & B_N & A_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix}, \quad N = 2^l - 1. \tag{4.1}$$

Der Algorithmus kann auf den Fall $N \neq 2^l - 1$ erweitert werden, den wir uns hier aber nicht genauer anschauen werden. Für den allgemeinen Fall verweisen wir auf [11].
Die Idee des Cyclic Reduction besteht nun darin, aus drei aufeinanderfolgenden Gleichungen zwei der fünf Unbekannten (bzw. vier am Rand) zu eliminieren [12]. Hierzu betrachte die drei Gleichungen

$$\begin{aligned} B_{i-1}x_{i-2} + A_{i-1}x_{i-1} + C_{i-1}x_i &= y_{i-1} && \text{(4.2a)} \\ B_i x_{i-1} + A_i x_i + C_i x_{i+1} &= y_i && \text{(4.2b)} \\ B_{i+1}x_i + A_{i+1}x_{i+1} + C_{i+1}x_{i+2} &= y_{i+1} && \text{(4.2c)} \end{aligned}$$

Multiplizieren wir die 1. Gleichung (4.2a) mit $-B_i A_{i-1}^{-1}$, die 3. Gleichung (4.2c) mit $-C_i A_{i+1}^{-1}$ von links und addieren beide zu der 2. Gleichung (4.2b), so erhalten wir:

$$\begin{aligned} (-B_i A_{i-1}^{-1} B_{i-1})x_{i-2} + (A_i - B_i A_{i-1}^{-1} C_{i-1} - C_i A_{i+1}^{-1} B_{i+1})x_i + (-C_i A_{i+1}^{-1} C_{i+1})x_{i+2} \\ = y_i - B_i A_{i-1}^{-1} y_{i-1} - C_i A_{i+1}^{-1} y_{i+1}, \qquad i = 2, 4, ..., N-1. \end{aligned} \tag{4.3}$$

Damit obige Berechnung auch für $i = 2$ und $i = N-1$ gilt, definieren wir $B_1 = C_0 = 0$. Transformiert man das gesamte System (4.1) nach (4.3), so erhalten wir

$$\begin{bmatrix} A_1 & C_1 & & & & & & \\ 0 & A_1^{(1)} & 0 & C_1^{(1)} & & & & \\ & B_3 & A_3 & C_3 & & & & \\ & B_2^{(1)} & 0 & A_2^{(1)} & 0 & C_2^{(1)} & & \\ & & & & \ddots & & & \\ & & & & B_{N-2} & A_{N-2} & C_{N-2} & \\ & & & & B_{\frac{N-1}{2}}^{(1)} & 0 & A_{\frac{N-1}{2}}^{(1)} & 0 \\ & & & & & & B_N & A_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{N-2} \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_1^{(1)} \\ y_3 \\ y_2^{(1)} \\ \vdots \\ y_{N-2} \\ y_{\frac{N-1}{2}}^{(1)} \\ y_N \end{bmatrix}, \tag{4.4}$$

mit

$$\left.\begin{array}{rcl} A_i^{(1)} & = & A_{2i} - B_{2i}A_{2i-1}^{-1}C_{2i-1} - C_{2i}A_{2i+1}^{-1}B_{2i+1} \\ B_i^{(1)} & = & -B_{2i}A_{2i-1}^{-1}B_{2i-1} \\ C_i^{(1)} & = & -C_{2i}A_{2i+1}^{-1}C_{2i+1} \\ y_i^{(1)} & = & y_{2i} - B_{2i}A_{2i-1}^{-1}y_{2i-1} - C_{2i}A_{2i+1}^{-1}y_{2i+1} \end{array}\right\} \quad i = 1, 2, ..., \frac{N-1}{2}.$$

Wir machen zwei wichtige Beobachtungen:

1. Die Zeilen mit geraden Indizes bilden für sich wiederum ein lineares Gleichungssystem mit Block-Tridiagonal-Struktur halber Größe:

$$\begin{bmatrix} A_1^{(1)} & C_1^{(1)} & & \\ B_2^{(1)} & A_2^{(1)} & C_2^{(1)} & \\ & & \ddots & \\ & & B_{\frac{N-1}{2}}^{(1)} & A_{\frac{N-1}{2}}^{(1)} \end{bmatrix} \begin{bmatrix} x_2 \\ x_4 \\ \vdots \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \\ \vdots \\ y_{\frac{N-1}{2}}^{(1)} \end{bmatrix}.$$

2. Die Unbekannten mit ungeraden Indizes lassen sich bei bekannten Werten der gerade indizierten Unbekannten bestimmen.

Da das reduzierte System ebenfalls eine Block-Tridiagonal-Struktur aufweist, kann gleiches Prinzip wieder angewendet, bis am Ende ein System übrig bleibt, welches nicht weiter reduzierbar ist und direkt gelöst wird. Mit der so erhaltenen Lösung können wir dann sukzessive die Lösungen der Systeme aus vorherigen Eliminationsschritten berechnen und erhalten schließlich die Lösung des Ausgangssystems (4.1).

4.2. Algorithmus im Detail

Unser Ausgangspunkt sei das System (4.1). Wir führen folgende Bezeichnung ein:

$$A^{(0)} := \begin{bmatrix} A_1 & C_1 & & & & \\ B_2 & A_2 & C_2 & & & \\ & B_3 & A_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & B_{N-1} & A_{N-1} & C_{N-1} \\ & & & & B_N & A_N \end{bmatrix}, \qquad x^{(0)} := \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix}, \qquad y^{(0)} := \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix}.$$

Für die einzelnen Komponenten gelte $A_i, B_i, C_i \in \mathbb{R}^{n\times n}, x_i \in \mathbb{R}^n, y_i \in \mathbb{R}^n, i = 1, 2, ..., N$ mit $N = 2^l - 1$ für ein $l \in \mathbb{N}$. $A^{(0)}$ hat somit Dimension $(Nn) \times (Nn)$.
Das *Cyclic Reduction* lässt sich in zwei Phasen teilen: einer *Reduktionsphase*, in der das Ausgangssystem so oft reduziert wird, bis man ein dichtbesetztes System hat, und einer *Substitutionsphase* zur Bestimmung der Lösung $x^{(0)}$ mit Hilfe vorangegangener Lösungen.

Reduktionsphase

Durch Elimination der Unbekannten mit ungeraden Indizes mit Hilfe der Unbekannten gerader Indizes (vgl. (4.2)) erhalten wir das reduzierte System $A^{(1)}x^{(1)} = y^{(1)}$ halber Größe. Rekursives Anwenden dieses Verfahrens liefert uns eine Folge von Subproblemen

$$A^{(0)}x^{(0)} = y^{(1)} \to A^{(1)}x^{(1)} = y^{(1)} \to ... \to A^{(i)}x^{(i)} = y^{(i)} \to ... \to A^{(l-1)}x^{(l-1)} = y^{(l-1)},$$

$$A^{(i)} := \begin{bmatrix} A_1^{(i)} & C_1^{(i)} & & & \\ B_2^{(i)} & A_2^{(i)} & C_2^{(i)} & & \\ & \ddots & \ddots & \ddots & \\ & & B_{N_i-1}^{(i)} & A_{N_i-1}^{(i)} & C_{N_i-1} \\ & & & B_{N_i}^{(i)} & A_{N_i}^{(i)} \end{bmatrix} \in \mathbb{R}^{(N_i n)\times(N_i n)},$$

$$x^{(i)} := \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_{N_i-1}^{(i)} \\ x_{N_i}^{(i)} \end{bmatrix} \in \mathbb{R}^{(N_i n)}, \qquad y^{(i)} := \begin{bmatrix} y_1^{(i)} \\ y_2^{(i)} \\ \vdots \\ y_{N_i-1}^{(i)} \\ y_{N_i}^{(i)} \end{bmatrix} \in \mathbb{R}^{(N_i n)}, \qquad N_i = 2^{l-i} - 1.$$

Im Allgemeinen erhalten wir ausgehend von $A^{(i)}x^{(i)} = y^{(i)}$ das nächste Problem

$A^{(i+1)}x^{(i+1)} = y^{(i+1)}$ durch

$$\begin{aligned} A_j^{(i+1)} &= A_{2j}^{(i)} - B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1}C_{2j-1}^{(i)} - C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1}B_{2j+1}^{(i)}, \\ B_j^{(i+1)} &= -B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1}B_{2j-1}^{(i)}, \\ C_j^{(i+1)} &= -C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1}C_{2j+1}^{(i)}, \\ y_j^{(i+1)} &= y_{2j}^{(i)} - B_{2j}^{(i)}(A_{2j-1}^{(i)})^{-1}y_{2j-1}^{(i)} - C_{2j}^{(i)}(A_{2j+1}^{(i)})^{-1}y_{2j+1}^{(i)}, \end{aligned} \tag{4.5}$$

für $i = 0, 1, ..., l-2, \quad j = 1, ..., N_{i+1}$.
Nach $l-1 = \Theta(log(N))$ Reduktionsschritten erhalten wir ein dichtbesetztes Gleichungssystem

$$A^{(l-1)}x^{(l-1)} = y^{(l-1)} \Leftrightarrow A_1^{(l-1)}x_1^{(l-1)} = y_1^{(l-1)},$$

welches wir nicht weiter reduzieren und mit einem direkten Verfahren für dichtbesetzte Systeme (z.B. über einer Cholesky-Zerlegung) lösen können.

