

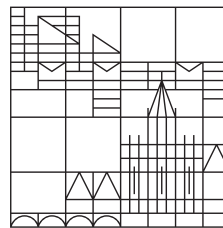
Implementierung des L-BFGS-B-Verfahrens in Python

Bachelorarbeit

vorgelegt von
Simon Buchwald

an der

Universität
Konstanz



Mathematisch-Naturwissenschaftlichen Sektion
Fachbereich Mathematik und Statistik

Gutachter: Prof. Dr. Stefan Volkwein

Konstanz, 2017

Inhaltsverzeichnis

| | |
|---|-----------|
| Abbildungsverzeichnis | 3 |
| Zusammenfassung | 4 |
| 1 Theorie des L-BFGS-B-Verfahrens | 5 |
| 1.1 Die Problemstellung | 5 |
| 1.2 Notwendige Optimalitätsbedingungen | 5 |
| 1.3 Hinreichende Optimalitätsbedingung | 6 |
| 1.4 Das projizierte Gradientenverfahren | 7 |
| 1.5 Konvergenzanalyse des projizierten Gradientenverfahrens | 8 |
| 1.6 Projiziertes limited-memory BFGS-Verfahren | 9 |
| 1.7 Konvergenzanalyse des L-BFGS-B-Verfahrens | 12 |
| 2 Implementierung in Python | 13 |
| 2.1 Grundlagen der Implementierung | 13 |
| 2.1.1 Allgemeine Anmerkungen zu Python | 13 |
| 2.1.2 Programmbibliotheken und die Import-Funktion | 13 |
| 2.1.3 Strukturierung von Skripten | 14 |
| 2.1.4 Vektoren und Matrizen | 15 |
| 2.1.5 Dictionaries | 16 |
| 2.1.6 Schleifen | 16 |
| 2.2 Implementierung des L-BFGS-B-Verfahrens | 16 |
| 2.2.1 Liniensuche bzw. projizierter Armijo-Algorithmus | 17 |
| 2.2.2 bfgsrecb | 18 |
| 2.2.3 L-BFGS-B-Verfahren | 19 |
| 2.2.4 Beispiel für ein main-file | 21 |
| 2.2.5 Vergleich zum projizierten Gradientenverfahren | 23 |
| 3 Beispiel aus der optimalen Steuerung | 24 |
| 3.1 Problemstellung | 24 |
| 3.2 Implementierung | 25 |
| 3.3 Resultate | 26 |
| 3.4 Vergleich zum projizierten Gradientenverfahren | 28 |
| Fazit | 30 |
| Literaturverzeichnis | 31 |
| Anhang | 32 |
| Ausformulierte Python-Codes | 32 |

Abbildungsverzeichnis

| | | |
|----|---|----|
| 1 | Contour-Plot Rosenbrock-Funktion | 22 |
| 2 | Vergleich proj. Gradienten- und L-BFGS-B-Verfahren anhand der Rosenbrock-Funktion | 23 |
| 3 | Zustandslösung y von Bsp. 3.3.1 | 26 |
| 4 | Kontrolllösung u von Bsp. 3.3.1 | 26 |
| 5 | Zustandslösung y von Bsp. 3.3.2 | 27 |
| 6 | Kontrolllösung u von Bsp. 3.3.2 | 27 |
| 7 | Zustandslösung y von Bsp. 3.3.3 | 27 |
| 8 | Kontrolllösung u von Bsp. 3.3.3 | 27 |
| 9 | Zustandslösung y von Bsp. 3.3.4 | 28 |
| 10 | Kontrolllösung u von Bsp. 3.3.4 | 28 |
| 11 | Zustand y | 28 |
| 12 | Kontrolle u | 28 |
| 13 | Vergleich proj. Gradienten- und L-BFGS-B-Verfahren anhand des Beispiels aus der optimalen Steuerung | 29 |

Zusammenfassung

Diese Arbeit bietet einen Überblick über das mathematische Programmieren in Python im Vergleich zu Matlab am Beispiel des L-BFGS-B-Verfahrens. Dessen Betrachtung ist stark durch die langsame Konvergenzgeschwindigkeit des projizierten Gradientenverfahren motiviert, was wir in den Abschnitten 2.2.5 und 3.4 veranschaulichen werden.

Im ersten Kapitel werden wir jedoch zuerst die theoretischen Grundlagen beider Verfahren zusammenstellen und damit die mathematische Motivation des L-BFGS-B-Verfahrens liefern.

Das zweite Kapitel beschäftigt sich dann mit der Implementierung des Verfahrens in Python. Aufbauend auf der Annahme von Grundkenntnissen in Matlab werden wir hier Gemeinsamkeiten und Unterschiede zu Python herausarbeiten und diskutieren, weshalb es sich lohnen kann, Python Matlab vorzuziehen.

Im dritten Kapitel wird ein Ausblick in die optimale Steuerung, anhand eines einfachen Beispiels einer elliptischen Differenzialgleichung unter Verwendung des Finite-Differenzen-Verfahrens gegeben.

Da die Implementierung im Vordergrund stehen soll, werden wir im Theorieteil auf die Beweise verzichten und bei der Konvergenzanalyse nur die wichtigsten Ergebnisse präsentieren. Eine ausführliche Konvergenzanalyse inklusive Beweise für das projizierte Gradientenverfahren findet sich in [Kel99, S.91-96], sowie zu Teilen in [GK02, S.294-302].

Für die Implementierung werden wir Python 3 verwenden.

KAPITEL 1

Theorie des L-BFGS-B-Verfahrens

1.1 Die Problemstellung

Wir betrachten das restringiertes Optimierungsproblem

$$\min_{x \in \Omega} f(x), \tag{1.1}$$

wobei $\Omega = \{x \in \mathbb{R}^N \mid L_i \leq (x)_i \leq U_i \text{ für } 1 \leq i \leq N\}$ und f eine ausreichend glatte und reellwertige Funktion $f : \mathbb{R}^N \rightarrow \mathbb{R}$ sind.

Die Menge Ω wird dabei als zulässige Menge und die Ungleichungen $L_i \leq (x)_i \leq U_i$ als Ungleichungs-Nebenbedingungen bezeichnet.

Wir sagen, die i -te Nebenbedingung ist *aktiv*, falls $(x)_i = L_i$ oder $(x)_i = U_i$ gilt. Ist die i -te Nebenbedingung nicht aktiv, so bezeichnen wir sie als *inaktiv*.

Mit $\mathcal{A}(x)$ bzw. $\mathcal{I}(x)$ bezeichnen wir die Menge der aktiven bzw. inaktiven Nebenbedingungen für ein gegebenes $x \in \mathbb{R}^N$.

Da die Menge Ω offensichtlich kompakt ist, hat (1.1) immer mindestens eine Lösung.

1.2 Notwendige Optimalitätsbedingungen

Anders als bei der unrestringierten Optimierung muss der Gradient der Funktion an der Lösung nicht unbedingt gleich 0 sein, falls die Lösung ein Randpunkt ist. Die notwendigen Optimalitätsbedingungen müssen daher angepasst werden. Dazu führen wir die Begriffe *stationärer Punkt* und *reduzierte Hesse-Matrix* ein.

Definition 1.2.1

- i) Ein Punkt $x^* \in \Omega$ heißt *stationärer Punkt* von Problem (1.1), falls gilt

$$\nabla f(x^*)^T (x - x^*) \geq 0 \quad \text{für alle } x \in \Omega. \tag{1.2}$$

- ii) Sei f zweimal stetig differenzierbar an $x \in \Omega$. Die Matrix $\nabla_R^2 f(x)$ mit den Einträgen

$$(\nabla_R^2 f(x))_{ij} = \begin{cases} \delta_{ij} & \text{falls } i \in \mathcal{A}(x) \text{ oder } j \in \mathcal{A}(x), \\ (\nabla^2 f(x))_{ij} & \text{sonst} \end{cases}$$

heißt *reduzierte Hesse-Matrix*.

Damit formulieren wir die notwendigen Optimalitätsbedingungen erster und zweiter Ordnung.

Satz 1.2.2 (Notwendige Optimalitätsbedingung erster Ordnung)

Sei f stetig differenzierbar auf Ω und sei x^* eine Lösung des Problems (1.1). Dann ist x^* ein *stationärer Punkt* von (1.1).

Beweis. Siehe [Kel99, S.88]

Satz 1.2.3 (Notwendige Optimalitätsbedingung zweiter Ordnung)

Seien f zweimal Lipschitz-stetig differenzierbar und x^* sei Lösung von (1.1). Dann ist die reduzierte Hesse-Matrix $\nabla_R^2 f(x^*)$ positiv semidefinit.

Beweis. Siehe [Kel99, S.89]

Um die für den Algorithmus relevante notwendige Optimalitätsbedingung formulieren zu können benötigen wir noch den Begriff der *Projektion*, in unserem Fall die Abbildung P auf den (in der l^2 -Norm) nächsten Punkt in Ω mit

$$P(x)_i = \begin{cases} L_i & \text{falls } (x)_i \leq L_i \\ (x)_i & \text{falls } L_i < (x)_i < U_i \\ U_i & \text{falls } (x)_i \geq U_i \end{cases} \quad (1.3)$$

Damit folgt

Satz 1.2.5 (Notwendige Optimalitätsbedingung)

Sei f stetig differenzierbar. Ein Punkt $x^* \in \Omega$ ist genau dann ein *stationärer Punkt* von (1.1), wenn

$$x^* = P(x^* - \lambda \nabla f(x^*))$$

für alle $\lambda \geq 0$ gilt.

Beweis. Siehe [Kel99, S.96]

1.3 Hinreichende Optimalitätsbedingung

Für die hinreichende Optimalitätsbedingung werden wir sowohl die bereits eingeführte reduzierte Hesse-Matrix als auch eine etwas stärkere Definition des stationären Punktes verwenden. Dafür machen wir uns klar, dass, falls x^* ein stationärer Punkt ist, für alle $i \in \mathcal{I}(x^*)$ gilt, dass $x^* \pm t e_i \in \Omega$ für t klein genug (e_i der i -te Einheitsvektor). Mit (1.2) gilt

$$\left. \frac{df(x^* \pm t e_i)}{dt} \right|_{t=0} = \pm \nabla f(x^*)^T e_i \geq 0$$

und somit

$$(\nabla f(x^*))_i = 0 \text{ für alle } i \in \mathcal{I}(x^*)$$

Damit formulieren wir die Definition eines nicht-degenerierten stationären Punktes.

Definition 1.3.1

Ein Punkt $x^* \in \Omega$ heißt nicht-degenerierter stationärer Punkt für Problem (1.1), falls x^* ein stationärer Punkt ist, und es gilt

$$(\nabla f(x^*))_i \neq 0 \text{ für alle } i \in \mathcal{A}(x^*). \quad (1.4)$$

Falls x^* zusätzlich Lösung von (1.1) ist, nennt man x^* ein nicht-degeneriertes lokales Minimum.

Bemerkung 1.3.2

- Die Bedingung (1.4) nennt man auch strikte Komplementaritäts-Bedingung.
- Für eine Menge S von Indizes definieren wir

$$(P_S x)_i = \begin{cases} (x)_i & \text{falls } i \in S \\ 0 & \text{falls } i \notin S \end{cases} \tag{1.5}$$

Die hinreichende Bedingung lässt sich damit ähnlich wie beim unrestringierten Fall formulieren.

Satz 1.3.3 (Hinreichende Optimalitätsbedingung)

Seien $x^* \in \Omega$ ein nicht-degenerierter stationärer Punkt von Problem (1.1), f zweimal stetig differenzierbar in einer Umgebung von x^* und die reduzierte Hesse-Matrix an x^* positiv definit. Dann ist x^* eine Lösung von (1.1) (und damit ein nicht-degeneriertes lokales Minimum).

