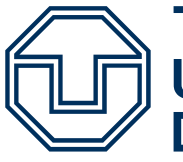
Per Natzschka

Simulationsbasierte Parameteroptimierung für Platoon-Regler

Bachelorarbeit in Informatik

8. August 2023

Please cite as:
Per Natzschka, “Simulationsbasierte Parameteroptimierung für Platoon-Regler,” Bachelor Thesis (Bachelorarbeit), Faculty of Computer Science, TU Dresden, Germany, August 2023.



TECHNISCHE UNIVERSITÄT DRESDEN

Networked Systems Modeling
Faculty of Computer Science, TU Dresden

Nöthnitzer Str. 46 · 01187 Dresden · Germany

https://nsm.inf.tu-dresden.de/

Simulationsbasierte Parameteroptimierung für Platoon-Regler

Bachelorarbeit in Informatik

vorgelegt von

Per Natzschka

geb. am 2. April 2002
in Dippoldiswalde

angefertigt an der

Technischen Universität Dresden
Fakultät Informatik
Networked Systems Modeling

Betreuer:	**Burkhard Hensel**
Gutachter:	**Burkhard Hensel**
	Christoph Sommer
Abgabe der Arbeit:	**8. August 2023**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

Per Natzschka

Signiert von : Per Natzschka

Datum: 2023-08-08 15:28:49 CEST

(Per Natzschka)
Dresden, 8. August 2023

Abstract

Platooning is a form of cooperative driving that can greatly decrease air drag, fuel consumption, and air pollution due to small gaps between vehicles. Many different controllers which control vehicle speed in order to ensure string stability have been proposed. A comparison of those controllers has not yet been done – not least because the performance of most controllers depends heavily on multiple parameters for which no optimal allocation is known. To analyze the impact of different parameters and maybe find an optimal allocation, simulation can be used. In this thesis the modular *Simopticon* framework, which automates the task of simulation optimization, is proposed. *Simopticon* is then used to optimize the parameters of the platoon controller proposed by Swaroop et al. in 1994. To achieve this, a new derivative of the DIRECT algorithm is proposed as optimization strategy and shown to perform better than the original algorithm for problems of low dimensionality. Moreover an interface to the Platooning Extension for VEINS (PLEXE) is implemented in *Simopticon* to automate the process of running and evaluating multiple platooning simulations in parallel in the course of an optimization. The framework is shown to be effective for the task of simulation optimization for platoon controllers.

Kurzfassung

Platooning ist eine Form des kooperativen Fahrens, bei der Luftwiderstand, Treibstoffverbrauch und Schadstoffausstoß der Platoon-Teilnehmer durch enge Sicherheitsabstände stark verringert werden können. Zu diesem Zweck wurde eine Vielzahl an Reglern entwickelt, die die Geschwindigkeit der Fahrzeuge im Platoon kontrolliert, um Kollisionen vorzubeugen und Kettenstabilität zu erreichen. Diese Regler wurden bisher kaum qualitativ verglichen, da deren Effektivität meist von Reglerparametern abhängt, für die a priori keine optimale Belegung bekannt ist. Um die Auswirkung der entsprechenden Parameter zu untersuchen, können Simulationen genutzt werden. In dieser Arbeit wird *Simopticon* vorgestellt – ein modulares Framework zur automatisierten Simulationsoptimierung. Dieses wird anschließend genutzt, um die Parameter des 1994 von Swaroop u. a. entwickelten Reglers zu optimieren. Zu diesem Zweck wird eine neue, auf dem DIRECT-Algorithmus basierende Optimierungsstrategie entwickelt und implementiert. Mithilfe von Experimenten wird gezeigt, dass die neue Strategie bei Problemen niedriger Dimensionalität besser abschneidet als der ursprüngliche DIRECT-Algorithmus. Des Weiteren werden eine automatisierte, parallele Ausführung von Platooning-Simulationen mithilfe der Platooning Extension for VEINS (PLEXE) und eine Auswertung der erhobenen Simulationsdaten implementiert. Anhand der Optimierung der Parameter des oben genannten Reglers wird die Effektivität von *Simopticon* bei der Optimierung von Platoon-Reglern gezeigt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Platooning bezeichnet das kooperative Fahren mehrerer Fahrzeuge in einem Verbund (Platoon). Dabei folgen die Platoon-Teilnehmer einem Platoon-Führer und halten jeweils einen Sicherheitsabstand zum vorausfahrenden Fahrzeug ein. Die simpelste Form eines Platoons stellen Fahrzeuge dar, die hintereinander fahren und dabei einen Abstandsregeltempomat (Adaptive Cruise Control, ACC) nutzen. ACCs sind eine Erweiterung der einfachen Geschwindgkeitskontrolle (Cruise Control, CC). Während CCs jedoch nur die eingestellte Geschwindigkeit halten, passt sich ein Fahrzeug mit eingeschaltetem ACC der Geschwindigkeit des vorausfahrenden Fahrzeuges an, um einen festgelegten Sicherheitsabstand zu halten. Ein Problem bei dieser Vorgehensweise ist, dass jedes Fahrzeug allein den Abstand zum unmittelbar vorausfahrenden Fahrzeug messen kann. Dadurch werden Abweichungen vom Sollabstand im Platoon nach hinten propagiert, was bei zunehmender Größe der Platoons dazu führt, dass hintere Fahrzeuge sehr stark vom Sollabstand abweichen. Um Kettenstabilität – vereinfacht gesagt die Eigenschaft, dass Abweichungen nicht nach hinten propagiert werden – zu erreichen, wird drahtlose Kommunikation zwischen den Fahrzeugen genutzt. So können Brems- und Beschleunigungsvorgänge sowie komplexere Manöver, wie zum Beispiel Spurwechsel, koordiniert werden. In diesem Fall handelt es sich um Kooperative Abstandsregeltempomaten (Cooperative Adaptive Cruise Control, CACC). CACC ermöglichen es, kleinere Sicherheitsabstände zwischen den Fahrzeugen zu wählen, wodurch die Kapazität von Straßen erhöht und der Luftwiderstand der Fahrzeuge verringert werden kann, was zu reduziertem Treibstoffverbrauch und geringerer Umweltbelastung führt [1]–[3].

Für CACC wurde eine Vielzahl von Abstandsreglern vorgeschlagen, welche auf Basis der empfangenen Informationen zu Geschwindigkeit, Beschleunigung und Abstand der anderen Platoon-Teilnehmer die Geschwindigkeit des Fahrzeugs regeln. Das Verhalten der Regler lässt sich anhand von Reglerparametern beeinflussen. Dies erschwert den qualitativen Vergleich verschiedener Regler, da deren Leistung von

Parametern abhängt, für die a priori meist keine optimale Belegung bekannt ist. Um die Auswirkung der Parameter zu untersuchen, können jedoch Simulationen von Platoons genutzt werden, die beispielsweise durch das Framework Platooning Extension for VEINS (PLEXE) [4] realisiert werden.

Das Gebiet, das sich mit dem systematischen Suchen von Optima einer sogenannten Blackboxfunktion beschäftigt, die sich nur evaluieren aber nicht ableiten lässt, nennt sich Simulationsoptimierung. Das wichtigste Bewertungskriterium für Algorithmen zur Simulationsoptimierung ist die Anzahl der benötigten Funktionsevaluationen, um ein Optimum zu finden, da diese oft sehr ressourcenintensiv sind. Im Kontext dieser Arbeit würde eine Funktionsevaluation beispielsweise das Simulieren eines Platoons mit bestimmten Reglerparametern sowie die Auswertung der Simulationsdaten beinhalten.

Ansätze zur Simulationsoptimierung lassen sich nach Amaran u. a. [5] in die in Tabelle 1.1 aufgelisteten Klassen unterteilen. Da Reglerparameter für gewöhnlich einen kontinuierlichen Wertebereich haben und ein globales Optimum für deren Belegung gefunden werden soll, können in diesem Kontext Metaheuristiken, Response-Surface-Methoden, modellbasierte Methoden oder Lipschitz-Optimierungen eingesetzt werden. Zu Metaheuristiken gehören beispielsweise zufällige Suche, evolutionäre Ansätze und Simulated Annealing. Response-Surface-Methoden spezialisieren sich auf die Analyse von Eingabe-Ausgabe-Beziehungen, mit deren Hilfe sie die zu optimierende Funktion durch eine Oberfläche (Metamodell genannt) approximieren. Modellbasierte Methoden leiten Wahrscheinlichkeitsverteilungen über bekannten Funktionswerten ab, mit deren Hilfe die Suche durchgeführt wird. Methoden der Lipschitz-Optimierung partitionieren für lipschitzstetige Funktionen einen gegebenen Parameterraum und bestimmen anhand der Stetigkeitsbedingung für jeden Teil der Partition, welches Optimum in diesem potenziell erreicht werden kann.

Ziel dieser Arbeit ist die Entwicklung eines modularen Frameworks zur Simulationsoptimierung. Dieses soll anhand einer Optimierungsstrategie automatisiert Simulationen durchführen und die erhobenen Daten auswerten können, um op-

Klasse	Wertebereich		Lokalität	
	diskret	kontinuierlich	lokal	global
Ranking und Auswahl	✓	×	×	✓
Metaheuristiken	✓	✓	×	✓
Response-Surface-Methoden	×	✓	✓	✓
Gradientenbasierte Methoden	×	✓	✓	×
Direkte Suche	✓	✓	✓	×
Modellbasierte Methoden	✓	✓	✓	✓
Lipschitz-Optimierung	×	✓	×	✓

Tabelle 1.1 – Klassifizierung von Algorithmen zur Simulationsoptimierung [5]

timale Simulationsparameter zu finden. Im Rahmen dieser Arbeit werden eine Optimierungsstrategie, eine Simulationsausführung für Platooning-Simulationen mittels PLEXE und eine Funktion zur Auswertung der Simulationsdaten im Framework implementiert. Diese werden anschließend evaluiert und zur Optimierung von Reglerparametern verwendet. Eine Erweiterung des Frameworks um weitere Optimierungsstrategien, Simulationsframeworks oder Auswertungsfunktionen ist möglich.

Kapitel 2

Grundlagen

In diesem Kapitel werden Optimierungsstrategien diskutiert, die im Framework implementiert werden sollen. In Abschnitt 2.1 wird Shuberts Algorithmus (SA) [6] erläutert, der eine globale Optimierung unbekannter Funktionen in einer Dimension ermöglicht. Dividing Rectangles (DIRECT) von Jones, Perttunen und Stuckman [7] erweitert SA, um Optimierung in mehreren Dimensionen zu ermöglichen, was in Abschnitt 2.2 gezeigt wird.

Bei beiden Ansätzen handelt es sich um Lipschitz-Optimierungen. Sie wurden ausgewählt, da sie im Vergleich zu anderen in Kapitel 1 genannten Methoden kaum Hyperparameter benötigen. Sie können also auch ohne ausführliche Untersuchung des zu optimierenden Problems angewendet werden, was sie für unterschiedlichste Simulationen einsetzbar macht. Zudem sind sie im Gegensatz zu den meisten Metaheuristiken und modellbasierten Methoden vollkommen deterministisch und konvergieren garantiert gegen ein Optimum. Eine mehrmalige Ausführung der Optimierung ist also nicht nötig und alle Ergebnisse sind vollständig reproduzierbar.

2.1 Shuberts Algorithmus

Shubert [6] schlägt einen sequentiellen Algorithmus vor, welcher das globale Maximum einer Zielfunktion findet, die eine begrenzte Änderungsrate aufweist und auf einem geschlossenen Intervall definiert ist.

2.1.1 Lipschitzbedingung

Sei $f : P \to \mathbb{R}$ eine Zielfunktion über $P \subseteq \mathbb{R}$. Einzige Voraussetzung für die Anwendung von SA ist, dass f Lipschitzstetigkeit [8] aufweisen muss. f erfüllt die Lipschitzbedingung über P, wenn es eine Lipschitzkonstante $K \in [0, \infty)$ gibt, für

die Gleichung (2.1) gilt:

$$|f(x_1)-f(x_2)| \leq K \cdot |x_1 - x_2| \text{ für alle } x_1, x_2 \in P \tag{2.1}$$

K ist hierbei eine obere Grenze für die Änderungsrate von f und kann nach Dudley [9] (sofern f differenzierbar ist) als Begrenzung der Ableitung f' von f gesehen werden:

$$|f'(x)| \leq K \text{ für alle } x \in P \tag{2.2}$$

Das kleinste K, für welches Gleichung (2.1) gilt, wird als *beste* Lipschitzkonstante bezeichnet. Sei K_{opt} die beste Lipschitzkonstante von f, so ist jedes K mit $K > K_{opt}$ ebenfalls eine Lipschitzkonstante von f. Dies bedeutet, dass für große K die Änderung des Funktionswertes $f(x)$ im Vergleich zur Änderung in x groß ausfallen *kann*, was jedoch nicht zwingend der Fall ist. Für kleine K ist die Änderung der Funktionswerte dagegen gering [10].

SA setzt voraus, dass f auf einem geschlossenen Intervall definiert ist, beziehungsweise dass das globale Maximum von f auf einem solchen Intervall gesucht wird. Es gilt also $P = [l,u]$ mit $l,u \in \mathbb{R}$. Stetige Funktionen über geschlossenen Intervallen erfüllen offensichtlich immer die Lipschitzbedingung.

2.1.2 Maximum über Teilintervallen

Sei $P' = [a,b] \subseteq P$ mit $a \geq l$ und $b \leq u$ ein Teilintervall von P. Um die obere Grenze von f auf einem solchen Teilintervall zu bestimmen, nutzt SA die Lipschitzbedingung aus. Da die Steigung von f nach oben durch K und nach unten durch $-K$ begrenzt ist (Gleichung (2.2)), folgen aus Gleichung (2.1) Grenzen m_1 und m_2 für f:

$$\begin{aligned} f(x) \leq m_1(x) &= f(a) + K \cdot (x-a) \\ f(x) \leq m_2(x) &= f(b) - K \cdot (x-b) \end{aligned} \text{ für alle } x \in P' \tag{2.3}$$

Ziel ist es nun, dasjenige $x \in P'$ zu finden, welches nach Gleichung (2.3) auf den höchsten Wert $f(x)$ abbilden kann. In Abbildung 2.1 ist ersichtlich, dass aus den Schranken m_1 und m_2 die obere Grenze von f auf P' folgt, indem man deren Schnittpunkt betrachtet. Dessen x-Koordinate soll im folgenden als $X(a,b,f,K)$ bezeichnet werden und lässt sich wie folgt berechnen:

$$X(a,b,f,K) = \frac{f(b)-f(a)}{2K} + \frac{b+a}{2} \tag{2.4}$$

Der Wert von m_1 und m_2 an $X(a,b,f,K)$ soll nachfolgend $\hat{\mathrm{M}}_{\mathrm{SHU}}(a,b,f,K)$ genannt werden und stellt die obere Grenze von f im Intervall P' nach Lipschitzbedingung

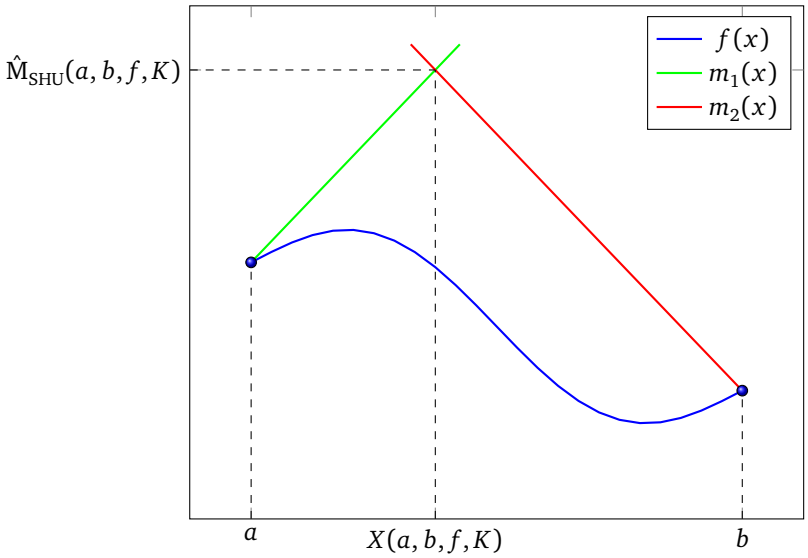


Abbildung 2.1 – Obere Grenze über P' nach Lipschitzbedingung [7]

dar. $\hat{M}_{SHU}(a,b,f,K)$ wird wie folgt berechnet:

$$\hat{M}_{SHU}(a,b,f,K) = \frac{f(b)+f(a)}{2} + K \cdot \frac{b-a}{2} \tag{2.5}$$

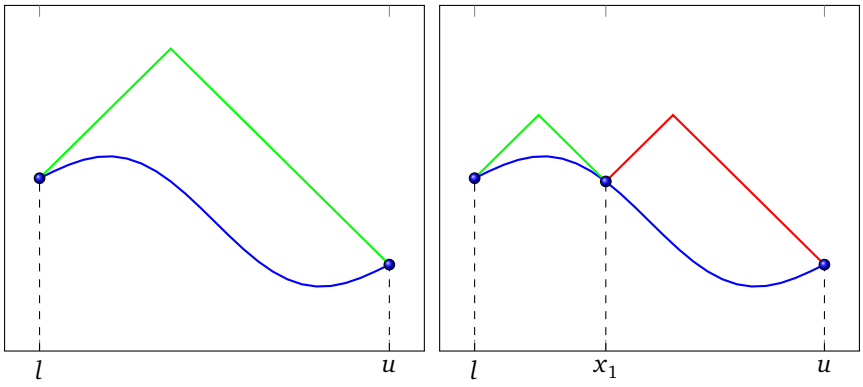
2.1.3 Suchstrategie

SA nutzt die Schätzung der oberen Grenze, um das nächste Argument zu bestimmen, bei dem die Zielfunktion abgetastet wird. Sei $I = \{[l,x_1],[x_1,x_2],\ldots,[x_{m-1},u]\}$ eine Unterteilung von P in m Teilintervalle. Zu Beginn des Algorithmus gilt $I = \{P\}$. In jedem Schritt wird das Teilintervall $[a,b] \in I$ mit der höchsten oberen Grenze ausgewählt und die Zielfunktion an $X(a,b,f,K)$ abgetastet:

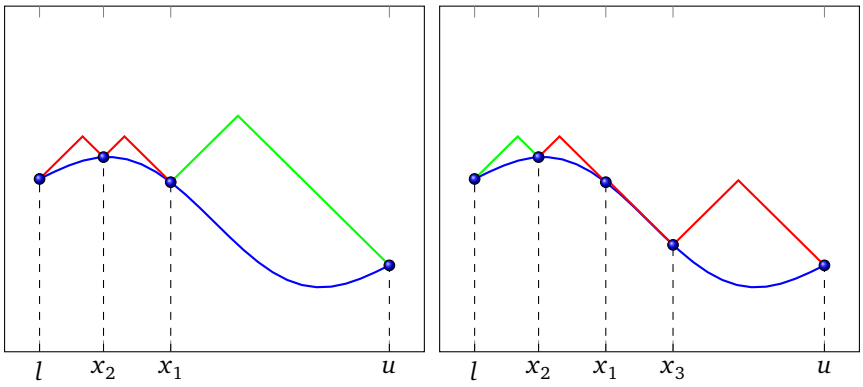
$$\begin{aligned} [a,b] &= \arg\max_{[a,b]\in I} \hat{M}_{SHU}(a,b,f,K) \\ x &= X(a,b,f,K) \end{aligned} \tag{2.6}$$

Daraufhin wird das abgetastete Intervall $[a,b]$ aus I entfernt und stattdessen die Teilintervalle $[a,x]$ sowie $[x,b]$ zu I hinzugefügt. Dann wird das nächste Teilintervall nach demselben Prinzip ausgewählt und der Vorgang wiederholt sich. Der Auswahlvorgang ist in Abbildung 2.2 dargestellt. Dort wird im Falle gleich großer Werte von $\hat{M}_{SHU}$ das am weitesten links stehende Intervall ausgewählt.

SA speichert zur Laufzeit für jedes Intervall $[a,b] \in I$ allein die durch X und $\hat{M}_{SHU}$ geschätzten Koordinaten (x',y') des Maximums auf diesem Intervall. Die Grenzen a und b, sowie die Werte $f(a)$ und $f(b)$ werden jeweils nicht gespeichert, da sie im



(a) $I = \{[l,u]\}$ **(b)** $I = \{[l,x_1],[x_1,u]\}$



(c) $I = \{[l,x_1],[x_1,x_2],[x_2,u]\}$ **(d)** $I = \{[l,x_1],[x_1,x_2],[x_2,x_3],[x_3,u]\}$

Abbildung 2.2 – Abtastreihenfolge bei SA [7]

weiteren Verlauf des Algorithmus nicht mehr benötigt werden. Die Koordinatenpaare werden in einer Menge D gespeichert, die nach der Höhe des geschätzten Maximums (also der y-Koordinate) sortiert ist. Dies ermöglicht ein schnelles Auffinden des nächsten Intervalls, das abgetastet werden soll. Wird dieses ausgewählt, tastet SA die Zielfunktion ab und nutzt deren Wert $y = f(x')$, um die Maxima der neuen Teilintervalle zu schätzen. Für diese Schätzung werden die Werte von f an den Intervallgrenzen nicht benötigt, da gilt:

$$\begin{aligned} f(a) &= y' - K \cdot (x' - a) \\ f(b) &= y' + K \cdot (x' - b) \end{aligned} \tag{2.7}$$

Demnach kann Speicher gespart werden, da die abgetasteten Funktionswerte später nicht mehr gebraucht werden und es folgen Gleichung (2.8) und Gleichung (2.9):

$$\begin{aligned} x_l' &= X(a, x', f, K) = x' - \frac{y' - y}{2K} \\ x_r' &= X(x', b, f, K) = x' + \frac{y' - y}{2K} \end{aligned} \tag{2.8}$$

$$\begin{aligned} y_l' = y_r' &= \hat{\mathrm{M}}_{\mathrm{SHU}}(a, x', f, K) = \hat{\mathrm{M}}_{\mathrm{SHU}}(x', b, f, K) \\ &= \frac{y + y'}{2} \end{aligned} \tag{2.9}$$

Dabei sind x_l' und x_r' die x-Koordinaten der links und rechts geschätzten Maxima sowie y_l' und y_r' die entsprechenden y-Koordinaten. Es ist ersichtlich, dass nun allein (x_l', y_l') und (x_r', y_r') in D gespeichert werden müssen (während das alte Maximum (x', y') entfernt wird), um den Algorithmus fortzusetzen, da Gleichung (2.8) und Gleichung (2.9) a, b, $f(a)$ und $f(b)$ nicht verwenden.

Der Algorithmus tastet die Zielfunktion solange ab, bis $\phi' - \phi \leq \varepsilon$ gilt, wobei ϕ' die höchste geschätzte Grenze, ϕ der höchste abgetastete Wert und ε eine frei gewählte Genauigkeitskonstante ist. Ist diese Bedingung erfüllt, gibt SA eine Menge $\Phi = \left\{ [a, b] \in I \;\middle|\; \hat{\mathrm{M}}_{\mathrm{SHU}}(a, b, f, K) \geq \phi \right\}$ von Teilintervallen zurück, in denen ein globales Maximum liegt (oder ein Maximum, welches höchstens um ε vom globalen Maximum abweicht). Diese wird wie folgt aus D berechnet:

$$\Phi = \left\{ \left[x' - \frac{y' - \phi}{K}, x' + \frac{y' - \phi}{K} \right] \;\middle|\; (x', y') \in D, y' \geq \phi \right\} \tag{2.10}$$

Um weiteren Speicher einzusparen, können nach jedem Iterationsschritt zudem alle Paare (x', y') aus D entfernt werden, für die $y' < \phi$ gilt, da dort nach Lipschitzbedingung kein Maximum liegen kann, das größer als der bereits gemessene Wert ϕ ist. Wie in Gleichung (2.10) leicht erkennbar ist, ändert dies nichts am Ergebnis.

