ulm university universität
uulm

CSE-Projektbericht

Numerische Optimierung eines optischen Systems

Leandro Ebner, Lewin Ernst, Michael Weinstein, Patrick Diem

In Kooperation mit Firma Zeiss

15. Februar 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

List of Algorithms

Code Liste

1 Einleitung

1.1 Zielsetzung

Dieses Projekt entstand in Zusammenarbeit mit der Firma Zeiss und hatte zum Ziel verschiedene numerische Optimierungsverfahren auf optische Systeme anzuwenden. Optische Systeme müssen gewisse Anforderungen erfüllen, welche in Kapitel 1.2 genauer erläutert werden. Um diese zu erfüllen, müssen die Bauteile des Systems gewisse Eigenschaften aufweisen welche wiederum unterschiedlicher Natur sein können. Sowohl geometrische, wie auch materielle Eigenschaften beeinflussen den Verlauf des Lichts durch die Bauteile des Systems.

Um geeignete Eigenschaften zu finden, die das gewünschte Verhalten erzeugen, ist es aufgrund der Vielzahl an möglichen Variationen nicht ohne weiteres möglich diese experimentell zu bestimmen, weshalb numerische Optimierungsverfahren benötigt werden. Im Rahmen dieses Projekts wurden dabei nur geometrische Eigenschaften optimiert.

Um eine numerische Optimierung der geometrischen Eigenschaften vornehmen zu können, muss eine Zielfunktion definiert werden, die die Abweichung vom soll/ist-Zustand repräsentiert. Diese Zielfunktion ist in Kapitel 3.3 genauer definiert, behandelt aber grundsätzlich die Abweichung der Position der Lichtstrahlen auf der Bildfläche, verglichen mit dem Wunschzustand.

Um die Zielfunktion auswerten zu können werden somit die Positionen der Lichtstrahlen am Ende des Systems (Bildfläche) benötigt. Diese Berechnung wird von Raytracern vorgenommen. Im Rahmen dieses Projektes wurde ein bereits vorhandener Raytracer verwendet, und an die Anforderungen der Verfahren und Systeme in diesem Projekt angepasst.

Das Projekt setzte sich also aus folgenden zentralen Arbeitsabschnitten zusammen:

- Installation und nachvollziehen des Aufbaus des Raytracers
- Anpassen des Raytracers an die Anforderungen des Projektes
- Implementieren verschiedener Optimierungsverfahren
- Bewertung der Verfahren

1.2 Technische Optik[1]

Trifft ein Lichtstrahl auf die Grenzfläche zweier Materialien, so wird ein Teil dieses Lichtstrahls gebrochen. Dabei gilt das *Snelliussche Brechungsgesetz*:

$$n_1 \sin(\alpha) = n_2 \sin(\beta) \tag{1.1}$$

Dabei beschreiben n_1 bzw. n_2 den Brechungsindex des oberen bzw. unteren Materials (siehe Abb. 1.1).

[1] Nach [14]

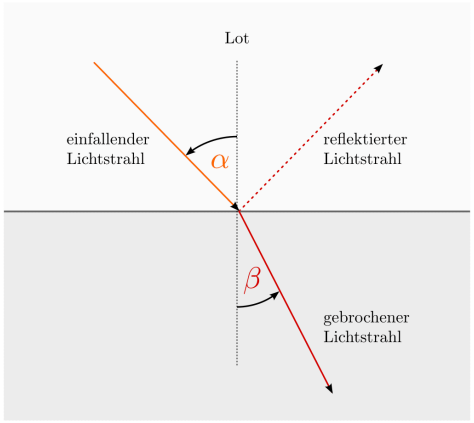

Abbildung 1.1: Lichtbrechung [][grund-wissen.de/physik/optik/lichtbrechung.html am 08.02.2020]

Werden nun Lichtstrahlen durch ein Linsensystem gesendet, kann für jeden Lichtstrahl mittels Formel 1.1 der zurückgelegte Weg und die Position auf der Bildfläche berechnet werden. Ein Strahlenbündel, welches von einem Objektpunkt ausgeht, wird im Idealfall durch das System als Punkt auf der Bildebene dargestellt. Aufgrund von sog. *Abbildungsfehlern* ist dies jedoch in der Praxis nur sehr schwer für alle Objektpunkte realisierbar.

Im Folgenden sollen erst optische Begriffe erläutert werden, welche für das Projekt von Bedeutung waren. Anschließend werden verschiedene Arten der Abbildungsfehlern erläutert. Dabei soll dieses Kapitel lediglich einen kurzen Einblick in die technische Optik geben und hat keinen Anspruch auf Vollständigkeit.

1.2.1 Begriffe aus der technischen Optik

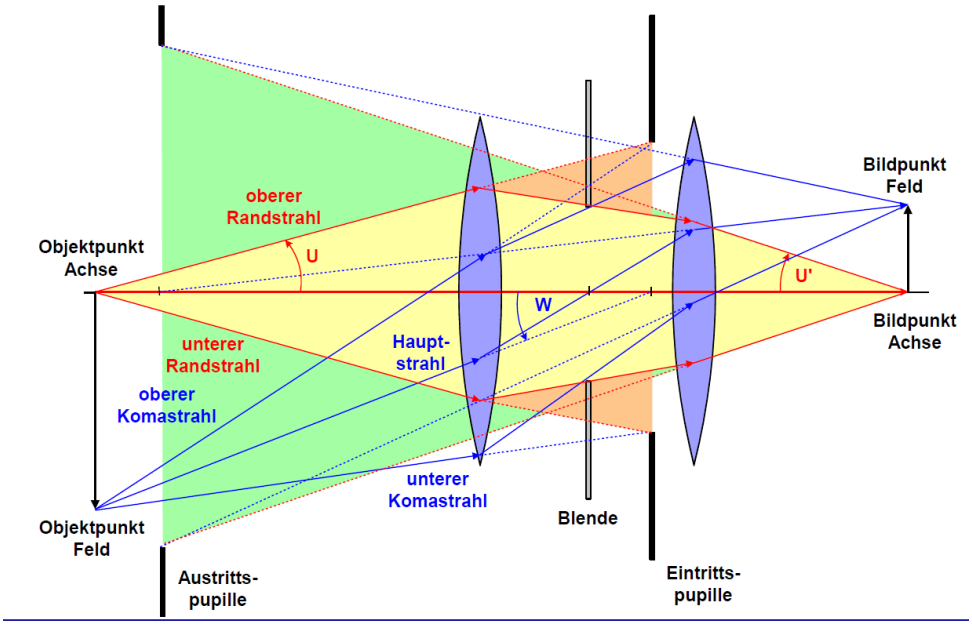

Abbildung 1.2: Begriffe der techn. Optik, entnommen aus [5], S. 445

Blende Begrenzt die Öffnungsweite (Winkel der einfallenden Strahlen) und somit die Helligkeit des Bildes

Eintrittspupille (Reelles oder virtuelles) Bild der Blende, entworfen mit dem Systemteil **vor** der Blende (siehe Abb. 1.2)

Austrittspupille (Reelles oder virtuelles) Bild der Blende, entworfen mit dem Systemteil **hinter** der Blende (siehe Abb. 1.2)

Hauptstrahl Strahl eines Bündels, der objektseitig durch die Mitte der Eintrittspupille geht, und bildseitig durch die Mitte der Austrittspupille (d.h. nicht der Strahl selbst verläuft durch die Mitte, sondern die Verlängerung betrachtet von der Objekt-/Bildseite)

Komastrahl Strahl, welcher von einem außeraxialem Objektpunkt durch den Pupillenrand verläuft

1.2.2 Abbildungsfehler

Allgemein unterscheidet man zwischen monochromatischen und chromatischen Abbildungsfehlern. Letzteres beschreibt den Fehler, der durch unterschiedliche Wellenlängen des Lichts (Farben) entsteht. Abhängig von dieser, wird der Lichtstrahl verschieden stark gebrochen. Ursache hierfür ist, dass der Brechungsindex eines Materials von der Wellenlänge des Lichtstrahls abhängt.

Monochromatische Abbildungsfehler können verschiedene Ursachen haben. Nachfolgend werden zwei dieser Gründe beispielhaft aufgeführt und kurz beschrieben.

Öffnungsfehler Parallel einfallende oder von einem Objektpunkt auf der opt. Achse stammende Strahlen fallen bildseitig nicht in einem Punkt zusammen (siehe Abb. 1.3)

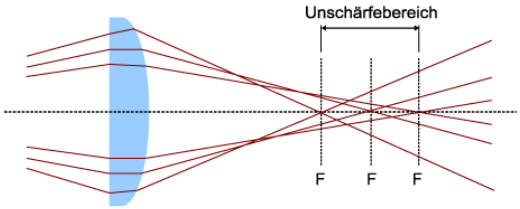

Abbildung 1.3: Öffnungsfehler [*univie.ac.at/mikroskopie/1_grundlagen/optik/opt_linsen/5a_sphaerisch.htm* am 08.02.2020]

Verzeichnung Bildhöhe (Abstand eines Bildpunkts von der Bildmitte) hängt auf nichtlinearer Weise von der Höhe des entsprechenden Objektpunkts ab (siehe Abb. 1.4)

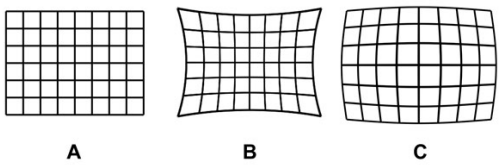

Abbildung 1.4: Verzeichnung [*slashcam.de/artikel/Basiswissen-Videoproduktion/Verzeichnung.html* am 08.02.2020]

2 Pyrate

Wie bereits in der Einleitung erwähnt, ist es nötig die Positionen der Strahlen auf der Bildfläche zu berechnen, um die Zielfunktion auswerten zu können. In diesem Projekt wurde für diese Aufgabe der Raytracer pyrate[1] verwendet, welcher ausschließlich in python programmiert ist.

Dieses Kapitel dient dazu den Aufbau und insbesondere die notwendigen Anpassungen des Raytracers zu beschreiben. Darüber hinaus sollen die wichtigsten Punkte über die Bedienung und sonstige Dateien erläutert werden.

2.1 Übersicht über verschiedene Klassen

Bevor auf den Aufbau und die benötigten Klassen des Raytracers genauer eingegangen wird, soll vorab eine kurze und auf die wichtigsten Klassen und Funktionen beschränkte Übersicht gegeben werden. Das UML-Diagram soll eine Orientierungshilfe für die späteren Kapitel darstellen.

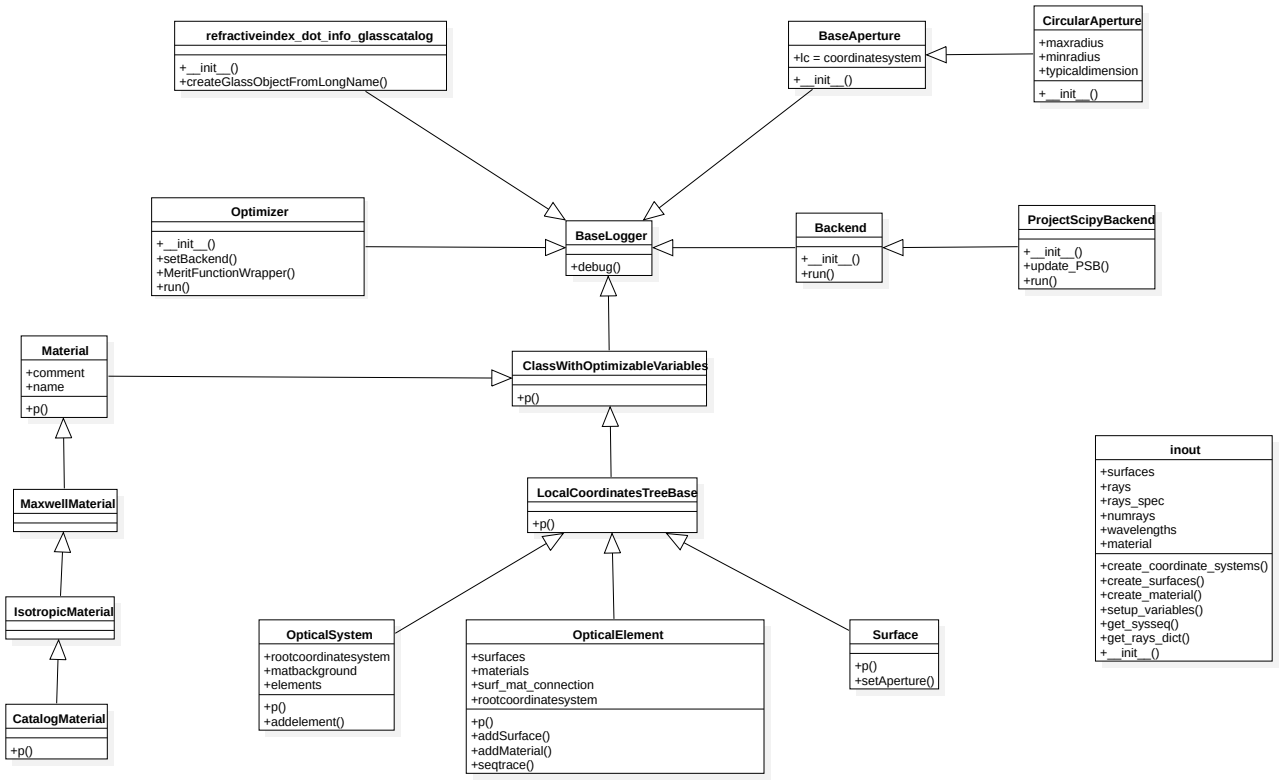

Abbildung 2.1: UML Diagramm verschiedener pyrate Klassen

[1] `https://github.com/mess42/pyrate`

2.2 Aufbau des optischen Systems

2.2.1 Geometrie

Geometriedaten bzw. Bauteile eines optischen Systems können in der Standardversion von pyrate auf zwei verschiedenen Wegen definiert werden. Zum einen können mit Hilfe eines CAD Tools Bauteile gezeichnet werden, zum anderen können diese auch direkt im Quellcode erzeugt werden. Da im Rahmen dieses Projektes die CAD Variante nicht zum Laufen gebracht werden konnte, wurde das Einlesen der Bauteile in Textfiles ausgelagert, um Anpassungen am System vornehmen zu können, ohne Veränderungen am Quellcode durchführen zu müssen. Im Ordner `Input_Output` befindet sich die Datei `surface_input` in der die Oberflächen der Bauteile festgelegt werden können. Das Einlesen der Geometriedaten erfolgt beim Ausführen der Hauptdatei automatisch.

```
# One row in this data-sheet stands for one surface with the properties

#NameSurface        decx        decxvar     decy        decyvar     decz        deczvar     tiltx
aperture            0.0         f           0.0         f           0.0         f           0.0
surf1               0.0         f           0.0         f           20.0        -20.0&70.0  0.0
surf2               0.0         f           0.0         f           10.0        2.0&72.0    0.0
surf3               0.0         f           0.0         f           5.00        0.0&70.0    0.0
surf4               0.0         f           0.0         f           10.0        2.0&72.0    0.0
surf5               0.0         f           0.0         f           5.00        0.0&70.0    0.0
surf6               0.0         f           0.0         f           10.0        2.0&72.0    0.0
surf7               0.0         f           0.0         f           5.00        0.0&70.0    0.0
surf8               0.0         f           0.0         f           10.0        2.0&72.0    0.0
image               0.0         f           0.0         f           100.0       0.0&170.0   0.0
```

Abbildung 2.2: `surface_input` Datei

Die Eigenschaften der Elemente wie z.B. die Position des Elements im Raum (`decx, decy, decz` siehe Abb. 2.2) können in dieser Datei festgelegt werden. Zu jeder möglichen optimierbaren Variable gibt es eine Spalte `*namedervariable*var`, welche angibt, ob die Eigenschaft optimiert werden soll. Dabei bedeutet `f`, dass der Wert der Größe fixiert bleiben soll, also nicht in die Optimierung mit eingeht, und z.B. `-5.0&10.0`, dass die Größe optimiert werden soll, aber den Bereich zwischen -5.0 und 10.0 nicht verlassen darf. Hier werden also die Contstraints der Eigenschaften festgelegt. Tabelle 2.1 bietet eine Übersicht über die unterstützen Eigenschaften einer Oberfläche.

Tabelle 2.1: Variablen der `surface_input` file

Variable	Bedeutung	Optimierbar J/N	Bemerkung
decx	Orientierung der Fläche bzgl. x- Achse	Ja	Wert relativ zur vorherigen Fläche
decy	Orientierung der Fläche bzgl. y- Achse	Ja	Wert relativ zur vorherigen Fläche
decz	Orientierung der Fläche bzgl. z- Achse	Ja	Wert relativ zur vorherigen Fläche
tiltx	Rotation der Fläche bzgl. x- Achse	Ja	-
tilty	Rotation der Fläche bzgl. y- Achse	Ja	-
tiltz	Rotation der Fläche bzgl. z- Achse	Ja	-
ca_max_rad	Maximale Größe Blende	Ja	-
ca_min_rad	Minimale Größe Blende	Ja	-
Conic_cc	Krümmung	Ja	-
Conic_curve	Konische Konstante	Ja	Beschreibt die Art der Fläche [2]
Connection1	Material vor Element	Nein	-
Connection2	Material des Elements	Nein	-
isStop	Sollen Strahlen gestoppt werden?	Nein	Nur bei Bildfläche nötig
Shape	Welches Objekt soll erzeugt werden	Nein	Nur Conic implementiert
Aperture	Art der Blende	Nein	Nur Circular implementiert

Wie die in der Textfile angegebenen Informationen verarbeitet werden, ist in Abschnitt 2.3 genauer erläutert.

[2] $-1 < cc < 0$ oblate rotational ellipsoid, $cc = 0$ sphere, $0 < cc < 1$ prolate rotational ellipsoid, $cc = 1$ rotational paraboloid, $cc > 1$ rotational hyperboloid

2.2.2 Materialien

Wie in Tabelle 2.1 gezeigt, können die Materialen durch die `surface_input` Datei definiert werden. Pyrate wurde dabei so abgeändert, dass Materialen aus einem online Katalog verwendet werden können. Dieser Katalog muss nach dem Download von pyrate manuell nachinstalliert und unter dem Pfad `../pyrateoptics/refractiveindex.info-database/database/` abgespeichert werden. Es können beliebige Kataloge mit einem gewissen Aufbau verwendet werden. Für dieses Projekt wurde der Katalog von polyanskiy[3] verwendet.

2.2.3 Strahlenbündel

Pyrate benötigt Strahlenbündel, um deren Weg durch das optische System bestimmen zu können. Diese Strahlenbündel werden in der Datei `bundle_input` definiert. Die dort definierten Bündel werden automatisch in Objekte umgewandelt, welcher der Raytracer interpretieren kann.

```
#startx    starty    startz    radius    anglex    angley    raster
0          0         0         16        0.0       0         RectGrid
0          0         0         16        0.1832595 0         RectGrid
```

Abbildung 2.3: `bundle_input` Datei

Die Datei enthält dabei folgende Informationen:

Tabelle 2.2: Variablen der `surface_input` file

Variable	Bedeutung	Bemerkung
startx	Startpunkt des Bündels in x-Richtung	-
starty	Startpunkt des Bündels in y-Richtung	-
startz	Startpunkt des Bündels in z-Richtung	-
radius	Radius bzw. Größe des Rasters	-
anglex	Winkel der Strahlen in x-Richtung	-
angley	Winkel der Strahlen in y-Richtung	-
raster	Raster in dem die Bündel starten	Aktuell nur RectGrid verfügbar

In Abb. 2.4 ist die Bedeutung der Parameter visualisiert.

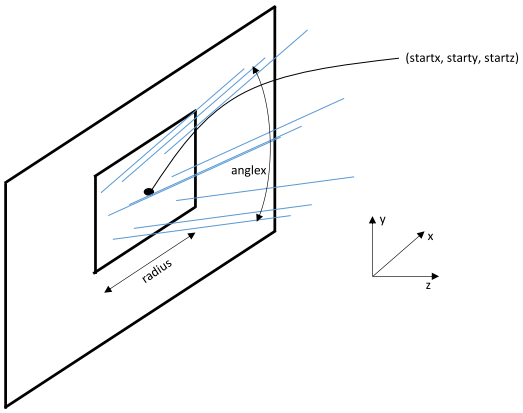

Abbildung 2.4: Visualisierung der bundle Parameter

Da Licht aus verschiedenen Wellenlängen besteht, ist es notwendig dies auch im Raytracer abzubilden. Die Wellenlängen sowie die Anzahl der Strahlen je Bündel können im angepassten

[3] `https://github.com/polyanskiy/refractiveindex.info-database`

Raytracer in der Datei `bundle_spec` angegeben werden. Jedes Bundle das in `bundle_input` definiert ist, wird für alle Wellenlängen die in `bundle_spec` angegeben sind erstellt. Ebenfalls haben der Einfachheit halber alle Bündel gleich viele Strahlen.

Die eingelesenen Daten werden in einer Form verarbeitet, die die neu geschriebene Funktion `buildInitialbundle` interpretieren kann. Diese befindet sich in der Datei `aux_merit_bundle.py` und wandelt die Informationen in Bündel um, die von pyrate interpretiert werden können.

2.2.4 Klasse `OpticalSystem` und Unterklassen

Die Klasse `OpticalSystem` ist die höchste Klasse, die schlussendlich alle Informationen über das optische System enthält.

Ein Objekt der Klasse `OpticalSystem` wird nicht über einen klassischen Konstruktor (`__init__`) erstellt, sondern über die Klassenmethode `p`. Ein solches Objekt enthält standardmäßig eine Objekt- und Bildfläche und legt das Hintergrundmaterial fest. Darüber hinaus kann durch die Funktion `seqtrace` der Raytracer gestartet werden. Um weitere Elemente wie z.B. Linsen hinzufügen zu können, wird eine Funktion `addElement` angeboten.

Elemente werden als eigene Klasse `OpticalElement` dargestellt. Wie schon bei `OpticalSystem` wird ein Objekt durch eine Klassenfunktion erstellt. Die `seqtrace` Funktion bei `OpticalSystem` ruft wiederum die `seqtrace` Funktion von `OpticalElement` auf. Dieses Objekt ist also ein weiterer Zwischenschritt zur eigentlichen Berechnung der Strahlverläufe. `OpticalElement` besitzt darüber hinaus Funktionen mit denen sich neue Oberflächen hinzufügen lassen. Nachdem die in der `surface_input` Datei definierten Oberflächen zu einem `Surface` Objekt umgewandelt wurden, kann mit dieser Funktion eine Oberfläche zu dem optischen Element hinzugefügt werden. Des Weiteren wird mit der Funktion `addMaterial` das Material, welches ebenfalls in `surface_input` definiert und anschließen beim Einlesen zu einem Objekt der Klasse `CatalogMaterial` umgewandelt wurde, dort den Oberflächen zugeordnet. Das `CatalogMaterial` Objekt wird mit der Funktion `createGlassObjectFromLongName()` erstellt.

Objekte der Klasse `Surface` werden ebenfalls durch eine Klassenfunktion erstellt. Dabei muss zwingend ein bereits erzeugtes Koordinatensystem angegeben werden. Darüber hinaus gibt es die Möglichkeit eine Form und eine Blende festzulegen. Die Form und die geometrischen Daten werden wie zuvor gezeigt in der `surface_input` Datei definiert.

2.3 Aufbau des Hauptprogramms

Im Standard pyrate Code werden sowohl die Parameter für Bündel und Geometrie sowie diverse andere Funktionen in einer zentralen main Datei verwaltet. Um die Flexibilität und Übersichtlichkeit des Programms zu erhöhen wurden wie bereits gezeigt die Parameter, aber auch diverse Funktionen ausgelagert und abgeändert. Dies führte zu einer starken Veränderung des Hauptprogramms. Der Aufbaudes neuen Programms, welcher in Listing 2.1 dargestellt ist, soll hier nun erläutert werden.

Listing 2.1: Aufbau des Hauptprogramms

```
1  import......
2  fi1          = inout()
3  s            = OpticalSystem.p()
4  cs           = fi1.create_coordinate_systems(s)
5  elem1        = OpticalElement.p(cs[0], 'name'=elem1)
6  surf         = fi1.create_surfaces(cs)
7  fi1.create_material(cs, elem1, surf)
8  s.addElement(elem1.name, elem1)
9  sysseq       = fi1.get_sysseq(elem1)
10 osa          = OpticalSystemAnalysis(s, sysseq)
11 rays_dict    = fi1.get_rays_dict()
12 (initialbundle, meritfunctionrms) =
13         get_bundle_merit(osa, s,.........)
14 fi1.setup_variables(s,elem1.name)
15 opt_backend = ProjectScipyBackend(optimize_func='cg',
16 methodparam='penalty', options={'maxiter',.......)
17 optimi       = Optimizer(s, meritfunctionrms, backend=opt_backend,...)
18 opt_backend.update_PSB(optimi)
19 res          = optimi.run()
```

Nachfolgend soll auf die jeweiligen Zeilen kurz eingegangen werden:

2: Erstellen eines Objektes, welches das Einlesen der Parameter aus den Textfiles erledigt. Das Einlesen der Daten erfolgt beim Konstruieren des Objekts automatisch.

3: Erstellen eines Objektes, welches ein optisches System repräsentiert. Das Objekt enthält von der Objekt/Bildfläche abgesehen noch keine Flächen.

4: Aus den eingelesenen Daten müssen Koordinatensysteme für jedes Element erstellt werden. Wie in Kapitel 2.2.4 erwähnt, werden die Koordinatensysteme für die Zuordnung zum optischen System benötigt.

5: Erstellen eines Objektes, welches ein optisches Element repräsentiert. Zu Beginn enthält es neben der Information, welches Referenzkoordinatensystem verwendet werden soll keine zusätzlichen Informationen.

6: Mit dieser Funktion werden die bei Punkt 2: geladenen Daten der Oberfläche und Blenden in Objekte der Klasse `Surface` bzw. `BaseAperture` umgewandelt und zurückgegeben.

7: Die bereits geladenen Informationen bzgl. der Materialien werden in vom Raytracer interpretierbare Objekte umgewandelt. Dabei wird auf die bereits genannte Datenbank zugegriffen. Es wird automatisch überprüft, welches Material zu welcher Oberfläche gehört, und dieses dieser zugeordnet.

8: Das so erzeugte Element, welches jetzt alle Informationen zu den Oberflächen und Materialien enthält, wird hier dem optischen System zugewiesen.

9: Aus den bisher bekannten Informationen muss eine Sequenz von Oberflächen (sysseq) in einer bestimmten Art und Weise erzeugt werden, damit der Raytracer den Weg der Strahlen durch die Elemente berechnen kann.

10: Erstellen eines bereits vorhandenen pyrate Objekts. Die Klasse wird vom Raytracer benötigt, und erhält als Übergabeparameter die zuvor erzeugte Sequenz aus Oberflächen.

11: Hier wird ein dictionary erzeugt, welches die Informationen über die Strahlen bzw. Bündel repräsentiert. Dies ist nötig, damit später Objekte erstellt werden können, die die Bündel für den Raytracer interpretierbar machen.

12: Mit der Funktion `get_bundle_merit` wird zum einen die Zielfunktion als Funktion zurückgegeben, zum anderen werden die Bündel erzeugt, die im Punkt zuvor definiert wurden.

14: In diesem Schritt werden die Variablen, welche zuvor in den Textfiles auf variabel gesetzt wurden auch im Programmcode so eingerichtet, dass diese vom Raytracer als variabel erkannt werden. Darüber hinaus werden direkt die in der `surface_input` Datei definierten Grenzen abgespeichert und angewendet.

15: Hier wird das zuvor erläuterte Backend erstellt. Details zum Backend sind in Abschnitt 2.4.2 zu finden.

18: Erzeugen eines Optimizer Objekts. Details zum Optimizer sind in Abschnitt 2.4.3 erläutert.

21: Updaten der Constraints. Dies muss aufgrund des Aufbaus des Codes nach dem erstellen des Optimizers geschehen. Auf Details soll hier verzichtet werden.

22: Starten des Optimiervorgangs.

Im nachfolgenden Kapitel wird noch genauer auf relevante Klassen oder Funktionen eingegangen.

2.4 Anbindung des Raytracers an die Optimierung

Am Raytracer selber, sprich an der Berechnung der Strahlen durch das optische System wurden keine Anpassungen vorgenommen. Jedoch musste die Anbindung des Raytracers an die Optimierungsverfahren angepasst werden.

Das zentrale Objekt, welches für die Optimierung benötigt wird gehört der Klasse `Optimizer` an. Dem Konstruktor des `Optimizer` Objekts muss unter anderem ein Objekt der Klassen `OpticalSystem`, `Backend` sowie eine Funktion `meritfunctionrms` übergeben werden. Diese Klassen sollen in diesem Abschnitt genauer erläutert werden, jedoch nur die elementaren und wichtigen Bausteine. Die Klassen sind wesentlich komplexer als hier dargestellt.

2.4.1 Funktion `meritfunctionrms` und `MeritFunctionWrapper`

Um die Anbindung des Raytracers abzuändern, wurde die Datei `aux_merit_bundle.py` erstellt. Innerhalb dieser befindet sich unter anderem die Funktion `meritfunctionrms` welche wiederum den eigentlichen Raytracer anbindet. `meritfunctionrms` bekommt ein Objekt der Klasse `OpticalSystem`, welches den aktuellen Zustand eines optischen Systems beschreibt. Für dieses wird dann mit den zuvor definierten Bündel der Verlauf der Strahlen berechnet, und zu einem Wert konvertiert.

Das optische System ändert sich nach jeder Iteration des Optimierers. Um die neuen Werte auf das zu berechnende optische System anzupassen, wird ein Wrapper verwendet. Dieser hat zur Aufgabe die neuen Werte der zu optimierenden Variablen auf das optische System anzuwenden, und dieses System der Zielfunktion zu übergeben. Als Rückgabewert nach dem Aufruf des Wrappers erhält man somit den Wert der Zielfunktion für das gewünschte optische System. Der Wrapper selbst ist ein Teil der `Optimizer` Klasse.

2.4.2 Klasse `ProjectScipyBackend`

Für die Anbindung der Optimierungsverfahren an den Raytracer wurde ein neues Backend programmiert. Dieses wird `ProjectScipyBackend` genannt, und befindet sich in einer gleichnamigen Datei. Innerhalb des Backends werden die verschiedenen implementierten Verfahren aufgerufen. Die `run` Funktion ist zentraler Bestandteil des Backends. Diese Funktion hat zur Aufgabe die eigentliche Optimierung durchzuführen, wobei verschiedene Verfahren angewendet werden können. Implementiert wurde *penalty*, *penalty-lagrange* und *logarithmic barrier* Verfahren. Die Theorie hinter den Begriffen wird später genauer erläutert.

