Implementierung von Innere-Punkte Verfahren in Python

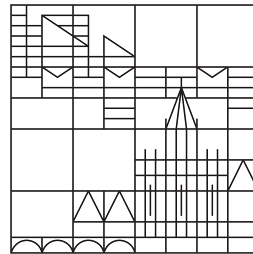
Bachelorarbeit

dargelegt von

Daniel Hoffmann

an der

Universität Konstanz



Mathematisch-Naturwissenschaftliche Sektion
Fachbereich Mathematik und Statistik

Gutachter: Herr Prof. Dr. Stefan Volkwein
Konstanz, 31. Mai 2019

Konstanzer Online-Publikations-System (KOPS)
URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-2-ek81av8rgix01

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

Implementierung von Innere-Punkte Verfahren in Python

selbständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinne nach entnommen sind, habe ich in jedem einzelnen Falle durch Angaben der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Konstanz, der 31. Mai 2019 ______________________________

Daniel Hoffmann

Inhaltsverzeichnis

1 Einführung 5

2 Mathematische Grundlagen 6
2.1 Optimalitätsbedingungen 6
2.2 Lineare Probleme 8
2.3 Dualität 9

3 Innere-Punkte Verfahren 11
3.1 Striktes Innere-Punkte Verfahren 11
3.2 Zentraler Pfad 13
3.3 Unzulässiges Innere-Punkte-Verfahren 14
3.4 Pfad-Verfolgungs Verfahren 15
3.4.1 Konvergenzverhalten 16
3.5 Prädiktor-Korrektor Verfahren 18
3.6 Lösen des Newtongleichungssystems 20

4 Implementierung in Python 22
4.1 Startpunkt Strategie 23
4.2 Implementierung eines einfachen Primal-Dualen Verfahrens 25
4.3 Implementierung des Prädiktor-Korrektor Verfahrens 29

5 Numerische Tests 31
5.1 Klee-Minty Problem 32

6 Anhang 35
6.1 iplp.py 35
6.2 klee_minty_test.py 43

1 Einführung

Die vorliegende Arbeit beschäftigt sich mit den mathematischen Grundlagen und der Implementierung von Innere-Punkte Verfahren, deren Thematik in der Praxis weiterhin an Bedeutung gewinnt. Vor allem in der Betriebswirtschaft scheint sich bezüglich der numerischen Optimierung eine große Bedeutung zu Entwickeln. So haben sich ganze Forschungsgebiete unter dem Namen *Operations Research* zusammengefasst.

In den einleitenden Abschnitten wollen wir in Anlehnung an Wright und Nocedal[5] eine Einführung in die Lineare Programmierung geben. Dabei wollen wir zuerst mit der Theorie zu linearen Programmen beginnen und führen aufbauend die Theorie zur Umsetzung von Innere-Punkte Verfahren ein.

Abschließen wird in einem praktischen Teil ein Innere-Punkte Verfahren implementiert, welches zu einem Vergleich mit einem in Python gängigen Verfahren herangezogen wird.

2 Mathematische Grundlagen

Im ersten Abschnitt werden wir die Ergebnisse aus [5] und [6] diskutieren, um die Grundlagen für das primale-duale Verfahren zu beweisen.

2.1 Optimalitätsbedingungen

Wir betrachten vorerst den allgemeinen Fall von Minimierungsproblemen um Aussagen über die Optimalitätsbedingungen der 1. und 2. Ordnung machen zu können. Die folgenden Ergebnisse sind vor allem für das von Bedeutung, da sich das direkt aus den Karush-Kuhn-Tucker (KKT) Bedingungen herleiten lässt.

Seien nun c und f glatte reellwertige Funktionen auf einer Teilmenge von $\mathbb{R}^n$. Wir bezeichnen dabei f als Ziel- oder Kostenfunktion und c als Nebenbedingung. Für endliche Indizes $\mathcal{G}$ und $\mathcal{U}$ definieren wir die zulässige Menge

$$\Omega := \{x \mid c_i(x) = 0 \text{ für } i \in \mathcal{G} \text{ und } c_i(x) \geq 0 \text{ für } i \in \mathcal{U}\}$$

.

Dann ist unser Optimierungsproblem mit Nebenbedingung von der Form

$$\min_{x \in \Omega} f(x). \tag{2.1}$$

Weiterhin sagen wir, dass die Ungleichungsbedingung aktiv ist, falls für ein $x \in \Omega$ ein $i \in \mathcal{U}$ existiert mit $c_i(x) = 0$. Entgegengesetzt sagen wir, die Ungleichungsbedingung ist inaktiv, falls die strikte Ungleichung $c_i(x) > 0$ gilt.

Definition 2.1.1. Wir bezeichnen einen Punkt $x^* \in \mathbb{R}^n$ als lokale Lösung von (2.1), falls x^* zulässig ist, d.h. $x \in \Omega$, und eine Umgebung $\mathcal{N}$ um x^* existiert mit $f(x) \geq f(x^*)$ für alle $x \in \mathcal{N} \cap \Omega$. Wir bezeichnen x^* als strikt lokale Lösung, falls zusätzlich $f(x) > f(x^*)$ für alle $x \in \mathcal{N} \cap \Omega$ und $x \neq x^*$ gilt.

Definition 2.1.2 (Lagrange-Funktion)**.** Die Lagrange-Funktion für allgemeine Minimierungsaufgaben mit Nebenbedingungen (2.1) ist definiert als

$$L(x, \lambda) = f(x) - \sum_{i \in \mathcal{G} \cup \mathcal{U}} \lambda_i c_i(x).$$

Es gilt zu beachten, dass der Lagrange-Multiplikator λ Aussagen über die Sensitivität des Optimierungsproblems macht, denn für ein $i \in \mathcal{G} \cup \mathcal{U}$ sagt λ_i aus, wie stark sich das Ergebnis $f(x^*)$ in der Lösung x^* unter der Nebenbedingung c_i ändert.

Definition 2.1.3. Wir definieren die Menge der aktiven Indizes $\mathcal{A}(x)$ für ein beliebiges zulässiges x, als die Menge für die alle Nebenbedingungen gleich null sind, also

$$\mathcal{A}(X) := \mathcal{G} \cup \{i \in \mathcal{U} \mid c_i(x) = 0\} \tag{2.2}$$

Definition 2.1.4. Ein Punkt x heißt regulär für (2.1), wenn $\nabla c_i(x)$ für $i \in \mathcal{G}$ und $\nabla c_i(x)$ für $i \in \mathcal{A}(x)$ linear unabhängig sind.

Satz 2.1.5 (Karush-Kuhn-Tucker Bedingungen)**.** *Sei x^* eine lokale Lösung zu* (2.1) *und ein regulärer Punkt. Dann existiert ein Lagrange-Multiplikator λ^* mit Komponenten λ_i, $i \in \mathcal{U} \cup \mathcal{G}$, so dass*

$$\nabla_x L(x^*, \lambda^*) = 0, \tag{2.3a}$$

$$c_i(x^*) = 0, \quad \textit{für alle } i \in \mathcal{G}, \tag{2.3b}$$

$$c_i(x^*) \geq 0, \quad \textit{für alle } i \in \mathcal{U}, \tag{2.3c}$$

$$\lambda_i^* \geq 0, \quad \textit{für alle } i \in \mathcal{U}, \tag{2.3d}$$

$$\lambda_i^* c_i(x^*) = 0, \quad \textit{für alle } i \in \mathcal{U} \cup \mathcal{G}. \tag{2.3e}$$

Beweis. Siehe [5, Kapitel 12]. □

Die letzte Bedingung werden wir im weiteren als Komplementär-Bedingung bezeichnen.

Bemerkung 2.1.6. Aus der Komplementärbedingung folgt, dass für die Menge der inaktiven Indizes $\lambda_i^* = 0$ gelten muss. Daraus lassen sich die KKT-Bedingungen wie folgt

$$\nabla_x L(x^*, \lambda^*) = \nabla f(x^*) - \sum_{i \in \mathcal{A}(x^*)} \lambda_i^* \nabla c_i(x^*) = 0, \tag{2.4a}$$

$$\lambda_i \geq 0, \quad \text{für } i \in \mathcal{A}(x^*), \tag{2.4b}$$

$$\lambda_i^* c_i(x^*) = 0, \quad \text{für } i \in \mathcal{U}, \tag{2.4c}$$

abkürzen.

Es gilt zu beachten, dass die KKT-Bedingungen nur eine notwendige Optimalitätsbedingung erster Ordnung sind. Daher sind die KKT-Bedingungen oftmals nicht hinreichend. Da wir aber weiterhin nur lineare Probleme betrachten werden und hinreichende Optimalitätsbedingungen für konvexe Probleme genügen, reichen uns die hiermit gemachten Aussagen. Die Beweise für eben jene lassen sich in [5, S. 351-353] finden.

Aus den KKT-Bedingungen kann, wie wir später sehen werden, das primal-duale Innere-Punkte-Verfahren hergeleitet werden. Für konvexe quadratische Funktionen können diese ebenfalls verwendet werden, um Innere-Punkte-Verfahren für diese Klasse von Problemen herzuleiten.

Häufig wird in der Literatur das Innere-Punkte-Verfahren mit logarithmischen Straftermen hergeleitet. Dabei wird eine Barrierefunktion $B(x, \mu) = f(x) - \mu \sum_{i \in \mathcal{U} \cup \mathcal{G}} \log(c_i(x))$ auf ihre Optimalitätsbedingungen erster und zweiter Ordnung untersucht.

2.2 Lineare Probleme

Definition 2.2.1. Gegeben sei

$$\begin{aligned} &\min c^T x \\ \text{u.d.Nb.} \quad &Ax \leq b \\ &x \geq 0, \end{aligned} \tag{2.5}$$

wobei c, x Vektoren in $\mathbb{R}^n$, b in $\mathbb{R}^m$ sind und A eine Matrix in $\mathbb{R}^{m\times n}$. Dabei ist x die zu suchende Größe. Wir bezeichnen (2.5) als Problem der Linearen Programmierung (LP), da die Zielfunkion, als auch die Nebenbedingungen linear sind.

Durch das Einführen von einer Schlupfvariable $\mu \in \mathbb{R}^m$ können wir die Ungleichung in einem LP zu einer Gleichung umformen. Dabei soll μ so gewählt werden, dass mit gleichen A, x und b

$$\begin{aligned} &\min \begin{pmatrix} c^T & 0 \end{pmatrix} \begin{pmatrix} x \\ \mu \end{pmatrix}, \\ \text{u.d.Nb.} \quad &\begin{pmatrix} A & I_m \end{pmatrix} \begin{pmatrix} x \\ \mu \end{pmatrix} = b, \\ &x, \mu \geq 0, \end{aligned} \tag{2.6}$$

gilt.

Bemerkung 2.2.2. Gegeben sei das LP

$$\min c^T x, \quad \text{u.d.Nb. } Ax = b, \quad x \geq 0.$$

Dann sagen wir das LP ist in Standardform.

Dabei gehen wir davon aus dass $m \leq n$ gilt, da wir ansonsten Zeilen eliminieren können. Dabei hätte die Gleichung $Ax = b$ keine oder eine eindeutige Lösung und wir können das Minimierungsproblem ignorieren und das Gleichungssystem lösen, um zu überprüfen, ob eine zulässige Lösung existiert.

Wir definieren nun $J(x) = c^T x, g(x) \coloneqq c_i(x)$ für $i \in \mathcal{G}$ und $e(x) \coloneqq -c_i(x)$ für $i \in \mathcal{U}$.

Wir sehen, dass $e(x) \coloneqq b - Ax$ und $g(x) \coloneqq -x$ linear sind. Sei weiterhin x^* eine lokale Lösung von (2.5). Dann existiert ein $\lambda^* \in \mathbb{R}^m$ und ein $\mu^* \in \mathbb{R}^n$ mit $\mu^* \geq 0$, so dass die Optimalitäts-Bedingung erster Ordnung

$$\nabla J(x^*) + (\lambda^*)^T \nabla e(x^*) + (\mu^*)^T \nabla g(x^*) \tag{2.7}$$

erfüllt ist.