Substitutionsphase

Ausgehend von einer Lösung $x^{(i+1)}$ des Gleichungssystems $A^{(i+1)}x^{(i+1)} = y^{(i+1)}$ kann die Lösung $x^{(i)}$ des Gleichungssystems $A^{(i)}x^{(i)} = y^{(i)}$ durch Rücksubstitution berechnet werden. Der Lösungsvektor $x^{(i+1)}$ entspricht genau dem Teilvektor von $x^{(i)}$, der nur die geraden Indizes beinhaltet. Einsetzen in die entsprechende Gleichung und ausrechnen ergibt:

$$\begin{aligned} x_{2j}^{(i)} &= x_j^{(i+1)}, && j = 1, 2, ..., N_{i+1}, \\ x_1^{(i)} &= (A_1^{(i)})^{-1}(y_1^{(i)} - C_1^{(i)}x_2^{(i)}), && \\ x_{2j-1}^{(i)} &= (A_{2j-1}^{(i)})^{-1}(y_{2j-1}^{(i)} - B_{2j-1}^{(i)}x_{2j-2}^{(i)} - C_{2j-1}^{(i)}x_{2j}^{(i)}), && j = 2, 3, ..., N_{i+1}, \\ x_{N_i}^{(i)} &= (A_{N_i}^{(i)})^{-1}(y_{N_i}^{(i)} - B_{N_i}^{(i)}x_{N_i-1}^{(i)}), && \end{aligned} \tag{4.6}$$

für $i = l-2, l-3, ..., 0$.
Nach $l-1 = \Theta(log(N))$ Substitutionsschritten erhalten wir damit die Lösung $x = x^{(0)}$ des Gleichungssystems (4.1).
Beachte, dass sich sowohl (4.5) als auch (4.6) für Parallelisierung eignen. Der komplette Algorithmus kann zusammengefasst werden als:

Algorithm 3: Cyclic Reduction

Input: Block-Tridiagonal-System gegeben durch
$A^{(0)} = \left[A_1^{(0)}, ..., A_N^{(0)}\right], B^{(0)} = \left[B_2^{(0)}, ..., B_N^{(0)}\right], C^{(0)} = \left[C_1^{(0)}, ..., C_{N-1}^{(0)}\right]$,
$y^{(0)} = \left[y_1^{(0)}, ..., y_N^{(0)}\right]^T, l \in \mathbb{N}$ (wobei $N = 2^l - 1$).

Output: Lösung $x^{(0)} = \left[x_1^{(0)}, ..., x_N^{(0)}\right]^T$ des Gleichungssystems (4.1).

for $i = 1, 2, ..., l-1$ **do**
  $N_i = 2^{l-i} - 1$;
  for $j = 1, 2, ..., N_i$ **do in parallel**
    $A_j^{(i)} = A_{2j}^{(i-1)} - B_{2j}^{(i-1)}(A_{2j-1}^{(i-1)})^{-1}C_{2j-1}^{(i-1)} - C_{2j}^{(i-1)}(A_{2j+1}^{(i-1)})^{-1}B_{2j+1}^{(i-1)}$;
    $B_j^{(i)} = -B_{2j}^{(i-1)}(A_{2j-1}^{(i-1)})^{-1}B_{2j-1}^{(i-1)}$,;
    $C_j^{(i)} = -C_{2j}^{(i-1)}(A_{2j+1}^{(i-1)})^{-1}C_{2j+1}^{(i-1)}$;
    $y_j^{(i)} = y_{2j}^{(i-1)} - B_{2j}^{(i-1)}(A_{2j-1}^{(i-1)})^{-1}y_{2j-1}^{(i-1)} - C_{2j}^{(i-1)}(A_{2j+1}^{(i-1)})^{-1}y_{2j+1}^{(i-1)}$;
Berechne x_1^{l-1} als Lösung des GLS $A_1^{(l-1)}x = y_1^{(l-1)}$;
for $i = l-2, l-3, ..., 0$ **do**
  $N_i = 2^{l-i} - 1$;
  $N_{i+1} = 2^{l-i-1} - 1$;
  $x_2^i = x_1^{(i+1)}$;
  $x_1^{(i)} = (A_1^{(i)})^{-1}(y_1^{(i)} - C_1^{(i)}x_2^{(i)})$;
  for $j = 2, 3, ..., N_{i+1}$ **do in parallel**
    $x_{2j-2}^{(i)} = x_{j-1}^{(i+1)}$;
    $x_{2j}^{(i)} = x_j^{(i+1)}$;
    $x_{2j-1}^{(i)} = (A_{2j-1}^{(i)})^{-1}(y_{2j-1}^{(i)} - B_{2j-1}^{(i)}x_{2j-2}^{(i)} - C_{2j-1}^{(i)}x_{2j}^{(i)})$;
  $x_{N_i-1}^{(i)} = x_{N_{i+1}}^{i+1}$;
  $x_{N_i}^{(i)} = (A_{N_i}^{(i)})^{-1}(y_{N_i}^{(i)} - B_{N_i}^{(i)}x_{N_i-1}^{(i)})$;

Es soll nur kurz angemerkt werden, dass der sequentielle Aufwand des Algorithmus etwa zweimal so hoch ist wie für das direkte Lösen des Gleichungssystems mit Cholesky, aber er eine parallele Laufzeit von $\mathcal{O}(log(N))$ hat [6].

4.3. Analytische Wohldefiniertheit von Cyclic Reduction

Wie der Algorithmus (3) zeigt, benötigt man an mehreren Stellen die Inversen $(A_j^{(i)})^{-1}$. Es stellt sich daher die Frage nach der Wohldefiniertheit, d.h. existieren die $(A_j^{(i)})^{-1}$ überhaupt?

Wir betrachten hierzu das Gleichungssystem $Ax = y$,

$$A = \begin{bmatrix} A_1 & C_1 & & & \\ B_2 & A_2 & C_2 & & \\ & \ddots & \ddots & \ddots & \\ & & B_{N-1} & A_{N-1} & C_{N-1} \\ & & & B_N & A_N \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix}, \quad N = 2^l - 1.$$

Sei P_1 eine Permutationsmatrix, so dass

$$P_1 \left[1, 2, ..., N\right]^T = \left[1, 3, ..., N | 2, 4, ..., N-1\right]^T,$$

d.h. P_1 ordnet die Indizes so an, dass die geraden Indizes auf die ungeraden Indizes folgen. Die permutierte Matrix $P_1 A P_1^T$ sieht dann folgendermaßen aus:

$$P_1 A P_1^T = \left[\begin{array}{ccccc|ccccc} A_1 & & & & & C_1 & & & & \\ & A_3 & & & & B_3 & C_3 & & & \\ & & \ddots & & & & \ddots & \ddots & & \\ & & & A_{N-2} & & & & B_{N-2} & C_{N-2} & \\ & & & & A_N & & & & & B_N \\ \hline B_2 & C_2 & & & & A_2 & & & & \\ & B_4 & C_4 & & & & A_4 & & & \\ & & \ddots & \ddots & & & & \ddots & & \\ & & & B_{N-1} & C_{N-1} & & & & & A_{N-1} \end{array}\right] \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} \left\lfloor \frac{N}{2} \right\rfloor + 1 \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \left\lfloor \frac{N}{2} \right\rfloor \end{matrix}$$

$$\underbrace{\qquad\qquad}_{\left\lfloor \frac{N}{2} \right\rfloor + 1} \qquad \underbrace{\qquad\qquad}_{\left\lfloor \frac{N}{2} \right\rfloor}$$

Wendet man nun die Block-Gauß-Elimination partiell auf die ersten $\left\lfloor \frac{N}{2} \right\rfloor + 1$ Blockspalten an (vlg. hierzu auch [13]), so erhalten wir:

$$P_1 A P_1^T \quad \xrightarrow{\text{Gauß}} \quad \left[\begin{array}{ccccc|ccccc} A_1 & & & & & C_1 & & & & \\ & A_3 & & & & B_3 & C_3 & & & \\ & & \ddots & & & & \ddots & \ddots & & \\ & & & A_{N-2} & & & & B_{N-2} & C_{N-2} & \\ & & & & A_N & & & & & B_N \\ \hline & & & & & A_1^{(1)} & C_1^{(1)} & & & \\ & & & & & B_2^{(1)} & A_2^{(1)} & C_2^{(1)} & & \\ & & & 0 & & & & \ddots & & \\ & & & & & & & B_{N_1}^{(1)} & A_{N_1}^{(1)} \end{array}\right],$$

wobei $A_i^{(1)}, B_i^{(1)}, C_i^{(1)}$ definiert sind wie in (4.5).
Ein Schritt der Reduktionsphase entspricht daher der partiellen Block-Gauß-Elimination angewendet auf die ersten $\left\lfloor \frac{N}{2} \right\rfloor + 1$ der Matrix $P_1 A P_1^T$, was nichts anderes bedeutet als auf die geraden Zeilen der Matrix A.
Die lezten $\left\lfloor \frac{N}{2} \right\rfloor$ Zeilen und Spalten bilden erneut eine Block-Tridiagonal-Matrix, wir

wenden das gleiche Prinzip nochmals an: Sei P_2 eine Permutationsmatrix, die nun die ersten $\left\lfloor \frac{N}{2} \right\rfloor + 1$ Indizes fest lässt und die letzten $\left\lfloor \frac{N}{2} \right\rfloor$ umordnet, so dass

$$\begin{aligned} P_2 P_1 \left[1, 2, ..., N\right]^T &= P_2 \left[1, 3, ..., N | 2, 4, ..., N-1\right]^T \\ &= \left[1, 2, ..., N | 2, 6, ..., N-1 | 4, 8, ...N-3\right]^T . \end{aligned}$$

Induktives Fortsetzen liefert uns eine Permutationsmatrix

$$P := P_{l-1} P_{l-2} ... P_2 P_1,$$

welche die Indizes so sortiert, dass zuerst die ungeraden Vielfachen von 2^0, dann die ungeraden Vielfachen von 2^1, dann von 2^2, usw. folgen [3].

Beispiel: Für $N = 2^4 - 1$ heißt dies konkret:
$P \left[1, 2, ..., 15\right]^T = \left[1, 3, 5, 9, 11, 13, 15 | 2, 6, 10, 14 | 4, 12 | 8\right]^T .$

Wir fassen zusammen: Cyclic Reduction auf A ist identifizierbar mit der Block-Gauß-Elimination ohne (Block-)Pivotierung auf PAP^T. Die Reduktionsphase entspricht dem Erzeugen der oberen Block-Dreiecksmatrix, die Substitutionsphase dem sukzessiven Rückwärtseinsezten zur Bestimmung der Unbekannten von $P \left[x_1, ..., x_N\right]^T$ von unten nach oben. Mit diesen Beobachtungen lassen sich Aussagen über die Block-Gauß-Elimination auf das Cyclic Reduction übertragen [3].
Wir zeigen, dass die Block-Gauß-Elimination für allgemeine positiv definite Blockmatrizen ohne Pivot durchführbar ist. Insbesondere für positiv definite Block-Tridiagonal-Matrizen ist Cyclic Reduction damit wohldefiniert.
Wir geben kurz einen Algorithmus der Block-Gauß-Elimination an, um die Notationen im nachfolgenden Satz besser verstehen zu können.

