OptimizerToolBox

Matthias Wacker
matthias.wacker@siemens.com

8. Oktober 2007

Zusammenfassung

Gegenstand der Dokumentation ist die "OptimizerToolBox" - eine Sammlung nichtlinearer Optimierungsverfahren. Es werden Grundlegende Konzepte erläutert, Stärken und Schwächen bestimmter Verfahren hervorgehoben und Beispiele zur Verwendung der Toolbox gegeben.

Inhaltsverzeichnis

1 Einleitung

Die "OptimizerToolBox"ist ein Sammlung von numerischen Verfahren der nichtlinearen Optimierung programmiert in C++. Sie soll dem Anwender als Standardwerkzeug für die Prototyp- und Produktentwicklung unterstützend zur Seite stehen und durch ein vorgegebenes Interface den Austausch eines Optimierungsverfahrens erleichtern.

2 Theoretischer Hintergrund

Grundsätzlich wird in der Optimierung zwischen restringierter und unrestringierter Optimierung, also Optimierung mit bzw. ohne Nebenbedingungen unterschieden.

2.1 Unrestringierte Optimierung

Zu lösen ist ein mathematisches Problem der Form:

$$\hat{x} = \arg\min_x f(x) \qquad , \ x \in \mathbb{R}^n$$

In Worten: Man sucht den Punkt x für den die Funktion $f(x)$ minimal wird.

Bemerkung: In diesem Dokument wird auf die Maximierung nicht eingegangen, da man dasselbe Resultat durch Minimierung der negierten Funktion erhält:

$$\arg\max_x f(x) = \hat{x} = \arg\min_x -f(x)$$

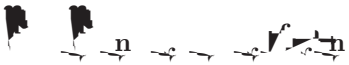
In der Regel zielen die Verfahren darauf ab, Punkte zu finden, in denen die notwendige und hinreichende Optimalitätsbedingung nullter Ordnung

$$f(\hat{x}) < f(x) \qquad , \ \forall x \in U(\hat{x}), \ U \text{ Umgebung um } \hat{x}$$

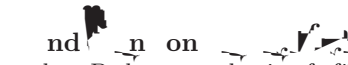
oder die notwendige Optimalitätsbedingung erster Ordnung

$$\nabla f(\hat{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1} f(\hat{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\hat{x}) \end{pmatrix} = 0_n$$

erfüllt ist.



Generell muss man vorwegnehmen, dass es nicht "den Optimierer schlechthin"gibt. Es gibt bei jedem Optimierer Problemstellungen, bei denen er versagt. Es gilt also zum Problem den passenden Optimierer zu wählen, und diesen entsprechend mit Parametern anzupassen.



Eindimensionale Verfahren haben besondere Bedeutung, da sie oft für die Schrittweitenbestimmung bei mehrdimensionalen Verfahren benutzt werden.

Das Verfahren der sukzessiven Dreiteilung nach dem golden Schnitt benötigt eine obere und eine untere Grenze. davon ausgehend, wird iterativ die Strecke im Goldenen Schnitt geteilt und damit die Suchstrecke entsprechend eingeschränkt . Das Verfahren wird als relativ robust angesehen.

B n M Das Verfahren nach Brent ist eine der bekanntesten Methoden der eindimensionalen Suche. Es verwendet (beispielsweise) das Verahren der sukzessiven Dreiteilung nach dem Goldenen Schnitt für den Fall, dass eine schnellere Methode (wie beispielsweise das Verfahren der qubischen Angleichung) nicht funktioniert oder anwendbar ist. Der Trick ist dabei der effiziente Wechsel zwischen den Verfahren, falls dieser nötig ist.

B N (BN Der BNO (deutsch: "Bester Nachbar") braucht einen Startpunkt und eine Schrittweite. Dann geht er iterativ wie folgt vor:

1. Ausgehend vom aktuellen Punkt werden pro Dimension die Nachbarn in der Entfernung der Schrittweite bestimmt und die Zielfunktion entsprechend ausgewertet.
2. nachdem die Nachbarschaft komplett bestimmt ist, wählt man den besten Nachbarn, sofern der Wert dort besser ist als am aktuellen Punkt, als nächsten Iterationspunkt und geht zu 1.
3. gibt es keinen besseren Nachbarn, so beendet man die Iteration, da man im Sinne des BNO am optimalen Punkt angelangt ist

Vorteile des BNO sind:

- sehr gut Nachvollziehbar
- es werden keine Ableitungen benötigt

Nachteile des BNO sind:

- bleibt sehr leicht in lokalen Optima hängen
- entscheidet sich für eine Dimension, iteriert also nicht in mehrere Richtungen gleichzeitig (Verhalten)

Der BNO liefert sehr gute Ergebnisse, wenn man eine montone Zielfunktion vorliegen hat und nur "gerade" Ergebnisse erwartet (z.B. Pixelkoordinaten).

d n c n (Der GDO (deutsch: Gradienten Abstieg) braucht einen Startpunkt, optional können verschiedene Parameter gesetzt werden. Der Algorithmus geht iterativ wie folgt vor:

1. Ausgehend vom aktuellen Punkt wird der Gradient der Funktion an diesem Punkt bestimmt. (Der Gradient zeigt in die Richtung des steilsten Anstiegs.)
2. optional wird eine Schrittweite bestimmt
3. der nächste Punkt ergibt sich aus dem aktuellen indem man genau entgegen der Richtung des Gradienten mit der bestimmten Schrittweite fortschreitet. Anschliessend kehrt man zurück zu Schritt 1.
4. Ein sinnvolles Abbruchkriterium ist eine Schranke für die Norm des Gradienten. Fällt die Norm unter die Schranke, bricht man die Iteration ab.