Shubert [6] beweist, dass sein Algorithmus für $n \to \infty$ Schritte gegen das globale Maximum konvergiert und die genutzte Abtastungsstrategie optimal ist. Der Ablauf ist in Algorithmus 2.1 zusammengefasst.

2.2 DIRECT-Methode

Die DIRECT-Methode erweitert SA für die Simulationsoptimierung in mehreren Dimensionen. Um dies zu ermöglichen, nehmen Jones, Perttunen und Stuckman [7] Verbesserungen an SA vor, da der ursprüngliche Algorithmus für diese Probleme laut Shubert zu ineffizient ist [6]. Zudem umgehen sie das Problem der Wahl einer

Lipschitzkonstante K, das bei mehrdimensionalen Blackboxfunktionen nicht trivial ist.

2.2.1 Nachteile von Shuberts Algorithmus

Jones, Perttunen und Stuckman [7] benennen drei Probleme bei der Anwendung von SA auf Simulationsoptimierung:

1. Wahl der Lipschitzkonstante,
2. Zahl der Auswertungen bis zur Konvergenz und
3. Komplexität in höheren Dimensionen.

Punkt 1 stellt ein Problem dar, da eine Lipschitzkonstante für komplexe Simulationen nicht leicht zu finden ist. Für gewöhnlich wird dabei eine sehr hohe Konstante gewählt, um dem Kriterium in Gleichung (2.1) zu entsprechen. Dies hat jedoch negative Auswirkungen auf Punkt 2, da für hohe K mehr Intervalle ein Optimum enthalten könnten und entsprechend überprüft werden müssen. DIRECT löst dieses Problem, indem der Algorithmus alle möglichen Änderungsraten $\tilde{K} \in (0, \infty)$ in Betracht zieht. Punkt 3 wird in den Methoden von Shubert [6], Galperin [11] und Pintér [12] deutlich. Diese unterteilen einen n-dimensionalen Parameterraum in kleinere Hyperquader, deren Eckpunkte ausgewertete Punkte sind. Somit müssen für jeden Hyperquader 2^n Eckpunkte ausgewertet werden. DIRECT wertet dagegen jeweils nur den Mittelpunkt der Hyperquader aus, wodurch weniger teure Simulationen durchgeführt werden müssen.

Jones, Perttunen und Stuckman [7] schlagen vor, die Lipschitzkonstante K bei SA als Gewichtung zwischen globaler und lokaler Suche zu betrachten. Das nächste zu untersuchende Intervall $[a, b]$ wird nach Gleichung (2.6) ausgewählt. Für $K = 0$

Require: l, u, f, K, ε
1: $(x', \phi') \leftarrow (X(l,u,f,K), \hat{\mathrm{M}}_{\mathrm{SHU}}(l,u,f,K))$
2: $D \leftarrow \{ (x', \phi') \}$
3: $\phi \leftarrow \max(f(l), f(u))$
4: **while** $\phi' - \phi > \varepsilon$ **do**
5: $\quad (x', y') \leftarrow \sup(D)$
6: $\quad y \leftarrow f(x')$
7: $\quad x'_l \leftarrow x' - \frac{1}{2K} \cdot (y' - y)$
8: $\quad x'_r \leftarrow x' + \frac{1}{2K} \cdot (y' - y)$
9: $\quad y'_{new} \leftarrow \frac{1}{2} \cdot (y' + y)$
10: $\quad D \leftarrow (D \setminus (x', y')) \cup \left\{ (x'_l, y'_{new}), (x'_r, y'_{new}) \right\}$
11: $\quad \phi \leftarrow \max(\phi, y),\ \phi' \leftarrow \max(\phi', y'_{new})$
12: **end while**

Algorithmus 2.1 – Shuberts Algorithmus

wird hier das Intervall selektiert, das bisher die höchsten Werte an den Grenzen hatte $\left(\frac{f(b)+f(a)}{2}\right)$, was eine lokale Suche darstellt. Bei $K \to \infty$ wird dagegen das größte Intervall $\left(\frac{b-a}{2}\right)$ gewählt, was äquivalent zu globaler Suche ist. Die Auswirkung von K auf die Auswahl des nächsten Intervalles ist in Abbildung 2.3 dargestellt.

Diese Sichtweise auf K zeigt, weshalb die Wahl einer möglichst kleinen Lipschitzkonstante so hohe Relevanz hat. Ziel von DIRECT ist es, mithilfe von globaler Suche potenziell optimale Hyperquader zu finden und in diesen dann mithilfe von lokaler Suche schnell ein Optimum anzunähern. Ist K zu hoch gewählt, liegt der Fokus des Algorithmus zu sehr auf globaler Suche und K muss zu einem späteren Zeitpunkt verringert werden, um die lokale Suche einzuleiten. Dabei sind allerdings wiederum Zeitpunkt und Umfang der Verringerung von K nicht leicht zu bestimmen.

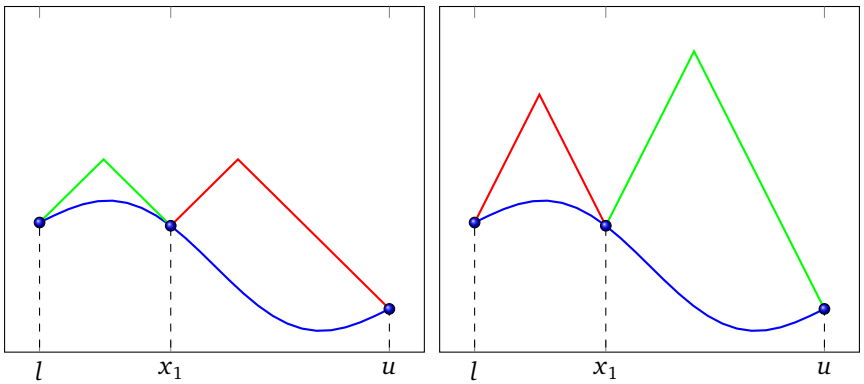
2.2.2 DIRECT in einer Dimension

Um die in Abschnitt 2.2.1 aufgezeigten Probleme zu lösen, ändern Jones, Perttunen und Stuckman [7] die Auswertungsstrategie von SA[1] von Auswertung an den Intervallgrenzen zu Mittelpunktauswertung. Zudem machen die Autoren den Algorithmus mithilfe von variablen Änderungsraten von einer vordefinierten Lipschitzkonstante unabhängig. Das Ergebnis ist äquivalent zum DIRECT Algorithmus in einer Dimension. Dieser wird nachfolgend in Abschnitt 2.2.3 für n Dimensionen verallgemeinert.

2.2.2.1 Mittelpunktauswertung

Sei f, wie in Abschnitt 2.1 definiert, eine Zielfunktion in einer Variablen über dem Intervall P. Ein Hauptunterschied zwischen DIRECT und SA ist, dass die untere

[1] Jones, Perttunen und Stuckman [7] nutzen hier eine Variante von SA, die Minima statt Maxima sucht, da DIRECT ebenfalls Minima finden soll.



(a) $K = K_{opt}$ **(b)** $K \gg K_{opt}$

Abbildung 2.3 – Wahl des Intervalls bei unterschiedlichen Lipschitzkonstanten

Grenze eines Teilintervalls nicht anhand der Werte an dessen Grenzen, sondern an dessen Mittelpunkt berechnet wird. Im Folgenden bezeichnet ein Tripel (a,c,b) das Intervall $[a,b] \subseteq P$ mit dem Mittelpunkt $c = \frac{a+b}{2}$. Steht nur der Wert $f(c)$ in der Intervallmitte zur Verfügung, ändern sich die Grenzen m_1 und m_2 wie folgt:

$$\begin{aligned} f(x) \geq m_1(x) &= f(c) + K \cdot (x-c) \text{ für } x \in [a,c] \\ f(x) \geq m_2(x) &= f(c) - K \cdot (x-c) \text{ für } x \in [c,b] \end{aligned} \tag{2.11}$$

Die untere Grenze von f auf P' mit Lipschitzkonstante K lässt sich dementsprechend mit dem nachfolgend definierten $\hat{\mathrm{M}}_{\text{1-DR}}$ berechnen:

$$\hat{\mathrm{M}}_{\text{1-DR}}(a,b,f,K) = f\left(\frac{a+b}{2}\right) - K \cdot \frac{b-a}{2} \tag{2.12}$$

Die Untere Grenze bei der DIRECT-Methode ist in Abbildung 2.4 dargestellt. Diese hängt allein von $f(c)$ ab. Man könnte zwar mithilfe der Funktionswerte in der Mitte von benachbarten Intervallen stärkere Grenze berechnen, dies wäre jedoch in höheren Dimensionen unmöglich [7].

Das nächste abgetastete Intervall (a,c,b) wird also nach der unteren Grenze aus Gleichung (2.12) ausgewählt. Dieses kann allerdings nicht mittig in $[a,c]$ und $[c,b]$ geteilt werden. In diesem Fall wäre die Bedingung, dass für jedes Intervall nur dessen Mittelpunkt abgetastet wird, nicht erfüllt, da c nun eine abgetastete Intervallgrenze wäre. Deshalb unterteilt DIRECT das Intervall in Drittel, was in Abbildung 2.5 dargestellt ist. Dadurch hat das mittlere Intervall weiterhin den Mittelpunkt c, für

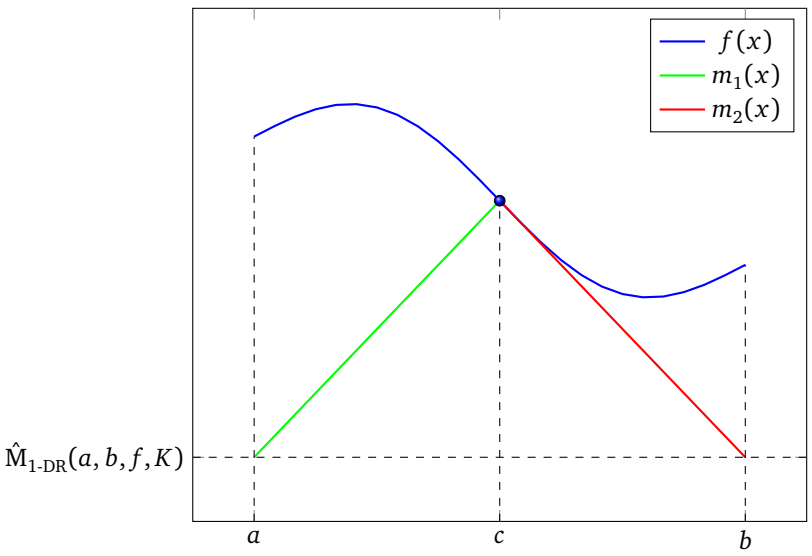


Abbildung 2.4 – Untere Grenze bei DIRECT [7]

den bereits $f(c)$ bestimmt wurde. Es müssen die neuen Mittelpunkte $c_l = c - \delta$ und $c_r = c + \delta$ abgetastet werden. Dabei ist $\delta = \frac{b-a}{3}$ ein Drittel der Intervallgröße.

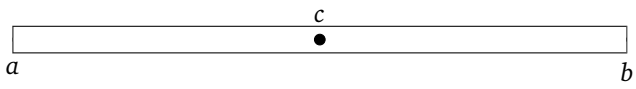
2.2.2.2 Variable Änderungsraten

Sei $I = \{ [l, x_1], [x_1, x_2], \ldots, [x_{m-1}, u] \}$ eine Unterteilung von P in m Teilintervalle. Zu Beginn des Algorithmus gilt $I = \{ P \}$. Um das nächste potenziell optimale Intervall auszuwählen, kann Gleichung (2.12) nur mit Eingabe einer Konstante K genutzt werden. Hier konstruieren Jones, Perttunen und Stuckman [7] die Menge $S \subseteq I$, die alle Teilintervalle $[a, b]$ enthält, die für eine Konstante $\tilde{K} \in (0, \infty)$ potenziell optimal sind, für die also gilt:

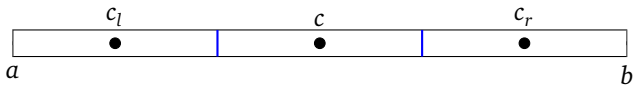
$$\exists_{\tilde{K} \in (0,\infty)} \forall_{[a',b'] \in I} : \hat{\mathrm{M}}_{\text{1-DR}}(a, b, f, \tilde{K}) \leq \hat{\mathrm{M}}_{\text{1-DR}}(a', b', f, \tilde{K}) \tag{2.13}$$

Hierbei ist $\tilde{K}$ nicht zwingend eine Lipschitzkonstante von f, sondern eine beliebige Änderungsrate. In einem Iterationsschritt bestimmt DIRECT zuerst S und teilt dann alle darin liegenden Intervalle.

Die Auswahl der Intervalle, die zu S nach Gleichung (2.13) hinzugefügt werden, ist in Abbildung 2.6 dargestellt. Dabei sind die verschiedenen Intervalle als Punkte mit ihrer Größe $(b - a)$ auf der horizontalen und dem Wert ihres Mittelpunktes $f(c)$ auf der vertikalen Achse eingezeichnet. Wie zu erkennen ist, befinden sich die ausgewählten, optimalen Intervalle auf der unteren rechten Seite der konvexen Hülle aller Punkte. Um diese zu bestimmen, schlagen Jones, Perttunen und Stuckman [7] den Graham Scan [13] vor. Dieser liegt für m Punkte, die bereits nach Abszisse geordnet sind, in $O(m)$. Zudem weisen die Autoren darauf hin, dass die Intervalle nur eine Größe von $\frac{u-l}{3^k}$ für $k \in \mathbb{N}$ haben können. Dies ist auf die Teilungsstrategie (vergleiche Abschnitt 2.2.2.1) zurückzuführen, die Intervalle ausschließlich drittelt. Demnach benötigt der Graham Scan nur $O(m')$ Operationen, wenn die Intervalle nach Größe gruppiert sind. Dabei bezeichnet m' die Anzahl der unterschiedlichen Intervallgrößen mit $m' \leq m$.



(a) Vor der Unterteilung



(b) Nach der Unterteilung

Abbildung 2.5 – Intervallunterteilung in einer Dimension [7]

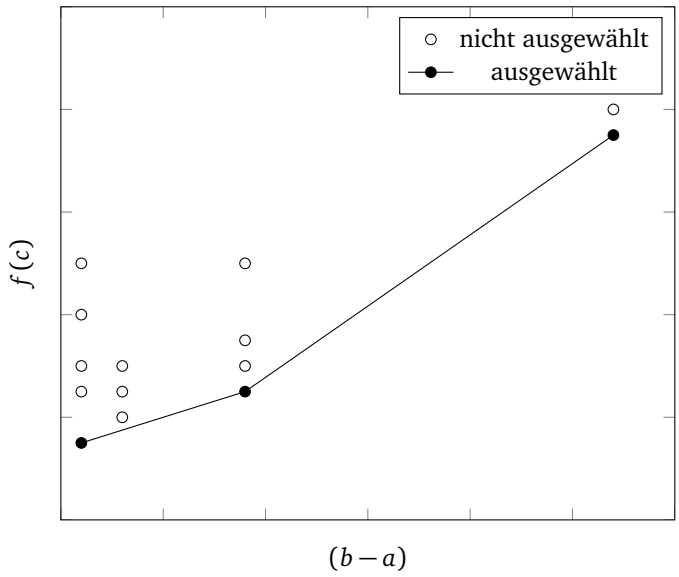


Abbildung 2.6 – Auswahl potenziell optimaler Intervalle [7]

Damit DIRECT nicht nur triviale Verbesserungen erzielt, werden die potenziell optimalen Intervalle in S zusätzlich danach gefiltert, ob sie den bisher kleinsten Wert ϕ um den Anteil ε unterschreiten, wobei ε frei gewählt werden kann. Wird zum Beispiel $\varepsilon = 0{,}01$ gewählt, muss das geschätzte Minimum das aktuelle um 1 % unterschreiten [7]. Dies soll verhindern, dass DIRECT viele Funktionsauswertungen für lokale Suche nutzt, wenn nur minimale Verbesserungen zu erwarten sind. Die Bedingung für Teilintervalle $[a,b] \in S$ in Gleichung (2.13) muss also um eine weitere Bedingung con_2 erweitert werden:

$$\begin{aligned} &\exists_{\tilde{K}\in(0,\infty)} : con_1 \wedge con_2 \\ &\quad con_1 \equiv \forall_{[a',b']\in I} : \hat{\mathrm{M}}_{\text{1-DR}}(a,b,f,\tilde{K}) \leq \hat{\mathrm{M}}_{\text{1-DR}}(a',b',f,\tilde{K}) \\ &\quad con_2 \equiv \hat{\mathrm{M}}_{\text{1-DR}}(a,b,f,\tilde{K}) \leq \phi - \varepsilon \cdot |\phi| \end{aligned} \tag{2.14}$$

Als Abbruchbedingung schlagen Jones, Perttunen und Stuckman [7] eine festgelegte Konstante T vor, die die maximale Zahl an Iterationen festlegt. Denkbar sei allerdings auch, die Anzahl an Funktionsauswertungen (sprich: Simulationsläufen) zu begrenzen. Ist eine Lipschitzkonstante bekannt, kann alternativ dieselbe Abbruchbedingung wie in SA genutzt werden. In diesem Fall kann außerdem Gleichung (2.14) angepasst werden, sodass $\tilde{K} \in (0,K]$ gelten muss.

Der vollständige DIRECT-Algorithmus in einer Dimension ist in Algorithmus 2.2 dargestellt.

2.2.3 DIRECT in mehreren Dimensionen

Im folgenden wird der DIRECT-Algorithmus aus Abschnitt 2.2.2 für n-dimensionale Funktionen erweitert. Dazu muss der höherdimensionale Parameterraum P definiert und die Teilungsstrategie desselben festgelegt werden.

2.2.3.1 Höherdimensionale Parameterräume

Sei f eine Zielfunktion in n Variablen. Jones, Perttunen und Stuckman [7] nehmen an, dass jede Variable allein Werte von 0 bis 1 annehmen kann. Dies schränkt die Allgemeinheit der DIRECT-Methode für kontinuierliche n-dimensionale Parameterräume nicht ein, da dies nur eine reversible Normalisierung voraussetzt. Der zu untersuchende Parameterraum P ist also ein n-dimensionaler Einheits-Hyperwürfel. f ist entsprechend wie folgt definiert: $f : P \to \mathbb{R}$ mit $P = [0,1]^n$. Dieser Hyperwürfel wird im Verlaufe des Algorithmus in kleinere Hyperquader unterteilt.

Die potenziell optimalen Hyperquader, die zerteilt werden sollen, werden ähnlich dem eindimensionalen Fall bestimmt. Statt der Seitenlänge wird für $n > 1$ Dimensionen der Abstand d zwischen Mittelpunkt $\vec{c}$ und den Eckpunkten des Hyperquaders genutzt, um dessen Größe zu repräsentieren. Sei $I = \{ (\vec{c}_1, d_1), \ldots (\vec{c}_m, d_m) \}$ eine Unterteilung von P in m Hyperquader. Für potenziell optimale Hyperquader $(\vec{c}, d) \in S \subseteq I$ muss analog zu Gleichung (2.14) folgende Bedingung gelten:

$$
\begin{aligned}
&\exists_{\tilde{K} \in (0,\infty)} : con_1 \wedge con_2 \\
&\quad con_1 \equiv \forall_{(\vec{c}',d') \in I} : \hat{\mathrm{M}}_{n\text{-DR}}(\vec{c}, d, f, \tilde{K}) \leq \hat{\mathrm{M}}_{n\text{-DR}}(\vec{c}', d', f, \tilde{K}) \\
&\quad con_2 \equiv \hat{\mathrm{M}}_{n\text{-DR}}(\vec{c}, d, f, \tilde{K}) \leq \phi - \varepsilon \cdot |\phi|
\end{aligned}
\tag{2.15}
$$

Require: l, u, f
1: $m \leftarrow 1$, $t \leftarrow 0$, $c \leftarrow \frac{l+u}{2}$
2: $I \leftarrow \{ (a, c, f(c), b) \}$
3: $\phi \leftarrow f(c)$
4: **while** $t < T$ **do**
5: $S \leftarrow \left\{ j \in I \;\middle|\; \exists_{\tilde{K} \in (0,\infty)} : con_1 \wedge con_2 \right\}$
6: **for all** $(a, c, d, b) \in S$ **do**
7: $\delta \leftarrow \frac{b-a}{3}$
8: $c_l \leftarrow c - \delta$, $c_r \leftarrow c + \delta$
9: $d_l \leftarrow f(c_l)$, $d_r \leftarrow f(c_r)$
10: $P_l \leftarrow (a, c_l, d_l, a + \delta)$, $P_r \leftarrow (b - \delta, c_r, d_r, b)$
11: $I \leftarrow (I \setminus \{ (a, c, d, b) \}) \cup \{ P_l, (a + \delta, c, d, b - \delta), P_r \})$
12: $m \leftarrow m + 2$
13: $\phi \leftarrow \min(\phi, d_l, d_r)$
14: **end for**
15: **end while**

Algorithmus 2.2 – DIRECT in einer Dimension [7]

Dabei ist $\hat{\mathrm{M}}_{n\text{-DR}}$ eine Schätzung, die wie folgt definiert ist:

$$\hat{\mathrm{M}}_{n\text{-DR}}(\vec{c}, d, f, K) = f(\vec{c}) - K \cdot d \tag{2.16}$$

2.2.3.2 Teilungsstrategie

Sei der zu unterteilende Teil des Parameterraumes ein Hyperwürfel mit Kantenlänge 3δ und Mittelpunkt $\vec{c}$. In jedem Iterationsschritt wird dieser in jeder Dimension partitioniert. Zuerst wertet DIRECT dazu die Zielfunktion f an den Punkten $\vec{c} \pm \delta\vec{e}_i$ mit $i = 1, \ldots, n$ aus. Dabei bezeichnet $\vec{e}_i$ den i-ten Einheitsvektor, also den Vektor, der in Dimension i eine 1 und sonst nur 0 enthält. Für $n = 2$ ist dies in Abbildung 2.7 dargestellt. In diesem Fall werden jeweils die Werte abgetastet, die sich um δ rechts, links, ober- und unterhalb des Mittelpunktes befinden. Nach der Teilung müssen diese Punkte wiederum Mittelpunkte von Hyperquadern sein. In Abbildung 2.7 ist ersichtlich, dass es für die Unterteilung $n!$ Möglichkeiten gibt, je nachdem, in welcher Reihenfolge die partitionierten Dimensionen ausgewählt werden. In Abbildung 2.7a wurde beispielsweise zuerst in der horizontalen Dimension 1, danach in der vertikalen Dimension 2 gedrittelt – bei Abbildung 2.7b ist die Reihenfolge vertauscht. Eine Möglichkeit, einen Hyperwürfel für $n = 3$ zu unterteilen, ist in Abbildung 2.8 dargestellt.