Die eigentliche Optimierung wird durch die `scipy.optimize.minimize` Funktion durchgeführt. Nachfolgend ist der Aufbau dieser Funktion für die penalty Variante dargestellt.

```
res = minimize(penalty_func,x0=xk,args=(), method=self.optimize_func,
        jac=grad_total, options=self.options, **self.kwargs)
```

Dieser Funktion wird der in diesem Fall auf das penalty Verfahren angepasste Wrapper welcher die meritfunction enthält übergeben (`penalty_func`). `self.optimize_func` stellt dabei die eigenen Programmierten Verfahren dar. Alternativ könnten hier auch Verfahren angegeben werden, die vom python optimize Paket standardmäßig unterstützt werden.

2.4.3 Klasse `Optimizer`

Wie bereits kurz erwähnt, führt die Optimizer Klasse alle Objekte und Funktionen zusammen. Wie in Listing 2.1 zu sehen ist, wird diese Klasse direkt in der Hauptdatei verwendet und benötigt unter anderem das Backend und die Zielfunktion. `Optimizer` besitzt eine `run` Funktion, die letztendlich die `run` Funktion des Backends ausführt. Rückgabewert ist der Wert der meritfunction nach der Optimierung.

3 Optimierungsaufgabe

In diesem Kapitel werden die in Python implementierten Funktionen abstrahiert, um sowohl die restringierte als auch die unrestringierte Optimierungsaufgabe zu definieren. Die Definitionen und Notationen wurden so gewählt, dass sie nah an der Implementierung bleiben.

3.1 Optisches System

Gegeben sei ein optisches System mit bestimmten Eigenschaften $S = \{$Linsenanzahl, Linsenpositionen, Krümmungen, Blendenposition,...$\}$. Die Menge S repräsentiert das optische System und teilt sich auf in optimierbare und nicht optimierbare Parameter, wobei erstere im Folgenden mit $p_i, i = 1, \ldots, N$ bezeichnet seien. Weiter sei $\mathcal{S} = \{S : S$ Menge der Eigenschaften eines optischen Systems$\}$ die Menge der optischen Systeme. Die optimierbaren Variablen mit den dazugehörigen Grenzen sind:

- $p_1 \in [a_1, b_1] =: I_1$
- $\vdots$
- $p_N \in [a_N, b_N] =: I_N$

mit $a_i, b_i \in \mathbb{R}$ und $a_i < b_i, \forall i = 1, ..., N$. Die Nebenbedingungen fassen wir zusammen zu $p \leq b$ und $p \geq a$. Wir betrachten hier also nur sogenannte Box-Constraints und damit definieren wir den konvexen Parameterraum $\mathcal{F}$

$$p \in I_1 \times \cdots \times I_N = \prod_{i=1}^{N} I_i =: \mathcal{F}$$

3.2 Raytracer

Der implementierte Raytracer (`seqtrace`) setzt sich aus verschiedenen verketteten Funktionen und Methoden von Objekten zusammen. Um Lichtstrahlen durch ein optisches System verfolgen zu können werden nicht nur die Größen S des Systems benötigt, sondern auch die Menge der Eigenschaften $B_j = \{n_j, r_j, \lambda_j, \phi_j, a_j\}$ der Strahlenbündel $\mathcal{B} = \{B_j : j = 1, \ldots, M\}$, wobei

- $n \in \mathbb{N}$ die Anzahl der Strahlen eines Bündels,
- $(x, y, z) = r \in \mathbb{R}^3$ die Startposition des Bündels,
- $\lambda \in \mathbb{R}^+$ die Wellenlänge,
- $\phi \in (0, \pi) \times (0, \pi) \times [0, 2\pi]$ der Winkel um die x, y, z-Achsen und
- $a \in \mathbb{R}^+$ der Radius des Bündels im Falle paralleler Strahlen

ist. Ganz Allgemein ist der Raytracer also eine Funktion in $S \in \mathcal{S}$ und in $B \in \mathcal{B}$. Der Rückgabewert des Raytracers sind alle Durchstoßpunkte auf allen Flächen gespeichert in einem mehrdimensionalen Array. Für die Definition der Optimierungsaufgabe sind nur die Durchstoßpunkte auf der Bildebene $C \subset \mathbb{R}^3$ wichtig und daher definieren wir die Raytrace-Funktion für einen Strahl ($n = 1$) wie folgt:

$$\begin{aligned} rt : \mathcal{S} \times \mathcal{B} &\to C \\ (S, B) &\mapsto \begin{pmatrix} x & y & z \end{pmatrix} \end{aligned} \tag{3.1}$$

Betrachten wir jedoch die Implementierung dann ist zum Zeitpunkt des Aufrufs des Raytracers das optische System S keine Variable. Des Weiteren werden während der Optimierung nur die Größen in p verändert und so ergibt sich die Raytrace-Funktion zu

$$\begin{aligned} rt_p : \mathcal{B} &\to C \\ B &\mapsto \begin{pmatrix} x & y & z \end{pmatrix} \end{aligned} \tag{3.2}$$

Der Index p deutet die Abhängigkeit des Raytracers von dem Parametervektor p während der Optimierung an. Oftmals ist $n > 1$, es wird also ein ganzes Strahlenbündel berechnet. Dadurch bilden wir auf den Raum $\mathbb{R}^{n\times 3}$ bzw. in unserem Fall (alle z-Werte der Bildebene sind gleich) auf $\mathbb{R}^{n\times 2}$ ab.

$$\begin{aligned} rt_p : \mathcal{B} &\to \mathbb{R}^{n\times 2} \\ B &\mapsto \begin{pmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix} \end{aligned} \tag{3.3}$$

Die Zeilen in der Matrix sind also die (x, y)-Koordinaten der Durchstoßpunkte der n Strahlen auf der Bildebene C.

3.3 Meritfunktion

Für die Optimierung eines optischen Systems können wir nun die Zielfunktion, die sogenannte Meritfunktion definieren. Dafür muss die Güte eines optischen Systems quantifiziert werden, wobei es mehrere Möglichkeiten gibt ein System zu bewerten. Im Allgemeinen werden die Abstände zwischen den Durchstoßpunkten auf der Bildebene und einem Referenzwert berechnet und aufsummiert. Dabei hat jedes Strahlenbündel einen eigenen Referenzwert. In diesem Projekt wurden zwei Arten von Referenzwerte gewählt. Zum einen der Durchschnitt der Durchstoßpunkte eines Bündels für die Wellenlängen von grünem λ_1, blauem λ_2 und rotem Licht λ_3. Sei nun $B_{ij} = \{n, r_j, \lambda_i, \phi_j, a_j\}$ ein Strahlenbündel (n ist für alle Bündel gleich), $\mathcal{B} = \{B_{ij} : i = 1, 2, 3 \text{ und } j = 1, \ldots, M\}$ und $e_k \in \mathbb{R}^n, k = 1, \ldots, n$ der k-te Einheitsvektor. Dann ist der Referenzwert

$$\bar{x}_p^{(1)}(B_{ij}) := \frac{1}{3n} \sum_{i=1}^{3} \sum_{q=1}^{n} (rt_p(B_{ij}))^T e_q \tag{3.4}$$

Dieser Referenzwert ändert sich also nur wenn sich der Index j verändert. Mit dieser Definition wird die Dispersion, also die Abhängigkeit der Lichtgeschwindigkeit von der Wellenlänge

im Medium in die Optimierung mit einbezogen. Zum anderen wurde als Referenzwert der Durchstoßpunkt des Hauptstrahls für eine Referenzwellenlänge, nämlich die des grünen Lichts gewählt. Durch die Implementierung von `pyrate` ist der Durchstoßpunkt des Hauptstrahls immer die erste Zeile in $rt(B)$ und es wurde im Projekt die Konvention getroffen, dass λ_1 immer die Wellenlänge des grünen Lichts ist.

$$\bar{x}_p^{(2)}(B_{ij}) := (rt_p(B_{1j}))^T e_1 \tag{3.5}$$

Die unterschiedlichen Referenzwerte führen auf unterschiedliche Definitionen der Meritfunktion. Des Weiteren gibt es verschiedene Metriken für die Abstandsmessung zwischen Durchstoßpunkte und Referenzwerte. Wir betrachten hier die von den p-Normen induzierten Metriken für $p = 1$ und $p = 2$, wobei der euklidische Abstand quadriert wird.

$$d_1(x, y) = \|x - y\|_1, \quad (d_2(x, y))^2 = \|x - y\|_2^2 \tag{3.6}$$

Die implementierten Meritfunktionen (`meritfunctionrms` innerhalb von `MeritFunctionWrapper`) sind, wie der Raytracer Verkettungen von Funktionen und Methoden. Alle Implementierungen bilden jedoch die optimierbaren Parameter p auf die entsprechende Bewertungsgröße ab.

$$mf(p) : \mathcal{F} \to \mathbb{R}_0^+, \quad p \mapsto mf(p) \tag{3.7}$$

$$mf_l^k(p) := \sum_{j=1}^{M} \sum_{i=1}^{3} \sum_{q=1}^{n} \left[d_l \left((rt_p(B_{ij}))^T e_q, \bar{x}_p^{(k)}(B_{ij}) \right) \right]^l \tag{3.8}$$

wobei $(k, l) \in \{1, 2\}^2$ ist.
Die Meritfunktion 3.8 lässt sich nun beliebig erweitern um verschiedene Nebenbedingungen bereits in der Zielfunktion zu beachten. Dies kann mit Straftermen realisiert werden, d.h. wird eine Nebenbedingung nicht erfüllt, wird zur Meritfunktion ein Wert addiert um die Verletzung der Nebenbedingung zu bestrafen. Idealerweise wird ein Strafterm so gewählt, dass der Optimierungsalgorithmus forciert wird ein Optimum zu finden, das die gewünschte Nebenbedingung ohne Reststrafe erfüllt. Im Projekt wurden mehrere Strafterme in die Meritfunktion aufgenommen.
Der erste Strafterm soll sicherstellen, dass alle Strahlen die in das System hinein geschickt werden auf der Bildebene ankommen. Diese Nebenbedingung ist essentiell, da die Optimierungsalgorithmen sonst das optische System so modifizieren könnten, dass kein Strahl auf der Bildebene ankommt, womit ein Meritwert von Null erzielt werden würde. Dafür wurde folgende Funktion gewählt

$$pen_1(n_j^*) = e^{\frac{3n}{n_j^*+0.1}} - e^{\frac{3n}{3n+0.1}}, \tag{3.9}$$

wobei n_j^* die Anzahl der Strahlen von drei Bündeln $B_{ij}, i = 1, 2, 3$, die auf der Bildebene ankommen ist. Ist also $n_j^* = 3n$, dann ist $pen(n_j^*) = 0$.
Der zweite Term der in die Meritfunktion aufgenommen wurde bestraft negative Dicken der Linsen. Physikalisch sind negative Dicken unmöglich, für den Raytracer spielt das jedoch keine Rolle. Daher muss diese Nebenbedingung wieder über einen Strafterm sichergestellt werden.

$$pen_2(B_{ij}) = (overlap(B_{ij}) + 1)^6 - 1 \tag{3.10}$$

$$overlap(B_{ij}) = \sum_{\text{Linse } l} \sum_{\text{Strahl } k} \max\{0, -(z_{l+1,k}^{ij} - z_{l,k}^{ij})\} \tag{3.11}$$

Die Funktion $overlap(B_{ij})$ berechnet also für alle Linsen und alle Strahlen die Differenz der z-Koordinaten der Durchstoßpunkte aufeinander folgender Linsen und summiert diese auf, falls sie negativ ist.
Der letzte Term soll die Verzeichnung geringer als ein Prozent halten. Sei y_{HS} die y-Koordinate des Hauptstrahls für grünes Licht auf der Bildebene und SBH die Sollbildhöhe, dann ist der Strafterm

$$pen_3(B_{1j}) = 10^3(|y_{HS}^{(j)} - SBH| - 0.01 \cdot |SBH|). \tag{3.12}$$

Hierzu sei angemerkt, dass dieser Strafterm nur für $k = 2$ implementiert wurde. Die unterschiedlichen Strafterme können natürlich auch anders gewählt werden, je nachdem wie stark oder schwach eine Nebenbedingung (NB) durchgesetzt werden soll. Wird eine NB gar nicht erfüllt, sollte der jeweilige Strafterm nach oben beschränkt sein und die Meritfunktion an dieser Stelle eine Steigung ungleich Null besitzen. Dadurch wird sichergestellt, dass wenn die k-te Iterierte x_k eines gradientenbasierten Optimierungsalgorithmus aufgrund eines sehr großen Gradienten im vorherigen Schritt eine NB gar nicht erfüllt, die Meritfunktion keine `NaN`'s erzeugt und durch die Steigung wieder in Richtung $\mathcal{F}$ gezwungen wird.
Um einen Eindruck der Meritfunktion 3.8 zu bekommen wurde diese für ein Linsensystem mit acht Linsen, $l = 2$, $k = 1$ sowie keinem penalty-Term in Abhängigkeit zweier Krümmungen geplottet. In der Abbildung 3.1 ist ein sehr flacher Bereich (banana shaped valley) zu erkennen.

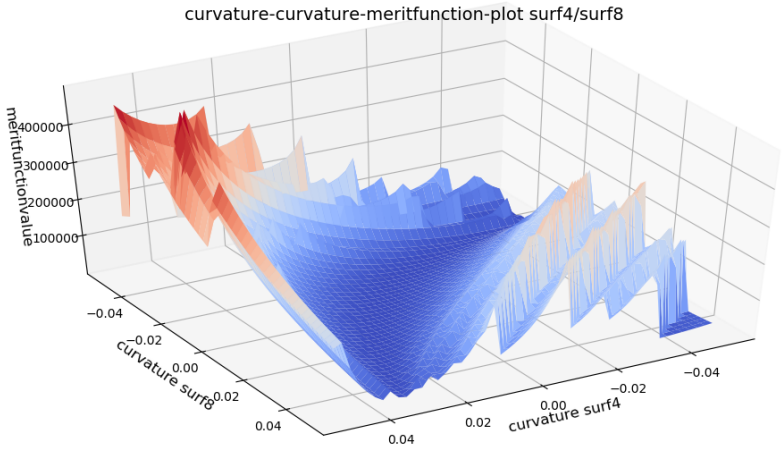

Abbildung 3.1: Meritfunktion über zwei Krümmungen geplottet

Sehr auffällig sind auch die Unstetigkeiten in den äußeren Bereichen. Kleine Änderungen in der Krümmung einer Linse reichen dabei aus, dass weniger Strahlen auf der Bildebene ankommen und somit kommt es zu Sprüngen im Funktionswert. Wie die penalty-Terme und die verschiedene Werte für k und l die Meritfunktion beeinflussen wurde nicht untersucht. Des Weiteren sei an dieser Stelle angemerkt, dass die Anzahl der variablen Parameter oftmals größer ist als zwei. Die Meritfunktion ist entgegen der Definition in 3.7 auch auf ganz $\mathbb{R}^N$ definiert, jedoch nimmt sie nur auf einer kleinen Teilmenge davon Werte ungleich Null an. Wie bereits angemerkt kann das optische System und insbesondere die Linsenkrümmungen so verändert werden, dass kein Strahl mehr die Bildebene trifft. Dies ist für betragsmäßig sehr große Krümmungen der Fall.

3.4 Restringierte Optimierungsaufgabe

Nun ist die Zielfunktion und der Parameterraum definiert und wir suchen für ein Paar $(k, l) \in \{1, 2\}^2$ ein $p^* \in \mathcal{F}$ mit

$$p^* = \arg\min_{p \in \mathcal{F}} mf_l^k(p)$$

Das allgemeine Minimierungsproblem lautet

$$\min_{p \in \mathbb{R}^N} mf_l^k(p) \tag{3.13a}$$

$$\text{unter } a \leq p \leq b \tag{3.13b}$$

Die Ungleichungs-Nebenbedingungen können umformuliert werden, sodass 3.13b zu

$$a - p \leq 0, \quad p - b \leq 0$$

wird. Damit definieren wir dann

$$g(p) := \begin{pmatrix} p - b \\ a - p \end{pmatrix} \quad \text{und} \quad h(p) := \max\{0, g(p)\}$$

Das Minimierungsproblem mit Gleichheits-Nebenbedingungen lautet dann

$$\min_{p \in \mathbb{R}^N} mf_l^k(p) \tag{3.14a}$$

$$\text{unter } h(p) = 0 \tag{3.14b}$$

3.5 Unrestringierte Optimierungsaufgabe

Um eine restringierte als nichtrestringierte Optimierungsaufgabe aufzufassen, existieren verschiedene Methoden. Im Rahmen des Projekts wurden dabei zwei Ansätze näher betrachtet, welche nachfolgend kurz vorgestellt werden. Für eine ausführliche Diskussion dieser Methoden siehe [10] (S. 497 ff).

3.5.1 Penalty Methode[1]

Eine Möglichkeit ist es, einen *Penalty Paramter* $\tau > 0$ einzuführen, welcher die Verletzung der Nebenbedingungen bestraft. Wir definieren die *Penalty Funktion* $\mathcal{L}_\tau : \mathbb{R}^N \to \mathbb{R}$

$$\mathcal{L}_\tau(p) = mf(p) + \frac{1}{2}\tau \left\| h(p) \right\|^2 .$$

Betrachten wir nun das neue Minimierungsproblem

$$\min_{p \in \mathbb{R}^N} \mathcal{L}_{\tau_k}(p) \tag{3.15}$$

für eine Sequenz mit ansteigendem *Penalty Parameter*

$$\tau_k \to \infty \quad \text{für} \quad k \to \infty,$$

[1] nach [12], S. 39

so strebt die *Penalty Funktion* gegen unendlich, es sei denn die Nebenbedingungen sind erfüllt. Die Hoffnung ist nun, dass für ein bestimmte Sequenz an τ_k strebt die *Penalty Funktion* gegen das gesuchte Minimum der Funktion $mf(p)$.

Anzumerken ist jedoch, dass für Ansteigende τ die Kondition der unrestringierten Optimierungsaufgabe zunehmend schlechter wird. Aus diesem Grund darf τ nicht zu schnell ansteigen. Eine Möglichkeit dieses Problem zu umgehen bietet die Penalty Lagrange Methode.

3.5.2 Penalty Lagrange Methode

Im Folgenden wird auf die Penalty-Lagrange-Methode eingegangen. Die Lagrange-Funktion ist in unserem Setting definiert als

$$\mathcal{L}(p, \lambda) := mf(p) + \lambda^T h(p),$$

wobei

$$h(p) := max\{0, g(p)\} \quad \text{und} \quad g(p) := \begin{pmatrix} p - b \\ a - p \end{pmatrix}$$

Damit lässt sich die Penalty-Lagrange-Funktion mit quadratischem Penalty-Term aufstellen.

$$\begin{aligned} \mathcal{G}_{\tau_k}(p, \lambda_k) : &= \mathcal{L}(x, \lambda_k) + \frac{1}{2}\tau_k ||h(p)||_2^2 \\ &= mf(p) + \lambda_k^T h(p) + \frac{1}{2}\tau_k ||h(p)||_2^2 \end{aligned}$$

Mit dieser Funktion lässt sich das ursprüngliche restringierte Optimierungsproblem in eine Folge von unrestringierten Optimierungsproblemen transformieren.

4 Algorithmen

4.1 Gradientenberechnung

Für ableitungsbasierte Optimierungsalgorithmen muss der Gradient der Meritfunktion und der Penalty-Lagrange-Funktion berechnet werden. Dafür gibt es mehrere Ansätze auf die in diesem Abschnitt eingegangen werden soll.

4.1.1 Finite Differenzen

Für die Meritfunktion 3.8 gibt es aufgrund des Raytracers keine explizite Darstellung, daher liegt es nahe den Gradienten durch finite Differenzen zu approximieren. Im Projekt wurden zentrale Differenzen genutzt um die partiellen Ableitungen anzunähern.

$$\frac{\partial m f_l^k(p)}{\partial p_i} \approx \frac{m f_l^k(p + he_i) - m f_l^k(p - he_i)}{2h} \quad \forall i \in \{1, \ldots, N\}, \quad h > 0 \tag{4.1}$$

Ob und welche Auswirkungen die verschiedenen Meritfunktionen auf die Gradientenberechnung haben wurde nicht untersucht. Um zu verdeutlichen welchen Einfluss eine Änderung der variablen Größen eines Linsensystems auf die Meritfunktion hat, soll an dieser Stelle der Gradient ausgewertet werden. Seien die variablen Größen die Positionen und die Krümmungen der Linsen, dann ist Gradient an einer Stelle $p \in \mathcal{F} \subset \mathbb{R}^4$

$$(\nabla m f_2^1)(p) \approx \begin{pmatrix} 730.7 \\ 119.6 \\ -5845766.9 \\ 7366252.0 \end{pmatrix}, \tag{4.2}$$

wobei die ersten zwei Einträge die partiellen Ableitungen nach den Linsenpositionen und die letzten zwei Einträge die partiellen Ableitungen nach den Krümmungen sind. Die Ableitungen können sich also um eine Größenordnung von 10^4 unterscheiden und Krümmungsänderungen tragen mehr zur Steigung bei als Positionsänderungen. Dieser Umstand stellt für die Algorithmen eine Herausforderung dar, da die Schrittweite je nach Algorithmus durch die Krümmungsänderungen bestimmt wird und dadurch die Änderungen in der Position der Linsen sehr kein werden können.

4.1.2 Automatisches Differenzieren

In diesem Kapitel soll nun eine kurze Einführung in das Automatische oder Algorithmische Differenzieren gegeben werden. Dieses Kapitel orientiert sich an [4, 1, 8, 6]. Darüber hinaus werden zwei AD-Tools betrachtet, und es wird aufgezeigt, wie mit diesen der Gradient einer Zielfunktion berechnet werden kann. Am Ende des Kapitels wird noch darauf eingegangen, ob es möglich war, den Gradienten der von pyrate verwendeten Zielfunktion mit diesen Tools zu berechnen.

Wie in Abschnitt 4.1.1 erwähnt, benötigen einige Optimierungsverfahren den Gradienten einer Funktion, um einen Schritt in die entsprechende Richtung zu gehen.

Es gibt verschiedene Möglichkeiten den Gradienten zu berechnen:

1. Händische Berechnung
2. Symbolische Ableitung z.B. mit maple
3. Differenzenquotient numerisch berechnen
4. Automatische Differentiation

Bisher wurde der Gradient wie in Gl. 4.1 angegeben über den zentralen Differenzenquotienten im Programm numerisch berechnet. Dies geschieht durch die Funktion `grad(func, x, h)`, welche den merritfunction Wrapper `func` sowie die Werte der zu optimierenden Variablen `x` und eine Schrittweite `h` übergeben bekommt. Aufgrund der Verwendung von Penalty/Penalty-Lagrange Verfahren, musste dieser noch etwas angepasst werden, dies soll in diesem Kapitel jedoch keine Rolle spielen. Details dazu sind in Kapitel 4.1.5 zu finden. Nachteilig bei der numerischen Berechnung des zentralen Differenzenquotienten ist, dass unklar ist, welche Schrittweiten gewählt werden sollten, um akzeptable Fehler zu erhalten. Außerdem kann es zu numerischen Instabilitäten kommen.

Eine händische Ableitung ist für ein komplexeres Problem offensichtlich nicht geeignet, und symbolische Ableitungen ergeben häufig extrem lange Ergebnisse und sind äußerst ineffizient was die Berechnungszeit angeht. [6]

Die Grundidee des automatischen Differenzierens ist, dass jede noch so komplizierte Funktion in einem Programm letztendlich auf eine Sequenz von elementaren Rechenoperationen und mathematischen Funktionen wie z.B. $sin(x)$, $cos(x)$ etc. zurückgeführt werden kann. Durch wiederholtes anwenden der Kettenregel kann somit die Ableitung einer Funktion exakt bestimmt werden.

Automatisches Differenzieren lässt sich grundsätzlich mit zwei verschiedenen Vorgehensweisen umsetzen:

- Code Transformation
- Operatoren überladen

Bei der Code Transformation wird ein neuer Code aus bestehendem erstellt. Da im Rahmen dieser Arbeit dieses Verfahren nicht betrachtet wurde, soll hier auf einschlägige Literatur z.B. [8] und insb. [6] verwiesen werden. Bei der Operatorüberladung werden elementare Rechenoperationen für neue Klassen neu definiert. Welche Klassen dies sind, wird später in Kapitel 4.1.3 gezeigt.

Es gibt grundsätzlich zwei verschiedene Kategorien, wie Automatische Differentiation konkret umgesetzt werden kann.

Vorwärts Methode: Bei der Vorwärts Methode werden Funktionswerte und deren Ableitung in einem Schritt berechnet. Sei als Beispiel der Gradient $\nabla f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n}\right] \in \mathbb{R}^{1\times n}$ einer Funktion $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ gesucht.[1] Es wird nun zwischen rationalen Kompositionen und Bibliothek Funktionen unterschieden. Seien zwei differenzierbare Funktionen $a : D \subseteq \mathbb{R}^n \to \mathbb{R}, b : D \subseteq \mathbb{R}^n \to \mathbb{R}$ gegeben. Die Ableitungen der rationalen Kompositionen der beiden Funktionen sind in Tabelle 4.1 dargestellt.

[1] Begrifflichkeiten und Beispiele sind [4] und [1] entnommen.

Tabelle 4.1: Rationale Kompositionen und deren Ableitung

Typ ω	**r**	**r'**
$+$	$a+b$	$a'+b'$
-	$a-b$	$a'-b'$
/	a/b	$b\cdot a'+a\cdot b'$
$\cdot$	$a\cdot b$	$(a'-r\cdot b')/b$

Diese Operationen werden mit *RAT* bezeichnet. (siehe Abb. 4.1)

Abbildung 4.1: *RAT* Operation, [4]

Zusätzlich zu den RAT Funktionen werden noch Bibliothek-Funktionen λ benötigt. Es sei $\lambda : E \subseteq \mathbb{R} \to \mathbb{R}$ und $a : D \subseteq \mathbb{R}^n \to \mathbb{R}$. Eine Funktion r mit $r : D \subseteq \mathbb{R}^n \to \mathbb{R}, r(x) := \lambda(a(x))$ sei gegeben. Die Ableitung ist dann $r'(x) = \lambda'(a(x) \cdot a'(x)$. Die Berechnung der Ableitung und des Funktionswertes kann dann wie in Abb. 4.2 dargestellt berechnet werden.

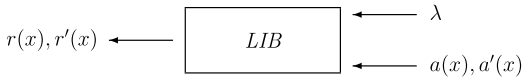

Abbildung 4.2: *LIB* Operation, [4]

Eine solche LIB Funktion könnte z.B. $sin(x), cos(x), exp(x), \ldots$ sein.

Das Vorgehen soll kurz anhand der Funktion $f : D \subseteq \mathbb{R}^3 \to \mathbb{R}, f(x) = \frac{(x_1-7)\cdot sin(x_1+x_2)}{x_3}, x = (x_1, x_2, x_3) \in \mathbb{R}, x_3 \neq 0$ gezeigt werden. Die Ableitung soll an der Stelle $x = (-13, 8, 0.3)$ berechnet werden. In Tabelle 4.2 ist das Vorgehen dargestellt. Benötigt werden dazu nur die Rechenoperationen aus 4.1 und die Information, wie die Ableitung der LIB Funktionen λ ausschaut. $(r'(x) = \lambda'(a(x)\cdot a'(x)))$ Die Berechnung des Funktionswerts und der Ableitungen erfolgt Schritt für Schritt, wie in Tabelle 4.2 zu sehen ist.

Tabelle 4.2: Ablauf Vorwärtsdifferenzieren

Fkt-Wert f_i	$d_i^1 := {}^{\partial f}/_{\partial x_1}$	$d_i^2 := {}^{\partial f}/_{\partial x_2}$	$d_i^3 := {}^{\partial f}/_{\partial x_3}$
$f_1 = x_1 = -13$	1	0	0
$f_2 = x_2 = 8$	0	1	0
$f_3 = x_3 = 0.3$	0	0	1
$f_4 = 7$	0	0	0
$f_5 = f_1 - f_4 = -20$	$d_1^1 - d_4^1 = 1$	$d_1^2 - d_4^2 = 0$	$d_1^3 - d_4^3 = 0$
$f_6 = f_1 + f_2 = -5$	$d_1^1 + d_2^1 = 1$	$d_1^2 + d_2^2 = 1$	$d_1^3 + d_2^3 = 0$
$f_7 = sin(f_6) = 0.96$	$cos(f_6)\cdot d_6^1 = 0.28$	$cos(f_6)\cdot d_6^2 = 0.28$	$cos(f_6)\cdot d_6^3 = 0$
$f_8 = f_5 \cdot f_7 = -19.18$	$d_5^1\cdot f_7 + d_7^1\cdot f_5 = -4.71$	$d_5^2\cdot f_7 + d_7^2\cdot f_5 = -5.67$	$d_5^3\cdot f_7 + d_7^3\cdot f_5 = 0$
$f_9 = \frac{f_8}{f_3} = -63.93$	$d_8^1 - \frac{f_8}{f_3}\cdot d_3^1 = -15.7$	$d_8^2 - \frac{f_8}{f_3}\cdot d_3^2 = -18.9$	$d_8^3 - \frac{f_8}{f_3}\cdot d_3^3 = 213$

Somit ergibt sich für den Funktionswert -63.93 und für den Gradient $(-15.7, -18.9, 213)$. Alle vier Werte entsprechen dem exakten Wert, den man auch erhält, wenn man die Funktion symbolisch ableitet, somit ist der Fehler 0.

Rückwärts Methode: Bei der Vorwärts Methode erscheinen die Ableitungen der „simplen" Bauteile zuerst, und die der zusammengesetzten ganzen Funktion zum Schluss. Bei der Rückwärtsmethode tauchen die Ableitungen in umgekehrter Reihenfolge auf. Ein weiterer Unterschied ist, dass bei der Rückwärts Methode zunächst alle Funktionswerte berechnet werden müssen, bevor die erste Ableitung bestimmt werden kann. Bei der Vorwärts Methode wurde in jedem Schritt der entsprechende Funktionswert und Ableitung berechnet.

Die Rückwärts Methode ist wesentlich komplexer zu erklären als die Vorwärts Methode. Hier soll deshalb nur eine kurze Einführung gegeben werden. Für Details soll auf die vorhandene Literatur z.B. [4, 6, 1, 8] verwiesen werden.