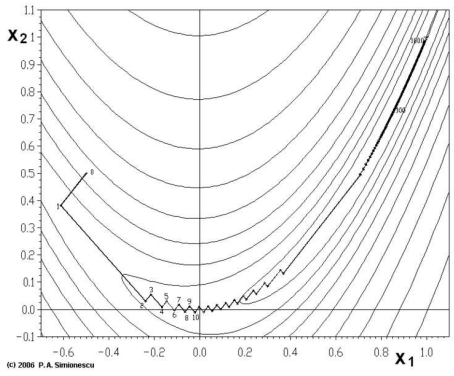


Abbildung 1: Iterationsweg eines Gradientenabstiegs bei der Rosenbrockfunktion. (Quelle: www.wikipedia.org)

Vorteile des GDO sind:

- intuitiver Ansatz (Iteration in Richtung des steilsten Abstiegs)
- relativ schnell durch Ausnutzung der Gradienteninformation
- ist in der Iterationsrichtung flexibler als der BNO

Nachteile des GDO sind:

- bei verrauschtem Gradienten (oft bei datengetriebenen Zielfunktionen) wird der GDO schnell fehlgeleitet
- bei Zielfunktionen die ein schmales Tal im Parameterraum bilden ("banana function"(Rosenbrocks' function)) oszilliert der GDO, ohne dem Optimum näher zu kommen

Der GDO ist sehr effektiv bei realitv einfachen Zielfunktionen, kann jedoch je nach Zielfunktion und Startwert leicht fehlschlagen oder nicht akzeptable Konvergenzgeschwindigkeit liefern ("zickzack"-Iterationen in schmalen Tälern).

d **c** (Der GSO (deutsch: Gitter Suche) braucht eine Beschränkung des Suchraumes. Über diesen Suchraum legt der GSO ein gleichmäßiges Gitter. Nach eine vollständigen Auswertung aller Gitterpunkte wählt er den besten Punkt aus. Vorteile des GSO sind:

- liefert auf dem Gitter die global optimale Lösung

Nachteile des GSO sind:

- Anzahl der Auswertung ist sehr hoch!(wächst exponentiell mit der Dimension)
- die Beschränkung muss korrekt gewählt werden. TradeOff: Rechenaufwand vs Globale Lösung

Der GSO hat seine Stärke

- bei Funktionen die einen sehr kleinen Konvergenzbereich haben, der zwar auf einen gewissen Bereich eingegrenzt werden kann, aber innerhalb dieses Bereiches unvorhersehbar auftritt
- wenn mit 100%iger Sicherheit das Optimum bis auf die "Gitterlochbreite" angenähert werden muss

o n (Der DSO ist ein robustes Iteratives Verfahren, das keine Ableitungsinformation benötigt. Es besitzt als aktuellen Zustand nicht nur einen Punkt, sondern Dimension +1 Punkte. Diese Stuktur wird Simplex genannt. Bei einem 2dimensionalen Suchraum ist der Simplex also ein Dreieck. Das Verfahren verändert das Dreieck iterativ durch folgende Operationen:

- Speigeln des schlechtesten Punktes an der Geraden durch die beiden anderen Punkte
- Heranziehen des schlechtesten Punktes an diese Gerade
- Kontraktion des Dreieck um den besten Punkt.

D.h. das Dreick klappt sich um, steckt und staucht sich, bis z.B. eines der Folgenden Kriterien erfüllt ist

- Punkte voneinander weniger entfernt als eine Schranke
- Funktionswerte der einzelnen Punkte weniger als Schranke von einander entfernt
- etc

Vorteile des DSO sind:

- relativ robust gegenüber verrauschtem Funktionsverlauf
- braucht keine Ableitungsinformation
- kann lokale Optima "überläufen"
- kann mit relativ schmalen Tälern im Parameterraum umgehen.

Nachteile des DSO sind:

- Konvergiert nicht so schnell wie ein Verfahren das Ableitungsinformation verwenden
- vergleichsweise unbequeme Initialisierung (Dimension+1 Punkte anstatt einem)

Der DSO hat seine Stärken da, wo ein Gradientenabstieg aufgrund verrauschter Ableitungsinformation oder schmaler Täler im Suchraum versagt.

o n o c Kombinatorische Optimierer sind iterative Zufallssuchverfahren, die folgendermaßen vorgehen: Ausgehend von einem Startpunkt und einer Steuerungssequenz (fallende folge von Zahlen z.b. $T_k = \frac{1}{k},\ k \in \mathbb{N}$) wird

1. aus dem aktuellen Punkt ein neuer Punkt generiert (meistens von der Steuerungssequenz gesteuert),

2. dieser neue Punkt wird gemäß einer Entscheidungsregel (die die Steuerungssequenz einbeziehen kann) abgelehnt oder akzeptiert und als neuer aktueller Punkt angesehen
3. das Verfahren bricht ab, falls z.B. die zugestandene Zeit verstrichen ist, andernfalls wird mit dem 1. Schritt und dem nächsten Element der Kontrollsequenz fortgefahren.

Durch die Wahl einer Entscheidungsregel, spalten sich die Verfahren der kombinatorischen Optimierung in folgende Unterverfahren auf:

- Simulated Annealing (deutsch: Simuliertes Ausfrieren)
- Threshold Acceptance (deutsch: Schwellwertakzeptanz)
- Great Deluge (deutsch: Sintflutalgorithmus)
- Stochastic Relaxation (deutsch: Stochastische Relaxation)

Im Folgenden bezeichnet T_k der Wert der Steuerungssequenz in der aktuellen Iteration, f_n den Funktionswert am neuen Punkt und f_c den Funktinswert am aktuellen Punkt. Damit lassen sich die einzelnen Verfahren beschreiben. Ein neuer Punkt wird akzeptiert, bei:

d Ann n falls ...

$$\begin{cases} \triangle f = f_n - f_c < 0 \text{ oder} \\ \triangle f \geq 0 \ \wedge \ q \leq \exp\left(-\frac{\triangle f}{T_k}\right) \end{cases}$$

wobei $q \in [0, 1]$ eine gleichverteilte Zufallszahl ist.

o d Acc nc falls ...

$$\triangle f \leq T_k$$

falls ...

$$f_n \leq T_k$$

oc c on falls ...

$$f_n \leq f_c$$

Vorteile kombinatorischer Verfahren sind:

- es lassen sich sehr komplexe Problemstellungen lösen
- man braucht keine Ableitungsinformation
- man kann lokale Optima "'überspringen"'
- je nach Entscheidungsregel kann man aus lokalen Optima "'entfliehen"'

Nachteile kominatorischer Verfahren sind:

- Ergebnisse sind in der Regel nicht 100%ig reproduzierbar

- keinerlei Information, wie "'nahe"' man einem Optimum ist (Abbruch z.B. über verbleibende Zeit)
- Verfahren müssen i.d.R. dem Problem genau angepasst werden, um gute Resultate zu erzielen.