Beweis. Siehe [Kel99, S.90]

1.4 Das projizierte Gradientenverfahren

Wir wollen zuerst das projizierte Gradientenverfahren einführen, auf dem das L-BFGS-B-Verfahren beruht, um die grundlegende Idee projizierter Verfahren zu verstehen. Das projizierte Gradientenverfahren funktioniert grundsätzlich genauso, wie das normale Gradientenverfahren und hat daher auch dieselben Vor- und Nachteile. Die Unterschiede sind die Abbruchbedingung, bei der wir den Fall eines Randpunktes mitberücksichtigen müssen und die Schrittweitensuche, bei der wir beachten müssen, dass wir innerhalb der zulässigen Menge bleiben.

Für die aktuelle Iterierte x_c wählen wir die neue Iterierte

$$x_+ = P(x_c - \lambda \nabla f(x_c)),$$

wobei $\lambda > 0$ ein Schrittweiten-Parameter eines Liniensuchverfahrens ist. Wir werden für die Ermittlung von λ den Armijo-Algorithmus verwenden. Um diesen anwenden zu können, müssen wir jedoch zuerst klären, was wir in unserem Fall unter einem *hinreichenden Abstieg* verstehen. Für $\lambda > 0$ definieren wir dafür

$$x(\lambda) = P(x - \lambda \nabla f(x)) \text{ für } x \in \Omega.$$

Die hinreichende Abstiegsbedingung schreiben wir damit als

$$f(x(\lambda)) - f(x) \leq -\alpha \nabla f(x)^T (x - x(\lambda)) \text{ für } x \in \Omega, \tag{1.6}$$

wobei $\alpha \in (0, 1)$.

Bei der Abbruchbedingung orientieren wir uns an der notwendigen Optimalitätsbedingung aus Satz 1.2.5. Dieser besagt, dass im Minimum $x(\lambda) = x$ für alle $\lambda > 0$ gilt. Wir wählen $\lambda = 1$ und brechen den Algorithmus ab falls

$$\|x^k - x^k(1)\| \leq \tau_a + \tau_r \|x^0 - x^0(1)\|, \quad 0 < \tau_a \leq \tau_r, \tag{1.7}$$

wobei wir mit τ_a und τ_r die absolute und relative Toleranz bezeichnen.

Damit können wir den Algorithmus formulieren:

Algorithmus 1.4.1 (*projiziertes Gradientenverfahren*)**Eingabe** : $x^0 \in \mathbb{R}^N$, $nmax, \beta \in (0, 1)$, $\alpha \in (0, 1)$, $\tau_a, \tau_r > 0$

```

1 for  $k = 0, 1, \dots, nmax$  do
2   while  $\|x^k - x^k(1)\| > \tau_a + \tau_r \|x^0 - x^0(1)\|$  do
3     Finde das kleinste  $m \in \mathbb{N}$ , sodass für  $\lambda^k = \beta^m$  gilt
        $f(x^k(\lambda^k)) - f(x^k) \leq -\alpha \nabla f(x^k)^T (x^k - x^k(\lambda^k))$ ;
4     Setze  $x^{k+1} = x^k(\lambda^k)$ ,  $k = k + 1$ ;
5   end
6 end

```

Bemerkung 1.4.2

- Für den unrestringierten Fall entspricht die Abstiegsbedingung (1.6) gerade der bekannten Armijo-Bedingung aus der Optimierung, denn dann gilt

$$x(\lambda) = P(x - \lambda \nabla f(x)) = x - \lambda \nabla f(x)$$

und somit

$$-\alpha \nabla f(x)^T (x - x(\lambda)) = -\alpha \lambda \|\nabla f(x)\|^2.$$

1.5 Konvergenzanalyse des projizierten Gradientenverfahrens

Wir wollen damit beginnen, dass die Abstiegsbedingung (1.6) wohldefiniert ist, dass also $f(x^{k+1}) < f(x^k)$ gilt.

Satz 1.5.1Sei $x \in \Omega$ und $\lambda > 0$. Dann gilt

$$\nabla f(x)^T (x - x(\lambda)) \geq \frac{1}{\lambda} \|x - x(\lambda)\|^2$$

Beweis. Siehe [Kel99, S.93f.].Als nächstes benötigen wir, dass die Abstiegsbedingung Schrittweiten $\lambda > 0$ liefert.**Satz 1.5.2**Seien ∇f Lipschitz-stetig mit Lipschitz-Konstante $L > 0$, $x \in \Omega$ und $\alpha \in (0, 1)$. Dann gilt die Abstiegsbedingung (1.6) für alle λ mit

$$0 < \lambda \leq \frac{2(1 - \alpha)}{L}$$

Beweis. Siehe [Kel99, S.94f.].

Somit existiert in jeder Iteration k ein $m_k \in \mathbb{N}$, sodass die Schrittweite $\lambda^k = \beta^{m_k}$ die Abstiegsbedingung erfüllt, sofern $\beta \in (0, 1)$ gilt.

Abschließend lässt sich feststellen, dass das projizierte Gradientenverfahren global konvergiert:

Satz 1.5.3

Jeder Häufungspunkt einer durch das projizierte Gradientenverfahren erzeugten Folge $\{x^k\}_k$ ist ein stationärer Punkt.

Ist f konvex, so ist dieser Häufungspunkt bereits Lösung des Problems (1.1).

Beweis. Siehe [GK02, S.300ff.].

Bemerkung 1.5.4

Wie im unrestringierten Fall erhalten wir somit ein global konvergentes Verfahren, dass jedoch häufig eine langsame Konvergenzgeschwindigkeit aufweist. Aus diesem Grund wird es in der Praxis normalerweise mit lokal schnell konvergenten Verfahren kombiniert, z.B.

1.6 Projiziertes limited-memory BFGS-Verfahren

Auch das projizierte BFGS-Verfahren ist dem unrestringierten sehr ähnlich und bringt, wie wir sehen werden, auch dessen Vorteile mit sich. Die limited-memory-Version spart zudem in hohen Dimensionen Rechenpeicher, indem in jedem Schritt nicht die approximierte Hessematrix gespeichert wird, sondern nur die Vektoren-Folgen $\{y_k\}_k$ und $\{s_k\}_k$ und damit rekursiv die neue Abstiegsrichtung berechnet wird.

Erinnerung BFGS

Die neue Iterierte des BFGS Algorithmus ist

$$x^{k+1} = x^k - \lambda^k H_k^{-1} \nabla f(x^k)$$

wobei λ^k wieder der Schrittweiten-Parameter eines Liniensuchverfahrens ist. Das Update der approximierten Hessematrix H_k ist durch die BFGS-Formel

$$H_{k+1} = H_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{H_k s_k (H_k s_k)^T}{s_k^T H_k s_k} \quad (1.8)$$

gegeben, mit $s_k = x^{k+1} - x^k$, $y_k = \nabla f(x^{k+1}) - \nabla f(x^k)$. Wir wollen nun eine Update-Formel für die Inverse $B_{k+1} = H_{k+1}^{-1}$. Dafür schreiben wir (1.8) um zu

$$H_{k+1} = \underbrace{H_k}_A + \underbrace{\begin{bmatrix} H_k s_k & y_k \end{bmatrix}}_U \underbrace{\begin{bmatrix} \frac{1}{s_k^T H_k s_k} & 0 \\ 0 & \frac{1}{y_k^T s_k} \end{bmatrix}}_{V^T} \begin{bmatrix} s_k^T H_k \\ y_k^T \end{bmatrix}.$$

Nun können wir die Sherman-Morrison-Woodbury-Formel¹ anwenden, dafür vernachlässigen wir aus Gründen der Übersichtlichkeit in den nächsten Zeilen den Index k .

$$\begin{aligned} B_{k+1} &= H_{k+1}^{-1} = (A + UV^T)^{-1} = A^{-1} - A^{-1}U(I + V^T A^{-1}U)^{-1}V^T A^{-1} \\ &= B - B \begin{bmatrix} Hs & y \end{bmatrix} \left(I + \begin{bmatrix} -\frac{1}{s^T Hs} & 0 \\ 0 & \frac{1}{y^T s} \end{bmatrix} \begin{bmatrix} s^T H \\ y^T \end{bmatrix} B \begin{bmatrix} Hs & y \end{bmatrix} \right)^{-1} \begin{bmatrix} -\frac{1}{s^T Hs} & 0 \\ 0 & \frac{1}{y^T s} \end{bmatrix} \begin{bmatrix} s^T H \\ y^T \end{bmatrix} B \\ &= B - \begin{bmatrix} s & By \end{bmatrix} \left(\begin{bmatrix} -s^T Hs & 0 \\ 0 & y^T s \end{bmatrix} + \begin{bmatrix} s^T H \\ y^T \end{bmatrix} B \begin{bmatrix} Hs & y \end{bmatrix} \right)^{-1} \begin{bmatrix} s^T \\ y^T B \end{bmatrix} \end{aligned}$$

¹ [NW99, S.605]

$$\begin{aligned}
&= B - [s \quad By] \left(\begin{bmatrix} 0 & s^T y \\ y^T s & y^T s + y^T By \end{bmatrix} \right)^{-1} \begin{bmatrix} s^T \\ y^T B \end{bmatrix} \\
&= B - [s \quad By] \begin{bmatrix} -\frac{y^T By}{(y^T s)^2} - \frac{1}{y^T s} & \frac{1}{y^T s} \\ \frac{1}{y^T s} & 0 \end{bmatrix} \begin{bmatrix} s^T \\ y^T B \end{bmatrix} \\
&= B + \frac{sy^T Bys^T}{(y^T s)^2} + \frac{ss^T}{y^T s} - \frac{sy^T B}{y^T s} - \frac{Bys^T}{y^T s} \\
&= (I - \rho sy^T) B (I - \rho ys^T) + \rho ss^T,
\end{aligned}$$

mit $\rho = \frac{1}{y^T s}$.

Limited-memory BFGS

Diese Darstellung können wir uns nun zunutze machen, um die Limited-Memory-Version des BFGS-Verfahrens aufzustellen. Dazu machen wir uns klar, dass die Matrix B_k im BFGS-Algorithmus allein dazu da ist, das Produkt $B_k \nabla f(x^k)$ zu berechnen. Dieses Produkt wollen wir mit der obigen Darstellung von B_k nun stattdessen rekursiv mit den Vektoren-Folgen $\{y_k\}_k$ und $\{s_k\}_k$ berechnen. Dazu betrachten wir $B_k d$ für ein allgemeines $d \in \mathbb{R}^N$, es gilt

$$\begin{aligned}
B_k d &= ((I - \rho_k s_k y_k^T) B_{k-1} (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T) d \\
&= (I - \rho_k s_k y_k^T) B_{k-1} (d - \rho_k y_k s_k^T d) + \rho_k s_k s_k^T d,
\end{aligned}$$

mit $\alpha_k = \rho_k s_k^T d$ folgt

$$B_k d = (I - \rho_k s_k y_k^T) B_{k-1} (d - \alpha_k y_k) + \alpha_k s_k,$$

setze $\tilde{d} = (d - \alpha_k y)$, dann

$$B_k d = (I - \rho_k s_k y_k^T) B_{k-1} \tilde{d} + \alpha_k s_k.$$

Dies können wir nun rekursiv mit $B_{k-1} \tilde{d}$ fortführen, bis wir bei $B_0 \tilde{d}$ ankommen. B_0 ist vorgegeben (im Normalfall $B_0 = I$), also können wir das Produkt $B_0 \tilde{d}$ ausrechnen, damit wieder vorwärts auflösen und erhalten

$$\begin{aligned}
B_k d &= (I - \rho_k s_k y_k^T) \hat{d} + \alpha_k s_k \\
&= \hat{d} + (\alpha_k - \rho_k y_k^T \hat{d}) s_k
\end{aligned}$$

Insgesamt bekommen wir so einen Algorithmus zur rekursiven Berechnung von $B_k d$:

Algorithmus 1.6.1 bfgsrec

Eingabe : $ns \in \mathbb{N}$, $\{s_k, y_k\}_{k=0}^{ns-1}$, B_0 , d

- 1 **if** $ns=0$ **then**
- 2 $d = B_0 d$;
- 3 **return** d
- 4 **end**
- 5 $\alpha = s_{ns-1}^T d / (y_{ns-1}^T s_{ns-1})$, $d = d - \alpha y_{ns-1}$;
- 6 **call** bfgsrec($ns - 1$, $\{s_k\}$, $\{y_k\}$, B_0 , d);
- 7 $d = d + (\alpha - (y_{ns-1}^T d / (y_{ns-1}^T s_{ns-1}))) s_{ns-1}$;
- 8 **return** d

Da wir das Speichern einer Matrix mit diesem Algorithmus ja gerade umgehen wollten, sollte ein geeignete Matrix B_0 , oder zumindest eine Speicher sparende Methode zur Berechnung von $B_0 d$ gewählt werden.