Angenommen man möchte die Ableitung einer Funktion $f : D \subseteq \mathbb{R} \to \mathbb{R}$ berechnen. Dabei sei $y_k = x_k, k = 1 \dots n$ und $y_k = \Theta(y_1, \dots, y_{k-1}), k = n+1, \dots, N$, wobei $\Theta_k : \mathbb{R}^{k-1} \to \mathbb{R}$ gelte. N sei die Anzahl an Teiloperationen die für die Berechnung von $f(x)$ benötigt werden. Für $k = n+1, \dots, N$ sei $E_k : \mathbb{R}^{k-1} \to \mathbb{R}^k, E_k(z) := \begin{bmatrix} z \\ \Theta(z) \end{bmatrix}$ sowie $L : \mathbb{R}^{n+1} \to \mathbb{R}$ welches den letzten Eintrag aus z darstellen soll. Damit lässt sich die Funktion durch

$$f(x) = L(E_N(E_{N-1}(\dots(E_{n+1}(x))))) \tag{4.3}$$

darstellen. Mit Hilfe der Kettenregel, lässt sich (4.3) ableiten. Es gilt dann

$$\nabla f(x) = L'(u_N) \cdot E'_N(u_{N-1}) \cdot \dots \cdot E'_{n+1}(u_n) \tag{4.4}$$

wobei $u_n := x$ und $u_k := E_k(E_{k-1}(\dots(E_{n+1})))$ gelten soll. Man kann nun (4.4) von rechts oder von links berechnen, was der Vorwärts oder Rückwärts Methode entspricht. Sei nun

$$p_{N+1} := L'(z_N) \tag{4.5}$$

und

$$p_k := p_{k+1} \cdot E'(z_{k-1}) \tag{4.6}$$

dann gilt

$$z_k = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} \tag{4.7}$$

für $k = n, \dots, N$. Hier erkennt man nun, dass man alle Funktionswerte zuerst berechnen muss, da man z_N als erstes benötigt.

Man kann (4.6) aufgrund der Form von $E'_k(z_{k-1})$ wiefolgt ausdrücken:

$$[p_{k,1} \dots p_{k,k-1}] = [p_{k+1,1} \cdots p_{k+1,k-1}] + p_{k+1,k} \cdots \Theta'_k(z_{k-1}) \tag{4.8}$$

Dies ist letztendlich das immer wieder Updaten einer Zeile $U = [U_1, \dots, U_n, \dots, U_N]$. Schlussendlich kann man das Updaten der Zeile U wiefolgt ausdrücken:

$$U_i = U_i + U_k \cdot \Theta'_k(z_{k-1})_i, i = 1, \cdots, k-1 \tag{4.9}$$

mit i als der entsprechenden Spalte von U.

Die Ableitung bzw. der Gradient lässt sich dann durch $\nabla f(x) = p_{n+1} = [U_1, \ldots, U_n]$ darstellen.

Auf ein konkretes Beispiel soll hier verzichtet werden. Ein solches ist z.B. in [4, S. 308] zu finden.

4.1.3 Ad-Tools

Im Rahmen dieser Arbeit wurden die Pakete `ad`, `AuDi` und `ADOL-C` betrachtet. Da letztgenanntes selbst mit viel Aufwand nicht installiert werden konnte, beschränkt sich dieses Kapitel auf die beiden erstgenannten. Nachfolgend soll anhand einer simplen Funktion die Verwendung der Tools gezeigt werden. Aufgabe sei es, von der Funktion

$$f(x) = \frac{sin(1/x)}{exp(cos(1/x))} \tag{4.10}$$

den Gradienten zu berechnen und somit einen ersten Einblick zu geben, wie AD-Tools angewendet werden können.[2]

4.1.3.1 ad

`ad` ist ausschließlich in python geschrieben, und lässt sich nach der Installation über `import ad` in python Programme einpflegen. Es ist ein Programm, welches Operatoren überlädt und hat nur die Rückwärtsmethode implementiert. Nachfolgend soll gezeigt werden, wie mit diesem Programm der Gradient der zuvor genannten Funktion berechnet werden kann. `ad` bietet dazu zwei Möglichkeiten:

- Zum einen bietet das Paket die Klasse `adnumber` an. Die elementaren mathematischen Operationen sind mit dieser Klasse überladen. Darüber hinaus werden noch elementare mathematische Funktionen durch `admath` angeboten. Es ist nicht möglich Funktionen aus dem Standard python paket `math` zu verwenden, da diese nicht mit `adnumber` umgehen können.

  Ein möglicher Programmcode zum Berechnen des Gradienten der oben genannten Funktion ist in Listing 4.1 dargestellt.

[2] Die Funktion wurde [8] entnommen.

Listing 4.1: ad Paket mit adnumber

```
from ad import adnumber
import math
from ad import gh
from ad import admath

def f1_ad(x):
    tmp=adnumber(x)
    fkt=admath.sin(1.0/tmp)/admath.exp(admath.cos(1.0/tmp))
    gradient=fkt.d(tmp)
    return [math.sin(1.0/tmp)/math.exp(math.cos(1.0/tmp)), gradient]

print(f1_ad(0.01)[1])

def f2_ad(x):
    x_=adnumber(x[0],x_)
    y_=adnumber(x[1],y_)
    fkt=admath.sin(1.0/x_)/admath.exp(admath.cos(1.0/y_))
    gradient=fkt.d()
    return [math.sin(1.0/x_)/math.exp(math.cos(1.0/y_)), gradient]

print(f2_ad([5,6]))
```

In der Funktion `f1_ad` wird der Übergabeparameter (kann auch eine Liste etc. sein) zu einer `adnumber` umgewandelt. Darüber hinaus müssen für die Zielfunktion wie bereits erwähnt die Funktionen $sin(x)$ etc. aus dem admath Paket verwendet werden. Eine so aufgebaute Funktion kann mit `fkt.d(wert)` abgeleitet werden. Rückgabe der Funktion ist der Wert des Gradienten an der gewünschten Stelle.

In diesem sehr einfachen Beispiel wurde nur eine Variable verwendet, das Paket funktioniert aber auch mit mehreren, was ebenfalls in Listing 4.1 zu sehen ist.

- Das `ad` Paket bietet auch eine Funktion `gh(meritfunction)` zum Tracken von Variablen an. Dies hat den großen Vorteil, dass im Code selber keine `admath` Deklarationen für die Optimierungsparameter vorgenommen werden müssen.

  Die Funktion kann sehr simpel genützt werden, was in Listing 4.2 dargestellt ist.

Listing 4.2: ad Paket ohne adnumber

```
def func(x):
    return admath.sin(1.0/x[0])/admath.exp(admath.cos(1.0/x[0]))

gradient, hessian = gh(func)
print(gradient(0.01))
```

Die Rückgabe der `gh` Funktion ist also eine Funktion, der man die Werte der Optimierungsvariablen übergeben kann, für die der Gradient anschließend ermittelt wird.

4.1.3.2 AuDi

Als zweites Paket wurde `AuDi` betrachtet. Bestandteil des Pakets ist das für Python optimierte `pyaudi` Paket. Dieses legt besonderen Wert auf Performance, und kann somit auch höhere Ableitungen schnell berechnen. Beim Ausführen des Pakets wird auf einen Threadpool zurückgegriffen und das Programm parallelisiert ausgeführt. Wie auch das `ad` Paket setzt `pyaudi` Automatisches Differenzieren durch Operatorüberladungen um. Jedoch bietet das Paket im Gegensatz zu `ad` nur die Vorwärtsdifferentiation an.

In Listing 4.3 ist die Verwendung des Pakets dargestellt.

Listing 4.3: pyaudi Paket

```
import pyaudi
from pyaudi import gdual_double as gdual
import numpy as np

def f1_pyaudi(x):
    x_=gdual(x,x,1)
    temp=1.0/x_
    return pyaudi.sin(temp)/pyaudi.exp(pyaudi.cos(temp))

grad_func=f1_pyaudi(0.01)
grad=grad_func.get_derivative({'dx': 1})
```

4.1.3.3 Vergleich AD-Tools zu Differenzenquotient

Um den Mehrwert des automatischen Differenzierens darzustellen, wurde ein Programm geschrieben, welches die oben eingeführte Funktion sowohl mit dem zentralen Differenzenquotient wie auch mit den beiden vorgestellten Tools an der Stelle 0.01 differenziert.

Tabelle 4.3: Fehler des Differenzenquotienten über verschiedene Schrittweiten

Schrittweite	2	0.2	0.02	0.002	0.0002	2E-05	2E-06	2E-07	2E-08	2E-10	2E-11	2E-12	2E-14
Fehler	-4723	-4719	-4750	-5063	978	47	0.47	0.0047	4.72E-05	-2.87E-05	5.54E-05	0.0021	-0.0840

Man erkennt in Tabelle 4.3, dass es nicht ohne weiteres möglich ist, die beste Schrittweite für den geringsten Fehler festzulegen und der Fehler wächst in diesem Fall mit kleiner werdender Schrittweite wieder an. Beide vorgestellten Tools weisen einen Fehler von exakt 0 auf, geben also den analytisch korrekten Wert aus.

4.1.4 Anwenden der Ad-Tools auf pyrate

Wie zuvor gezeigt, müsste man bei der Verwendung des `pyaudi` Pakets händisch die zu optimierenden Variablen zu `gdual` casten, sollte man das Paket auf pyrate anwenden wollen. Dies ist mit enormen Aufwand verbunden, weshalb im Rahmen dieses Projektes darauf verzichtet wurde. Das `ad` Paket bietet jedoch die tracking Funkion `gh` an. Es wurde versucht, diese auf den vorhandenen pyrate Code anzuwenden. Dies ist jedoch nicht gelungen. Das `ad` Paket unterstützt keine Objekte der Klasse `np.ndarray`, welche aber in pyrate häufig verwendet werden. Das konkrete Problem soll nachfolgend kurz beschrieben werden. Die `gh` Funktion verfolgt die zu optimierbaren Variablen bis auf die unterste Ebene des Raytracers, um dort die Umwandlung zu `ad` Objekten vorzunehmen. Diese Ebene ist in Abb. 4.3 zu sehen.[3]

```
F = rayDir[2] - curv * (rayDir[0] * r0[0] + rayDir[1] * r0[1] + rayDir[2] * r0[2] * (1+cc))
```

Abbildung 4.3: Letzte Ebene des Raytracers

Dort werden die Variablen wie z.B. *curvature (cc)* verwendet. Alle Variablen in dieser Funktion (auch Konstanten wie z.B. die *1* in Abb. 4.3) werden dahingehend überprüft, ob es sich um eine optimierbare Variable handelt. Bei `r0[2]` ist dies z.B. der Fall. Dementsprechend werden die Elemente in r0 in einen ad Typ umgewandelt, was in Abb. 4.4 zu sehen ist.

Nach der Umwandlung, wird wie in Abb. 4.3 zu sehen ist, eine Multiplikation mit dem `r0` array der Klasse `np.ndarray` ausgeführt. Dafür sind die Operatoren im Paket aber noch nicht

[3]Der Raytracer ist natürlich sehr viel komplexer, der Übersichtlichkeit halber wurde hier nur eine Zeile abgebildet.

```
type autodiff variable
<type 'numpy.ndarray'>
value autodiff variable
[ad(-10.0) ad(-10.0) ad(-10.0) ad(-10.0) ad(-10.0) ad(-10.0) ad(-10.0)
 ad(-10.0) ad(-10.0) ad(-10.0) ad(-10.0) ad(-10.0)]
id autodiff variable
139888897626320
```

Abbildung 4.4: `r0[2]` mit `ad` Objekten

überladen, dementsprechend wird ein entsprechender Fehler angezeigt, welcher in Abb. 4.5 dargestellt ist.

```
NotImplementedError: Automatic differentiation not yet supported for <type 'numpy.ndarray'> objects
```

Abbildung 4.5: Fehlermeldung Operatorüberladung für `np.ndarrays` Objekte

Um dieses Problem zu umgehen, müsste man alle `np.ndarrays` welche optimierbare Variablen enthält z.B. in Listen umschreiben. Darüber hinaus müssten aber auch die Formeln selbst verändert werden, da mit `np.ndarrays` Rechenoperationen möglich sind, die bei Listen nicht funktionieren. Außerdem besitzen die array Objekte in pyhton Methoden, die von Listen nicht unterstützt werden. Ein Umschreiben der arrays war im Rahmen dieses Projektes aufgrund des Umfangs des Raytracers nicht möglich, weshalb AD nicht angewendet werden konnte.

Aufgrund der Komplexität des Raytracers, war es nicht möglich, zu überprüfen, ob die `gh` Funktion die Variablen tatsächlich korrekt verfolgt. Die Dokumentation des `ad` Pakets gibt nur an, dass die Zielfunktion, welche der `gh` Funktion übergeben wird, aus elementarer Mathematik bestehen muss. Es ist deshalb unklar, ob das Paket überhaupt für eine verschachtelte objektorientierte Struktur wie sie bei pyrate vorliegt geeignet ist. Um dennoch Aussagen zur Eignung des Pakets treffen zu können, wurde die Struktur von pyrate nachgebaut und der raytracer selbst durch ein simples System mit einer Linse ersetzt. Dabei wurde auf Objekte der Klasse `np.ndarray` verzichtet.

An dieser Stelle wurde ein sehr simples Problem für den raytracer verwendet. Es wird ein Strahl durch eine konvexe Linse betrachtet. Die Gegenstandshöhe sowie die Bildhöhe ist vorgegeben. Optimiert werden soll die Position der Linse, sodass eine vorgegebene Bildhöhe erreicht wird. Dieses Problem lässt sich auch leicht analytisch berechnen.

Nachfolgend ist das Programm aufgeführt, welches die numerische Optimierung des simplen Systems mit automatischem Differenzieren unter Verwendung des Pakets `ad` umsetzt.

Listing 4.4: Verwendung von ad mit numpy Optimierung

```
from __future__ import print_function
from ad import adnumber
import math
from ad import gh
from scipy.optimize import minimize

class backend:
    def init(self, func):
    self.func=func

class projectbackend(backend):
    def run(self, x0):
        res=0
        bnds = (0, 50)
        def grad_fdm(x):
            h=0.000001
            return (self.func(x+h)-self.func(x-h))/(2*h)

        grad_ad, hess_ad=gh(self.func)

        def grad_ad_man(x):
            return grad_ad(x)

        #res = minimize(self.func, x0, bounds=bnds, method='L-BFGS-B',
        #               jac=my_gradient, options={'ftol': 1e-8, 'disp':False})
        #res = minimize(self.func, x0, bounds=bnds, method='nelder-mead',
        #         options={'xatol': 1e-8, 'disp': True})
        res = minimize(self.func, x0, method='CG', jac=grad_ad_man,\
        options={'disp': True})
        return res

    def update_system(opt_sys, x):
        opt_sys.position=x

class optimizer:
    def __init__(self,opt_sys,meritfunction,backend):
        self.optical_system=opt_sys
        self.meritfunction=meritfunction
        self.setBackend(backend)

    def meritfunctionwrapper(self,x):
        update_system(self.optical_system, x)
        meritwert=self.meritfunction(self.optical_system)
        return meritwert

    def setBackend(self,backend):
        self._backend=backend
        self._backend.init(self.meritfunctionwrapper)

    def run(self):
        x0=self.optical_system.position
        value=self._backend.run(x0)
        return value

class optical_system:
    def __init__(self):
        self.position=1
        self.brennweite=-5.0
        self.gegenstandshoehe=2

    def seqtrace(self):
        b=(1.0/self.brennweite-1.0/self.position)**(-1.0)
        V=-b/self.position
        B=V*self.gegenstandshoehe
        return B

    def get_merit(opt_sys):
        def meritfunctionrms(my_s):
            istwert=my_s.seqtrace()
            sollwert=0.67
            return (istwert-sollwert)**2
        return meritfunctionrms

optsys=optical_system()
backend=projectbackend()
meritfkt=get_merit(optsys)
opt=optimizer(optsys,meritfkt,backend)
res=opt.run()
```

Nun soll kurz auf die wichtigsten Zeilen in Listing 4.4 eingegangen werden:

7: Definieren der Oberklasse für das eigens programmierte `projectbackend`. Dies entspricht exakt der Strutkur wie sie in pyrate verwendet wurde.

11: Wie im ursprünglichen pyrate Code werden in `projectbackend` die Gradienten von `self.func` berechnet. `self.func` ist dabei der `meritfunctionwrapper`. Dargestellt ist die Berechnung des Gradienten über finite Differenzen und über die `gh` Funktion, welche den Gradienten über AD bestimmt.

28: Hier wird wie auch bei pyrate die `minimize` Funktion von `scipy.optimize` verwendet, um die Optimierung durchzuführen. Als Verfahren wurde der Einfachheit halber hier das CG-Verfahren gewählt und nicht wie zuvor die eigens implementierten Optimierungsverfahren, da hier ohne das Lagrange/Penalty Verfahren gearbeitet werden soll. An der Struktur des Programms ändert dies jedoch nichts.

35: Wie bei pyrate ist die Klasse `optimizer` die zentrale Klasse welches das Backend, die Zielfunktion und das optische System verwaltet. Die `run` Funktion wird letztendlich im Hauptprogramm aufgeführt. Diese ruft wiederum die `run` Funktion des Backends auf, welches die eigentliche Optimierung startet.

67: Die Zielfunktion wurde hier sehr vereinfacht angegeben. Es wird die momentane Position des Strahls durch den Raytracer bestimmt, und die Abweichung zum Sollwert quadriert.

74: Ab Zeile 74 werden die gleichen Befehle wie im eigentlichen Hauptprogramm ausgeführt. (vgl. Listing 2.1)

Wird das oben aufgeführte Programm ausgeführt, so wird innerhalb von 10 Iterationen der korrekte Wert für die Position der Linse bestimmt. Die vorgegebene Bildhöhe wird erreicht. Der korrekte Wert wird sowohl mit dem zentralen Differenzenquotient, wie auch mit AD erreicht. Beim zentralen Differenzenquotient wird jedoch für größere Schrittweiten kein Minimum mehr gefunden. Insbesondere für komplexere Zielfunktionen mit mehreren Parametern kann AD seine Stärken ausspielen.

Die `gh` Funktion trackt auch die Position der Linese `self.position` korrekt, was in nachfolgender Abbildung zu sehen ist.

```
type autodiff variable
<class 'ad.ADV'>
value autodiff variable
ad(9.926010648232111)
id autodiff variable
139747034425936
```

Abbildung 4.6: Die Variable `self.position` wird in ad umgewandelt

Die in Abb. 4.6 dargestellte Variable entspricht der Position der Linse und wurde korrekt gecastet.

Auch hier zeigt sich das Problem der fehlenden `np.ndarray` Unterstützung, wenn man in Listing 4.4 in Zeile 57 ein `np.ndarray` für die Position verwendet. Darauf muss also verzichtet werden, möchte man das Paket verwenden.

4.1.5 Lagrange- und Penalty-Term

Nun soll auf die Ableitung der Penalty-Lagrange-Funktion eingegangen werden, um sie anschließend für den Fall $N = 1$ zu veranschaulichen. Für die Ableitung des Lambda-Terms muss

auf Subgradienten zurückgegriffen werden, da dieser nicht überall differenzierbar ist.
Der Lambda-Term hat ausgeschrieben folgende Form:

$$\lambda_k^T h(p) = \sum_{i=1}^{N} \lambda_{k,i} \max\{0, p_i - b_i\} + \sum_{j=1}^{N} \lambda_{k,j+N} \max\{0, a_j - p_j\}$$

Zur einfacheren Darstellung wird hier nur die partielle Ableitung nach einer Variablen p_l aufgeführt.

$$\frac{\partial}{\partial p_l}\left(\lambda_k^T h(p)\right) = \lambda_{k,l}\left(\begin{cases} 1 & , b_l < p_l \\ c_1 & , p_l = b_l \\ 0 & , p_l < b_l \end{cases}\right) + \lambda_{k,l+N}\left(\begin{cases} 0 & , a_l < p_l \\ c_2 & , p_l = a_l \\ -1 & , p_l < a_l \end{cases}\right)$$

wobei $c_1 \in \partial h_l(b_l) = [0,1]$ und $c_2 \in \partial h_{l+N}(a_l) = [-1,0]$ ist. Die Darstellung lässt sich zusammenfassen zu

$$\frac{\partial}{\partial p_l}\left(\lambda_k^T h(p)\right) = \begin{cases} \lambda_{k,l} & , b_l < p_l \\ \lambda_{k,l} \cdot c_1 & , p_l = b_l \\ 0 & , a_l < p_l < b_l \\ \lambda_{k,l+N} \cdot c_2 & , p_l = a_l \\ -\lambda_{k,l+N} & , p_l < a_l \end{cases}$$

Diese Ableitung ist in der Funktion `grad_lag` implementiert. Der Penalty-Term ist wegen dem differenzierbaren $g(p)$ und durch das Quadrieren sogar stetig differenzierbar. Der Penalty-Term ergibt sich wie folgt.

$$\begin{aligned} \frac{1}{2}\tau_k ||h(p)||_2^2 &= \frac{1}{2}\tau_k \left(\sum_{i=1}^{N} (h_i(p))^2 + \sum_{j=1}^{N} (h_{j+N}(p))^2\right) \\ &= \frac{1}{2}\tau_k \left(\sum_{i=1}^{N} (\max\{0, p_i - b_i\})^2 + \sum_{j=1}^{N} (\max\{0, a_j - p_j\})^2\right) \end{aligned}$$

Dieser Term wird nach p_l partiell abgeleitet und nach der Kettenregel folgt:

$$\begin{aligned} \frac{\partial}{\partial p_l}\left(\frac{1}{2}\tau_k ||h(p)||_2^2\right) &= \tau_k \left[h_l(p)\left(\begin{cases} 0 & , p_l \le b_l \\ 1 & , b_l < p_l \end{cases}\right) + h_{l+N}(p)\left(\begin{cases} -1 & , p_l < a_l \\ 0 & , a_l \le p_l \end{cases}\right)\right] \\ &= \tau_k \begin{cases} h_l(p) & , b_l < p_l \\ 0 & , a_l \le p_l \le b_l \\ -h_{l+N}(p) & , p_l < a_l \end{cases} \end{aligned}$$

Diese Ableitung ist in der Funktion `grad_pen` implementiert. Zur Veranschaulichung sei nun der Penalty-Term und der Lambda-Term sowie deren Ableitungen für $\tau = 1$, $\lambda = 1$, $N = 1$ und $[a,b] = [1,4]$ abgebildet.
Wie in den Abbildungen zu erkennen ist, führen die Ableitungen der Terme wieder zurück in den Bereich $\mathcal{F}$. Unter 3.1 wurde angemerkt, dass die Meritfunktion nur in einem kleinen Bereich von $\mathbb{R}^N$ ungleich Null ist und dieser Bereich wird von $\mathcal{F}$ begrenzt. Werden die Terme in der penalty-lagrange-Funktion und die Strafterme 3.9-3.12 alle addiert müssen diese aufeinander abgestimmt werden. Die Wahl von $\mathcal{F}$ sowie 3.9-3.12 wird dadurch sehr schwierig und problemabhängig.

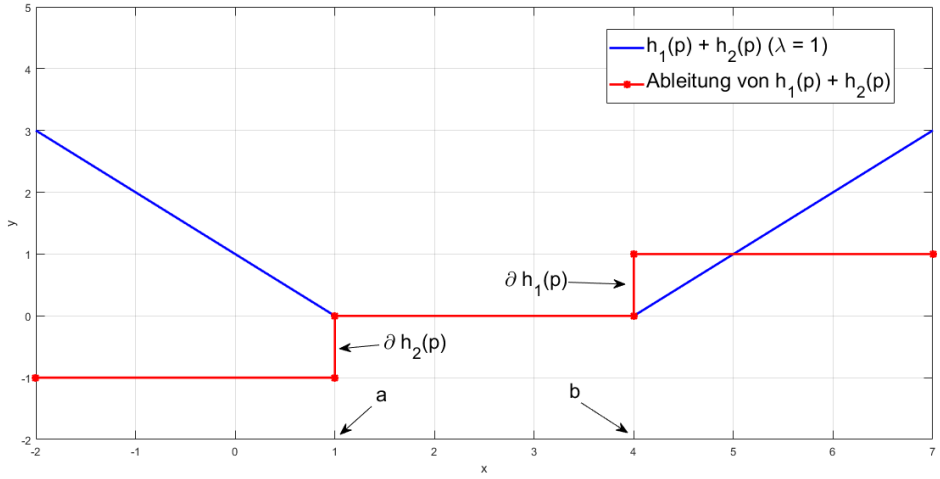

Abbildung 4.7: Lambda-Term und dessen Ableitung

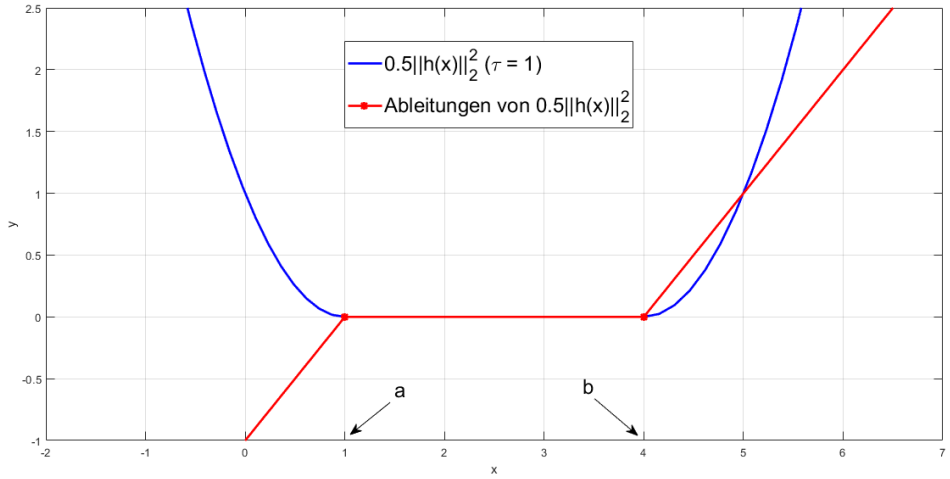

Abbildung 4.8: Penalty-Term und dessen Ableitung

4.2 Particle-Swarm-Optimization mit Nelder-Mead

In diesem Kapitel wird ein neuartiger hybrider Algorithmus vorgestellt, welcher eine Kombination aus den Algorithmen *Nelder-Mead* und *Particle Swarm Optimization* darstellt. Dieser wird im folgenden mit *PSO-NM* abgekürzt. Das Verfahren basiert auf dem Paper [17] und dem Artikel [9]. Der hybride *PSO-NM* Algorithmus löst restringierte Optimierungsprobleme folgender Form.

$$\begin{aligned} \text{Minimiere} \quad & f(x) \\ \text{u.d.N.:} \quad & g_j(x) \leq 0, \quad j = 1, \ldots, q \\ & h_j(x) = 0, \quad j = q+1, \ldots, m \\ & l_i \leq x_i \leq u_i, \quad i = 1, \ldots, n \end{aligned} \tag{4.11}$$

Dabei können sowohl die Nebenbedingungen, als auch die Zielfunktion nichtlinear sein. Beim hier betrachteten Optimierungsproblem sind vor allem die oberen und unteren Grenzen l und u für eine mögliche Lösung x von Bedeutung, da man für diese zulässige Intervalle vorgibt.
In Fällen, in welchen die Zielfunktion oder die Nebenbedingungen nicht differenzierbar sind oder deren numerische Bestimmung zu aufwendig sind, bieten sich heuristische Methoden wie *Particle Swarm Optimization (PSO)* an. Diese haben jedoch den großen Nachteil, dass diese beim Lösen von restringierten Optimierungsproblemen nur sehr langsam konvergieren und selten lokale Optima finden. Bei unrestringierten Optimierungsproblemen hat sich ein hybrider Ansatz aus *PSO* und *Nelder-Mead (NM)* bereits als sehr effizient erwiesen. Hierbei sorgt der *PSO* Algorithmus für die globale und der *NM* für die lokale Suche. Durch Modifikationen, welche im folgenden vorgestellt werden, lässt sich dieser hybride Ansatz auch auf restringierte Optimierungsprobleme anwenden.

4.2.1 Particle-Swarm-Optimization

Particle Swarm Optimization gehört zu den neusten evolutionären Techniken und wurde 1995 von Eberhart und Kennedy entwickelt. Dabei wird das Schwarmverhalten von Vögeln und Fischen aus der Natur nachgeahmt. Bei den künstlichen Schwärmen bewegen sich alle Individuen durch den n-dimensionalen Suchraum (n ist die Anzahl der Dimensionen des Optimierungsproblems (OP)). Dabei entspricht die Position x eines Individuums einer möglichen Lösung des OP und der Zielfunktionswert $f(x)$ gibt die Qualität dieser an. Die Bewegung der Partikel ist dabei nicht willkürlich, sondern folgt einem definierten Muster. Wie in der Natur neigen die Partikel dazu, den Besten zu folgen. Beim künstlichen Schwarm wären im Falle eines Minimierungsproblems diejenigen Partikel gut, welche an einem Ort x sind oder waren, an welchem der Zielfunktionswert $f(x)$ möglichst klein ist. Der erste Schritt beim *PSO* ist die Initialisierung. Hier werden alle Partikel zufällig auf den Suchraum verteilt und ihnen wird eine willkürliche anfängliche Geschwindigkeit zugewiesen. Dann wird in jeder Iteration ein Update der Position und der Geschwindigkeit durchgeführt. Dies geschieht für jedes Individuum nach folgender Vorschrift.

$$\begin{aligned} V_{id}^{New}(t+1) &= c_0 \cdot V_{id}^{old}(t) + c_1 \cdot \text{rand}() \cdot (p_{id}(t) - x_{id}^{old}(t)) + c_2 \cdot \text{rand}() \cdot (p_{swarm}(t) - x_{id}^{old}) \\ x_{id}^{New}(t+1) &= x_{id}^{old}(t) + V_{id}^{New}(t+1) \end{aligned} \tag{4.12}$$

Dabei steht V_{id} für die Geschwindigkeit und x_{id} für die Position eines Partikels. Der Wert p_{id} ist spezfisch für jedes Individuum und gibt den besuchten Ort x eines Individuums an, an

welchem der Zielfunktionswert bislang am besten war. Gleiches gilt für p_{swarm}, jedoch ist dies der von allen Partikeln in allen bisherigen Iterationen beste gefundene Ort bzw. Lösung. Die Konstanten c_1 und c_2 gewichten den Einfluss der besten Lösung eines Partikels im Vergleich zur besten Lösung des Schwarms beim Geschwindigkeitsupdate (meist $c_1 = c_2 = 2$). Die Konstante c_0 zeigt gute Resultate, wenn sie $c_0 = \text{rand}()/3$ gewählt wird. Dabei ist *rand* eine Zufallszahl zwischen 0 und 1. Die Geschwindigkeiten sind dabei durch eine Maximalgeschwindigkeit V_{max} begrenzt. Diese kann wie folgt gewählt werden.

$$v_i^{max} = (u_i - l_i) \cdot 0.1; \quad i = 1, \ldots, n \tag{4.13}$$

Hier sind l und u die unteren bzw. oberen Intervallgrenzen.