Algorithm 4: Block-Gauß-Elimination ohne Block-Spaltenpivotierung

Input: Block-Matrix $A = \begin{bmatrix} A_{11} & \dots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \dots & A_{NN} \end{bmatrix}, A_{ij} \in \mathbb{R}^{n \times n}, i, j = 1, ..., N.$

Output: $A^{(N)}$ ist rechte obere Block-Dreiecksmatrix.

1 $A^{(1)} = A$;
2 **for** *k=1,...,N-1* **do**
3 $M_k = \begin{bmatrix} I & & & & \\ & \ddots & & & \\ & & I & & \\ & & -T_{k+1}^{(k)} & I & \\ & & \vdots & & \ddots \\ & & -T_N^{(k)} & & I \end{bmatrix}, \quad T_i^{(k)} = A_{i,k}^{(k)} (A_{kk}^{(k)})^{-1}, \quad i = k+1, ...N;$
4 $A^{(k+1)} = M_k A^{(k)}$;

Die Multiplikation von links mit M_k lässt die ersten k Blockzeilen unverändert. Für $i = k+1, ..., N$ erhält man die neue i-te Blockzeile durch Subtraktion der mit $T_i^{(k)}$ multiplizierten k-ten Blockzeile von der alten i-ten Blockzeile.
Der Beweis des folgenden Satzes ist angelehnt an den Beweis für die klassische Gauß-Elimination aus [14], lässt sich aber ohne Probleme auf Block-Matrizen übertragen.

Satz 1. *Sei*

$$A = \begin{bmatrix} A_{11} & \dots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \dots & A_{NN} \end{bmatrix}$$

eine Blockmatrix mit $A_{ij} \in \mathbb{R}^{n\times n}, i, j = 1, ..., N$. *Sei*

$$\det \begin{bmatrix} A_{11} & \dots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \dots & A_{kk} \end{bmatrix} \neq 0, \quad \forall k = 1, ..., N. \tag{4.7}$$

Sei $A_{kk}^{(k)}$ *der Eintrag in der* k*-ten Blockzeile und Blockspalte der Matrix* $A^{(k)}$ *nach dem* $(k-1)$*-ten Eliminationsschritt (vgl. Algorithmus* (4)*).*
Dann gilt: Die Block-Gauß-Elimination ist wegen der Invertierbarkeit von $A_{kk}^{(k)}$ *für* $k = 1, ..., N$ *ohne Spaltenpivotisierung durchführbar.*

Beweis. Nach Vorausssetzung (4.7) ist $\det A_{11} \neq 0$, also A_{11} invertierbar, d.h. der 1. Schritt der Elimination für $k = 1$ ist wohldefiniert und die Matrix hat folgende Form:

$$A^{(2)} = \begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} & \dots & A_{1N}^{(2)} \\ 0 & A_{22}^{(2)} & \dots & A_{2N}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & A_{N2}^{(2)} & \dots & A_{NN}^{(2)} \end{bmatrix}.$$

Insbesondere ist

$$\begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} & \dots & A_{1k}^{(2)} \\ 0 & A_{22}^{(2)} & \dots & A_{2k}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & A_{k2}^{(2)} & \dots & A_{kk}^{(2)} \end{bmatrix} = \begin{bmatrix} I & & & \\ -T_2^{(1)} & I & & \\ \vdots & & \ddots & \\ -T_N^{(1)} & & & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1k} \\ A_{21} & A_{22} & \dots & A_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k1} & A_{k2} & \dots & A_{kk} \end{bmatrix}, \quad k = 2, ..., N \tag{4.8}$$

und daher

$$\det A_{11}^{(2)} \cdot \det \begin{bmatrix} A_{22}^{(2)} & \dots & A_{2k}^{(2)} \\ \vdots & \ddots & \vdots \\ A_{k2}^{(2)} & \dots & A_{kk}^{(2)} \end{bmatrix} = \det \begin{bmatrix} A_{11}^{(2)} & A_{12}^{(2)} & \dots & A_{1k}^{(2)} \\ 0 & A_{22}^{(2)} & \dots & A_{2k}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & A_{k2}^{(2)} & \dots & A_{kk}^{(2)} \end{bmatrix}$$
$$\overset{(4.8)}{=} 1 \cdot \det \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1k} \\ A_{21} & A_{22} & \dots & A_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k1} & A_{k2} & \dots & A_{kk} \end{bmatrix} \neq 0.$$

Wegen $\det A_{11}^{(2)} = \det A_{11} \neq 0$ folgt

$$\det \begin{bmatrix} A_{22}^{(2)} & \dots & A_{2k}^{(2)} \\ \vdots & \ddots & \vdots \\ A_{k2}^{(2)} & \dots & A_{kk}^{(2)} \end{bmatrix} \neq 0, \quad k = 2, ..., N.$$

Die Voraussetzung (4.7) gilt somit auch für das reduzierte System. Durch Induktion folgt die Behauptung. □

Satz 2. *Sei*

$$A = \begin{bmatrix} A_{11} & \dots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \dots & A_{NN} \end{bmatrix}, \quad A_{ij} \in \mathbb{R}^{n \times n}, i, j = 1, 2, ..., N.$$

eine positiv definite Blockmatrix. Dann gilt:

$$\det \begin{bmatrix} A_{11} & \dots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \dots & A_{kk} \end{bmatrix} \neq 0, \quad k = 1, ..., N.$$

Beweis. Bezeiche $A^k = \begin{bmatrix} A_{11} & \dots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \dots & A_{kk} \end{bmatrix}$.
Sei $x = [x_1, x_2, ..., x_k, 0, ..., 0]^T \in \mathbb{R}^{kn}, x \neq 0$. Wegen der positiven Definitheit von A folgt:

$$0 < x^T A x = \tilde{x}^T A^k \tilde{x} \quad \text{mit} \quad \tilde{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix}.$$

Somit sind alle $A^k, k = 1, ..., N$ positiv definit und somit $\det A^k \neq 0$. □

5. Anwendung auf MPC

Zur Lösung des MPC-Problems (2.2) wollen wir die Innere-Punkte-Verfahren verwenden, die in Kapitel 3.1 beschrieben wurden. Das Gleichungssystem in jeder Iteration soll wiederum mit der Methode des Cyclic Reduction gelöst werden. Wir werden zeigen, dass sich das Gleichungssystem durch Transformationen auf Block-Tridiagonal-Struktur bringen lässt, sodass Cyclic Reduction angewendet werden kann.

5.1. Transformation der Gleichungssysteme

Die nachfolgenden Transformationen orientieren sich an [15].
Wir betrachten hierzu zunächst das KKT-System des allgemeinen QPs (2.1) in der Variante (3.4)

$$\begin{bmatrix} Q & H^T & G^T & 0 \\ H & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & T & \Lambda \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\mu \\ \Delta\lambda \\ \Delta t \end{bmatrix} = - \begin{bmatrix} r_q \\ r_h \\ r_g \\ r_{comp} \end{bmatrix}. \tag{5.1}$$

Die spezielle Struktur erlaubt es uns, (5.1) in eine kompaktere Form zu transformieren. Da Λ eine Diagonalmatrix mit strikt positiven Elementen ist, ist Λ invertierbar, wir können Δt aus (5.1) eliminieren mit $\Delta t = \Lambda^{-1}(-r_{comp} - T\Delta\lambda)$ und erhalten

$$\begin{bmatrix} Q & H^T & G^T \\ H & 0 & 0 \\ G & 0 & -\Lambda^{-1}T \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\mu \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -r_q \\ -r_h \\ -r_g + \Lambda^{-1} r_{comp} \end{bmatrix}. \tag{5.2}$$

Mit der gleichen Argumentation ist auch T invertierbar und wir können $\Delta\lambda$ aus (5.2) eliminieren mit $\Delta\lambda = -\Lambda T^{-1}(-r_g + \Lambda^{-1} r_{comp} - G\Delta z)$, wir erhalten das sogenannte *augmented system* [15]:

$$\begin{bmatrix} Q + G^T\Lambda T^{-1} G & H^T \\ H & 0 \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\mu \end{bmatrix} = \begin{bmatrix} -\tilde{r}_q \\ -r_h \end{bmatrix}, \qquad \tilde{r}_q = r_q - G^T\left(-\Lambda T^{-1} r_g + T^{-1} r_{comp}\right). \tag{5.3}$$

Unter der Annahme, dass Q positiv definit ist, ist auch $Q + G^T\Lambda T^{-1}G$ positiv definit, denn:

1. ΛT^{-1} ist positiv definit, da Diagonalmatrix mit positiven Einträgen.
2. $G^T\Lambda T^{-1}G$ ist positiv semidefinit, denn mit $x \neq 0$ gilt

$$x^T(G^T\Lambda T^{-1}G)x = (Gx)^T\Lambda T^{-1}(Gx) \geq 0.$$

($\geq$ statt $>$, da $Gx = 0$ sein kann.)

3. $Q + G^T \Lambda T^{-1} G$ positiv definit, da die Summe einer positiv definiten Matrix mit einer positiv semidefiniten Matrix wieder positiv definit ist:

$$x^T(Q + G^T \Lambda T^{-1} G)x = \underbrace{x^T Q x}_{>0} + \underbrace{x^T (G^T \Lambda T^{-1} G)x}_{\geq 0} > 0.$$

Damit ist auch die letzte Elimination von Δz mit $\Delta z = (Q + G^T \Lambda T^{-1} G)^{-1}(-\tilde{r}_q - H^T \Delta\mu)$ möglich, da aufgrund der positiven Definitheit die Matrix $Q + G^T \Lambda T^{-1} G$ invertierbar ist. Multiplikation mit (-1) bringt uns schließlich auf die *normal-equations* Form

$$\begin{aligned} H(Q + G^T \Lambda T^{-1} G)^{-1} H^T \Delta\mu &= \tilde{r}_h, \\ \tilde{r}_h &= r_h - H(Q + G^T \Lambda T^{-1} G)^{-1} \tilde{r}_q. \end{aligned} \tag{5.4}$$

In der Praxis werden die System (5.3) oder (5.4) gelöst und die Unbekannten $\Delta\lambda, \Delta t$ bzw. $\Delta z, \Delta\lambda, \Delta t$ durch Rücksubstitution ermittelt.
Wir werden nachfolgend sehen, dass für das Cyclic Reduction, angewendet für unser MPC-Problem, es unwesentlich ist, ob wir das *augmented system* oder die *normal equations* Form verwenden.