Da Anzahl und Größe der Hyperquader unabhängig von der gewählten Reihenfolge ist, könnte diese willkürlich bestimmt werden. An Abbildung 2.7 wird allerdings ersichtlich, dass die Hyperquader mit Mittelpunkten $\vec{c} \pm \delta\vec{e}_i$ am größten sind, wenn zuerst in Dimension i geteilt wird. Mit dieser Beobachtung begründen Jones, Perttunen und Stuckman [7] den Vorschlag, zuerst in derjenigen Dimension

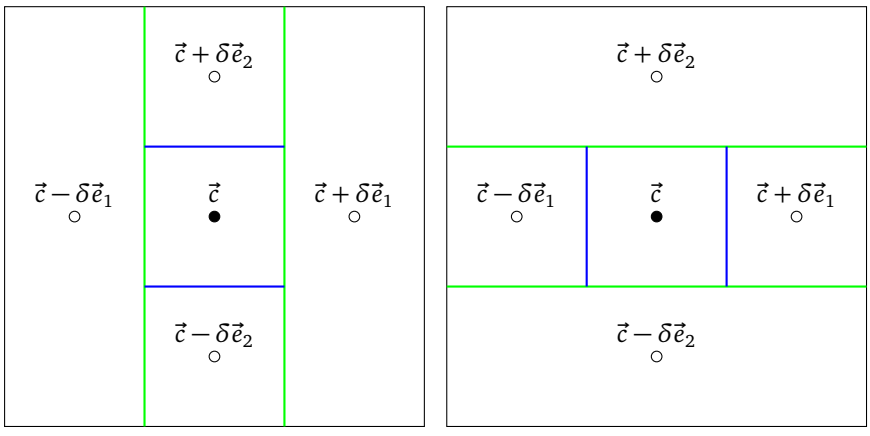


Abbildung 2.7 – Unterteilung von Quadraten [7]

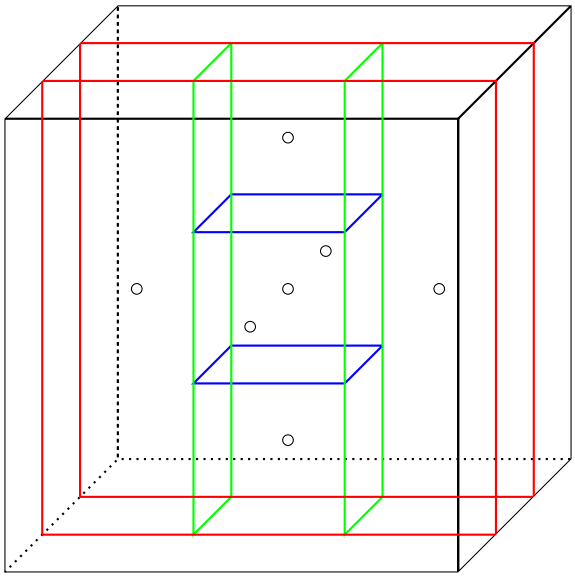


Abbildung 2.8 – Unterteilung eines Würfels

zu teilen, in der der beste Werte abgetastet wurden. Es sei w_i also der beste Wert in Dimension i mit:

$$w_i = \min(f(\vec{c} - \delta\vec{e}_i), f(\vec{c} + \delta\vec{e}_i)) \text{ für } i = 1, \ldots, n \tag{2.17}$$

Nun wird die die nächste Dimension j, in der geteilt wird, gewählt durch:

$$j = \underset{i \in \{1,\ldots,n\}}{\arg\min} w_i \tag{2.18}$$

Somit enthalten nach der Teilung die größten Hyperquader die besten Werte. Dadurch sind diese bei der Auswahl der zu teilenden Hyperquader attraktiver, was den Fokus auf die lokale Suche verstärkt.

Hat der zu teilende Hyperquader unterschiedliche Seitenlängen, ändert sich das Vorgehen nur geringfügig. Die Hyperquader werden in diesem Fall nur an ihren längsten Seiten partitioniert. Sei l_i die Seitenlänge des Hyperquaders in Dimension i. Dann wird allein entlang der Dimensionen in L geteilt, mit:

$$L = \underset{i \in \{1,\ldots,n\}}{\arg\max} l_i \tag{2.19}$$

Aus L wird analog zu Gleichung (2.18) die erste Dimension ausgewählt, nach der unterteilt wird. In einem Iterationsschritt werden die ausgewählten Hyperquader anhand dieser Reihenfolge in $|L|$ Dimensionen partitioniert.

Der vollständige Ablauf von DIRECT ist in Algorithmus 2.3 dargestellt. Dabei bezeichnet $split(P', i)$ die Drittelung des Hyperquaders P' in Dimension i.

Jones, Perttunen und Stuckman [7] weisen darauf hin, dass es möglich ist, den Speicherverbrauch der DIRECT-Methode von der Zahl der Dimensionen unabhängig zu machen. Um dies zu erreichen, werden je Hyperquader in I dessen Mittelpunkt $\vec{c}$ und Seitenlängen nicht gespeichert. Da diese zur Anwendung der Teilungsstrategie jedoch benötigt werden, müssen Informationen gespeichert werden, mit denen sie rekonstruiert werden können. Die Autoren schlagen hier eine baumartige Betrachtungsweise vor, bei der die Drittel, in die ein Hyperquader geteilt wird, als dessen Kindknoten gelten. Welche Informationen über diesen Baum gespeichert werden muss und wie die Rekonstruktion abläuft, elaborieren sie nicht näher.

Jones, Perttunen und Stuckman [7] beweisen, dass DIRECT für Zielfunktionen konvergiert, die in der Nähe des globalen Optimums stetig sind.

2.2.4 Erweiterungen

Der bisher beschriebene, ursprüngliche DIRECT-Algorithmus wurde seit seiner Vorstellung durch Jones, Perttunen und Stuckman [7] im Jahre 1993 vielfach erweitert und optimiert. Jones und Martins [14] analysieren Nachteile der Methode und vergleichen verschiedene Erweiterungen.

Als Hauptproblem ist unter anderem die hohe Anzahl an benötigten Funktionsauswertungen zu nennen. Da diese im Falle von Simulationen meist teuer sind, stellen sie bei der Effizienz eines Optimierungsverfahrens den größten Faktor dar.

Require: f, n
1: $m \leftarrow 1$, $t \leftarrow 0$
2: $I \leftarrow \{ [0,1]^n \}$
3: $\phi \leftarrow f(center([0,1]^n))$
4: **while** $t < T$ **do**
5: $S \leftarrow \left\{ j \in I \;\middle|\; \exists_{\tilde{K} \in (0,\infty)} : con_1 \wedge con_2 \right\}$
6: **for all** $P' \in S$ **do**
7: $L \leftarrow sortByMinValue(\{ i \in \{1,\ldots,n\} \mid l_i = \sup(\{ l_1,\ldots,l_n \}) \})$
8: $P'_m \leftarrow P'$, $I \leftarrow I \setminus \{ P' \}$
9: **for all** $i \in L$ **do**
10: $(P'_l, P'_m, P'_r) \leftarrow split(P'_m, i)$
11: $I \leftarrow I \cup \left\{ P'_l, P'_r \right\}$
12: $m \leftarrow m + 2$
13: $\phi \leftarrow \min(\phi, f(center(P'_l)), f(center(P'_r)))$
14: **end for**
15: $I \leftarrow I \cup \left\{ P'_m \right\}$
16: **end for**
17: **end while**

Algorithmus 2.3 – DIRECT in n Dimensionen [7]

DIRECT findet in seiner Grundform zwar meist effizient das Becken des globalen Optimums, benötigt jedoch zur genauen Bestimmung desselben überpropotional viele Evaluationen. Dies ist darauf zurückzuführen, dass in jedem Iterationsschritt nicht nur der vermeintlich optimale Hyperquader, sondern auch bis zu $m'-1$ weitere Hyperquader geteilt werden, wobei m' die Anzahl unterschiedlicher Größen der Hyperquader ist (vergleiche Abschnitt 2.2.2.2). Diese Verschiebung auf globale Optimierung verlangsamt DIRECT enorm, weshalb einige der von Jones und Martins [14] vorgestellten Erweiterungen auf andere, lokale Optimierer zurückgreifen [15]–[17]. Diese lassen sich jedoch nicht ohne weiteres in das Partitionierungsschema von DIRECT integrieren, weshalb andere Ansätze versuchen, DIRECT entweder phasenweise oder vollständig für lokale Optimierung anzupassen [18]–[23].

Ein weiteres Problem von DIRECT ist, dass con_2 aus Gleichung (2.15) dazu führt, dass DIRECTs Effektivität von der additiven Skalierung der Zielfunktion abhängt. Nutzt man beispielsweise $f + a$ mit $a \in \mathbb{R}$ statt f als Zielfunktion, erhöht sich ϕ ebenfalls um a, wodurch sich die Bedingung verändert.

Auch die Mittelpunktauswertung von Hyperquadern kann zu schlechteren Ergebnissen führen. Ist beispielsweise der Funktionswert im Mittelpunkt des Hyperquaders, der das globale Optimum enthält, sehr schlecht, wird dieser erst sehr spät abgetastet. Die Bewertung eines Hyperquaders anhand eines einzigen Funktionswerts ist also eine Vereinfachung, die sich negativ auswirken kann.

Das Problem der Verschiebung zu globaler Optimierung, sowie das der additiven Skalierung löst Robustes Mehrebenen-DIRECT (Multilevel robust DIRECT, MRDIRECT) von Liu, Zeng und Yang [18], das in Abschnitt 2.2.4.1 vorgestellt wird. Adaptive Diagonale Kurven (Adaptive Diagonal Curves, ADC) von Sergeyev und Kvasov [20] vermeidet die potenziell schlechteren Ergebnisse durch Mittelpunktauswertung und reduziert die Zahl der nötigen Funktionsauswertungen. ADC wird in Abschnitt 2.2.4.2 beleuchtet.

2.2.4.1 MRDIRECT

MRDIRECT von Liu, Zeng und Yang [18] löst das Problem der additiven Skalierung sowie das zu hohe Gewicht, das DIRECT globaler Suche beimisst.

Das Problem der additiven Skalierung kann nach Finkel und Kelley [22] gelöst werden, indem con_2 aus Gleichung (2.15) wie folgt angepasst wird:

$$con_2 \equiv \hat{\mathrm{M}}_{n\text{-DR}}(\vec{c}, d, f, \tilde{K}) \leq \phi - \varepsilon \cdot |\phi - \tilde{f}| \tag{2.20}$$

Dabei bezeichnet $\tilde{f}$ den Median der bisher abgetasteten Werte von f. Somit ist con_2 unabhängig von linearer Skalierung der Zielfunktion. DIRECT-Adaptionen mit dieser Eigenschaft ordnet Liu [24] der Klasse der robusten DIRECT-Algorithmen zu.

Ebene i	Sortierung	ς_i	ε_i
2	–	100 %	10^{-5}
1	aufsteigend nach Größe	90 %	10^{-7}
0	aufsteigend nach Größe	10 %	0

Tabelle 2.1 – Übersicht der Ebenen in MRDIRECT

Um die Gewichtung von globaler Suche zu limitieren nutzt MRDIRECT in Anlehnung an Mehrgitterverfahren [25] verschiedene Ebenen. Je höher die Ebene, desto globaler die Suche. Ziel ist dabei, mit globaler Suche das Becken des globalen Optimums zu finden und dieses daraufhin mithilfe von lokaler Suche zu untersuchen. Eine Übersicht der Ebenen ist in Tabelle 2.1 abgebildet.[2]

Auf Ebene 2 wählt MRDIRECT – wie der ursprüngliche DIRECT-Algorithmus – die potenziell optimalen Hyperquader aus der Menge aller Hyperquader I aus. Bei den Ebenen 1 und 0 werden nur die besten ς_1 beziehungsweise ς_0 Hyperquader in Betracht gezogen. Dazu werden die Hyperquader aufsteigend nach Größe geordnet – bei gleichgroßen Hyperquadern werden jene präferiert, die einen optimaleren Wert am Mittelpunkt haben. Auf Ebene 1 werden von dieser sortierten Liste von Hyperquadern nur die ersten $\varsigma_1 = 90\,\%$, auf Ebene 0 nur die ersten $\varsigma_0 = 10\,\%$ in Betracht gezogen. Die restlichen Hyperquader werden in beiden Fällen ignoriert. Dadurch wird auf den beiden niedrigeren Ebenen der Fokus auf globale Suche reduziert, indem große Hyperquader ignoriert werden.

Liu, Zeng und Yang [18] zeigen, dass die effizienteste Reihenfolge der Ebenen einem „W-Zyklus“ folgt, dass also die Ebenen in der in Abbildung 2.9 dargestellten Reihenfolge genutzt werden. Diese W-Zyklen wiederholen sich, bis der Algorithmus konvergiert.

In einer späteren Veröffentlichung erweitern Liu u. a. [19] MRDIRECT zusätzlich um eine Variation des Parameters ε. Für die globale Suche auf Ebenen 2 und 1 wählen sie $\varepsilon_2 = 10^{-5}$ beziehungsweise $\varepsilon_1 = 10^{-7}$. Dies soll wie im ursprünglichen DIRECT-Algorithmus verhindern, dass bei globaler Suche Hyperquader mit zu niedrigen Werten ausgewählt werden und dass lokale Optima zu sehr im Fokus stehen. Auf Ebene 0 wählen Liu u. a. [19] $\varepsilon_0 = 0$, um lokale Verfeinerung über den kleinsten Hyperquadern selbst bei geringen Verbesserungen zu ermöglichen.

Liu u. a. [19] zeigen Konvergenz von MRDIRECT und weisen experimentell nach, dass der Algorithmus deutlich weniger Funktionsevaluationen benötigt, um globale Optima anzunähern.

[2] MRDIRECT ermöglicht den Einsatz von mehr als diesen Ebenen, Liu, Zeng und Yang [18] raten jedoch aufgrund hoher Komplexität davon ab.

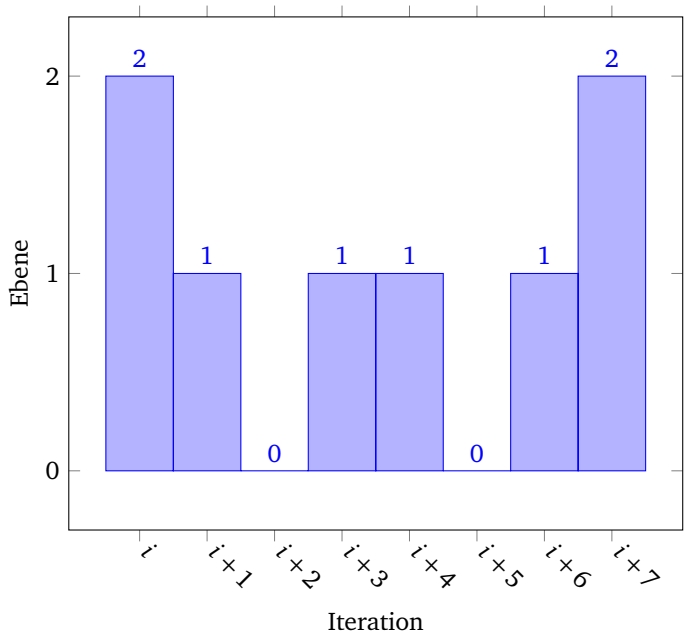


Abbildung 2.9 – W-Zyklus der MRDIRECT-Ebenen

2.2.4.2 Adaptive Diagonale Kurven

ADC von Sergeyev und Kvasov [20] nutzt ebenfalls mehrere Ebenen, um zwischen lokaler und globaler Optimierung zu wechseln. Vor allem sticht jedoch ADCs Teilungsstrategie heraus, die nach Jones und Martins [14] auf die meisten DIRECT-Derivate anwendbar und sehr effizient ist.

Sergeyev und Kvasov [20] unterteilen den Parameterraum ebenfalls in Drittel. Dabei teilen sie allerdings die ausgewählten Hyperquader je Iterationsschritt in nur einer Dimension. Dieses Vorgehen hatte Jones [15] bereits in einer Überarbeitung der DIRECT-Methode vorgeschlagen. Jones und Martins [14] weisen experimentell nach, dass dies die Anzahl benötigter Funktionsevaluationen deutlich senkt.

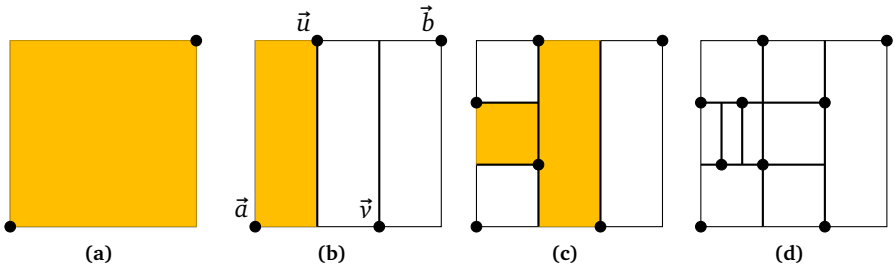


Abbildung 2.10 – Unterteilung bei ADC nach Sergeyev und Kvasov [20]

ADC verfolgt zudem eine andere Abtastungsstrategie als DIRECT. Während dort je Hyperquader allein dessen Mittelpunkt abgetastet wird, evaluiert ADC jeweils die Werte an zwei gegenüberliegenden Ecken des Hyperquaders. Entgegen der Intuition führt dies nicht zu einer Verdopplung sondern vielmehr zu einer Verringerung der abgetasteten Funktionswerte. Beispielhaft ist dies in Abbildung 2.10 zu sehen. Dort ist zwar das Verhältnis von Hyperquadern zu Abtastpunkten in Abbildung 2.10a 1:2, in Abbildung 2.10d ist es bereits ein 1:1 Verhältnis. Sergeyev und Kvasov [20] nennen sogar einen experimentellen Fall, in dem für 106 359 Hyperquader nur 15 343 Funktionswerte abgetastet werden mussten, während für DIRECT jeder Hyperquader genau einem Funktionswert zugeordnet werden müsste.

Bei dieser Abtaststrategie lässt sich Dimension i, in der ein Hyperquader zuerst partitioniert wird, nicht wie in Gleichung (2.18) bestimmen, da hierzu Werte innerhalb des Hyperquaders abgetastet werden müssten. Deshalb wählen Sergeyev und Kvasov [20] die niedrigste der Dimensionen, in denen der Hyperquader seine kürzeste Seitenlänge hat (vergleiche Gleichung (2.19)). Es gilt also:

$$i = \inf(L) \tag{2.21}$$

Seien $\vec{a} = (a_1, a_2, \ldots, a_n)^T$ und $\vec{b} = (b_1, b_2, \ldots, b_n)^T$ die Eckpunkte eines Hyperquaders, der in Dimension i geteilt werden soll. Durch die Unterteilung entstehen drei Hyperquader, die $\vec{a}$ und $\vec{u}$, $\vec{u}$ und $\vec{v}$ beziehungsweise $\vec{v}$ und $\vec{b}$ als Eckpunkte haben. Dies ist in Abbildung 2.10b dargestellt. $\vec{u}$ und $\vec{v}$ werden dabei wie folgt berechnet:

$$\begin{aligned} u &= \left(b_1, \ldots, b_{i-1}, b_i + \frac{2}{3}(a_i - b_i), b_{i+1}, \ldots, b_n\right)^T \\ v &= \left(a_1, \ldots, a_{i-1}, a_i + \frac{2}{3}(b_i - a_i), a_{i+1}, \ldots, a_n\right)^T \end{aligned} \tag{2.22}$$

Neben der Reduktion der Funktionsauswertungen verringert ADC laut Jones und Martins [14] zudem die Wahrscheinlichkeit, dass optimale Hyperquader aufgrund eines suboptimalen Mittelpunktes nicht abgetastet werden. Bei der Auswahl der potenziell optimalen Hyperquader werden beide Werte in Betracht gezogen, was die Bewertung des Hyperquaders von zwei statt nur einer Messung abhängig und somit genauer macht. Dazu nutzt ADC $\hat{\mathrm{M}}_{\mathrm{ADC}}$ – eine Abwandlung von $\hat{\mathrm{M}}_{\mathrm{SHU}}$ – für mehrere Dimensionen (vergleiche Gleichung (2.5)). Dabei handelt es sich nicht um die Berechnung einer tatsächlichen unteren Grenze im Hyperquader, da diese für viele Dimensionen zu teuer wäre. Es wird stattdessen allein das eindimensionale Segment $\left[\vec{a}, \vec{b}\right]$ der beiden Eckpunkte $\vec{a}$ und $\vec{b}$ betrachtet:

$$\hat{\mathrm{M}}_{\mathrm{ADC}}\left(\vec{a}, \vec{b}, f, K\right) = \frac{f\left(\vec{a}\right) + f\left(\vec{b}\right)}{2} - K \cdot \frac{\|\vec{b} - \vec{a}\|}{2} \tag{2.23}$$

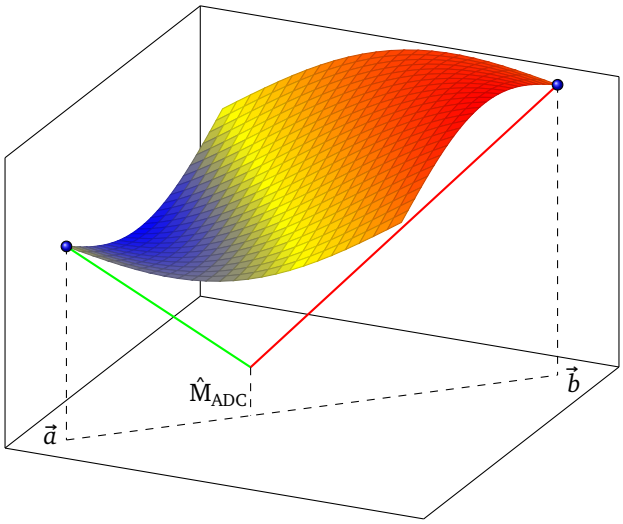


Abbildung 2.11 – Untere Grenze eines Hyperquaders bei ADC [20]

Dabei stellt $\|\vec{v}\|$ die euklidische Norm des Vektors $\vec{v}$ dar. Die Schätzung der unteren Grenze ist in Abbildung 2.11 visualisiert. Hier kommt das gleiche Prinzip wie in Abbildung 2.1 zum Einsatz. So setzen Sergeyev und Kvasov [20] eine Bewertung der Hyperquader anhand von Eckpunkten um, während sie gleichzeitig die Zahl der benötigten Funktionsauswertungen verringern.

Zudem bemängeln Sergeyev und Kvasov [20] eine zu starke Fokussierung von DIRECT auf lokale Minima, sobald deren Becken gefunden sind. Aus diesem Grund unterscheiden Sie lokale und globale Phasen. In der lokalen Phase soll ähnlich wie bei MRDIRECT der Fokus auf lokaler Optimierung liegen, während in der globalen Phase ein neues potenziell optimales Becken gesucht wird. In der globalen Phase werden entgegen der MRDIRECT-Ebenen nur die größeren Hyperquader der aktuellen Partitionierung betrachtet. Dabei handelt es sich um alle Hyperquader, die größer als P_{opt} sind, wobei P_{opt} den kleinsten Hyperquader bezeichnet, der als Eckpunkt die beste bisher gefundene Parameterkombination hat. Analog dazu betrachtet die lokale Phase alle Hyperquader die kleiner als P_{opt} oder gleicher Größe sind.

Die Optimierung beginnt bei ADC mit einer lokalen Phase. Nach jeder Iteration wird folgende Bedingung überprüft:

$$\phi \leq \phi' - 10^{-2} \cdot |\phi'| \tag{2.24}$$

Dabei bezeichnen ϕ und ϕ' den besten abgetasteten Wert nach beziehungsweise vor der Iteration. Gilt die Bedingung in Gleichung (2.24) nicht, erfolgt ein Wechsel in die

globale Phase.[3] Diese wird solange genutzt, bis nach einer Iteration Gleichung (2.24) gilt, dann wird wieder in die lokale Phase gewechselt.

[3]Neben dieser müssen weitere Bedingungen erfüllt sein, die im Folgenden aber keine Relevanz haben.

Kapitel 3

Implementierung

In diesem Kapitel soll die Implementierung eines Frameworks zur automatisierten Simulationsoptimierung vorgestellt werden, welches im Folgenden *Simopticon* genannt wird. Dieses ist in C++ implementiert und ermöglicht es, verschiedene Optimierungsstrategien, Simulationsumgebungen und Zielfunktionen miteinander zu kombinieren. Zuerst soll dabei in Abschnitt 3.1 die Top-Level-Architektur des Frameworks beschrieben werden. Zudem werden Optimierungs- (Abschnitt 3.2), Simulations- (Abschnitt 3.3) und Evaluationsmodul (Abschnitt 3.4) vorgestellt. Des Weiteren wird eine Implementierung beschrieben, die es ermöglicht, Simulationen in PLEXE mit einer neuen Form der DIRECT-Methode zu optimieren.