Hat man z.B. $B_0 = I$ kann man die zweite Zeile des Algorithmus streichen und die Eingabe der Matrix entfällt komplett. Gespeichert werden dann im Vergleich zum normalen BFGS-Verfahren nur die Vektoren-Folgen $\{y_k\}$ und $\{s_k\}$, also ist der Speicherbedarf $2mn$ ($m \hat{=}$ Anzahl der Iterationen) gegen $n^2 + 2n$ für die approximierte inverse Hessematrix und die Vektoren y_k, s_k beim normalen BFGS-Verfahren. Für große Dimension n lohnt sich dieser Algorithmus also, solange man davon ausgehen kann, dass das Verfahren in wenigen Schritten zur Lösung findet.

Bemerkung 1.6.2

- Die Matrix B_0 kann auch in jeder Iteration des L-BFGS-Verfahrens neu gewählt werden, bspw. als die skalierte Einheitsmatrix $\frac{s_{ns-1}^T y_{ns-1}}{y_{ns-1}^T y_{ns-1}} I$. Dies benötigt bei geschickter Implementierung nur n zusätzlich zu speichernde Einträge und scheint sehr effizient zu sein².
- Es gibt Versionen des Algorithmus, die nur eine Höchstzahl an Vektoren speichern, egal wie viele Iterationen absolviert werden. Um dies zu erreichen wird beim Erreichen der Höchstzahl entweder die Anzahl an Vektoren beibehalten, indem man alte überschreibt, oder es werden alle gelöscht und man fängt wieder von vorne an die Vektoren-Folge aufzubauen. Das letztere Vorgehen werden wir in der Implementierung in Kapitel 2 verwenden.

Projiziertes limited-memory BFGS

Im projizierten Fall approximieren wir nun natürlich die reduzierte Hesse-Matrix aus Definition 1.2.1. Betrachtet man die Struktur dieser Matrix, erkennt man, dass die Multiplikation $\nabla_{\mathcal{I}}^2 f(x) d$ alle Einträge von d mit aktivem Index erhält und nur die Einträge mit inaktivem Index ändert. Daher können wir uns in der rekursiven Berechnung von $B_k d$ auf die inaktiven Indizes beschränken und die aktiven später wieder hinzufügen (siehe Algorithmus 1.6.4). Auch bei den Vektoren s und y sind nur die inaktiven Einträge interessant, da wir nur in inaktiven Richtungen positive Definitheit benötigen

Algorithmus 1.6.3 bfgsrech

Eingabe : $ns \in \mathbb{N}, \{s_k, y_k\}_{k=0}^{n-1}, B_0, d, P_{\mathcal{I}}$

- 1 $d = P_{\mathcal{I}} d;$
- 2 **if** $n = 0$ **then**
- 3 $d = B_0 d;$
- 4 **return** d
- 5 **end**
- 6 $s_{n-1} = P_{\mathcal{I}} s_{n-1}, y_{n-1} = P_{\mathcal{I}} y_{n-1};$
- 7 $\alpha = s_{n-1}^T d / y_{n-1}^T s_{n-1};$
- 8 $d = d - \alpha y_{n-1};$
- 9 **call** $\text{bfgsrec}(n-1, \{s_k\}, \{y_k\}, B_0, d);$
- 10 $d = d + (\alpha - (y_{n-1}^T d / y_{n-1}^T s_{n-1})) s_{n-1};$
- 11 $d = P_I d;$
- 12 **return** d

Für den L-BFGS-B-Algorithmus benötigen wir nun noch die sogenannte ϵ -aktive Menge. Sie ist definiert als

$$\mathcal{A}^\epsilon(x) = \{i \mid U_i - x_i \leq \epsilon \vee x_i - L_i \leq \epsilon\}$$

²<http://www.desy.de/~blobel/lvmini.pdf>, S.8, letzter Zugriff am 02.10.2017

mit $\epsilon = \min(\min(U_i - L_i)/2, \|x - x(1)\|)$ und überschätzt die Menge der tatsächlich aktiven Indizes. Das garantiert, dass die Abstiegsbedingung (1.6) auch dann noch gilt, wenn man in $x(\lambda)$ den Gradienten mit einer positiv semidefiniten Matrix multipliziert, denn für das projizierte L-BFGS-Verfahren lautet die neue Iterierte natürlich

$$x^{k+1} = P(x^k - \lambda^k B_k \nabla f(x^k)) =: x^{B_k}(\lambda).$$

und die hinreichende Abstiegsbedingung somit

$$f(x^{B_k}(\lambda)) - f(x^k) \leq -\alpha \nabla f(x^k)^T (x^k - x^{B_k}(\lambda)).$$

Damit können wir nun den Algorithmus aufstellen:

Algorithmus 1.6.4 L-BFGS-B - Verfahren

Eingabe : f handle, $x^0 \in \mathbb{R}^N$, $\tau_a, \tau_r > 0$, $nmax$, $\alpha, \beta \in (0, 1)$, $L, U \in \mathbb{R}^N$

```

1   $ns = n = 0$ ;  $pg_0 = pg = x - P(x - \nabla f(x))$ ;
2   $\epsilon = \min(\min(U_i - L_i)/2, \|pg\|)$ ;  $\mathcal{A} = \mathcal{A}^\epsilon(x)$ ;  $\mathcal{I} = \mathcal{I}^\epsilon(x)$ ;
3  while  $\|pg\| \geq \tau_a + \tau_r \|pg_0\|$  and  $n < nmax$  do
4       $d = -\nabla f(x)$ ;
5       $d = bfgsrecb(ns, \{y_k\}, \{s_k\}, l, d, P_{\mathcal{I}})$ ;
6       $d = -P_{\mathcal{A}} \nabla f(x) + d$ ;
7      Finde das kleinste  $m \in \mathbb{N}$ , sodass (1.6) für  $\lambda = \beta^m$  gilt;
8       $xp = P(x + \lambda d)$ ;
9       $pg = xp - P(xp - \nabla f(xp))$ ;
10      $\epsilon = \min(\min(U_i - L_i)/2, \|pg\|)$ ;  $\mathcal{A} = \mathcal{A}^\epsilon(x)$ ;  $\mathcal{I} = \mathcal{I}^\epsilon(x)$ ;
11      $y_{ns} = P_{\mathcal{I}}(\nabla f(xp) - \nabla f(x))$ ;  $s_{ns} = P_{\mathcal{I}}(x(\lambda) - x)$ ;
12     if  $y_{ns}^T s_{ns} > 0$  then
13          $ns = ns + 1$ 
14     else
15          $ns = 0$ 
16     end
17      $x = xp$ ;  $n = n + 1$ 
18 end

```

In Zeile 6 sehen wir das oben angesprochene Wiederhinzufügen der aktiven Einträge. Dies bedeutet letztendlich, dass, wenn keine Vektoren s und y im Speicher sind, heißt im Fall $ns = 0$, dieses Verfahren gerade dem projizierten Gradientenverfahren entspricht, da dann die Zeilen 5 und 6 den Vektor d am Ende unverändert lassen.

Die Zeilen 12-16 besagen, dass die neuen Vektoren s_{ns} und y_{ns} nur gespeichert werden, wenn sie die positive Definitheitsbedingung $y_{ns}^T s_{ns} > 0$ erfüllen, ansonsten wird der komplette Speicher gelöscht und man geht erst einmal wieder zum projizierten Gradientenverfahren über.

1.7 Konvergenzanalyse des L-BFGS-B-Verfahrens

Wir erhalten erneut globale Konvergenz³. Das L-BFGS-B-Verfahren ist jedoch im Gegensatz zum projizierten Gradientenverfahren zusätzlich lokal (und in einigen Fällen sogar global) superlinear konvergent³. Wie entscheidend dieser Unterschied sein kann, werden wir in Kapitel 2.2.5 sehen.

³ [Kel99, S.104]

KAPITEL 2

Implementierung in Python

Wir wollen erst die wichtigsten Grundlagen der Programmierung in Python im Vergleich zu Matlab diskutieren, bevor wir zur Implementierung des L-BFGS-B-Verfahrens kommen. Für eine Implementierung des oben angesprochenen projizierten Gradientenverfahrens siehe [Buc17].

2.1 Grundlagen der Implementierung

2.1.1 Allgemeine Anmerkungen zu Python

Für die Installation von Python gibt es mehrere Möglichkeiten, abhängig vom Betriebssystem. Eine einfache, Variante, die für alle gängigen Betriebssysteme funktioniert, ist die Installation über das Paket *Anaconda*. Dieses lässt sich auf <https://www.anaconda.com/download/> herunterladen und enthält neben Python und allen wichtigen Python-Bibliotheken auch die grafische Benutzeroberfläche *Spyder*, die sich gerade für Matlab-Nutzer anbietet. Was die Version angeht, so bezieht sich diese Arbeit auf Python 3, vieles lässt sich aber auch auf Python 2 übertragen. Als Konsole wird hier die *IPython console* verwendet.

2.1.2 Programmbibliotheken und die Import-Funktion

Python ist im Vergleich zu Matlab kostenlos und hat zudem ein breiteres Anwendungsgebiet. Dies beruht unter Anderem darauf, dass jeder Nutzer eigene Programm- und Befehlspakete (auch „Programmbibliotheken“ genannt) erstellen und einbinden kann. In der Konsequenz kann die Eindeutigkeit von Befehlen nicht wie in Matlab ohne weiteres gewährleistet werden. Gelöst wird dieses Problem, indem man die Programmbibliothek, aus der man einen Befehl nutzen möchte, erst importieren und ihren Namen jedes Mal (mit einem Punkt getrennt) vor den Befehl schreiben muss. Damit es nicht zu unübersichtlich wird, kann man den Bibliotheken Abkürzungen geben.

Wir werden bspw. einige Befehle aus der Programmbibliothek *numpy*, die viele Vektor- und Matrixoperationen enthält, verwenden und geben ihr daher die Abkürzung *np*.

Will man Konstanten wie π , trigonometrische Funktionen oder die Exponentialfunktion verwenden, so benötigt man zusätzlich die Bibliothek *math*.

Im Code sieht dies wie folgt aus:

```
import numpy as np
import math
```

Beachte: Sowohl in Skripten als auch in der IPython-Konsole müssen die Bibliotheken importiert werden, bevor man Befehle daraus verwenden kann.

Die Import-Funktion erlaubt es aber auch, einzelne Befehle aus Programmbibliotheken, anderen Python-Versionen oder -Skripten zu importieren. Haben wir bspw. die Funktion *test* in dem

Skript „*costfunctions.py*“ definiert, können wir diese in ein anderes Skript importieren, indem wir am Anfang die folgende Zeile einfügen.

```
from costfunctions import test
```

Das bedeutet aber auch, dass der Name einer Funktion nicht wie in Matlab an den Skript-Namen gekoppelt sein muss, und erlaubt es somit, mehrere Funktionen in einem Skript zu definieren. Eine empfehlenswerte Strukturierung ist daher alle Beispielfunktionen in ein Skript „*costfunctions.py*“, alle Methoden in „*methods.py*“, die Visualisierung (Erstellen von Plots und Co.) in „*visualization.py*“ und die Parameterbelegung sowie das Aufrufen der Funktionen und Methoden wie in Matlab in ein main-file (im obigen Beispiel etwa „*main_test.py*“) zu schreiben.