5.2. Übertragung auf MPC

Wir betrachten zuerst die *normal equations* Form (5.4)

$$\begin{aligned} H(Q + G^T \Lambda T^{-1} G)^{-1} H^T \Delta\mu &= \tilde{r}_h, \\ \tilde{r}_h &= r_h - H(Q + G^T \Lambda T^{-1} G)^{-1} \tilde{r}_q. \end{aligned}$$

für unser MPC-Problem, wobei die Matrizen und Vektoren definiert sind wie in (2.3). Ausgerechnet und ausgeschrieben ergibt sich für die Koeffizientenmatrix:

$$H(Q + G^T \Lambda T^{-1} G)^{-1} H^T = \begin{bmatrix} Y_{11} & Y_{12} & & & \\ Y_{12}^T & Y_{22} & Y_{23} & & \\ & \ddots & \ddots & \ddots & \\ & & Y_{N-2,N-1}^T & Y_{N-1,N-1} & Y_{N-1,N} \\ & & & Y_{N-1,N}^T & Y_{N,N} \end{bmatrix} \tag{5.5}$$

mit

$$Y_{i,i} = C_{i-1} \Phi_{i-1}^{-1} C_{i-1}^T + E \Phi_i E^T, \qquad i = 1, 2, ..., N, \tag{5.6a}$$

$$Y_{i,i+1} = -C_i \Phi_i^{-1} E^T, \qquad i = 1, 2, ..., N-1, \tag{5.6b}$$

$$\Phi_i = B_i + D_i^T \Lambda T^{-1} D_i \qquad i = 0, 1, ..., N. \tag{5.6c}$$

Die Matrix ist symmetrisch und positiv definit, da

1. $Q = \begin{bmatrix} B_0 & & & & \\ & B_1 & & & \\ & & \ddots & & \\ & & & B_{N-1} & \\ & & & & B_N \end{bmatrix}$ ist positiv definit, da wir angenommen haben, dass alle $B_i, i = 0, ..., N$ positiv definit sind.

2. Nach vorherigen Überlegungen ist somit $Q + G^T \Lambda T^{-1} G$ positiv definit und da die Inverse einer positiv definiten Matrix wieder positiv definit ist [16], auch $(Q + G^T \Lambda T^{-1} G)^{-1}$.

3. Die Matrix $H^T = \begin{bmatrix} C_0^T & & & \\ -E^T & C_1^T & & \\ & \ddots & \ddots & \\ & & -E^T & C_{N-1}^T \\ & & & -E^T \end{bmatrix}$ hat vollen Spaltenrang, unabhängig von den $C_i^T, i = 0, 1, ..., N-1$, aufgrund der Struktur von E^T ($E^T = \begin{bmatrix} I & 0 \end{bmatrix}^T$). Multiplikation von links mit H und rechts mit H^T erhält die positive Definitheit [16].

Die *normal equations* Form hat somit eine positiv definite Koeffizientenmatrix, und damit - wie vorher in Kapitel 4.2 gezeigt - ist das Cyclic Reduction anwendbar.
Schauen wir uns nun das *augmented system* (5.3) an. Es ergibt sich:

$$\left[\begin{array}{ccccc|ccccc} \Phi_0 & & & & & C_0^T & & & & \\ & \Phi_1 & & & & -E^T & C_1^T & & & \\ & & \ddots & & & & \ddots & \ddots & & \\ & & & \Phi_{N-1} & & & & -E^T & C_{N-1}^T & \\ & & & & \Phi_N & & & & & -E^T \\ \hline C_0 & -E & & & & & & & & \\ & C_1 & -E & & & & & & & \\ & & \ddots & \ddots & & & & 0 & & \\ & & & C_{N-1} & -E & & & & & \end{array}\right] \left[\begin{array}{c} \Delta z_0 \\ \Delta z_1 \\ \vdots \\ \Delta z_{N-1} \\ \Delta z_N \\ \hline \Delta \mu_0 \\ \Delta \mu_1 \\ \vdots \\ \Delta \mu_{N-1} \end{array}\right] = \left[\begin{array}{c} -\tilde{r}_{q,0} \\ -\tilde{r}_{q,1} \\ \vdots \\ -\tilde{r}_{q,N-1} \\ -\tilde{r}_{q,N} \\ \hline -r_{h,0} \\ -r_{h,1} \\ \vdots \\ -r_{h,N-1} \end{array}\right]$$

Ordnen wir die Variablen um als $[\Delta z_0, \Delta \mu_0, \Delta z_1, \Delta \mu_1, ..., \Delta z_{N-1}, \Delta \mu_{N-1}, \Delta z_N]^T$ und entsprechend die Zeilen und Spalten, so erhalten wir:

$$\begin{bmatrix} \Phi_0 & C_0^T & & & & & & \\ C_0 & 0 & -E & & & & & \\ & -E^T & \Phi_1 & C_1^T & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & -E^T & \Phi_{N-1} & C_{N-1}^T & & \\ & & & & C_{N-1} & 0 & -E & \\ & & & & & -E^T & \Phi_N & \end{bmatrix} \begin{bmatrix} \Delta z_0 \\ \Delta \mu_0 \\ \Delta z_1 \\ \vdots \\ \Delta z_{N-1} \\ \Delta \mu_{N-1} \\ \Delta z_N \end{bmatrix} = \begin{bmatrix} -\tilde{r}_{q,0} \\ -r_{h,0} \\ -\tilde{r}_{q,1} \\ \vdots \\ -\tilde{r}_{q,N-1} \\ -r_{h,N-1} \\ -\tilde{r}_{q,N} \end{bmatrix}. \tag{5.7}$$

Dieses System ist indefinit, denn:

1. Das *augmented system* (5.3) ist indefinit:

$$\begin{bmatrix} x^T & y^T \end{bmatrix} \begin{bmatrix} Q + G^T \Lambda T^{-1} G & H^T \\ H & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x^T (Q + G^T \Lambda T^{-1} G) x + y^T H x + x^T H^T y$$
$$= x^T (Q + G^T \Lambda T^{-1} G) x + 2 y^T H x.$$

   Wählen wir x so, dass $Hx \neq 0$, dann kann durch Wahl von y der zweite Summand $2y^T Hx$ beliebig groß bzw. beliebig klein werden. Der erste Summand ist unabhängig von y. Somit kann die rechte Seite sowohl negativ als auch positiv werden.

2. Das System (5.7) geht durch Multiplikation mit einer Permutationsmatrix von links und rechts aus (5.3) hervor. Die Indefinitheit bleibt hierbei erhalten.

Damit können wir also zunächst noch keine Aussagen über die Anwendbarkeit des Cyclic Reduction auf (5.7) aussagen. Man könnte sogar denken, dass die Nulleinträge in der Diagonalen uns Probleme machen könnten. Führen wir aber nur einen Schritt des Cyclic Reduction aus nach (4.3), welcher genau die ungeraden Indizes eliminiert (diese entsprechen hier aber den $\Delta z_i, i = 0, 1, ..., N$), so erhalten wir wieder unsere *normal equations* Form (5.4) mit der Koeffizientenmatrix (5.5) (nach Multiplikation mit (-1)). Die Umformung vom *augmented system* in die *normal equations* Form entspricht also genau einem Schritt des Cyclic Reduction.

6. Implementation und Tests

Implementation

Eine Testimplementation in MATLAB der Prädiktor-Korrektor-Methode und des Cyclic Reduction für Probleme der Form (2.2) findet sich im Anhang (A). Wir verwenden im IPM-Code als einfache Fehlerfunktion das Maximum der euklidischen Norm über die Residuen [1]

$$E(z, \mu, \lambda, t) = \max\{\|r_q\|_2, \|r_g\|_2, \|r_h\|_2, \|r_{comp}\|_2\}. \tag{6.1}$$

Der Algorithmus terminiert, wenn die Fehlerfunktion eine Toleranzschranke $\epsilon = 10^{-6}$ unterschreitet oder aber die maximale Anzahl an Iterationen überschritten wird. Zur Schrittweitenbestimmung (3.15) verwenden wir $\tau = 0.99$. Der Algorithmus arbeitet mit dem transformiertem *augmented system* (5.7) und verwendet zum Lösen dieses Gleichungssystems die Methode des Cyclic Reduction. Die Faktorisierung der Matrix im Prädiktor-Schritt wird im Korrektor-Schritt wiederverwendet. Die Testimplementation des Cyclic Reduction ist sequentiell und berücksichtigt nur den Fall $N = 2^l - 1$, daher werden wir unsere Testbeispiele entsprechend anpassen. Zur Invertierung der Matrizen in der Reduktionsphase verwenden wir die Matlab-Funktion *inv*, die sicherlich nicht optimal ist.