Kombinatorische Verfahren sind dann zu empfehlen, wenn nicht nicht die beste, sondern eine hinreichend "'gute"' Lösung gesucht wird und das Problem von der Komplexität und Charakteristik her analytische Verfahren ungünstig macht. Das bekannteste kombinatorische Verfahren ist das Simulierte Ausfrieren.

(P) Adaptive Random Search

(P)ADRS ist ein stochastisches Optimierungsverfahren das sich stets eine Menge von Punkten als Parametermenge hält. Iterativ wird wie folgt vorgegangen:

1. ausgehend von jedem Punkt wird ein neuer Punkt generiert.
2. ist der neue Punkt besser als der bisherige Punkt, so ersetzt er diesen in der Parametermenge
3. Schlägt die Suche nach einem besseren Punkt oft genug fehl, so wird in einer kleineren Umgebung gesucht oder die Optimierung für diesen einen Punkt wird gestoppt. Ist der resultierende Punkt schlechter als bereits bekannte Punkte, so wird er verworfen, andernfalls wird er gespeichert. Der freiwerdende Platz in der Parametermenge wird durch ziehen aus der bestehenden Parametermenge neu besetzt, wobei bessere Punkte eine höhere Wahrscheinlichkeit erhalten.
4. gestoppt wird, wenn z.B. eines Zeit- , Iterationslimit erreicht wurde, oder die Parametermenge hinreichend stark kontrahiert ist

Der Vorteil gegenüber vielen anderen stochastischen Verfahren liegt in der Art der Punktgenerierung. Diese ergibt sich aus einer Balance aus neuer Zufallsrichtung und dem Mittel der bisherigen erfolgreichen Schrittrichtungen. Schlägt die Suche nach einem besseren Punkt fehl, so bekommt die neue Zufallsrichtung einen höheren Anteil als die "gemerkten erfolgreichen "Richtungen. Ist die Suche erfolgreich, so wird der Anteil der gemerkten Richtungen erhöht.

Wenn die Zielfunktion entsprechend Implementiert ist, so lassen sich die Funktionsauswertungen der gesamte Parametermenge gut parallelisieren.

Vorteile von (P)ADRS sind:

- das merken erfolgreicher Schrittrichtungen hat eine klare Driftrichtung zur Folge
- man braucht keine Ableitungsinformation
- man kann lokale Optima "überspringen"
- durch Resampling und Parallelisierung liegen kaum Ressourcen brach

Nachteile von PADRS sind:

- Konvergenz nahe des Optimums sehr langsam
- relativ komplexe Parameter zur Anpassung an das Problem

(P)ADRS ist sehr effizient um einen Startpunkt im Konvergenzbereich für Verfahren zu liefern, die an der Struktur des Problems sonst scheitern würden.

Newton Matrixiterativen Verfahren Unter Matrixiterativen Verfahren versteht man eine Klasse von ableitungsbasierten Verfahren, die in jedem Iterationsschritt ausgehend von einem Startpunkt versuchen ein lineares Gleichungssystem zu lösen, um eine Abstiegsrichtung h zu bestimmen.

$$Mh = -\nabla f, \qquad M \in \mathbb{R}^{n\times n},\ h, \nabla f \in \mathbb{R}^n.$$

Ist die Lösung des Gleichungssystems nicht möglich, oder führt die Lösung auf eine orthogonale Richtung zum Gradienten, so setzt man die Abstiegsrichtung auf den negativen Gradienten. Für diese Abstiegsrichtung wird dann eine Schrittweite λ bestimmt, wodurch sich der nächste Iterationspunkt ergibt.

$$x_{k+1} = x_k + \lambda * h$$

Matrixiterative Verfahren versuchen die Zielfunktion quadratisch anzunähern, d.h. ausgehend vom aktuellen Punkt, der Ableitung an diesem Punkt und zweiten Ableitungen an diesem Punkt (oder einer Approximation dieser) iterieren sie in die Richtung, in der eine Quadratische Funktion ihr Minimum hätte.

Newton Newton con cave Abstiegsverfahren Das Newton'sche Abstiegsverfahren nutzt als Systemmatrix M die Hessematrix H von f.

$$M_k = H(x_k)$$

Das Newton'sche Abstiegsverfahren konvergiert dank der Verwendung von Information zweiter Ordnung vergleichsweise sehr schnell (lokal quadratisch), allerdings liegt diese Information selten vor.

Newton BFGS-Verfahren Das BFGS-Verfahren (benannt nach Broyden,Fletcher, Goldfarb, Shanno) benutzt als Systemmatrix eine numerische Aufdatierungsformel, die unter gewissen Vorraussetzungen gegen die Hessematrix konvergiert.

$$M_{k+1} = M_k + \frac{1}{p_k^T r_k}\Big[(p_k - B_k r_k)\, p_k^T + \\ + p_k\,(p_k - B_k r_k)^T - \frac{r_k^T\,(p_k - B_k r_k)}{p_k^T r_k} p_k p_k^T\Big],$$

mit

$$p_k := x_{k+1} - x_k; \qquad r_k := \nabla f(x^{k+1}) - \nabla f(x^k)$$

Der Vorteil des BFGS-Verfahrens ist, dass es mit Ableitungen erster Ordnung auskommt, welche man sich auch numerisch relativ stabil verschaffen

kann. Es bildet quasi einen Spagat zwischen Gradientenabstieg und Newton'schem Abstieg.

[illegible] "Richtungsmengen Methoden" sind Optimierungsverfahren, die eine Menge an Richtungen iterativ verändern und in jedem Schritt entlang jeder dieser Richtungen eine eindimensionale Minimierung durchführen: Vorteile von DSM sind:

- hat das Tal eine Elliptische Form, und gelingt es dem Verfahren die Hauptrichtung der Ellipse zu bestimmen so ist die Minimierung entlang dieser Richtung sehr effizient
- man braucht keine Ableitungsinformation
- man kann Vorwissen über die Täler mit einfließen lassen.

Nachteile von DSM sind:

- ist das Tal gekrümmt, so bricht die eindimensionale Minimierung sehr früh ab, und das Verfahren wird sehr langsam
- gelingt es dem Verfahren nicht, die Hauptrichtung des Tals zu bestimmen, so Verläuft die Iteration im Zickzack Verhalten

[illegible] Der Powell Brent Optimizer gehört zur Klasse der Direction Set Methods, wobei für die Richtungsbestimmung Powells Methode angewendet wird, und die eindimensionale Minimierung durch Brents' Algorithmus erfolgt. Vor- und Nachteile des PBO sind die der allgemeinen DSMs.