4.2.2 Nelder-Mead mit Nebenbedingungen

Der *Nelder-Mead (NM)* Algorithmus ist ein Verfahren, welcher für die lokale Optimierung bei unrestringierten Optimierungsproblemen eingesetzt werden kann. Zudem ist er ableitungsfrei. Die Grundidee ist es, einen Simplex mit $n+1$ Ecken, wobei n die Dimension des Problems ist, in Abhängigkeit der Funktionswerte an den Ecken umzubauen. Dabei gibt es die vier Operationen Reflexion, Expansion, Kontraktion und Komprimierung. Beim klassischen *NM* spielt dabei nur der Zielfunktionswert $f(x)$ an den Ecken eine Rolle. Um das Verfahren jedoch auch für restringierte Optimierungsprobleme brauchbar zu machen, wird eine zweite Größe eingeführt, die sogenannte *Constraint Fitness (Cf)*. Diese Idee stammt aus [17]. Der Wert $Cf(x)$ bewertet, inwiefern eine Ecke x die Nebenbedingungen einhält.

$$C_j(x) = \begin{cases} 1 & \text{,wenn } g_j(x) \leq 0 \\ 1 - \frac{g_j(x)}{g_{max}(x)} & \text{, wenn } g_j(x) > 0 \end{cases} \tag{4.14}$$

Dies gilt für die Ungleichungsnebenbedingungen $g_j(x) \leq 0$ mit $g_{max}(x) = \max\{g_j(x), j = 1, \ldots, q\}$. Die *Constraint Fitness*-Funktion an einem Punkt x ist wie folgt definiert.

$$Cf(x) = \sum_{j=1}^{q} w_j C_j, \quad \sum_{j=1}^{q} w_j = 1, \quad 0 \leq w_j \leq 1 \quad \forall j \tag{4.15}$$

Die Gewichte w_j für die Ungleichung j werden zufällig generiert. Ist $Cf(x) = 1$, so ist x im zulässigen Bereich. Gilt $0 < Cf(x) < 1$, so ist x umso weiter vom zulässigen Bereich entfernt, je kleiner dieser Wert ist.
Damit der Simplex beim *Nelder-Mead* nicht in den unzulässigen Bereich läuft, wird dabei sowohl der Funktionswert $f(x)$, als auch der Wert $Cf(x)$ optimiert.

Algorithm 1 Nelder-Mead-Constraint

function NELDER-MEAD-CONSTRAINT
  Werte $f(x)$ und $Cf(x)$ an jeder Ecke x aus
  while Abbruchkriterium nicht erfüllt **do**
    Reflektion: Bestimme x_{high} und x_{low} in Abhängigkeit der Funktionswerte $f(x)$
    Bestimme Mittelpunkt x_{cent} ohne x_{high}
    Bestimme $x_{refl} = (1+\alpha)x_{cent} - \alpha x_{high}$
    if $Cf_{refl} < 1 \text{und} Cf_{refl} > Cf_{low} \text{oder} Cf_{refl} = 1 \text{und} f_{refl} < f_{low}$ **then**
      Expansion:
      if $Cf_{refl} < 1 \text{und} Cf_{refl} > Cf_{low}$ **then**
        $x_{exp} = \gamma x_{refl} + (1-\gamma)x_{cent}$
      end if
      if $Cf_{exp} > Cf_{low} \text{oder} Cf_{exp} = 1$ **then**
        $x_{high} = x_{exp}$
      else
        $x_{high} = x_{refl}$
      end if
    else
      Kontraktion
      if $Cf_{refl} < 1 \text{und} Cf_{refl} \leq Cf_{low}$ **then**
        Berechne $x_{cont} = \beta x_{high} + (1-\beta)x_{cent}$
        if $Cf_{cont} > Cf_{low} \text{oder} Cf_{cont} = 1$ **then**
          $x_{high} = x_{cont}$
        else
          $x_i \leftarrow \delta x_i + (1-\delta)x_{low} \quad \forall$Ecken außer x_{low}
        end if
      end if
      if $Cf_{refl} = 1 \text{und} f_{refl} \geq f_{low} \text{und} f_{refl} \leq f_{high}$ **then**
        $x_{high} = x_{refl}$
        $x_i \leftarrow \delta x_i + (1-\delta)x_{low} \quad \forall$Ecken außer x_{low}
      end if
      if $Cf_{refl} = 1 \text{und} f_{refl} \geq f_{low} \text{und} f_{refl} > f_{high}$ **then**
        $x_i \leftarrow \delta x_i + (1-\delta)x_{low} \quad \forall$Ecken außer x_{low}
        if $Cf_{cont} = 1 \text{und} f_{cont} < f_{low}$ **then**
          $x_{high} = x_{cont}$
        else
          $x_i \leftarrow \delta x_i + (1-\delta)x_{low} \quad \forall$Ecken außer x_{low}
        end if
      end if
    end if
  end while
end function

Wenn die Parameter $\alpha = 1$, $\beta = 0.5$, $\gamma = 2$ und $\delta = 0.5$ gewählt werden, erzielt man gute Ergebnisse. Mögliche Abbruchkriterien sind das Erreichen einer maximalen Iterationszahl oder wenn sich die Funktionswerte an allen Ecken des Simplex nur noch geringfügig unterscheiden.

4.2.3 PSO-NM Hybrid

Das Grundkonzept des hybriden *PSO-NM* Algorithmus ist es, die Position eines Partikels in jeder Iteration durch das Anwenden des *NM*-Algorithmus zu verbessern (basierend auf [17]). In der Literatur wird empfohlen, als Populationsgröße $21n + 1$ Partikel zu verwenden, wobei n die Dimension des Optimierungsproblems ist.

Algorithm 2 PSO-NM

function PSO-NM
  Initialisierung:
    Verteile $21n + 1$ Partikel im zulässigen Bereich
    Gebe jedem Partikel Geschwindigkeit $v_{random} < v_{max}$
  while Abbruchkriterium nicht erfüllt **do**
    Repair-Methode:
      Schiebt alle Partikel, welche im nichtzulässigen Bereich sind in den zulässigen Bereich
    Ranking:
      Ordne alle Partikel entsprechend des Funktionswertes von gut nach schlecht
    Nelder-Mead:
      Wende NM-Verfahren auf die besten $n + 1$ Partikel an und update $(n + 1)^{th}$ Partikel
    PSO:
      Bestimme global bestes Partikel (gl) und bestes Partikel aus jeder Nachbarschaft (nb)
      Führe folgendes Update für die verbleibenden $20n$ Partikel durch:
      $V_{id}^{New} = c_0 \cdot V_{id}^{old} + c_1 \cdot \text{rand}() \cdot (p_{gl} - x_{id}^{old}) + c_2 \cdot \text{rand}() \cdot (p_{nb} - x_{id}^{old})$
      $x_{id}^{New} = x_{id}^{old} + V_{id}^{New}$
  end while
end function

Zu Beginn werden die Partikel erzeugt und per Zufall im zulässigen Bereich verteilt, d.h. es gilt für alle Partikel $Cf(x) = 1$. Zudem wird diesen eine zulässige zufällige Anfangsgeschwindigkeit zugewiesen. Die *Repair-Methode* bringt Partikel, die in der vorherigen Iteration nach dem PSO-Schritt aus dem zulässigen Bereich geflogen sind, wieder in diesen zurück. Da die einzigen Nebenbedingungen in den hier vorgestellten OP's obere und untere Intervallgrenzen sind, reicht es aus, die Partikel in den Komponenten, in welchen sie die Grenzen verletzen, auf diese zu verschieben.

$$x_i = \begin{cases} x_i & \text{,wenn } l_i \leq x_i \leq u_i \\ l_i & \text{,wenn } x_i < l_i \\ u_i & \text{,wenn} x_i > u_i \end{cases} \qquad , i = 1, \ldots, n \tag{4.16}$$

Gleichung 4.16 veranschaulicht diesen Vorgang. Dabei sind l_i und u_i die untere bzw. obere Grenze der Komponente i. Anschließend werden die Partikel aufgrund ihres aktuellen Zielfunktionswertes $f(x)$ von gut nach schlecht sortiert. Die $n + 1$ besten Partikel bilden dabei den Ausgangssimplex für das *Nelder-Mead* Verfahren. Der schlechteste dieser Partikel, also der $(n + 1)$-te, wird mit der durch den *NM*-Algorithmus erhaltenen Lösung ersetzt und somit verbessert. Nun werden die restlichen $20n$ Partikel mithilfe eines PSO-Schrittes upgedatet. Dabei wird zuerst die aktuell global beste Lösung aus allen $21n + 1$ Partikeln ermittelt. Dann werden die $20n$ Individuen in $10n$ 2-er Nachbarschaften eingeteilt. Für jede Nachbarschaft wird der bessere der beiden Partikel ermittelt. Das Update der $20n$ Partikel erfolgt nach Algorithmus 2. Dabei berücksichtigen alle die Position des global besten Partikels und die Position des besseren Individuums in ihrer jeweiligen Nachbarschaft.
Für die Abbruchbedingung gibt es zwei Kriterien. Zum einen, wenn eine maximale Anzahl an Iterationen erreicht wurde. Zum anderen, wenn sich der global beste Funktionswert $f_{best}(t)$ nach k Iterationen um weniger als $0.01 \cdot f_{best}(t - k)$ verbessert hat. Es hat sich gezeigt, dass ein Wert von $k = 70$ zu sinnvollen Ergebnissen führt.

4.2.4 Weitere Varianten

Variante 1:
Diese Variante entspricht dem oben beschriebenem Konzept.

Variante 2:
Hierbei ändert sich die Regel für das Update beim PSO-Schritt. Dabei werden die Partikel nicht mehr in Nachbarschaften eingeteilt. Dieses Konzept entspricht dem klassischen *Particle Swarm Optimization*.

$$V_{id}^{New} = c_0 \cdot V_{id}^{old} + c_1 \cdot \text{rand}() \cdot (p_{id} - x_{id}^{old}) + c_2 \cdot \text{rand}() \cdot (p_{gl} - x_{id}^{old}) \tag{4.17}$$

Hier ist p_{id} die in allen Iterationen beste gefundene Lösung eines jeden Partikels und p_{gl} wie bei Variante 1 die aktuell beste globale Lösung. Durch diese Variation soll die Suche in Bereichen, in welchen zwar gute, aber nicht optimale Lösungen gefunden wurden, verstärkt werden.

Variante 3:
Auch hier werden die $20n$ Partikel in Nachbarschaften eingeteilt. Dies sind jedoch keine 2-er Nachbarschaften, sondern sie werden gleichmäßig auf S Gruppen aufgeteilt. Zudem bleiben die Individuen in allen Generationen in derselben Nachbarschaft.

$$V_{id}^{New} = c_0 \cdot V_{id}^{old} + c_1 \cdot \text{rand}() \cdot (p_{id} - x_{id}^{old}) + c_2 \cdot \text{rand}() \cdot (p_{nbest} - x_{id}^{old}) \tag{4.18}$$

Dabei gibt p_{nbest} die aus allen Partikeln einer Nachbarschaft in allen Iterationen beste gefundene Lösung an. p_{id} ist äquivalent zu Variante 2. Somit werden die Partikel nur von Partikeln aus deren Nachbarschaft beeinflusst. Dadurch soll die Gefahr einer vorzeitigen Konvergenz verringert werden, da die Gruppen unabhängig voneinander arbeiten.

Variante 4:
Diese Variante ist äquivalent zu der Ausgangsvariante mit dem Unterschied, dass hier nicht nur auf die restlichen $20n$ Partikel ein PSO-Schritt angwandt wird, sondern auf alle $21n + 1$ Partikel (also auch auf die $n + 1$ besten Partikel). Die Idee stammt aus [9].

Variante 5:
Dies ist an sich keine neue Variante, sondern kann auf alle vorherigen Varianten angewandt werden. Dabei wird die beste und finale Lösung x_{best}, welche der *PSO-NM* Algorithmus liefert, als Startwert für einen effizienten lokalen Algorithmus verwendet. Gradientenbasierte lokale Verfahren liefern meist bessere Ergebnisse, haben jedoch das Problem, dass deren Resultat stark vom Startwert abhängt. Da sich die Lösung des *PSO-NM* Algorithmus jedoch in einem guten globalen Bereich befindet, kann man durch diese Kombination verbesserte Ergebnisse erwarten. Als lokale Verfahren bieten sich beispielsweise das CG- oder Truncated-Newton-Verfahren an.

4.3 Population Based Incremental Learning - Hybrid

Dieser hybride Algorithmus kombiniert den evolutionären Algorithmus (EA) *Population Based Incremental Learning (PBIL)* mit effizienten lokalen Suchmethoden. Dieser basiert auf dem Paper [2, S. 379-391]. Evolutionäre Algorithmen haben im Verlgeich zu klassischen Optimierungsverfahren große Vorteile. So sind diese relativ leicht zu implementieren, sind sehr robust und ermöglichen die Suche nach globalen Optima. Die entscheidenden Nachteile sind jedoch die langsame Konvergenz und die meist große Anzahl an Funktionsauswertungen. Zudem gibt es keine Garantie, dass sie gegen ein Optimum konvergieren. Aus diesem Grund ist es unablässig, die Vorteile der globalen Suche von EA's mit effizienten lokalen Algorithmen zu kombinieren.

Im folgenden wird zuerst auf die Funktionsweise des *Population Based Incremental Learning*-Verfahrens eingegangen. Anschließend werden zwei lokale Suchverfahren vorgestellt, welche den *Evolutionary Direction Operator* und den *Approximate Gradient* nutzen. Zuletzt wird der hybride Algorithmus aufgezeigt.

4.3.1 Population Based Incremental Learning (PBIL)

Hier wird der sogenannte *Real-Code PBIL (RPBIL)* betrachtet (siehe [11]). Dieser eignet sich für Optimierungsprobleme mit Intervallgrenzen.

$$\begin{aligned} &\text{Min } f(x) \\ &L_i \leq x_i \leq U_i; \quad i = 1, \ldots, n \end{aligned} \tag{4.19}$$

Dabei ist x der Lösungsvektor und L_i und U_i die untere bzw. obere Grenze für jede Komponente x_i (n ist Dimension des Problems). Beim *RPBIL* wird zu Beginn jeder Iteration eine Population aus zulässigen Lösungen x erzeugt. Dies geschieht in Abhängigkeit der sogenannten *Probability-Matrix P*. Die Dimension von P ist $n \times m$. Dabei werden die zulässigen Intervalle $[L_i, U_i]$ in m Subintervalle eingeteilt. Der Eintrag P_{ij} gibt dabei die Wahrscheinlichkeit an, dass sich die i-te Komponente vom Lösungsvektor x im Subintervall $j = [L_i + (j-1)\delta_i, L_i + j\delta_i]$ mit $\delta_i = (U_i - L_i)/m$ befindet. Ein großer P_{ij} Wert bedeutet, dass viele der erzeugten zulässigen Lösungsvektoren x in der x_i-ten Komponente einen Wert aus dem Subintervall j haben. Dabei bekommt x_i einen zufälligen Wert aus j. Nach diesem Prinzip werden N_P (Populationsgröße) zulässige Lösungsvektoren x erzeugt.
Zu Beginn hat P in allen Einträgen denselben Wert $1/m$. Ausgehend davon wird eine erste Population (erste Generation) erzeugt. Die Zielfunktion wird an allen Lösungsvektoren x ausgewertet und somit x_{best} ermittelt. Zum Abschluss jeder Iteration wird die *Probability Matrix* P nach x_{best} upgedatet, sodass die Subintervalle, in welcher sich aktuell die beste Lösung befindet, einen höheren Wahrscheinlichkeitswert erhalten und somit in der nächsten Iteration mehr Individuen in diesem Bereich platziert werden.

$$\begin{aligned} P'_{ij} &= (1 - L_{R,j}) P_{ij}^{old} + L_{R,j} \\ L_{R,j} &= L_{R0} \cdot \exp(-(j - r(i))^2) \end{aligned} \tag{4.20}$$

Dabei gibt $r(i)$ das Subintervall an, in welchem sich die i-te Komponente von x_{best} befindet ($i = 1, \ldots, n$). Die Größe L_{R0} wird als Lernrate bezeichnet und wird meist $L_{R0} = 0.5$ gewählt. Die normierte Matrix ergibt sich wie folgt.

$$P_{ij} = \frac{1}{\sum_{j=1}^{m} P'_{ij}} P'_{ij} \tag{4.21}$$

Algorithm 3 Real-Code Population Based Incremental Learning

```
1: function RPBIL
2:     Initialisierung: P_ij = 1/m, δ_i = (U_i − L_i)/m, x_best = {}
3:     for it = 1 : N_G do
4:         Erzeuge Population
5:         for i = 1 : n do
6:             for j = 1 : m do
7:                 Erzeuge per Zufall N_P · P_ij Elemente von x_i im Intervall [L_i + (j − 1)δ_i, L_i + jδ_i]
8:             end for
9:             Mische per Zufall die N_P Elemente von x_i
10:            Setze die N_P Werte von x_i in die i-te Zeile der Populationsmatrix X(it)
11:        end for
12:        Berechne f(it) = f(X(it))
13:        Finde x_best aus X ∪ x_best(it − 1)
14:        Update P_ij in Bezug auf x_best
15:        for i = 1 : n do
16:            Bestimme r(i), sodass gilt x_i^best ∈ [L_i + (r(i) − 1)δ_i, L_i + r(i)δ_i]
17:            Update P_ij
18:        end for
19:    end for
20: end function
```

In Algorithmus 3 ist N_G die maximale Anzahl an Iterationen (Anzahl der Generationen). Durch das Mischen in Zeile 9 werden die einzelnen Komponenten der Lösungsvektoren x auf unterschiedliche Weise neu zusammengestellt und somit eine vielfältigere Suche ermöglicht.

4.3.2 Evolutionary Direction Operator

Das erste lokale Suchverfahren nutzt den sogenannten *Evolutionary Direction Operator (EDO)*. Dieses benötigt als Input drei Individuen bzw. zulässige Lösungsvektoren x_1, x_2 und x_3. Dabei hat x_1 den besten Zielfunktionswert der drei.

$$s = (x_1 - x_2) + (x_1 - x_3) + c \tag{4.22}$$

Formel 4.22 zeigt die Berechnung der Evolutionären Richtung. Der Vektor c soll eine vorzeitige Konvergenz verhindern.

$$c_i = 0.05 \cdot randn() \quad ; i = 1, \ldots, n \tag{4.23}$$

Dabei ist $randn()$ eine normalverteilte Zufallszahl mit Mittelwert 0 und Standardabweichung 1. Aus den drei Eltern-Individuen werden zwei Kinder-Individuen wie folgt berechnet.

$$\begin{aligned} y_1 &= x_1 + rand() \cdot \lambda(it) \cdot s \\ y_2 &= x_1 - rand() \cdot \lambda(it) \cdot s \end{aligned} \tag{4.24}$$

Hierbei ist $rand()$ eine uniforme Zufallszahl aus dem Intervall $[0, 1]$. Der Parameter $\lambda(it)$ definiert die maximale Schrittweite in der Iteration it. Diese beträgt in der ersten Generation $\lambda(1) = 0.5$ und in der letzten Generation nur noch $\lambda(N_G) = 0.001$. Somit beschränkt sich in späteren Iterationen die Suche nach verbesserten Lösungen auf einen immer kleiner werdenden Bereich.

$$\lambda(it) = \exp\left(\frac{6.2146}{N_G - 1} - 0.6931\right) \exp\left(-\frac{6.2146}{N_G - 1} \cdot it\right) \tag{4.25}$$

Wenn die neuen Lösungen $y_{1,2}$ im zulässigen Bereich liegen, werden deren Zielfunktionswerte bestimmt und sie werden in die Population aufgenommen. Andernfalls werden diese verworfen.

4.3.3 Approximate Gradient

Dieses Verfahren gehört zu den ableitungsbasierten Methoden und eignet sich daher sehr gut als lokaler Suchalgorithmus. Hierbei wird der Gradient an einem Punkt x jedoch nicht numerisch bestimmt, was zu einigen zusätzlichen Funktionsauswertungen führen würde, sondern approximativ. Man nimmt dazu eine gewisse Anzahl an Individuen aus der Population und deren zuvor berechneten Zielfunktionswert $f(x)$. Die Richtungsableitung an einem Punkt x in Richtung s kann wie folgt ausgedrückt werden.

$$u_s^T \nabla f = \frac{df}{ds} \approx \frac{f(x) - f(x + \Delta x)}{\Delta s} \tag{4.26}$$

Dabei beschreibt u_s den Einheitsvektor von s. Um den approximativen Gradienten zu berechnen, benötigt man x_{best} und den dazugehörigen Funktionswert f_{best}. Zusätzlich benötigt man R weitere Individuen, welche am nächsten an x_{best} liegen.

$$\begin{bmatrix} (x_{best} - x_1)^T \\ \vdots \\ (x_{best} - x_R)^T \end{bmatrix} \nabla f = A \nabla f = \begin{Bmatrix} f_{best} - f_1 \\ \vdots \\ f_{best} - f_R \end{Bmatrix} = b \tag{4.27}$$

Dabei sollte R größer als n sein um eine Singularität zu vermeiden. Das Gleichungssystem kann mithilfe der Pseudo-Inverse gelöst werden. Da ∇f in die Richtung des steilsten Aufstiegs zeigt, muss gelten $s = -\nabla f$. Für die neuen Lösungen gilt:

$$\begin{aligned} z_1 &= x_{best} + \beta_1 \cdot s \quad \text{mit } \beta_{min} \leq \beta_1 \leq \beta_{start} \\ z_2 &= x_{best} + \beta_2 \cdot s \quad \text{mit } \beta_{min} \leq \beta_2 \leq \beta_{start} \end{aligned} \tag{4.28}$$

Die Schrittweiten β_1 und β_2 werden über einen Backtracking-Algorithmus bestimmt. Dabei wird β_{start} ausgehend von einem zufälligen Startwert aus dem Intervall $(0, 1]$ schrittweise bis zu einem β_{min} verkleinert, bis gilt:

$$f(z) < f(x_{best}) \text{ und } z \text{ ist im zulässigen Bereich} \tag{4.29}$$

Dabei soll gelten $\beta_1 \neq \beta_2$. Wenn neue Lösungen z_1 und z_2 gefunden werden, welche 4.29 erfüllen, so wird deren Zielfunktionswert ausgewertet und sie werden in die Population aufgenommen. Außerdem wird eine dritte Lösung mithilfe von quadratischer Interpolation bestimmt. Dabei geht man davon aus, dass die Zielfunktion entlang der Suchrichtung einer quadratischen Funktion entspricht.

$$f(x_{best} + \beta \cdot s) = C_1 \beta^2 + C_2 \beta + C_3 \tag{4.30}$$

Die Koeffizienten von Gleichung 4.30 können über das folgende Gleichungssystem bestimmt werden.

$$\begin{bmatrix} 0 & 0 & 1 \\ \beta_1^2 & \beta_1 & 1 \\ \beta_2^2 & \beta_2 & 1 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} f_{best} \\ f(z_1) \\ f(z_2) \end{bmatrix} \tag{4.31}$$

Nun lässt sich β_3 derart bestimmen, dass man das Minimum der quadratischen Funktion erreicht. Daher muss man eine Nullstelle der Ableitung in Abhängigkeit von β_3 finden.

$$\begin{aligned} f(x_{best} + \beta_3 \cdot s) &= C_1 \beta_3^2 + C_2 \beta_3 + C_3 \\ f'(x_{best} + \beta_3 \cdot s) &= 2 C_1 \beta_3 + C_2 \stackrel{!}{=} 0 \end{aligned} \tag{4.32}$$

Somit lässt sich die dritte Lösung z_3 wie folgt berechnen:

$$\begin{aligned} z_3 &= x_{best} + \beta_3 \cdot s \\ \text{mit } \beta_3 &= -C_2/(2C_1) \end{aligned} \tag{4.33}$$

Befindet sich z_3 im zulässigen Bereich, so wie die Zielfunktion an dieser Stelle ausgewertet und das Individuum in die Population aufgenommen. Andernfalls wird die Lösung verworfen.

4.3.4 Hybrider Algorithmus

Das hybride Verfahren ist in Algorithmus 4 gezeigt. Zu Beginn jeder Iteration werden $N_P/2$ Individuen in Abhängigkeit der *Probability Matrix* erzeugt. Ausgehend von dieser Population werden mithilfe des approximativen Gradienten drei weitere Individuen erzeugt. Die restlichen $N_P/2 - 3$ Lösungen werden über das *EDO* Verfahren ermittelt. Anschließend wird aus der gesamten Population x_{best} ermittelt und P wird upgedatet. Der Algorithmus terminiert, wenn eine maximale Anzahl an Iterationen, also N_G, erreicht wurde oder sich f_{best} nach k Iterationen nicht um mindestens $0.01 * f_{best}(it - k)$ verbessert hat.

Algorithm 4 Population Based Incremental Learning - Hybrid

function PBIL-HYBRID
  Initialisierung: $P_{ij} = 1/m, \delta_i = (U_i - L_i)/m, x_{best} = \{\}$
  for $it = 1 : N_G$ **do**
    Erzeuge Population mit $N_P/2$ Individuen
    for $i = 1 : n$ **do**
      for $j = 1 : m$ **do**
        Erzeuge per Zufall $(N_P/2) \cdot P_{ij}$ Elemente von x_i im Intervall $[L_i + (j-1)\delta_i, L_i + j\delta_i]$
      end for
      Mische per Zufall die $(N_P/2)$ Elemente von x_i
      Setze die $(N_P/2)$ Werte von x_i in die i-te Zeile der Populationsmatrix X_1
    end for
    Berechne approximativen Gradienten
    Berechne drei neue Individuen X_2 mithilfe des approximativen Gradienten
    Erzeuge $(N_P/2) - 3$ Individuen X_3 mithilfe des *Evolutionary Direction Operator*
    Vereinige $X(it) = X_1 \cup X_2 \cup X_3$
    Berechne $f(it) = f(X(it))$
    Finde neues $x_{best}(it)$ aus $X(it) \cup x_{best}(it-1)$
    Update P_{ij} in Bezug auf x_{best}
    for $i = 1 : n$ **do**
      Bestimme $r(i)$, sodass gilt $x_i^{best} \in [L_i + (r(i)-1)\delta_i, L_i + r(i)\delta_i]$
      Update P_{ij}
    end for
  end for
end function

4.4 Innere Punkte Verfahren[4]

Betrachten wir folgendes, allgemeines Minimierungsproblem

$$\begin{aligned} \min_{x,s} \; & f(x) \\ \text{unter } & c_E(x) = 0 \\ & c_I(x) - s = 0 \\ & s \geq 0 \end{aligned} \tag{4.34}$$

[4]Nach [10], S. 563 ff

wobei der Vektor $c_I(x)$ aus den skalaren Funktionen $c_i(x)$ mit $i \in \mathcal{I}$ besteht, und $c_E(x)$ analog aus den skalaren Funktionen $c_e(x)$ mit $e \in \mathcal{E}$. $\mathcal{I}$ bzw. $\mathcal{E}$ beschreiben dabei die Indices der Ungleichheits- (*Inequality*) bzw. Gleichheits-Nebenbedingungen (*Equality*). Dabei wurden die Ungleichheits-Nebenbedinungen mithilfe des Vektors s in Gleichheits-Nebenbedingungen umgewandelt.

Die *Karush-Kuhn-Tucker-Bedingungen* von 4.34 lauten nun:

$$\begin{aligned} \nabla f(x) - A_E^T(x)y - A_I^T(x)z &= 0 \\ Sz - \mu e &= 0 \\ c_E(x) &= 0 \\ c_I(x) - s &= 0 \end{aligned} \tag{4.35}$$

Mit $\mu = 0$, $s \geq 0$ und $z \geq 0$.

$A_E(x)$ und $A_I(x)$ sind die jeweiligen Jacobi Matrizen der Funktionen $c_E(x)$ bzw. $c_I(x)$, und y und z sind die dazugehörigen *Lagrange-Multiplikatoren*. S und Z seien die Diagonalmatrizen mit den Diagonaleinträgen von s bzw. z, e sei der Vektor $e = (1, 1, ..., 1)^T$.

Die Idee ist nun, die gestörten *KKT-Bedingungen* (4.35) für eine Sequenz an positiven Parametern μ_k zu lösen. Dabei konvergiert die Sequenz gegen Null, während $s, z \geq 0$ gilt. Dies geschieht, indem man das nichtlineare Gleichungssystem 4.35 mittels der *Newton-Methode* löst:

$$\begin{pmatrix} \nabla^2_{xx}\mathcal{L} & 0 & -A_E^T(x) & -A_I^T(x) \\ 0 & Z & 0 & S \\ A_E(x) & 0 & 0 & 0 \\ A_I(x) & -I & 0 & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_s \\ p_y \\ p_z \end{pmatrix} = - \begin{pmatrix} \nabla f(x) - A_E^T(x)y - A_I^T(x)z \\ Sz - \mu e \\ c_E(x) \\ c_I(x) - s \end{pmatrix} \tag{4.36}$$

$\mathcal{L}$ ist dabei die Lagrangefunktion

$$\mathcal{L}(x, s, y, z) = f(x) - y^T c_E(x) - z^T(c_I(x) - s)$$

Nachdem der Schritt $p = (p_x, p_s, p_y, p_z)$ berechnet wurde, ergibt sich die neue Iterierte $(x_{k+1}, s_{k+1}, y_{k+1}, p_{k+1})$ $(k = 1, 2, 3, ...)$ folgendermaßen:

$$\begin{aligned} x_{k+1} &= x_k + \alpha_s^{max} p_x \\ s_{k+1} &= s_k + \alpha_s^{max} p_s \\ y_{k+1} &= y_k + \alpha_z^{max} p_y \\ z_{k+1} &= z_k + \alpha_z^{max} p_z \end{aligned} \tag{4.37}$$

wobei

$$\begin{aligned} \alpha_s^{max} &= \max(\alpha \in (0, 1] : s + \alpha p_s \geq (1 - \tau)s) \\ \alpha_z^{max} &= \max(\alpha \in (0, 1] : z + \alpha p_z \geq (1 - \tau)s) \end{aligned} \tag{4.38}$$

mit $\tau \in (0, 1)$. Die Bedingung 4.38 (*boundary rule*) soll dabei verhindern, dass die Variablen s bzw. z zu schnell gegen Null laufen.