Test

Um zu vergewissern, dass die implementierte Prädiktor-Korrektor-Methode funktioniert, testen wir den Code an mehreren Beispielen. Das Testproblem sei (2.2)

$$\begin{aligned} \min_z \quad & \sum_{k=1}^{N} (\frac{1}{2} z_k^T B_k z_k + b_k^T z_k) & \\ \text{s.t.} \quad & E z_{k+1} = C_k z_k + c_k & \forall k \in \{1, ..., N-1\}, \\ & D_k z_k \leq d_k & \forall k \in \{1, ..., N\}, \end{aligned}$$

wobei $z_k = [x_k, u_k]^T \in \mathbb{R}^{n_x + n_u} = \mathbb{R}^{n_z}, B_k \in \mathbb{R}^{n_z \times n_z}, C_k \in \mathbb{R}^{n_x \times n_z}, D_k \in \mathbb{R}^{n_d \times n_z}, E = \begin{bmatrix} I & 0 \end{bmatrix} \in \mathbb{R}^{n_x \times n_z}, b_k \in \mathbb{R}^{n_z}, c_k \in \mathbb{R}^{n_x}, d_k \in \mathbb{R}^{n_d}$. Die Matrizen B_k seien symmetrisch, positiv definit.
Die Matrizen und Vektoren werden mit der in Matlab vorimplementierten Funktion *rand* zufallsgeneriert. Unser Algorithmus erhält als Eingabe die generierten Matrizen und Vektoren, die Dimensionen der Zustandsvariablen und Steuerungen n_x und n_u, eine natürliche Zahl $l \in \mathbb{N}$ ($N = 2^l$, beim Cyclic Reduction Schritt ergibt sich dann eine Dimension von $\tilde{N} = 2^{l+1} - 1$, vgl. (5.7)), Startvektoren $(z_0, \mu_0, \lambda_0, t_0)^T$ mit $(\lambda_0, t_0) \geq 0$

MPC Problem				Iterationszahl	
n_x	n_u	n_d	N	ipmPC_MPC	quadprog
3	2	5	16	5.41	5.64
3	2	5	64	6.14	7.05
3	2	5	128	7.46	7.63
6	5	7	16	5.25	6.69
6	5	7	64	6.66	7.84
6	5	7	128	8.22	8.27
8	12	15	16	5.54	7.57
8	12	15	64	11.2857	8.21
8	12	15	128	17.7021	8.62

Abbildung 6.1.: durchschnittliche Iterationszahl nach 100 Durchläufen

und die maximale Iterationszahl $maxIter$. Bei allen Testbeispielen setzen wir $z_0 = (0, ..., 0)^T \in \mathbb{R}^{N \cdot n_z}, \mu_0 = (0, ..., 0)^T \in \mathbb{R}^{(N-1) \cdot n_x}$, $\lambda_0 = t_0 = (1, ..., 1)^T \in \mathbb{R}^{N \cdot n_d}$ und $maxIter = 100$.
Wir vergleichen unsere Ergebnisse mit Matlabs *quadprog* mit Default-Konfiguration und *interior-point-convex* Solver. Da Cyclic Reduction nur sequentiell implementiert wurde, verzichten wir auf einen Vergleich der Laufzeiten und werden nur die Anzahl der gebrauchten Iterationen gegenüberstellen. Alle Berechnungen wurden mit der Version MATLAB R2013a gemacht. Abbildung (6.1) zeigt die Ergebnisse im Durchschnitt für verschiedene n_x, n_u, n_d mit jeweils $N = 16, N = 64, N = 128$ nach 100 Durchläufen.
Wir beobachten, dass die Testimplementation in etwa so viele Iterationen benötigt wie quadprog. Dennoch kam es bei unserem Code zu einigen numerischen Schwierigkeiten in Ausnahmefällen (*Matrix nahe an der Singulariät bzw. schlecht konditionirt*). Die vorletzte und letzte Zeile der Tabelle zeigen einen deutlichen Unterschied in der Iterationszahl, die vor allem dadurch zustande kam, dass einige Ausnahmefälle die durchschnittliche Iterationszahl bei der Testimplementation in die Höhe trieben. Es sei trotzdem angemerkt, dass die Testimplementation in den meisten Fällen gut funktioniert.
Auffällig ist auch die relativ geringe Anzahl an Iterationen bis zur Termination, auch wenn die Problemgröße stark ansteigt (von $N = 16$ auf $N = 128$). Jeder Iterationsschritt kann einen signifikanten Fortschritt zur Lösung machen, wobei jeder einzelne Schritt teuer sein kann [1]. Um diese Eigenschaft zu veranschaulichen, plotten wir den Wert der Fehlerfunktion in jeder Iteration für das oben angegebene Problem mit $n_x = 3, n_u = 2, n_d = 5, N = 128$ für fünf zufällig generierte Instanzen in (6.2). Jede Linie stellt eine neue Instanz dar. Man erkennt den schnellen Abfall der Fehlerfunktion schon nach den ersten Iterationen, bis nach (hier maximal) 8 Iterationen der Algorithmus terminiert.

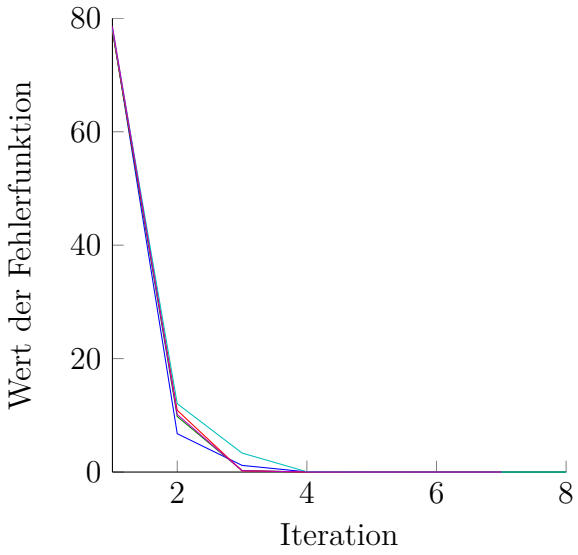

Abbildung 6.2.: Fehlerfunktion (6.1) als Funktion der Iteration

7. Schlusswort

Im Rahmen dieser Arbeit wurde ein in der Praxis sehr erfolgreicher Algorithmus zur Lösung von Optimierungsproblemen, das primal-duale Innere-Punkte-Verfahren, vorgestellt. Das Verfahren wurde näher für konvexe quadratische Optimierungsprobleme, die im Zusammenhang mit Optimaler Steuerung und Model Predictive Control auftreten, betrachtet. Wir haben die spezielle Struktur solcher Probleme studiert und sie mit Hilfe der Methode des Cyclic Reduction zur effizienten Lösung der in Innere-Punkte-Verfahren aufkommenden Gleichungssysteme genutzt. Eine Testimplementation wurde in Kapitel 6 gegeben.
Im Weiteren wäre eine genaue Beschäftigung mit den erwähnten numerischen Schwierigkeiten sinnvoll. Außerdem gibt es viele mögliche Baustellen, um das Verfahren effizienter zu machen, z.B. durch die Wahl des Startpunktes, einer anderen Fehlerfunktion, geschicktere Berechnung der Inversen beim Cyclic Reduction, etc. Der nächste Schritt wäre eine echte parallele Implementation des Cyclic Reduction.

A. Programmcode

An dieser Stelle listen wir alle Funktionen auf, die im Rahmen dieser Arbeit implementiert wurden. Wir geben am Anfang jedes Algorithmus eine kurze Beschreibung an. Die Codes orientieren sich an den Algorithmen (2) und (3).

A.1. IPM Prädiktor-Korrektor Code

Die Funktion ipmPC_MPC ist eine Implementation der Prädiktor-Korrektor-Methode für Probleme der Form (2.2). Vom Ablauf her folgt sie streng dem Schema des Algorithmus (2). Die Beschreibung der Eingabe findet sich im Kommentar, zum besseren Verständnis ist ein Test-script in (A.8) gegeben.