2.2 Restringierte Optimierung

Zu lösen ist ein mathematisches Problem der Form:

$$\hat{x} = \arg\min_{x} f(x) \quad , \; x \in \mathbb{R}^n$$

Unter den Nebenbedingungen:

$$g_i(x) \leq 0 \quad , \; i \in \{1, \ldots, q\}$$
$$h_j(x) = 0 \quad , \; j \in \{1, \ldots, p\}$$

[illegible]

In der Regel zielen die Verfahren der restringierten Optimierung darauf ab, Punkte zu finden, in denen die sog. Karash-Kuhn-Tucker (KKT) Bedingungen erfüllt sind. Die KKT-Bedingungen stellen ein notwendiges Optimalitätskriterium erster Ordnung mit Nebenbedingungen dar und basieren auf der Lagrange Theorie.

Unter bestimmten Voraussetungen lässt sich folgende Aussage Beweisen: Ist $\bar{x} \in S$ (S ist die Restriktionsmenge) eine Minimalstelle von f auf S, und sind die Vektoren $\nabla h_1(x), \ldots, \nabla h_p(x), \nabla g_j(x) \quad , j \in I(\bar{x})$ linear unabhängig ($I(\bar{x})$ ist die Indexmenge der in $\bar{x}$ aktiven Ungleichheitsnebenbedingungen: $\forall i \in I(\bar{x}) : g_i(\bar{x}) = 0$), so gibt es Multiplikatoren $\mu_j \geq 0 \ (j \in I(\bar{x}))$ und $v_1, \ldots, v_p \in \mathbb{R}$ mit

$$\nabla f(\bar{x}) + \sum_{j \in I(\bar{x})} \mu_j \nabla g_j(\bar{x}) + \sum_{j=1}^{p} v_j \nabla h_j(\bar{x}) = 0_n$$

Die "verschobene Multiplikatoren Methode" gehört zu den sog. Straffunktionsmethoden. Straffunktionsmethoden beziehen die Nebenbedingungen in die Zielfunktion mit ein. Auf die neue Zielfunktion kann dann ein unrestringiertes Optimierungsverfahren angewendet werden. Es wird unterschieden zwischen inneren und äußeren Straffunktionsmethoden:

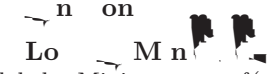
- *Innere Straffunktionsmethoden* arbeiten dabei mit Straffunktionen, die bereits innerhalb der Restriktionsmenge positive Werte annehmen, wenn man sich dem Rand nähert, d.h. sie "garantieren"die Einhaltung der Restriktionen allerdings verfälschen sie das Optimum evtl geringfügig
- *Äußere Straffunktionsmethoden* arbeiten mit Funktionen, die erst dann positive Werte liefern, wenn die Nebenbedingungen verletzt (also nicht eingehalten) werden. Sie neigen dazu, dass die Restriktionen geringfügig verletzt werden, während das Optimum geringfügig genauer approximiert werden kann, falls am Rand der Restriktionsmenge liegt

Bemerkung: Implementiert wurde die ALM mit äußeren Straffunktionen und variablem inneren Optimierer, d.h. die innere unrestringierte Optimierung kann durch die oben erwähnten Verfahren zu unrestringierten Optimierung erledigt werden. Somit ist das ALG Verfahren an die Charakteristik der Zielfunktion anpassbar.

Das Verfahren der sequenziellen quadratischen Angleichung ähnelt den Quasi-Newton-Verfahren für unrestringierte Probleme. Es bildet für jeder Iteration eine quadratische Näherung des Problems mit affin-linearen Nebenbedingungen. Für dieses quadratische Unterproblem wird dann in jeder Iteration eine interne restringierte Optimierung gestartet. Die Lösung des Unterproblems (, die z.B. mit der verschobenen Multiplikatoren Methode gewonnen werden kann,) repräsentiert eine Abstiegsrichtung, für die anschließend eine Schrittweite bestimmt wird (, in der Implementierung wird dies durch die "Iterative Dreiteilung nach dem goldenen Schnitt" erledigt). Steht der neue Iterationspunkt fest, so werden einige Parameter (unter anderem eine

Approximation der Hessematrix nach der BFGS-Formel) aufdatiert, die das Unterproblem für den nächsten Iterationsschritt festlegen. Das SQP Verfahren zählt heute zu den leistungsfähigsten Verfahren der nichtlinearen restringierten Optimierung.

2.3 Lokal vs Global


Mathematisch:
$\hat{x}$ heisst lokales Minimum von $f(x)$ genau dann wenn:

$$\exists U(\hat{x}) \neq \emptyset : \quad \forall x \in U : \quad f(\hat{x}) < f(x)$$

In Worten:
ein Punkt ist genau dann ein lokales Minimum, wenn es eine nichtleere Umgebung um den Punkt gibt, in der kein weiterer Punkt existiert, dessen Funktionswert kleiner oder gleich dem des lokalen Minimum ist.

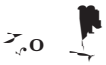

Mathematisch:
$\hat{x}$ heisst globales Minimum von $f(x)$ genau dann wenn:

$$\forall x \in \mathbb{D} : \quad f(\hat{x}) < f(x)$$

In Worten:
ein Punkt ist genau dann ein globales Minimum, wenn es in der Definitionsmenge keinen weiteren Punkt gibt, dessen Funktionswert kleiner oder gleich dem des globlen Minimum ist.

Bemerkung: Aufgrund der unterschiedlichen Fähigkeiten teilt man Optimierungsverfahren auch oft in "lokale"und "globale"Optimierungsverfahren ein - je nachdem ob ein Verfahren unabhängig vom Startpunkt in der Lage ist, das globale Minimum zu finden.

Bemerkung: Vorsicht: Nur weil man ein "globales"Verfahren verwendet, heisst das in der Regel nicht, dass das Resultat garantiert das globale Optimum ist.



Das Problem nahezu aller numerischen Optimierungsverfahren ist, dass man nicht sicher sein kann, ob es sich bei dem gefundenen Punkt nur um ein lokales oder um das globale Minimum handelt.

Um diesem Problem entgegenzuwirken gibts es mehrere Ansätze, unter anderem den Einstatz von Auflösungshierarchien oder "Meherere Startpunkte"-Strategien.