2.1.3 Strukturierung von Skripten

Es gibt weitere wesentliche Unterschiede zwischen Matlab und Python, gerade was die Strukturierung innerhalb von Skripten angeht. Die wichtigsten sind:

- Python zeigt (Zwischen-)Ergebnisse nur in der Konsole an, wenn dies explizit durch den *print* Befehl gefordert wird. Beispiel:

```
In [1]: a = 2
In [2]: print(a)
2
```

Das Beenden von Zeilen mit Semikolon entfällt somit gegenüber Matlab komplett.

- Python arbeitet ohne *end*-Befehl und verwendet stattdessen Einrückungen für Schleifen und Funktionen. Innerhalb einer Schleife bzw. Funktion befindet sich alles, was hinter dem Doppelpunkt (der nach den Bedingungen bzw. der Definition kommt) und in der nächsten „Einrückungsebene“ steht. Das Ende einer Schleife wird dadurch markiert, dass wieder zur ursprünglichen Ebene zurückgekehrt wird. Beispiel:

```
def test(a):
    if a < 0:
        a = 0
    return a
```

- Wie gerade gesehen beginnen Funktionen mit dem Befehl *def* und enden mit *return*, wobei der Output das ist, was nach *return* steht.

Kommentierung funktioniert in Python mit dem *#*-Symbol, das Hervorheben bestimmter Bereiche wie in Matlab mit *\$\$* ist nicht möglich.

Ähnlich wie in Matlab gibt es auch in Python die Möglichkeit, Funktionen eine Beschreibung zu geben, die dann mit dem *help*-Befehl abrufbar sind. Dafür muss man die Beschreibung in die erste Zeile nach der Definition einer Funktion schreiben und entweder mit einfachen oder, falls die Beschreibung mehrzeilig ist, dreifachen Anführungszeichen einrahmen, z.B.

```
def test(a):
    """Dies ist eine
       mehrzeilige Beschreibung"""
    return a
```

2.1.4 Vektoren und Matrizen

Die erste wichtige Notiz für alle Matlab-Nutzer ist, dass Python mit seiner Indizierung bei 0 beginnt und nicht bei 1.

Python unterscheidet zudem stark zwischen ganzen Zahlen („integer“), Gleitkommazahlen („float“). Die Indizierung von Vektoren und Matrizen funktioniert ausschließlich mit ganzen Zahlen, es gilt in diesem Fall $1 \neq 1.0$. Es muss daher unbedingt darauf geachtet werden, dass man nur integer-Variablen verwendet, wenn man damit Vektor- oder Matrixeinträge aufrufen möchte.

Auch unterscheidet Python zwischen Zahlen und Matrizen, weist man in Matlab einer Variable eine einzige Zahl zu, so macht Matlab daraus eine 1×1 -Matrix mit der man ohne Probleme Matrixoperationen durchführen kann. In Python hingegen wäre diese Variable explizit keine Matrix, d.h. man könnte damit auch keine Matrixoperationen wie Transponieren, Matrixmultiplikation oder Eingabe in einen LGS-Löser.

Um in Python eine Matrix zu definieren, benötigt man den Befehl `numpy.array` aus der Programmbibliothek `numpy`. Ein Beispiel für einen Zeilenvektor wäre:

```
In [1]: import numpy as np
In [2]: np.array([1, 2, 3])
Out [2]: array([1, 2, 3])
```

Eine 2×2 -Matrix sieht z.B. so aus:

```
In [3]: np.array([[1, 2], [3, 4]])
Out [3]:
array([[1, 2],
       [3, 4]])
```

Python unterscheidet zwischen Zeilenvektor und Matrix, so kann man ersteren nicht transponieren bzw. erhält einfach denselben Zeilenvektor.

Es ist daher essentiell, dass alle Variablen von der richtigen Klasse sind. Zu beachten ist vor allem, dass man im eindimensionalen Fall 1×1 -Arrays verwendet, damit Methoden, die für mehrdimensionale Objekte geschrieben wurden, funktionieren.

Des Weiteren muss bei der Einführung von Arrays beachtet werden, dass folgendes Vorgehen

```
In [1]: a = np.array([1])
In [2]: b = a
```

den Array `b` als Spiegelbild von `a` behandelt, das heißt, alle Veränderungen der Einträge in einer der beiden Variablen werden automatisch auch in der anderen durchgeführt:

```
In [3]: print(b)
[1]

In [4]: a[0]=2
In [5]: print(b)
[2]
```

Dies passiert jedoch nur bei Veränderungen der Einträge einer der Variablen, belegt man eine der beiden komplett neu, löst dies die Verbindung wieder auf.

Dieses Verhalten gilt unter Anderem auch für

2.1.5 Dictionaries

Ein häufig verwendeter Datentyp von Python ist das sog. *Dictionary*, das dem Typ *struct* von Matlab entspricht. Ein Dictionary wird durch geschweifte Klammern definiert:

```
In [1]: b = { 'eins': 1, 'zwei': 2 }
```

Auf die Einträge wird mit eckigen Klammern zugegriffen:

```
In [1]: b[ 'eins' ]  
Out [1]: 1
```

Es empfiehlt sich den Output von Beispielfunktionen als Dictionary { 'f': f, 'grad': grad, 'Hess': Hess } zu wählen wobei f der Funktionswert, grad der Gradient und Hess die Hessematrix der Beispielfunktion in einem Punkt x ist. Ein Beispiel hierfür findet sich im Anhang.

Der Vorteil gegenüber Matlab-structs besteht darin, dass Python einen Befehl der Art *function(x)['f']* erlaubt, während Matlab für den analogen struct-Befehl *function(x).f* einen Fehler ausgibt. Für Matlab ist von vornherein nicht klar, dass der Output den Typ *struct* hat, man muss daher erst die Funktion ausführen und kann dann in einem separaten Befehl auf die struct-Einträge zugreifen. Python hingegen führt von links nach rechts aus, heißt an dem Punkt, an dem der Dictionary-Eintrag aufgerufen wird, ist schon klar, dass es sich um ein Dictionary handelt.

2.1.6 Schleifen

Schleifen in Python funktionieren grundsätzlich gleich wie in Matlab. Ein paar Unterschiede, die man beachten sollte sind:

- Eine *for*-Schleife in Python sieht bspw. so aus:

```
for i in I:  
    print ( i )
```

Die Sequenz *I* kann hierbei sowohl eine Liste, als auch ein Array, ein Dictionary, o.Ä. sein. Die Variable *i* ist somit nicht auf Zahlen beschränkt, sondern kann auch Wörter oder ganze Matrizen annehmen.

- Es können auch mehrere Bedingungen an *if*- und *while*-Schleifen gekoppelt werden, diese müssen mit *and/and not* bzw. *or/or not* voneinander getrennt werden.

Mit diesem Vorwissen kommen wir nun zur

2.2 Implementierung des L-BFGS-B-Verfahrens

Wir beginnen mit der Implementierung der Liniensuche und kommen dann zum eigentlichen Verfahren. Anschließend werden wir noch kurz auf ein Beispiel eines main-files eingehen. Die ausformulierten Codes finden sich jeweils im Anhang.

Der Einfachheit halber wählen wir $B_0 = I$, dadurch wird es möglich, den Algorithmus *bfgsrecb* so zu implementieren, dass er sich rekursiv selbst aufruft und wir uns den Algorithmus *bfgsrec* sparen können.

2.2.1 Liniensuche bzw. projizierter Armijo-Algorithmus

Die Liniensuche benötigt als Input wie im unrestringierten Fall die Kostenfunktion $fhandle$, den aktuellen Punkt x , eine Anfangsschrittweite λ_0 , eine maximale Anzahl an Iterationen $amax$ und Parameter α, β . Im restringierten Fall kommen nun noch Vektoren l und u hinzu (untere und obere Schranke). Als letzten Input lassen wir zusätzliche Argumente für die Zielfunktion zu (z.B. Parameter, die man im main-file festlegen will).

```
def linesearch(fhandle, x, Lambda0, alpha, beta, amax, l, u, *fargs):
```

Der Stern vor dem Variablennamen erlaubt es, dass jede beliebige Anzahl an zusätzlichen Input-Argumenten akzeptiert wird und weitergegeben werden kann.

Bevor wir in die *while*-Schleife einsteigen definieren wir einen *count*, der die Iterationen mit zählt, erklären λ_0 zur aktuellen Schrittweite und ermitteln aus Gründen der Übersichtlichkeit die Werte $xLambda = x(\lambda) = P(x - \lambda \nabla fhandle(x))$, den aktuellen Funktionswert $fcurrent = f(x(\lambda))$ und den zu unterbietenden Wert $fgoal = -\alpha \nabla f(x)^T (x - x(\lambda)) + f(x)$.

Als Projektion verwenden wir den Befehl *numpy.clip*, der das macht, was (1.3) beschreibt.

```
count = 0
Lambda = Lambda0
xLambda = np.clip(x - Lambda * fhandle(x, *fargs) ['grad'], a, b)
fcurrent = fhandle(xLambda, *fargs) ['f']
fgoal = -alpha * fhandle(x, *fargs) ['grad'].T.dot(x - xLambda) \
        + fhandle(x, *fargs) ['f']
```

Anmerkung: Wir haben *Lambda* bewusst groß geschrieben, da das kleingeschriebene *lambda* ein vorbelegter Befehl in Python ist, der analog zum @-Zeichen von Matlab funktioniert.

Der Backslash steht in Python für einen Zeilenumbruch und ist somit das Analogon zu '...' in Matlab.

Der Befehl '.T' ist die Matrix-Transponierung von Python, '.dot()' entspricht der Matrixmultiplikation.

In der *while* Schleife wird nun *Lambda* so oft verkleinert, bis die hinreichende Abstiegsbedingung (1.6) erfüllt oder die maximale Anzahl an Iterationen erreicht ist.

```
while fcurrent > fgoal and count < amax:

    Lambda = Lambda * beta
    xLambda = np.clip(x - Lambda * fhandle(x, *fargs) ['grad'], a, b)
    fcurrent = fhandle(xLambda, *fargs) ['f']
    fgoal = -alpha * fhandle(x, *fargs) ['grad'].T.dot(x - xLambda) \
            + fhandle(x, *fargs) ['f']
    count += 1
```

Anmerkung: Soll etwas zu einer Variable dazu addiert werden, kann man den Befehl '+=' verwenden, um die Variable nicht doppelt schreiben zu müssen.

Wir führen am Ende eine Variable *flag* ein, die gleich 1 ist, falls der Algorithmus aufgrund des Erreichens der maximalen Anzahl an Iterationen abgebrochen wurde, und gleich 0, falls der Algorithmus mit einer zulässigen Schrittweite terminiert ist.

```
if count == amax:
    flag = 1
if fcurrent <= fgoal:
    flag = 0
```

Der Output der Liniensuche ist ein Dictionary, das die letzte Schrittweite und die *flag*-Variable enthält:

```
return { 'Lambda': Lambda, 'flag': flag }
```

2.2.2 bfgsrecb

Durch die Wahl von $B_0 = I$ erübrigt sich die Implementierung von *bfgsrec*, wir können alles in dem Algorithmus *bfgsrecb* zusammenfassen, indem dieser sich rekursiv selbst aufruft.

Die Inputs sind die aktuelle Iteration ns (bzw. Anzahl der aktuell gespeicherten Vektoren), die Vektoren-Folgen $\{s_k\}$, $\{y_k\}$ jeweils als Matrix zusammengefasst, die aktuelle Abstiegsrichtung d und einen ganzzahligen Vektor *activeset*, der alle aktiven Indizes enthält.

```
def bfgsrecb(ns, sstore, ystore, d, activeset):
```

Nach (1.5) ist das Anwenden des Operators $P_{\mathcal{I}}$ äquivalent dazu, alle Einträge mit aktivem Index gleich 0 zu setzen.

```
d[activeset] = 0
```

Da wir $B_0 = I$ gewählt haben können wir im Fall $ns = 0$ einfach d zurückgeben.

```
if ns == 0:
    return d
```

Für $ns \neq 0$ setzen wir auch alle aktiven Indizes von s_{ns-1} und y_{ns-1} gleich 0, berechnen damit das α aus dem Algorithmus und passen anschließend die Richtung d an.

```
sstore[ns-1, :][activeset] = 0
ystore[ns-1, :][activeset] = 0
Alpha = sstore[ns-1, :].dot(d)
d = d - Alpha * ystore[ns-1, :]
```

Anmerkung: Wir vernachlässigen die Multiplikation mit dem Kehrruch von $y^T s$, da wir diese geschickt in den L-BFGS-B-Algorithmus einbauen werden.