3.1 Top-Level-Architektur

Dieser Abschnitt soll einen Überblick über den Optimierungsvorgang mit *Simopticon* geben. Dabei werden neben dem allgemeinen Aufbau des Frameworks die *Controller*-Klasse und Möglichkeiten, diese zu konfigurieren, vorgestellt.

3.1.1 Strategie zur Optimierung

Zentrale Klasse des Frameworks ist die *Controller*-Klasse, welche den Optimierungsvorgang steuert. Sie besitzt eine Optimierungseinheit (*Optimizer*), eine Simulationseinheit (*SimulationRunner*) und eine Evaluationseinheit (*Evaluation*). Der *Optimizer* beinhaltet eine Strategie, die das Minimum einer Zielfunktion $f : X_1 \times \cdots \times X_n \to \mathbb{R}$, die nur durch Argument-Wert-Paare gegeben ist, ermitteln kann. Die Simulationseinheit ermöglicht es, Simulationen mit gegebenen Parameterkombinationen $x \in X_1 \times \cdots \times X_n$ durchzuführen. Anhand der dadurch entstandenen Simulationsdaten berechnet die *Evaluation* einen Zahlenwert, der als Funktionswert $f(x)$ der Zielfunktion genutzt wird. Abbildung 3.1 zeigt die Komponenten des Frameworks.

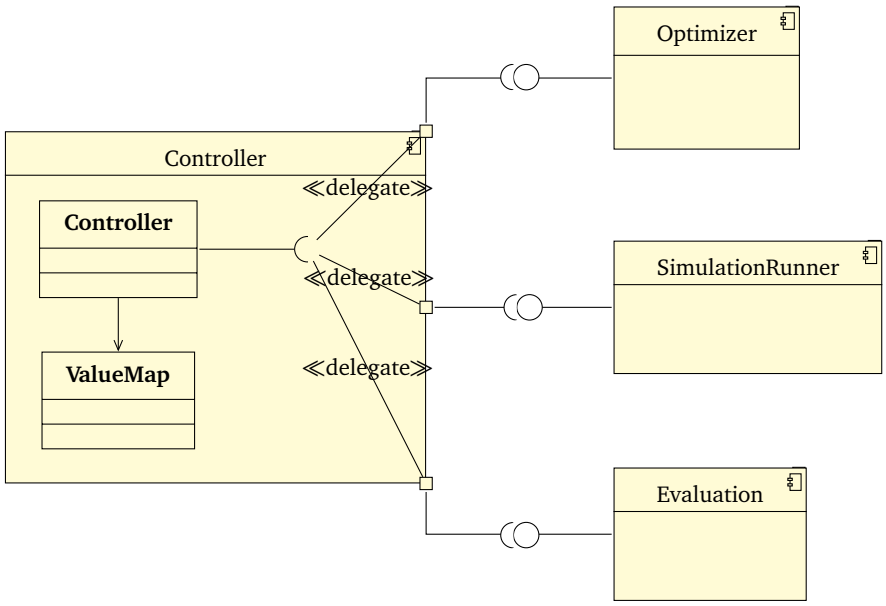


Abbildung 3.1 – Komponenten des *Simopticon*-Frameworks

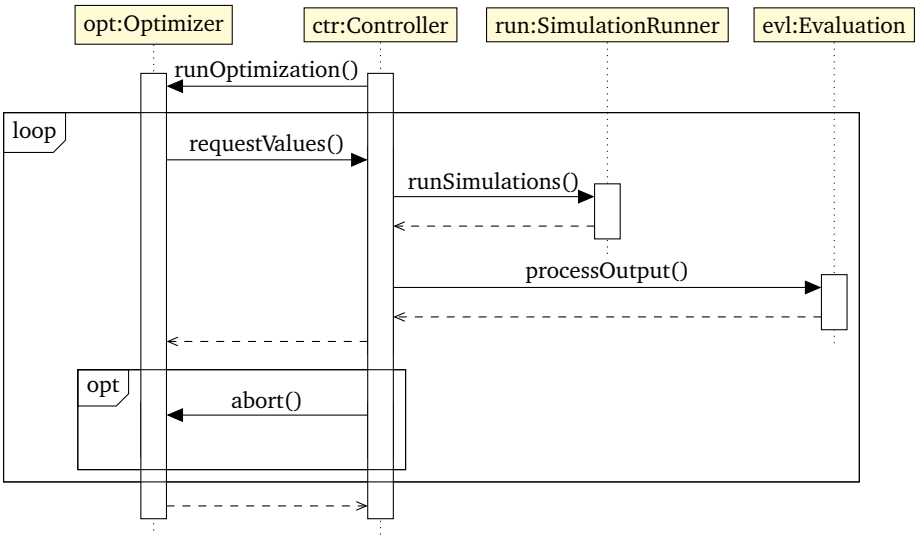


Abbildung 3.2 – Ablauf einer Optimierung

Der Ablauf einer Optimierung ist in Abbildung 3.2 dargestellt. Zu Beginn übergibt der *Controller* die Kontrolle an den *Optimizer*. Wenn die dort implementierte Strategie die Funktionswerte $f(x_1), \ldots, f(x_k)$ benötigt, fragt sie diese Werte vom *Controller* ab. Dieser übermittelt die Kombinationen an den *SimulationRunner*, welcher die entsprechenden Simulationen durchführt. Je Parameterkombination gibt dieser einen Pfad zu den entsprechenden Ergebnisdateien sowie eindeutige Kennungen der Simulationsläufe zurück. Diese leitet der *Controller* an das Evaluationsmodul weiter, welches anhand der Simulationsdaten den Funktionswert berechnet. Die so bestimmten Funktionswerte $f(x_1), \ldots, f(x_k)$ werden dem *Optimizer* übergeben, welcher mit diesen nun weiterrechnen kann, bis erneut Funktionswerte abgefragt werden müssen.

Dieses Vorgehen wiederholt sich, bis der *Optimizer* die Kontrolle an den *Controller* zurückgibt. Dies geschieht, wenn sich die Optimierungsstrategie selbst beendet oder der *Controller* die Optimierung abbricht. Ab diesem Punkt gilt die Optimierung als abgeschlossen. Der *Controller* gibt die Ergebnisse der Optimierung aus und das Programm wird beendet.

3.1.2 Implementierungen der Controller-Klasse

Aufgabe des *Controller*s ist die Kommunikation mit dem Nutzer sowie die Steuerung des Optimierungsvorgangs. Die in diesem Kontext relevanten Klassen sind in Abbildung 3.3 aufgezeigt.[4]

Sobald der Nutzer den Optimierungsvorgang von der Kommandozeile startet, wird in selbiger ein Statusbericht angezeigt, der regelmäßig aktualisiert wird. In diesem wird neben dem besten bisher gefundenen Wert der allgemeine Status von *Optimizer*, *SimulationRunner* und *Evaluation* (nachfolgend „Module“ genannt) angezeigt. Dazu kommt eine Statusleiste, die anzeigt, welches der Module im Moment arbeitet. Die Ausgabe erzeugt der *Controller* mithilfe eines *StatusBar*-Objektes. Dieses fragt je Modul dessen Namen und Statusbericht ab. Dazu nutzt es die Funktionalität vom *Status*-Interface, das alle Module implementieren. Ist die Optimierung beendet, gibt der *Controller* – ebenfalls mithilfe von *StatusBar* – die besten gefundenen Funktionswerte aus.

Wie im vorigen Abschnitt beschrieben, kann die Optimierung durch den Nutzer abgebrochen werden. Ein Abbruch wird dadurch herbeigeführt, dass der Nutzer dem ausführende Prozess das POSIX-Signal SIGINT (nach ISO/IEC/IEEE 9945) sendet. Dieses löst die Abbruchfunktion des *Controllers* aus, die wiederum den *Optimizer* abbricht. Die Abbruchfunktion ist im *Abortable*-Interface definiert, das von beiden implementiert wird. Sollte der Abbruch zu lange dauern – zum Beispiel durch zu

[4] Hierbei sind Implementierungsdetails wie Klassenvariablen und Hilfsfunktionen sowie Parameter und Rückgabewerte von Funktionen der Übersichtlichkeit halber ausgelassen worden.

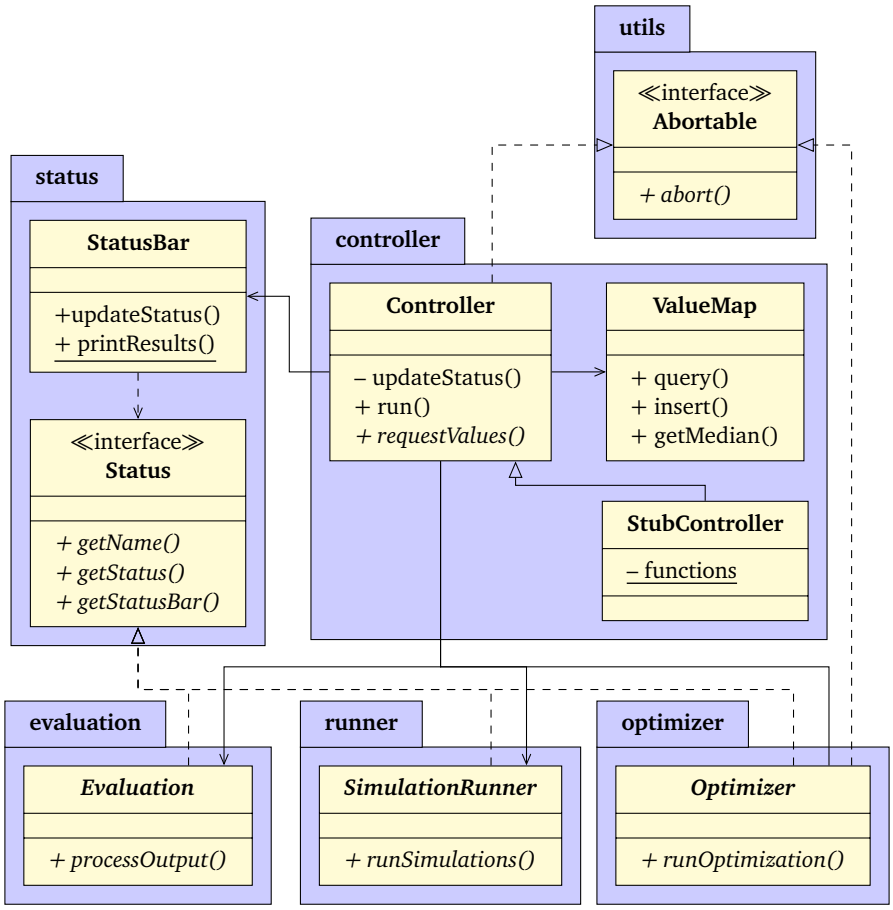


Abbildung 3.3 – Struktur des *controller*-Packages

langsames Beenden der Simulationen – kann der Nutzer den Prozess mithilfe eines zweiten SIGINT-Signals endgültig beenden. In diesem Fall ist undefiniert, ob die Ergebnisse noch ausgegeben werden können.

Zentrale Funktion des *Controller*s ist die Abfrage von Werten der Zielfunktion. Hat er durch den in Abbildung 3.2 beschriebenen Ablauf einen Funktionswert zu einer Parameterkombination ermittelt, speichert er diesen in einer *ValueMap*. Dadurch können teure Simulationen und Evaluationen gespart werden, sollte dieselbe Parameterkombination später erneut abgefragt werden. Zudem können durch die zentrale Speicherung leicht Informationen über die abgetasteten Werte, wie beispielsweise Median oder Minimum, ermittelt werden.

Um implementierte *Optimizer* anhand einfacher Testfunktionen zu erproben, kann *Simopticon* neben der Standardimplementierung des *Controller*s auch den *StubController* nutzen. Dieser verhält sich aus Sicht des *Optimizer*s wie ein *Controller*,

führt jedoch keine echten Simulationen durch. Vielmehr evaluiert er eine vorab definierte Zielfunktion.

3.1.3 Parameter-Klasse

Um die mithilfe von *Simopticon* optimierten Parameter darzustellen, wird das *parameter*-Package benutzt, dessen Struktur in Abbildung 3.4 dargestellt ist. Für jeden der *n* zu optimierenden Parameter wird zu Beginn der Optimierung eine *ParameterDefinition* erstellt. Diese definiert Maximal- und Minimalwert sowie optional Einheit und Konfiguration des Parameters. Maximum und Minimum können hierbei vom *Optimizer* genutzt werden, um den Parameterraum zu definieren, in dem die Zielfunktion optimiert werden soll. Einheit und Konfiguration sind allgemeine Textfelder, die der *SimulationRunner* für die Durchführung der Simulationsläufe nutzen kann (siehe Abschnitt 3.3.3.1).

Die abstrakte Klasse *Parameter* stellt einen Container für die Belegung eines Parameters dar. Jeder *Parameter* wird einer *ParameterDefinition* und damit einem der zu optimierenden Parameter zugeordnet. Die Klassen *ContinuousParameter* und *DiscreteParameter* erben von *Parameter* und implementieren kontinuierliche

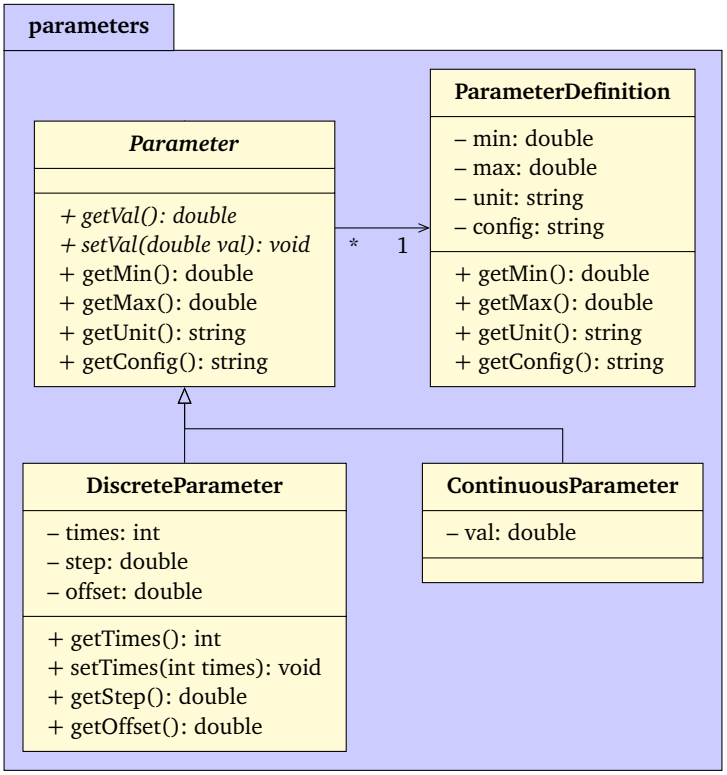


Abbildung 3.4 – Struktur des *parameters*-Packages

beziehungsweise diskrete Parameterwerte. *ContinuousParameter* speichert den Wert dabei direkt als Fließkommazahl, während *DiscreteParameter* den diskreten Charakter der Werte darstellt, indem er die Werte *step* und *offset* als Fließkomma- sowie den Wert *times* als Ganzzahl speichert. Aus diesen errechnet er den Parameterwert *val* nach folgender Gleichung:

$$val = times \cdot step + offset \tag{3.1}$$

Nach Erstellung des *DiscreteParameters* ist allein die Änderung der *times*-Komponente möglich, um nur Vielfache von *step* als Parameterwert zu erlauben.

Durch die Abstraktion von *Parameter* sind auch weitere Implementierungen möglich, falls solche für später implementierte Optimierungsprobleme notwendig sind. Denkbar wäre hier zum Beispiel eine Implementierung, die Parameter auf Basis von Enumerationen ermöglicht, um zwischen verschiedenen Optionen der Simulationsumgebung zu wechseln.

3.1.4 Konfiguration

Vor der Ausführung muss *Simopticon* mithilfe von Dateien im Format der JavaScript Object Notation (JSON) konfiguriert werden. Der Pfad zur Hauptkonfiguration muss mit dem Konsolenaufruf des Programms als Argument übergeben werden. Die Konfigurationsdateien werden mithilfe eines Open Source JSON-Parsers[5] eingelesen.

In der Hauptkonfiguration ist festgelegt,

- mit welcher Frequenz die *StatusBar* mindestens aktualisiert wird,
- wie viele der besten Ergebnisse ausgegeben werden,
- ob die Simulationsdaten der besten Ergebnisse gelöscht werden,
- welche Parameter optimiert werden sowie
- welcher *Optimizer*, welcher *SimulationRunner* und welche *Evaluation* genutzt werden.

Die Definition der zu optimierenden Parameter ist dabei in eine externe Datei ausgelagert. In der Hauptkonfiguration wird allein ein Verweis auf diese gespeichert. Dies ermöglicht die Nutzung von verschiedenen Parameterlisten, ohne umfangreiche Veränderungen in der Hauptkonfiguration vorzunehmen.

Nach dem gleichen Prinzip sind auch die Konfigurationen der einzelnen Module festgelegt. Neben der Information, welche Implementierung jeweils genutzt werden soll, ist in der Hauptkonfiguration ein Verweis zur Konfiguration der Komponente

[5] `https://json.nlohmann.me`

hinterlegt. Dies vereinfacht es, für unterschiedliche Implementierungen desselben Moduls unterschiedlich strukturierte JSON-Konfigurationen zu verwenden. Das ist von Vorteil, da unterschiedliche Implementierungen meist unterschiedliche Optionen haben.

3.2 Optimierungsmodul

Im Folgenden wird eine Implementierung des *Optimizer*s vorgestellt. Dazu wird MRDIRECT mit ADC kombiniert. Die Optimierungsstrategie arbeitet also nach der in Abschnitt 2.2 vorgestellten DIRECT-Methode, mithilfe der von Liu, Zeng und Yang [18] vorgeschlagenen Ebenen und der Unterteilungsstrategie von Sergeyev und Kvasov [20]. Der Aufbau des Optimierers ist in Abbildung 3.5 dargestellt.

3.2.1 Speicherung der Hyperquader

Wie in Abschnitt 2.2 gezeigt, basiert die DIRECT-Methode auf der Partitionierung eines Parameterraumes $P = [0,1]^n$ in mehrere Hyperquader. Um diese im Speicher darzustellen, soll im Folgenden eine Baumstruktur zur Beschreibung der Partitionierung und deren Implementierung in *Simopticon* vorgestellt werden.

3.2.1.1 Baumstruktur

Um die Teilungsstrategie von ADC umzusetzen, müssen je Hyperquader die abgetasteten Eckpunkte bekannt sein. Speichert man jeden Hyperquader mit dessen Eckpunkten, steigt der dafür benötigte Speicherplatz mit der Anzahl an Dimensionen n. Zudem nutzt ADC aus, dass sich die Eckpunkte unterschiedlicher Hyperquader überschneiden, was zu redundanter Speicherung führte, wenn die genutzte Datenstruktur jeden Hyperquader als Paar von Eckpunkten darstellen würde. Jones, Perttunen und Stuckman [7] empfehlen deshalb, die Partitionierung des Suchraumes als Baum aufzufassen und anhand dessen Informationen über die Partitionierung nachzuvollziehen (vergleiche Abschnitt 2.2.3.2).

In Abbildung 3.6 ist die Korrelation zwischen Unterteilung und Baumdarstellung aufgezeigt. Dort werden Hyperquader mit P_t^k bezeichnet, wobei $t \in \mathbb{N}$ dessen Tiefe im Baum und $k \in \{1, \ldots, 3^t\}$ die Nummer des Hyperquaders ist, die auf Ebene t eindeutig ist. Wurzel des Baumes ist der Hyperquader P_0^1, der dem gesamten Parameterraum P entspricht. Wird ein Hyperquader P_t^k geteilt, entstehen daraus drei kleinere Hyperquader P_{t+1}^{3k-2}, P_{t+1}^{3k-1} und P_{t+1}^{3k}, die im Baum als dessen Kindknoten repräsentiert werden. Die Partitionierung in Abbildung 3.6a ist entstanden, indem P vertikal in die Hyperquader P_1^1 (orange), P_1^2 (oliv) und P_1^3 geteilt wurde. P_1^1 und P_1^2 wurden daraufhin jeweils horizontal geteilt. Es ist ersichtlich, dass für die Auswahl

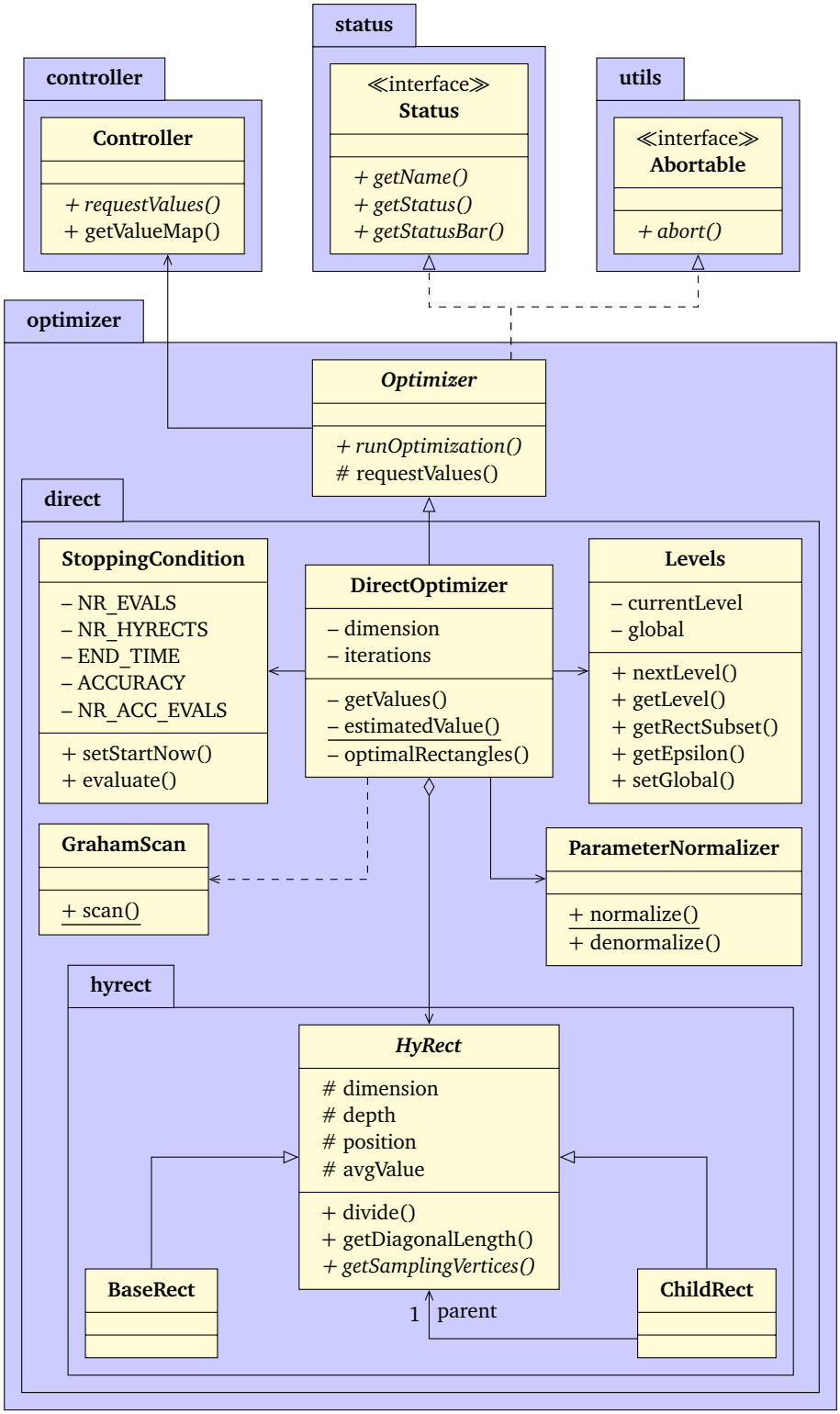
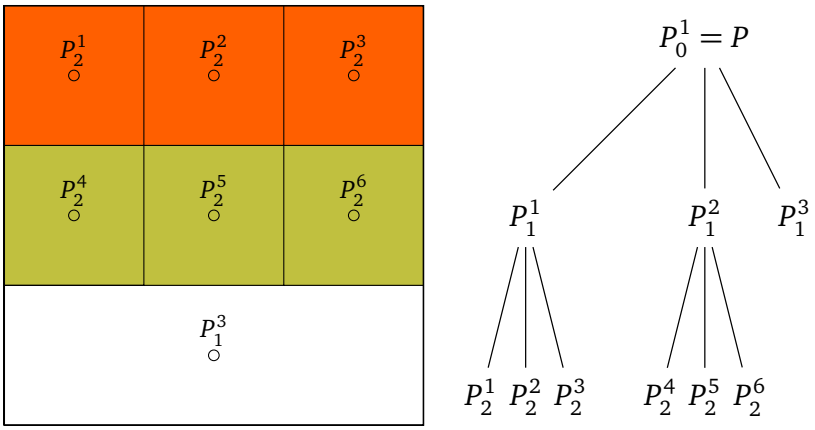


Abbildung 3.5 – Struktur des *optimizer*-Packages



(a) Unterteilung von P **(b)** Baumdarstellung

Abbildung 3.6 – Baumdarstellung einer Unterteilung

der zu teilenden Hyperquader allein die Blätter des Baumes betrachtet werden müssen, da diese die Hyperquader der aktuellen Partitionierung des Parameterraumes darstellen.