4.4.1 Systemreduzierung

Um das primal-duale System 4.36 effektiver lösen zu können, wird es auf symmetrische Form gebracht:

$$\begin{pmatrix} \nabla^2_{xx}\mathcal{L} & 0 & A_E^T(x) & A_I^T(x) \\ 0 & \Sigma & 0 & -I \\ A_E(x) & 0 & 0 & 0 \\ A_I(x) & -I & 0 & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_s \\ -p_y \\ -p_z \end{pmatrix} = - \begin{pmatrix} \nabla f(x) - A_E^T(x)y - A_I^T(x)z \\ z - \mu S^{-1}e \\ c_E(x) \\ c_I(x) - s \end{pmatrix} \tag{4.39}$$

mit $\Sigma = S^{-1}Z$. Das System kann noch weiter vereinfacht werden, indem man zuerst mit der zweiten Zeile aus 4.36 p_s eliminiert, und anschließend p_z analog aus dem entstehenden System eliminiert. Dadurch entsteht eine wesentlich kleinere Koeffizentenmatrix, wodurch das System 4.39 effektiver gelöst werden kann.

$$\begin{pmatrix} \nabla^2_{xx}\mathcal{L} + A_I^T \Sigma A_I & A_E^T(x) \\ A_E(x) & 0 \end{pmatrix} \tag{4.40}$$

4.4.2 Anpassungen für nichtkonvexe Probleme

Da das System 4.36 unter Umständen auch gegen ein Maximum oder Sattelpunkt konvergieren kann, muss sichergestellt sein, dass p eine Abstiegsrichtung ist. Dies ist der Fall, wenn die Matrix

$$\begin{pmatrix} \nabla^2_{xx}\mathcal{L} & 0 \\ 0 & \Sigma \end{pmatrix}$$

positiv definit auf dem Nullraum der Matrix

$$\begin{pmatrix} A_E(x) & 0 \\ A_I(x) & -I \end{pmatrix}$$

ist.

Nach *Theorem 16.3* aus [10] ist dies der Fall, wenn die primal-dual Matrix in 4.36 $n + m$ positive und $l + m$ negative Eigenwerte besitzt. Dabei bezeichnet

- l die Anzahl der Gleichheits-Nebenbedingungen,
- m die Anzahl der Ungleichheits-Nebenbedingungen und
- n die Anzahl der Einträge in x.

Um dies und die nicht-Singularität zu gewährleisten, passen wir das System 4.39 an, mit $\delta \geq 0$ und $\gamma \geq 0$.

$$\begin{pmatrix} \nabla^2_{xx}\mathcal{L} + \delta I & 0 & A_E^T(x) & A_I^T(x) \\ 0 & \Sigma & 0 & -I \\ A_E(x) & 0 & -\gamma I & 0 \\ A_I(x) & -I & 0 & 0 \end{pmatrix} \tag{4.41}$$

4.4.3 Anwendung auf die Problemstellung

Für die zu untersuchende Optimierungsaufgabe existiert für jede Variable x_i eine obere und untere Grenze (b_{upper} und b_{lower}). Dadurch können wir für jede Variable zwei Ungleichheits-Bedingungen definieren.

$$x_i - b_{lower} \geq 0$$
$$b_{upper} - x_i \geq 0$$

D. h., bei n Variablen existieren $2n$ Ungleichheits- und keine Gleichheits-Bedingungen. Dadurch entfällt die Matrix $A_E(x)$ bzw. der Vektor y, wodurch das System 4.40 weiter vereinfacht werden kann und lediglich noch aus $\nabla^2_{xx}\mathcal{L} + A_I^T \Sigma A_I$ besteht.

Des Weiteren sind alle Ungleichheits-Bedingungen lineare Funktionen. Dadurch gilt in unserem Fall

$$\nabla^2_{xx}\mathcal{L} = \nabla^2_{xx} f.$$

4.4.4 Einfacher IP-Algorithmus

Mit obigen Voraussetzungen wurde ein Algorithmus in *Python* implementiert, welcher auf das Optimierungsproblem angewendet werden kann. Dieser soll im Folgenden genauer beschrieben werden.

Algorithm 5 Innere Punkte Verfahren

```
1: function BASIC_IPM(x_0, s_0, z_0, μ)
2:     k ← 0
3:     while Konvergenzkriterium nicht erreicht do
4:         while E(x_k, s_k, z_k, μ_k) > μ_k do
5:             Berechne δ und addiere es auf die Matrix (siehe 4.41)
6:             Löse 4.41
7:             Berechne α_s^max und α_z^max
8:             Berechne x_{k+1}, s_{k+1}, z_{k+1} nach 4.37
9:             k ← k + 1
10:        end while
11:        Update μ
12:    end while
13:    return x
14: end function
```

Zeile 1: Der Algorithmus benötigt folgende Startparameter

- Startvektor x_0
- Vektor $s_0 \geq 0$
- Vektor $z_0 \geq 0$
- $\mu \geq 0$

Zeile 3: Start der äußeren Schleife, bis der Gradient der Funktion f eine gewisse Grenze unterschreitet

Zeile 4: Start der inneren Schleife, wobei der Fehler $E(x_k, s_k, z_k, \mu_k)$ definiert ist als

$$E(x_k, s_k, z_k, \mu_k) = max(\left\|\nabla f(x) - A_I^T z\right\|, \|Sz - \mu e\|, \|c_I(x) - s\|) \tag{4.42}$$

Zeile 5: Um sicherzustellen, dass ein Lösungsschritt in Abstiegsrichtung erfolgt, müssen die Anzahl der positiven und negativen Eigenwerte berechnet werden (Abschnitt 4.4.2). Dies geschieht indem der *Bunch-Kaufmann-Algorithmus* (in *scipy* enthalten) auf die primal-dual Matrix angewendet wird. Dadurch kann die Anzahl der pos./neg. Eigenwerte einfach festgestellt werden, und ein größerwerdendes δ solange aufaddiert werden, bis die notwendige Bedingung erfüllt ist. Die Wahl von δ erfolgt hierbei nach dem *Inertia Correction and Regularization* Algorithmus aus [10] (S. 635).

Zeile 6: Anschließend wird das Gleichungssystem gelöst, wobei hier das reduzierte System 4.40 verwendet wird. Im implementierten Algor. wird hierfür die *Generalized Minimal Residual* Methode verwendet (ebenfalls in *scipy* enthalten).

Zeile 7: Die Berechnung der α's erfolgt nach der Bedingung 4.38, wobei $\tau = 0,995$ gesetzt wurde.

Zeile 11: μ wird reduziert für den nächsten Schleifendurchgang nach der adaptiven Formel aus [10], S. 572 f

4.5 Varianten des stochastic gradient descent

In den letzten Jahren wurden die Methoden im Bereich Machine Learning und insbesondere Neural Networks immer populärer. Dabei tritt oftmals ein Minimierungsproblem der Form

$$\min_{w\in\mathbb{R}^n} \frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} L(x,y,w) \tag{4.43}$$

ähnlich zu 3.13a auf, wobei $\mathcal{S}$ die Trainingsmenge ist und $L(x,y,w)$ eine Funktion die nach $\mathbb{R}_0^+$ abbildet. Häufig ist $|\mathcal{S}| \gg 1$ und $n \gg 1$, was für gradientenbasierte Optimierungsverfahren eine Herausforderung darstellt, denn es muss der Gradient von jedem Summanden berechnet werden. Um dies zu umgehen wurde vermehrt der *stochastic gradient descent* (sgd) und seine Varianten eingesetzt. Diese wurden im Projekt auf das unrestringiertes Optimierungsproblem angewandt.

4.5.1 Stochastischer Gradient

Es gibt mehrere Möglichkeiten um den stochastischen Gradienten von 3.8 zu berechnen. Diese variieren in der Anzahl der Summanden die zufällig gezogen werden. Sei wieder $B_{ij} \in \mathcal{B}$ wie in Definition 3.4 und $[N] = \{1,2,\dots,N\}, N \in \mathbb{N}$, dann sind die Möglichkeiten die implementiert wurden:

$$\underset{z\sim[M]}{\nabla} mf_l^k(p) = M \sum_{i=1}^{3}\sum_{q=1}^{n} \nabla\left(\left[d_l\left(\left(rt_p(B_{ij_z})\right)^T e_q, \bar{x}_p^{(k)}(B_{ij_z})\right)\right]^l\right) \tag{4.44}$$

$$\underset{z\sim[3M]}{\nabla} mf_l^k(p) = 3M \sum_{q=1}^{n} \nabla\left(\left[d_l\left(\left(rt_p(B_{i_zj_z})\right)^T e_q, \bar{x}_p^{(k)}(B_{i_zj_z})\right)\right]^l\right) \tag{4.45}$$

$$\underset{z\sim[3Mn]}{\nabla} mf_l^k(p) = 3Mn\nabla\left(\left[d_l\left(\left(rt_p(B_{i_zj_z})\right)^T e_{q_z}, \bar{x}_p^{(k)}(B_{i_zj_z})\right)\right]^l\right) \tag{4.46}$$

Es wird also zuerst eine Zahl z aus einer Menge gezogen (z.B. $z \sim [M]$) und dann das dazu assoziierte Bündeltripel (4.44), Bündel (4.45) oder ein einzelner Strahl (4.46) ausgewählt. Die

Multiplikation mit der jeweiligen Größe M, $3M$ oder $3Mn$ trägt der Annahme Rechnung, dass der Erwartungswert des stochastischen Gradienten der Gradient ist (s.h. zum Beispiel [15] S.930 (2.1)).

$$\mathop{\mathbb{E}}_{z\sim[M]}\left[\mathop{\nabla}_{z} mf_l^k(p)\right] = \frac{1}{M}\sum_{j=1}^{M}\mathop{\nabla}_{j} mf_l^k(p) = \nabla mf_l^k(p) \tag{4.47}$$

Diese stochastischen Gradienten tragen also unterschiedlich viel Information über den tatsächlichen Gradienten und lassen sich schneller berechnen. Ein Nachteil ist, dass der negative stochastische Gradient nicht die Richtung des steilsten Abstiegs ist und noch nicht einmal eine Abstiegsrichtung sein muss.
Aufgrund der Implementierung der Meritfunktion 3.8 und der Verwendung von finiten Differenzen musste für die Berechnung der stochastischen Gradienten eine stochastische Meritfunktion implementiert werden. Die Auswertung des Gradienten über finite Differenzen stellt einige Herausforderungen für die Implementierung dar, da bei finite Differenzen die Meritfunktion mehrmals ausgewertet wird und es muss pro Auswertung die gleiche Zahl z gezogen werden. Dadurch gibt es einige Einschränkungen bezüglich der Meritfunktion, die optimiert werden kann. Der Strafterm 3.12 für die Verzeichnung ist nur für 4.44 sinnvoll, da es sein kann, dass das Bündel mit der Wellenlänge von grünem Licht bei den anderen Definitionen nicht ausgewählt wird. Da diese Wellenlänge für $k = 2$ auch der Referenzwert für die Abstandsberechnung ist kann 4.45 und 4.46 nur für $k = 1$ berechnet werden. Diese Schwierigkeiten können mit `pyrate` gar nicht oder nur sehr umständlich gelöst werden, da die Objekte die die verschiedenen Bestandteile eines optischen Systems repräsentieren für diese Anwendung nicht ausgelegt sind.
Die Definition 4.46 wurde bei den Algorithmen nicht verwendet, da sie aufgrund der Implementierung keine Vorteile gegenüber 4.44 bzw. 4.45 besitzt und für die Optimierung eines optischen Systems sehr problematisch ist. Die Strafterme können nicht verwendet werden und ein stochastischer Optimierungsalgorithmus muss sehr viele Iterationen durchführen um ein Minimum zu finden. Es sind auch noch weitere stochastische Gradienten denkbar, wie zum Beispiel

$$\mathop{\nabla}_{z\sim[n]} mf_l^k(p) = n\sum_{j=1}^{M}\sum_{i=1}^{3}\nabla\left(\left[d_l\left(\left(rt_p(B_{ij})\right)^T e_z, \bar{x}_p^{(k)}(B_{ij})\right)\right]^l\right).$$

Es wird also ein Strahl für jedes Bündel gezogen, sodass jede Wellenlänge i und alle übrigen Bündelparameter j betrachtet werden. Wegen `pyrate` kann diese Möglichkeit aber nicht effizient umgesetzt werden.

4.5.2 Stochastic gradient descent

Nachfolgend soll nun der sgd Algorithmus und seine direkten Abwandlungen vorgestellt werden. Die Algorithmen sind aus [13].

Algorithm 6 sgd

```
1: function SGD(f, x_0, η, γ ∈ (0,1), methods, gtol, maxiter)
2:     k ← 0, x_{k-1} ← x_0, v_{k-1} ← 0
3:     while True do
4:         k ← k + 1
5:         if (methods == vanilla) then
6:             v_k ← η ∇_{z∼[M]} f(x_{k-1})
7:         end if
8:         if (methods == momentum) then
9:             v_k ← γ v_{k-1} + η ∇_{z∼[M]} f(x_{k-1})
10:        end if
11:        if (methods == nag) then
12:            v_k ← γ v_{k-1} + η ∇_{z∼[M]} f(x_{k-1} − γ v_{k-1})
13:        end if
14:        x_k ← x_{k-1} − v_k
15:        if (k ≥ maxiter) or (‖(∇f(x_k)‖ ≤ gtol) then
16:            break
17:        end if
18:    end while
19:    return x_k
20: end function
```

Der Algorithmus erhält einen Parameter `methods` mit dem die gewünschte Methode ausgewählt werden kann. Dabei ist der unveränderte sgd (`vanilla`), der sgd mit Impuls (`momentum`) und der `nesterov accelerated gradient` (`nag`) implementiert worden. Alle drei Algorithmen basieren auf dem *gradient descent* Standardalgorithmus. Im `vanilla` wird nur anstatt dem Gradienten der stochastische Gradient berechnet. Das Verhalten des Verfahrens sollte also dem des *gradient descent* ähneln. Beim (`momentum`) wird noch zusätzlich ein mit γ gewichteter Term v_{k-1} addiert. Dieser trägt Information über die früheren Gradienten und bringt somit zur Iteration k ein gewissen Impuls mit. Dadurch soll das bekannte Zick-Zack Verhalten des *gradient descent* geglättet werden. Die Iterationsvorschrift des `nag` wiederum ist eine Abwandlung der Vorschrift des `momentum`-Algorithmus. Hierbei wird der stochastische Gradient aber nicht bei x_{k-1} ausgewertet, sondern bei einer gedämpften vorherigen Iteration.
Die Wahl von γ ist sehr stark davon abhängig, wie sich die Funktion f global verhält und wie der Startwert x_0 gewählt wird. Befindet sich x_0 in einem Bereich in dem der Gradient sehr große Einträge hat dann sollte $\gamma \leq 0.5$ gewählt werden. Kommt die Folge $\{x_k\}_{k=1}^{K}$ dann in einen Bereich von f in dem der Gradient um mehrere Größenordnungen kleiner ist, wie z.B. in Abb. 3.1 zu sehen, dann ist die Wahl von $\gamma \leq 0.5$ zu gering und das Verhalten des Algorithmus ähnelt dem von `vanilla`. Soll `momentum` bzw. `nag` in den flacheren Gebieten von f ihre Vorteile gegenüber `vanilla` ausnutzen muss $\gamma > 0.5$ gewählt werden. Unter der Annahme, dass der Gradient bei $\{x_k\}_{k=c}^{K}, K - c$ klein genug, um mehrere Größenordnungen kleiner ist als der Gradient an den Stellen $\{x_k\}_{k=1}^{c-1}$ und die Schrittweite durch die ersten Gradienten bestimmt

wird, ist die Wahl von $\gamma > 0.5$ sinnvoll, da beim `momentum`-Verfahren gilt

$$\begin{aligned} v_K &= \sum_{i=1}^{K} \gamma^{K-i} \eta \nabla_{z\sim[M]} mf_l^k(x_{i-1}) \\ &= \underbrace{\gamma^{K-1}\eta \nabla_{z\sim[M]} mf_l^k(x_0) + \ldots + \gamma^{K-c-1}\eta \nabla_{z\sim[M]} mf_l^k(x_{c-2})}_{\text{sehr klein wegen } \gamma} \\ &+ \underbrace{\gamma^{K-c}\eta \nabla_{z\sim[M]} mf_l^k(x_{c-1}) + \ldots}_{\text{sehr klein wegen } \eta\text{, durch } \gamma \geq 0.5 \text{ aber höher gewichtet}} + \eta \nabla_{z\sim[M]} mf_l^k(x_K). \end{aligned}$$

Das global stark variierende Verhalten der Meritfunktion 3.8 ist für die Algorithmen und insbesondere für die Parameterwahl problematisch. Ein weiteres Problem ist, wie bereits unter (4.2) angemerkt, dass der Gradient sich in zwei Teile aufspaltet. Zum einen die Ableitungen nach den Linsenpositionen und zum anderen die, relativ dazu sehr großen Ableitungen nach den Krümmungen. Dadurch wird erwartet, dass der Algorithmus und auch die folgenden Algorithmen keine nennenswerten Änderungen in den Linsenpositionen erreichen.
Die Abbruchbedingung besteht aus zwei Teile. Zum einen soll abgebrochen werden wenn die maximale Iterationsanzahl `maxiter` erreicht wurde. Zum anderen wird mit der Norm des echten Gradienten eine Extremstelle detektiert, falls diese eine gewünschte Toleranz unterschritten hat. Diese Abbruchbedingungen wurden in allen folgenden Algorithmen verwendet.

4.5.3 Adam und Adamax

Der folgende Algorithmus stammt aus [7] und soll die Schrittweite adaptiv anpassen. Dabei wird ein erster m_{k-1} und zweiter v_{k-1} Momententerm in die Iteration mit aufgenommen. Der Name Adam kommt von *adaptive moment estimation* und der Algorithmus kombiniert die Vorteile von AdaGrad (Duchi et al., 2011), der später noch vorgestellt wird und RMSProp (Tieleman und Hinton, 2012). Eine Eigenschaft des Adam ist, dass die Schrittweite approximativ durch die übergebene Schrittweite η beschränkt ist (s.h. [7] S.3 ff).

Algorithm 7 Adam

function ADAM($f, x_0, \eta, \beta_1 \in (0,1), \beta_2 \in (0,1), \epsilon \approx 10^{-8}, gtol, maxiter$)
  $k \leftarrow 0$, $x_{k-1} \leftarrow x_0$, $m_{k-1} \leftarrow \mathbf{0}$, $v_{k-1} \leftarrow \mathbf{0}$
  while True **do**
    $k \leftarrow k + 1$
    $g_k \leftarrow \nabla_{z\sim[M]} f(x_{k-1})$
    $m_k \leftarrow \beta_1 m_{k-1} + (1-\beta_1) g_k$
    $v_k \leftarrow \beta_2 v_{k-1} + (1-\beta_e) g_k \odot g_k$
    $\hat{m}_k \leftarrow \frac{1}{1-\beta_1^k} m_k$
    $\hat{v}_k \leftarrow \frac{1}{1-\beta_2^k} v_k$
    $x_k \leftarrow x_{k-1} - \eta(\mathrm{diag}(\hat{v}_k)^{0.5} + \epsilon I)^{-1} \hat{m}_k$
    if ($k \geq maxiter$) or ($\|(\nabla f(x_k)\| \leq gtol$) **then**
      break
    end if
  end while
  return x_k
end function

Im referenzierten Paper wird ein Konvergenzresultat (Theorem 4.1) bewiesen. Dabei sind die Voraussetzungen unter anderem, dass $\beta_{1,k} = \beta_1 \lambda^{k-1}, \lambda \in (0,1)$ ist und $\eta_k = \frac{\eta}{\sqrt{k}}$. Ob diese

Wahl die Konvergenz verbessert, wird in der Auswertung genauer diskutiert. Des Weiteren ist eine a-priori Wahl der Parameter nicht mehr so einfach möglich, da das Verhalten des Terms $\eta(\text{diag}(\hat{v}_k)^{0.5} + \epsilon I)^{-1}\hat{m}_k$ nicht mehr so einfach abgeschätzt werden kann.
Im Adam ist der Update-Schritt invers proportional zu einer Art skalierten L_2-Norm des momentanen und der früheren Gradienten. Dies kann auf die L_p-Normen verallgemeinert werden, jedoch hat sich herausgestellt, dass die Algorithmen instabil werden. Nur die L_2- und die L_∞-Norm ergeben stabile Verfahren (s.h. [7] Kaptiel 7.1). Im Folgenden ist der Adam verallgemeinert auf die L_∞-Norm dargestellt. Das resultierende Verfahren heißt Adamax.

Algorithm 8 Adamax

function ADAMAX($f, x_0, \eta, \beta_1 \in (0,1), \beta_2 \in (0,1), gtol, maxiter$)
  $k \leftarrow 0$, $x_{k-1} \leftarrow x_0$, $m_{k-1} \leftarrow \mathbf{0}$, $v_{k-1} \leftarrow \mathbf{0}$
  while True **do**
    $k \leftarrow k + 1$
    $g_k \leftarrow \underset{z\sim[M]}{\nabla} f(x_{k-1})$
    $m_k \leftarrow \beta_1 m_{k-1} + (1-\beta_1) g_k$
    $v_k \leftarrow \max\{\beta_2 v_{k-1}, |g_k|\}$ # eintragsweise
    $x_k \leftarrow x_{k-1} - \frac{\eta}{1-\beta_1^k}(\text{diag}(v_k))^{-1} m_k$
    if $(k \geq maxiter)$ or $(\|(\nabla f(x_k)\| \leq gtol)$ **then**
      break
    end if
  end while
  return x_k
end function

4.5.4 Adagrad

Der bereits erwähnte Adagrad-Algorithmus [3] basiert darauf, den momentanen Gradienten wieder mit dem Inversen der früheren Gradienten zu skalieren. Der Nachteil ist, dass die Änderung von x_k für k hinreichend groß gegen Null geht. Dies ist problematisch, wenn der Startwert x_0 in einem Gebiet von f liegt in dem die Einträge des Gradienten sehr groß sind. Die Schrittweite kann dadurch größer gewählt werden, da mit dem Inversen skaliert wird. Gelangt der Algorithmus in ein Gebiet in dem der Gradient dann sehr klein ist, macht die Iterierte x_k kaum noch Fortschritte und die Toleranz *gtol* kann nicht mehr erreicht werden.

Algorithm 9 Adagrad

function ADAGRAD($f, x_0, \eta, \epsilon \approx 10^{-8}, gtol, maxiter$)
  $k \leftarrow 0$, $x_{k-1} \leftarrow x_0$, $G \leftarrow \mathbf{0}$
  while True **do**
    $k \leftarrow k + 1$
    $g_k \leftarrow \underset{z\sim[M]}{\nabla} f(x_{k-1})$
    $G \leftarrow G + \text{diag}(g_k \odot g_k)$
    $x_k \leftarrow x_{k-1} - \eta(G + \epsilon I)^{-0.5} g_k$
    if $(k \geq maxiter)$ or $(\|(\nabla f(x_k)\| \leq gtol)$ **then**
      break
    end if
  end while
  return x_k
end function

4.5.5 Adadelta

Der Adadelta-Algorithmus soll die Schrittweite wieder adaptiv anpassen, in dem er nur die Information des Gradienten nutzt. Im Machine Learning Kontext soll der Algorithmus robust gegenüber Veränderungen der Hyperparameter sein [18]. Hyperparameter sind Größen die der Algorithmus als Eingabe bekommt und die nicht optimiert werden. Oft muss die Schrittweite und die anderen Hyperparameter in einer Tuningphase so lange angepasst werden, bis der Algorithmus ein gewünschtes Konvergenzverhalten zeigt. Dies soll Adadelta vereinfachen. Hierbei gibt es nur zwei Parameter ρ sowie ϵ und keine Schrittweite. Des Weiteren soll das Verfahren robust gegenüber sehr großen Gradienten sein. Die Idee ist die Schrittweite in der Nähe eines Minimums kleiner werden zu lassen, um das Hin- und Herspringen um das Minimum herum zu dämpfen. Des Weiteren wird die Update-Regel in [18] Kapitel 3.2 so hergeleitet, dass eine Approximation der Diagonalen der Hessematrix verwendet wird. Dies wurde nicht überprüft.

Algorithm 10 Adadelta

function ADADELTA($f, x_0, \rho \in (0,1), \epsilon \approx 10^{-8}, gtol, maxiter$)
  $k \leftarrow 0$, $x_{k-1} \leftarrow x_0$, $E_{g,k-1} \leftarrow \mathbf{0}$, $E_{x,k-1} \leftarrow \mathbf{0}$
  while True **do**
    $k \leftarrow k + 1$
    $g_k \leftarrow \nabla_{z \sim [M]} f(x_{k-1})$
    $E_{g,k} = \rho E_{g,k-1} + (1-\rho) g_k \odot g_k$
    $RMS_g = \sqrt{E_{g,k} + \epsilon \mathbf{1}}$ # eintragsweise
    $RMS_x = \sqrt{E_{x,k-1} + \epsilon \mathbf{1}}$ # eintragsweise
    $\Delta x_k = \mathrm{diag}(RMS_g)^{-1}(g_k \odot RMS_x)$
    $x_k \leftarrow x_{k-1} - \Delta x_k$
    $E_{x,k} = \rho E_{x,k-1} + (1-\rho) \Delta x_k \odot \Delta x_k$
    if $(k \geq maxiter)$ or $(\|(\nabla f)(x_k)\| \leq gtol)$ **then**
      break
    end if
  end while
  return x_k
end function

5 Auswertung

5.1 Systeme

5.1.1 4-Linser-System

Dieses optische System basiert auf dem *Monochromatischen Quartett* aus dem Jahr 1990. Dieses besteht aus ingesamt vier Linsen, welche auf der Achse eine Mindestdicke von 2mm haben müssen. Zudem muss der Abstand zwischen den Linsen auf der Achse mindestens 0mm betragen, d.h. diese dürfen sich berühren. Die Brennweite soll 100mm betragen und der Durchmesser der Eintrittspupille (100/3)mm.
In Abbildung 5.1 ist eine einfache Form dieses Systems dargestellt, welches als Ausgangsystem für Optimierungsverfahren verwendet werden kann. Dieses besteht aus drei Linsen mit Planflächen und einer Linse mit gekrümmter Fläche. Die Blende darf hier auch im System liegen. Die zu optimierenden Parameter lauten wie folgt:

- Krümmung aller 8 Flächen (S1 - S8)
- Abstand Blende - Fläche 1 (S1) auf der Achse
- Dicken der 4 Linsen $\geq$ 2mm
- 3 Abstände zwischen den Linsen auf der Achse $\geq$ 0mm
- Abstand Fläche 8 (S8) - Bildfläche auf der Achse

Dies führt zu einem Optimierungsproblem mit insgesamt 17 Dimensionen.
Alle Linsen sollen aus dem Glas *BK7* bestehen. Das System soll dabei in Bezug auf folgende Lichtbündel optimiert werden.

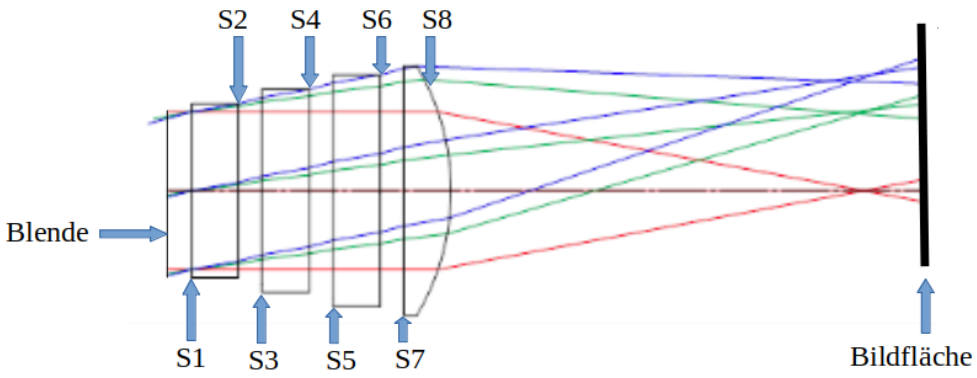

Abbildung 5.1: 4-Linser-System

startx	starty	startz	radius [mm]	anglex [°]	angley[°]	raster
0	0	0	16	0	0	rechteckiges Raster
0	0	0	16	10.5	0	rechteckiges Raster
0	0	0	16	15	0	rechteckiges Raster

Tabelle 5.1: Definition Lichtbündel

Somit gibt es drei unterschiedliche Lichtbündel. Die jeweils ersten drei Komponenten beschreiben den Startpunkt des Bündels im System. Dieser geht bei allen durch den Ursprung. Der Radius der Bündel ist bei allen 16mm. Dann wird definiert, unter welchem Winkel die Bündel in das System gelangen. Das Raster beschreibt, wie die einzelnen Strahlen im Bündel verteilt sind. Jedes Bündel besteht jeweils aus 20 Strahlen. Zudem ist jedes Bündel für die drei Wellenlängen *550nm, 650nm* und *480nm* definiert, weshalb man am Ende auf insgesamt $3 * 3 = 9$ Bündel kommt.
Die Sollbildhöhe (SBH) berechnet sich für die drei Bündel wie folgt.

$$\begin{aligned} &\text{anglex} = 0° : \quad \text{SBH} = f * \tan(0°) = 100\text{mm} * \tan(0°) = 0\text{mm} \\ &\text{anlgex} = 10.5° : \quad \text{SBH} = f * \tan(10.5°) = 100\text{mm} * \tan(10.5°) = 18.534\text{mm} \\ &\text{anglex} = 15° : \quad \text{SBH} = f * \tan(15°) = 100\text{mm} * \tan(15°) = 26.795\text{mm} \end{aligned} \tag{5.1}$$

5.2 Auswertung Particle-Swarm-Optimization mit Nelder-Mead

Verlauf des besten gefundenen Meritwerts über Iterationen
In einem ersten Schritt sollte das Verhalten des Algorithmus' am Beispiel des *4-Linser-Systems* untersucht werden. Dazu wurde zu jeder Iteration der aktuell beste gefundene Zielfunktionswert (f_{best}) aller Partikel bei der Lösung x_{best} aufgenommen. Zudem wurde kein Abbruchkriterium neben der maximalen Iterationsanzahl (1000 Iterationen) verwendet, um aus den Ergebnissen sinnvolle Bedingungen für das Terminieren des Algorithmus' herzuleiten. Alle Parameter wurden auf Werte aus der Literatur gesetzt und sind im folgenden definiert:

Variante: 1; maxIt = 1000; $c_1 = c_2 = 2$*;*
maxIt-Nelder-Mead = 100; tol-Nelder-Mead = 1e-6; N (Anzah der Partikel) $= 21n + 1 = 358$
(n: Dimension des OP)

Bei der Meritfunktion gibt es jedoch eine Abweichung in Bezug auf den Strafterm 3.9. Um das ankommen zu weniger Strahlen noch mehr zu bestrafen, wurde das Ergebnis aus Gleichung 3.9 noch mit dem Faktor 10^4 multipliziert. In dieser Auswertung soll weniger auf die absoluten Zahlen der Meritfunktion nach der Optimierung eingegangen werden, sondern mehr auf das Verhalten des Algorithmus'. Dadurch lassen sich allgemeine Aussagen treffen, welche dann auch auf andere Definitionen der Meritfunktion übertragen werden können. Der Optimierungsalgorithmus wurde drei mal mit denselben Parametern auf das System angewendet, da das Verfahren zum Großteil auf stochastischen Entscheidungen beruht.
Für 1000 Iterationen benötigte der Algorithmus je Lauf ca. 532.120 Funktionsauswertungen und dafür ungefähr 8 Stunden. Das bedeutet in einer Iteration muss die Meritfunktion durchschnittlich 532 mal ausgewertet werden.