Listing A.1: ipmPC_MPC.m

```
1  function [ val,z,err,iterations,plot ] = ipmPC_MPC( ...
       B,C,D,b,c,d,nx,nu,l,z_0,mu_0,lambda_0,t_0,maxIter )
2  % this function solves a subclass of QPs arising in MPC of the ...
       form (2.2)
3  % (see bachelor thesis)
4  % using the Predictor-Corrector Method
5  %
6  % Input: nx dimension of state variables,
7  %        nu dimension of control variables,
8  %        B, C, D are 3-dim. matrices,
9  %        b, c, d are 2-dim. matrices,
10 %        N = 2^l,
11 %        B = [B_1,...,B_N],          (B_i is matrix of dimension ...
       (nx+nu)x(nx+nu))
12 %        C = [C_1,...,C_N-1],        (C_i is matrix of dimension ...
       (nx)x(nx+nu))
13 %        D = [D_1,...,D_N],          (D_i is (nd)x(nx+nu)-matrix)
14 %        b = [b_1,...,b_N],          (b_i is vector of dimension nx+nu)
15 %        c = [c_1,...,c_N-1],        (c_i is vector of dimension nx)
16 %        d = [d_1,...,d_N],          (d_i is vector of dimension nd)
17 %        starting points: z_0        (z_0 is vector of dimension ...
       N*(nx+nu))
18 %                         mu_0       (mu_0 is vector of dimension ...
       (N-1)*nx)
19 %                         lambda_0   (lambda_0 is vector of ...
       dimension N*nd)
20 %                         t_0        (t_0 is vector of dimension N*nd)
21 %        maxIter: maximum number of iterations
22 %        epsilon: tolerance for termination
23 % Output:
```

```
24 %        function value val,
25 %        optimal point z,
26 %        first order optimality err,
27 %        number of iterations iterations
28 %        plot contains value of error-function of each iteration, ...
       needed
29 %        for plotting
30
31
32 %%%% STEP 0: initialization
33
34 nz = nx + nu;
35 [nd,¬] = size(d);
36
37 % check for correct dimensions
38 assert(((isequal(size(B),[nz,nz,2^l])) && ...
       (isequal(size(C),[nx,nz,2^l-1])) && ...
       (isequal(size(D),[nd,nz,2^l])) && (isequal(size(b),[nz,2^l])) ...
       && (isequal(size(c),[nx,2^l-1])) && (isequal(size(d),[nd,2^l])) ...
       && (isequal(size(z_0),[nz*2^l,1])) && ...
       (isequal(size(mu_0),[nx*(2^l-1),1])) && ...
       (isequal(size(lambda_0),[nd*2^l,1])) && ...
       (isequal(size(t_0),[nd*2^l,1]))),'dimension mismatch');
39
40 plot = [];
41 epsilon = 10^(-6);
42 counter = 0;
43
44 z = zeros(nz,2^l);
45 mu = zeros(nx,2^l-1);
46 lambda = zeros(nd,2^l);
47 t = zeros(nd,2^l);
48
49 for i=1:2^l
50     z(:,i) = z_0((i-1)*nz+1:i*nz);
51     lambda(:,i) = lambda_0((i-1)*nd+1:i*nd);
52     t(:,i) = t_0((i-1)*nd+1:i*nd);
53 end
54 for i=1:2^l-1
55      mu(:,i) = mu_0((i-1)*nx+1:i*nx);
56 end
57
58 T = zeros(nd,nd,2^l);
59 inv_T = zeros(nd,nd,2^l);
60 Lambda = zeros(nd,nd,2^l);
61 inv_Lambda = zeros(nd,nd,2^l);
62 for i=1:2^l
63     T(:,:,i) = diag(t(:,i));
64     inv_T(:,:,i) = diag(1./t(:,i));
65     Lambda(:,:,i) = diag(lambda(:,i));
66     inv_Lambda(:,:,i) = diag(1./lambda(:,i));
67 end
68
69 E = [eye(nx),zeros(nx,nz-nx)];
70
71 r_q = zeros(nz,2^l);
```

```
72  r_h = zeros(nx,2^l-1);
73  r_g = zeros(nd,2^l);
74  r_complementary = zeros(nd,2^l);
75
76  % compute residuum r_q
77  r_q(:,1) = B(:,:,1) * z(:,1) + C(:,:,1)' * mu(:,1) + D(:,:,1)' * ...
        lambda(:,1) + b(:,1);
78  for i=2:2^l-1
79      r_q(:,i) = B(:,:,i) * z(:,i) - E' * mu(:,i-1) + C(:,:,i)' * ...
            mu(:,i) + D(:,:,i)' * lambda(:,i) + b(:,i);
80  end
81  r_q(:,2^l) = B(:,:,2^l) * z(:,2^l) - E' * mu(:,2^l-1) + ...
        D(:,:,2^l)' * lambda(:,2^l) + b(:,2^l);
82
83  % compute residuum r_h
84  for i=1:2^l-1
85      r_h(:,i) = C(:,:,i) * z(:,i) - E * z(:,i+1) + c(:,i);
86  end
87
88  % compute residuum r_g
89  for i=1:2^l
90      r_g(:,i) = D(:,:,i) * z(:,i) + t(:,i) - d(:,i);
91  end
92
93  % compute residuum r_complementary
94  for i=1:2^l
95      r_complementary(:,i) = T(:,:,i) * lambda(:,i);
96  end
97
98  %%%% STEP 1: stopping criteria
99
100 % check if KKT conditions are satisfied
101 while (max([norm(r_q(:));norm(r_g(:));norm(r_h(:)); ...
        norm(r_complementary(:))]) ≥ epsilon && counter ≤ maxIter)
102     plot = [plot;max([norm(r_q(:));norm(r_g(:));norm(r_h(:)); ...
            norm(r_complementary(:))])];
103     counter = counter + 1;
104
105     %%%% STEP 2 : obtain predictor step (affine scaling direction)
106
107     % obtain system for cyclic reduction step
108     % diagonal elements
109     A_CR = cell(2^l+2^l-1);
110     for i=1:2:2^(l+1)-2
111         A_CR{i} = B(:,:,(i+1)/2) + D(:,:,(i+1)/2)' * ...
                Lambda(:,:,(i+1)/2) * inv_T(:,:,(i+1)/2) * D(:,:,(i+1)/2);
112         A_CR{i+1} = zeros(nx);
113     end
114     A_CR{2^(l+1)-1} = B(:,:,2^l) + D(:,:,2^l)' * Lambda(:,:,2^l) * ...
            inv_T(:,:,2^l) * D(:,:,2^l);
115
116     % off-diagonal elements
117     B_CR = cell(2^l+2^l-2);
118     C_CR = cell(2^l+2^l-2);
119     for i=1:2:2^(l+1)-2
120         B_CR{i} = C(:,:,(i+1)/2);
```

```
121         C_CR{i} = C(:,:,(i+1)/2)';
122         B_CR{i+1} = -E';
123         C_CR{i+1} = -E;
124     end
125
126     % compute right-hand site for cyclic reduction step
127     y_CR = cell(2^l+2^l-1);
128     for i=1:2:2^(l+1)-3
129         y_CR{i} = -r_q(:,(i+1)/2) - D(:,:,(i+1)/2)' * ...
                Lambda(:,:,(i+1)/2) * inv_T(:,:,(i+1)/2) * ...
                D(:,:,(i+1)/2) * z(:,(i+1)/2) + D(:,:,(i+1)/2)' * ...
                Lambda(:,:,(i+1)/2) * inv_T(:,:,(i+1)/2) * d(:,(i+1)/2);
130         y_CR{i+1} = -r_h(:,(i+1)/2);
131     end
132     y_CR{2^(l+1)-1} = -r_q(:,2^l) - D(:,:,2^l)' * Lambda(:,:,2^l) ...
            * inv_T(:,:,2^l) * D(:,:,2^l) * z(:,2^l) + D(:,:,2^l)' * ...
            Lambda(:,:,2^l) * inv_T(:,:,2^l) * d(:,2^l);
133
134     % solve system with cyclic reduction
135     [sol_CR,sub_A_CR,sub_B_CR,sub_C_CR,inv_A_CR,sub_inv_A_CR] = ...
            blk_CR(A_CR,B_CR,C_CR,y_CR,nx,nz,l+1);
136
137     % stop loop if NaN
138     if(not(isequal(isnan(vertcat(sol_CR{:})), ...
            zeros(length(vertcat(sol_CR{:})),1))))
139         break
140     end
141
142     % obtain affine scaling direction with back-substitution
143     Δ_z = zeros(nz,2^l);
144     Δ_mu = zeros(nx,2^l-1);
145     Δ_lambda = zeros(nd,2^l);
146     Δ_t = zeros(nd,2^l);
147
148     % obtain Δ_z, Δ_mu
149     for i=1:2^l-1
150         Δ_z(:,i) = sol_CR{2*i-1};
151         Δ_mu(:,i) = sol_CR{2*i};
152     end
153     Δ_z(:,2^l) = sol_CR{2^(l+1)-1};
154
155     % obtain Δ_lambda
156     for i=1:2^l
157         Δ_lambda(:,i) = -Lambda(:,:,i) * inv_T(:,:,i) * (-r_g(:,i) ...
                + inv_Lambda(:,:,i) * r_complementary(:,i) - D(:,:,i) * ...
                Δ_z(:,i));
158     end
159
160     % obtain Δ_t
161     for i=1:2^l
162         Δ_t(:,i) = inv_Lambda(:,:,i) * (-r_complementary(:,i) - ...
                T(:,:,i) * Δ_lambda(:,i));
163     end
164
165
166
```

```
167     %%%% STEP 3: compute step length for lambda and t for affine ...
            scaling direction
168
169     j = find(Δ_lambda<0);
170     if (isempty(j))
171         alpha_lambda = 1;
172     else
173         alpha_lambda = min(1,min(-lambda(j)./Δ_lambda(j)));
174     end
175
176     j = find(Δ_t<0);
177     if (isempty(j))
178         alpha_t = 1;
179     else
180         alpha_t = min(1,min(-t(j)./Δ_t(j)));
181     end
182
183
184
185     %%%% STEP 4: compute duality measure xi and xi_aff
186
187     xi = t(:)'*lambda(:)/(2^l*nd);
188
189     xi_aff = (t(:) + alpha_t*Δ_t(:))' * (lambda(:) + alpha_lambda ...
            * Δ_lambda(:)) / (2^l*nd);
190
191
192
193     %%%% STEP 5: compute centering parameter
194
195     sigma = (xi_aff/xi)^3;
196
197
198
199     %%%% STEP 6: obtain corrector step
200
201     Delta_Lambda = zeros(nd,nd,2^l);
202     Delta_T = zeros(nd,nd,2^l);
203     e=ones(nd,1);
204     for i=1:2^l
205         Delta_T(:,:,i) = diag(Δ_t(:,i));
206         Delta_Lambda(:,:,i) = diag(Δ_lambda(:,i));
207     end
208
209     % obtain right hand site for corrector step
210     for i=1:2:2^(l+1)-1
211         y_CR{i} = y_CR{i} + D(:,:,(i+1)/2)' * inv_T(:,:,(i+1)/2) * ...
                (Delta_T(:,:,(i+1)/2) * Delta_Lambda(:,:,(i+1)/2) * e - ...
                sigma * xi * e);
212     end
213
214     for i=2^(l+1)-1:-1:2
215     B_CR{i} = B_CR{i-1};
216     end
217     B_CR{1} = zeros(nz,nx);
218     C_CR{2^(l+1)-1} = zeros(nz,nx);
```

```
219
220     sub_y_CR = ...
            reduction_step_RHS(B_CR,C_CR,sub_B_CR,sub_C_CR,inv_A_CR, ...
            sub_inv_A_CR,y_CR,l+1);
221     sol_CR = ...
            substitution_step(B_CR,C_CR,y_CR,sub_A_CR,sub_B_CR,sub_C_CR, ...
            sub_y_CR,inv_A_CR,sub_inv_A_CR,l+1);
222
223     % stop loop if NaN
224     if(not(isequal(isnan(vertcat(sol_CR{:})), ...
            zeros(length(vertcat(sol_CR{:})),1))))
225         break
226     end
227
228     % obtain Δ_z, Δ_mu
229     for i=1:2^l-1
230         Δ_z(:,i) = sol_CR{2*i-1};
231         Δ_mu(:,i) = sol_CR{2*i};
232     end
233     Δ_z(:,2^l) = sol_CR{2^(l+1)-1};
234
235     % change r_complementary for corrector step
236     for i=1:2^l
237         r_complementary(:,i) = r_complementary(:,i) + ...
                Delta_T(:,:,i) * Delta_Lambda(:,:,i) * e - xi * sigma * e;
238     end
239
240     % obtain Δ_lambda
241     for i=1:2^l
242         Δ_lambda(:,i) = -Lambda(:,:,i) * inv_T(:,:,i) * (-r_g(:,i) ...
                + inv_Lambda(:,:,i) * r_complementary(:,i) - D(:,:,i) * ...
                Δ_z(:,i));
243     end
244
245     % obtain Δ_t
246     for i=1:2^l
247         Δ_t (:,i) = inv_Lambda(:,:,i) * (-r_complementary(:,i) - ...
                T(:,:,i) * Δ_lambda(:,i));
248     end
249
250
251
252     %%%% STEP 7: compute step length
253
254     j = find(Δ_lambda<0);
255     if (isempty(j))
256         alpha_lambda = 1;
257     else
258         alpha_lambda = min(1,min(-0.99*lambda(j)./Δ_lambda(j)));
259     end
260
261     j = find(Δ_t<0);
262     if (isempty(j))
263         alpha_t = 1;
264     else
265         alpha_t = min(1,min(-0.99*t(j)./Δ_t(j)));
```

```
266     end
267
268     alpha = min(alpha_lambda,alpha_t);
269
270
271
272     %%%% STEP 8: compute new iterate
273
274     z = z + alpha * Δ_z;
275     mu = mu + alpha * Δ_mu;
276     lambda = lambda + alpha * Δ_lambda;
277     t = t + alpha * Δ_t;
278
279
280
281     %%%% STEP 9: update
282
283     % update diagonal matrices
284     for i=1:2^l
285         T(:,:,i) = diag(t(:,i));
286         inv_T(:,:,i) = diag(1./t(:,i));
287         Lambda(:,:,i) = diag(lambda(:,i));
288         inv_Lambda(:,:,i) = diag(1./lambda(:,i));
289     end
290
291     % update residuum r_q
292     r_q(:,1) = B(:,:,1) * z(:,1) + C(:,:,1)' * mu(:,1) + D(:,:,1)' ...
            * lambda(:,1) + b(:,1);
293     for i=2:2^l-1
294         r_q(:,i) = B(:,:,i) * z(:,i) - E' * mu(:,i-1) + C(:,:,i)' ...
                * mu(:,i) + D(:,:,i)' * lambda(:,i) + b(:,i);
295     end
296     r_q(:,2^l) = B(:,:,2^l) * z(:,2^l) - E' * mu(:,2^l-1) + ...
            D(:,:,2^l)' * lambda(:,2^l) + b(:,2^l);
297
298     % update residuum r_h
299     for i=1:2^l-1
300        r_h(:,i) = C(:,:,i) * z(:,i) - E * z(:,i+1) + c(:,i);
301     end
302
303     % update residuum r_g
304     for i=1:2^l
305         r_g(:,i) = D(:,:,i) * z(:,i) + t(:,i) - d(:,i);
306     end
307
308     % update residuum r_complementary
309     for i=1:2^l
310         r_complementary(:,i) = T(:,:,i)*lambda(:,i);
311     end
312
313 end
314
315 %%%% STEP 10: compute output
316
317 % test for feasibility
318 for i=1:2^l-1
```

```
319      if ((norm((E * z(:,i+1) - C(:,:,i) * z(:,i) - c(:,i))) > ...
             epsilon) || (not(isempty((find(D(:,:,i) * z(:,i) > d(:,i) + ...
             epsilon * ones(nd,1)))))))
320          disp('No feasible point found.');
321          val = NaN;
322          z = NaN;
323          err = NaN;
324          iterations = NaN;
325          return;
326      end
327  end
328
329  iterations = counter;
330  err = max([norm(r_q(:));norm(r_g(:));norm(r_h(:)); ...
         norm(r_complementary(:))]);
331  plot = [plot;max([norm(r_q(:));norm(r_g(:));norm(r_h(:)); ...
         norm(r_complementary(:))])];
332
333  val = 0;
334  for i=1:2^l
335  val = val + 1/2 * z(:,i)' * B(:,:,i) * z(:,i) + b(:,i)' * z(:,i);
336  end
337
338  end
```