Zur Anwendung des DIRECT-Algorithmus müssen für jedes Blatt P_t^k des Baumes dessen abgetastete Eckpunkte $\vec{a}$ und $\vec{b}$ sowie deren Abstand $d = \|\vec{b} - \vec{a}\|$ abgeleitet werden. Der Mittelwert zwischen $f(\vec{a})$ und $f(\vec{b})$ kann in der Datenstruktur zu P_t^k direkt gespeichert werden.

Die Eckpunkte eines Hyperquaders P_t^k, können rekursiv anhand der Baumstruktur ermittelt werden. Gilt $k = 1$ und $t = 0$, handelt es sich um den gesamten Parameterraum P, dessen Eckpunkte als $(0,\ldots,0)^T$ und $(1,\ldots,1)^T$ festgelegt sind. Andernfalls können die Eckpunkte $\vec{a}, \vec{b}$ von P_t^k mithilfe von Gleichung (2.22) aus den Eckpunkten $\vec{a}', \vec{b}'$ von $P_{t-1}^{\lceil k:3 \rceil}$, dem Elternknoten von P_t^k, abgeleitet werden. Diese können wiederum rekursiv nach dem gleichen Verfahren berechnet werden. Um Gleichung (2.22) anzuwenden, muss einerseits bekannt sein, in welcher Dimension der Elternknoten geteilt wurde. Da zu teilende Dimensionen beginnend bei 1 in aufsteigender Reihenfolge gewählt werden, lässt sich die Dimension i, in der $P_{t-1}^{\lceil k:3 \rceil}$ geteilt wurde, anhand von $t-1$ errechnen. Es gilt:

$$i = (t-1) \mod n + 1 \tag{3.2}$$

Andererseits muss bestimmt werden, ob es sich bei P_t^k um den linken, mittleren oder rechten Kindknoten von $P_{t-1}^{\lceil k:3 \rceil}$ handelt. Hier gilt laut Definition des Baumes:

$$richtung\left(P_t^k\right) = \begin{cases} \text{links}, & \text{wenn } k \equiv 1 \mod 3 \\ \text{mittig}, & \text{wenn } k \equiv 2 \mod 3 \\ \text{rechts}, & \text{wenn } k \equiv 0 \mod 3 \end{cases} \tag{3.3}$$

Mit diesen Informationen, können die Knoten $\vec{u}$ und $\vec{v}$ mithilfe von Gleichung (2.22) bestimmt werden. Handelt es sich bei P_t^k um den linken Kindknoten, gilt $\vec{a} = \vec{a}'$ und $\vec{b} = \vec{u}$. Analog dazu gilt bei rechten Kindknoten $\vec{a} = \vec{v}$ und $\vec{b} = \vec{b}'$. Für mittlere Kindknoten gilt $\vec{a} = \vec{u}$ und $\vec{b} = \vec{v}$.

Der Abstand $d = \|\vec{b} - \vec{a}\|$ der Eckpunkte von P_t^k kann zwar mithilfe der euklidischen Distanz zwischen $\vec{a}$ und $\vec{b}$ ermittelt werden, lässt sich jedoch auch direkt aus der Tiefe t im Baum ableiten, da alle Hyperquader auf Ebene t des Baumes t-mal geteilt wurden. Entsprechend müssen hierfür $\vec{a}$ und $\vec{b}$ nicht bestimmt werden, was die zuvor beschriebene, komplexere, rekursive Funktion umgeht. Nach der Teilungsstrategie von ADC, sind alle P_t^k mit $n \mid t$ Hyperwürfel der Seitenlänge $3\delta = 3^{-\lfloor t:n \rfloor}$. Daraus folgt, dass P_t^k im Allgemeinen $l = t \mod n$ Seiten der Länge δ hat – nämlich jene, die seit der letzten Würfeltiefe geteilt wurden. Entsprechend hat der Hyperquader $(n-l)$ Seiten, die noch nicht geteilt wurden und somit 3δ lang sind. Mit diesen Seitenlängen kann nun auch ohne die Koordinaten der Eckpunkte $\vec{a}$ und $\vec{b}$ deren Abstand d mit der euklidischen Distanz berechnet werden:

$$\begin{aligned} d &= \sqrt{l \cdot \delta^2 + (n-l) \cdot (3\delta)^2} \\ &= 3\delta \cdot \sqrt{n - \frac{8}{9} l} \\ &= 3^{-\lfloor t:n \rfloor} \cdot \sqrt{n - \frac{8}{9} \cdot (t \mod n)} \end{aligned} \tag{3.4}$$

Es ist ersichtlich, dass der Abstand zwischen den abgetasteten Eckpunkten allein von der Tiefe t im Baum und der Anzahl der Dimensionen n abhängig ist.

3.2.1.2 Datenstruktur

Den Vorbetrachtungen des vorigen Abschnitts folgend, würde es für die Implementierung des DIRECT-Algorithmus reichen, zu jedem Blatt P_t^k des Baumes dessen Tiefe t, dessen Nummer k sowie den durchschnittlichen Wert $\frac{f(\vec{a})+f(\vec{b})}{2}$ an dessen abgetasteten Eckpunkten zu speichern. Alle weiteren relevanten Informationen lassen sich aus diesen Werten ableiten. In der Implementierung können die Blätter des Baumes jedoch aufgrund technischer Beschränkungen nicht allein als Container dieser Werte dargestellt werden. Grund hierfür ist das exponentielle Wachstum

der Nummern k. Um alle Hyperquader P_t^k auf Ebene t zu referenzieren werden Nummern $k \in \{ 1, 2, \ldots, 3^t \}$, also 3^t Nummern benötigt. Die größte Ganzzahleinheit in C++ nutzt 64 bit, kann also 2^{64} Werte darstellen. Das bedeutet, dass höchstens $t_{max} = \lfloor \log_3 2^{64} \rfloor = 40$ Ebenen dargestellt werden können, was besonders bei Zielfunktionen in vielen Dimensionen eine geringe Granularität bedeutet.

Um tiefere Bäume zu ermöglichen, nutzt die im *hyrect*-Package implementierte Datenstruktur eine an das Kompositum-Entwurfsmuster von Gamma u. a. [26] angelehnte Struktur (siehe Abbildung 3.5). Hyperquader werden durch Objekte der abstrakten Klasse *HyRect* repräsentiert. Diese hat zwei erbende Klassen – *BaseRect* und *ChildRect*. *BaseRect* stellt dabei den Wurzelknoten eines Baumes, also den gesamten Parameterraum dar. *ChildRect* kann beliebige Hyperquader darstellen und unterscheidet sich von *BaseRect* allein durch die Tatsache, dass *ChildRect* einen Verweis auf dessen Elternknoten besitzt. Dadurch entsteht also eine Baumstruktur, in der Kindknoten auf ihren Elternknoten verweisen. Um die im vorigen Abschnitt beschriebene rekursive Berechnung der abgetasteten Eckpunkte zu ermöglichen, muss jedes *ChildRect* zudem speichern, ob es der linke, mittlere oder rechte Kindknoten des referenzierten Elternknotens ist, da dies sonst nicht aus der Datenstruktur hervorgeht.

3.2.2 Ablauf des Algorithmus

Die DIRECT-Methode wurde in der Klasse *DirectOptimizer*, die von *Optimizer* erbt, implementiert (siehe Abbildung 3.5). Während des Optimierungsvorganges werden die aktuellen *HyRect*s, also die Blätter des oben beschriebenen Baumes, in einer Map-Datenstruktur verwaltet. Diese nutzt die Tiefe der Knoten als Schlüssel und speichert dazu jeweils eine Menge aller Blätter, die sich auf dieser Tiefe befinden. Die Hyperquader in diesen Mengen sind aufsteigend nach dem durchschnittlichen Wert an ihren Eckpunkten $\frac{f(\vec{a})+f(\vec{b})}{2}$ sortiert. Wird ein Hyperquader geteilt, wird er aus der Datenstruktur entfernt und stattdessen werden seine Kindknoten eingefügt.

Phase	Ebene i	ς_i	ε_i
global	3	50 %	10^{-5}
lokal	2	100 %	10^{-5}
	1	95 %	10^{-7}
	0	4 %	0

Tabelle 3.1 – Parameter der *Levels*-Klasse

3.2.2.1 Verwendung von Ebenen

Es wird ein neues Ebenensystem eingeführt, das die Ansätze von MrDirect und ADC vereinen soll und in der *Levels*-Klasse implementiert ist. Es besteht aus vier Ebenen, die in Tabelle 3.1 zusammengefasst sind. Wie von Liu u. a. [19] vorgeschlagen, wird der ε-Parameter auf den verschiedenen Ebenen variiert. Zudem wird jeweils eine andere Teilmenge der Partitionierung für die weitere Teilung in Betracht gezogen. Auf den lokalen Ebenen $i = 0, 1, 2$ sind das jeweils die kleinsten ς_i, auf der globalen Ebene $i = 3$ die größten 50 % der Hyperquader.

Ein Ebenenwechsel kann nach jeder Iteration auftreten. Zu Beginn befindet sich der *DirectOptimizer* in der lokalen Phase auf Ebene 2. Solange sich der Optimierer in der lokalen Phase befindet, wird nach dem in Abschnitt 2.2.4.1 beschriebenen W-Zyklus zwischen den Ebenen 2, 1 und 0 gewechselt. Dabei wird immer nach vier Iterationen durch folgende Bedingung überprüft, ob ein signifikant besseres Ergebnis gefunden wurde:

$$\phi \leq \phi' - \varepsilon_3 \cdot |\phi' - \tilde{f}| \tag{3.5}$$

Dabei bezeichnen ϕ und ϕ' den besten abgetasteten Wert nach beziehungsweise vor den vergangenen vier Iterationen. Ist diese Bedingung nicht erfüllt, wechselt der Optimierer in die globale Phase und somit auf Ebene 3. Auf dieser bleibt er, bis Gleichung (3.5) erfüllt ist – wobei ϕ' in diesem Fall das Optimum zu Beginn der globalen Phase darstellt – maximal aber bis 35 Iterationen auf Ebene 3 durchgeführt wurden und wechselt danach zurück in die lokale Phase.

Beim Wechsel in die lokale Phase beginnt *DirectOptimizer* nicht zwingend am Anfang des W-Zyklus, sondern setzt dort an, wo dieser zuletzt unterbrochen wurde. Dies ist in Abbildung 3.7 veranschaulicht – dort wird nach vier Iterationen in der lokalen Phase zur globalen gewechselt. Nach drei Iterationen auf Ebene 3 erfolgt der Wechsel zurück zur lokalen Phase, in der der angefangene W-Zyklus beendet wird.

3.2.2.2 Initialisierung und Abbruchbedingung

Zur Initialisierung der Optimierung wird ein *BaseRect* in die Map-Datenstruktur eingefügt und dessen Eckpunkte $(0, \ldots, 0)^T$ und $(1, \ldots, 1)^T$ ausgewertet. Da Direct einen Hyperwürfel $P = [0, 1]^n$ als Parameterraum annimmt, müssen die auszuwertenden Punkte auf den tatsächlichen Parameterraum $\tilde{P}$ übertragen werden. Diese Aufgabe übernimmt, der *ParameterNormalizer*, der eine Normalisierungsfunktion $normalize : \tilde{P} \to P$ sowie deren Umkehrfunktion $denormalize = normalize^{-1}$ implementiert. Gilt beispielsweise $\tilde{P} = [0, \frac{1}{2}] \times [5, 7]$, würde $denormalize(1, \frac{1}{2}) = (\frac{1}{2}, 6)$ gelten. Die denormalisierten Parameterkombinationen können dann an den *Controller* übergeben werden, welcher die entsprechenden Funktionswerte bestimmt.

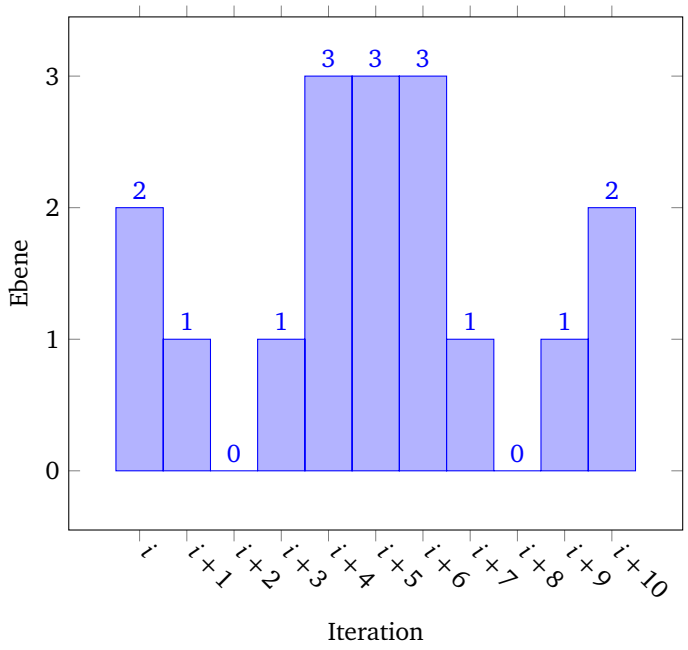


Abbildung 3.7 – Möglicher Verlauf der Ebenen

Nach der Initialisierung beginnt die iterative Suche nach einem globalen Minimum der Zielfunktion. Zu Beginn jeder Iteration wird überprüft, ob der Algorithmus beendet werden soll. Dies ist der Fall, wenn *DirectOptimizer*, der indirekt von *Abortable* erbt, abgebrochen wurde oder wenn eine der vordefinierten Abbruchbedingungen eingetreten ist. Die Abbruchbedingungen werden mit der Hilfsklasse *StoppingCondition* überprüft. In der Konfigurationsdatei des *DirectOptimizer*s können folgende Abbruchbedingungen definiert werden:

- Abbruch ab einer bestimmten Zahl an Evaluationen (Simulationsläufen),
- Abbruch ab einer bestimmten Zahl an Hyperquadern in der Partitionierung,
- Abbruch nach Ablauf eines Zeitlimits und
- Abbruch, wenn in einer bestimmten Zahl an Iterationen keine signifikante Verbesserung erzielt wurde.

Sobald eine dieser Bedingungen zutrifft, wird die Optimierung beendet. Dies ermöglicht dem Nutzer eine genaue Spezifikation, bis zu welcher Genauigkeit oder welchem Zeitpunkt eine Optimierung stattfinden soll.

3.2.2.3 Iteration

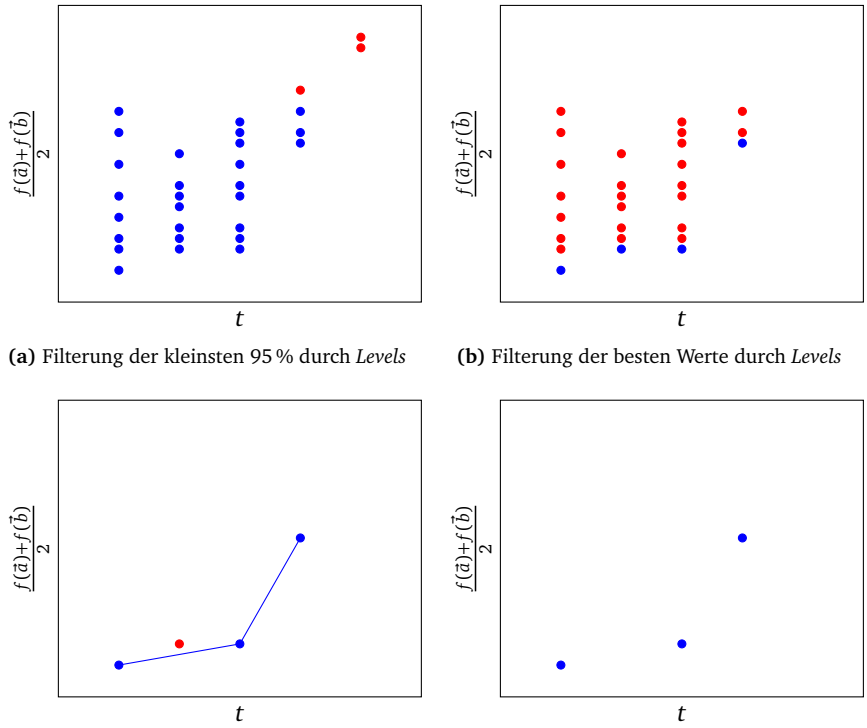
Ist die Abbruchbedingung noch nicht erfüllt, werden zuerst die zu teilenden Hyperquader ermittelt. Dies geschieht nach dem in Algorithmus 3.1 beschriebenen Verfahren. Dafür werden die Map-Datenstruktur der aktuellen Hyperquader *rectMap*, der Parameter ε_i der aktuellen Ebene i, der beste bisher abgetastete Wert ϕ und der Median aller abgetasteten Werte $\tilde{f}$ benötigt. ϕ und $\tilde{f}$ werden aus der *ValueMap* des *Controller*s bestimmt. ε_i wird mithilfe eines Objektes der *Levels*-Klasse bestimmt. Dieses speichert, auf welcher Ebene die aktuelle Iteration durchgeführt wird und gibt die entsprechenden Werte zurück. Es ist auch für die Vorfilterung der Hyperquader nach Größe mithilfe des Parameters ς_i zuständig, was in Algorithmus 3.1 vereinfacht durch die Funktion *filter* dargestellt ist.

Zuerst wird nun diese Vorfilterung durchgeführt. Dies ist in Abbildung 3.8 am Beispiel einer Iteration auf Ebene 1 dargestellt. Wie in Abbildung 3.8a gezeigt, werden dabei nur die kleinsten $\varsigma_1 = 95\,\%$ der Hyperquader betrachtet. Zudem wird für alle Hyperquader mit dem selben Durchmesser nur derjenige mit dem besten Durchschnittswert betrachtet, was in Abbildung 3.8b abgebildet ist. Dabei wird der Aufbau der *rectMap* ausgenutzt. Diese speichert jeweils alle Hyperquader mit derselben Größe in einer Menge. In Abbildung 3.8 sind dies alle Punkte, die dieselbe Abszisse haben. Da diese Mengen aufsteigend nach Wert sortiert sind, können je Menge alle Hyperquader, die nicht an deren Anfang stehen, herausgefiltert werden, da diese nicht zur unteren rechten konvexen Hülle gehören können, die der Graham-Scan später bestimmt.

Nach dieser Vorfilterung wird mit der Klasse *GrahamScan* die untere rechte konvexe Hülle der in *points* liegenden Punkte bestimmt. Diese entspricht allen Hyperquadern, die die erste Bedingung aus Gleichung (2.15) erfüllen. *GrahamScan* gibt eine Menge von Paaren (*hull*) zurück, in der jedem Hyperquader der Hülle die größte Änderungsrate K zugeordnet ist, für die er diese Bedingung erfüllt. Diese größte Änderungsrate entspricht dem Anstieg zwischen dem entsprechenden Punkt und dessen rechten Nachbarn in der konvexen Hülle.

Require: *rectMap*, ε_i, ϕ, $\tilde{f}$
1: $points \leftarrow filter(rectMap)$
2: $hull \leftarrow grahamScan(points)$
3: $optimal \leftarrow \emptyset$
4: **for all** $(rect, K) \in hull$ **do**
5: **if** $boundary(rect, K) \leq \phi - \varepsilon_i \cdot |\phi - \tilde{f}|$ **then**
6: $optimal \leftarrow optimal \cup \{ rect \}$
7: **end if**
8: **end for**
9: **return** *optimal*

Algorithmus 3.1 – Feststellung der optimalen Hyperquader



(a) Filterung der kleinsten 95 % durch *Levels* **(b)** Filterung der besten Werte durch *Levels*

(c) Finden der konvexen Hülle durch *GrahamScan* **(d)** Auswahl potenziell optimaler Hyperquader

Abbildung 3.8 – Auswahl potenziell optimaler Hyperquader im *DirectOptimizer*

In der folgenden Schleife in Algorithmus 3.1 wird abschließend nach der zweiten Bedingung für optimale Hyperquader aus Gleichung (2.20) gefiltert. Dabei berechnet *boundary* die untere Grenze nach der folgenden Gleichung:

$$\hat{\mathrm{M}}\left(\vec{a},\vec{b},f,K\right)=\frac{f\left(\vec{a}\right)+f\left(\vec{b}\right)}{2}-K\cdot\|\vec{b}-\vec{a}\| \tag{3.6}$$

Dabei werden $\frac{f(\vec{a})+f(\vec{b})}{2}$ und $\|\vec{b}-\vec{a}\|$ wie in Abschnitt 3.2.1.2 beschrieben aus dem entsprechenden *HyRect* bestimmt. Gleichung (3.6) entspricht hier Gleichung (2.23), allerdings wird als Maß für die Größe eines Hyperquaders nicht dessen Abstand von Mittelpunkt zu den Ecken $\frac{\|\vec{b}-\vec{a}\|}{2}$, sondern dessen gesamte Diagonallänge $\|\vec{b}-\vec{a}\|$ verwendet.

Sind die potenziell optimalen Hyperquader bestimmt, werden diese geteilt und aus der *rectMap* entfernt. Die Werte der Zielfunktion an den Eckpunkten der dabei entstehenden Hyperquader werden vom *Controller* angefragt. Dabei werden diesem alle Eckpunkte in einem einzelnen Funktionsaufruf übergeben. Dies erleichtert es *SimulationRunner* und *Evaluation*, die Bearbeitung der Anfragen zu parallelisieren.

Nach der erfolgten Anfrage an den *Controller* werden die neuen *HyRect*s der Map-Datenstruktur der aktiven Hyperquader hinzugefügt.

Am Ende einer Iteration werden verschiedene Daten, wie die Anzahl an Hyperquadern oder die Anzahl an Evaluationen erhoben, die für die Auswertung der Abbruchbedingung von Interesse sind. Zudem wird die Ebene des *Levels*-Objektes wie in Abschnitt 3.2.2.1 beschrieben gewechselt. Die für einen Iterationsschritt nötigen Funktionsaufrufe sind in Abbildung 3.9 dargestellt.