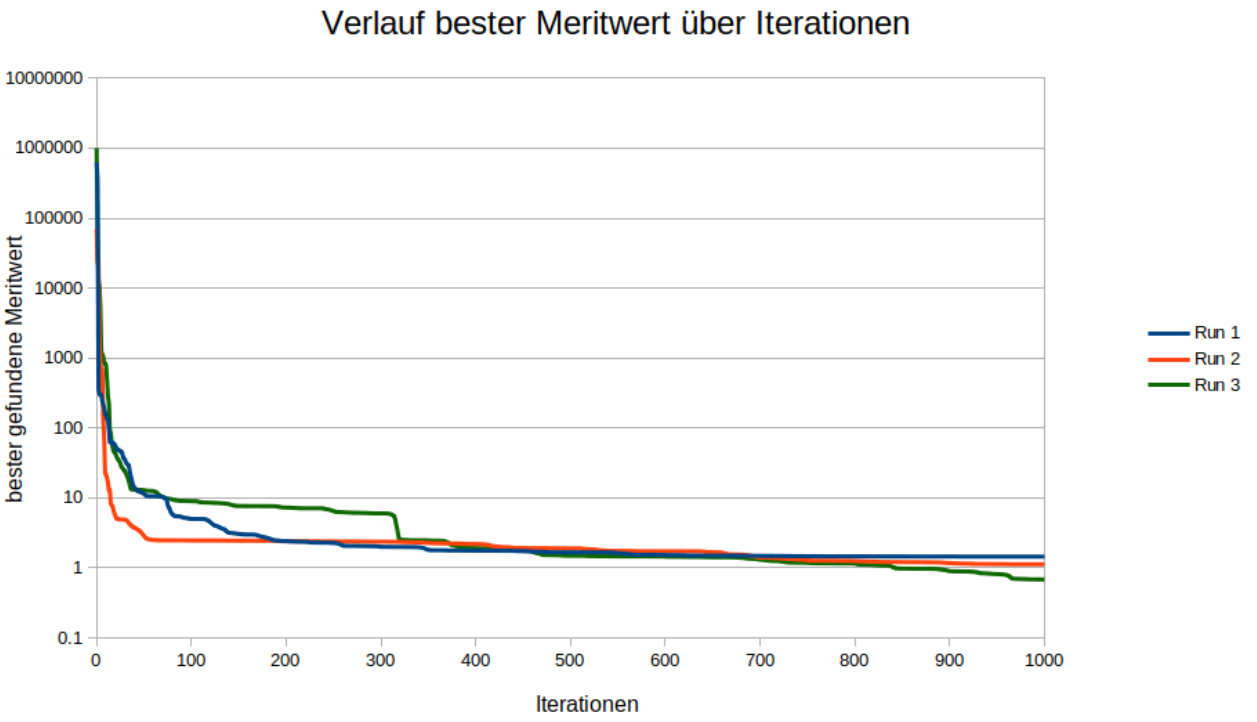

Abbildung 5.2: Verlauf bester gefundene Meritwert über Iterationen

Die finalen Meritwerte f_{best} nach 1000 Iterationen liegen bei den drei Läufen in einem ähnlichen Bereich. So erhält man die Lösungen: $f_{best}^{Run1} = 1,44; f_{best}^{Run2} = 1,12; f_{best}^{Run3} = 0,68$. Dies deutet darauf hin, dass durch den Algorithmus mit hoher Wahrscheinlichkeit in verschiedenen Läufen ähnlich gute Ergebnisse erzielt werden können (bezogen auf finalen Meritwert f_{best}). In Abbildung 5.2 ist das Verhalten der drei Durchläufe (*Runs*) dargestellt. Die y - Achse ist logarithmisch abgebildet. Zu Beginn ist die beste gefundene Lösung f_{best} bei allen Läufen sehr groß (Größenordnung 1e6). Dies liegt an der zufälligen Verteilung der Partikel im zulässigen Bereich. Hierbei landen alle Individuen auf Positionen, welche zwar die Intervallgrenzen erfüllen, jedoch durch die Randbedingungen unzulässig sind (Dazu zählen: System ist unphysikalisch, es kommen nicht alle Strahlen auf der Bildfläche an, Sollbildhöhen werden nicht eingehalten). Somit sind auch die Strafterme in der Meritfunktion sehr hoch. Jedoch zeigt sich, dass f_{best} in nur wenigen Iterationen deutlich kleiner wird. Somit werden die Partikel sehr schnell in Gebiete gezogen, in welchen zulässige und physikalisch umsetzbare Lösungen liegen. Nach diesem starken Abfall, welcher nach ca. 100 bis 200 Iterationen abgeschlossen ist, nimmt die Verbesserung pro Iteration deutlich ab.

In Abbildung 5.3 ist der Bereich zwischen Iteration 400 und 1000 vergrößert dargestellt. Wichtig ist hierbei zu erkennen, dass f_{best} pro Iteration zwar nur sehr wenig abnimmt, das System jedoch trotzdem stetig in kleinen Schritten verbessert wird. Dies zeigt sich vor allem bei *Run 2* und *Run 3*. Hier wäre vermutlich bei noch längerer Laufzeit ein noch besseres Ergebnis erzielt worden. Spannend ist dabei vor allem das Verhalten von *Run 3*. Zwischen Iteration 480 und 700 kann kaum eine Verbesserung erzielt werden (220 Iterationen). Dann kommt es jedoch erneut zu einer stetigen Verbesserung. Dies erschwert die Definition eines Abbruchkriteriums. Man darf den Algorithmus nicht nach einer Stagnationsphase abbrechen. *Run 1* zeigt jedoch ein anderes Verhalten. Hier verbessert sich der Meritwert selbst über 500 Iterationen nur minimal. Hier wäre ein vorzeitiges terminieren angebracht. Ein weiteres interessantes Kriterium

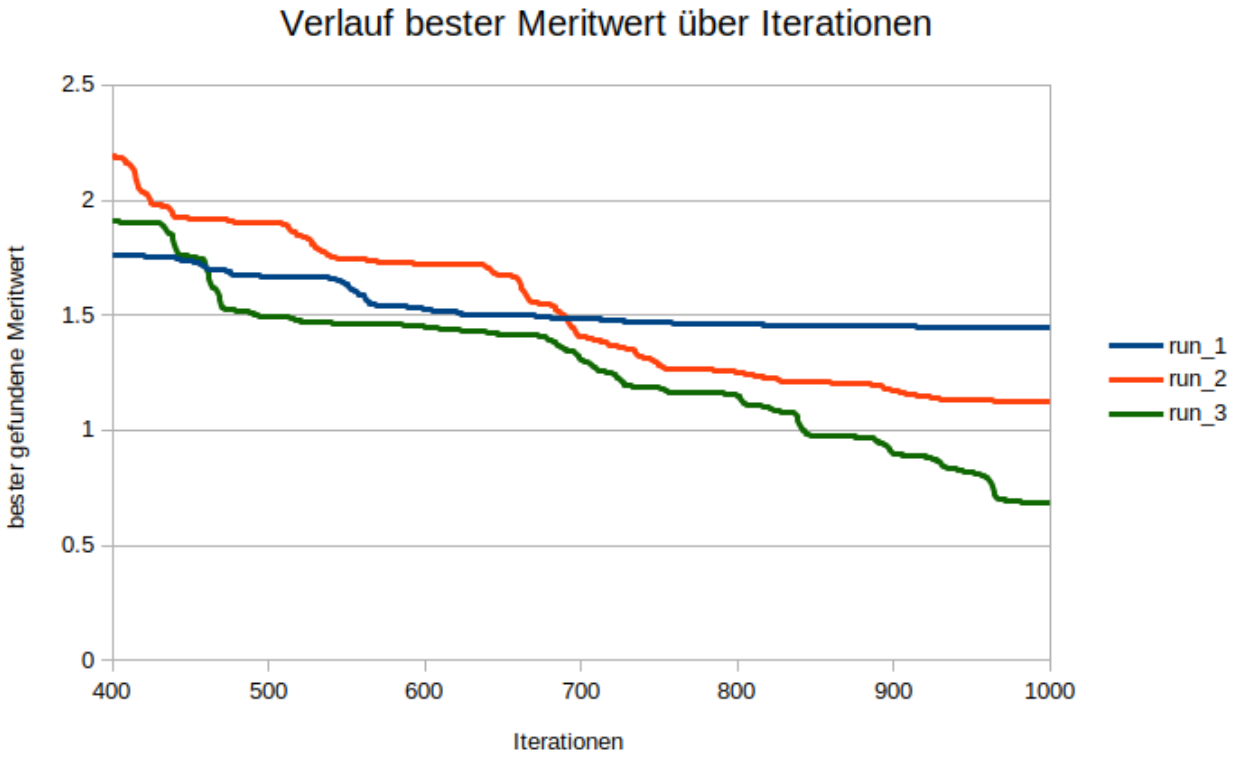

Abbildung 5.3: Verlauf bester gefundene Meritwert über Iterationen - vergrößerter Ausschnitt

ist die Norm des Gradienten der Meritfunktion an den finalen besten Lösungen x_{best} nach 1000 Iterationen. Dieser liegt bei *Run 1* bei 320, bei *Run 2* bei 166 und bei *Run 3* bei 850. *Run 1* und *Run 2* haben im Vergleich zu *Run 3* einen kleineren Gradienten. Da diese auch in ihrem Verlauf zum Ende hin ein stagnierendes Verhalten zeigen, lässt sich vermuten, dass die gefundenen Lösungen schon im Bereich von Minima sind. Wenn sich nun alle oder zumindest der Großteil der Partikel in der Nähe von diesen Minima befinden, wäre es eventuell sinnvoll, den Hauptalgorithmus zu beenden und x_{best} als Startwert für einen effizienteren lokalen Suchalgorithmus zu verwenden, welcher eine Lösung findet, die noch näher am Minimum liegt. Dies kann jedoch nur sehr schwer ermittelt werden, denn es könnte auch sein, dass nahe bei diesem Minimum noch ein besseres liegt, welches in weiteren Iterationen von den Partikeln entdeckt werden könnte. Somit darf der Gradient allein keine Bedingung für einen Abbruch sein. Da der Algorithmus global nach Lösungen sucht, kann er auch aus lokalen Minima wieder ausbrechen und bessere finden.

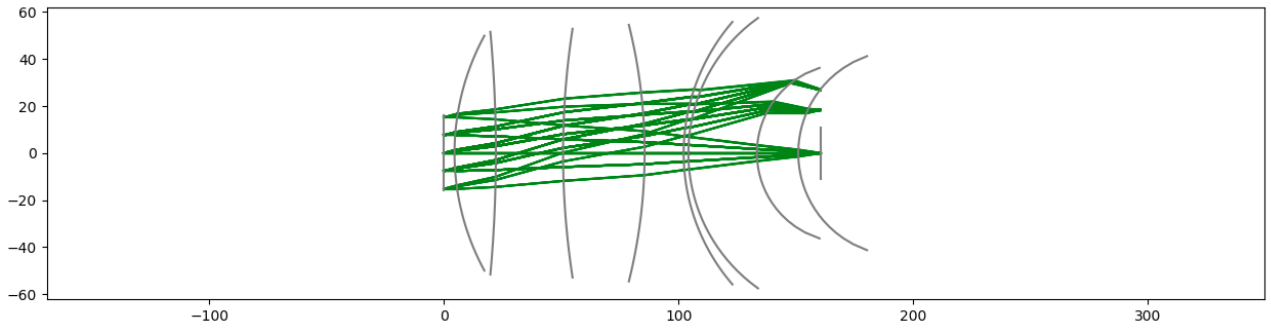

Abbildung 5.4: Optimiertes System - Run 1

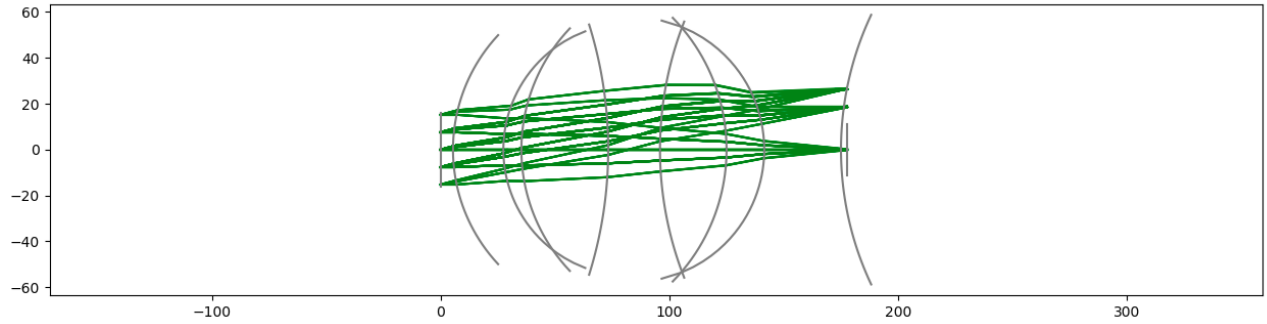

Abbildung 5.5: Optimiertes System - Run 3

In den Abbildungen 5.4 und 5.5 sind zwei optimierte Systeme von Run1 und Run3 dargestellt. Die grünen Linien symbolisieren die Strahlen der drei verwendeten Lichtbündel. Diese gehen von der Blende aus (kleiner senkrechter Strich links) durch die Linsen zur Bildfläche (kleiner senkrechter Strich rechts). Dazwischen befinden sich die 8 Linsenflächen, wobei jeweils 2 aufeinanderfolgende Linsenflächen eine Linse bilden. Die Räume dazwischen sind mit Luft gefüllt. Dabei unterscheiden sich die Systeme stark voneinander. Dies bestätigt, dass durch den Algorithmus viele unterschiedliche Resultate gefunden werden können. Bei *Run 1* befindet sich das optimierte Systeme aufgrund vorheriger Erkenntnisse vermutlich nahe an einem Minimum. Das System von *Run 3* ist zwar besser, jedoch ist es vermutlich noch weiter von einem Minimum entfernt und könnte durch weitere Iterationen noch verbessert werden.

Zusammenfassend hat sich gezeigt, dass mit dem *PSO-NM-Variante 1* Algorithmus gute Lösungen und auch eine weite Bandbreite an verschiedenen guten Systemen gefunden werden kann (bei mehrmaligem Start unterschiedliche Lösungen). Jedoch ist es aufgrund des stochastischen Verhaltens schwierig, ein gutes Abbruchkriterium zu ermitteln. Daher muss dieser im Idealfall bis zu einer maximalen Anzahl an Iterationen ausgeführt werden, um auszuschließen, dass keine verbesserte Lösung mehr gefunden werden kann. Außerdem hat sich gezeigt, dass der Algorithmus in allen Fällen zuerst sehr schnell in den Bereich von zulässigen Lösungen und dann mit hoher Wahrscheinlichkeit gegen zumindest lokale Minima konvergiert. Jedoch erzielt er pro Iteration nur sehr kleine Verbesserungen, was letztendlich dazu führt, dass man den Algorithmus lange laufen lassen muss um auszuschließen, dass man noch bessere Systeme findet. Dies führt letztendlich zu einer langen Laufzeit aufgrund der enormen Anzahl an Funktionsauswertungen (ca. 532.000 Funktionsauswertungen bei 1000 Iterationen). Eine Lösung wäre, den Algorithmus nach einer langen Stagnationsphase abzubrechen und die dabei erhaltenen meist

guten Lösungen x_{best} als Startwert für effizientere lokale Suchalgorithmen zu verwenden.

Reduktion der Funktionsauswertungen
Im nächsten Schritt soll untersucht werden, ob durch eine Parameterwahl, durch welche die Anzahl der Funktionsauswertungen reduziert wird, die Ergebnisse erheblich verschlechtert werden. Zwei Faktoren bestimmen maßgeblich die Anzahl der Funktionsauswertungen. Zum einen ist dies die Anzahl der Partikel, zum anderen die Parameter *maxIt-Nelder-Mead* und *tol-Nelder-Mead*. Da der Nelder-Mead in jeder Iteration ausgeführt wird, hat dieser enormen Einfluss auf die Gesamtanzahl an Funktionsauswertungen. Diese Parameter wurden wie folgt verändert.

	Run 1.1	Run 1.2	Run 2.1	Run 2.2	Run 3.1	Run 3.2
maxIt	500	500	500	500	500	500
N	86	86	86	86	256	256
maxIt-NM	20	20	70	70	70	70
tol-NM	1e-2	1e-2	1e-5	1e-5	1e-5	1e-5
Final-Merit	2,93	2,2	1,84	2,12	1,21	2,9
Gradient	1557	69	392	2059	377	241
fEval	54.216	49.416	76.173	59.133	141.290	136.876
Zeit [min]	48	44	63	48	115	115
fEval / It	108	98	152	118	282	273

Tabelle 5.2: Wahl der Parameter und Resultate

Dabei ist *maxIt* die Anzahl der Iterationen des Hauptalgorithmus', *N* ist die Anzahl der Partikel, *maxIt-NM* die maximale Anzahl an Iterationen von Nelder-Mead und *tol-NM* die dazugehörige Toleranz für das Abbruchkriterium. Zudem wurden bei jeder Parameterwahl zwei Läufe durchgeführt. Bei *Run 1.1/1.2* wurden alle Parameter relativ klein gewählt. Bei *Run 2.1/2.2* wurde die Genauigkeit und die Anzahl der Iterationen von Nelder-Mead erhöht im Vergleich zu Lauf 1. Bei Run 3 wurde zusätzlich noch die Anzahl der verwendeten Partikel deutlich erhöht. Obwohl sich die Parameter stark unterscheiden, unterscheiden sich die finalen Meritwerte nicht gravierend. Jedoch zeigt sich ein leichter Trend. So sind die Meritwerte von Lauf 1 tendenziell etwas höher als die von Run 2 und Run 3. Die beste Lösung wurde bei Run 3.1 mit 1,21 erreicht. Jedoch wurde beim Wiederholungslauf mit den gleichen Parametern eine schlechtere Lösung erzielt. Dies verdeutlicht nochmal das stochastische Verhalten, was es schwierig macht, konkrete Aussagen über die Auswirkung von Parametern zu treffen. Zudem muss man die Ergebnisse mit Vorsicht betrachten. Die bei Lauf 1 erzielten Ergebnisse erhielt man nach ca. 50.000 Meritfunktionsauswertungen. Die von Run 3 hingegen benötigten fast 100.000 mehr. Dies liegt an der höheren Anzahl an Partikeln und der längeren Laufzeit des Nelder-Mead Algorithmus (führt zu mehr Funktionsauswertungen pro Iteration). Auch interessant ist der Wert des Gradienten an der Lösung. Dieser ist bei beiden Wiederholungen von Lauf 3 relativ klein. Dies deutet darauf hin, dass man nach mehr Funktionsauswertungen mit höherer Wahrscheinlichkeit in der Nähe eines Minimums landet.

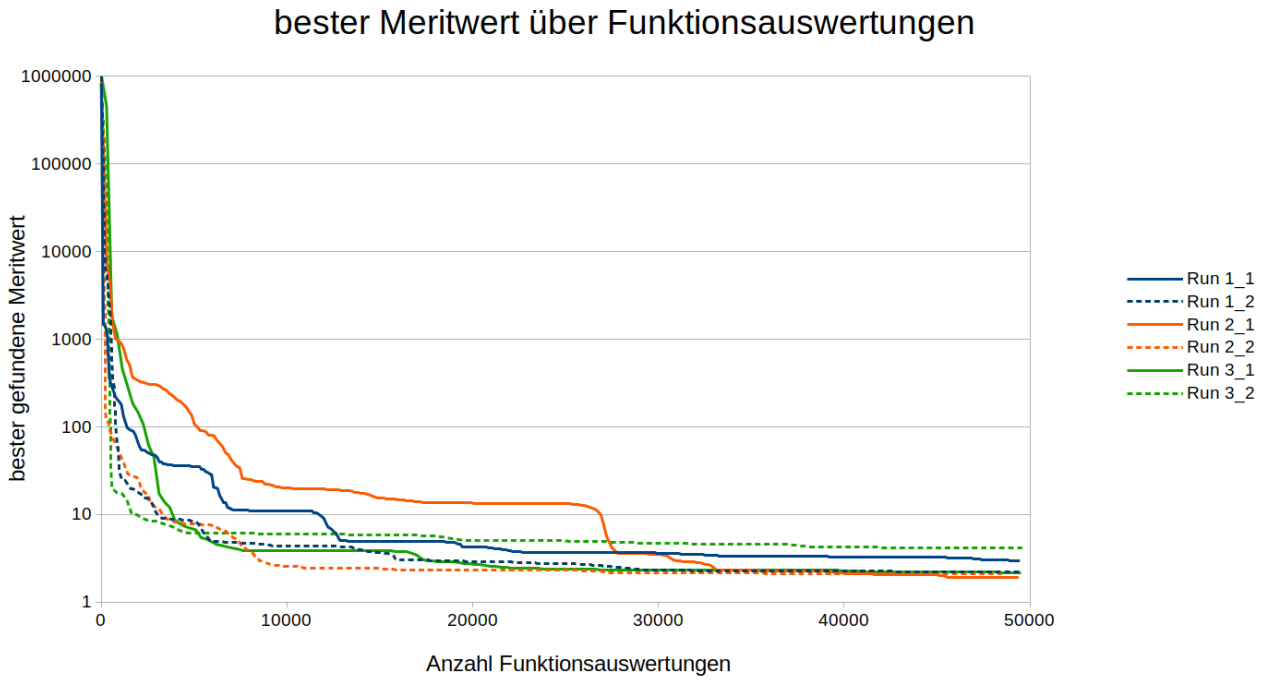

Abbildung 5.6: bester gefundene Meritwert über Funktionsauswertungen - PSO-NM

In Abbildung 5.6 wird der Zusammenhang zwischen Funktionsauswertungen und f_{best} verdeutlicht. Run 3 braucht aufgrund seiner Parameter für die vorgegebenen 500 Iterationen deutlich mehr Funktionsauswertungen als Run 2 und Run 3. Somit verwundert es auch nicht, dass man bei Run 3 die besten Resultate erhält. Hier wird nun f_{best} in Abhängigkeit der dazu benötigten Anzahl an Funktionsauswertungen gezeigt. Dabei zeigt sich, dass beim Großteil der Läufe die besten Meritwerte etwa gleich schnell verbessert werden und in den zulässigen Bereich wandern. Run 1.2 und Run 3.2 zeigen dabei fast das gleiche Verhalten, obwohl sich deren Parameter stark unterscheiden. Wenn man also den Algorithmus mit wenig Partikeln und geringer Genauigkeit bei Nelder-Mead mehr Iterationen laufen lässt als mit vielen Partikeln und höherer Genauigkeit (also insgesamt gleiche Anzahl an Funktionsauswertungen), kann man zumindest in diesem Beispiel ungefähr ähnliche Ergebnisse erwarten. Somit ist es für das Verfahren an sich nicht von Bedeutung, welche der beiden Varianten gewählt wird (zumindest für den Startbereich, in welchem die Lösungen in den zulässigen Bereich gehen). Nachdem die Lösungen im niedrigeren Bereichen liegen (Meritwert zwischen 1 und 50), geht die Verringerung von f_{best} pro Funktionsauswertung bei allen Läufen stark zurück. Auf den ersten Blick scheinen die Kurven auf dem gleichen Meritwert zu stagnieren. Die Meritwerte bei diesen Plateus unterscheidet sich dabei zwischen den Kurven. Dabei kann man jedoch keinen Zusammenhang im Bezug auf die Wahl der Parameter erkennen. Auch hängt das weitere Verhalten der Kurven scheinbar nicht mit den Parametern zusammen. Bei allen gibt es eine etwa gleich große negative Steigung mit kleinen auftretenden Sprüngen, welche f_{best} mal stärker und mal weniger verbessern. Die unterschiedlichen Höhen dieser Plateus hängen vermutlich mit dem Verhalten der Partikel zusammen. Diese landen während des Verfahrens in unterschiedlich guten Regionen. Dies hängt in diesem Beispiel jedoch augenscheinlich nicht mit der Wahl der Parameter zusammen. Der Verlauf von f_{best} hängt dabei hauptsächlich von der Anzahl der Funktionsauswertungen ab. Zudem lässt sich erkennen, dass die Kurven nicht völlig stagnieren. So hat Run 3.1 nach den 50.000 Funktionsauswertungen einen Wert von $f_{best} = 2,14$ (dies entspricht etwa Iteration

180). Bis zur Iteration 500 sinkt der Wert jedoch noch auf 1,21. Dies verdeutlicht nochmal das Verhalten des Algorithmus', wie es schon beschrieben wurde. Das ändert sich auch nicht durch Veränderung der hier untersuchten Parameter.

Benchmark
Zuletzt wird noch eine Benchmark zwischen den verschiedenen Varianten des *PSO-NM* Algorithmus durchgeführt. Zwischen den vier Varianten unterscheidet sich die Art und Weise des PSO-Updates. Bei Variante 2 orientieren sich die Partikel am global Besten und zusätzlich an der von ihnen bislang besten gefundenen Lösung. Dadurch soll die Gefahr einer frühzeitigen Konvergenz verringert werden. Variante 4 ist vom Prinzip gleich wie Variante 1. Hier werden jedoch im PSO-Schritt alle Partikel upgedatet. Bei Variante 3 werden die Partikel in feste Nachbarschaften (S Stück) eingeteilt. Dabei orientieren sich die Partikel nur an der global besten Lösung ihrer Nachbarschaft und an ihrer eigenen besten gefundenen Lösung. Dadurch soll auch eine frühzeitige Konvergenz verhindert werden und zudem die globale Suche ausgeweitet werden. Für die *PSO-NM* Algorithmen wurden folgende Parameter verwendet:

maxIter = 1000; maxIter-NM = 50; tol-NM = 1e-3;
stopNumber = 70; stopTol = 0.01
$S = 6$

Das bedeutet, wenn sich der beste globale Meritwert f_{best} nach 70 Iterationen nicht um 1% verbessert hat (d.h. $f_{best}(t-70) - f_{best}(t) < 0.01 \cdot f_{best}(t-70)$), wird der Algorithmus abgebrochen. Dies führt zu folgenden Resultaten. Der Parameter S gibt die Anzahl der Nachbarschaften bei Variante 3 an.

	Variante 1	**Variante 2**	**Variante 3**	**Variante 4**
Final Merit	1,58	3,85	1,44	5,13
Norm Gradient	2100	4000	1900	41.500
Iterationen	993	314	342	237
f-Eval	138.750	50.000	50.400	89.000
Zeit [min]	113	41	41	74

Dabei erhält man bei Variante 1 und Variante 3 die besten Ergebnisse. Zudem ist bei diesen auch der Gradient an der finalen Lösung am niedrigsten. Somit könnten die Lösungen nahe an einem Minimum sein. Auffällig ist dabei, dass bei Variante 1 fast die gesamten 1000 Iterationen ausgeführt werden. Hier findet also eine eher langsame Konvergenz statt, welche jedoch eine gute Lösung findet. Bei diesem Beispiel erhielt man bei Variante 3 das beste Ergebnis. Somit scheint es sinnvoll zu sein, die Partikel in Gruppen aufzuteilen, welche relativ unabhängig voneinander agieren und somit frühzeitige Konvergenz gegen gute, aber nicht optimale Lösungen, verhindern. Da dann jedoch nicht die Gesamte Population, sondern nur Partikel aus der jeweiligen Nachbarschaft die global beste gefundene Region nach noch besseren Lösungen absuchen, erschwert dies die lokale Suche. Dies zeigt sich auch daran, dass der Algorithmus nach schon ca. 300 Iterationen abbricht, da über 70 Iterationen keine nennenswerte Verbesserung mehr gefunden werden konnte. Hier wäre es sinnvoll, die final gefundene Lösung als Startwert für einen lokalen Suchalgorithmus zu verwenden. Das schlechteste Ergebnis erhielt man bei Variante 4. Hier ist die Norm des Gradienten noch sehr groß (ca.41.000). Somit ist diese Lösung noch weit von einem Optimum entfernt. Es ist daher nicht sinnvoll, alle Partikel je Iteration upzudaten.

Somit können Partikel, die durch den Nelder-Mead Algorithmus in gute Positionen gebracht wurden, sich wieder von diesen entfernen. Da sich die Partikel in jeder Iteration nur an der Position des aktuell besten Partikels orientieren (und nicht an den global besten Lösungen aus der Vergangenheit), geht somit die Information der besseren Lösung in der vorherigen Iteration verloren. Damit ist Variante 1 und Variante 3 zu bevorzugen.

5.3 Auswertung Population Based Incremental Learning - Hybrid

Um die Effizienz und die Qualität des Algorithmus' zu bewerten, wurde dieser auf das *4-Linser-System* angewendet. Bei diesem Verfahren gibt es zudem einige Parameter, bei welchen gute bzw. optimale Werte für das hier gegebene Optimierungsproblem nicht bekannt sind. Deswegen wurde der Algorithmus mit unterschiedlichen Parameterwerten getestet und dabei der global beste gefundene Meritwert f_{best} über die Iterationen für jeden Lauf aufgetragen. Auch hier wurde die Meritfunktion leicht verändert. Der Strafterm in Gleichung 3.9 wurde mit dem Faktor 10^4 multipliziert. Daher sind die erhaltenen Resultate von f^{best} nicht direkt mit anderen Algorithmen zu vergleichen. Hier soll zudem mehr das Verhalten des Verfahrens untersucht werden und weniger das absolute Endergebnis. Zu den variierten Parametern gehören:

- Anzahl der Individuen N
- Lernrate L_{R0}
- Anzahl der Subinervalle m
- Anzahl der verwendeten Individuen zur Berechnung des approximativen Gradienten R
- Anzahl der Iterationen bzw. der Generationen N_R
- Berechnung des Gradienten (approximativ oder numerisch)

Es wurde kein Abbruchkriterium festgelegt, um das Verhalten des Algorithmus auch nach langen Stagnationsphasen zu untersuchen. Insgesamt wurden 8 Läufe mit folgenden Parametern durchgeführt.