Statt *bfgsrec* als neuen Algorithmus zu schreiben, können wir nun auch *bfgsrecb* rekursiv aufrufen, mit einem leeren Vektor als letzten Input-Parameter, sodass fortan keine Einträge mehr gleich 0 gesetzt werden.

```
newset = np.int_ ([])
d = bfgsrecb(ns-1, sstore, ystore, d, newset)
```

Anmerkung: Der Befehl *np.int_* erstellt einen integer-array, also einen ganzzahligen Vektor, nur dann kann man diesen wie oben als Indexmenge verwenden.

Nun ändern wir wieder den Vektor d , wie im Algorithmus beschrieben, setzen nochmals alle aktiven Einträge 0 und geben den resultierenden Vektor als Output zurück.

```
d = d + (Alpha - ystore[ns-1, :].dot(d)) * sstore[ns-1, :]
d[activeset] = 0
return d
```

2.2.3 L-BFGS-B-Verfahren

Inputs, die wir noch nicht eingeführt haben sind *tol*, ein Dictionary, das eine absolute und relative Toleranz für das Abbruchkriterium des Verfahrens beinhaltet, und *nmax*, die maximale Anzahl an Iterationen, die das Verfahren durchlaufen soll.

```
def l_bfgs_b(fhandle, x0, tol, nmax, Lambda0, alpha, beta, amax, l, u, *fargs):
```

Zu Beginn definieren wir wieder einen *count*, erklären x_0 zur aktuellen iterierten und starten mit x_0 eine Matrix *X*, die am Ende alle Iterierten enthalten soll.

```
    count = 0
    x = x0
    X = np.array(x)[np.newaxis].T
```

Beachte: Wie erwähnt unterscheidet Python zwischen Zeilenvektoren und Matrizen. Wir müssen dem Zeilenvektor x_0 daher mithilfe von *numpy.newaxis* zuerst Matrixstruktur verleihen, bevor wir damit eine Matrix einführen können.

Wie im Theorie-Teil angesprochen wollen wir die Anzahl der gespeicherten Vektoren für *bfgsrecb* begrenzen, dafür führen wir einen neuen Zähler *ns* und eine entsprechende Schranke *nsmax* ein.

```
    ns = 0
    nsmax = 5
```

Damit können wir nun auch die Matrizen, in denen wir die Vektoren-Folgen speichern wollen, vorbelegen.

```
    ndim = int(len(x))
    sstore = np.zeros((nsmax, ndim))
    ystore = sstore.copy()
```

Anmerkung: Der Befehl *.copy* kopiert den Array, das Problem des Spiegelbilds aus Kapitel 2.1.4 tritt daher nicht auf.

Wir berechnen aus Gründen der Übersichtlichkeit noch ein paar Dinge im Voraus, der Befehl *numpy.linalg.norm* berechnet dabei standardmäßig die euklidische Norm eines Vektors/einer Matrix, kann jedoch auch zur Berechnung anderer Matrix- und Vektornormen verwendet werden.

```
    gc = fhandle(x, *fargs)['grad']
    norm_pg0 = np.linalg.norm(x0-clip(x0-gc, l, u))
    norm_pg = norm_pg0
```

und kommen abschließend zur ϵ -aktiven Indexmenge.

```
    epsilon = min(np.min(u-l)/2, norm_pg)
    activeset = np.int_([ ])
    for i in range(len(l)):
        if u[i]-x[i] <= epsilon or x[i]-l[i] <= epsilon:
            activeset = np.hstack((activeset, int(i)))
    inactiveset = np.setdiff1d(range(ndim), activeset)
```

Anmerkung: Das Zusammenfügen von NumPy-Arrays funktioniert nicht wie in Matlab nur mit Klammern, sondern hat eigene Befehle *hstack* bzw. *vstack*, je nachdem ob man horizontal oder vertikal zusammenfügen will.

Nun beginnt der eigentliche Algorithmus, der dann abbrechen soll, wenn entweder die Abbruchbedingung (1.7) erfüllt oder die maximale Anzahl an Iterationen überschritten ist.

```
while norm_pg >= tol['abs']+tol['rel']*norm_pg0 and count < nmax:
```

Sind diese Bedingungen nicht erfüllt, wenden wir den Algorithmus *bfgsrecb* auf die Abstiegsrichtung (den negativen Gradienten) an und fügen, da dieser auf die inaktiven Einträge einschränkt, die aktiven Einträge anschließend wieder hinzu.

```
d = -gc
d = bfgsrecb(ns, sstore, ystore, d, activeset)
gc_active = gc.copy()
gc_active[inactiveset] = 0
d = -gc_active+d
```

Nun können wir unser Liniensuchverfahren anwenden.

```
Lambda = linesearch(fhandle, x, d, Lambda0, alpha, beta, amax, l, u, \
                    *fargs)['Lambda']
```

Mit dieser Schrittweite berechnen wir nun die nächste Iterierte und damit die zu speichernden Vektoren s und y .

```
xLambda = np.clip(x+Lambda*d, l, u)

y = fhandle(xLambda, *fhandleargs)['grad']-gc
s = xLambda-x
y[activeset] = 0
s[activeset] = 0
```

Wir speichern s und y jedoch nur, falls die Speichermatrizen nicht voll sind und sie die Bedingung für die positive Definitheit $y^T s > 0$ erfüllen. Ansonsten verwerfen wir sie und starten den Speicherprozess von Neuem.

```
yts = y.dot(s)
if yts <= 0:
    ns = 0
if ns == nsmax:
    print('ns_reached_maximum_size')
    ns = 0
elif yts > 0:
    ns += 1
    alpha0 = yts**0.5
    sstore[ns-1,:]=s/alpha0
    ystore[ns-1,:]=y/alpha0
```

Im Fall, dass wir die Vektoren in den Speicher aufnehmen, multiplizieren wir sie mit der Wurzel des Kehrrbruchs von $y^T s$ und sparen uns somit das wiederholte Ausrechnen des Kehrrbruchs in *bfgsrecb*.

Nun führen wir das Update der Variablen durch und fügen den neuen Punkt (nach Umwandlung zu Matrixstruktur) zur Matrix X hinzu

```
x = xLambda
gc = fhandle(x, *fhandleargs)['grad']
norm_pg = np.linalg.norm(x-np.clip(x-gc, l, u))
epsilon = min(np.min(u-l)/2, norm_pg)
```

```

    activeset = np.int_ ([])
    for i in range(len(l)):
        if u[i]-x[i] <= epsilon or x[i]-l[i] <= epsilon:
            activeset = np.hstack((activeset , i))
    inactiveset = np.setdiff1d(range(ndim), activeset)
    xnew = np.array(x)[np.newaxis].T
    X = np.hstack((X, xnew))
    count += 1

```

Der Output ist nun die Matrix X mit allen Iterierten.

```

return X

```

2.2.4 Beispiel für ein main-file

Will man zu Beginn alle Variablen zurücksetzen, kann man folgende Zeilen an den Anfang eines Skripts setzen:

```

from IPython import get_ipython
get_ipython().magic('reset -sf')

```

Nun müssen wir alle Funktionen und Methoden „zusammenimportieren“. Für die bekannte Rosenbrock-Funktion könnte dies bspw. so aussehen:

```

from costfunctions import rosenbrockfunktion
from methods import l-bfgs-b
from visualization import contourplot

```

Da wir im *main*-file nicht direkt auf *linesearch* zugreifen, müssen wir es auch nicht importieren. Wir benötigen aber noch die Programmbibliotheken, die wir im *main*-file verwenden wollen.

```

import time
import numpy as np
import matplotlib.pyplot as plt

```

Die Bibliothek *time* erlaubt es uns die Rechenzeit der Methoden zu ermitteln, die Bibliothek *matplotlib.pyplot* benötigen wir an dieser Stelle nur, um zu Beginn alle noch offenen Plots schließen zu können:

```

plt.close('all')

```

Nun legen wir die Parameter fest, ein Beispiel wäre:

```

fhandle = rosenbrockfunktion
tolerance = {'abs': 1e-2, 'rel': 1e-4}
Lambda0 = 1
alpha = 1e-2
beta = 0.5
amax = 30
nmax = 1e+3
x0 = np.array([1, -0.5])
l = np.array([-1, -1])
u = np.array([2, 2])

```

Mit diesen Variablen führen wir nun den Algorithmus aus.

```

start = time.time()
X = l-bfgs-b(fhandle, x0, tolerance, nmax, Lambda0, alpha, beta, \
            amax, l, u)
comptime = (time.time() - start)

```

Der Wert von *comptime* gibt die vergangene Rechenzeit in Sekunden an.

Lassen wir uns das Resultat anschließend als Contour-Plot anzeigen, liefert das Ausführen dieses main-files die folgende Grafik

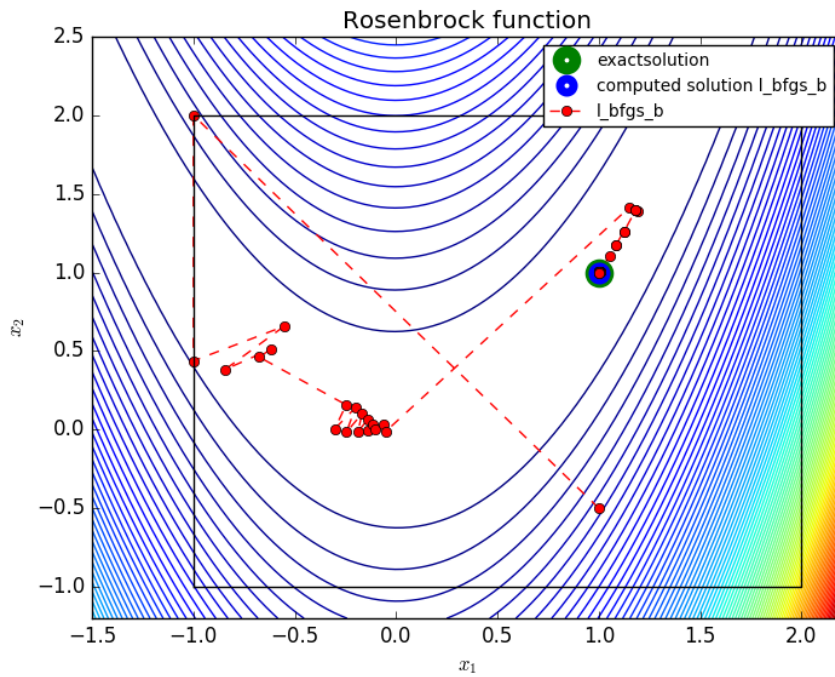


Abbildung 1 Contour-Plot Rosenbrock-Funktion

2.2.5 Vergleich zum projizierten Gradientenverfahren

Da unsere ursprüngliche Motivation, das L-BFGS-B-Verfahren zu betrachten, von der langsamen Konvergenzgeschwindigkeit des projizierten Gradientenverfahrens kam, wollen wir nun am Beispiel der Rosenbrock-Funktion zeigen, warum sich die Betrachtung gelohnt hat.

Die folgenden Graphen wurden mit den Parametern aus Kapitel 2.2.4 erstellt, mit projizierte Gradientennorm ist dabei $\|x - x(1)\|$ gemeint.