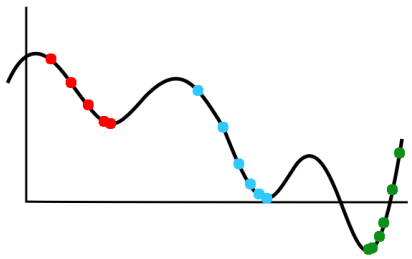


Abbildung 2: global VS lokal: Die Abbildung zeigt potentielle Pfade, die Beispielweise ein Gradientenabstiegsverfahren bei der gezeigten Funktion zurücklegen wird. Je nach Startpunkt wird ein anderes lokales Minimum gefunden - man kann nicht ohne weiteres feststellen, ob weitere, evtl. bessere Minima existieren

A [illegible] n [illegible] c[illegible]n Bei Auflösungshierarchien betrachtet man die Zielfunktion auf mehreren Auflösungsebenen von "gorb" nach "fein". Die Idee dahinter ist der Versuch zu vermeiden, dass man frühzeitig in einem lokalen Minimum konvergiert, das auf hohem Niveau liegt. Die Schwierigkeit liegt dabei darin, in wiefern und in welchem Maße man die Zielfunktion gröber betrachten kann. Das kann bei manchen Verfahren eine große Anfangsschrittweite, oder die Maskenbreite zur numerischen Gradientenbestimmung sein oder bei der Bildregistrierung die Registrierung geglätteter und verkleinerter Bilder. Es muss auch abgestimmt werden, wann die Ebene zu wechseln ist, um beispielsweise nicht unnötig genau auf gober Auflösung zu optimieren.

M[illegible] n[illegible] Benutzt man zufällig gewählte Startpunkte um ein Verfahren mehrmals anzuwenden erhöht das die Wahrscheinlichkeit, mit mindestens einem Startpunkt im Konvergenzbereich des besten Minimums zu landen. Allerdings steigert sich der Aufwand.

3 Implementierung

Details zu Implementierung können der Doxygen Dokumentation entnommen werden. Um beispielsweise eine Kostenfunktion zu implementieren muss von der allgemeinen CostFunction Klasse geerbt werden.

3.1 Parallelisieren, Rechnen auf der GPU

Wie das Auswerten der Kostenfunktion implementiert wird, bleibt dem Anwender überlassen, es muss lediglich das Interface der allgemeinen CostFuntion geerbt werden. Eine Implementierung auf der GPU ist durchaus

möglich. Um Funktionsauswertungen zu parallelisieren, kann die Funktion evaluateSet() überladen werden. Die Algorithmen sind darauf ausgelegt, dass wenn es die Möglichkeit gibt, mehrere Funktionsauswertungen "gleichzeitig" zu tätigen, die Funktion evaluateSet() aufgerufen wird. Wird diese Funktion nicht überladen, so ruft diese für jeden vorhandenen Parametervektor ein einzelnes evaluate() auf.

Bemerkung: die Algorithmen sind unterschiedlich stark geeignet, Teile parallel zu berechnen. Das geht bei (P)ADRS beispielsweise sehr viel besser (Auswertung der Punktmenge), als bei einem Downhill Simplex (keine lohnende Parallelisierung möglich).

4 Beispiele

Hier folgen zwei Beispiele, wie man die "OptimizerToolBox"benutzt.

4.1 Ein einfacher Optimierer

```
1  /**
2   * Ein Einfacher Optimiereraufruf
3   *
4   */
5  #include "StdioLog.h"
6  #include "BestNeighborOptimizer.h"
7  #include "CostFunction_Polynomial_1dim_4roots.h"
8  #include "AxA3dNUM/AxA3dNUMdoubleMtrx.h"
9
10 int main(int argc, char ** argv)
11 {
12    /* kostenfunktion */
13    CostFunction_Polynomial_1dim_4roots func;
14    StdioLog loger;
15    BestNeighborOptimizer optimizer(nop, &func, &
          loger);
16
17    doubleMtrx x(1,1);
18    x= 1.0;
19
20    /* um zu maximieren auf true setzen */
21    optimizer.setMaximize(false);
22
23    /* z.b. obere zeitgrenze setzen */
24    optimizer.setTimeLimit(true, 0.3);
25
26    /* startwert setzen */
27    optimizer.setParameters(x);
28
29    /* init muss vor optimize aufgerufen werden */
```

```
30      optimizer.init()
31
32      /* optimize startet die eigentliche optimierung
            */
33      optimizer.optimize();
34
35      /* das ergebnis holen */
36      doubleMtrx result = optimizer.getLastParameters()
            ;
37
38
39  }
```