3.3 Simulationsmodul

Im Folgenden wird eine Implementierung des *SimulationRunner*s vorgestellt. Diese ermöglicht es, automatisiert Simulationen mit dem Simulationsframework PLEXE durchzuführen. Der Aufbau der Simulationsausführung ist in Abbildung 3.10 dargestellt.

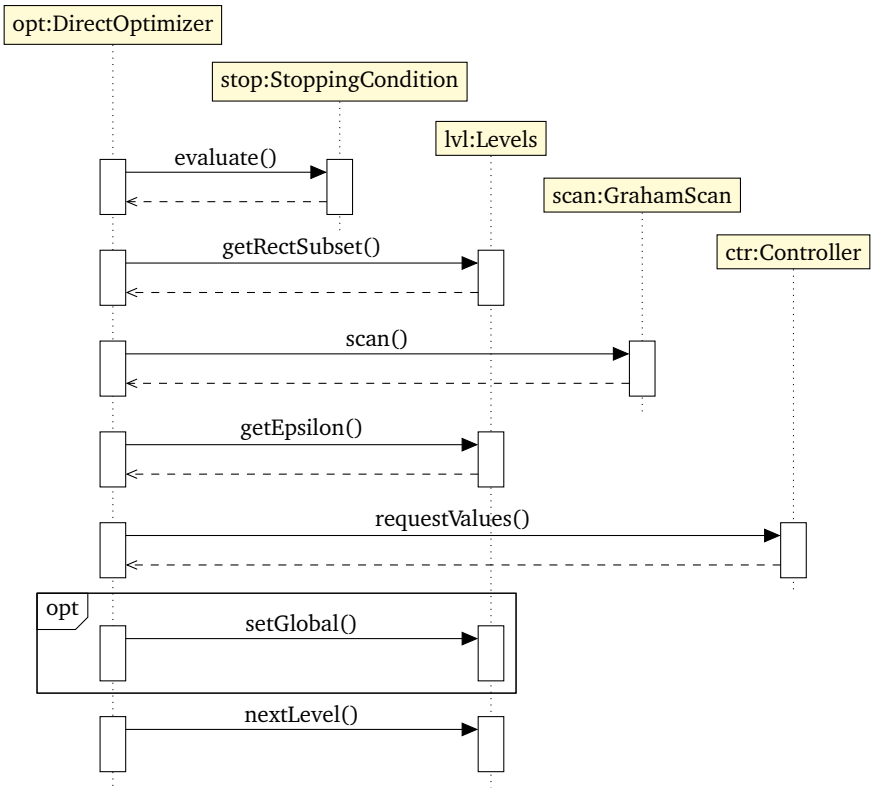


Abbildung 3.9 – Ablauf eines Optimierungsschrittes beim *DirectOptimizer*

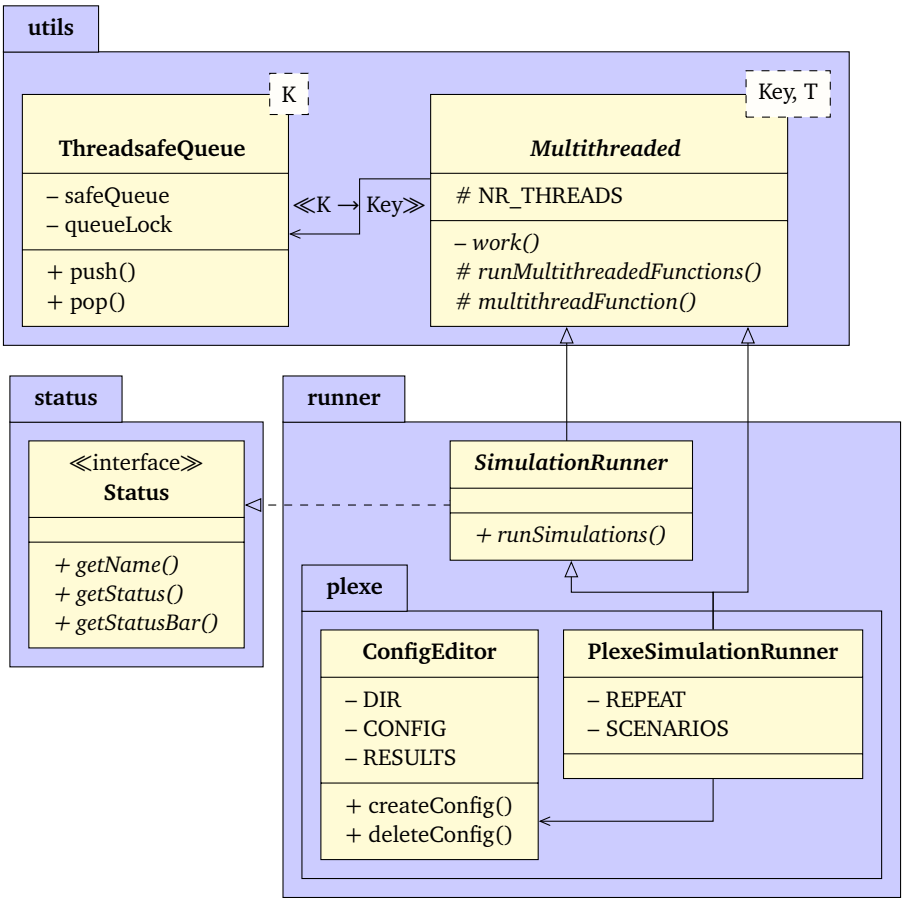


Abbildung 3.10 – Struktur des *runner*-Packages

3.3.1 Schnittstelle für Simulatoren

Wie in Abbildung 3.10 dargestellt, müssen Implementierungen einer Simulationsausführung von der abstrakten Klasse *SimulationRunner* erben. Diese implementiert das *Status*-Interface und erbt ihrerseits von der abstrakten, generischen Klasse *Multithreaded*.

Multithreaded ermöglicht es einer Klasse, die die gleiche Funktion (nachfolgend „Arbeitsfunktion“) auf mehreren Objekten ausführen muss, dies zu parallelisieren. Dazu müssen Klassen, die von *Multithreaded* erben, eine Arbeitsfunktion definieren. Die Typparameter *Key* und *T* stellen dabei den Typ des Arguments beziehungsweise des Rückgabewertes der Arbeitsfunktion dar. *Multithreaded* arbeitet dabei nach dem Thread-Pool-Entwurfsmuster [27]. Das bedeutet, dass *Multithreaded* eine vordefinierte Anzahl an Threads verwaltet, die Aufträge aus einer gemeinsamen FIFO-Liste abarbeiten, bis diese leer ist. Die FIFO-Liste ist in *ThreadsafeQueue* implementiert und ist gegenüber gleichzeitigem Zugriff aus mehreren Threads robust. Aufträge sind hier

durch Objekte des Typs *Key* repräsentiert. Jeder Thread nimmt solange Argumente aus der *ThreadsafeQueue* und berechnet den Funktionswert der Arbeitsfunktion bei Eingabe des Argumentes, bis alle Aufträge bearbeitet sind und die *ThreadsafeQueue* leer ist.

Im Falle des *SimulationRunner*s wird *Multithreaded* genutzt, um die Simulation von mehreren Parameterkombinationen $\vec{x}_1, \ldots, \vec{x}_k$ zu paralleliseren. Arbeitsfunktion ist also die Ausführung der mit einer Kombination $\vec{x}$, verbundenen Simulation(en). Argument der Arbeitsfunktion (*Key*) ist die entsprechende Parameterkombination $\vec{x}$, Rückgabewert (*T*) sind Dateipfad zu den Ergebnisdateien und Identifikatoren der einzelnen Simulationsdurchgänge. Wird *SimulationRunner* beauftragt, Simulationen durchzuführen, wird die zu diesem Zweck übergebene Liste von Parameterkombinationen an die Auftragsliste angehangen. Die tatsächliche Arbeitsfunktion – also die Ausführung der Simulation(en) – wird abhängig vom genutzten Simulationsframework in Klassen wie *PlexeSimulationRunner* implementiert, die von *SimulationRunner* erben.

3.3.2 PLEXE

PLEXE [4] ist ein Framework, das die Simulation kooperativen Fahrens in Platoons ermöglicht. Dazu erweitert es Vehicles in Network Simulation (VEINS) [28] und Simulation of Urban Mobility (SUMO) [29]. SUMO ist ein Verkehrssimulationspaket, mit welchem große Verkehrsnetze mikroskopisch modelliert werden können. Mit PLEXE wird SUMO um ACC- und CACC-Kontrollsysteme sowie Modelle für Motordynamiken erweitert. Zudem kann PLEXE verschiedene Manöver, Szenarien und Kommunikationsprotokolle im Kontext von Platooning simulieren. VEINS ist ein Framework, das umfangreiche Modelle für die Kommunikation von Verkehrsteilnehmern bereitstellt. Dazu nutzt es OMNeT++ [30], um die Kommunikation und SUMO, um den Verkehr zu simulieren. OMNeT++ ist eine C++-Simulationsbibliothek, die die Modellierung von Netzwerksimulationen mit diskreten Ereignissen ermöglicht. Der genaue Zusammenhang der verschiedenen Frameworks soll im Folgenden nicht weiter elaboriert werden, da *PlexeSimulationRunner* nur eine Schnittstelle zu PLEXE darstellt und dessen Funktionalität nicht erweitert.

Simulationen mit PLEXE können über Konsolenkommandos gestartet werden. Um dies zu tun muss unter anderem eine Konfigurationsdatei (meist „omnetpp.ini“ genannt) geschrieben werden. In dieser Datei werden verschiedene Simulationsparameter und Experimente festgelegt. Zu den dort konfigurierten Parametern gehören auch die Reglerparameter von Platoon-Reglern, für deren Optimierung *Simopticon* genutzt werden kann.

Im Kontext von PLEXE handelt es sich bei den in der omnetpp.ini definierten Experimenten um verschiedene Fahrtszenarien. Hier sind unter anderem die Szenarien

„Sinusoidal“ und „Braking“ nativ implementiert. Bei beiden Szenarien wird ein einzelner Platoon mit fester Anzahl Fahrzeugen simuliert, die zu Beginn der Simulation mit festem Abstand und fester Geschwindigkeit in die Simulation eingefügt werden. Beim Sinusoidal-Szenario beschleunigt und bremst der Platoonführer, sodass seine Geschwindigkeit einer Sinuskurve mit festgelegter Amplitude und Frequenz gleicht. Beim Braking-Szenario bremst der Platoonführer ab einem festgelegten Zeitpunkt ab, bis der Platoon zum stehen kommt.

Da es sich bei den Simulationen der Kommunikation zwischen Fahrzeugen um stochastische Simulationen handelt, ermöglicht es PLEXE außerdem, Simulationsdurchläufe mit den gleichen Parameterkombinationen aber unterschiedlichen Seeds für die Zufallsgeneratoren zu wiederholen. Die Anzahl dieser Wiederholungen wird ebenfalls in der Konfiguration festgelegt.

3.3.3 Implementierung der Simulationsausführung

Die Ausführung von Simulationen mit PLEXE wird durch die Klasse *PlexeSimulationRunner* implementiert. Wenn eine Simulation für die Parameterkombination $\vec{x}$ angefragt wird, wird eine neue Konfigurationsdatei mit den entsprechenden Werten erstellt und die entsprechenden Simulationsläufe werden parallelisiert gestartet.

3.3.3.1 Erstellung der Konfigurationsdatei

Um für eine gegebene Parameterkombination $\vec{x}$ eine neue Konfigurationsdatei zu erzeugen, nutzt *PlexeSimulationRunner* ein Objekt der Hilfsklasse *ConfigEditor*. Diese erstellt auf Basis einer in der Konfiguration des *PlexeSimulationRunner*s gewählten omnetpp.ini eine neue INI-Datei, in der die gegebenen Parameterwerte eingefügt werden. Dies ermöglicht es dem Nutzer, die Simulationen im Vorhinein auf gewohnte Weise in der gegebenen omnetpp.ini zu konfigurieren. Während der Simulation werden nur die zu optimierenden Simulationsparameter variiert.

Um die Parameterwerte zu setzen, werden das Konfigurations- und Einheitsfeld der *ParameterDefinition* genutzt (vergleiche Abschnitt 3.1.3). Einträge in INI-Dateien sind wie folgt aufgebaut:

$$Schlüssel = Wert\,[Einheit]$$

Der Schlüssel, an dem ein zu optimierender Parameter konfiguriert wird, wird im Konfigurationsfeld von *ParameterDefinition* gespeichert. Soll diesem Parameter der in *Parameter* gespeicherte Wert zugewiesen werden, wird in der omnetpp.ini bei Auftreten des definierten Schlüssels der entsprechende Wert samt Einheit eingefügt.

Neben den zu optimierenden Parametern und dem zu verwendenden Regler werden auch andere Optionen der omnetpp.ini angepasst. Diese sind in Tabelle 3.2

Schlüssel	Wert	Funktion
repeat	r	Anzahl Wiederholungen
output-*-file	"optResults"	Ordner für Simulationsdaten
debug-on-errors	*false*	bei Fehlern Debugger starten
print-undisposed	*false*	Fehlermeldung bei ungelöschten Objekten
cmdenv-autoflush	*false*	Regelmäßiges flushen des Ausgabepuffers
cmdenv-status-frequency	100 s	Frequenz der Statusausgabe

Tabelle 3.2 – Veränderte Optionen in generierten INI-Dateien

aufgelistet. Dazu gehören die Anzahl an Wiederholungen eines Experimentes r, die in der Konfiguration von *PlexeSimulationRunner* definiert werden kann und der Speicherort für Simulationsdaten. Die restlichen Optionen werden festgelegt, um die Ausgabe von PLEXE auf ein Minimum zu begrenzen und um auftretende Fehler zu ignorieren. Ausgaben von PLEXE werden durch *Simopticon* weder verarbeitet noch dem Nutzer ausgegeben, entsprechend verlangsamen zu viele Ausgabeoperationen unnötig die Simulationsdurchführung. Zudem geht *Simopticon* davon aus, dass die mit PLEXE simulierten Modelle bereits ausführlich getestet wurden und keine schwerwiegenden Fehler auftreten. *PlexeSimulationRunner* nimmt keine Fehlererkennung oder -behandlung vor.

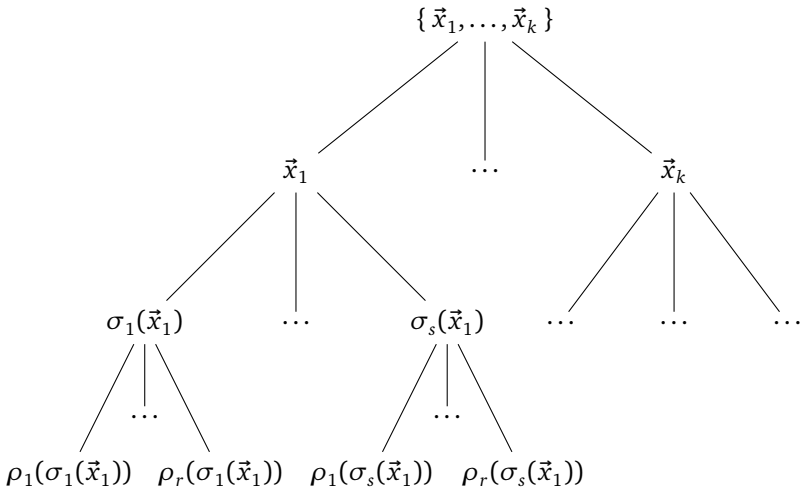


Abbildung 3.11 – Aufteilung der Simulationsdurchgänge

3.3.3.2 Parallelisierung von Simulationsläufen

Wie in Abschnitt 3.3.1 beschrieben, wird *SimulationRunner* mit einer Menge an Parameterkombinationen $\{ \vec{x}_1, \ldots, \vec{x}_k \}$ aufgerufen. Diese werden durch parallele Aufrufe an den *PlexeSimulationRunner* bearbeitet. Je Parameterkombination $\vec{x}$ können in PLEXE jedoch auch mehrere Simulationsläufe durchgeführt werden, was in Abbildung 3.11 dargestellt ist. In der Konfiguration des *PlexeSimulationRunner*s kann festgelegt werden, mit welchen Szenarien $\sigma_1, \ldots, \sigma_s$ jede Parameterkombination simuliert werden soll. Zudem können diese Simulationsläufe wie im vorigen Abschnitt beschrieben r-mal wiederholt werden. Dies ist in Abbildung 3.11 durch die Symbole $\rho_1, \ldots, \rho_r$ dargestellt.

Es müssen also je Aufruf $\vec{x}$ an den *PlexeSimulationRunner* p Simulationsdurchläufe gestartet werden mit:

$$\begin{aligned} p &= |\{ \rho_1(\sigma_1(\vec{x})), \ldots, \rho_r(\sigma_1(\vec{x})), \rho_1(\sigma_2(\vec{x})), \ldots, \rho_r(\sigma_s(\vec{x})) \}| \\ &= r \cdot s \end{aligned} \tag{3.7}$$

Da jeder dieser Durchläufe Rechenzeiten im Sekundenbereich benötigt, liegt es nahe, auch diese Durchläufe nicht sequenziell sondern parallel durchzuführen. Aus diesem Grund erbt *PlexeSimulationRunner* nicht nur indirekt über *SimulationRunner*, sondern auch direkt von *Multithreaded*. Während bei *SimulationRunner* die Bearbeitung der Parameterkombinationen $\vec{x}_1, \ldots, \vec{x}_k$ parallelisiert wird, wird in *PlexeSimulationRunner* die Ausführung eines Simulationsdurchlaufes $\rho_i(\sigma_j(\vec{x}))$ als Arbeitsfunktion gewählt. In Abbildung 3.11 ist erkennbar, dass *SimulationRunner* die Knoten der ersten Ebene parallelisiert. Jeder dieser parallelen Aufrufe parallelisiert wiederum jeweils die Ausführung der verschiedenen Simulationsläufe, also die Blätter des entsprechenden Teilbaumes.

Würde jeder Simulationslauf in einem eigenen Thread ablaufen, würden insgesamt $k \cdot s \cdot r$ Threads benötigt werden. Damit nicht zu viele Threads erstellt werden, die das System verlangsamen, kann in der Konfiguration des *PlexeSimulationRunner*s eine Maximalanzahl $thread_{max}$ an Threads festgelegt werden. So wird sichergestellt, dass je Aufruf des Simulationsmoduls maximal $\min(k \cdot s \cdot r, thread_{max})$ Threads gestartet werden.

3.4 Evaluationsmodul

Im Folgenden wird eine Implementierung der *Evaluation* vorgestellt. Diese bewertet, wie sehr die Fahrzeuge eines Platoons vom erwarteten Sicherheitsabstand zum direkten Vorausfahrer abweichen. Der Aufbau der Simulationsausführung ist in Abbildung 3.12 dargestellt.

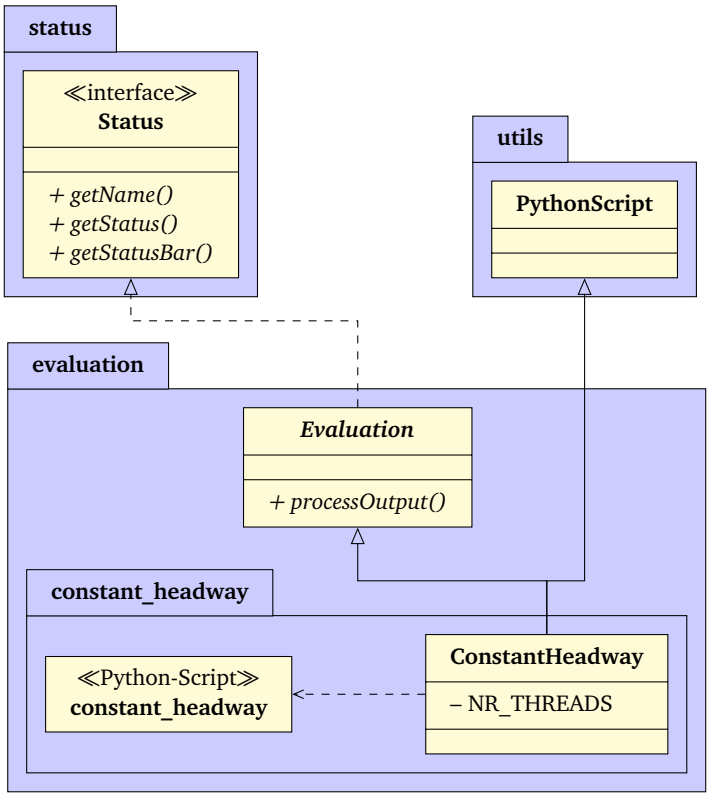


Abbildung 3.12 – Struktur des *evaluation*-Packages

3.4.1 Schnittstelle für Evaluationsfunktionen

Wie in Abbildung 3.12 dargestellt, müssen Implementierungen einer Evaluationsfunktion von der abstrakten Klasse *Evaluation* erben, die das *Status*-Interface implementiert. Eine Evaluationsfunktion muss anhand von Simulationsdaten einen Fließkommawert bestimmen, der als Bewertung des Experiments bezüglich eines Optimierungsziels verstanden wird. Je kleiner der Funktionswert, desto besser war das Experiment und umso besser war entsprechend die Wahl der Parameter. Diese Bewertung von Simulationen ist an statistische Verlustfunktionen angelehnt, allerdings kann der Verlust der durch *Evaluation* modellierten Zielfunktionen auch negativ werden. Da die Optimierungsstrategie des *Optimizer*s das Minimum der Zielfunktion sucht, wird negativer Verlust als gutes Simulationsergebnis gewertet.

Die Modellierung der Zielfunktion als Verlustfunktion erlaubt zudem simple, nichtinteraktive Pareto-Optimierung [31], also Optimierung in Bezug auf mehrere Optimierungsziele. Sollen beispielsweise mit einer Verlustfunktion Reglerparameter für Platoon-Regler auf Kettenstabilität bei möglichst geringem Sicherheitsabstand optimiert werden, kann dies als Linearkombination zweier Teilfunktionen modelliert

werden:

$$f = \lambda_{string} \cdot f_{string} + \lambda_{gap} \cdot f_{gap} \text{ mit } \lambda_{string}, \lambda_{gap} \in \mathbb{R} \tag{3.8}$$

Dabei bezeichnet f_{string} eine Verlustfunktion zur Bewertung der Kettenstabilität anhand der Simulationsdaten und f_{gap} eine Funktion, die den durchschnittlichen Abstand der Fahrzeuge im Platoon ausgibt. Die Koeffizienten λ_{string} und λ_{gap} können genutzt werden, um die beiden Ziele gegeneinander zu wichten.

3.4.2 Implementierung der Evaluation

Bei der implementierten Verlustfunktion f_{const} handelt es sich um eine naive Bewertung der Kettenstabilität des simulierten Platoons. Dazu berechnet f_{const} die mittlere, quadratische Abweichung der Fahrzeuge vom Sollabstand zum unmittelbar vorausfahrenden Fahrzeug. Ob diese Metrik tatsächlich Rückschlüsse auf die Kettenstabilität zulässt, wird im Folgenden nicht weiter betrachtet. f_{const} stellt eher eine beispielhafte Verlustfunktion dar, die zum Testen von *Simopticon* genutzt werden kann.