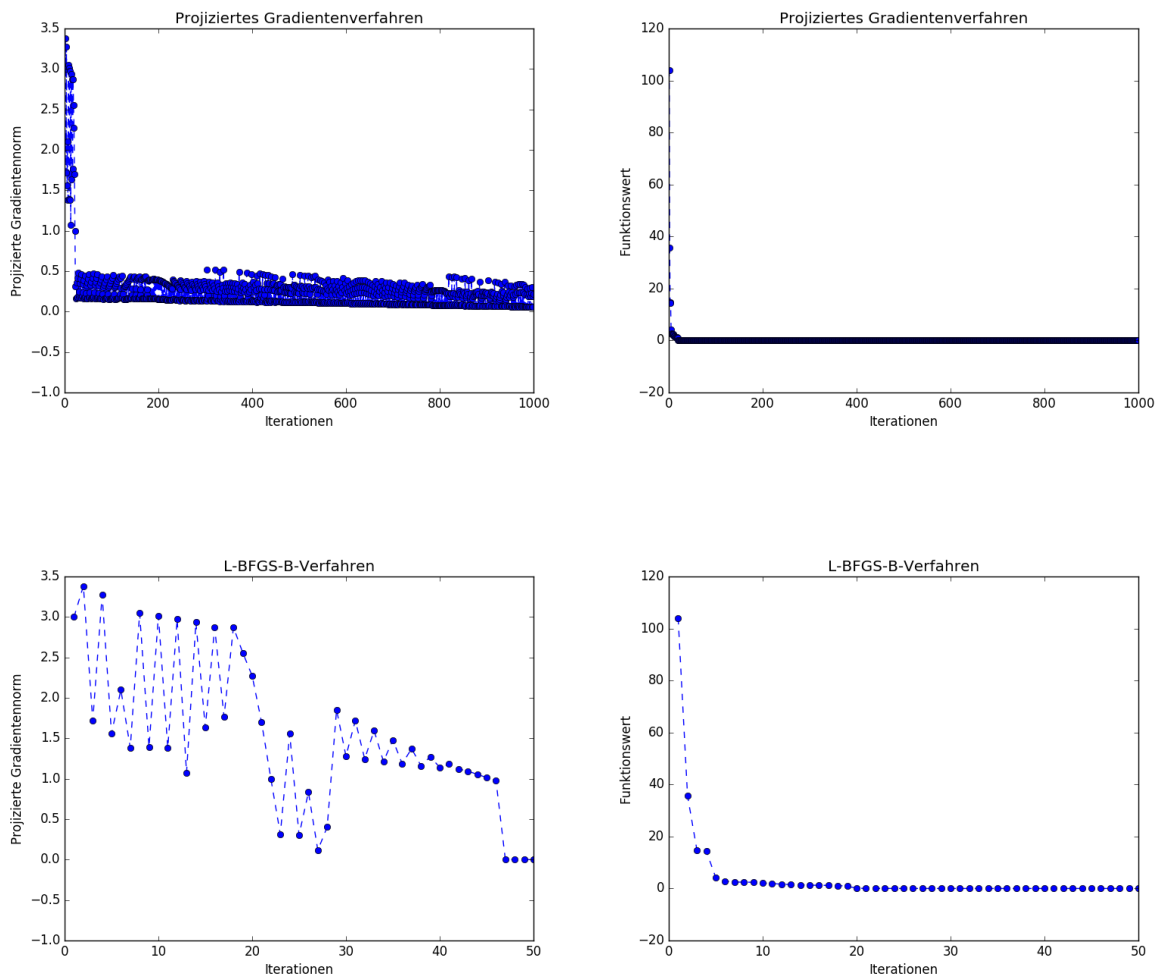


Abbildung 2 Vergleich proj. Gradienten- und L-BFGS-B-Verfahren anhand der Rosenbrock-Funktion

Wir sehen, dass das L-BFGS-B Verfahren bereits nach 50 Iterationen konvergiert, während das projizierte Gradientenverfahren auch nach 1000 noch nicht am Ziel ist (tatsächlich konvergiert das projizierte Gradientenverfahren mit diesen Parametern erst nach 6090 Iterationen).

Dies war zu erwarten, da, wie angesprochen, das L-BFGS-B-Verfahren im „Worst-Case-Szenario“, also wenn keine Vektoren gespeichert sind, gerade dem projizierten Gradientenverfahren entspricht und in allen anderen Fällen ein schnellerer Abstieg zu erwarten ist.

Durch Einsetzen der ersten Nebenbedingung können wir (3.1) umschreiben zu

$$\begin{aligned} \min \hat{J}(u) &= \frac{1}{2}(A^{-1}u - y_d)^T Q(A^{-1}u - y_d) + \frac{\sigma}{2}u^T R u \\ \text{u.d.N. } &u_a \leq u \leq u_b, \end{aligned} \quad (3.2)$$

was nur noch von u abhängt.

Der Gradient dieser Funktion ist gegeben durch

$$\nabla \hat{J}(u) = p + \nabla_u J(y, u) \quad (3.3)$$

mit

$$A p = A^T p = \nabla_y J(y, u) = Q(y - y_d) \quad (3.4)$$

$$\nabla_u J(y, u) = R u. \quad (3.5)$$

Zusammenfassend ergeben sich die folgenden (notwendigen und hinreichenden) Optimalitätsbedingungen für jede Lösung $(\bar{u}, \bar{y}, \bar{p})$ von (3.1) und (3.3):

$$\begin{aligned} A\bar{y} &= \bar{u} \\ A\bar{p} &= Q(\bar{y} - y_d) \\ \langle \bar{p} + \nabla_u J(\bar{y}, \bar{u}), u - \bar{u} \rangle &\geq 0 \quad \forall u \in U_{ad} := \{v \in \mathbb{R}^N : u_a \leq v \leq u_b\}, \end{aligned}$$

wobei die letzte Ungleichung auch *Variationsungleichung* genannt wird und unserer Definition eines stationären Punktes (1.2) entspricht.

3.2 Implementierung

Wir werden an dieser Stelle nicht erneut den Code Zeile für Zeile durchgehen, sondern nur ein paar Punkte ansprechen, die auch in anderen Situationen interessant sein könnten. Der vollständige Code von \hat{J} findet sich im Anhang.

Parameter von \hat{J}

Es empfiehlt sich immer, globale Variablen zu vermeiden, da diese speziell in Python zu einigen Problemen führen können. Daher nutzen wir bei der Implementierung von \hat{J} zum ersten Mal die zusätzlich zugelassenen Input-Parameter des L-BFGS-B-Algorithmus, um die Matrix A , den Zielzustand y_d und den Kostenfaktor σ weiterzugeben (alle drei werden zu Beginn im main-file festgelegt und bleiben danach unverändert). Dafür definieren wir ein neues Dictionary *fargs*, in dem die drei Parameter gespeichert werden und geben dieses dem Algorithmus als letzten Input-Parameter mit.

A als *sparse*-Matrix

Nachdem wir die FD-Matrix A in einem externen Programm haben aufbauen lassen, können wir sie mit

```
A = scipy.sparse.csr_matrix(A)
```

in *sparse*-Form umwandeln, dafür benötigen wir die Programmbibliothek *scipy*. Dies lohnt sich, da A wie oben gesehen dünn besetzt ist, das Umwandeln in *sparse*-Form also eine deutliche Speichereinsparung bedeutet. Um das Gleichungssystem (3.4) zu lösen, oder auch $A^{-1}u$ in (3.2) zu berechnen (wir rechnen natürlich nicht die Inverse von A aus, sondern lösen das Gleichungssystem $Ay = u$), verwenden wir dann den *sparse*-Löser von *scipy*. Das sieht dann z.B. so aus:

```
y = scipy.sparse.linalg.spsolve(A, u)
```

3.3 Resultate

Für den gesamten Abschnitt wählen wir einen Vektor, dessen Einträge gleich eins sind als Zielzustand, 10^{-4} als absolute und 10^{-2} als relative Toleranz für die Abbruchbedingung. Die Parameter der Liniensuche setzen wir wie in Kapitel 2.2.4 und die Anzahl an Gitterpunkten in jeder Dimension legen wir auf 40 fest.

Den Startwert u_0 , sowie die Schranken u_a und u_b wählen wir als Vektoren, deren Einträge alle gleich der zugewiesenen Zahl sind.

Beispiel 3.3.1

Wir betrachten die Input-Kombination

| u_0 | u_a | u_b | σ |
|-------|-----------|----------|----------|
| 100 | $-\infty$ | ∞ | 0.01 |

Damit konvergiert der L-BFGS-B-Algorithmus nach 4 Iterationen und liefert die folgenden Lösungen für y und u :

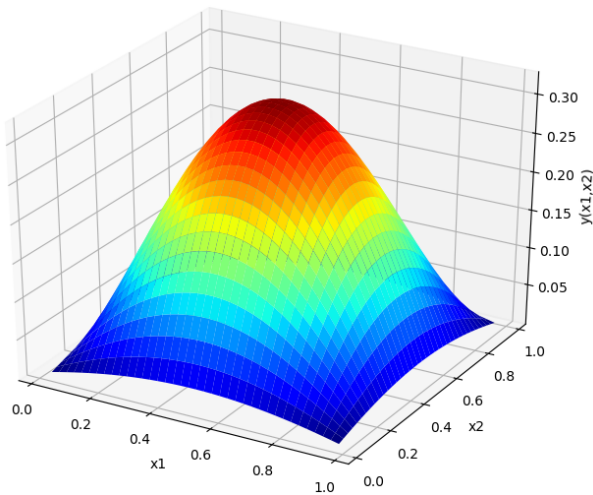


Abbildung 3 Zustandslösung y von Bsp. 3.3.1

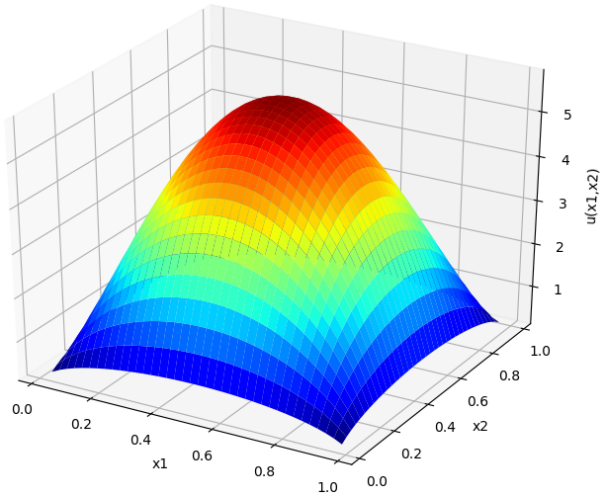


Abbildung 4 Kontrolllösung u von Bsp. 3.3.1

Wie zu erwarten war, strebt die Zustandslösung im Inneren gegen den Zielzustand, unter gleichzeitiger Beibehaltung der Nullrandwerte.