4.2 Kombinieren verschiedener Optimierer

Folgendes Beispiel zeigt die Möglichkeiten der "OptimizerToolBox"schon besser auf:
Eine Funktion soll optimiert werden indem 100 zufällige Startpunkte generiert werden und ausgehend von diesen ein "Downhill Simplex Optimizer" gestartet wird. Die Resultate werden automatisch verglichen und das beste zurückgegeben.

```
1
2   /**
3    * Erweiterte Anwendung
4    *
5    */
6   #include "MultipleStartStrategyOptimizer.h"
7   #include "DownhillSimplexOptimizer.h"
8   #include "CostFunction_Polynomial_1dim_4roots.h"
9   #include "AxA3dNUM/AxA3dNUMdoubleMtrx.h"
10  #include "StdioLog.h"
11  #include <iostream>
12
13  int main(int argc, char ** argv)
14  {
15      CostFunction_Polynomial_1dim_4roots obj;
16      StdioLog loger;
17
18      /* scales legt die Skalierung der parameter
19         untereinander fest */
20      doubleMtrx scales(4,1);
21
22      scales = 10.0;
23      double timelimit = 1.0;
24
25      /* wieviele parameter hat die zielfunktion */
26      unsigned int nop = obj.getNumberOfParameters();
27
```

```
28      /* erzeugung des downhill simplex optimierers*/
29      DownhillSimplexOptimizer downhill(, &obj, &loger)
            ;
30
31      /* setzen des scale vektor */
32      downhill.setScales(scales);
33
34      /* begrenzung auf maximal 500 iterationen */
35      downhill.setMaxNumIter(true,500);
36
37      /* setVerbose(true) macht den optimierer
            gesprächig, sodass er zwischenergebnisse
            ausgibt. -> Gut zum einstellen eventueller
            Parameter */
38      downhill.setVerbose(false);
39
40      /* algorithmen spezifische parameter */
41      downhill.setDownhillParams(1, 0.5 ,1);
42
43      /* Parametertoleranz (Abbruchschranke) */
44      downhill.setParamTol(true, 0.05);
45
46      /* Funktionswerttoleranz (abbruchschranke) */
47      downhill.setFuncTol(true, 0.01);
48
49      /* anzahl der zu generierenden Startpunkte*/
50      unsigned int nstartpoints = 100;
51
52      /* Grenzen für die Generierung */
53      doubleMtrx lowerBound(4,1);
54      doubleMtrx upperBound(4,1);
55      lowerBound(0,0) = - 40.0;
56      lowerBound(1,0) = - 80.0;
57      lowerBound(2,0) = - 40.0;
58      lowerBound(3,0) = - 80.0;
59      upperBound(0,0) =  40.0;
60      upperBound(1,0) =  80.0;
61      upperBound(2,0) =  40.0;
62      upperBound(3,0) =  80.0;
63
64      /* erzeugung des MultipleStartStrategyOptimizer
            */
65      MultipleStartStrategyOptimizer multi(4,
            nstartpoints, &obj,&loger,&downhill,NULL,NULL
            );
66
67      /* setzen der oben erstellten Grenzen */
68      multi.setLowerParameterBound(true, lowerBound);
69      multi.setUpperParameterBound(true, upperBound);
70
```

```
71      /* keine extra Informationen ausgeben */
72      multi.setVerbose(false);
73
74      /* Das Zeitlimit ist gerade deaktiviert */
75      multi.setTimeLimit(false,1.0);
76
77      /* init() nicht vergessen*/
78      multi.init();
79
80      /* Optimierung starten und Rückgabewert speichern
            */
81      unsigned int retval = multi.optimize();
82
83      /* Überprüfen des Rückgabestatus*/
84      switch(retval)
85      {
86      case multi.SUCCESS_FUNCTOL:
87         std::cout << "   AbortReason was FuncTol" ;
88         break;
89      case multi.SUCCESS_PARAMTOL:
90         std::cout << "   AbortReason was ParamTol" ;
91         break;
92      case multi.SUCCESS_MAXITER:
93         std::cout << "   AbortReason was NumIter" ;
94         break;
95      case multi.SUCCESS_TIMELIMIT:
96         std::cout << "   AbortReason was TimeLimit" ;
97         break;
98      case multi.ERROR_XOUTOFBOUNDS:
99         std::cout << "   AbortReason was
               ERROR_XOUTOFBOUNDS" ;
100        break;
101     case multi.ERROR_COMPUTATION_UNSTABLE:
102        std::cout << "   AbortReason was
               ERROR_COMPUTATION_UNSTABLE" ;
103        break;
104     case multi.ERROR_INNER_OPT_FAILED:
105        std::cout << "   AbortReason was
               ERROR_INNER_OPT_FAILED" ;
106        break;
107     case multi.SUCCESS_ALLPOINTSOPTIMIZED:
108        std::cout << "   AbortReason was
               SUCCESS_ALLPOINTSOPTIMIZED" ;
109        break;
110     default:
111        std::cout << "   AbortReason was not
               recognized .. maybe it is not yet listed
               in the Test Scheduler" ;
112     };
113     std::cout << std::endl;
```

```
114
115     /* Resultat holen */
116     doubleMtrx Result= multi.getLastParameters();
```

Mit dem letzten Beispiel sollten sich schon einige Problem lösen lassen, ohne zu große Anpassungen tätigen zu müssen. Fast alle Optimierer haben Einstellungsmöglichkeiten, auf deren Hilfe man angewiesen ist.

4.3 Beispieleinstellungen auf Aufrufe

Die folgenden Unterkapitel sollen schematisch Darstellen, wie man die einzelnen Optimierer aufruft und typische Einstellungen setzt. Hierbei wurde nicht auf Komplettheit geachtet - es handelt sich lediglich um Ausschnitte, die Überprüfung des Rückgabegrundes wurde stets weggelassen.

```
    optimizer.setFuncTol(true,0.01);
    optimizer.setParamTol(true, 0.005);
    optimizer.setMaxNumIter(true, 100);
    optimizer.setTimeLimit(true, 1 );

    /* alg specific calls */
    optimizer.setStepSize(stepSize);
    optimizer.setNumberOfLevels(levels);
    optimizer.setDownscaleFactor(scalefactor);

    /* init and optimize call */
    optimizer.init();
    optimizer.optimize();

    /* get the result */
    x = optimizer.getLastParameters();
}
```

```
/*
    test Gradient Descent in analytic mode (uses
        gradient of costfunction if available)
*/
void testGradientDescentInAnalyticMode(CostFunction
    *func)
{
    unsigned int nop = func->getNumOfParameters();
    StdioLog loger;

    /* construction */
    GradientDescentOptimizer optimizer(nop, func, &
        loger);

    /* some locals */
    doubleMtrx x(nop,1);
    doubleMtrx stepSize(nop,1);
    doubleMtrx scales(nop,1);
    doubleMtrx lower(nop,1);
    doubleMtrx upper(nop,1);
    lower = -10;
    upper = 30;
    x = ((double)rand()) / RAND_MAX;
    stepSize = 1;
    scales = 1;

    /* general calls */
    optimizer.setLowerParameterBound(true, lower);
```