3.4.2.1 Definition der Verlustfunktion

In den Simulationsdaten eines Durchlaufes wird von PLEXE für jedes der n Fahrzeuge im Platoon eine Zeitreihe $v_i = (g_{t_1}, \ldots, g_{t_l})$ mit $i \in \{1, \ldots, n\}$ gespeichert, die die Abstände g des Fahrzeugs zum vorausfahrenden Fahrzeug an den Zeitpunkten $t_1, \ldots, t_l$ enthält. Bei jeder Simulation werden Daten für $s \cdot r$ Wiederholungen des Experiments erfasst, wobei s die Zahl der unterschiedlichen Szenarien und r die Anzahl der jeweiligen Wiederholungen ist (vergleiche Abschnitt 3.3.2). Sei $D = \{V_1, \ldots, V_{s \cdot r}\}$ die Menge der relevanten Daten je Durchlauf. Dabei sei $V_j = \{v_2, \ldots, v_n\}$ die Menge der oben beschriebenen Abstandsvektoren der Fahrzeuge im Platoon. Dabei gilt $v_1 \notin V_j$ für alle $j \in \{1, \ldots, s \cdot r\}$, da der Abstandsvektor des ersten Fahrzeugs von PLEXE zwar aufgenommen wird, allerdings nur 0 und somit keine relevanten Informationen enthält.

Die Berechnung von f_{const} findet nach der folgenden Formel statt:

$$f_{const}(D) = \frac{1}{s \cdot r} \sum_{j=1}^{s \cdot r} \sum_{v \in V_j} \frac{1}{\|v\|} \sum_{g \in v} (g - g')^2 \tag{3.9}$$

Dabei stellt g' den Sollabstand dar. Es wird also zuerst für jeden Vektor v die mittlere quadratische Abweichung vom Sollabstand $\left(\frac{1}{\|v\|} \sum_{g \in v} (g - g')^2\right)$ berechnet. Dieser Wert wird für alle Abstandsvektoren $v \in V_j$ aufaddiert, was den Gesamtfehler des Simulationslaufes repräsentiert. Aus den Gesamtfehlern aller $s \cdot r$ Simulationsläufe wird wiederum das arithmetische Mittel gebildet, was den Wert von f_{const} darstellt. Es gilt $f_{const} \geq 0$.

3.4.2.2 Implementierung in Python

Die Auswertung der Simulationsdaten wurde in der Klasse *ConstantHeadway* implementiert. Diese erbt von *Evaluation* und *PythonScript*. *PythonScript* ist eine Klasse, die bei Konstruktion mithilfe der Python/C API[6] ein Pythonskript einbettet und die Ausführung einer Funktion des Skriptes ermöglicht. Dies nutzt *ConstantHeadway*, um das Skript *constant_headway* einzubetten, in dem die eigentliche Berechnung der Verlustfunktion stattfindet. *ConstantHeadway* dient hier also nur als Adapterklasse, um Anfragen zur Auswertung der Daten an ein Pythonskript weiterzuleiten.

Die Verwendung eines Pythonskripts liegt darin begründet, dass die Simulationsdaten von PLEXE-Simulationen in einem von OMNeT++ definierten, textbasierten Datenformat gespeichert werden und OMNeT++ eine Python-Schnittstelle zur Verarbeitung der Daten anbietet. Diese ermöglicht unter anderem Abrufoperationen für die gewünschten Vektoren und implementiert Vektoroperationen, wie die in f_{const} verwendete Ausführung von Funktionen auf jedem Vektorwert – hier Subtraktion mit g' und Exponentiation mit 2 – oder die Mittelwertbildung über einem Vektor.

Da die Berechnung von f_{const} viele Ein- und Ausgabeoperationen benötigt, wird auch *constant_headway* parallelisiert, um die Berechnungszeit bei Anfragen mit vielen Simulationsergebnissen zu verringern. In diesem Fall ist dies jedoch nicht mithilfe von *Multithreaded*, sondern im Pythonskript selbst realisiert. Für eine Parallelisierung in C++ müsste das Skript mehrmals eingebettet werden. Dies ist jedoch nicht möglich, da Python je System nur eine Instanz des Python-Interpreters erlaubt. Deshalb nutzt *constant_headway* das in Python integrierte *concurrent*-Modul zur Parallelisierung mithilfe des Thread-Pool-Entwurfsmusters. Auch hier lässt sich die Anzahl maximal zu nutzender Threads in der Konfiguration von *ConstantHeadway* festlegen.

[6] `https://docs.python.org/c-api/`

Kapitel 4

Evaluation

In diesem Kapitel soll die vorgestellte Implementierung von *Simopticon* validiert und evaluiert werden. Zuerst soll dabei in Abschnitt 4.1 auf die neuartige Implementierung des DIRECT-Algorithmus eingegangen werden. Dazu wird der *DirectOptimizer* anhand gängiger Benchmark-Funktionen getestet. Zudem wird *Simopticon* in Abschnitt 4.2 mithilfe der Implementierungen von *DirectOptimizer*, *PlexeSimulationRunner* und *ConstantHeadway* genutzt, um optimale Werte für Parameter eines Platoon-Reglers von Swaroop u. a. [32] zu finden.

4.1 Optimierer

Im Folgenden soll die Leistungsfähigkeit der vorgestellten Implementierung des *DirectOptimizer*s mit der des ursprünglichen DIRECT-Algorithmus verglichen werden. Zudem wird untersucht, welche Auswirkung die Nutzung verschiedener Ebenen durch die *Levels*-Klasse auf die Effizienz der Optimierung hat.

Funktion	Abk.	n	lokale Minima	globale Minima	Parameterraum
Branin	BR	2	3	3	$[-5,10]\times[0,15]$
Goldstein-Price	GP	2	4	1	$[-2,2]^2$
Six-Hump Camelback	C6	2	6	2	$[-3,3]\times[-2,2]$
Shubert	SHU	2	760	18	$[-10,10]^2$
Hartman 3	H3	3	4	1	$[0,1]^3$
Shekel 5	S5	4	5	1	$[0,10]^4$
Shekel 7	S7	4	7	1	$[0,10]^4$
Shekel 10	S10	4	10	1	$[0,10]^4$
Hartman 6	H6	6	4	1	$[0,1]^6$

Tabelle 4.1 – Verwendete Benchmark-Funktionen

Für die Validation und Evaluation des *DirectOptimizer*s, wird der *StubController* verwendet, um Benchmark-Funktionen zu optimieren. Diese sind in Tabelle 4.1 aufgelistet. Sie wurden jeweils von Jones, Perttunen und Stuckman [7], Liu, Zeng und Yang [18] und Sergeyev und Kvasov [20] genutzt, um DIRECT, MRDIRECT und ADC zu testen. Die Funktionen unterscheiden sich voneinander in der Anzahl an lokalen und globalen Minima sowie im abgetasteten Parameterraum. Allgemein sind Funktionen als komplexer anzusehen, je höher deren Dimensionalität und je höher die Differenz zwischen Anzahl an lokalen und globalen Optima ist. Beispielsweise haben sowohl Shekel 5 als auch Shekel 10 genau ein globales Optimum bei $(4,4,4,4)^T$. Trotzdem ist Shekel 10 komplexer als Shekel 5, da hier neben diesem globalen Optimum 9 lokale Optima die Optimierung erschweren. Zudem sind DIRECT-Derivate dafür bekannt, mit zunehmender Dimensionalität des Optimierungsproblems deutlich schlechter abzuschneiden – Jones [15] spricht sogar von einem „curse of dimensionality“, also einem „Fluch der Dimensionalität“.

Die Bewertung, ab wann ein globales Optimum gefunden wurde, wird dabei wie bei Jones, Perttunen und Stuckman [7] anhand der prozentualen Abweichung δ der aktuell besten Lösung ϕ vom tatsächlichen globalen Minimum f^* definiert. Ist diese kleiner als 0,01 %, gilt das globale Optimum als gefunden. Es muss also Folgendes gelten:

$$\delta = \frac{\phi - f^*}{|f^*|} < 10^{-4} \tag{4.1}$$

4.1.1 Vergleich mit ursprünglichem DIRECT-Algorithmus

Abbildung 4.1 zeigt je Benchmark-Funktion den prozentualen Fehler δ nach einer bestimmten Anzahl an Evaluationen der jeweiligen Funktion. Dabei wurden Shubert-Funktion und Hartman 6 der Übersichtlichkeit halber separat in Abbildung 4.2 dargestellt, da diese deutlich mehr Evaluationen als die restlichen Benchmark-Funktionen benötigen. Außerdem sei angemerkt, dass die Messung des Prozentfehlers nach jeder Iteration des Algorithmus durchgeführt wurde. Wenn also beispielsweise eine Verringerung während einer Iteration auftritt, in der zehn Funktionswerte evaluiert werden, wird diese im Diagramm erst nach diesen zehn Evaluationen sichtbar, selbst wenn das bessere Ergebnis bereits bei der ersten Evaluation gefunden wird.

Jede der in Abbildung 4.1 gezeigten Funktionen erreicht nach weniger als 260 Evaluationen eine Abweichung von weniger als 0,01 %. Dabei sind Häufigkeit und Umfang der Verringerungen von δ je Funktion verschieden. Auffällig sind hier die Shekel-Funktionen, bei denen diese Verbesserungen sehr ähnlich verteilt sind. Dies zeigt, dass die Implementierung des *DirectOptimizer*s in *Simopticon* bei ähnlichen Funktionen ähnliche Ergebnisse liefert. Dies ist auch beim ursprünglichen DIRECT-Algorithmus von Jones, Perttunen und Stuckman [7] der Fall (vergleiche Tabelle 4.2) und ist darauf zurückzuführen, dass die DIRECT-Methode stark kausal arbeitet.

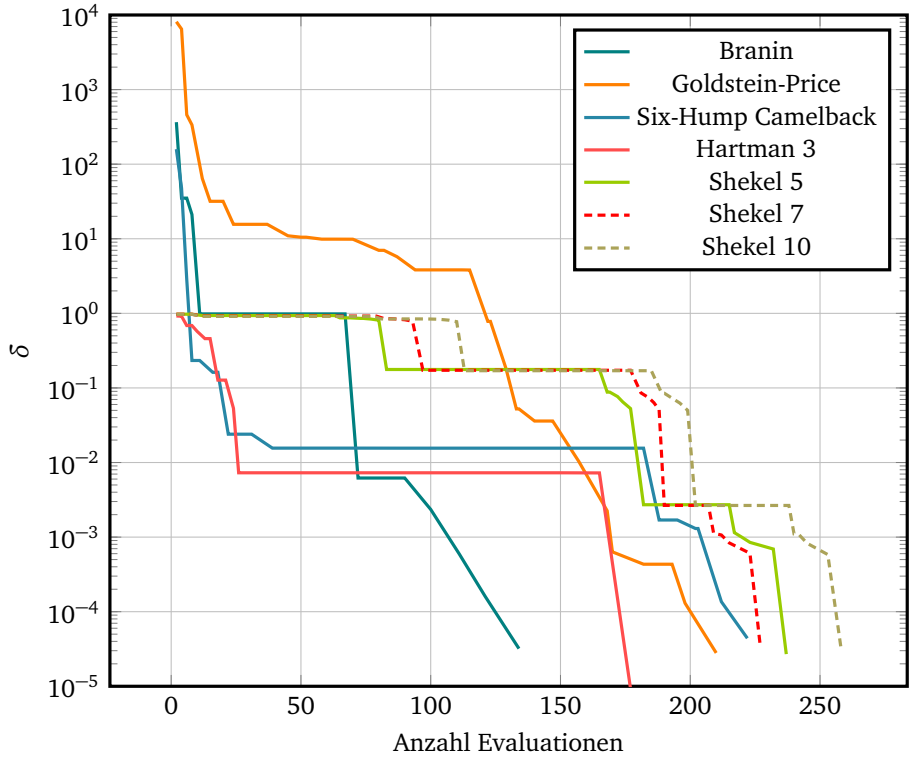


Abbildung 4.1 – Verlauf des Prozentfehlers ausgewählter Funktionen

Des Weiteren fällt auf, dass bei der Six-Hump Camelback-Funktion und Hartman 3 jeweils für über 130 Evaluationen keine signifikante Verbesserung auftritt. In diesen Beispielen hat der Optimierer ein lokales Minimum gefunden und muss zuerst durch globale Abtastung der Funktion das Becken des globalen Minimums finden. Bis dies erfolgt ist, verändert sich das aktuelle Optimum nicht.

Tabelle 4.2 zeigt je Benchmark-Funktion, wie viele Evaluationen DIRECT von Jones, Perttunen und Stuckman [7] und der in *Simopticon* implementierte *Direct-Optimizer* brauchen, um das globale Optimum zu finden. Dort ist erkennbar, dass die Implementierung des *DirectOptimizer*s in jedem Fall das globale Minimum der Funktion findet – wenn auch mit relativ vielen Evaluationen bei Hartman 6.

Methode	Benchmark-Funktion								
	BR	GP	C6	SHU	H3	S5	S7	S10	H6
DIRECT [7]	195	191	285	2967	199	155	145	145	571
Simopticon	134	210	222	1822	181	237	227	258	13 443

Tabelle 4.2 – Anzahl an Evaluationen bis zur Konvergenz

Bei den Benchmarks mit $n \leq 3$ schneidet *Simopticon* in vier von fünf Fällen besser ab als DIRECT. Dies spricht für die Kombination von MRDIRECT und ADC, da ADC von Sergeyev und Kvasov [20] auf diesen Benchmarks ausnahmslos schlechtere Effizienz als DIRECT hat. Die Optimierung der Shubert-Funktion sticht hier besonders heraus – während sonst die Differenz der beiden Ansätze nicht mehr als 81 Evaluationen beträgt, benötigt *Simopticon* hier 1145 Evaluationen weniger, um das Minimum zu finden. Dies ist vor allem bemerkenswert, da es sich um eine Funktion mit einer hohen Zahl an lokalen Minima handelt. Zudem befindet sich in direkter Umgebung jedes globalen Minimums ein globales Maximum, was eine Optimierung der Shubert-Funktion zusätzlich erschwert.

Für höherdimensionale Probleme mit $n > 3$ schneidet die Implementierung dagegen schlechter ab als DIRECT, was eine Beobachtung von Sergeyev und Kvasov [20] bestätigt, die bei ADC ebenfalls von diesem Verhalten berichten. Bei den Shekel-Funktionen ist die Differenz zwar signifikant, liegt aber noch in einer verkraftbaren Größenordnung, wenn die Evaluationen die Kosten von PLEXE-Simulationen haben.

Bei Hartman 6 benötigt die Implementierung weit mehr Evaluationen als DIRECT. Abbildung 4.2 zeigt, dass der Prozentfehler δ nach 989 Evaluationen zwar nur noch 3,8 % beträgt, sich danach jedoch über 12 291 Evaluationen hinweg kaum verringert. Bezeichne P^* den Hyperquader, der ein Blatt des aktuellen Partitionierungsbaumes ist und die optimale Parameterkombination enthält. Abbildung 4.3 zeigt die Tiefe, in der sich P^* befindet sowie den Verlauf der Gesamttiefe des Partitionsbaumes. Dort ist ersichtlich, dass P^* äußerst selten geteilt wird, also selten auf eine tiefere Ebene

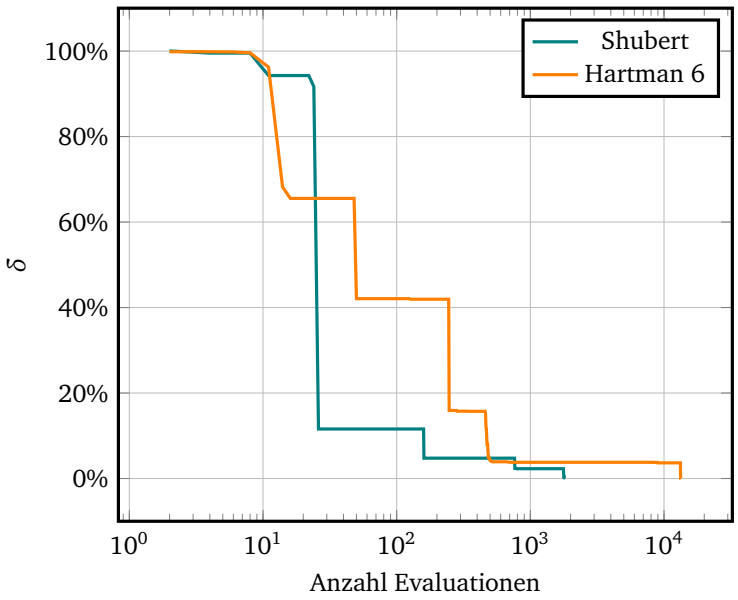


Abbildung 4.2 – Verlauf des Prozentfehlers bei Shubert und Hartman 6

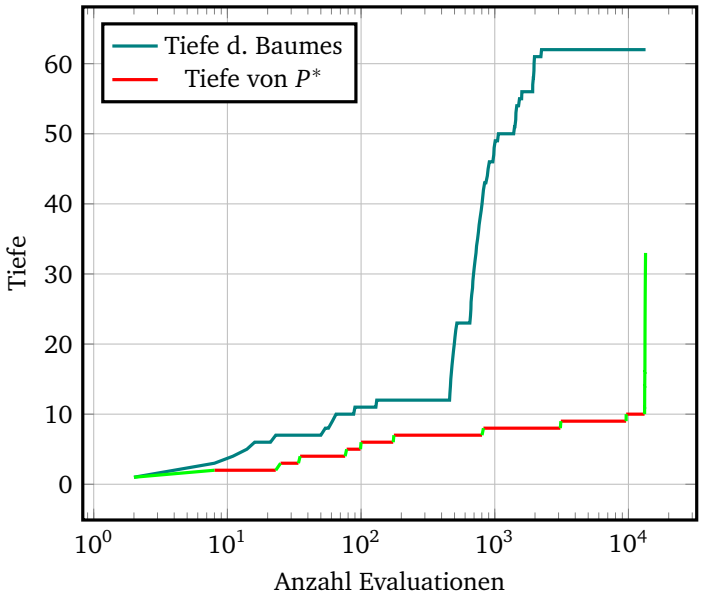


Abbildung 4.3 – Tiefe des optimalen Hyperquaders bei Hartman 6

des Baumes übergeht, andere Hyperquader allerdings schon (da die Gesamttiefe regelmäßig steigt). Der Grund hierfür sind schlechte Durchschnittswerte an den abgetasteten Ecken von P^* und die hohe Dimensionalität von Hartman 6. In Abbildung 4.3 ist der Graph der Tiefe von P^* grün eingefärbt, wenn P^* auf der jeweiligen Tiefe den besten Wert hat. Dies ist während der meisten Evaluationen nicht der Fall, was bedeutet, dass P^* durch die in Abschnitt 3.2.2.3 beschriebene Vorfilterung nicht als potenziell optimaler Hyperquader in Betracht gezogen wird, da andere Hyperquader auf derselben Ebene bessere Werte aufweisen. Es handelt sich also um ein Problem, welches durch die Abtastungsstrategie von ADC hervorgerufen wird. Deren Annahme, dass ein Hyperquader anhand zweier Eckpunkte genauer bewertet werden könne als anhand eines Mittelpunktes, trifft in diesem Fall nicht zu. Besonders für höherdimensionale Funktionen liefert ADC auch bei Sergeyev und Kvasov [20] deutlich schlechtere Ergebnisse. Dies liegt unter anderem daran, dass Hyperquader, die selten geteilt wurden, in höheren Dimensionen zu groß sind, um anhand der Eckpunkte auf mögliche Optimalität zu schließen. Beispielsweise liegt P^* in Abbildung 4.3 die meiste Zeit auf den Ebenen 7, 8 und 9, auf denen die meisten Dimensionen erst einmal geteilt wurden.

4.1.2 Auswirkung der verschiedenen Ebenen

Tabelle 4.3 zeigt vergleichend die Anzahl der zur Konvergenz benötigten Evaluationen je nachdem, welche Ebenen der *Levels*-Klasse verwendet werden. Dabei wurde variiert, ob Ebene 3 zur globalen Optimierung nach Sergeyev und Kvasov [20]

Ebenen		Benchmark-Funktion									geom.
3	1 & 0	BR	GP	C6	SHU	H3	S5	S7	S10	H6	Mittel
✓	✓	134	210	222	1822	181	237	227	258	13 443	417
	×	141	210	222	1543	216	12 748	7328	33 840	19 528	1725
×	✓	174	230	197	1175	32	8128	4681	7160	10 527	982
	×	143	230	197	1161	32	8126	4671	61 900	11 642	1233

Tabelle 4.3 – Auswirkung der Ebenennutzung auf die Anzahl an Evaluationen

beziehungsweise, ob Ebenen 1 und 0 zur lokalen Optimierung nach Liu u. a. [19] verwendet werden. Ebene 2 kann hier als „Standardebene“ betrachtet werden, da sie der Konfiguration des ursprünglichen DIRECT-Algorithmus entspricht und wird in jedem Versuch verwendet. Die Verwendung von allen Ebenen (siehe erste Zeile) entspricht der Konfiguration, die in Abschnitt 4.1.1 gezeigt wurde. Die Variation der genutzten Ebenen hatte dabei außer bei der Shubert-Funktion keinen Einfluss auf das gefundene Optimum. Bei der Shubert-Funktion wurde bei alleiniger Nutzung der Ebenen 3 und 2 das globale Optimum bei $\left(-\frac{190}{27}, \frac{130}{27}\right)$ gefunden, in den anderen Fällen bei $\left(\frac{190}{27}, -\frac{130}{27}\right)$.

Das gerundete geometrische Mittel über den benötigten Evaluationen je Benchmark ist bei der Verwendung aller Ebenen am geringsten, was darauf schließen lässt, dass bei Verwendung aller Ebenen allgemein die wenigsten Evaluationen benötigt werden. Auffällig ist jedoch, dass der Verzicht auf Ebene 3 und damit auf die globale Phase der Optimierung bei Six-Hump Camelback-, Shubert- und den Hartman-Funktionen mit weniger benötigten Evaluationen einhergeht. Hier scheint also jeweils die Beschränkung auf die größten Hyperquader keinen Vorteil zu bringen. Bei Branin- und Goldstein-Price-Funktion verschlechtert sich die Effizienz durch das Weglassen von Ebene 3 zwar, allerdings nur marginal. Grund für die im Mittel trotzdem bessere Leistung bei Hinzunahme von Ebene 3 sind die Shekel-Funktionen. Auf diesen konvergiert die Implementierung deutlich langsamer als wenn alle Ebenen verwendet werden.

Die Nutzung der lokalen Optimierung mit Ebenen 1 und 0 hat größtenteils positive Auswirkungen, was ebenfalls am jeweiligen geometrischen Mittel bei Verwendung beziehungsweise Auslassung dieser ablesbar ist. Die Optimierung von Goldstein-Price- und Six-Hump Camelback-Funktion benötigt unabhängig davon die gleiche Zahl an Evaluationen, was dafür spricht, dass hier der optimale Hyperquader P^* jeweils einer der kleinsten Hyperquader ist und somit nicht vorgefiltert wird – unabhängig, ob gerade Ebene 2, 1 oder 0 verwendet wird. Bei der Shubert-Funktion sind die Ergebnisse auch beim Weglassen der Ebenen 1 und 0 besser. Erneut sind die Shekel-Funktionen hier ausschlaggebend für die im Mittel höhere Effizienz bei Nutzung der lokalen Optimierung. Besonders bei Shekel 10 wird die Optimierung durch

Ebenen 1 und 0 deutlich verkürzt und auch bei Hartman 6 tritt eine Verbesserung ein.

Allgemein ist ersichtlich, dass die globale Optimierung mit Ebene 3 nur sinnvoll ist, wenn zum Ausgleich der dadurch bewirkten Effizienzeinbußen zusätzlich die Ebenen 1 und 0 verwendet werden. Die Kombination beider Ansätze ermöglicht es durch globale Suche auf Ebene 3 vor allem bei den Shekel-Funktionen das Becken des globalen Optimums zu finden und dieses mithilfe der lokalen Phase abzutasten.