Weiterhin sind dann die KKT-Bedingungen für (2.6)

$$
\begin{aligned}
A^T\lambda + \mu &= c, \\
Ax &= b, \\
x, \mu &\geq 0, \\
x^T\mu &= 0.
\end{aligned}
\tag{2.8}
$$

Bemerkung 2.2.3. Jedes LP lässt sich in ein Maximierungsproblem transformieren, da $\min c^T x = \max -c^T x$ gilt. Weiterhin ist die Nebenbedingung $Ax \leq b$ äquivalent zu $-Ax \geq -b$.

2.3 Dualität

Definition 2.3.1. Gegeben sei das Problem (2.5). Wir bezeichnen

$$
\begin{aligned}
&\max b^T\lambda, \\
\text{u.d.Nb. } &A^T\lambda \leq c
\end{aligned}
\tag{2.9}
$$

als das duale Problem zu (2.5). Gegenläufig bezeichnen wir (2.5) als das primale Problem.

Um den Zusammenhang zwischen primalen und dualen Problemen besser zu verstehen, formen wir zuerst das duale Problem um in

$$
\begin{aligned}
&\min -b^T\lambda, \\
\text{u.d.Nb. } &c - A^T\lambda \geq 0.
\end{aligned}
\tag{2.10}
$$

Dann ist die Lagrange-Funktion mit Lagrange-Multiplikator x hiervon gerade

$$L(\lambda, x) = -b^T\lambda - x^T(c - A^T\lambda).$$

In diesem Fall existiert laut den KKT-Bedingungen ein x mit

$$
\begin{aligned}
Ax &= b, \\
A^T\lambda &\leq c, \\
x &\geq 0, \\
x^T(c - A^T\lambda)&, \quad \text{für } i = 1, \dots, n.
\end{aligned}
\tag{2.11}
$$

Setzen wir nun $\mu = c - A^T\lambda$ so sehen wir, dass (2.8) und (2.11) in wechselwirkender Relation zueinander stehen.

Bemerkung 2.3.2. Wir nennen (2.10) das duale Problem zum primalen Problem. Man kann zeigen dass die Relation beider Probleme symmetrisch, d.h. das duale Problem des dualen Problems wieder das primale Problem ist. Einen Beweis hierzu findet man in [5, S. 366].

Satz 2.3.3 (Schwacher Dualitätssatz)**.** *Gegeben sei ein zulässiges* $x \in \mathbb{R}^n$ *für das primale Problem und* $\lambda \in \mathbb{R}^m$ *für das duale Problem. Dann gilt*

$$c^T x \leq b^T \lambda.$$

Beweis. Es gilt

$$c^T x \leq (A^T \lambda)^T x = \lambda^T A x \leq b^T \lambda$$

Dabei folgen beide Ungleichungen aus den KKT-Bedingungen der primalen und dualen Probleme. □

Das Ergebnis des Schwachen Dualitätssatzes liefert uns später in den primalen-dualen Innere-Punkte-Verfahren ein starkes Werkzeug als Abbruchkriterium. Mit dem bisherigen Wissen können wir später eine Dualitätslücke definieren, welche im besten Fall gegen Null konvergieren sollte. Das ist jedoch abhängig von unserem Verfahren und von dem Ergebnis des nächsten Satzes.

Satz 2.3.4 (Starker Dualitätssatz)**.** *Gegeben sei nun das zulässige Lösungstripel* (x^*, λ^*, μ^*) *für das primale und duale Problem gegeben. Dann gilt nun*

$$c^T x^* = b^T \lambda^*.$$

Sollte eines der Probleme unbeschränkt sein, so existieren für das jeweilige andere Problem keine zulässigen Punkte.

Beweis. Siehe [5, S. 367]. □

Wir sehen also, falls ein Verfahren gegen ein lokales Minimum konvergiert, so sollte auch die Dualitätslücke $c^T x^{(k)} - b^T \lambda^{(k)}$ für steigende Iterationen k gegen Null konvergieren.

3 Innere-Punkte Verfahren

Im nächsten Abschnitt werden nun die Innere-Punkte-Verfahren aus den vergangenen Resultaten hergeleitet. Hierbei wird versucht, die Zielfunktion unter Berücksichtigung der KKT-Bedingungen, mithilfe des Newton-Verfahrens zu minimieren. Das Verfahren wird terminiert, sollte die Dualitätslücke klein genug werden. Insbesondere die Komplementär-Bedingung (2.3e) darf dabei bei keiner Iteration verletzt werden.

Die Motivation für Innere-Punkte Verfahren ist dabei, bessere theoretische Eigenschaften bezüglich Worst-Case Szenarien. Die Komplexität von Simplex-Methoden kann dabei exponentiell bei ansteigenden Dimensionen des LPs wachsen. Wir werden später ein Problem konstruieren, bei welchem das Simplex-Verfahren sehr ineffizient ist. Innere-Punkte-Verfahren versprechen eine bessere garantierte Leistung, sind aber oftmals bei kleineren Problemen langsamer. Im Best-Case Szenario sollte das Simplex-Verfahren performanter als das Innere-Punkte-Verfahren sein, was wir später im 5. Abschnitt sehen werden.

Außerdem sollte später deutlich sein, dass es nicht zwangsläufig von Vorteil sein wird die Komplementär-Bedingung strikt einzuhalten. Deshalb werden wir unzulässige Verfahren betrachten, welche größere Schrittweiten ermöglichen.

Zuletzt werden wir ein Prädiktor-Korrektor-Verfahren einführen, welches zuerst in [4] diskutiert wurde. Das Verfahren von Mehrotra wird auch heute meist als Standard der linearen Programmierung verwendet, da es in der Praxis meist eine schnellere Konvergenzgeschwindigkeit als herkömmliche Innere-Punkte-Verfahren aufweist.

3.1 Striktes Innere-Punkte Verfahren

Gegeben sei nun das LP in Standardform

$$\min c^T x, \quad \text{u.d.Nb. } Ax = b, \quad x \geq 0, \tag{3.1}$$

mit $c, x \in \mathbb{R}^n, b \in \mathbb{R}^m$ und $A \in \mathbb{R}^{m \times n}$, sowie das duale Problem

$$\max b^T \lambda, \quad \text{u.d.Nb. } A^T \lambda + \mu = c, \quad \mu \geq 0, \tag{3.2}$$

mit $\lambda \in \mathbb{R}^m$ und $\mu \in \mathbb{R}^n$. Wir wissen, dass das Lösungstripel (x^*, λ^*, μ^*) durch die KKT-Bedingungen

$$A^T \lambda^* + \mu = 0, \tag{3.3a}$$
$$Ax = b, \tag{3.3b}$$
$$(x^*)^T \mu^* = 0, \tag{3.3c}$$
$$x^*, \mu^* \geq 0 \tag{3.3d}$$

gegeben ist. Um eine Lösung zu finden, werden wir das Newton-Verfahren auf (3.3a)-(3.3c) anwenden. Die Bedingung (3.3d) werden wir im Nachhinein überprüfen und eine geeignete Schrittweite wählen, damit diese Bedingung strikt erfüllt ist.

Wir bringen nun die KKT-Bedingungen in eine kompaktere Form

$$F(x,\lambda,\mu) := \begin{pmatrix} A^T\lambda + \mu \\ Ax - b \\ XMe \end{pmatrix} = 0, \tag{3.4a}$$

$$(x,\mu) \geq 0, \tag{3.4b}$$

mit $X := \operatorname{diag}(x_1, x_2, \dots, x_n), M := \operatorname{diag}(\mu_1, \mu_2 \dots, \mu_n)$ und $e := (1,1,\dots,1)^T$.

Es ist dabei zu beachten, dass das Gleichungssystem (3.4a) nicht-linear ist.

Das zulässige erfordert dabei, dass jede Iteration $(x^{(k)}, \lambda^{(k)}, \mu^{(k)})$ strikt zulässig ist. Daher führen wir nun die zulässige Menge $\mathcal{F}$ und die strikt zulässige Menge $\mathcal{F}_0$ ein mit

$$\mathcal{F} := \{(x,\lambda,\mu) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \mid \text{(3.4a) und (3.4b) sind erfüllt.}\}$$
$$\mathcal{F}_0 := \{(x,\lambda,\mu) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \mid \text{(3.4a) erfüllt und (3.4b) strikt erfüllt.}\}$$

Wir fordern also für jede k-te Iterierte $(x^{(k)}, \lambda^{(k)}, \mu^{(k)}) \in \mathcal{F}_0$.

Für die Suchrichtung wenden wir das Newton-Verfahren für nichtlineare Gleichungen auf (3.4a) an. Dann erhalten wir

$$\nabla F(x^{(k)}, \lambda^{(k)}, \mu^{(k)}) \begin{pmatrix} \Delta x^{(k)} \\ \Delta\lambda^{(k)} \\ \Delta\mu^{(k)} \end{pmatrix} = -F(x^{(k)}, \lambda^{(k)}, \mu^{(k)}).$$

Sollte die aktuelle Iterierte bereits strikt zulässig sein, so erhalten wir den nichtlinearen Newton-Schritt

$$\underbrace{\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix}}_{\in\mathbb{R}^{(n+m+n)}} \begin{pmatrix} \Delta x^{(k)} \\ \Delta\lambda^{(k)} \\ \Delta\mu^{(k)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -XMe \end{pmatrix}. \tag{3.5}$$

Wir sehen, dass F nur in λ linear war, da die Funktionalmatrix nur noch in X und M von x und μ abhängig ist.

Führen wir nun den Newton-Schritt für die nächste Iterierte

$$(x^{(k+1)}, \lambda^{(k+1)}, \mu^{(k+1)}) = (x^{(k)}, \lambda^{(k)}, \mu^{(k)}) + (\Delta x^{(k)}, \Delta\lambda^{(k)}, \Delta\mu^{(k)})$$

durch, so berücksichtigen wir noch nicht, ob die neue Iterierte überhaupt in der strikt zulässigen Menge $\mathcal{F}_0$ liegt. Wir dämpfen daher den Newtonschritt mit einem $\alpha^{(k)} \in (0,1]$. Dabei wird $\alpha^{(k)}$ so gewählt, dass die nächste Iterierte

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

in $\mathcal{F}_0$ liegt.

Damit die nächste Iterierte überhaupt in der strikt zulässigen Menge liegt, muss eventuell eine sehr kleine Schrittweite gewählt werden. Daher kann es vorkommen, dass das Verfahren sehr langsam gegen ein lokales Minimum konvergiert, bzw. sehr viele Iterationen benötigt.

3.2 Zentraler Pfad

Im folgenden Verfahren wird die Suchrichtung weiterhin modifiziert, indem wir die Suchbedingungen für die Abstiegsrichtung etwas lockern. Wir fordern nun, dass die Komplementär-Bedingung nicht exakt gilt, sondern dass sie für ein Skalar $\tau > 0$ mit $x^T\mu = n\tau$ gilt. Dabei konstruieren wir die Suchrichtung so, dass τ im Verlauf des Verfahrens gegen Null konvergiert. Wir erzeugen also eine Umgebung um den zentralen Pfad, die trichterartig gegen die Lösung konvergiert.

Dabei modifizieren wir die KKT-Bedingungen wie folgt

$$\begin{aligned} A^T\lambda + \mu &= c, \\ Ax &= b, \\ x, \mu &\geq 0, \\ x^T\mu &= e\tau, \end{aligned} \tag{3.6}$$

wobei e der n-dimensionale Einheitsvektor ist.