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**	**Run 8**
Iter.	250	250	250	250	250	250	250	250
N	358	358	358	358	358	358	358	100
m	30	30	30	150	150	150	150	30
L	0.5	0.1	0.8	0.75	0.75	0.75	0.3	0.7
Gradient	approx.	approx.	approx.	approx.	approx.	numerisch	numerisch	numerisch
R	25	25	25	25	50	-	-	-

Tabelle 5.3: Wahl der Parameter - Population Based Incremental Learning Hybrid

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8
Final Merit	6.06	12,12	3,33	6,96	3,98	11,08	13,62	3,25
Norm Gradient	59.800	22.650	75.000	56.470	55.960	40.928	6.744	844
# fEval	111.630	109.130	109.615	109.118	108.160	104.153	104.179	37.590
Laufzeit [min]	99	92	90	90	88	84	90	31

Tabelle 5.4: Resultate Population-Based-Incremental-Learning-Hybrid

Bei *Run 6,7* und *8* wurde der Gradient bei x_{best} nicht approximativ bestimmt, sondern durch den Differenzenquotienten numerisch. Dies hat mehr Funktionsauswertungen zufolge. Jedoch sollte überprüft werden, ob eine genauere Bestimmung des Gradienten zu besseren Resultaten führt. Zudem wurde bei *Run 8* die Anzahl der Partikel deutlich von 358 auf 100 reduziert, um die Auswirkungen dessen zu untersuchen. Es wurden 358 ($21*n+1 = 21*17+1 = 358$) Individuen betrachtet, damit der Algorithmus vergleichbar mit *PSO-NM* ist, in welchem genauso viele Partikel verwendet wurden.

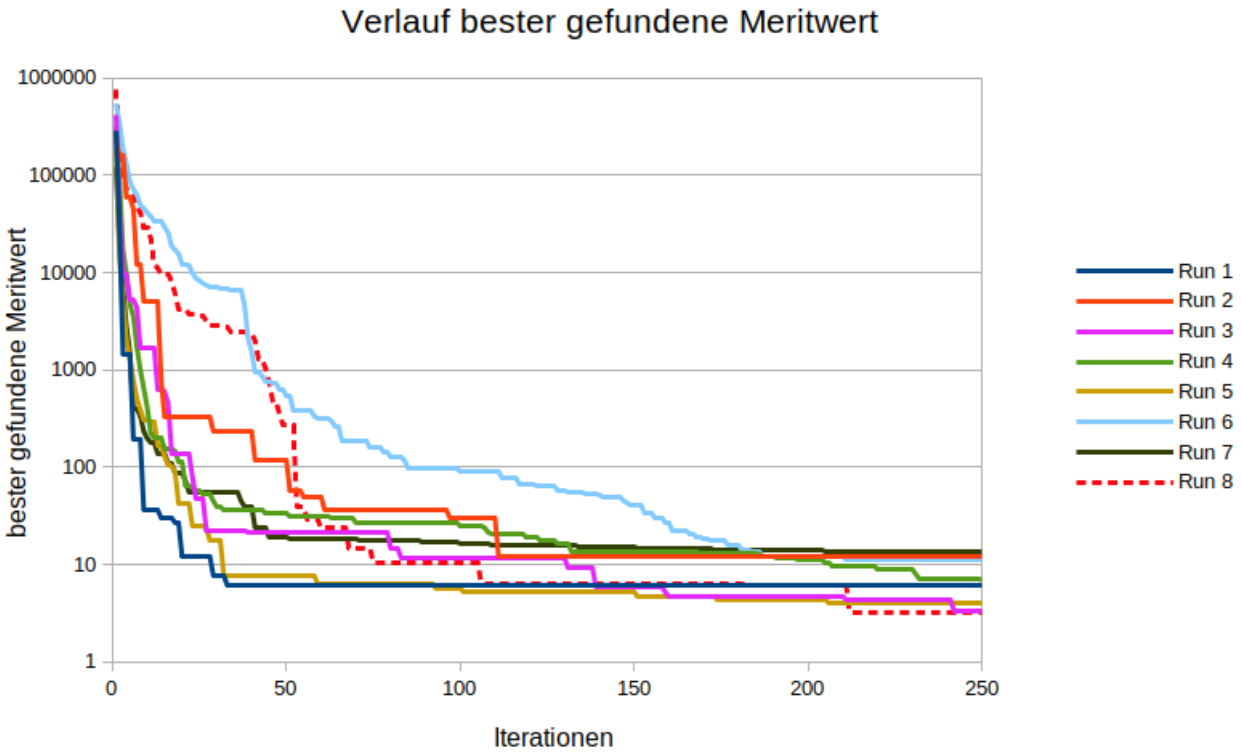

Abbildung 5.7: Resultat Population Based Incremental Learning Hybrid

Aufgrund des stochastischen Verhaltens des Algorithmus' kann man aus den Ergebnissen nicht eindeutig herauslesen, welche Parameterwahl gut ist. Dazu müssten viele Läufe mit denselben Parametern nacheinander durchgeführt werden. Jedoch lassen sich einige allgemeine Aussagen treffen. Zudem wurde der Lauf mit nur 100 Individuen schraffiert dargestellt, da dieser nicht mit den anderen vergleichbar ist (da hier deutlich weniger Funktionsauswertungen durchgeführt wurden).
Nach den anfänglichen hohen Meritwerten, welche durch das zufällige Verteilen der Individuen entstehen, werden die besten Werte f_{best} in wenigen Iterationen deutlich besser bei allen Läu-

fen. Dies zeigt, dass durch den Algorithmus relativ schnell Gebiete bzw. Lösungen gefunden werden können, welche in einem zulässigen und technisch umsetzbaren Bereich liegen. Zudem scheint dies relativ unabhängig von den verwendeten Parametern zu sein. Fast alle Läufe zeigen zwischen Iteration 50 und 250 über längere Iterationen Stagnationsphasen, in welchen keine bessere Lösung gefunden werden kann. Jedoch zeigt sich, dass selbst nach ca. 80 Iterationen ohne Veränderung noch eine verbesserte Lösung gefunden werden kann. Dies macht die Definition eines Abbruchkriteriums nahezu unmöglich. Deshalb scheint es auch hier sinnvoll zu sein, den Algorithmus über eine gesamte maximale Anzahl an Iterationen laufen zu lassen oder ihn dann abzubrechen, wenn das gefundene f_{best} unterhalb eines Sollwertes liegt. In der Tabelle 5.4 wird nun auf die gefundenen finalen Ergebnisse (also bei Iteration 250) eingegangen.

Dabei ist *Final Merit* der Wert der Meritfunktion an der finalen Lösung x_{best} und zudem wird die Norm des Gradienten an dieser Stelle angegeben. *fEval* ist die Gesamtanzahl der benötigten Funktionsauswertungen. Bei der ersten Betrachtung wird *Run 8* außen vor gelassen, da hier die Anzahl der verwendeten Individuen kleiner ist als bei den anderen Läufen. Das beste Ergebnis kann bei *Run 3* erzielt werden. Es zeigt sich jedoch, dass sich die Ergebnisse zwischen den Läufen nicht gravierend unterscheiden, obwohl die Parameter jeweils deutlich verändert wurden. Zudem lässt sich kein direkter Zusammenhang zwischen den finalen Meritwerten und der dazugehörigen Parameterwahl feststellen. Um sichere Zusammenhänge erkennen zu können, müsste man aufgrund des stochastischen Verhaltens viel mehr Läufe durchführen. Hier sollte viel mehr verdeutlicht werden, dass die Resultate trotz deutlich verschiedener Parameter in einem ähnlichen Bereich liegen. Die Resultate von f_{best} liegen dabei alle zwischen 3 und 13. Dies zeigt, unter Berücksichtigung des stochastischen Verhaltens, dass die Wahl der Parameter auf das Endresultat keine extremen Auswirkungen haben. Vergleicht man die Resultate mit den Ergebnissen vom Algorithmus *PSO-NM*, so sind die hier erreichten Werte relativ schlecht. Betrachtet man zusätzlich die Norm des Gradienten bei allen Läufen, so zeigt sich, dass diese relativ groß ist. Dies deutet darauf hin, dass die gefundenen Lösungen nicht im Bereich eines Minima liegen. Zudem werden sehr viele Funktionsauswertungen (ca. 100.000 $\rightarrow$ ca. 400 Funktionsauswertungen pro Iteration) benötigt, was zu einer Laufzeit von ca. 90 min führt (im Vergleich: Bei *PSO-NM* wurden nach vergleichbar vielen Funktionsauswertungen schon Resultate von f_{best} in der Größenordnung 1,xxx erzielt). Interessant ist dabei auch, dass *Run 6* und *Run 7* ca. 4000 Funktionsauswertungen weniger benötigen, obwohl hier der Gradient numerisch berechnet wird und nicht approximativ. Dies sollte theoretisch zu einer größeren Anzahl an Funktionsauswertungen führen. Vermutlich liegt der Gewinn aber im Backtracking Algorithmus (siehe 4.29 auf Seite 44), welcher eine geeignete Schrittweite berechnet. Bei jeder iterativen Verkleinerung der Schrittweite muss die Meritfunktion ausgewertet werden. Wenn der Gradient nahezu exakt bestimmt wird, findet sich vermutlich schneller eine zulässige Lösung, als wenn der Gradient nur approximativ bestimmt wird. Somit muss auch die Schrittweite nicht so oft verkleinert werden, was wiederum zu einer kleineren Anzahl an Funktionsauswertungen führt. Eine andere Möglichkeit wäre auch, R groß zu wählen. Bei *Run 5* wurde $R = 50$ gewählt. Somit flossen mehr Individuen in die Berechnung des approximativen Gradienten mit ein als bei den anderen Läufen mit approximativen Gradienten (mit $R = 25$). Hier ergibt sich auch eine Ersparnis von ca. 1000 Funktionsauswertungen. Somit ist es sinnvoll, den approximativen Gradienten mit vielen Individuen zu berechnen oder den numerischen Gradienten zu verwenden (wurde deshalb bei *Run 8* numerisch gewählt). Bei *Run 8* wurden nur 100 Individuen verwendet. Jedoch erhält man hier das beste Ergebnis aller Runs mit einem Meritwert von 3,25. Auch der Gradient ist hier mit 844 deutlich kleiner. Daraus kann man jedoch nicht die Schlussfolgerung ziehen, dass weniger Individuen zu besseren Ergebnissen führen. Jedoch zeigt sich, dass

man auch mit deutlich weniger Individuen zu vergleichbaren Resultaten kommen kann. Dies führt zu deutlich weniger Funktionsauswertungen und damit zu einer enormen Zeitersparnis.
Bei den Verläufen der Kurven in Diagramm 5.7 lassen sich keine gravierenden Unterschiede in Abhängigkeit von den Parametern feststellen. Lässt man *Run 8* außer betracht, so zeigt sich, dass alle Kurven mit einer Lernrate $L \geq 0.5$ einen sehr starken Abfall zeigen (ausgenommen von *Run 6*). Bei *Run 2*, in welchem die Lernrate nur 0.1 beträgt, findet eher ein langsamer Abstieg statt. Wenn die Lernrate groß ist, so werden pro Iteration mehr Individuen auf Gebiete verteilt, in welchen gute Lösungen gefunden wurden. Dadurch kann man schnell noch bessere Resultate finden. Jedoch besteht auch die Gefahr, dass dadurch Gebiete nicht mehr betrachtet werden, in welchen vielleicht noch bessere Systeme liegen. Auch zeigt sich, dass die Runs, welche mit dem numerischen Gradienten durchgeführt wurden und zudem große Lernraten haben (siehe *Run 6* und *Run 8*), deutlich weniger Stagnationsphasen zeigen.
Zusammenfassend kann man folgendes Resultat aus den Beobachtungen ziehen. Mithilfe des Algorithmus *Population Based Incremental Learning - Hybrid* lassen sich trotz einer hohen Anzahl an Funktionsauswertungen keine optimalen Ergebnisse erzielen. Es wäre zwar möglich, dass in weiteren Iterationen bessere Lösungen gefunden werden. Da dies jedoch zu noch mehr Funktionsauswertungen mit ungewissem Ausgang führt, ist davon abzuraten. Positiv ist jedoch, dass man hier in nur wenigen Iterationen in allen hier durchgeführten Läufen ohne die Vorgabe eines Startsystems viele unterschiedliche Lösungen erhält, welche zwar nicht optimal sind, aber in einem zulässigen und guten Bereich liegen. Sinnvoll wäre es, diesen Algorithmus somit als Verfahren anzusehen, welches eine Variation an Systemen liefert, die als Startwerte für effiziente lokale Algorithmen genutzt werden können. Somit besteht die Möglichkeit, eine Vielzahl an unterschiedlichen Systemen im Bereich von lokalen Minima zu erhalten und daraus das beste auszuwählen.

Verbesserung der Lösung durch lokales Verfahren
Im folgenden Experiment wird das Verfahren *Population-Based-Incremental-Learning-Hybrid (PBIL)* als Algorithmus benutzt, welcher einen guten Startwert für einen effizienteren Algorithmus liefert. Dazu wurde *PBIL* mit folgenden Parametern ausgeführt:

maxIt = 100; Anzahl Individuen = 100; m = 20;
L = 0.7; Gradient numerisch berechnet

Zudem gibt es außer der maximalen Anzahl an Iterationen *maxIt* kein Abbruchkriterium. Als zweiter Algorithmus wurde das *Truncated Newton Verfahren (TNC)* aus der *Scipy-Bibliothek* gewählt. Dieser wird bis maximal 1000 Funktionsauswertungen ausgeführt.

	PBIL	**TNC**
Final Merit	8,9	1,86
Norm Gradient	4000	560
f-Eval	49.000	1.000

Tabelle 5.5: Kombination PBIL mit TNC

In Tabelle 5.5 sind die Ergebnisse dargestellt. Nach etwa 49.000 Funktionsauswertungen erhält man eine Lösung x in einem relativ guten Bereich mit einem Meritwert von 8.9. Durch weitere 1000 Schritte des TNC-Algorithmus' kann die Lösung deutlich auf etwa 1.86 verbessert

werden. Zudem verkleinert sich der Gradient ebenso, was darauf hindeutet, dass die neue Lösung näher an einem Minimum liegt. Zum Vergleich wurde der *TNC* Algorithmus mit einem weiteren Startwert x_0 ausgeführt. Das Startsystem war dabei das auf Seite 55 in Abbildung 5.1 dargestellte System. Dabei erreichte der *TNC* Algorithmus nach 20.000 Funktionsauswertungen nur eine Lösung mit einem Meritwert von 11,81. Somit erhielt man zumindest in diesem Beispiel ein viel besseres Ergebnis, wenn man als Startwert das Ergebnis von *PBIL* verwendet. Zudem ist es aufgrund der globalen Suche von PBIL möglich, viele unterschiedliche Systeme zu erhalten, wenn man die Algorithmen öfters startet.

5.4 Auswertung Innere Punkte Verfahren

Der Erfolg des inneren Punkte Verfahren hängt maßgeblich von der Wahl der verschiedenen Parameter ab, sowie deren Anpassung innerhalb des Algorithmus. Deshalb soll im ersten Schritt die Wahl des Startwertes für μ diskutiert werden. Anschließend wird betrachtet, wie mit dem implementierten Algorithmus ein möglichst gutes Ergebnis erzielt werden kann und mögliche Verbesserungen/Anpassungen diskutiert.

5.4.1 Wahl von μ

Wir betrachten Anfangs das Ausgangssystem (Abb. 5.1) und starten den Algorithmus 5, ohne dabei μ zu aktualisieren.

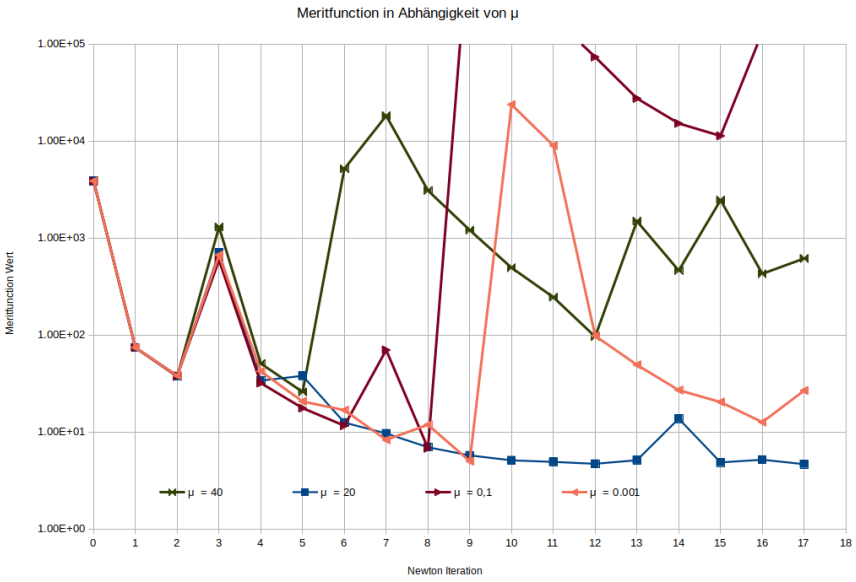

Abbildung 5.8: Wert der Meritfunktion, abhängig von μ

Abbildung 5.8 zeigt den Wert der Meritfunktion über die Anzahl der Schritte des Newton-Verfahrens. Anfangs verhält sich der Wert für alle μ sehr ähnlich, jedoch nur für $\mu = 20$ konvergiert das Verfahren gegen einen Wert von ca. 4.8 (mit einem Ausreißer bei Iteration Nr. 14). Auffällig ist, dass die anderen μ Werte Ausreißer haben, dann den Wert weiter minimieren bis es zu einem weiteren Ausreißer kommt.

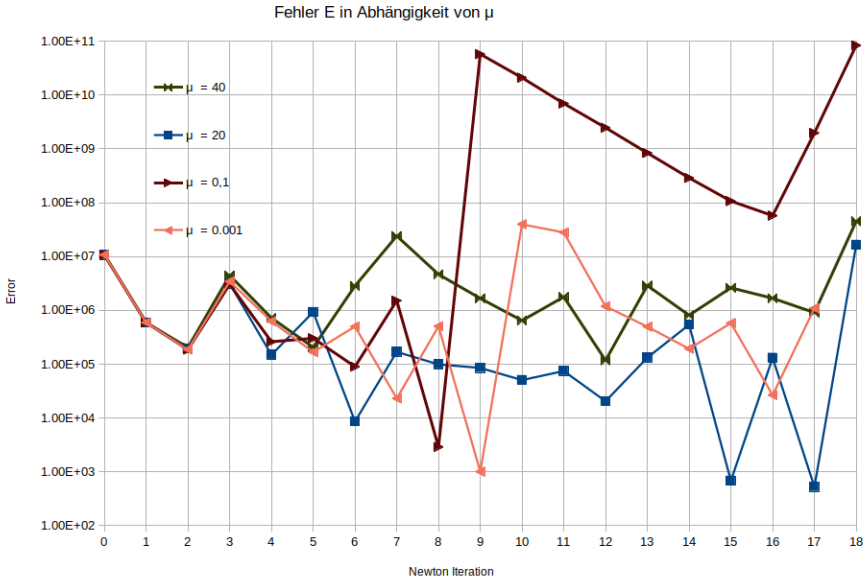

Abbildung 5.9: Fehler E, abhängig von μ

Die Abbruchbedingung für das Newton-Verfahren in Algorithmus 5 war ursprünglich $E(x_k, s_k, z_k, \mu_k) > \mu_k$. Abbildung 5.9 zeigt jedoch, das keiner der Startwert diesen Wert erreichen würde. Betrachtet man die Fehlerfunktion E etwas genauer, wird deutlich dass bereits nach wenigen Iterationen immer $\left\|\nabla f(x) - A_I^T z\right\|$ das Maximum bildet (vgl. 4.42).

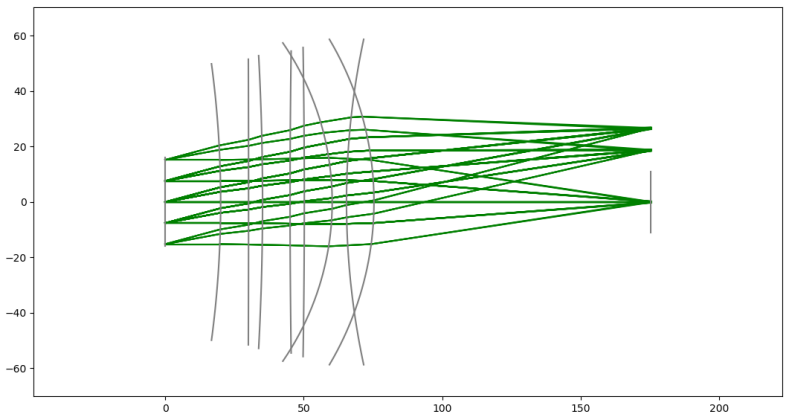

Abbildung 5.10: gefundenes Optimum für $\mu = 20$

Betrachtet man das erreichte Optimum in Abbildung 5.10 wird deutlich, dass die Verschiebung der Linsen kaum verändert wurden. Dies liegt, wiederum an dem (bereits erwähnten) Größenunterschied zwischen Krümmung und Verschiebung (siehe Kapitel 4.1.1). Auch die Krümmung wurde nur bei einigen Oberflächen geändert, die dritten bis fünfte Linsenoberfläche haben allesamt eine Krümmung von nahezu null ($< 10^{-3}$]).

5.4.2 Anpassungen und Optimierungsmöglichkeiten

Die (zugegebenermaßen) auf *Try-and-Error* basierende Anpassungen des Algorithmus sollen lediglich untersuchen, ob das Verfahren akzeptable Ergebnisse liefert. Komplexere Programme, welche die Wahl der Parameter auf Basis erprobter Algorithmen bestimmen, können ein noch besseres Ergebnis erzielen und werden später erwähnt.

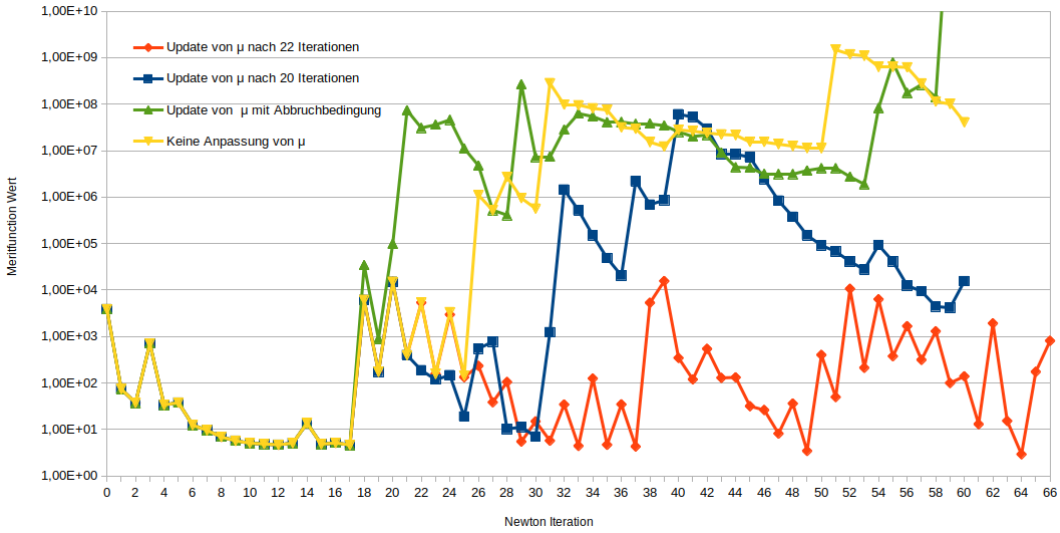

Abbildung 5.11: Meritfunktion über Newton Iterationen

Abb. 5.11 (gelb) zeigt den weiteren Verlauf der Meritfunktion, ohne dass μ angepasst wird. Der Funktionswert steigt schnell in eine Größenordnung von 10^{-7}. Es muss also ein Update stattfinden, welcher verhindert dass der Algorithmus divergiert.

Wie gezeigt, ist die Abbruchbedingung für E jedoch keine gute Wahl. Deshalb wird dieses angepasst, indem die zu erreichende Grenze μ um einen Faktor $\sigma > 0$ vergrößert wird.

$$E(x_k, s_k, z_k, \mu_k) > \sigma\mu_k$$

Um sicherzustellen, dass Abbruchbedingung am gefundenen Minimum (Abb. 5.8) erfüllt ist, wurde $\sigma = 30$ gewählt. Dies aktualisiert μ im Algorithmus erstmals am erreichten Minimum (Abb. 5.11, grün).

Es wird deutlich, dass mit dieser Aktualisierung das Verfahren divergiert. Mit dem angewandten Update-Algorithmus für μ (Kapitel 4.4.4) erreichen wir also keine Verbesserung des bereits gefundenen Werts der Meritfunktion.

Alternativ wurde der Algorithmus ohne Abbruchbedingung getestet, wobei μ nach jeder 22. Iteration aktualisiert wird (orange). Diese Variante liefert für das betrachtete System die besten Ergebnisse. So gibt es zwar wieder Ausreißer, jedoch erreicht der Algorithmus ein neues Minimum Nach 64. Iterationen. Durch das *Zick-Zack-Verhalten* schwankt das Verfahren lange zwischen kleinen und sehr großen Werten. Der Algorithmus ist sehr sensibel, was deutlich wird wenn man μ nach jeder 20. Iteration aktualisiert (blau). Dann sind die Ausreißer wieder sehr groß und der Algorithmus gelangt nicht in die Nähe des Bestwertes.

Erwähnenswert ist noch, dass die Aktualisierung von μ bei allen Varianten sofort einen Wert von nahezu Null bewirkt ($\approx 10^{-20}$). Auch hier besteht Optimierungsbedarf, wie später noch diskutiert werden soll.

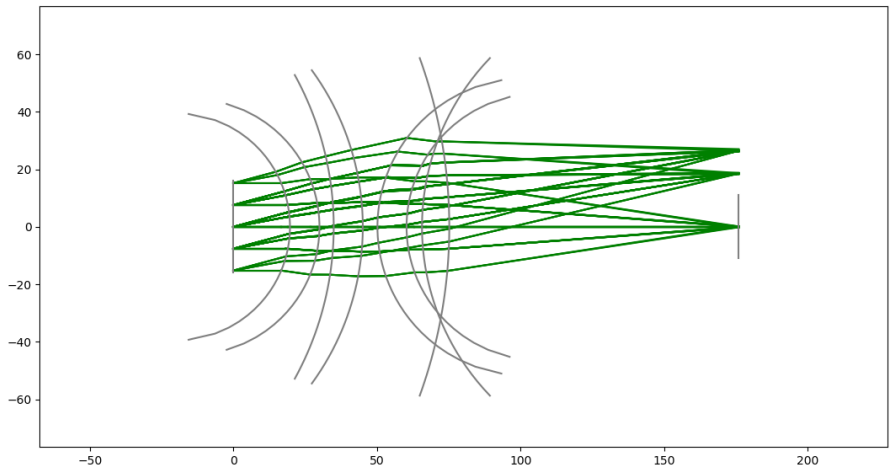

Abbildung 5.12: System mit Funktionswert $2,9$

Ein Hauptproblem des Algorithmus ist, wie erwähnt, der Größenunterschied zwischen Linsenkrümmung und -verschiebung (Kapitel 4.1.1). Außerdem wurde deutlich, dass oftmals Newton-Schritte ausgeführt werden, welche jedoch keine Verbesserung des Funktionswertes erzeugen. Dies könnte verbessert werden, indem man einen Filter implementiert, wie beispielsweise von in [16] vorgeschlagen. Dieser stellt sicher, dass nicht jeder Schritt akzeptiert wird.

Des Weiteren könnte man im vorhandenen Algorithmus weitere Parameter untersuchen. So wurden z. B. für die Startwerte der Vektoren $s = z = (1, 1, ..., 1)^T$ gewählt. Hier hätte man weiteren Spielraum für Optimierung. Außerdem könnte die Aktualisierung von μ weiter untersucht werden (Zeitpunkt und Algorithmus des Updates).

Nicht verschwiegen werden soll, dass es sich bei dem implementierten Algorithmus um ein sehr simples innere Punkte Verfahren handelt. Leider war es nicht mehr möglich, den Algorithmus weiter anzupassen und zu optimieren. Hier würde es sich anbieten, auf das *open source* Programm *IPOPT*[1] zurückzugreifen. Dieses löst nichtlineare Optimierungsprobleme mittels der inneren Punkte Methode und verschiedener Filtermethoden. *IPOPT* ist in *Fortran* und *C* implementiert, kann jedoch mittels verschiedener Module auch für *Python* genutzt werden. Dann wäre auch sichergestellt, dass die Wahl und Anpassungen der vielen Parameter auf Basis von erprobten Algorithmen stattfindet. Der implementierte Algorithmus hat gezeigt, dass das Verfahren vielversprechende Ergebnisse liefern kann. Mit einem komplexeren Algorithmus wie z. B. *IPOPT* kann dieses Ergebnis weiter verbessert und auf allgemeinere Problemstellungen angewandt werden.

5.4.3 Rechenzeit

Im Algorithmus werden pro Newton Schritt ca. 2.400 Funktionsaufrufe benötigt. D. h. für das erreichte Minimum wurde die Meritfunktion ca. 154.000 mal ausgewertet.

Interessanter ist jedoch, dass bereits nach ca. 17.000 Aufrufen ein Wert von < 10 erreicht wurde. Durch das sprunghafte Verhalten (siehe auch Abb. 5.11) werden sehr viele weitere Iterationen benötigt, um den Funktionswert weiter zu verbessern. So wird ein Wert von ca. $4,6$

[1] https://coin-or.github.io/Ipopt/

nach 29.000 Funktionsaufrufen erreicht, nach weiteren 125.000 wurde dieser Wert um gerade einmal $1,7$ reduziert. Der Vergleich der beiden gefundenen Minima (Abb. 5.10 und 5.12) zeigt jedoch, dass sich diese beiden nicht im Funktionswert, wohl aber in der Geometrie deutlich unterscheiden.

Beachtet werden muss auch, dass im implementierten Algorithmus viel Zeit für die Berechnung von δ benötigt wird (siehe Abschnitt 4.4.2). Dies kann aber evtl. ebenfalls noch optimiert werden, z. B. durch das verwenden von geeigneteren Algorithmen zur Berechnung der (Anzahl der pos./neg.) Eigenwerte.

5.5 Varianten des stochastic gradient descent

Bei der Auswertung der unterschiedlichen Varianten des stochastic gradient descent wird nicht für jeden Algorithmus das gefundene optische System dargestellt, da das Verhalten der Algorithmen im Vordergrund steht. Optimiert wurde die Meritfunktion 3.8 für $k = 1$ und $l = 2$ sowie den Strafterm 3.9. Im Folgenden sei `sample_param` der Parameter der festlegt welcher stochastische Gradient berechnet wurde. Dabei entspricht

- `sample_param='bundle'` der Definition 4.44 und
- `sample_param='wave'` der Definition 4.45.