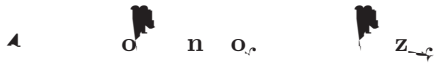
```
    optimizer.setUpperParameterBound(true, upper);
    optimizer.setParameters(x);
    optimizer.setScales(scales);
    optimizer.setMaximize(false);
    optimizer.setFuncTol(true,0.01);
    optimizer.setParamTol(true, 0.005);
    optimizer.setMaxNumIter(true, 100);
    optimizer.setTimeLimit(true, 1);

    /* alg specific calls */
    optimizer.setStepSize(stepSize);
    optimizer.useAnalyticalGradients(true);
    optimizer.setGradientTol(true, 0.01);

    /* init and optimize call */
    optimizer.init();
    optimizer.optimize();

    /* get the result */
    x = optimizer.getLastParameters();

}
```

```
/*
    test grid search optimizer
*/
void testGridSearch(CostFunction *func)
{
    unsigned int nop = func->getNumOfParameters();
    StdioLog loger;

    /* construction */
    GridSearchOptimizer optimizer(nop, func, &loger);

    /* some locals*/
    doubleMtrx x(nop,1);
    doubleMtrx stepSize(nop,1);
    doubleMtrx scales(nop,1);
    doubleMtrx lower(nop,1);
    doubleMtrx upper(nop,1);
    unsigned int levels = 3;
    double downscale = 0.2;
    lower = -5;
    upper = 10;
    stepSize = 0.1;
    x = 2;
```

```
    scales = 1;


    /* general calls */
    optimizer.setTimeLimit(true,15.0);
    optimizer.setScales(scales);
    optimizer.setLowerParameterBound(true, lower);
    optimizer.setUpperParameterBound(true, upper);
    optimizer.setParameters(x);


    /* alg specific calls */
    optimizer.setStepSize(stepSize);
    optimizer.setNumberOfLevels(levels);
    optimizer.setDownscaleFactor(downscale);

    /* init and optimize calls */
    optimizer.init();
    optimizer.optimize();

    /* get the result */
    x = optimizer.getLastParameters();
}
```

```
/*
    test the downhill simplex optimizer
*/
void testDownhillSimplex(CostFunction *func)
{

    unsigned int nop = func->getNumOfParameters();
    StdioLog loger;


    /* construction*/
    DownhillSimplexOptimizer optimizer(nop, func, &
        loger);

    /* some locals */
    doubleMtrx x(nop,1);
    doubleMtrx simplex(nop,nop+1);
    doubleMtrx scales(nop,1);
    doubleMtrx lower(nop,1);
    doubleMtrx upper(nop,1);
    x=1.0;
    lower = -20;
```

```
22      upper = 40;
23      scales = 1;
24
25      /* random initialization of the simplex */
26      for(int i = 0; i < nop; i++)
27      {
28         for(int j = 0; j < nop+1; j++)
29         {
30            simplex(i,j) = x(i,0);
31            if( j == i+1 )
32            {
33               double tmpRand = ((double)rand()) /
                     RAND_MAX;
34               simplex(i,j) += tmpRand;
35            }
36         }
37      }
38
39
40      /* general calls */
41      optimizer.setScales(scales);
42      optimizer.setLowerParameterBound(true, lower);
43      optimizer.setUpperParameterBound(true, upper);
44      optimizer.setFuncTol(true, 1.0e-1);
45      optimizer.setParamTol(true, 1.0e-2);
46      optimizer.setMaxNumIter(true, 200);
47      optimizer.setTimeLimit(true, 1   );
48
49      /* alg specifig calls*/
50      optimizer.setWholeSimplex(simplex);
51      optimizer.setDownhillParams(1, 0.5 ,1);
52
53
54      optimizer.init();
55      optimizer.optimize();
56
57      x = optimizer.getLastParameters();
58  }
```



```
1   /*
2      test a combinatorial optimizer
3   */
4   void testCombinatorialOptimizer(CostFunction *func)
5   {
6      unsigned int nop = func->getNumOfParameters();
7      StdioLog loger;
```

```
8
9       /* construction */
10      CombinatorialOptimizer optimizer(nop, func, &
            loger);
11
12      /* some locals */
13      doubleMtrx x(nop,1);
14      doubleMtrx scales(nop,1);
15      double steplength = 5.0;
16      double T0 = 2.0;
17      double alpha = 0.95;
18      unsigned int mode = 4;
19      x = ((double)rand()) / RAND_MAX;
20
21
22      /* alg specifig calls */
23      optimizer.setMode(mode);
24      optimizer.setSteplength(steplength);
25      optimizer.setControlSeqParam(T0,alpha);
26
27      /* general calls */
28      optimizer.setParameters(x);
29      optimizer.setScales(scales);
30      optimizer.setMaximize(false);
31      optimizer.setVerbose(false);
32      optimizer.setMaxNumIter(true, 1000);
33      optimizer.setTimeLimit(true, 3 );
34
35      /* init and optimize call*/
36      optimizer.init();
37      optimizer.optimize();
38
39      /* get the result */
40      x = optimizer.getLastParameters();
41  }
```

```
1   /*
2       test the adaptive direction random search
            optimizer
3   */
4   void testADRSOptimizer(CostFunction *func)
5   {
6
7       unsigned int nop = func->getNumOfParameters();
8       StdioLog loger;
```

```
    unsigned int nopp = 20; // number of parallel
        points

    /* construction */
    AdaptiveDirectionRandomSearchOptimizer optimizer(
        nop, nopp, func, &loger);

    /* some locals */
    doubleMtrx x(nop,1);
    doubleMtrx scales(nop,1);
    double scalefactor = 0.2;
    int numberOfCriteria = 0;
    int maxNumberOfCriteria = 2;
    doubleMtrx lower(nop,1);
    doubleMtrx upper(nop,1);
    lower = -10;
    upper = 30;
    x = ((double)rand()) / RAND_MAX;
    scales = 1;



    /* the bounds are needed for random number
        generation*/
    optimizer.setLowerParameterBound(true, lower);
    optimizer.setUpperParameterBound(true, upper);

    /* general calls */
    optimizer.setParameters(x);
    optimizer.setScales(scales);
    optimizer.setMaximize(false);
    optimizer.setMaxNumIter(true, 50);
    optimizer.setTimeLimit(true, 0.1 );

    /* init and optimize */
    optimizer.init();
    optimizer.optimize();

    /* get the result */
    x = optimizer.getLastParameters();
}
```