Weiterhin kann man zeigen, dass für beliebiges $\tau > 0$ eine eindeutige Lösung zu (3.6) existiert, genau dann, wenn $\mathcal{F}_0$ nicht leer ist. Dann ist der zentrale Pfad als Abbildung,

$$\mathcal{C}\colon\ \mathbb{R}^+ \to \mathbb{R}, \quad \tau \mapsto \mathcal{C}(\tau) := (x_\tau, \lambda_\tau, \mu_\tau)$$

für die gerade

$$F(\mathcal{C}(\tau)) = \begin{pmatrix} 0 \\ 0 \\ e\tau \end{pmatrix}, \quad (x, \mu) > 0$$

gilt, definiert. Wir sehen auch, dass per Definition von F, $\mathcal{C}$ für $\tau \searrow 0$ gegen die primale-duale Lösung des LPs konvergiert, sofern der Grenzwert existiert.

Da wir nun nicht mehr entlang linearer Schrittrichtungen strikt auf dem zentralen Pfad entgegen der Lösung laufen, ermöglicht dieses Verfahren große Schrittweiten und wir erhoffen uns damit eine schnellere Konvergenz ohne die Positivitätskriterien zu verletzen.

Um die Umgebung entlang des zentralen Pfads zu konstruieren, führen wir zwei weitere Variablen ein. Einmal einen zentrierenden Parameter $\sigma \in [0, 1]$, welcher beliebig gewählt werden darf, und die gewichtete Dualitätslücke $\eta := \sum_{i=1}^{n} \frac{x_i s_i}{n} = \frac{x^T s}{n}$. Dabei ist σ als zentrierender Parameter zu verstehen, welcher steuert, wie stark die nächste Suchrichtung gegen den Zentralen Pfad laufen soll. Für $\sigma = 0$ haben wir gerade wieder die strikte Suchrichtung, welche später auch affiner Newtonschritt genannt wird. Für $\sigma = 1$ wird

eine zentrierende Laufrichtung gesucht, welche erst einmal nicht zwangsläufig gegen eine Lösung laufen muss.

Dann definieren wir $\tau := \sigma\eta$ und stellen die neue Suchrichtung, analog wie in (3.5), auf. Dadurch erhalten wir die Gleichung zur Suchrichtung

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{(k)} \\ \Delta\lambda^{(k)} \\ \Delta\mu^{(k)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -XMe + \sigma\eta e \end{pmatrix}. \tag{3.7}$$

Wir können nun einen Algorithmus formulieren, welcher unser primal-duales-Verfahren beschreibt.

Algorithmus 3.2.1 (Primal-Duales Verfahren)**.** *1. Wähle* $(x_0, \lambda_0, \mu_0) \in \mathcal{F}_0$

2. For k = 1,2,...

3. löse

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{(k)} \\ \Delta\lambda^{(k)} \\ \Delta\mu^{(k)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -XMe + \sigma\eta e \end{pmatrix} \textit{ mit } \quad \sigma \in [0,1] \textit{ und } \eta = \frac{(x^{(k)})^T \mu^{(k)}}{n}.$$

4. Setze $(x^{(k+1)}, \lambda^{(k+1)}, \mu^{(k+1)}) = (x^{(k)}, \lambda^{(k)}, \mu^{(k)}) + \alpha_k(\Delta x^{(k)}, \Delta\lambda^{(k)}, \Delta\mu^{(k)})$ *mit* α_k *so dass* $(x_i^{(k+1)}, \mu_i^{(k+1)}) > 0$ *für alle* $i = 1, \ldots, n$.

5. end (For).

3.3 Unzulässiges Innere-Punkte-Verfahren

Da bei der Wahl eines Startpunktes bisher immer ein Punkt innerhalb der zulässigen Menge gefordert war, modifizieren wir weiterhin die Suchrichtung. Für ein zulässiges Innere-Punkte-Verfahren muss für jede Iteration, und damit insbesondere auch für den Startpunkt,

$$A^T x^{(0)} - b = 0 \quad \textit{und} \quad A\lambda^{(0)} + \mu^{(0)} - c = 0$$

gelten.

Wir definieren nun Residuen, welche wir ebenfalls in unseren Newtonschritten minimieren wollen. Wir definieren

$$r_b := A^T x^{(0)} - b \quad \text{und} \quad r_c := A\lambda^{(0)} + \mu^{(0)} - c$$

und modifizieren die Gleichung zu unserem Newtonschritt

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{(k)} \\ \Delta\lambda^{(k)} \\ \Delta\mu^{(k)} \end{pmatrix} = \begin{pmatrix} -r_b \\ -r_c \\ -XMe + \sigma\eta e \end{pmatrix}. \tag{3.8}$$

Es wird nun also nicht nur versucht, in einem Newtonschritt eine zentrierende Suchrichtung zu finden, sondern auch eine Suchrichtung entgegen der zulässigen Menge. Bei einer vollen Schrittlänge sollte dabei die nächste Iteration bereits in der zulässigen Menge liegen.

3.4 Pfad-Verfolgungs Verfahren

Bisher wurden nur Verfahren diskutiert, welche strikt dem Zentralen Pfad folgen. Wir fordern nun lediglich, dass unsere nächste Iterierte sich in einer Umgebung des Zentralen Pfads $\mathcal{C}$ befindet und dass sich die gewichtet Dualitätslücke $\nu_k = \frac{1}{n}\sum_{i=1}^{n} x_i^{(k)} s_i^{(k)}$ für $k \to \infty$ verkleinert.

Geeignete Umgebungen des Zentralen Pfads sind dabei

$$\mathcal{N}_2(\gamma) = \{(x, \lambda, \mu) \in \mathcal{F}_0 \mid \|XMe - \eta e\|_2 \leq \gamma\eta\},$$

$$\mathcal{N}_{-\infty}(\gamma) = \{(x, \lambda, \mu) \in \mathcal{F}_0 \mid x_i s_i \geq \gamma\eta \text{ für alle } i = 1, \dots, n\}$$

für $\gamma \in (0, 1]$, wobei oftmals $\gamma = 0.5$ für $\mathcal{N}_2(\gamma)$ und $\gamma = 10^{-3}$ für $\mathcal{N}_\infty(\gamma)$ gewählt wird. Für $\mathcal{N}_{-\infty}(\gamma)$ müssen also $x_i s_i$ größer sein als $\gamma\eta$, also eines Anteils ihres Durchschnitts. Das kann zur Folge haben, dass für ein groß gewähltes γ die Umgebung $\mathcal{N}_{-\infty}$ zumeist in der zulässigen Menge $\mathcal{F}$ liegt. Im Gegensatz dazu ist die Umgebung $\mathcal{N}_2$ deutlich restriktiver, da die strikt zulässige Menge $\mathcal{F}_0$ selbst für groß gewählte γ nicht in der Umgebung liegt.

Wir können nun, wie in [6] beschrieben, einen Algorithmus für ein zulässiges Pfad-Verfolgungs Verfahren formulieren.

Algorithmus 3.4.1. *1. Wähle $\gamma \in (0, 1)$, $0 < \underline{\sigma} < \overline{\sigma} < 1$, $(x_0, \lambda_0, \mu_0) \in \mathcal{N}_{-\infty}(\gamma)$ und $\varepsilon > 0$.*

2. *for k = 1,2,...*
3. *if $\eta_k < eps$:*
   STOPP
4. *Wähle $\sigma_k \in [\underline{\sigma}, \overline{\sigma}]$ und löse*

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{(k)} \\ \Delta\lambda^{(k)} \\ \Delta\mu^{(k)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -XMe + \sigma_k \eta_k e \end{pmatrix}.$$

5. *Wähle α_k so, dass*

$$(x^{(k+1)}, \lambda^{(k+1)}, \mu^{(k+1)}) = (x^{(k)}, \lambda^{(k)}, \mu^{(k)}) + \alpha_k(\Delta x^{(k)}, \Delta\lambda^{(k)}, \Delta\mu^{(k)}) \in \mathcal{N}_{-\infty}(\gamma).$$

6. *end (for)*

Bemerkung 3.4.2. Weiterhin können wir beispielsweise die Umgebung $\mathcal{N}_{-\infty}$ für unzulässige Verfahren erweitern, in dem wir zusätzlich fordern, dass die Normen der Residuen

$\|r_b\|_2$ und $\|r_c\|_2$ ebenfalls in einem Anteil der Dualitätslücke $\omega\eta$, für $\omega \in (0,1]$, liegen. Wir modifizieren also die Umgebung $\mathcal{N}_{-\infty}$ folgendermaßen um:

$$\mathcal{N}_{-\infty}(\gamma,\omega) = \Bigg\{(x,\lambda,\mu) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n | x_i\mu_i \geq \gamma\eta \text{ für } i = 1,\dots,n$$
$$\text{und } \|(r_b, r_c)\|_2 \leq \frac{\|(r_b^0, r_c^0\|_2}{\eta_0}\beta\eta\Bigg\}$$

und berücksichtigen die Residuen r_b und r_c im vierten Schritt in Algorithmus (3.4.1).

3.4.1 Konvergenzverhalten

Wir betrachten im folgenden kurzen Abschnitt, wie sich die Konvergenz des vorherigen Algorithmus verhält. Zunächst betrachten wir hierzu einige Resultate aus [8] und [6], um anschließend die Konvergenzeigenschaften zu untersuchen.

Lemma 3.4.3. *Sei $(x,\lambda,\mu) \in \mathcal{N}_{-\infty}(\gamma)$ für $\gamma \in (0,1]$. Dann gilt*

$$\|\Delta X \Delta M e\| \leq 2^{-3/2}(1 + 1/\gamma)n\eta.$$

Beweis. Aus [8, S. 89, Lemma 5.4] folgt

$$\|\Delta X \Delta M e\| \leq 2^{-3/2}\|(XM)^{-1/2}(-XMe + \sigma\eta e)\|^2.$$

Aus $x^T\mu = n\eta$ und $e^Te = n$ erhalten wir dann

$$\begin{aligned}
\|\Delta X \Delta M e\| &\leq 2^{-3/2}\| - (XM)^{1/2}e + \sigma\eta(XM)^{-1/2}e\|^2 \\
&\leq 2^{-3/2}\left(x^T\mu - 2\sigma\eta e^Te + \sigma^2\eta^2\sum_{i=1}^n (x_i\mu_i)^{-1}\right) \\
&\leq 2^{-3/2}\left(x^T\mu - 2\sigma\eta e^Te + \sigma^2\eta^2\frac{n}{\gamma\eta}\right) \\
&\leq 2^{-3/2}\left(1 - 2\sigma + \frac{\sigma^2}{\gamma}\right)n\eta \\
&\leq 2^{-3/2}(1 + 1/\gamma)n\eta.
\end{aligned}$$

□

Satz 3.4.4. *Sei $(x^{(k)}, \lambda^{(k)}, \mu^{(k)}) \in \mathcal{N}_{-\infty}(\gamma)$. Dann gilt*

$$(x^{(k)}(\alpha), \lambda^{(k)}(\alpha), \mu^{(k)}(\alpha)) \in \mathcal{N}_{-\infty}(\gamma)$$

Beweis. Siehe [6, S. 22] und [8, Ss 98]. □

Satz 3.4.5. *Gegeben sei* $(x^{(k)}, \lambda^{(k)}, \mu^{(k)})_{k\in\mathbb{N}_0}$ *eine durch Algorithmus* (3.4.1) *gegebene Folge, dann existiert eine Konstante* δ*, so dass*

$$\eta_{k+1} \leq (1 - \delta/n)\eta_k$$

gilt.