Um eine Aussage über die Konvergenzeigenschaften zu erhalten wurde jeder Algorithmus für die dargestellten Parameter jeweils fünf mal gestartet und der Durchschnitt der Funktionswerte gebildet. Die Abbruchkriterien sind bei allen Algorithmen $\|\nabla f(x_k)\| \leq 500$ und Iterationszahl $k \geq 250$. Falls nicht explizit angemerkt ist `sample_param='bundle'`.

5.5.1 Stochastic gradient descent

In Abbildung 5.13 ist der Vergleich der Algorithmen `gradient descent`, `vanilla`, `momentum` und `nesterov accelerated gradient` dargestellt. Alle Algorithmen sind aufgrund des stochastischen Gradienten bereits nach dem ersten Iterationsschritt in unterschiedliche Richtungen gelaufen und weisen dadurch verschiedene Meritwerte auf. Die Schrittweite wurde so groß wie möglich gewählt. Eine noch größere Schrittweite führt innerhalb von `pyrate` zu Fehlermeldungen. Es ist zu erkennen, dass der gd eine sehr langsame Konvergenz aufweist und der vanilla folgt diesem Verlauf. Die anfänglichen Unterschiede im Funktionswert werden beibehalten. Dies entspricht der Erwartung, da sich der gd und der vanilla nur durch die Gradientenberechnung unterscheiden. Das momentum-Verfahren ist trotz der mehrmaligen Durchläufe sehr instabil. Jedoch unterschreitet der Funktionswert nach 30 Iterationen den Verlauf von gd und vanilla. Der nag-Algorithmus zeigt im Vergleich zu diesen Verfahren das beste Verhalten. Er erreicht als einziger einen Meritwert von unter 10. Nach 100 Iterationen scheint aber auch dieser Algorithmus instabil zu werden. Wird der zeitliche Verlauf des nag-Algorithmus mit dem vom gd-Verfahren verglichen kann nochmals eine Verbesserung des Konvergenzverhaltens festgestellt werden. Das Gradientenverfahren benötigt $\approx 9 \cdot 10^2$ Sekunden für 250 Iterationen, wohingegen das nag-Verfahren mit `sample_param='bundle'` nur $\approx 5 \cdot 10^2$ Sekunden und mit `sample_param='wave'` $\approx 4.5 \cdot 10^2$ Sekunden benötigt.
Das resultierende optische System des nag-Verfahrens ist in der Abbildung 5.15 mit dem Ausgangssystem dargestellt. Die Linsenkrümmungen wurden nur leicht verändert und die Positionen, wie erwartet gar nicht.

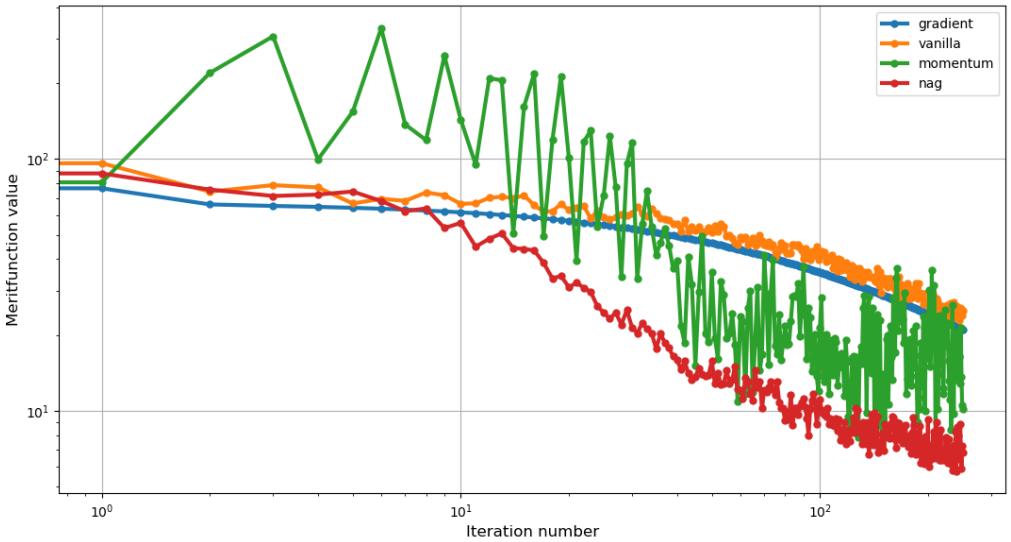

Abbildung 5.13: gradient descent - vanilla - momentum - nag ($stepsize = 10^{-9}, \gamma = 0.9$)

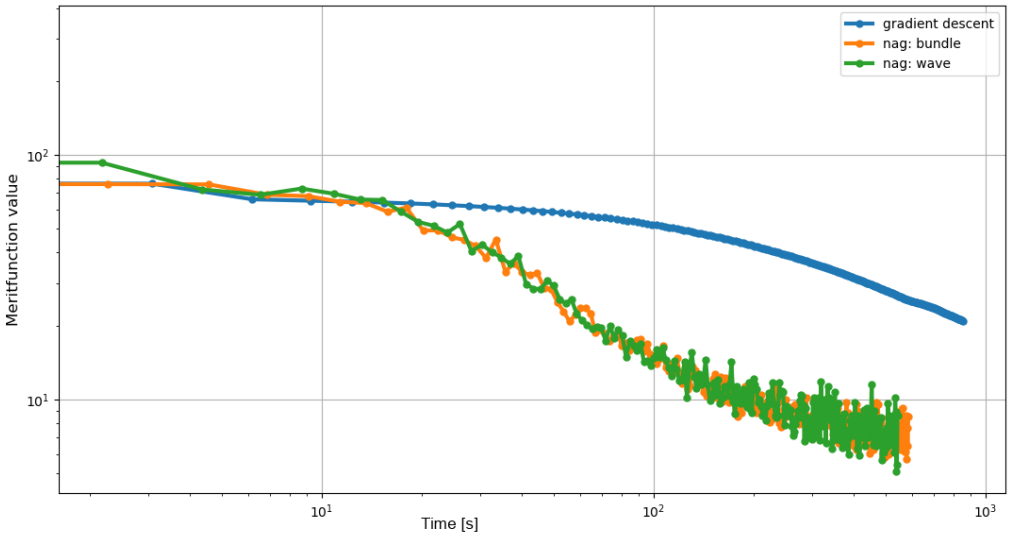

Abbildung 5.14: Zeitvergleich: gradient descent - nag ($stepsize = 10^{-9}, \gamma = 0.9$)

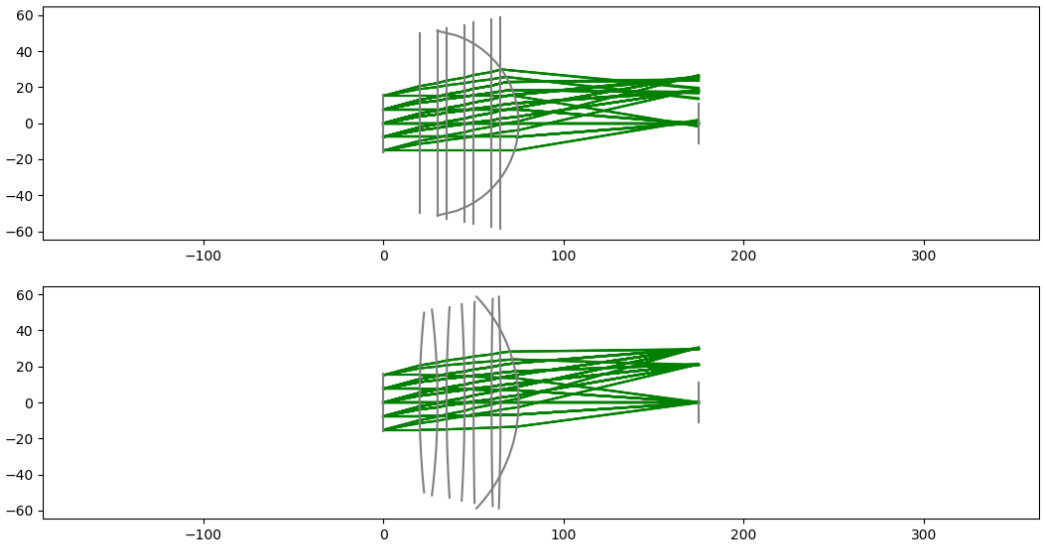

Abbildung 5.15: Vom nag-Verfahren optimiertes optische System

5.5.2 Adam und Adamax

Beim Adam wurde vorab eine Parameterstudien durchgeführt um gute Parameter zu finden, bei denen der Algorithmus konvergiert. Zuerst soll $\beta_{1,k} = \beta_1 \lambda^{k-1}, \lambda = 0.98$ und $\eta_k = \frac{\eta}{\sqrt{k}}$ untersucht werden. Hier sei angemerkt, dass bei x_0 der Funktionswert ≈ 331 beträgt. In Abbildung 5.16

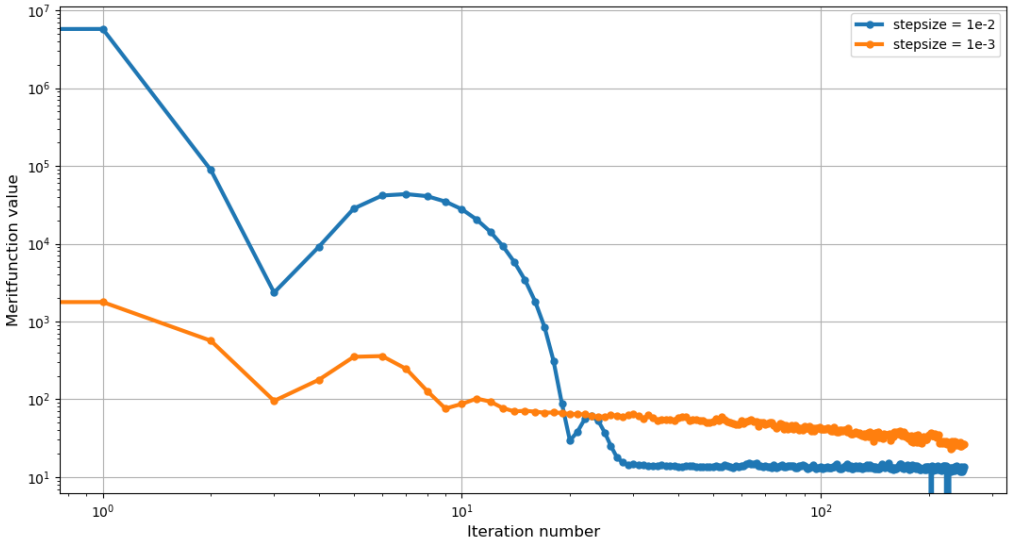

Abbildung 5.16: Adam ($\beta_1 = 0.9, \beta_2 = 0.99, \epsilon = 10^{-8}$)

ist zu sehen, dass der Funktionswert nach dem ersten Iterationsschritt bei beiden Verläufen sehr hohe Werte annimmt. Dies ist der großen Schrittweite geschuldet. Danach verringert sich der Wert wieder relativ schnell. Interessant ist, dass bei der größeren Schrittweite das besse-

re Konvergenzverhalten auftritt. Vom fünften bis zum zwanzigsten Iterationsschritt fällt der Funktionswert bei der Schrittweite von 10^{-2} sehr stark. Dieses Verhalten kann auch, bei einer entsprechenden Parameterwahl bei den anderen Algorithmen beobachtet werden. Anschließend erhöht der Wert sich wieder leicht und konvergiert dann letztendlich nach insgesamt 30 Iterationen. Das Abbruchkriterium wurden aber nicht erreicht. Bei der kleineren Schrittweite verringert sich der Funktionwert nur sehr langsam und erreicht auch nach 250 Iterationswerten keine Konvergenz. Wird β_1 und η_k nicht in jedem Schritt angepasst, sieht das Verhalten bei gleichen den Parametern ganz anders aus, wie in der folgenden Abbildung zu erkennen ist. Hier wird

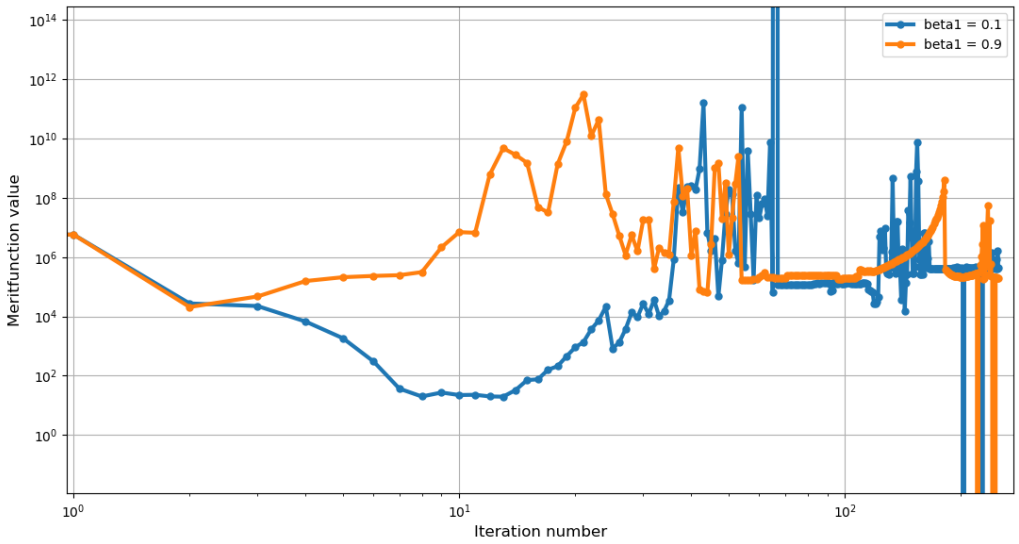

Abbildung 5.17: Adam ($stepsize = 10^{-2}, \beta_2 = 0.99, \epsilon = 10^{-8}$)

sehr schnell deutlich, dass der Algorihmus gar nicht mehr konvergiert. Bei $\beta_1 = 0.1$ ist nach der dritten Iteration kurz das Konvergenzverhalten wiederzuerkennen, bis er dann schließlich doch divergiert. Dadurch wird auch deutlich wie wichtig eine geeignete Parameterwahl ist. Aufgrund der kombinatorischen Komplexität ist das Parametertuning aber sehr langwierig. Beim Adamax-Algorithmus wurde für die größtmögliche Schrittweite ebenfalls eine Parameterstudie durchgeführt. Diese ist in Abbildung 5.18 dargestellt. Ist $\beta_1 > \beta_2$ (rote Kurve) divergiert das Verfahren, wobei immer wieder das bereits beobachtete Konvergenzverhalten auftritt. Der Meritwert springt aber immer wieder zu höheren Funktionswerten. Mit $\beta_1 \leq \beta_2$ zeigt das Verfahren eine sehr langsames Konvergenzverhalten. Bei dieser Schrittweite ist die Wahl von $\beta_1 = \beta_2 = 0.9$ am besten. Der Meritwert ist jedoch noch bei ≈ 30, was im Vergleich zum nag-Algorithmus schlecht ist. Es kann also bereits das Fazit gezogen werden, dass sowohl Adam als auch Adamax keine besseren Resultate liefern als das nag-Verfahren.

5.5.3 Adagrad

Beim Adagrad-Algorithmus ist die Schrittweite der einzige Parameter der das Konvergenzverhalten effektiv steuert. Bei einer größeren Schrittweite als 10^{-2} gibt `pyrate` wieder Fehlermeldungen aus. Daher ist der Verlauf der Meritfunktion für diese und kleinere Schrittweiten dargestellt. Was die Sprünge bei der grünen Kurve verursacht konnte nicht herausgefunden werden. Die Konvergenz bei der kleinsten Schrittweite ist, wie zu erwarten sehr langsam. Es

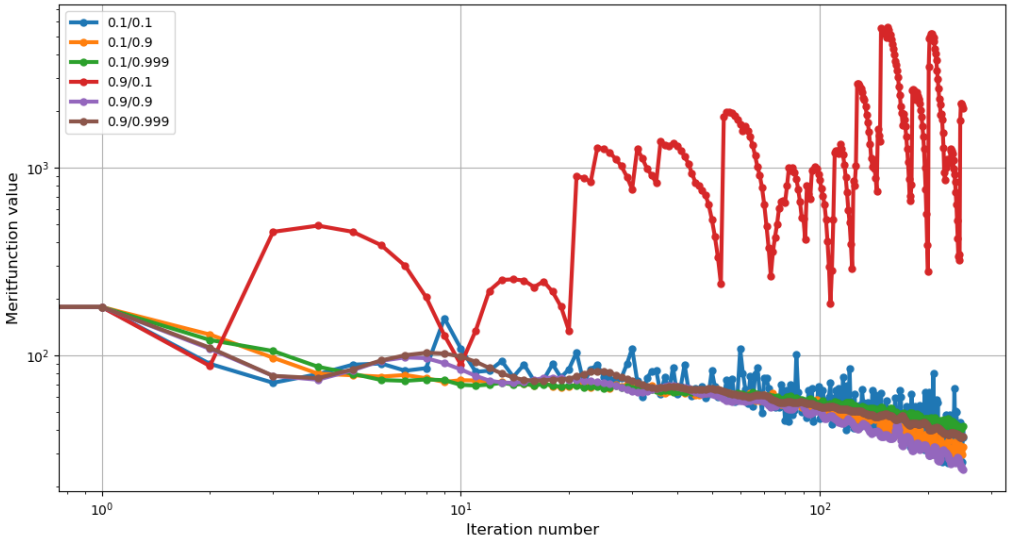

Abbildung 5.18: Adamax ($stepsize = 10^{-4}$) für unterschiedliche β_1/β_2

zeigt sich, dass eine Schrittweite von 10^{-3} das beste Verhalten aufweist, wobei hier nach 350 Iterationen ein Meritwert von ca. 20 erreicht wird. Also scheint auch hier das nag-Verfahren besser zu sein.

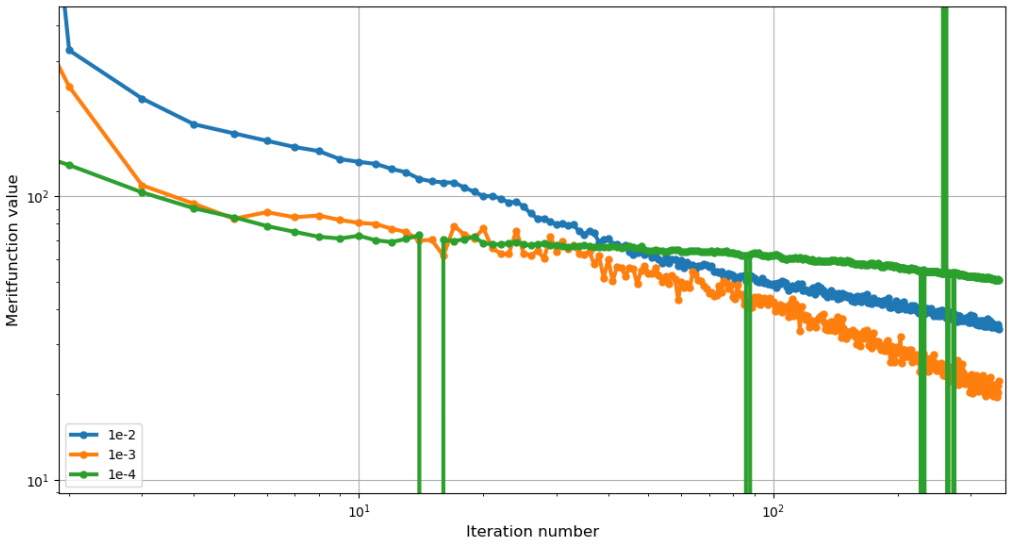

Abbildung 5.19: Adagrad ($\epsilon = 10^{-3}$) für unterschiedliche *stepsize*

5.5.4 Adadelta

Das Adadelta-Verfahren hat nur die Parameter ϵ und ρ. In Abbildung 5.20 ist der Meritfunktionswert über die Iterationszahl für unterschiedliche Werte von ρ geplottet. Für alle Parameter

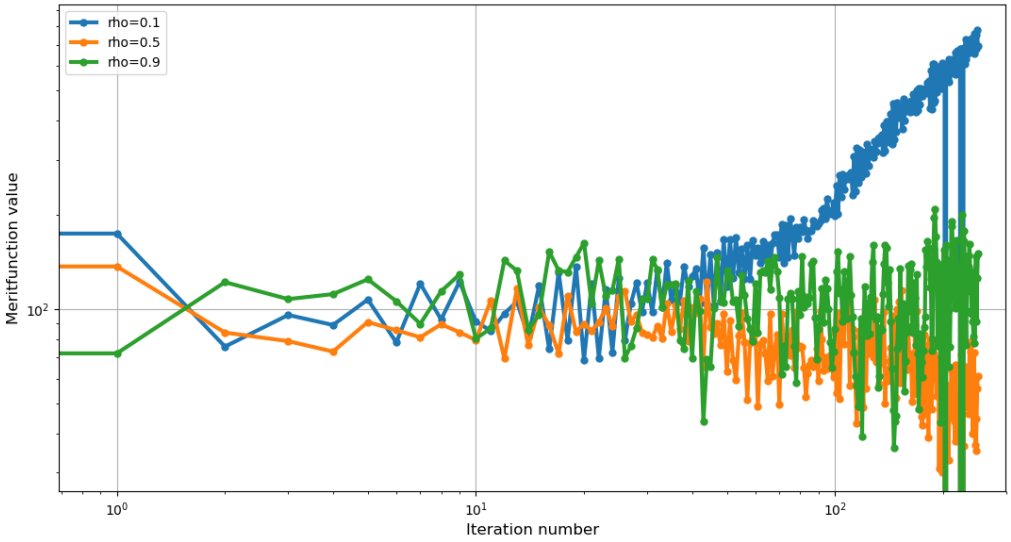

Abbildung 5.20: Adadelta ($\epsilon = 10^{-8}$) für unterschiedliche ρ

zeigt das Verfahren ein sehr unruhiges Verhalten und der Funktionswert bleibt bei allen Kurven bis zur Iteration 40 mehr oder weniger konstant. Danach divergiert das Verfahren für $\rho = 0.1$. Für $\rho = 0.9$ scheint es keine Verbesserung im Funktionswert zu geben und für den Parameter $\rho = 0.5$ sinkt er leicht. Jedoch nehmen die Ausschläge zu und der Meritwert ist auch nach den 250 Iterationen noch bei ca. 60. Dieses Verfahren zeigt also das schlechteste Verhalten, selbst bei diesem einfachen optischen System.

5.5.5 Fazit und Anmerkungen

Auch wenn die Algorithmen in der Machine Learning Community großen Anklang finden und sie dort Vorteile haben mögen, sind sie nicht für die schwierigere Aufgabe ein optisches System zu optimieren geeignet. Schon bei einer einfachen Definition der Meritfunktion und einem kleinen optischen System zeigen sie kein oder nur ein sehr schlechtes Konvergenzverhalten. Der mitunter einfachste Algorithmus `nag` lieferte das beste Ergebnis. Wie bereits in der Beschreibung der Algorithmen angemerkt, macht das globale Verhalten der Meritfunktion den Algorithmen große Schwierigkeiten. Durch die Gradienten die sich je nach Stelle $p \in \mathcal{F}$ um mehrere Größenordnungen unterscheiden ist eine feste Wahl der Schrittweite problematisch. Dieses Problem könnte eine Schrittweitenregelung, wie zum Beispiel nach der starken Wolfe-Bedingung oder der Armijo-Bedingung in Verbindung mit kubischer oder quadratischer Interpolation lösen. Beide Möglichkeiten wurden implementiert, jedoch verschieb sich dann die Frage nach der Parameterwahl zu diesen Algorithmen. Eine Idee für eine einfache Schrittweitenregelung wäre beispielsweise auch den `nag` mit einer größeren Schrittweite neu zu starten, sobald sich der Meritwert von beispielsweise 20 Iterationen nicht merklich verbessert hat und noch kein Minimum erreicht wurde. Dadurch wird der Vektor v_k auf Null gesetzt und der Impuls passt sich an das Verhalten der Meritfunktion nochmal neu an. Ob dieses Vorgehen ein deutlich bessere Verhalten hervorruft ist jedoch fraglich.
Des Weiteren sind die unterschiedlichen Größenordnungen innerhalb des Gradienten auch ein Problem mit dem die Algorithmen umgehen müssen, da die Schrittweite maßgeblich durch die

Ableitungen nach den Linsenkrümmungen bestimmt wird und dadurch die Änderungen der Linsenpositionen sehr klein werden. Eine Idee wäre hierbei zwei Schrittweiten zu benutzen. Eine für die Positionen und eine für die Krümmungen, wobei dann die Richtung nicht mehr mit dem Gradienten übereinstimmt.
Ein weiteres mögliches Verfahren wäre ein stochastisches Quasi-Newton Verfahren wie in [15]. Dadurch werden die Vorteile des stochastischen Gradienten, nämlich die geringere Rechenzeit ausgenutzt und durch die Information der approximierten Hessematrix sollte die Konvergenz noch weiter verbessert werden. Ein erster Entwurf wurde auch implementiert (`SdLBFGS`), jedoch treten hierbei neue Schwierigkeiten bezüglich des stochastischen Gradienten auf. Auch ein hybrider Algorithmus (`get_scipy_stochastic_hybrid`) wurde implementiert, der ein stochastisches Verfahren mit einem scipy-Algorithmus in generischen Weise kombiniert. Die Input-Parameter sind nämlich ein stochastisches Verfahren sowie ein scipy-Algorithmus und der Output ist ein hybrider Algorithmus (`scipy_stochastic_hybrid`). Die Idee dabei ist das stochastische Verfahren zu benutzten um schnell in die Umgebung eines Minimums zu kommen und dann auf das deterministische Verfahren umzuschalten, sodass die Konvergenz gesichert wird. Die Abbruchbedingung des stochastischen Algorithmus muss dabei so gewählt werden, dass die Umgebung eines Minimums erkannt wird.

6 Fazit und Ausblick

Ziel des Projektes war es, verschiedene Optimierungsmethoden zu untersuchen und hinsichtlich ihrer Brauchbarkeit für optische Systeme zu testen. Im Rahmen des Projekts wurden sehr viele dieser Methoden analysiert und einige in `Python` implementiert. Die Auswertung mittels des Raytracers `Pyrate` haben die Schwierigkeiten, Vor- und Nachteile der Methoden gezeigt.

Nicht verschwiegen werden soll, dass der Hauptanteil des Projekts der Raytracer `Pyrate` dargestellt hat. Die anfängliche Vermutung, dass sich die eigenen Algorithmen schnell und einfach mit `Pyrate` anwenden lassen, hat sich nicht bestätigt. So wurde sehr viel Zeit für das Nachvollziehen und Anpassen des Raytracers benötigt, wodurch die eigentliche Aufgabe, nämlich Algorithmen zu analysieren und zu testen, nicht so ausführlich bearbeitet werden konnte wie gewünscht. Nach Aussagen der Entwickler, wurde `Pyrate` noch nicht ausgiebig getestet und es werden immernoch einige *Bugs* erwartet.

Eine weitere Schwierigkeit stellte die *Meritfunktion* des Raytracers dar. So berücksichtigt diese weder den Hauptstrahl, noch die Verzeichnung des entstehenden Bildes. Des Weiteren musste die Funktion so angepasst werden, dass sich die einzelnen Linsen nicht Überschneiden, um ein physikalisch sinnvolles Ergebnis zu erzielen. Ebenfalls nicht berücksichtigt wurden die Anzahl der ankommenden Strahlen im Verhältnis zu den erzeugten Strahlen. All diese Punkte wurden im Raytracer manuell ergänzt. Jedoch stellt sich hier die Frage, wie man die einzelnen Punkte (Position der Strahlen zum Hauptstrahl, Verzeichnung, ankommende Strahlen) gewichtet. Dies müsste man einer genaueren Untersuchung unterziehen, sollte man mit dem Raytracer `Pyrate` über dieses Projekt hinaus arbeiten wollen.

Es wurden einige vielversprechende Ansätze für Optimierungsmethoden im Optik-Design gefunden und analysiert. In einem weiterführenden Projekt könnten diese Ansätze weiter verfolgt und mit kommerziellen Raytracer getestet werden. Der Vergleich mit bewährten Optimierungsmethoden im Optik-Design kann dann zeigen, ob die vorgestellten Methoden tatsächlich eine Verbesserung des aktuellen Stand der Technik darstellen.

Literatur

[1] H Martin Bücker u. a. *Automatic differentiation: applications, theory, and implementations*. Bd. 50. Springer Science & Business Media, 2006.

[2] Carlos A Coello Coello. *Learning and Intelligent Optimization*. Springer, 2011.

[3] John Duchi, Elad Hazan und Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of machine learning research* 12.Jul (2011), S. 2121–2159.

[4] Herbert Fischer. "Algorithmisches Differenzieren". In: *Technische Universität München* (2005).

[5] Herbert Gross. "Handbook of Optical Systems, Volume 1, Fundamentals of Technical Optics". In: *Handbook of Optical Systems, Volume 1, Fundamentals of Technical Optics, by Herbert Gross, pp. 848. ISBN 3-527-40377-9. Wiley-VCH, April 2005.* (2005), S. 848.

[6] Johannes Grotendorst. *Modern methods and algorithms of quantum chemistry*. Bd. 1. NIC, 2000.

[7] Diederik P Kingma und Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[8] Ralf Leidenberger. "Automatisches Differenzieren in Simulationen". Diss. Universität Ulm, 2014.

[9] An Liu und Ming-Ta Yang. "A new hybrid nelder-mead particle swarm optimization for coordination optimization of directional overcurrent relays". In: *Mathematical Problems in Engineering* 2012 (2012).

[10] Jorge Nocedal und Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[11] Nantiwat Pholdee und Sujin Bureerat. "Hybridisation of real-code population-based incremental learning and differential evolution for multiobjective design of trusses". In: *Information Sciences* 223 (2013), S. 136–152.

[12] Prof. Dr. Karsten Urban Prof. Dr. Stefan Funken. "Numerical optimization". In: *Skript zur Vorlesung 'Numerische Optimierung', Universität Ulm*. 2019.

[13] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[14] G. Schröder und H. Treiber. *Technische Optik: Grundlagen und Anwendungen*. Kamprath-Reihe. Vogel, 2007. ISBN: 9783834330864.

[15] Xiao Wang u. a. "Stochastic quasi-Newton methods for nonconvex stochastic optimization". In: *SIAM Journal on Optimization* 27.2 (2017), S. 927–956.

[16] A. Wächter und L. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Math. Program* 106.5-6 (2006), 25–57.

[17] Erwie Zahara und Yi-Tung Kao. "Hybrid Nelder–Mead simplex search and particle swarm optimization for constrained engineering design problems". In: *Expert Systems with Applications* 36.2 (2009), S. 3880–3886.

[18] Matthew D Zeiler. "Adadelta: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).