Die Kontrolllösung des obigen Beispiels wollen wir nutzen, um einen restringierten Fall zu rechnen. Wir erkennen aus dem Graph, dass die Werte der Kontrolllösung zwischen 0 und etwas über 5 liegen. Daher beschränken wir die Kontrollvariable (in jeder Komponente) nach unten durch 3 und nach oben durch 5:

Beispiel 3.3.2

| u_0 | u_a | u_b | σ |
|-------|-------|-------|----------|
| 4 | 3 | 5 | 0.01 |

Diese Kombination liefert (nach ebenfalls 2 Iterationen) die folgenden Lösungen:

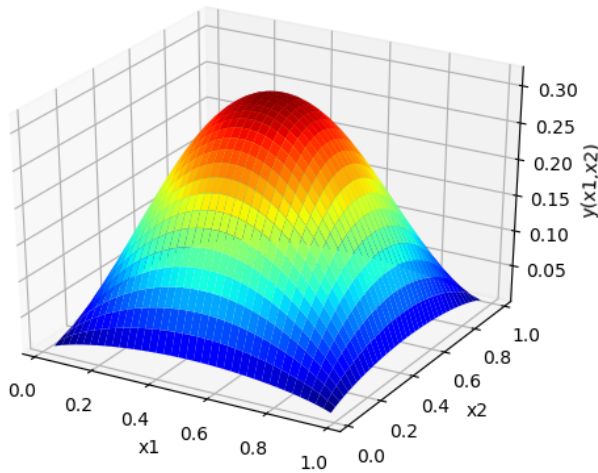


Abbildung 5 Zustandslösung y von Bsp. 3.3.2

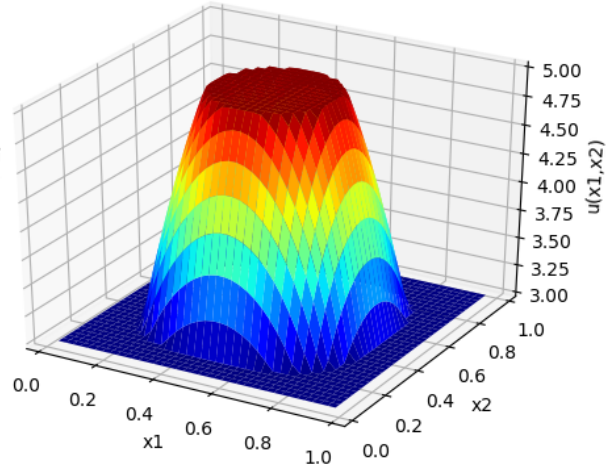


Abbildung 6 Kontrolllösung u von Bsp. 3.3.2

Wir sehen, dass sich die Zustandslösung kaum verändert hat, während die Kontrolllösung an den Schranken „abgeschnitten“ wurde und dafür im Inneren des Gebiets steiler ansteigt, um die verlorene „Spitze“ wieder gutzumachen.

Die Wahl eines geeigneten Startwerts ist im restringierten Fall hier übrigens von großer Bedeutung. Wählt man Werte außerhalb des zulässigen Gebiets, so terminiert der Algorithmus nach einer Iteration mit der (in der euklidischen Norm) am nächsten gelegenen Schranke.

Abschließend wollen wir noch den Einfluss des Kostenfaktors σ untersuchen. Zunächst wählen wir ihn größer als bisher:

Beispiel 3.3.3

| u_0 | u_a | u_b | σ |
|-------|-----------|----------|----------|
| 100 | $-\infty$ | ∞ | 0.1 |

Wir erhalten nach 4 Iterationen

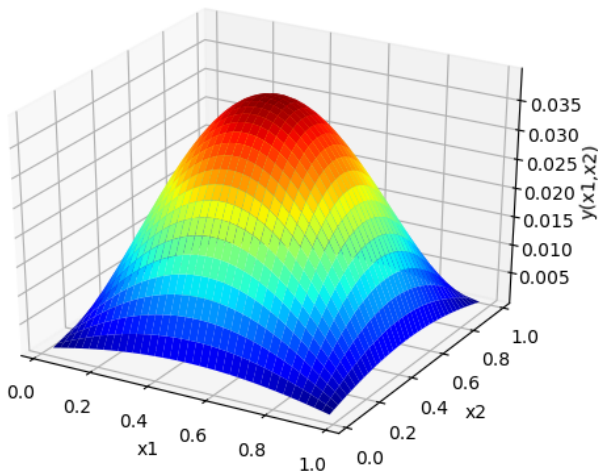


Abbildung 7 Zustandslösung y von Bsp. 3.3.3

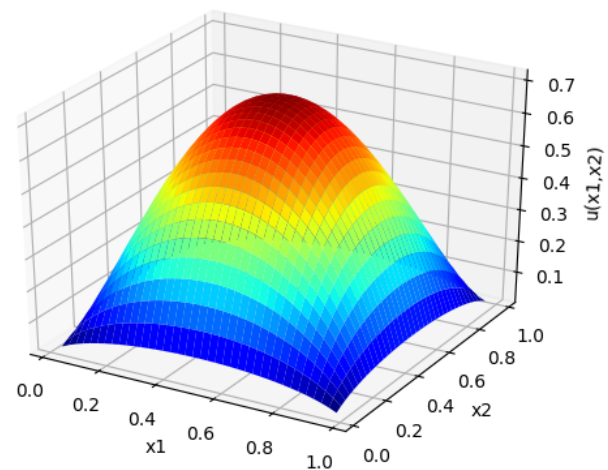


Abbildung 8 Kontrolllösung u von Bsp. 3.3.3

Beide Lösungen nehmen kleinere Werte an, da eine größere Steuerung nun „teurer“ ist und es daher nicht mehr so relevant ist, mit der Zustandslösung nahe am Zielzustand zu liegen.

Verkleinert man σ hingegen, so kehrt sich dieser Effekt für geeignete Startwerte um und wir erhalten größere Werte für beide Lösungen. Wählt man jedoch z.B. einen großen Startwerte wie oben, so kann das Folgende passieren:

Beispiel 3.3.4

Wir betrachten

| u_0 | u_a | u_b | σ |
|-------|-----------|----------|----------|
| 100 | $-\infty$ | ∞ | 0.0001 |

und erhalten nach 2 Iterationen

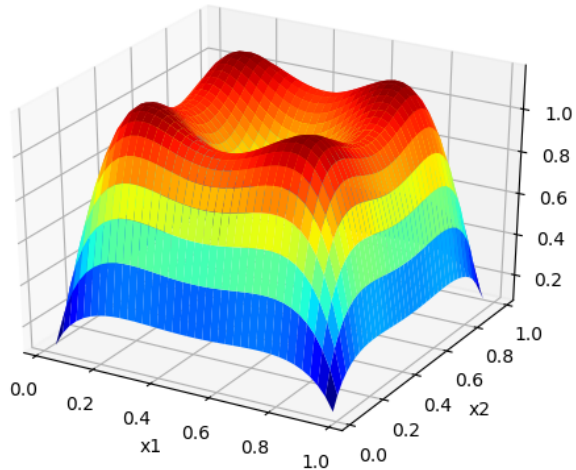


Abbildung 9 Zustandslösung y von Bsp. 3.3.4

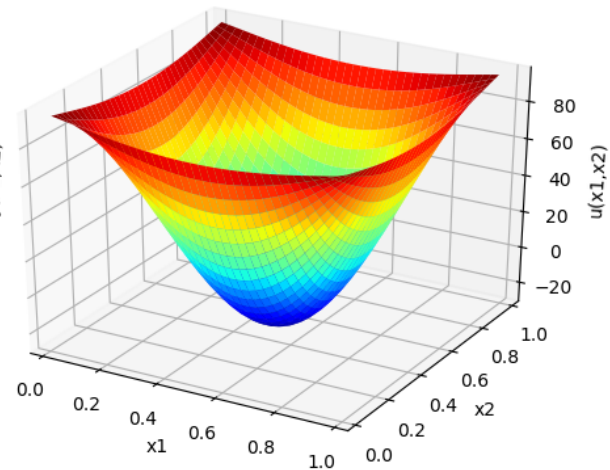


Abbildung 10 Kontrolllösung u von Bsp. 3.3.4

Wie wir sehen, ist hier die Konkavität der Lösungen verloren gegangen. Ein zu kleiner Kostenfaktor kann also bei schlecht gewählten Startwerten zu Problemen führen.

3.4 Vergleich zum projizierten Gradientenverfahren

Wir betrachten die Situation aus Beispiel 3.3.2. Das projizierte Gradientenverfahren terminiert mit diesen Inputs selbst nach 1000 Iterationen nicht und bricht mit dem folgenden Zwischenstand ab:

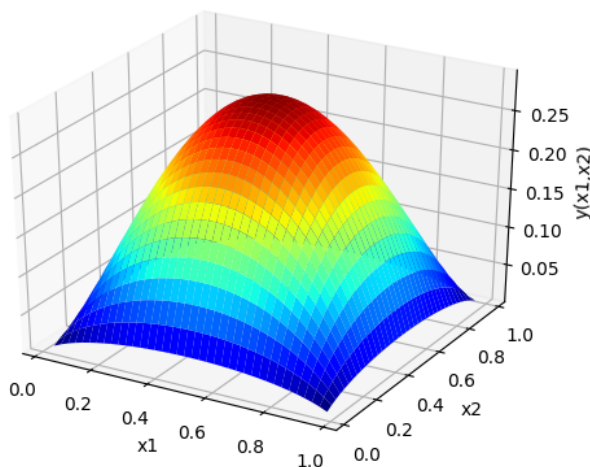


Abbildung 11 Zustand y

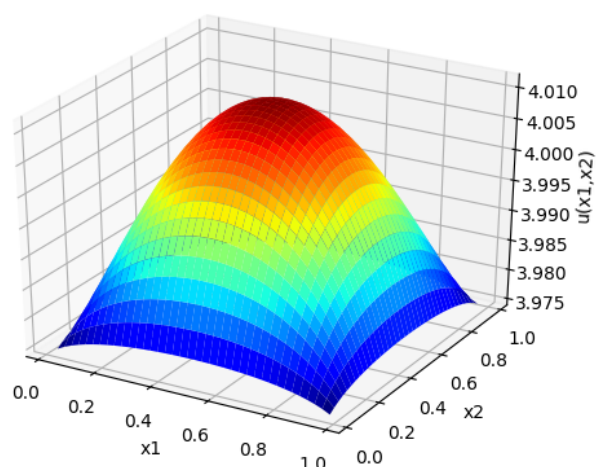


Abbildung 12 Kontrolle u

Es scheint, als hätte sich das Verfahren kaum vom Startwert (alle Komponenten gleich 4) entfernt. Das bestätigen auch die folgenden Graphen.

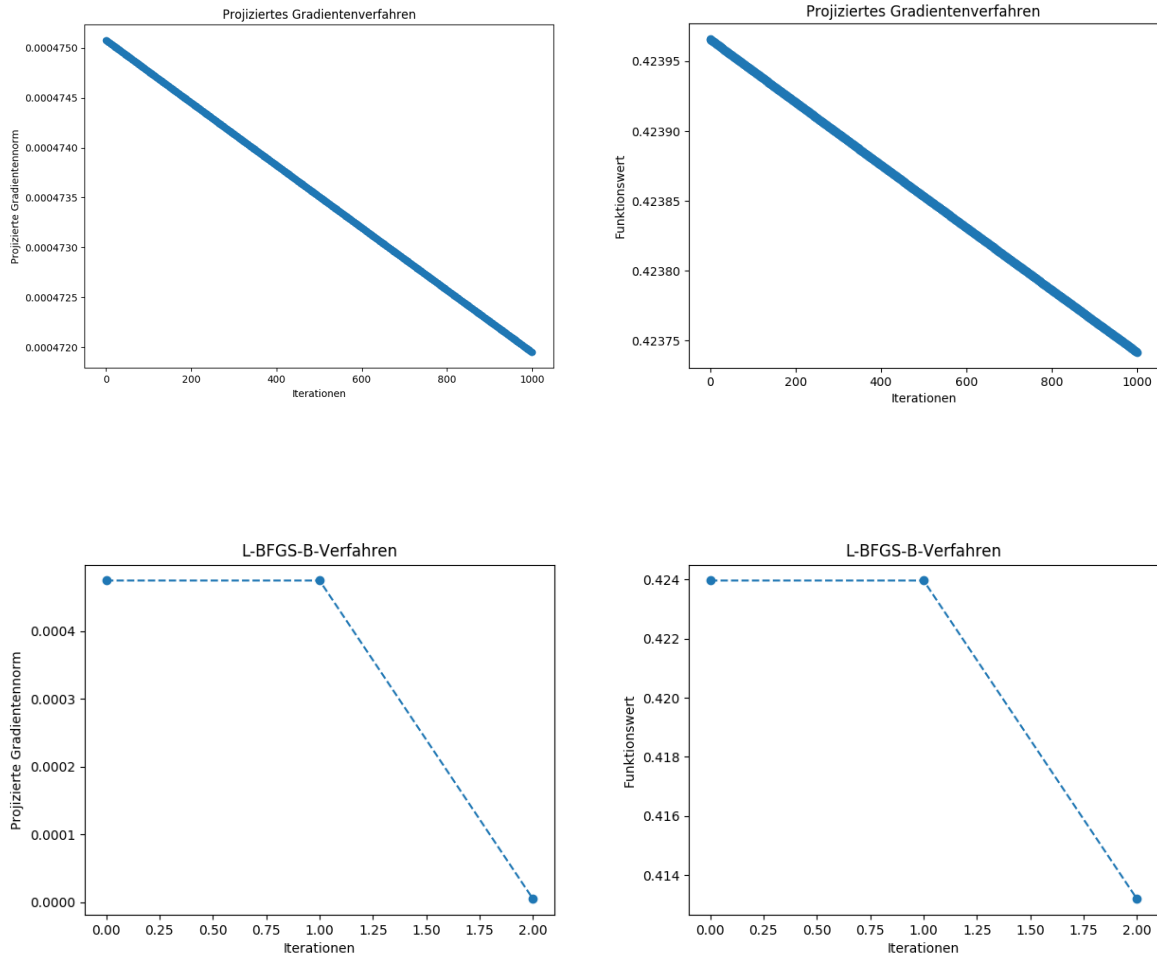


Abbildung 13 Vergleich proj. Gradienten- und L-BFGS-B-Verfahren anhand des Beispiels aus der optimalen Steuerung

Wir sehen, dass das projizierte Gradientenverfahren nur sehr kleine Schritte macht und daher selbst nach 1000 Iterationen noch keine wesentlichen Fortschritte aufweisen kann. Das L-BFGS-B-Verfahren hingegen macht nur einen solchen kleinen Schritt und läuft im nächsten direkt in eine Lösung hinein.