Beweis. Siehe [6, S. 22] und [8, S. 98]. □

Satz 3.4.6. *Gegeben sei* $(x^{(k)}, \lambda^{(k)}, \mu^{(k)})_{k\in\mathbb{N}_0}$ *eine durch Algorithmus* (3.4.1) *gegebene Folge, so dass für* $(x^{(0)}, \lambda^{(0)}, \mu^{(0)})$

$$\eta_0 \leq 1/\varepsilon^\rho$$

für $\rho > 0$ *gilt. Dann existiert ein* $K \in \mathbb{N}$ *mit* $K = O(n|log(\varepsilon)|)$ *und* $\eta_k \leq \varepsilon$ *für alle* $k \geq K$*.*

Beweis. Wir haben bereits gezeigt, dass

$$\eta_{k+1} \leq (1 - \delta/n)\eta_k$$

gilt. Per Voraussetzung erhalten wir durch Logarithmieren

$$\begin{aligned} \ln \eta_l &\leq \ln(1 - \delta/n) + \ln \eta_{k-1} \\ &\leq 2\ln(1 - \delta/n) + \ln \eta_{k-2} \\ &\leq k\ln(1 - \delta/n) + \ln \eta_0 \\ &\leq \rho \ln 1/\varepsilon \end{aligned}$$

Dann folgt aber aus $1 + \beta \leq \exp(\beta)$

$$\ln(1 + \beta) \leq \beta \quad \text{für alle} \quad \beta > -1$$

und wir bekommen

$$\ln \eta_k \geq k(-\delta/n)\rho log\varepsilon^{-1}$$

Dann gilt aber auch für

$$k(-\delta/n) + \rho \ln \varepsilon^{-1} \leq log\varepsilon$$

gerade $\eta_k \leq \varepsilon$. Durch Umstellen folgt damit dann aber auch schon die Bedingung

$$k \geq K = (1 + \rho)\frac{n}{\delta} \ln \varepsilon^{-1}.$$

□

3.5 Prädiktor-Korrektor Verfahren

Ein praktisches und auch weit verwendetes Innere-Punkte-Verfahren ist das Prädiktor-Korrektor-Verfahren, welches von Mehrotra in [4] beschrieben wird.

Bisherige Innere-Punkte-Verfahren iterierten dabei immer tangential entlang des Lösungspfad, wobei Mehrotras Verfahren im ersten Schritt oder auch affinen Schritt Selbiges macht. Im Korrekturschritt wird nun allerdings auch eine Krümmung des Lösungspfades berücksichtigt. Somit wird nach jeder Iteration der Lösungspfad in Richtung des Zentralen Pfads geschoben. Daher wird der zweite Schritt des Verfahrens häufig auch als zentrierender Schritt bezeichnet.

Weiterhin kann der zentrierende Parameter σ nach dem affinen Schritt adaptiv bestimmt werden, indem die Dualitätslücke betrachtet wird.

Wir definieren nun eine modifizierte Lösungskurve

$$\mathcal{H} := \{(x^{(k)}(s), \lambda^{(k)}(s), \mu^{(k)}(s))\mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \ : s \in [0, 1)\},$$

welche den verschobenen Zentralen Pfad darstellt und mit

$$(x^{(k)}(0), \lambda^{(k)}(0), \mu^{(k)}(0)) = (x^{(k)}, \lambda^{(k)}, \mu^{(k)})$$

sowie mit $s \to 1$ im Zentralen Pfad $\mathcal{C}$ liegt.

Konkret besteht das Verfahren also aus drei verschiedenen Schritten.

1. Bestimmung der affinen Schrittrichtung zur Bestimmung des zentrierenden Parameters σ.
2. Ein Korrekturschritt mit Hilfe der Information der Krümmung von $\mathcal{H}$.
3. Ein zentrierender Schritt mit Hilfe des zentrierenden Parameters.

Als Erstes bestimmen wir, wie in den vorherigen unzulässigen Verfahren, die affine Schrittrichtung mit $\sigma = 0$

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{\text{aff}} \\ \Delta \lambda^{\text{aff}} \\ \Delta \mu^{\text{aff}} \end{pmatrix} = \begin{pmatrix} -r_b \\ -r_c \\ -XMe + \sigma \eta e \end{pmatrix} \tag{3.9}$$

und bestimmen die größtmöglichen Schrittweiten, ohne das Positivitätskriterium zu verletzen. Diese können wir explizit mit

$$\alpha_{\text{aff}}^{\text{prim}} := \min \left(1, \min_{i:\, \Delta x_i > 0} \frac{-x_i}{\Delta x_i^{\text{aff}}}\right)$$

$$\alpha_{\text{aff}}^{\text{dual}} := \min \left(1, \min_{i:\, \Delta \mu_i > 0} \frac{-\mu_i}{\Delta \mu_i^{\text{aff}}}\right)$$

berechnen, wie in [6] gezeigt wird.

Um nun σ zu berechnen, betrachten wir wie sich die Dualitätslücke nach einem realisierten Schritt ändert. Dabei bezeichnen wir

$$\eta^{\text{aff}} := (x + \alpha_{\text{aff}}^{\text{prim}} \Delta x)^T (\mu + \alpha_{\text{aff}}^{\text{prim}} \Delta \eta)/n$$

als die affine Dualitätslücke und setzen damit unseren zentrierenden Parameter

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

Wir sehen: sollte der affine Schritt nun sehr effektiv sein, so ist $\mu^{\text{aff}} \ll \eta$ und damit σ sehr klein.

Im Korrekturschritt ersetzen wir die rechte Seite von (3.9) durch $(0, 0, -\Delta X^{\text{aff}} \Delta M^{\text{aff}} e)$ und im zentrierenden Schritt die rechte Seite durch $(0, 0, \sigma\eta e)$. Dabei können wir alle drei Schritte durch Addition vereinen und lösen letztendlich

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta \lambda \\ \Delta \mu \end{pmatrix} = \begin{pmatrix} -r_b \\ -r_c \\ -XMe - \Delta X^{\text{aff}} \Delta M^{\text{aff}} e + \sigma \eta e \end{pmatrix} \tag{3.10}$$

.

Zuletzt werden, analog zum Prädiktorschritt, die größtmöglichen Schrittweiten berechnet mit

$$\alpha^{\text{prim}} := \min \left(1, \min_{i:\, \Delta x_i > 0} \frac{-x_i}{\Delta x_i} \right)$$

$$\alpha^{\text{dual}} := \min \left(1, \min_{i:\, \Delta \mu_i > 0} \frac{-\mu_i}{\Delta \mu_i} \right)$$

und letztendlich die Schritte mit

$$\alpha_k^{\text{prim}} = \min(1, \nu \alpha^{\text{prim}}) \quad \text{und} \quad \alpha_k^{\text{dual}} = \min(1, \nu \alpha^{\text{dual}}) \tag{3.11}$$

für $\nu \in [0.9, 1)$ realisiert.

Wir können das Verfahren damit wie folgt beschreiben:

Algorithmus 3.5.1 (Prädiktor-Korrektor Verfahren). *1. Wähle* $(x^{(0)}, \lambda^{(0)}, \mu^{(0)}) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n$ *mit* $(x^{(0)}, \mu^{(0)}) > 0, \varepsilon > 0$

2. *for* $k = 1, 2, \dots$
3. *if* $\eta_k \leq \varepsilon$ *:*
   STOPP
4. *Löse*

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x^{aff} \\ \Delta \lambda^{aff} \\ \Delta \mu^{aff} \end{pmatrix} = \begin{pmatrix} -r_b \\ -r_c \\ -XMe + \sigma \eta e \end{pmatrix} .$$

5. *Berechne* α_{aff}^{prim} *und* α_{aff}^{dual} *mit*

$$\alpha_{aff}^{prim} := \min\left(1, \min_{i:\,\Delta x_i > 0} \frac{-x_i}{\Delta x_i^{aff}}\right),$$

$$\alpha_{aff}^{dual} := \min\left(1, \min_{i:\,\Delta \mu_i > 0} \frac{-\mu_i}{\Delta \mu_i^{aff}}\right).$$

6. *Berechne* η^{aff} *mit*

$$\eta^{aff} = (x + \alpha_{aff}^{prim}\Delta x)^T(\mu + \alpha_{aff}^{prim}\Delta\mu)/n.$$

7. *Setze* $\sigma_k = \left(\frac{\eta^{aff}}{\eta_k}\right)^3$.
8. *Löse*

$$\begin{pmatrix} 0 & A^T & I_n \\ A & 0 & 0 \\ M & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta\lambda \\ \Delta\mu \end{pmatrix} = \begin{pmatrix} -r_b \\ -r_c \\ -XMe - \Delta X^{aff}\Delta M^{aff}e + \sigma\eta e \end{pmatrix}.$$

9. *Berechne* α_k^{prim} *und* α^{dual} *mit*

$$\alpha^{prim} := \min\left(1, \min_{i:\,\Delta x_i > 0} \frac{-x_i}{\Delta x_i}\right)$$
$$\alpha^{dual} := \min\left(1, \min_{i:\,\Delta \mu_i > 0} \frac{-\mu_i}{\Delta \mu_i}\right)$$

und anschließend

$$\alpha_k^{prim} = \min(1, \nu\alpha^{prim}) \quad \textit{und} \quad \alpha_k^{dual} = \min(1, \nu\alpha^{dual})$$

für ein $\nu \in [0.9, 1)$.

10. *Setze*

$$(x^{(k+1)}, \lambda^{(k+1)}, \mu^{(k+1)}) = (x^{(k)}, \lambda^{(k)}, \mu^{(k)}) + (\alpha_k^{prim}\Delta x^{(k)}, \alpha_k^{dual}\Delta\lambda^{(k)}, \alpha_k^{dual}\Delta\mu^{(k)})$$

11. *end (for).*

3.6 Lösen des Newtongleichungssystems

Da in der Implementierung von Innere-Punkte Verfahren für das Lösen der Gleichungssysteme der meiste Rechenaufwand benötigt wird, ist man daran interessiert den Rechenprozess zu vereinfachen. Allerdings sind in den meisten Anwendungen oftmals Matrizen sehr groß und dünn besetzt, wodurch weitere Probleme auftreten können. Aufgrund der Struktur der Gleichungssysteme lassen sich diese jedoch in Einfachere herunterbrechen

und können somit effizienter gelöst werden. Da x, als auch μ strikt positiv sind, lässt sich das Gleichungssystem im Newtonschritt mit Hilfe von Blockelimination nach $\Delta\mu$ auflösen:

$$\begin{pmatrix} 0 & A \\ A^T & -D^{-2} \end{pmatrix} \begin{pmatrix} \Delta\lambda \\ \Delta x \end{pmatrix} = \begin{pmatrix} -r_b \\ -r_c + \mu - \sigma\eta X^{-1} e \end{pmatrix} \tag{3.12a}$$

$$\Delta\mu = -\mu + \sigma\eta X^{-1} e - X^{-1} M \Delta x \tag{3.12b}$$

mit $D = \mu^{-1/2} X^{1/2}$ und wird in der Literatur häufig als *augmented system* bezeichnet.

Hierbei ist zu erwähnen, dass die invertierten Diagonalmatrizen aufgrund des Positivitätskriteriums einfach zu berechnen sind.

Weiterhin können wir aufgrund der Struktur von X und M das Gleichungssystem nach Δx auflösen und erhalten somit ein Gleichungssystem von der Form:

$$AD^2 A^T \Delta\lambda = -r_b + A(-M^{-1} X r_c + x - \sigma\mu M^{-1} e) \tag{3.13a}$$

$$\Delta\mu = -r_c - A^T \Delta\lambda \tag{3.13b}$$

$$\Delta x = -x + \sigma\eta M^{-1} e - M^{-1} X \Delta\mu, \tag{3.13c}$$

welches in den meisten Implementierungen gelöst wird und in der Literatur oftmals als Normalform oder *normal form* bezeichnet wird. (3.13a) kann dabei sowohl mit der Least-Squares-Methode und dem CG-Algorithmus gelöst werden oder mit einem modifizierten Cholesky-Verfahren faktorisiert und dann sehr einfach gelöst werden. Das Cholesky-Verfahren ist hierbei sehr interessant, da bei einem Mehrotra Verfahren die Matrix AD^2A^T für zwei Schritte verwendet wird. Allerdings gilt zu beachten, dass die Diagonaleinträge von X und M sehr groß oder klein werden und herkömmliche Cholesky-Verfahren daran scheitern könnten.