A.2. Cyclic Reduction Code

Die Funktion blk_CR ist die Implementation des Cyclic Reduction. Sie orientiert sich an (3). Die einzelnen Schritte sind getrennt implementiert. Die Funktionen reduction_step_LHS und reduction_step_RHS entsprechen der Reduktionsphase, die Funktion substitution_step der Substituionsphase. Die Hauptfunktion blk_CR ruft die drei anderen Funktionen auf.

Listing A.2: blk_CR.m

```
1  function [ x,sub_A,sub_B,sub_C,inv_A,sub_inv_A ] = blk_CR( ...
       A,B,C,y,m,n,l )
2  % A, B, C, y are cells
3  % A = [A_1,...,A_N], B=[B_2,...,B_N], C=[C_1,...,C_N-1], ...
       y=[y_1,...,y_N]
4  % with N=2^l-1
5  % x is solution of blocktridiag(B,A,C) * x = y
6
7  for i=2^l-1:-1:2
8      B{i} = B{i-1};
9  end
10 B{1} = zeros(n,m);
11 C{2^l-1} = zeros(n,m);
12
13 % reduction step
```

```
14 [sub_A,sub_B,sub_C,inv_A,sub_inv_A] = reduction_step_LHS( A,B,C,l );
15 sub_y = reduction_step_RHS(B,C,sub_B,sub_C,inv_A,sub_inv_A,y,l);
16
17 % substitution step
18 x = substitution_step( ...
       B,C,y,sub_A,sub_B,sub_C,sub_y,inv_A,sub_inv_A,l );
19 end
```

Listing A.3: reduction_step_LHS.m

```
1  function [ sub_A,sub_B,sub_C,inv_A,sub_inv_A] = ...
       reduction_step_LHS( A,B,C,l )
2  %%%% reduction step %%%%
3
4  % compute submatrices A^i_j, B^i_j, C^i_j and inverse of A^i_j for ...
       j odd
5  sub_A = cell(1,l-1);
6  sub_inv_A = cell(1,l-1);
7  sub_B = cell(1,l-1);
8  sub_C = cell(1,l-1);
9
10 for i=1:l-1
11     sub_A{i} = cell(1,2^(l-i)-1);
12     sub_inv_A{i} = cell(1,2^(l-i)/2);
13     sub_B{i} = cell(1,2^(l-i)-1);
14     sub_C{i} = cell(1,2^(l-i)-1);
15 end
16
17 inv_A = cell(1,2^(l-1));
18
19 for j=1:2^l/2
20     inv_A{j} = inv(A{2*j-1});
21 end
22
23 % initialization (i=1)
24 for j=1:2^(l-1)-1
25     sub_A{1}{j} = A{2*j} - B{2*j} * inv_A{j} * C{2*j-1} - C{2*j} * ...
           inv_A{j+1} * B{2*j+1};
26     sub_B{1}{j} = -B{2*j} * inv_A{j} * B{2*j-1};
27     sub_C{1}{j} = -C{2*j} * inv_A{j+1} * C{2*j+1};
28 end
29
30 for i=2:l-1
31     for j=1:2^(l-(i-1))/2
32         sub_inv_A{i-1}{j} = inv((sub_A{i-1}{2*j-1}));
33     end
34     for j=1:2^(l-i)-1
35         sub_A{i}{j} = sub_A{i-1}{2*j} - sub_B{i-1}{2*j} * ...
               sub_inv_A{i-1}{j} * sub_C{i-1}{2*j-1} - sub_C{i-1}{2*j} ...
               * sub_inv_A{i-1}{j+1} * sub_B{i-1}{2*j+1};
36         sub_B{i}{j} = -sub_B{i-1}{2*j} * sub_inv_A{i-1}{j} * ...
               sub_B{i-1}{2*j-1};
37         sub_C{i}{j} = -sub_C{i-1}{2*j} * sub_inv_A{i-1}{j+1} * ...
               sub_C{i-1}{2*j+1};
38     end
```

```
39 end
```

Listing A.4: reduction_step_RHS.m

```
1  function [ sub_y ] = reduction_step_RHS( ...
       B,C,sub_B,sub_C,inv_A,sub_inv_A,y,l )
2
3  % compute subvectors y^i_j
4  sub_y = cell(1,l-1);
5
6  for i=1:l-1
7      sub_y{i} = cell(1,2^(l-i)-1);
8  end
9
10 for j=1:2^(l-1)-1
11     sub_y{1}{j} = y{2*j} - B{2*j} * inv_A{j} * y{2*j-1} - C{2*j} * ...
           inv_A{j+1} * y{2*j+1};
12 end
13
14 for i=2:l-1
15     for j=1:2^(l-i)-1
16         sub_y{i}{j} = sub_y{i-1}{2*j} - sub_B{i-1}{2*j} * ...
               sub_inv_A{i-1}{j} * sub_y{i-1}{2*j-1} - sub_C{i-1}{2*j} ...
               * sub_inv_A{i-1}{j+1} * sub_y{i-1}{2*j+1};
17     end
18 end
```

Listing A.5: substitution_step.m

```
1  function [ x ] = substitution_step( ...
       B,C,y,sub_A,sub_B,sub_C,sub_y,inv_A,sub_inv_A,l )
2  %%%% substitution step %%%%
3
4  sub_x = cell(1,l-1);
5
6  for i=1:l-1
7      sub_x{i} = cell(1,2^(l-i)-1);
8  end
9
10 % compute solution of A^(l-1)_1 * x = y^(l-1)_1
11 sub_x{l-1}{1} = sub_A{l-1}{1}\sub_y{l-1}{1};
12
13 % compute subsolutions x^i_j
14 for i=l-2:-1:1
15     for j=1:2^(l-(i+1))-1
16          sub_x{i}{2*j} = sub_x{i+1}{j};
17     end
18
19     sub_x{i}{1} = sub_inv_A{i}{1} * (sub_y{i}{1} - sub_C{i}{1} * ...
           sub_x{i}{2});
20
21     for j=2:2^(l-(i+1))-1
22         sub_x{i}{2*j-1} = sub_inv_A{i}{j} * (sub_y{i}{2*j-1} - ...
               sub_B{i}{2*j-1} * sub_x{i}{2*j-2} - sub_C{i}{2*j-1} * ...
```

```
            sub_x{i}{2*j});
    end

    sub_x{i}{2^(l-i)-1} = sub_inv_A{i}{2^(l-i)/2} * ...
        (sub_y{i}{2^(l-i)-1} - sub_B{i}{2^(l-i)-1} * ...
        sub_x{i}{2^(l-i)-1-1});

end

% compute solution x
x = cell(1,2^l-1);

for j=1:2^(l-1)-1
    x{2*j} = sub_x{1}{j};
end

x{1} = inv_A{1}*(y{1} - C{1}*x{2});

for j=2:2^(l-1)-1
    x{2*j-1} = inv_A{j} * (y{2*j-1} - B{2*j-1} * x{2*j-2} - ...
        C{2*j-1} * x{2*j});
end

x{2^l-1} = inv_A{2^(l-1)} * (y{2^l-1} - B{2^l-1} * x{2^l-2});
end
```

A.3. Hilfsfunktionen

Die Funktion blk_tridiag erzeugt eine Block-Tridiagonal-Matrix aus ihren Eingaben. Sie wird nur von der nachfolgenden Funktion transMPC verwendet.