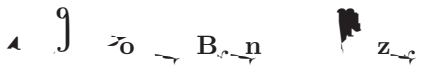
B

```
/*
    test the Broyden Flecher Goldfarb Shanno
        Optimizer
*/
```

```
4  void testBFGSOptimizer(CostFunction *func)
5  {
6
7     unsigned int nop = func->getNumOfParameters();
8     StdioLog loger;
9
10    /* the constructor call */
11    BFGSOptimizer optimizer(nop, func, &loger);
12
13    /* generate local variables to set the according
          options */
14    doubleMtrx x(nop,1);
15    doubleMtrx stepSize(nop,1);
16    doubleMtrx scales(nop,1);
17    doubleMtrx lower(nop,1);
18    doubleMtrx upper(nop,1);
19
20    lower = -10;
21    upper = 30;
22    x = ((double)rand()) / RAND_MAX;
23    stepSize = 0.5;
24    scales = 1;
25
26    /* alg specific call */
27    optimizer.setStepSize(stepSize);
28
29    /* general calls */
30    optimizer.setParameters(x);
31    optimizer.setScales(scales);
32    optimizer.setMaximize(false);
33    optimizer.setFuncTol(true,0.0001);
34    optimizer.setParamTol(true, 0.000005);
35    optimizer.setMaxNumIter(true, 50);
36    optimizer.setTimeLimit(true, 1.0 );
37    optimizer.setLowerParameterBound(true, lower);
38    optimizer.setUpperParameterBound(true, upper);
39
40
41    /* init and optimize call */
42    optimizer.init();
43    optimizer.optimize();
44
45    /* get result */
46    x = optimizer.getLastParameters();
47 }
```

```
/*
   test the SQP optimizer
*/
void testSQPOptimizer()
{
   unsigned int nop =  2; //number of parameters
   unsigned int noec = 1; //number of equality
       constraints
   unsigned int noic = 2; //number of inequality
       constraints

   /* construction of loger, costfunction and
       constraints */
   StdioLog loger;
   CostFunction_ndim_2ndOrder       costFunc(nop);
   InequalityConstraint_ndim_1stOrder  ineqConst(nop
       ,noic);
   EqualityConstraint_ndim_1stOrder eqConst(nop,noec
       );

   /* initialize the parametric inequality
       constraints */

   ... cutted ...

   /* Constructor */
   AugmentedLagrangianOptimizer  InnerOpt(nop, NULL,
        NULL, NULL, &loger, NULL);

   /* some local variables */
   doubleMtrx x(nop,1);
   doubleMtrx scales(nop,1);
   doubleMtrx lower(nop,1);
   doubleMtrx upper(nop,1);
   doubleMtrx alpha(noic);
   doubleMtrx beta(noec);
   doubleMtrx s(noec+noic);
   alpha =1.0;
   beta=1.0;
   s=10.0;
   s(0,0) = 100;
   lower = -10;
   upper = 30;
   scales = 1.0;
   x = ((double)rand()) / RAND_MAX;
   scales = 1;


   /* general calls */
   InnerOpt.setLowerParameterBound(true, lower);
```

```
44      InnerOpt.setUpperParameterBound(true, upper);
45      InnerOpt.setScales(scales);
46      InnerOpt.setParamTol(true,0.005);
47
48      /* alg specific calls */
49      if( 0 != InnerOpt.setAlgorithmParameters(alpha,
            beta, s, 5, 0, 0.01) )
50      {
51          std::cout << "setAlgorithmParameters failed"
                << std::endl;
52      }
53
54      /*    build SQP optimizer */
55      SQPOptimizer   optimizer(nop, &costFunc, &eqConst
            ,
56                               &ineqConst, &loger, &
                                     InnerOpt);
57
58      /* general calls */
59      optimizer.setLowerParameterBound(true, lower);
60      optimizer.setUpperParameterBound(true, upper);
61      optimizer.setParamTol(true,0.05);
62      optimizer.setParameters(x);
63      optimizer.setScales(scales);
64      optimizer.setMaximize(false);
65      optimizer.useAnalyticalGradients(true);
66      optimizer.setTimeLimit(true, 20.0);
67      optimizer.setParamTol(true,0.00000005);
68      optimizer.setFuncTol(true, 0.00000003);
69
70      /* get ouputs of the inner optimizer to adjust
            settings */
71      optimizer.setVerbose(false);
72      InnerOpt.setVerbose(true);
73
74      /* init and optimize call*/
75      optimizer.init();
76      optimizer.optimize();
77
78      /* get the result */
79      x = optimizer.getLastParameters();
80  }
```



```
1   /*
2       test the powell brent optimizer
3   */
```

```
4  void testPowellBrentOptimizer(CostFunction *func)
5  {
6
7     unsigned int nop = func->getNumOfParameters();
8     StdioLog loger;
9
10    /* Constructor */
11    PowellBrentOptimizer optimizer(nop, func, &loger)
          ;
12
13    /* generate and set start point */
14    doubleMtrx x = ((double)rand()) / RAND_MAX;
15    optimizer.setParameters(x);
16
17    /* init and optimize call */
18    optimizer.init();
19    optimizer.optimize();
20
21    /*  get result */
22    x = optimizer.getLastParameters();
23
24 }
```

5 Optimization Quick Guide

Es folgt eine Auflistung von Stichpunkten, deren Abarbeitung die Anwendung der Toolbox erleichtern könnte. Ausgehend von der Problemstellung:

1. Kostenfunktion implementieren (von der allgemeinen CostFunction erben)
2. analytische Ableitungen berechnen, falls möglich
3. Plotten der Kostenfunktion um Charakteristik einschätzen zu können (besonders wellig, glatt, etc...)
4. Wahl eines passenden Optimierers (entsprechend der Vor- und Nachteile in obiger Ausführung)
5. Optimierer starten und Resultate überprüfen
6. Anhand der Resultate die Parameter des Algorithmus anpassen bis ein zufriedenstellendes Verhalten erreicht wird

6 Vermissen Sie etwas?

Wo sind . . . ?

- TrustRegion Verfahren ⇒ TrustRegion Verfahren fallen in der Bereich der Restringierten Optimierung. Der Anwender sei auf das SQP-Verfahren oder die "Augmented-Lagrangian-Method" verwiesen. Zum Zeitpunkt der Erstellung ist kein Trust-Region Verfahren Implementiert.

- Levenberg-Marquardt ⇒ Der LM- Algorithmus fällt in den Bereich der Quasi-Newtonverfahren. Da oft das BFGS-Verfahren für Leistungsstärker gehalten wird, sei der Anwender auf dieses verwiesen - der LM Algorithmus ist zum Zeitpunkt der Erstellung der Toolbox nicht Implementiert.
- Eindimensionale Verfahren? (Einschachtelung, Sekantenverfahren, etc.) ⇒ Zum Zeitpunkt der Erstellung ist "nur" das Verfahren der Sukzessiven Dreiteilung nach dem "Goldenen Schnitt" und das Verfahren von Brent implementiert.

Was ist mit Graphenoptimierung ?

Da sich die Optimierung auf Graphen sehr stark von der allgemeinen kontinuierlichen nichtlinearen Optimierung unterscheidet, konnte dieser große Bereich nicht mit abgedeckt werden. Der Leser sei an dieser Stelle auf andere Quellen verwiesen.