4 Implementierung in Python

Im jetzigen Kapitel werden wir darauf eingehen wie wir Innere-Punkte Verfahren in Python implementieren können. Da Python open source und für objektorientiertes programmieren geeignet ist, ist Python, vor allem in der Wissenschaft, als auch in der Wirtschaft, sehr weit verbreitet. Für unsere Zwecke nutzen wir hauptsächlich die Libraries NumPy und SciPy, welche viele Tools für das numerische Rechnen liefern, allerdings kann Python auch für vielschichtiges objektorientiertes Programmieren verwendet werden, weshalb Python häufig für die Web Entwicklung verwendet wird.

Wir werden im folgenden, wie auch in [2], Python 3 verwenden und im späteren einen Vergleich zu dem Innere-Punkte-Verfahren aus der SciPy-Library machen. Dabei werden wir einfache Logistikprobleme, sowie das Klee-Minty-Problem betrachten, welches als Hauptmotivation für die Implementierung dienen wird.

Zunächst müssen, ähnlich zu Latex, die benötigten Pakete importiert werden. Dabei können auch einzelne Module eines Paketes importiert werden, sollte nicht alles benötigt werden.

Wir importieren dabei die Pakete mit dem *import* Befehl und benennen sie um in *np* und *sp*.

```
1   import numpy as np
2   import scipy as sp
```

Damit können wir nun alle Befehle die uns die jeweiligen Pakete liefern nutzen.

Für die Implementierung werden wir häufiger Funktionen definieren müssen:

```
1   def fun(a, b, c):
2       return a, b, c
```

Dabei liefert uns der *return* Befehl am Ende eines Durchlaufs die Variablen die wir als Output benötigen, in diesem Fall gerade den Input. Im Gegensatz zu Matlab müssen wir hier nun aber beachten, dass die Doppelpunkte richtig gesetzt werden. Außerdem gilt zu beachten, dass Python Einrückungen berücksichtigt.

Wir beginnen also die Implementierung unserer Verfahren mit der Definition unserer Hauptfunktion, welche wir in zukünftige Testprogramme importieren und abrufen wollen.

```
1   def interior_point(
2       c, A, b, method=0, maxiter=1000,
3       tol=1e-8,alpha0=0.9995
4       ):
```

Hierbei sind c, A, b gerade unsere Vektoren und Matrizen aus unserem LP, welches wir lösen wollen, welche als np.arrays definiert werden.

Method=0 gibt an, welches Verfahren schlussendlich gewählt werden soll. *Maxiter und tol* sind die Abbruchkriterien der Verfahren und *alpha0* gibt an, inwiefern die ermittelte Schrittweite letztendlich noch einmal gekürzt werden soll. Ist in der Definition bereits im

Input ein Wert einer Variable angegeben, so wird dieser Wert standardmäßig benutzt, sollte dieser nicht weiter spezifiziert werden.

Dabei gilt zu beachten, dass davon ausgegangen wird, dass das LP noch nicht in Standardform vorliegt. In der Hauptfunktion geschieht das als Erstes bevor das Problem an die einzelnen Verfahren weitergegeben wird:

```
1  m0,n0 = np.shape(A)
2
3  I = np.eye(m0)
4  A = np.hstack((A, I))
5  c = np.hstack((c, np.zeros(m0)))
6
7  c = np.array(c, dtype=np.float64)
8  A = np.array(A, dtype=np.float64)
9  b = np.array(b, dtype=np.float64)
10
11 if method==0:
12     sol = _predictor_corrector(c, A, b, tol, m0, n0, maxiter,
13     alpha0)
14 if method==1:
15     sol = _shortstep(c, A, b, m0, n0, tol, maxiter, alpha0)
16
17 return sol
```

Dabei steuern wir mit einer einfachen If-Verzweigung, welches der beiden Verfahren nun verwendet werden soll. Im nächsten Abschnitt beschäftigen wir uns mit einer Startpunktstrategie. Wir widmen uns zunächst der Implementierung eines einfachen Primalen-Dualen Verfahren und betrachten dabei auch die Berechnung der Normalengleichung, sowie der Schrittweiten. Darauf aufbauend werden wir die Implementierung des Prädiktor-Korrektor-Verfahrens betrachten.

4.1 Startpunkt Strategie

Aufgrund der einfacheren Implementierung von unzulässigen Verfahren, werden im Folgenden diese weiterhin diskutiert. Wie wir allerdings in Abschnitt 3.4.1 gesehen haben, spielt die Wahl eines geeigneten Startpunktes dennoch eine Rolle bei der Leistung des Verfahrens. Einige Implementierungen, wie zum Beispiel das MOSEK Interior-Point Verfahren (siehe [1]), welches im SciPy Paket implementiert wurde, starten das Verfahren „kalt“ und wählen zunächst immer einen trivialen Startpunkt, wie

$$(x, \lambda, \mu) = ((1, \dots, 1)^T, (0, \dots, 0)^T, (1, \dots, 1)^T) \tag{4.1}$$

um zumindest dem Positivitätskriterium zu genügen.

In den hier vorgestellten Implementierungen werden die Startpunkte ermittelt, wie sie in [4, S. 589], [8, S. 224] oder [7], von welchem der nachfolgende Code stammt, vorgestellt werden.

Der Einfachheit halber definieren wir nun $(x, y, z) := (x, \lambda, \mu)$. Außerdem werden alle

Unterfunktionen unserer Pythonimplementierung mit einem vorausgehenden Unterstrich gekennzeichnet.

Zuerst setzen wir

$$y^{(0)} = (AA^T)^{-1}c$$

und beginnen die Berechnung von

$$\tilde{z} = c - Ay^{(0)} \text{ und } \tilde{x} = A^T(AA^T)^{-1}b.$$

```
1  def _get_starting_point(c, A, b, n, m):
2      A2 = np.dot(A,A.T)
3      xb = np.linalg.solve(A2, b)
4      xt = np.dot(A.T,xb)
5      y0 = np.linalg.solve(A2,np.dot(A,c))
6      zt = c - np.dot(A.T,y0)
```

Anschließend wählen wir

$$\delta_x := \max(-1.5 \min_i(\tilde{x}_i), 0) \text{ und } \delta_z := \max(-1.5 \min_i(\tilde{z}_i), 0)$$

mit denen wir

$$\tilde{\delta}_x = \delta_x + 0.5 \frac{(\tilde{x} + \delta_x e)^T(\tilde{z} + \delta_z e)}{\sum_{i=1}^n (\tilde{z}_i + \delta_z)}$$

und

$$\tilde{\delta}_z = \delta_z + 0.5 \frac{(\tilde{x} + \delta_x e)^T(\tilde{z} + \delta_z e)}{\sum_{i=1}^n (\tilde{x}_i + \delta_x)}$$

berechnen. Zuletzt setzen wir $x_i^{(0)} = \tilde{x}_i + \tilde{\delta}_x$ und $z_i^{(0)} = \tilde{z}_i + \tilde{\delta}_z$ für alle $i = 1, \dots, n$.

```
1  dx = -1.5*np.min(xt)
2  dx = max(0,dx)
3
4  dz = -1.5*np.min(zt)
5  dz = max(0,dz)
6
7  xhat = xt + dx
8  zhat = zt + dz
9
10 xs = np.dot(xhat,zhat)
11 dxhat = 0.5*xs/np.sum(zhat)
12 dzhat = 0.5*xs/np.sum(xhat)
13
14 x0 = xhat + dxhat
15 z0 = zhat + dzhat
```

Abschließend überprüfen wir, ob die Startpunkte das Positivitätskriterium verletzen. Wir verwenden hier eine für Python typische Syntax, in dem wir mit

```
x0[x0 <= 0]
```

und

```
z0[z0 <= 0]
```

einen Vektor ausgeben lassen, welcher ausschließlich die negativen Werte der beiden Arrays ausgibt. Anschließen überprüfen wir mit dem Numpybefehl *np.any*, ob die beiden Arrays leer sind. Sollte einer der Vektoren einen negativen Eintrag haben, so wählen wir (4.1) als Startpunkt.

```
1  if np.any(x0[x0 <= 0]) or np.any(z0[z0 <= 0]):
2      x0, z0 = np.ones(n)
3      y0 = np.zeros(m)
4
5  return x0, y0, z0
```

Mit der Wahl eines geeigneten Startpunktes können wir nun mit der Implementierung der Innere-Punkte Verfahren beginnen.

4.2 Implementierung eines einfachen Primal-Dualen Verfahrens

Wir beginnen also zunächst mit der Definition einiger initialen Variablen:

```
1  def _shortstep(c, A, b, m0, n0, tol, maxiter=1000, alpha0=0.99):
2
3      # init
4      m, n = np.shape(A) # Dimensionen of matrix A
5      path = np.zeros((m0, maxiter+1)) # Solutionpath
6      status = 0 # status changes if the maximum of iterations is reached
7      sigma0 = 0.4 # centering parameter
8      iteration = 0 # Counter
9
10     # set maximum value for diagonal matrices
11     max_diag = 1e16
12     min_diag = 1e-16
13
14     # get starting point for x, lambda and s
15     x0, y0, z0 = _get_starting_point(c, A, b, n, m)
16
17     path[:,0] = x0[:m0]
```

Wir sehen, dass für das jeweilige Verfahren ein zentrierender Parameter von 0.4 gewählt worden ist und ein Zähler auf null gesetzt wird. Weiterhin werden maximale Einträge für die Diagonalmatrizen gewählt, da sonst bei der Lösung des Newtongleichungssystems die Matrizen sehr schlecht konditioniert oder gegebenenfalls singulär sein könnten. Bevor wir nun eine Approximation unserer Lösung iterieren, erzeugen wir zuletzt, wie in Abschnitt 4.1 gezeigt, einen Startpunkt.

Außerdem vermeiden wir - sofern dies möglich ist - die Verwendung von *For*-Schleifen. Diese sind, wie in [2] gezeigt wird, nicht so performant wie *While*-Schleifen. Wir beginnen also unsere Iterationen mit

```
while iteration <= maxiter:
```

Dabei ist jetzt zu beachten, dass der Iterationszähler am Ende der Schleife mit

```
iteration += 1
```

aufdatiert werden muss.

Wir beginnen mit der Definition unserer gewichteten Dualitätslücke, sowie den Residuen des unzulässigen Verfahrens.

```
# set duality gap
mu = np.mean(x0*z0)

# set residuals

# primal residual vector
r_P = np.dot(A,x0) - b
# dual residual vector
r_D = np.dot(np.transpose(A), y0) + z0 - c
# last entry on the right side
r_XZ = x0*z0 - sigma0*mu*np.ones(n)
```

Hier nutzen wir aus, dass Python bei der Multiplikation von Vektoren elementweise multipliziert und können uns das Definieren der Diagonalmatrizen X und M sparen.

Im nächsten Schritt definieren wir uns weitere Abbruchkriterien für unsere gewählte Toleranz

$$r_P^{\text{tol}} := \frac{\|r_P\|}{1 + \|b\|} \leq \text{tol}, \tag{4.2a}$$

$$r_D^{\text{tol}} := \frac{\|r_D\|}{1 + \|c\|} \leq \text{tol}, \tag{4.2b}$$

$$\text{rel} := \frac{|c^T x - b^T \lambda|}{1 + |c^T x|} \leq \text{tol}, \tag{4.2c}$$

nach [9, S. 22].

```
# set termination conditions
res_P = np.linalg.norm(r_P) / (1 + np.linalg.norm(b))
res_D = np.linalg.norm(r_D) / (1 + np.linalg.norm(c))
res_rel = np.abs(np.dot(c.T, x0) - np.dot(b.T, y0)) ...
    / (1 + np.abs(np.dot(c.T, x0)))

# check termination condition
if res_P < tol and res_D < tol and res_rel < tol:
    break
```

Wir überprüfen auch gleich zu Beginn, ob unser Startpunkt schon ausreichend nahe an unserer Lösung liegt und brechen gegebenenfalls die *While*-Schleife ab.