Listing A.6: blk_tridiag.m

```
function [ X ] = blk_tridiag( B,A,C,m,n,N )
% A,B,C are 3D matrices of size mxnxN
% A contains diagonal blocks
% B and C contain off-diagonal blocks
% function generates a block-tridiagonal matrix from A, B, C

for i=1:N
    X((i-1)*m+1:i*m,(i-1)*n+1:i*n) = A(:,:,i);
end

for i=1:N-1
    X(i*m+1:(i+1)*m,(i-1)*n+1:i*n) = B(:,:,i);
end

for i=1:N-1
    X((i-1)*m+1:i*m,i*n+1:(i+1)*n) = C(:,:,i);
end
```

```
18
19 end
```

Die Funktion transMPC transformiert das in der Form (2.2) gegebene Problem in die allgemeine Form für QPS (2.1) nach (2.3). Sie ist notwendig, um die Matrizen und Vektoren für den Matlab Solver quadprog zu generieren.

Listing A.7: transMPC.m

```
1  function [ Q,H,G,q,h,g ] = transMPC( B,C,D,b,c,d,nx,nu,l)
2  % Input: MPC Problem of the form (2.2) given by
3  %        nx dimension of state variables,
4  %        nu dimension of control variables,
5  %        B, C, D are 3-dim. matrices,
6  %        b, c, d are 2-dim. matrices,
7  %        N = 2^l,
8  %        B = [B_1,...,B_N],          (B_i is matrix of dimension ...
       (nx+nu)x(nx+nu))
9  %        C = [C_1,...,C_N-1],        (C_i is matrix of dimension ...
       (nx)x(nx+nu))
10 %        D = [D_1,...,D_N],          (D_i is (nd)x(nx+nu)-matrix)
11 %        b = [b_1,...,b_N],          (b_i is vector of dimension nx+nu)
12 %        c = [c_1,...,c_N-1],        (c_i is vector of dimension nx)
13 %        d = [d_1,...,d_N],          (d_i is vector of dimension nd)
14 % Output: MPC Problem in form of (2.1)
15
16 nz = nx + nu;
17
18 Q = [];
19 for i=1:2^l
20     Q = blkdiag(Q,B(:,:,i));
21 end
22 E = [eye(nx),zeros(nx,nu)];
23 S = zeros(nx,nz,2^l-2);
24 for i=1:2^l-2
25     S(:,:,i) = -E;
26 end
27 H = blk_tridiag(zeros(nx,nz,2^l-2),C,S,nx,nz,2^l-1);
28 H = [H,zeros((2^l-1)*nx,nz)];
29 H((2^l-2)*nx+1:(2^l-1)*nx,(2^l-1)*nz+1:(2^l)*nz) = -E;
30 G = [];
31 for i=1:2^l
32     G = blkdiag(G,D(:,:,i));
33 end
34 h = -c(:);
35 g = d(:);
36 q = b(:);
37
38 end
```

A.4. Matlab Test-script

Hier finden sich zwei Test-scrips für die Funktion ipmPC_MPC. Das Matlab-script simple_test.m soll zur Veranschaulichung der Eingaben dienen, mit test.m wurden die Ergebnisse der Tabelle (6.1) erzeugt.

Listing A.8: simple_test

```
% simple test problem for function ipmPC_MPC

clear all;

% problem dimensions
nx = 10;
nu = 10;
nz = nx + nu;
nd = 15;
l = 6;

%generate matrices and vectors
B = zeros(nz,nz,2^l);
for i=1:2^l
    B(:,:,i) = eye(nz);
end
C = ones(nx,nz,2^l-1);
D = ones(nd,nz,2^l);
b = ones(nz,2^l);
c = ones(nx,2^l-1);
d = ones(nd,2^l);

% generate starting point
z_0 = zeros(2^l*nz,1);
mu_0 = zeros((2^l-1)*nx,1);
lambda_0 = ones(2^l*nd,1);
t_0 = ones(2^l*nd,1);

% maximum number of iterations
maxIter = 100;

% solve problem
[val,sol,err,iter] = ...
    ipmPC_MPC(B,C,D,b,c,d,nx,nu,l,z_0,mu_0,lambda_0,t_0,maxIter);
```

Listing A.9: test.m

```
% test.m
% this script produces results for the table 6.1, first row

clear all;

% number of runs
runs=100;

% dimensions of our problem
```

```
10 nx = 3;
11 nu = 2;
12 nz = nx + nu;
13 nd = 5;
14 l = [4;6;7];
15
16 maxIter = 100;
17
18 result = zeros(2,runs,length(l));
19
20 for j=1:length(l);
21     for k=1:runs;
22
23         % generate random matrices and vectors
24         B = rand(nz,nz,2^l(j));
25         for i=1:2^l(j)
26             B(:,:,i) = B(:,:,i) * B(:,:,i)';
27             B(:,:,i) = B(:,:,i) + nz*eye(nz);
28         end
29         C = rand(nx,nz,2^l(j)-1);
30         D = rand(nd,nz,2^l(j));
31         d = rand(nd,2^l(j));
32         b = rand(nz,2^l(j));
33         c = rand(nx,2^l(j)-1);
34
35         % generate starting point
36         z_0 = zeros(2^l(j)*nz,1);
37         mu_0 = zeros((2^l(j)-1)*nx,1);
38         lambda_0 = ones(2^l(j)*nd,1);
39         t_0 = ones(2^l(j)*nd,1);
40
41         % solve with testimplementation
42         [~,~,~,iter] = ipmPC_MPC(B,C,D,b,c,d,nx,nu,l(j),z_0,mu_0, ...
               lambda_0,t_0,maxIter);
43
44         % generate matrices and vectors for quadprog
45         [Q,H,G,q,h,g] = transMPC(B,C,D,b,c,d,nx,nu,l(j));
46
47         % set options for quadprog
48         options = optimoptions(@quadprog,'Algorithm', ...
               'interior-point-convex','Display','off');
49
50         % solve with quadprog
51         [~,~,~,output] = quadprog(Q,q,G,g,H,h,[],[],[],options);
52         result(1,k,j) = iter;
53         result(2,k,j) = output.iterations;
54     end
55 end
56
57 % compute average number of iterations
58 average = zeros(2,length(l));
59 for j=1:length(l);
60     for k=1:2;
61         counter = 0;
62         sum = 0;
63         for i=1:runs;
```

```
64                 if(not(isnan(result(k,i,j))))
65                     counter = counter + 1;
66                     sum = sum + result(k,i,j);
67                 end
68             end
69             average(k,j) = sum/counter;
70         end
71     end
```

Literaturverzeichnis

[1] J. Nocedal, S. J. Wright, *Numerical Optimization*, Springer-Verlag, 2000

[2] R. J. Vanderbei, *LOQO: an interior point code for quadratic programming*, Technical Report SOR-94-15, Statistics and Operations Research, Priceton University, 1998

[3] D. Heller, *Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems*, SIAM Journal on Numerical Analysis, Vol. 13, No. 4: pp. 484-496, 1976

[4] S. Sahni, *Computationally Related Problems*, SIAM J. Comput., Vol. 3, No. 4, 1974

[5] J. B. Rawlings, D. Q. Mayne, *Model Predictive Control: Theory and Design*, Nob Hill Publishing, 2009

[6] J. V. Frasch, S. Sager, M. Diehl, *A Parallel Quadratic Programming Method for Dynamic Optimization Problems*, optimization-online.org, 2013

[7] M. Ulbrich, S. Ulbrich, *Nichtlineare Optimierung*, Birkhäuser-Verlag, 2012

[8] S. Mehrotra, *On the implementation of a primal-dual interior point method*, SIAM Journal on Optimization, Vol. 2, No. 4, pp. 575?601, 1992

[9] S. J. Wright, *Primal-Dual Interior-Point Methods*, SIAM, 1997

[10] J. Gondzio, *Interior Point Methods 25 Years Later*, European Journal of Operational Research, Vol. 218, pp. 587-601, 2012

[11] R. A. Sweet, *A Cyclic Reduction Algorithm for Solving Block Tridiagonal Systems of Arbitrary Dimension*, SIAM Journal on Numerical Analysis, Vol. 14, No. 4, pp. 706-720, 1977

[12] H. R. Schwarz, N. Köckler, *Numerische Mathematik*, 8. Auflage,Vieweg+Teubner Verlag, 2011

[13] W. Gander, G. H. Golub, *Cyclic Reduction ? History and Applications*, Scientific Computing, Hong Kong, 1997

[14] G. Lube, *Gauß-Elimination ohne Spaltenpivotisierung*, unter: https://lp.uni-goettingen.de/get/text/1016 (abgerufen am 10.08.2014)

[15] A. Domahidi, A. U. Zgraggen, M. N. Zeilinger, M. Morari, C. N. Jones, *Efficient Interior Point Methods for Multistage Problems Arising in Receding Horizon Control*, IEEE Conference on Decision and Control, pp. 668-674, USA, 2012

[16] J. E. Gentle, *Matrix Algebra: Theory, Computations, and Applications in Statistics*, Springer-Verlag, 2007

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ort, Datum, Unterschrift