Fazit

Abschließend können wir festhalten, dass uns das L-BFGS-B-Verfahren die schnellere Konvergenzgeschwindigkeit geliefert hat, die wir erwartet haben. Selbst wenige gespeicherte Vektoren s_k und y_k haben bereits zu einer deutlichen Reduktion der benötigten Iterationen geführt (auch beim Beispiel der Rosenbrock-Funktion in Abschnitt 2.2.5 wurde die maximal zugelassene Anzahl an gespeicherten Vektoren (5) nie erreicht).

Bezogen auf die Implementierung haben wir gesehen, dass sich viele Grundlagen der mathematischen Programmierung von Matlab auf Python übertragen lassen. Wir haben aber auch ein paar Vorteile von Python ausgemacht, wie die Import-Funktion oder das einfachere Handling von *Dictionaries* im Vergleich zu Matlab *structs*. Dass man zu Python als Open-Source-Software mit einem weitaus breiteren Anwendungsgebiet auch mehr Tipps und Hilfen im Internet findet ist ein weiterer Punkt, der nicht unerwähnt bleiben soll.

Der Blick auf Python als Alternative oder Ergänzung zu Matlab kann sich also lohnen und ist gerade für Matlab Nutzer mit keiner großen Umstellung verbunden.

Literaturverzeichnis

- [Buc17] Simon Buchwald *Projiziertes Gradientenverfahren in Python*. Seminararbeit, 2017, abrufbar unter <https://drive.google.com/open?id=0B2c2-gJNl996RjRsWkZ1cEhTU1E>, letzter Zugriff am 02.10.2017
- [GK02] Carl Geiger, Christian Kanzow. *Theorie und Numerik restringierter Optimierungsaufgaben*. Springer-Verlag, 2002
- [Kel99] C. T. Kelley. *Iterative Methods for Optimization*. SIAM Frontiers in Applied Mathematics, Philadelphia, 1999
- [NW99] Jorge Nocedal, Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, 1999
- [Tro10] Fredi Tröltzsch. *Optimal Control of Partial Differential Equations: Theory, Methods and Applications*. American Mathematical Society, Graduate Studies in Mathematics, Vol. 112, 2010
- [Vol16] Stefan Volkwein. *Optimierung*. Vorlesungsskript, 2016
- [url01] <https://math.stackexchange.com/questions/1841949/quasi-newton-methods-sr1-and-bfgs-inverse-update>, letzter Zugriff am 02.10.2017
- Seite zum Download des *Anaconda*-Pakets, das Python und die IDE *Spyder* enthält:
- [url02] <https://www.anaconda.com/download/>, letzter Zugriff am 02.10.2017

Anhang

Ausformulierte Python-Codes

```
#####  
#                               Beispiel einer Kostenfunktion  
#####  
  
def rosenbrockfunction(x,*args):  
    """Rosenbrock function:  
  
         $f(x)=100*(x_2-x_1^2)^2+(1-x_1)^2$ ,  $x=(x_1,x_2)^T$  in  $\mathbb{R}^2$   
  
    Parameters  
    -----  
    x  
        A ndarray containing the current point.  
  
    Returns  
    -----  
    Dict containing the following fields  
  
        :f:      function value at u  
        :grad:   gradient at x  
    """  
    f = 100*(x[1]-x[0]**2)**2 + (1-x[0])**2  
    grad = np.array([-400*x[0]*(x[1]-x[0]**2)-2*(1-x[0]),  
                    200*(x[1]-x[0]**2)])  
    return {'f': f, 'grad': grad}
```

```

#=====
#                               Beispiel eines main-files
#=====

#clear variables from memory
from IPython import get_ipython
get_ipython().magic('reset -sf')

#import function and methods
from costfunctions import rosenbrockfunction
from projectionmethods import gradproj, l_bfgs_b

#import python packages
import numpy as np
import time

plt.close('all')

#set parameters:
#-----

fhandle = rosenbrockfunction

#termination-condition tolerance:
tolerance = {'abs': 1e-4, 'rel': 1e-4}

#maximum number of iterations:
nmax = 1000

#parameters for armijo:
Lambda0 = 1
alpha = 1e-4
beta = 0.5
amax = 30

#starting point:
x0 = np.array([1,-0.5])

#lower boundaries:
a = np.array([-1,-1])
#upper boundaries:
b = np.array([2,2])

#execute algorithm:
#-----

start = time.time()
X = l_bfgs_b(fhandle, x0, tolerance, nmax, Lambda0, alpha, beta, amax, a, b)
comptime1 = (time.time() - start)

```

```

#-----
#                               Liniensuche, bfgsrecb und L-BFGS-B-Verfahren
#-----
def linesearch(fhandle, x, d, Lambda0, alpha, beta, amax, l, u, *fhandleargs):
    """Line search with the sufficient decrease condition (1.6)"""

    #set initial values:
    count = 0
    Lambda = Lambda0
    xLambda = np.clip(x+Lambda*d, l, u)
    fcurrent = fhandle(xLambda,*fhandleargs)['f']
    fgoal = fhandle(x,*fhandleargs)['f']\
            -alpha*fhandle(x,*fhandleargs)['grad'].T.dot(x-xLambda)

    while fcurrent > fgoal and count < amax:

        #update stepsize, x and count:
        Lambda = Lambda*beta
        xLambda = np.clip(x+Lambda*d, l, u)
        fcurrent = fhandle(xLambda,*fhandleargs)['f']
        fgoal = fhandle(x,*fhandleargs)['f']\
                -alpha*fhandle(x,*fhandleargs)['grad'].T.dot(x-xLambda)
        count += 1

    #set flag to cause of termination:
    if count == amax:
        flag = 1
    if fcurrent <= fgoal:
        flag = 0

    return {'Lambda': Lambda, 'flag': flag}

#-----
def bfgsrecb(nt,sstore,ystore,d,activeset):
    """Recursively compute action of an approximated Hessian on a vector using
    stored information of the history of the iteration"""

    d[activeset] = 0
    if nt == 0:
        return d

    sstore[nt-1,:][activeset] = 0
    ystore[nt-1,:][activeset] = 0

    Alpha = sstore[nt-1,:].dot(d)
    d = d-Alpha*ystore[nt-1,:]

    newset = np.int_([ ])
    d = bfgsrecb(nt-1,sstore,ystore,d,newset)

    d = d+(Alpha-ystore[nt-1,:].dot(d))*sstore[nt-1,:]
    d[activeset] = 0
    return d

#-----
def l_bfgs_b(fhandle, x0, tolerance, nmax, Lambda0, alpha, beta, amax, l, u,
            *fhandleargs):
    #set initial parameters
    count = 0
    ns = 0
    nsmax = 5
    x = x0
    gc = fhandle(x,*fhandleargs)['grad']

```

```

norm_pg0 = np.linalg.norm(x0-np.clip(x0-gc, l, u))
norm_pg = norm_pg0
X = np.array(x0)[np.newaxis].T
ndim = int(len(x))
sstore = np.zeros((nsmax,ndim))
ystore = sstore.copy()

#build arrays with active and inactive indices
epsilon = min(np.min(u-l)/2,norm_pg)
activeset = np.int ([])
for i in range(len(l)):
    if u[i]-x[i] <= epsilon or x[i]-l[i] <= epsilon:
        activeset = np.hstack((activeset,int(i)))
inactiveset = np.setdiff1d(range(ndim),activeset)

while count < nmax and norm_pg >= tolerance['abs']\
    +tolerance['rel']*norm_pg0:

    #descent direction (recursive)
    d = -gc
    d = bfqsrecb(ns,sstore,ystore,d,activeset)
    gc_active = gc.copy()
    gc_active[inactiveset] = 0
    d = -gc_active+d

    #line-search
    Lambda = linesearch(fhandle,x,d,Lambda0,alpha,beta,amax,l,u,\
        *fhandleargs)['Lambda']
    xLambda = np.clip(x+Lambda*d, l, u)

    v = fhandle(xLambda,*fhandleargs)['grad']-gc
    s = xLambda-x
    x = xLambda
    gc = fhandle(x,*fhandleargs)['grad']
    norm_pg = np.linalg.norm(x-np.clip(x-gc, l, u))

    #build the new arrays with active and inactive indices
    epsilon = min(np.min(u-l)/2,norm_pg)
    activeset = np.int_ ([])
    for i in range(len(l)):
        if u[i]-x[i] <= epsilon or x[i]-l[i] <= epsilon:
            activeset = np.hstack((activeset,i))
    inactiveset = np.setdiff1d(range(ndim),activeset)

    y[activeset] = 0
    s[activeset] = 0

    #reset storage, if y.dot(s) is not positive or reserved storage is full
    yts = y.dot(s)
    if yts <= 0:
        ns = 0
    if ns == nsmax:
        print('ns reached maximum size')
        ns = 0
    elif yts > 0:
        ns += 1
        alpha0 = yts**0.5
        sstore[ns-1,:]=s/alpha0
        ystore[ns-1,:]=y/alpha0

    count += 1
    xnew = np.array(x)[np.newaxis].T
    X = np.hstack((X,xnew))
return X

```

```

#=====
#      Kostenfunktional Jhat des Beispiels aus der optimalen Steuerung
#=====

def Jhat(u,args):
    """Costfunction for the optimal control example, only depending on the
    control variable u:

        Jhat(u) = 1/2*(A^(-1)u-yd)^T*Q*(A^(-1)u-yd) + sigma/2*u^T*R*u

    Parameters
    -----
    u
        A ndarray containing the current point.
    args
        Dict containing the following fields

            :A:      FE-matrix A
            :yd:     desired state vector
            :sigma:  control factor sigma (optional)

    Returns
    -----
    Jout
        Dict containing the following fields

            :f:      function value at u
            :grad:   gradient at x
    """
    #get A and yd from args:
    A = args['A']
    yd = args['yd']

    #get sigma from args or set it to some default value:
    if 'sigma' in args:
        sigma = args['sigma']
    else:
        sigma = 0.01

    #build matrix Q:
    length_u = len(u)
    Nx = int(length_u**0.5)+1
    h=1/Nx
    Q = h**2*np.eye(length_u)

    #calculate state-variable from current control variable:
    y = sparse.linalg.spsolve(A,u)

    #calculate costfunction value and gradient value at current point:
    f = 1/2*(y-yd).dot(Q.dot(y-yd))+sigma/2*u.dot(Q.dot(u))
    gradJ_y = Q.dot(y-yd)
    gradJ_u = sigma*Q.dot(u)
    p = sparse.linalg.spsolve(A.T,gradJ_y)
    grad = p+gradJ_u

    Jout = {'f': f, 'grad': grad}
    return Jout

```