Das Lösen des Newtongleichungssystems wird in den meisten Fällen den größten Rechenaufwand des Prozesses verursachen. Daher ist es, wie bereits in Abschnitt 3.6 erwähnt,

empfehlenswert, diese in die Normalform zu bringen. Dabei definieren wir unsere Variablen wie folgend:

```
1  # get normalform
2  x0[x0 > max_diag] = max_diag
3  x0[x0 < min_diag] = min_diag
4  z0[z0 > max_diag] = max_diag
5  z0[z0 < min_diag] = min_diag
6  D = x0 / z0 # D^2
7  D[D > max_diag] = max_diag
8  Zinv = 1/z0
9  M = np.dot(A, np.dot(np.diag(D),np.transpose(A))) # AD^2A^T
10 b_y = -r_P + np.dot(A,(Zinv*r_XZ - Zinv*x0*r_D))
11 L = [] # will be used in mehrotra
12 I = [] # will be used in mehrotra
13
14 # get the newton direction
15 d_x, d_y, d_z = _get_direction(
16     A, L, M, b_y, r_D, r_XZ, Zinv, D, I, m
```

Wir schränken zuerst die Einträge unserer aktuell Iterierten ein, da extreme Werte zu Floatingfehlern bei der Berechnung von D^2 führen können und überprüfen nachträglich, ob keine extremen Werte in D^2 vorkommen. Zuletzt berechnen wir $M = AD^2A^T$ und die rechte Seite der ersten Normalengleichung (3.13a).

Wir betrachten nun die Solver für die Normalengleichung.

```
1  def _get_direction(A, L, M, b_y, r_D, r_XZ, Zinv, D, I, m):
2
3      err = 0
4      if np.any(L) and np.any(np.isnan(L)):
5          err = 1
6          try:
7              r = sp.linalg.solve_triangular(L,b_y)
8              d_y = sp.linalg.solve_triangular(L.T,r)
9          except:
10             err = 0
11     if err == 0:g
12         try:
13             d_y = np.linalg.lstsq(M, b_y, rcond=None)[0]
14         except:
15             d_y = sp.sparse.linalg.cg(M, b_y)[0]
16
17     d_z = -r_D - np.dot(np.transpose(A),d_y)
18     d_x = - Zinv * r_XZ - D*d_z
19
20     return d_x, d_y, d_z
```

Wir überprüfen zunächst, ob eine Cholesky-Zerlegung von AD^2A^T vorhanden ist. Da sich diese allerdings mehr für das Prädiktor-Korrektor Verfahren lohnt, wenden wir diese erst später an. Außerdem suchen wir in den Einträgen von L nach Fehlern und überprüfen, ob sich in L NaN-Einträge befinden. Wir nutzen, wie wir später in Abschnitt 4.3 sehen werden, herkömmliche Cholesky-Verfahren, welche für Innere-Punkte Verfahren häufig

ungeeignet sind. Eine modifizierte Variante wird in [9] vorgestellt.

Sollte also eine Lösung durch eine Cholesky-Zerlegung nicht möglich sein, so versuchen wir die Least-Square Methode anzuwenden, welche für singuläre Matrizen ebenfalls scheitern kann. Sollten beide Verfahren scheitern, so nutzen wir ein CG-Verfahren des SciPy Pakets und beginnen, nachdem alle Vektoren erfolgreich ausgegeben worden sind, mit der Schrittweitenberechnung in

```
1 # get the step size
2 a_x, a_z = _get_step(x0, z0, d_x, d_z)
```

Die Schrittweitensteuerung wird wie in [5] auf Seite 404 umgesetzt.

```
1  def _get_step(x0, z0, d_x, d_z):
2
3      d_xs = d_x[d_x < 0]
4      xs = x0[d_x < 0]
5
6      d_zs = d_z[d_z < 0]
7      zs = z0[d_z < 0]
8
9      a_x = np.min(xs / -d_xs) if np.any(d_xs) else 1
10     a_x = min(1, a_x)
11     a_z = np.min(zs / -d_zs) if np.any(d_zs) else 1
12     a_z = min(1, a_z)
13
14 return a_x, a_z
```

Sollten hier allerdings keine negativen Wert in Δx und Δz vorkommen, so wählen wir die volle Schrittweite. Alternativ wäre auch eine Backtracking Strategie für die Schrittweitensteuerung möglich.

Mit gegebener Schrittweite und -Richtung können wir die aktuelle Iterierte updaten:

```
1      # update the solution
2      x0 = x0 + a_x*d_x
3      y0 = y0 + a_z*d_y
4      z0 = z0 + a_z*d_z
5
6      # update path
7      path[:, iteration + 1] = x0[:m0]
8
9      iteration += 1
10
11 if iteration == maxiter:
12     status = 1
13
14     path = path[:, :iteration]
15
16 return {"x_opt": x0,
17         "f_opt": np.dot(c.T, x0),
18         "path": path,
19         "iteration": iteration,
20         "status": status,
21         "residuals": np.hstack((res_P, res_D, res_rel))}
```

Schlussendlich wird der neue Punkt in einem Pfad-Vektor gespeichert, damit die Möglichkeit besteht das Ergebnis zu plotten. Ausgegeben werden die Ergebnisse so, dass die einzelnen Elemente mit *Ergebnis('name')* abgerufen werden können.

4.3 Implementierung des Prädiktor-Korrektor Verfahrens

Ausgehend von der vorherigen Implementierung werden wir in diesem Abschnitt eine Implementierung eines Prädiktor-Korrektor Verfahrens nach Mehrotra betrachten. Da sich hierbei einige Dinge mit denen der vorherigen Implementierung überschneiden, soll hier näher auf die Unterschiede eingegangen werden.

Wir beginnen die Funktion mit

```
1  def _predictor_corrector(c, A, b, maxiter=1000, tol=1e-8, alpha0=0.9995):
```

Wobei wir hier zur späteren Schrittweitensteuerung ein *alpha0* fordern, welches dem Schrittweitenparameter ν aus (3.11) entspricht. Außerdem wählen wir für das Verfahren eine kleinere Toleranz.

```
1  def _predictor_corrector(c, A, b, tol, m0, n0 ,maxiter=1000, alpha0=0.9995):
2
3      m, n = np.shape(A)
4      path = []
5      status = 0
6      iteration = 0
7      # set maximum value for diagonal matrices
8      max_diag = 1e32
9      min_diag = 1e-32
10
11     # get starting point for x, lambda and s
12     x0, y0, z0 = _get_starting_point(c, A, b, n, m)
13     path[:,0] = x0[:m0]
```

Die initialen Variablen werden analog zu Abschnitt 4.2 definiert, mit dem Unterschied, das der zentrierende Parameter σ noch nicht gewählt werden muss. Dieser wird erst nach dem affinen Schritt gewählt und wird vorerst, wie wir im nächsten Teil sehen werden, gleich Null gesetzt.

```
1  while iteration <= maxiter:
2
3      # set residuals
4      r_P = np.dot(A,x0) - b
5      r_D = np.dot(np.transpose(A), y0) + z0 - c
6      r_XZ = x0*z0
```

Die Abbruchkriterien werden exakt wie in Abschnitt 4.2 gewählt. Der wesentliche Unterschied wird allerdings mit dem Lösen des Gleichungssystems klar. Da sich die rechte Seite des Gleichungssystems (3.13a) nach dem affinen Schritt nicht ändert, können wir eine Cholesky-Zerlegung für zwei Newtonschritte nutzen.

```
1  # get the cholesky decomposition of A*D²*A (if possible)
```

```
2  x0[x0 > max_diag] = max_diag
3  x0[x0 < min_diag] = min_diag
4  z0[z0 > max_diag] = max_diag
5  z0[z0 < min_diag] = min_diag
6  D = x0 / z0
7  D[D > max_diag] = max_diag
8  D[D < min_diag] = min_diag
9  Zinv = 1/z0
10 Zinv[Zinv > max_diag] = max_diag
11 Zinv[np.isnan(Zinv)] = max_diag
12 M = np.dot(A, np.dot(np.diag(D),np.transpose(A)))
13 b_y = -r_P + np.dot(A,(Zinv*r_XZ - Zinv*x0*r_D))
14 try:
15     L = sp.linalg.cholesky(M, lower=True)
16 except:
17     L = []
```

Wir erzeugen also, sofern es die Beschaffenheit der Diagonalmatrizen X und M erlaubt, vor dem ersten Newtonschritt bereits die mit Cholesky faktorisierte Dreiecksmatrix L mit $LL^T = AD^2A^T$, um mit Hilfe dieser durch Vorwärts- und Rückwärtssubstitution den Vektor $\Delta\lambda$ zu berechnen. Sollte dies allerdings nicht möglich sein, so werden wir analog zu Abschnitt 3.6 das Gleichungssystem lösen.

Im nächsten Schritt berechnen wir den affinen Schritt, wie in Abschnitt 3.6 mit $\sigma = 0$, um damit unser zentrierenden Parameter σ für den Korrekturschritt zu berechnen.

```
1  # get the affine newton direction
2  d_x, d_y, d_z = _get_direction(A, L, M, b_y, r_D, r_XZ, Zinv,
3      D, I, m)
4
5  # get the step size
6  a_x, a_z = _get_step(x0, z0, d_x, d_z)
7
8  # update the affine duality gap
9  mu_aff = np.mean((x0 + a_x*d_x)*(z0 + a_z*d_z))
10
11 # update sigma
12 sigma0 = pow((mu_aff/mu),3)
```

An dieser Stelle ergibt sich die Unterscheidung zum vorherigen Algorithmus. Wir updaten, wie in (3.10) beschrieben, mit dem jetzt gegebenen zentrierenden Parameter die Residuen für den Korrekturschritt und nutzen - falls möglich - unsere Cholesky-Zerlegung ein weiteres Mal.

```
1  # update the third residual for the corrector step
2  r_XZ = r_XZ + d_x*d_z - sigma0*mu
3
4  # update the right side of the normal equation
5  b_y = -r_P + np.dot(A,(Zinv*r_XZ - Zinv*x0*r_D))
6
7  # get the correcting newton direction
8  d_x, d_y, d_z = _get_direction(A, L, M, b_y, r_D, r_XZ, Zinv,
9      D, I, m)
```

Zum Schluss wird, nach der üblichen Ermittlung der Schrittweiten, diese mit Verwendung des Parameters ν (hier alpha0) weiter gestaucht.

```
1  # update the correcting step size
2  a_x, a_z = _get_step(x0, z0, d_x, d_z)
3  a_x, a_z = alpha0*a_x, alpha0*a_z
4
5  # update the solution
6  x0 = x0 + a_x*d_x
7  y0 = y0 + a_z*d_y
8  z0 = z0 + a_z*d_z
9
10 # update path
11 path[:, iteration + 1] = x0[:m0]
12
13 iteration += 1
```

5 Numerische Tests

Im letzten Kapitel werde ich noch ein wenig auf die Effizienz der vorgestellten Algorithmen eingehen und diese bei einem Beispiel und später beim Klee-Minty Problems, welches in [3] vorgestellt wurde, anwenden. Dabei werde ich hier die Linprog-Implementierung des SciPy-Pakets (s. [1]), dem Simplex-Algorithmus von Christian Jäkle aus [2] und den hier vorgestellten Algorithmen gegenüberstellen.

Zu Beginn testen wir die jeweiligen Algorithmen an

$$\begin{aligned} &\min c^T x \\ \text{u.d.Nb.}\quad &Ax \leq b \\ &x \geq 0, \end{aligned}$$

mit

$$c = \begin{pmatrix} 50 \\ -9 \\ -3 \\ 0 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 100 & 18 & 0 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 50 \\ 200 \\ 5000 \end{pmatrix}$$

und erhalten folgende Ergebnisse:

Verfahren	linprog	simplex	dual_primal	mehrotra
Zeit	0.004996	0.000999	0.009993	0.006995
Iterationen	5	3	36	12
f_{opt}	-2607.999770	2608.0	-2607.999986	-2607.999999

Hier ist schon auffällig, dass bereits deutlich mehr Iterationen für Innere-Punkte Verfahren notwendig

sind. Die Performance des Simplex-Algorithmus hatte sich bereits in der Praxis als sehr effizient erwiesen und übertrifft in den meisten Fällen die Innere-Punkte Verfahren, allerdings liegt die Motivation der Forschung an Innere-Punkte Verfahren in der Lösung von Problemen, welche für den Simplex-Algorithmus ungünstig sind. Denn Klee und Minty konnten mit ihrem Beispiel zeigen, dass die Komplexität des Simplex-Algorithmus im schlechtesten Fall exponentiell ansteigend ist.

5.1 Klee-Minty Problem

Zum Schluss betrachten wir noch eine einfache Form des Klee-Minty Problems, dass wie folgt aussieht:

$$\begin{aligned} \max \sum_{j=1}^{n} 2^{n-j} x_j \\ \text{u.d.Nb.} \quad \sum_{j=1}^{i-1} 2^{i-j} x_j + x_i \leq 5^{i-1} \\ x_i > 0 \end{aligned}$$

für $i = 1, \ldots, n$.

Das Problem wird wie folgt erzeugt:

```
1  n = 3
2
3  a = np.zeros(n)
4  c = np.zeros(n)
5  A = np.zeros((n,n))
6  b = np.zeros(n)
7
8  a[0] = 1
9  a[1] = 2
10
11 A[n-1][n-1] = 1
12 A[n-2:n,n-2] = np.copy(a[0:1])
13
14 k = 0
15 while k < n:
16     c[k] = (pow(2,(n-k-1)))
17     b[k] = pow(5, k)
18     if k >= 2:
19         a[k] = a[k-1]*2
20         A[n-k:n, n-k] = np.copy(a[0:k])
21     k += 1
22 A[0:n, 0] = np.copy(a)
```

Wir können damit also das Problem mit der Wahl von n beliebig dimensionieren. Wir werden allerdings schnell feststellen, dass das Klee-Minty Problem die hier vorgestellten Verfahren, sowie das Simplex-Verfahren, schnell unbrauchbar macht. Das ist aller-

dings nicht weiterhin verwunderlich, da das Klee-Minty Problem eben für diesen Zweck konstruiert wurde und für viele Algorithmen der linearen Optimierung das Worst-Case Szenario darstellt. Denn für das Klee-Minty Problem ist die Zeitkomplexität von exponentiellem Ausmaß, was wir in den folgenden Werten sehen werden.

	Verfahren	linprog	simplex	mehrotra
$n = 5$	Zeit	0.004996	0.001998	0.009994
	Iterationen	6	5	12
	f_{opt}	-624.999998	625.0	-624.999996
$n = 7$	Zeit	0.006995	0.001998	0.019987
	Iterationen	8	7	35
	f_{opt}	-15624.99	15625.0	-15624.99
$n = 9$	Zeit	0.007994	0.016988	0.002997
	Iterationen	9	9	27
	f_{opt}	-390624.99	390625.0	-390624.99
$n = 11$	Zeit	0.010992	0.007995	0.037975
	Iterationen	11	11	43
	f_{opt}	-9765624.94	9765625.0	-9765624.99
$n = 13$	Zeit	0.009993	0.004996	6.755798
	Iterationen	12	13	10000
	f_{opt}	-244140605.51	244140625.0	244140625.0
$n = 15$	Zeit	0.010993	0.005995	-
	Iterationen	12	15	-
	f_{opt}	-5548966355	6103515625	-
$n = 17$	Zeit	0.008994	-	-
	Iterationen	11	-	-
	f_{opt}	-2689669628	-	-

Wie vorauszusehen war, lässt sich das Innere-Punkte Verfahren des SciPy Moduls nicht von dem Klee-Minty Problem beeindrucken. Die hier beschriebene Implementierung bricht allerdings bereits bei einer Dimension von $n = 13$ ab und das Simplex-Verfahren ab einer Dimension von $n = 15$. Wir sehen, dass bei solchen Dimensionen sehr große Werte vorkommen können. Daher ist für eine effizientere Implementierung von Innere-Punkte Verfahren eine Präkonditionierung des Problems von Vorteil. Ebenso sollte eine modifzierte Version des Cholesky-Verfahrens verwendet werden. Es bleibt daher festzuhalten, dass die Innere-Punkte Verfahren gerade bei Worst-Case Szenarien sehr effiziente Werkzeuge sein können.

Literatur

[1] Erling D. Andersen and Knud D. Andersen. *The Mosek Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm*, pages 197–232. Springer, Boston, 2000.

[2] Christian Jäkle. *Simplex und das Branch-and-Bound-Verfahren mit Implementierung in Python*. Bachelorarbeit, Universität Konstanz, 2017.

[3] Victor Klee and George J. Minty. How good is the simplex algorithm? *Academic Press, New York*, pages 159–175, 1972.

[4] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–60, 1992.

[5] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, 1999.

[6] S. Volkwein. *Numerische Verfahren der restringierten Optimierung*. Vorlesungsmanuscript, Universität Konstanz, 2009.

[7] Yingwei Wang. *Computational Methods in Optimization*. Vorlesungsmanuscript, Purdue University, 2013.

[8] Stephen J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, New York, 1997.

[9] Stephen J. Wright. Modified cholesky factorizations in interior-point algorithms for linear programming. *SIAM Journal on Optimization*, 9(4):1159–1191, 1999.

6 Anhang

6.1 iplp.py

```python

import numpy as np
import scipy as sp



#==============================================================================
# get starting point
#==============================================================================

def _get_starting_point(c, A, b, n, m):
    """
    method to solve for a starting point as described in
    Sanjay Mehrotra. On the implementation of a primal-dual interior point
        method.
    SIAM Journal on Optimization, p-589591, 1992
    args:    A has to be 2-d numpy.array
             c, b has to be 1-d numpy.array
             A: is the matrix from the submitted
             c: is the costfunction from c^Tx
             b: is the vector from the equalities
    returns:
    returns a starting point (x0, y0, z0).
    """
    A2 = np.dot(A,A.T)
    xb = np.linalg.solve(A2, b)
    xt = np.dot(A.T,xb)
    y0 = np.linalg.solve(A2,np.dot(A,c))
    zt = c - np.dot(A.T,y0)

    dx = -1.5*np.min(xt,0)
    dx = max(0,dx)

    dz = -1.5*np.min(zt,0)
    dz = max(0,dz)

    xhat = xt + dx
    zhat = zt + dz

    xs = np.dot(xhat,zhat)
    dxhat = 0.5*xs/(np.sum(zhat))
    dzhat = 0.5*xs/(np.sum(xhat))

    x0 = xhat + dxhat
    z0 = zhat + dzhat

    if np.any(x0[x0 <= 0]) or np.any(z0[z0 <= 0]):
        x0 = np.ones(n)
        z0 = np.ones(n)
```

```
49            y0 = np.zeros(m)
50
51        return x0, y0, z0
52
53
54
55 #==========================================================================
56 # modified cholesky
57 #==========================================================================
58
59 def _modchol(M, m):
60     '''
61     modified cholesky decomposition from [1] for a m*m symmetric positive
62     definite matrix A
63
64     '''
65
66     # set initial values
67     E = []
68     beta = max(np.diag(M)) # ... determines which cholesky steps are skipped
69     I = [] # .................... safes all processed indices
70     L = np.zeros((m,m)) # ...... cholesky factor
71     tol2 = 1e-8 # ............... tolerance for skipping elimination
72
73     for i in range(m):
74
75         #skip elimination process
76         if M[i,i] <= beta*tol2:
77             E = np.zeros((m,m))
78             E[i, i:m] = M[i, i:m]
79             E[i:m, i] = M[i:m, i]
80             M = M - E
81
82         #normal cholesky elimination process
83         else:
84             C = M.copy()
85             L[i,i] = np.sqrt(C[i,i])
86             for j in range(i+1,m):
87                 L[j,i] = C[i,j] / L[i,i]
88             for j in range(i+1,m):
89                 for k in range(i+1,m):
90                     M[j,k] = C[j,k] - L[j,i]*L[k,i]
91             # safe processed index
92             I.append(i)
93
94     return L, I
95
96
97
98 #==========================================================================
99 # get newton direction
100 #==========================================================================
101
102 def _get_direction(A, L, M, b_y, r_D, r_XZ, Zinv, D, I, m):
```

```
103     """
104     method to solve for a single newton direction
105     args:   A, L, M has to be 2-d numpy.array
106             b_y, r_D, r_XZ, Zinv, D has to be 1-d numpy.array
107             A: is the matrix from the submitted
108             L: is the lower triangular cholesky matrix
109             M: is the positive definite matrix A*D²*AT, which is only used if
110                 d_y can not be solved with the cholesky decomposition
111             b_y: is the righthand side of the normal equation
112             r_D: is the dual residual
113             r_XZ: is the third residual
114             Zinv: is the diagonal matrix with the inverse
115                 elements from z0.
116             D: is the diagonal matrix with the elements from
117                 x0/z0.
118             I: are the non-zero indices from the modified cholesky
119                 decomposition.
120     returns:
121     returns a full newton-step for x, y and z.
122     """
123     err = 0
124     if np.any(L) and np.any(np.isnan(L)):
125         err = 1
126         try:
127             r = sp.linalg.solve_triangular(L,b_y)
128             d_y = sp.linalg.solve_triangular(L.T,r)
129         except:
130             err = 0
131     if err == 0:
132         try:
133             d_y = np.linalg.lstsq(M, b_y, rcond=None)[0]
134         except:
135             d_y = sp.sparse.linalg.cg(M, b_y)[0]
136 
137     d_z = -r_D - np.dot(np.transpose(A),d_y)
138     d_x = - Zinv * r_XZ - D*d_z
139 
140     return d_x, d_y, d_z
141 
142 
143 
144 #==============================================================================
145 # get step size
146 #==============================================================================
147 
148 def _get_step(x0, z0, d_x, d_z):
149     """
150     method to solve for the primal and dual stepsize as described in
151     Jorge Nocedal and Stephen J. Wright. Numerical Optimization.
152     Springer Series in Operations Research and Financial Engineering.
153     Springer, Berlin-Heidelberg, 2 edition, p405, 2008.
154     args:   x0, z0, d_x, d_z has to be 1-d numpy.array
155             x0, z0: is the last iteration.
156             d_x, d_y: is the updated newton direction
```

```
157     returns:
158     returns a primal and dual stepsize.
159     """
160 
161     d_xs = d_x[d_x < 0]
162     xs = x0[d_x < 0]
163 
164     d_zs = d_z[d_z < 0]
165     zs = z0[d_z < 0]
166 
167     a_x = np.min(xs / -d_xs) if np.any(d_xs) else 1
168     a_x = min(1, a_x)
169     a_z = np.min(zs / -d_zs) if np.any(d_zs) else 1
170     a_z = min(1, a_z)
171 
172     return a_x, a_z
173 
174 
175 
176 #===========================================================================
177 # mehrothra's interior-point method
178 #===========================================================================
179 
180 def _predictor_corrector(c, A, b, tol, m0, n0 ,maxiter=1000, alpha0=0.9995):
181     """
182     interior-point method to solve problems like min c^Tx s.T. Ax = b
183     args:       c, b has to be 1-d numpy.array,
184                 N has to be a 2-d numpy.array
185                 c: from the costfunction c^Tx
186                 N: is the matrix from the submitted
187                 b: is the vector of the inequalities
188     opt args:   maxiter = 1000:
189                     sets the maximum amount of iterations.
190                 tol = 1e-8:
191                     sets the maximum tolerance for the termination conditions
                        .
192                 alpha0 = 0.9995:
193                     dampens the final stepsize in the corrector step.
194     returns:
195     if the problem is feasible and bound than it returns the optimum, if not,
196     then it return the status unbound or infeasible:
197     packages:   numpy as np
198     """
199 
200     m, n = np.shape(A)
201     path = np.zeros((m0, maxiter+1))
202     status = 0
203     iteration = 0
204     # set maximum value for diagonal matrices
205     max_diag = 1e64
206     min_diag = 1e-64
207 
208     # get starting point for x, lambda and s
209     x0, y0, z0 = _get_starting_point(c, A, b, n, m)
```

```
210     x0 = np.ones(np.shape(x0))
211     y0 = np.ones(np.shape(y0))
212     z0 = np.ones(np.shape(z0))
213 
214     path[:,0] = x0[:m0]
215 
216 
217     while iteration <= maxiter:
218 
219         # set residuals
220         r_P = np.dot(A,x0) - b
221         r_D = np.dot(np.transpose(A), y0) + z0 - c
222         r_XZ = x0*z0
223 
224         # set duality gap and termination conditions
225         mu = np.mean(r_XZ)
226 
227         res_P = np.linalg.norm(r_P) / (1 + np.linalg.norm(b))
228         res_D = np.linalg.norm(r_D) / (1 + np.linalg.norm(c))
229         res_rel = np.abs(np.dot(c.T, x0) - np.dot(b.T, y0)) / (1 + np.abs(np.
            dot(c.T, x0)))
230 
231         # check termination conditions
232         if mu < tol and res_P < tol and res_D < tol and res_rel < tol:
233             break
234 
235         # get the cholesky decomposition of A*D²*A (if possible)
236         x0[x0 > max_diag] = max_diag
237         x0[x0 < min_diag] = min_diag
238         z0[z0 > max_diag] = max_diag
239         z0[z0 < min_diag] = min_diag
240         D = x0 / z0
241         D[D > max_diag] = max_diag
242         D[D < min_diag] = min_diag
243         Zinv = 1/z0
244         Zinv[Zinv > max_diag] = max_diag
245         Zinv[np.isnan(Zinv)] = max_diag
246         M = np.dot(A, np.dot(np.diag(D),np.transpose(A)))
247         b_y = -r_P + np.dot(A,(Zinv*r_XZ - Zinv*x0*r_D))
248         L = sp.linalg.cholesky(M, lower=True)
249         I = []
250 
251         # get the affine newton direction
252         d_x, d_y, d_z = _get_direction(A, L, M, b_y, r_D, r_XZ, Zinv, D, I, m
            )
253 
254         # get the step size
255         a_x, a_z = _get_step(x0, z0, d_x, d_z)
256 
257         # update the affine duality gap
258         mu_aff = np.mean((x0 + a_x*d_x)*(z0 + a_z*d_z))
259 
260         # update sigma
261         sigma0 = pow((mu_aff/mu),3)
```

```
262
263         # update the third residual for the corrector step
264         r_XZ = r_XZ + d_x*d_z - sigma0*mu*np.ones(n)
265
266         # update the right side of the normal equation
267         b_y = -r_P + np.dot(A,(Zinv*r_XZ - Zinv*x0*r_D))
268
269         # get the correcting newton direction
270         d_x, d_y, d_z = _get_direction(A, L, M, b_y, r_D, r_XZ, Zinv, D, I, m
                )
271
272         # update the correcting step size
273         a_x, a_z = _get_step(x0, z0, d_x, d_z)
274         a_x, a_z = alpha0*a_x, alpha0*a_z
275
276         # update the solution
277         x0 = x0 + a_x*d_x
278         y0 = y0 + a_z*d_y
279         z0 = z0 + a_z*d_z
280
281         if np.any([x0 < 0]) or np.any([z0 < 0]):
282             break
283
284         # update path
285         path[:, iteration] = x0[:m0]
286
287         iteration += 1
288
289     if iteration == maxiter:
290         status = 1
291         message = "The method could not find an optimal solution.",\
292         "The problem is probably not bounded."
293     else:
294         message = "The program terminated succesfully."
295
296     path = path[:, :iteration]
297
298     return {"x_opt": x0,
299             "f_opt": np.dot(c.T, x0),
300             "path": path,
301             "iterations": iteration,
302             "status": status,
303             "message": message,
304             "residuals": np.hstack((res_P, res_D, res_rel))}
305
306
307 def _shortstep(c, A, b, m0, n0, tol, maxiter=1000, alpha0=0.99):
308     """
309     interior-point method to solve problems like min c^Tx s.T. Ax = b
310     args:       c, b has to be 1-d numpy.array,
311                 N has to be a 2-d numpy.array
312                 c: from the costfunction c^Tx
313                 N: is the matrix from the submitted
314                 b: is the vector of the inequalities
```

```
315         opt args:    maxiter = 1000:
316                          sets the maximum amount of iterations.
317                      tol = 1e-8:
318                          sets the maximum tolerance for the termination conditions
                             .
319                      alpha0 = 0.9995:
320                          dampens the final stepsize in the corrector step.
321         returns:
322         if the problem is feasible and bound than it returns the optimum, if not,
323         then it return the status unbound or infeasible:
324         packages:    numpy as np
325         """
326 
327         # init
328         m, n = np.shape(A) # Dimensionen of matrix A
329         path = np.zeros((m0, maxiter+1)) # Solutionpath
330         status = 0 # status changes if the maximum of iterations is reached
331         sigma0 = 0.4 # centering parameter
332         iteration = 0 # Counter
333 
334         # set maximum value for diagonal matrices
335         max_diag = 1e32
336         min_diag = 1e-32
337 
338         # get starting point for x, lambda and s
339         x0, y0, z0 = _get_starting_point(c, A, b, n, m)
340 
341         path[:,0] = x0[:m0]
342 
343         while iteration <= maxiter:
344 
345             # set duality gap
346             mu = np.mean(x0*z0)
347 
348             # set residuals
349             r_P = np.dot(A,x0) - b # primal residual vector
350             r_D = np.dot(np.transpose(A), y0) + z0 - c # dual residual vector
351             r_XZ = x0*z0 - sigma0*mu*np.ones(n) # last entry on the right side
352 
353             # set termination conditions
354             res_P = np.linalg.norm(r_P) / (1 + np.linalg.norm(b))
355             res_D = np.linalg.norm(r_D) / (1 + np.linalg.norm(c))
356             res_rel = np.abs(np.dot(c.T, x0) - np.dot(b.T, y0)) \
357                 /(1 + np.abs(np.dot(c.T, x0)))
358 
359             # check termination condition
360             if mu < tol and res_P < tol and res_D < tol and res_rel < tol:
361                 break
362 
363             # get normalform
364             x0[x0 > max_diag] = max_diag
365             x0[x0 < min_diag] = min_diag
366             z0[z0 > max_diag] = max_diag
367             z0[z0 < min_diag] = min_diag
```

```
368            D = x0 / z0 # D^2
369            D[D > max_diag] = max_diag
370            D[D < min_diag] = min_diag
371            Zinv = 1/z0 # Z^-1
372            M = np.dot(A, np.dot(np.diag(D),np.transpose(A))) # AD^2A^T
373            M[M > max_diag] = max_diag
374            M[M < min_diag] = min_diag
375            b_y = -r_P - np.dot(A,(D*r_D - Zinv*r_XZ)) # first eq. of normal form
376            L = [] # will be used in mehrotra
377            I = [] # will be used in mehrotra
378
379            # get the newton direction
380            d_x, d_y, d_z = _get_direction(A, L, M, b_y, r_D, r_XZ, Zinv, D, I, m
                   )
381
382            # get the step size
383            a_x, a_z = _get_step(x0, z0, d_x, d_z)
384
385            # update the solution
386            x0 = x0 + a_x*d_x
387            y0 = y0 + a_z*d_y
388            z0 = z0 + a_z*d_z
389
390            # update path
391            path[:, iteration] = x0[:m0]
392
393            iteration += 1
394
395        if iteration == maxiter:
396            status = 1
397
398        path = path[:, :iteration]
399
400        return {"x_opt": x0,
401                "f_opt": np.dot(c.T, x0),
402                "path": path,
403                "iterations": iteration,
404                "status": status,
405                "residuals": np.hstack((res_P, res_D, res_rel))}
406
407
408
409
410    def interior_point(c, A, b, method=0, tol=1e-5, maxiter=10000, alpha0=0.95):
411
412        m0,n0 = np.shape(A)
413
414        I = np.eye(m0)
415        A = np.hstack((A, I))
416        c = np.hstack((c, np.zeros(m0)))
417
418        c = np.array(c, dtype=np.float64)
419        A = np.array(A, dtype=np.float64)
420        b = np.array(b, dtype=np.float64)
```

```
421
422     if method==0:
423         sol = _predictor_corrector(c, A, b, tol, m0, n0, maxiter, alpha0)
424     if method==1:
425         sol = _shortstep(c, A, b, m0, n0, tol, maxiter, alpha0)
426
427     return sol
```

6.2 klee_minty_test.py

```
1  import time
2  import numpy as np
3  from scipy.optimize import linprog
4  from iplp import interior_point
5  from simplex import simplex
6  import matplotlib.pyplot as plt
7
8
9  comptime_ip = [] # ........computational time for inter-point
10 comptime_linprog = [] # ...computational time for linprog
11 comptime_sim = [] # .......computational time for simplex
12
13 n = 17
14
15 a = np.zeros(n)
16 c = np.zeros(n)
17 A = np.zeros((n,n))
18 b = np.zeros(n)
19
20 a[0] = 1
21 a[1] = 2
22
23 A[n-1][n-1] = 1
24 A[n-2:n,n-2] = np.copy(a[0:1])
25
26 k = 0
27 while k < n:
28     c[k] = (pow(2,(n-k-1)))
29     b[k] = pow(5, k)
30     if k >= 2:
31         a[k] = a[k-1]*2
32         A[n-k:n, n-k] = np.copy(a[0:k])
33     k += 1
34 A[0:n, 0] = np.copy(a)
35
36 #c = np.array([-50, -9, -3, 0])
37 #A = np.array([[1, 0, 1, 0],
38 #              [0, 1, 0, 1],
39 #              [100, 18, 0, 0]])
40 #b = np.array([50, 200, 5000])
41
42
43 #test linprog optimization
44 start_linprog = time.time()
```

```
45  P_linprog = linprog(
46          -c, A_ub = A, b_ub = b, method='interior-point',
47          options={"maxiter": 10000}
48          )
49  comptime_linprog.append(time.time() - start_linprog)
50
51
52  #test interior point optimization
53  start_ip = time.time()
54  P_ip = interior_point(-c, A, b, method=0)
55  comptime_ip.append(time.time() - start_ip)
56
57
58  #test simplex optimization
59  start_sim = time.time()
60  P_sim = simplex(c, A, b)
61  comptime_sim.append(time.time() - start_sim)
62
63
64  x = P_ip['path']
65  y = P_linprog['x']
66
67  figure = plt.figure()
68
69  axes = figure.add_axes([0.2, 0.2, 0.8, 0.8])
70
71  axes.plot(x[0,:P_ip['iterations']],
72      x[1,:P_ip['iterations']], 'bo--')
73  #axes.plot(x[0,0], x[1,0], 'g*')
74  #axes.plot(x[0,-1], x[1,-1], 'r*')
75  #axes.plot(y[0], y[1], 'r*')
76
77
78
79
80  print("------------------------------------------------------------")
81  print('comptime␣by␣using␣scipy␣linprog_ip:␣', comptime_linprog)
82  print('comptime␣by␣using␣interior␣point:␣', comptime_ip)
83  print('comptime␣by␣using␣simplex:␣', comptime_sim)
84  print("------------------------------------------------------------")
85  print('iterations␣by␣using␣scipy␣linprog_ip:␣', P_linprog['nit'])
86  print('iterations␣by␣using␣interior␣point:␣', P_ip['iterations'])
87  print('iterations␣by␣using␣simplex:␣', P_sim['iterations'])
88  print("------------------------------------------------------------")
89  print('f_opt␣by␣using␣scipy␣linprog_ip:␣', P_linprog['fun'])
90  print('f_opt␣by␣using␣interior␣point:␣', P_ip['f_opt'])
91  print('f_opt␣by␣using␣simplex:␣', P_sim['f_opt'])
92  print("------------------------------------------------------------")
```