4.2 Framework

In diesem Abschnitt wird *Simopticon* mit dem *DirectOptimizer* als Optimierer genutzt, um durch *PlexeSimulationRunner* automatisiert durchgeführte und durch *ConstantHeadway* bewertete PLEXE-Simulationen zu optimieren. Ziel ist dabei, eine optimale Belegung der Reglerparameter des von Swaroop u. a. [32] vorgeschlagenen Platoon-Reglers für ausgewählte Szenarien zu finden. Im Folgenden wird die Funktionsweise des Reglers nicht weiter beleuchtet, da diese keinen Einfluss auf die Arbeitsweise des Optimierers hat, aus dessen Perspektive die Simulationen eine Blackbox darstellen. Es wurde die nativ in PLEXE 3.1 enthaltene Implementierung eines Platooning-Protokolls genutzt, welches den in SUMO implementierten Regler von Swaroop u. a. [32] verwendet.

4.2.1 Vorbetrachtungen

Bedingung für die Nutzung des DIRECT-Algorithmus ist, dass die Zielfunktion in unmittelbarer Umgebung des Optimums Lipschitzstetigkeit aufweist. Da PLEXE neben dem deterministischen Abstandsregler auch die Netzwerkkommunikation simuliert, welche beispielsweise zufälligen Paketverlusten unterliegt, handelt es sich hier um stochastische Simulationen. Entsprechend können die Ergebnisse zweier adjazenter Parameterkombinationen aufgrund von statistischem Rauschen teilweise weit auseinander liegen, was die Stetigkeitsbedingung verletzt.

Um mit PLEXE-Simulationen trotzdem eine möglichst stetige Zielfunktion zu erreichen, wird jede Parameterkombination mehrmals mit unterschiedlichen Seeds für die Zufallsgeneratoren simuliert. Über den Werten dieser Wiederholungen wird das arithmetische Mittel gebildet (vergleiche Abschnitt 3.4.2.1), was das statistische Rauschen abschwächt. Bei den durchgeführten Optimierungen wurden je Parameterkombination – wenn nicht anders angegeben – 6 Wiederholungen durchgeführt. Die Auswirkung der Anzahl an Wiederholungen je Simulationslauf auf das Ergebnis der Optimierung wird in Abschnitt 4.2.3 aufgegriffen.

Der von Swaroop u. a. [32] vorgeschlagene Platoon-Regler besitzt drei einstellbare Parameter – C_1, ω_n und ξ [33] – auf deren Bedeutung hier nicht weiter ein-

Parameter	Wertebereich	Einheit
C_1	$[0,1]$	–
ω_n	$\left[\frac{1}{20},5\right]$	Hz
ξ	$[1,10]$	–

Tabelle 4.4 – Abgetasteter Parameterraum

gegangen wird. Dabei gilt $0 \leq C_1 \leq 1$, $\omega_n > 0$ und $\xi \geq 1$. Da DIRECT je Parameter einen Wertebereich in Form eines geschlossenen Intervalls benötigt, wurden die in Tabelle 4.4 aufgelisteten Wertebereiche abgetastet.

Neben diesen Parametern erlaubt PLEXE es außerdem, den Sollabstand g' zu konfigurieren. Dieser ist im Folgenden auf $g' = 5\,\text{m}$ festgelegt und nicht Teil des optimierten Parameterraumes. Andere Konfigurationswerte, wie Netzwerkparameter, Anzahl an Platoon-Teilnehmern (im Folgenden 8 Fahrzeuge) und deren Motormodelle wurden aus der Standardkonfiguration übernommen, welche PLEXE in Version 3.1 beiliegt.

Für die Parameter des Reglers sind a priori keine optimalen Werte bekannt. Um das Ergebnis trotzdem mit ausreichender Sicherheit als globales Minimum zu verifizieren, werden die Optimierungen jeweils durchgeführt, bis 1200 Iterationen ohne eine Verbesserung des Optimums vergangen sind oder bis 0 – der bestmögliche Wert, den *ConstantHeadway* zurückgeben kann – erreicht wurde, je nachdem welche Bedingung zuerst eintritt.

Zudem steht zu vermuten, dass eine optimale Belegung der Parameter vom simulierten Szenario abhängt. Deshalb werden die in Abschnitt 3.3.2 beschriebenen Szenarien „Sinusoidal“ und „Braking“ jeweils einzeln optimiert (siehe Abschnitt 4.2.2.1 und Abschnitt 4.2.2.2). Um Parameter zu finden, die für beide Szenarien die besten Ergebnisse liefern, wird außerdem eine gemeinsame Optimierung beider Szenarien durchgeführt (Abschnitt 4.2.2.3).

Szenario	Iterationen	Evaluationen	Hyperquader	C_1	ω_n	ξ	Minimum
Sinusoidal	727	763	1973	$\frac{25}{27}$	$\frac{3}{5}$	$\frac{16}{3}$	$1{,}294\,41 \cdot 10^{-5}$
Braking	5	17	21	1	5	4	0
Sinusoidal & Braking	2588	3176	8407	$\frac{3693}{6561}$	$\frac{673}{810}$	$\frac{604}{243}$	$1{,}983\,23 \cdot 10^{-5}$

Tabelle 4.5 – Ergebnisse der Optimierungen

4.2.2 Optimierung

Tabelle 4.5 zeigt die Ergebnisse der Optimierungen von „Sinusoidal“- und „Braking“-Szenario sowie der gemeinsamen Optimierung beider Szenarien. Dabei ist jeweils angegeben, wie viele Iterationen und Evaluationen der Algorithmus durchlaufen hat sowie die Anzahl an Hyperquadern, in die der Suchraum unterteilt wurde. Zudem sind je Szenario der beste gefundene Wert und die entsprechenden Parameter angegeben. Dabei ist zu beachten, dass in den Fällen, in denen nicht das Minimum 0 erreicht werden konnte, die letzten 1200 Iterationen nicht in die Tabelle einfließen, da während dieser kein besserer Wert gefunden wurde (vergleiche Abschnitt 4.2.1).

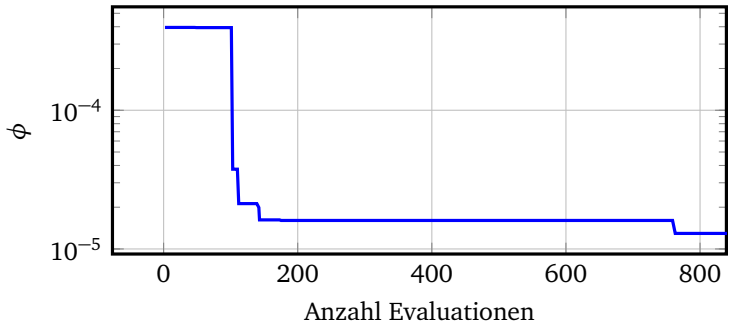


Abbildung 4.4 – Verlauf des besten Ergebnisses beim „Sinusoidal“-Szenario

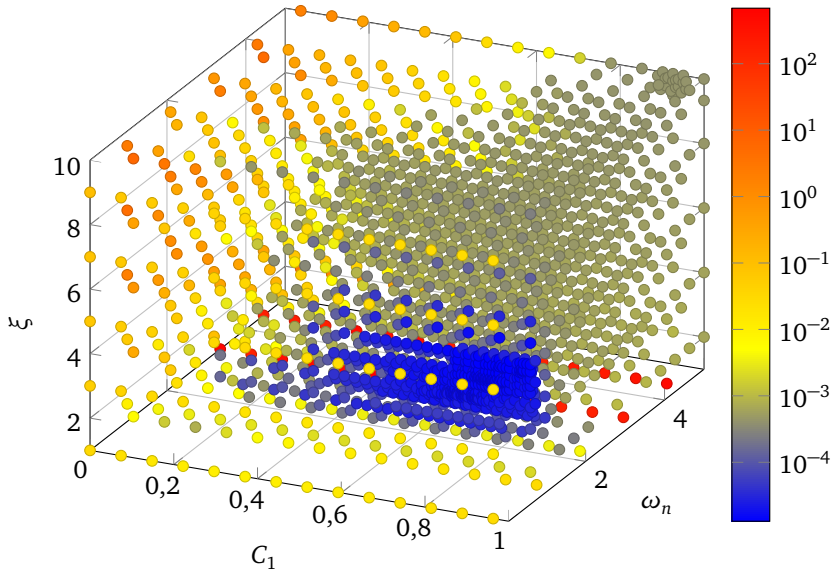
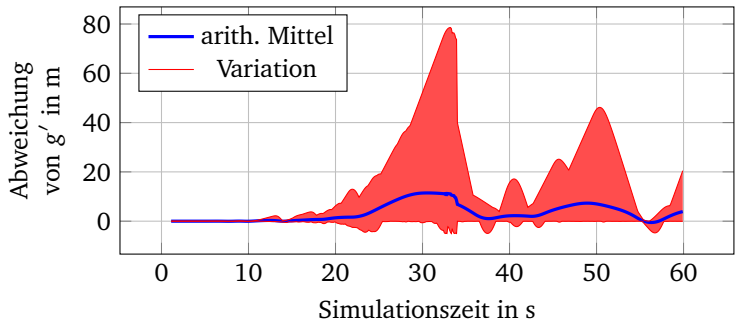


Abbildung 4.5 – Abtastung bei Optimierung des „Sinusoidal“-Szenarios

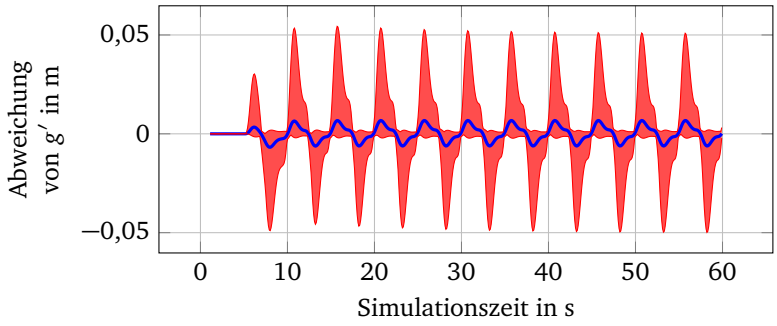
4.2.2.1 Optimierung des „Sinusoidal“-Szenarios

Abbildung 4.4 zeigt den besten gefundenen Wert im Verlauf der Optimierung des „Sinusoidal“-Szenarios, Abbildung 4.5 zeigt die dabei abgetasteten Punkte im durchsuchten Parameterraum. Dabei ist in Abbildung 4.5 das Becken des globalen Optimums klar an einer großen Ansammlung niedriger Werte in der Umgebung des in Tabelle 4.5 genannten Minimums zu erkennen. Zudem fällt auf, dass die Simulationen mit $\omega_n = 3{,}35\,\text{Hz}$ oder $\omega_n = 4{,}45\,\text{Hz}$ bei $\xi = 1$ unabhängig von C_1 schlechte Ergebnisse liefern – diese Parameterkombinationen scheinen also eindeutig suboptimal für das betrachtete Szenario zu sein. Des Weiteren zeigt Abbildung 4.5, wo der *DirectOptimizer* während der lokalen Phasen ein lokales Optimum erforscht hat. Erkennbar ist dies an einer hohen Granularität der Abtastungen in einem Teil des Parameterraumes, wie sie zum Beispiel in der rechten oberen Ecke des Diagramms oder um das globale Optimum herum zu sehen ist. Diese stehen im Kontrast zu suboptimalen Regionen, die aufgrund schlechter Werte nur sporadisch abgetastet wurden, was beispielsweise bei $C_1 \leq 0{,}2$ zu sehen ist.

Abbildung 4.6 zeigt, welche Auswirkungen die unterschiedliche Wahl der Parameter auf die Abweichung der Fahrzeuge vom Sollabstand $g' = 5\,\text{m}$ hat. Die zugrunde liegenden Simulationen wurden sechsmal wiederholt. Dabei sind die mittlere Ab-



(a) $C_1 = \frac{10}{27}, \omega_n = \frac{67}{20}, \xi = 1$



(b) $C_1 = \frac{25}{27}, \omega_n = \frac{3}{5}, \xi = \frac{16}{3}$

Abbildung 4.6 – Abweichung vom Sollabstand g' beim „Sinusoidal“-Szenario

weichung der Fahrzeuge im Platoon (blau) sowie der Bereich von minimaler bis maximaler Abweichung einzelner Fahrzeuge (rot) dargestellt. Abbildung 4.6a zeigt dabei den Verlauf bei Nutzung der – nach *ConstantHeadway* – schlechtesten gefundenen Parameterkombination, Abbildung 4.6b zeigt das Optimum. Es ist ersichtlich, dass die Fahrzeuge beim gefundenen Optimum im Mittel nicht mehr als 0,7 cm vom Sollabstand g' abweichen. Im schlechtesten Fall weichen sie weniger als 6 cm von g' ab. Im Gegensatz dazu weichen die Fahrzeuge bei der suboptimalen Parameterkombination im Mittel bis zu 11,5 m ab, im schlechtesten Fall sogar 78,6 m. Zudem ist in Abbildung 4.6b erkennbar, dass die Abweichung analog zur Beschleunigung des Platoon-Führers alterniert, der mit einer Frequenz von 0,2 Hz beschleunigt und wieder abbremst. Ein solches, berechenbares Muster ist in Abbildung 4.6a nicht zu erkennen, was ebenfalls gegen die Stabilität des Platoon-Regelers bei diesen Parametern spricht. Es ist ersichtlich, dass die Optimierung der durch *ConstantHeadway* definierten Zielfunktion tatsächlich Parameter gefunden hat, die eine vergleichsweise geringe Abweichung der Fahrzeuge vom Sollabstand g' ermöglichen.

4.2.2.2 Optimierung des „Braking“-Szenarios

Bei der Optimierung des „Braking“-Szenarios wurde – wie in Tabelle 4.5 ersichtlich ist – sehr schnell der minimale Wert 0 erreicht. Um eine bessere Darstellung der optimierten Zielfunktion zu erhalten, wurde der *DirectOptimizer* deshalb in Abbildung 4.7 angewendet, bis 1000 Evaluationen durchgeführt wurden. Dort ist zu sehen, dass die optimalen Werte in einem deutlich größeren Teil des Parameterrau-

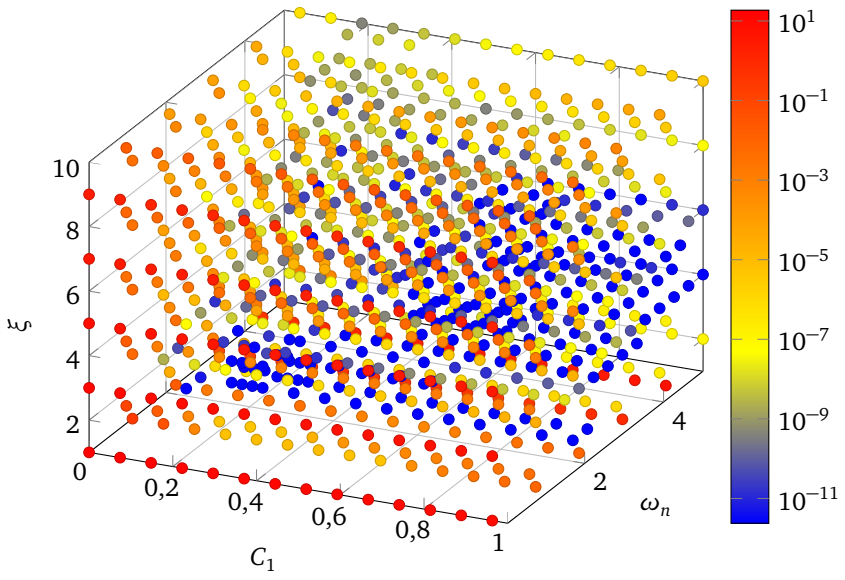


Abbildung 4.7 – Abtastung von *ConstantHeadway* beim „Braking“-Szenario

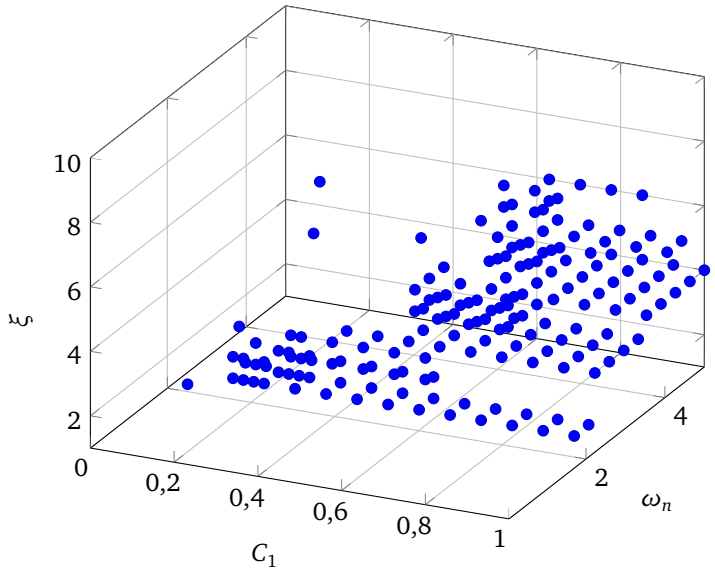
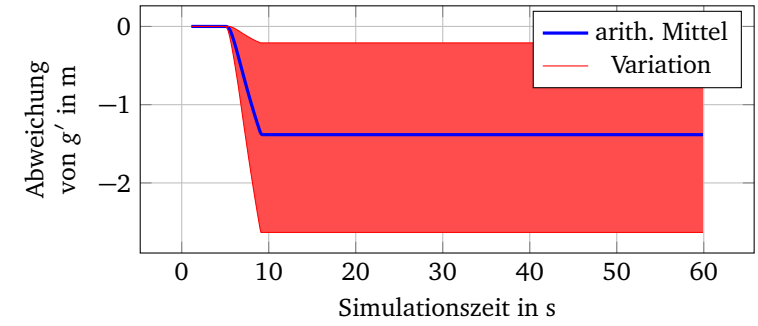


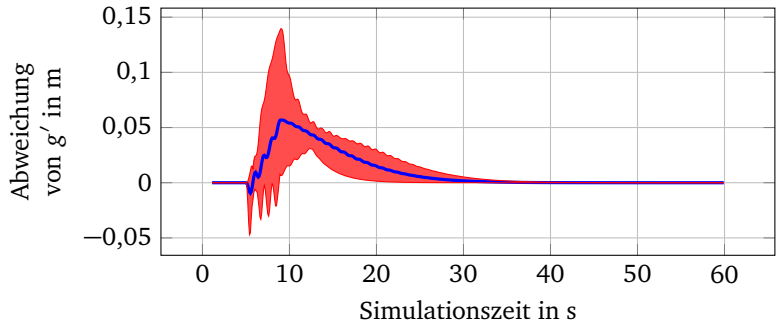
Abbildung 4.8 – Optima von *ConstantHeadway* beim „Braking“-Szenario

mes verteilt liegen. Bei den 1000 Evaluationen, die in Abbildung 4.7 durchgeführt wurden, wurde 175 Mal der minimale Wert 0 erreicht. Diese abgetasteten Werte sind in Abbildung 4.8 nochmals gesondert dargestellt. Im Vergleich mit Abbildung 4.5 fällt hier auf, dass es eine Schnittmenge zwischen dem Becken des globalen Optimums beim „Sinusoidal“-Szenario und den optimalen Werten beim „Braking“-Szenario gibt. Dies weist darauf hin, dass die Minimierung der durch *ConstantHeadway* gegebenen Zielfunktion für die beiden Szenarien keine gegensätzlichen Ziele sind, sondern eine Parameterkombination gefunden werden kann, die für beide Szenarien eine geringe Abweichung vom Sollabstand ermöglicht. Dass sich die Zielfunktionen teilweise ähneln ist zudem an den Werten bei $\xi = 1$ und $\omega_n = 3{,}35\,\mathrm{Hz}$ beziehungsweise $\omega_n = 4{,}45\,\mathrm{Hz}$ zu sehen, die wie beim „Sinusoidal“-Szenario zu den höchsten Werten gehören.

Die vielen optimalen Werte in Abbildung 4.8 lassen vermuten, dass es sich beim „Braking“-Szenario um ein einfacheres Problem als beim „Sinusoidal“-Szenario handelt. Der Unterschied im Verhalten des Platoon-Reglers von Swaroop u. a. [32] bei guten und schlechten Parameterkombinationen wird in Abbildung 4.9 sichtbar. Dort sind der Verlauf der Abweichung von g' im schlechtesten (Abbildung 4.9a) und im besten (Abbildung 4.9b) gefundenen Fall dargestellt (vergleiche Abbildung 4.6). Hauptunterschied der beiden Abläufe ist, dass die Abweichung in Abbildung 4.9b gegen 0 konvergiert, während sie bei Abbildung 4.9a ab dem Zeitpunkt $t = 10\,\mathrm{s}$ im Mittel konstant $-1{,}38\,\mathrm{m}$ beträgt. Grund hierfür ist das durch die Reglerparameter unterschiedliche Bremsverhalten der Fahrzeuge. In Abbildung 4.9b bremsen die meisten Fahrzeuge zunächst stärker als der Platoon-Führer, weshalb es zu einer



(a) $C_1 = 0, \omega_n = \frac{1}{20}, \xi = 1$



(b) $C_1 = 1, \omega_n = 5, \xi = 4$

Abbildung 4.9 – Abweichung vom Sollabstand g' beim „Braking“-Szenario

positiven Abweichung vom Sollabstand g' kommt. Diesen erhöhten Abstand nutzen die Fahrzeuge, um mit sehr niedriger Geschwindigkeit zu der Position zu fahren, an der sie den exakten Sollabstand einhalten. Im Kontrast dazu bremsen die Fahrzeuge in Abbildung 4.9a schwächer ab als der Platoon-Führer und verringern so den Abstand zum jeweiligen Vorausfahrenden. Um diese negative Abweichung auszugleichen, müssten sie rückwärts fahren, was im simulierten Szenario nicht vorgesehen ist. Somit bleiben sie mit suboptimalen Abstand zum Vorausfahrenden stehen.

Hier offenbart sich ein Nachteil der in *ConstantHeadway* implementierten Auswertungsfunktion. Diese betrachtet die Differenz zwischen Ist- und Sollabstand im gesamten Simulationsverlauf, nicht nur für den Teil der Simulation, in der sich die Fahrzeuge bewegen. Beim „Sinusoidal“-Szenario stellt dies kein Problem dar, da die Fahrzeuge während der gesamten simulierten 60 s positive Geschwindigkeit haben. Beim „Braking“-Szenario bewegen sich die Fahrzeuge jedoch einen großen Teil der simulierten Zeit nicht – bei Abbildung 4.9a bleiben sie nach 9,1 s, bei Abbildung 4.9b nach 43,5 s stehen. Dies sorgt dafür, dass *ConstantHeadway* bei diesem Szenario größeren Wert auf die Abstände nach als auf die Abstände während des Bremsvorgangs legt. Dieses Phänomen erklärt auch, warum die Berechnung von *ConstantHeadway* in so vielen Fällen niedrige Werte liefert.

Dass *ConstantHeadway* in vielen Fällen 0 ergibt, weist auf einen Fehler in der Auswertungsfunktion hin, da in Abbildung 4.9b ersichtlich wird, dass der eigentliche Wert größer als 0 sein müsste. Wäre er tatsächlich 0, müsste $g(t) = g'$ für alle $t \in [0, 60]$ gelten, wobei $g(t)$ den Istabstand an Zeitpunkt t bezeichnet. Das verfälschte Ergebnis kommt hierbei durch eine fehlerhafte Implementierung der Mittelwertsbestimmung über Vektoren durch die Python-Schnittstelle von OMNeT++ zustande. Infolgedessen werden viele Parameterkombinationen, bei denen die Fahrzeuge im richtigen Abstand halten, durch den Optimierer als gleichwertig angesehen, obwohl die Abweichung der Abstände während des Bremsvorgangs bei einigen dieser Kombinationen geringer als bei anderen sein kann.

4.2.2.3 Gemeinsame Optimierung mehrerer Szenarien

Abbildung 4.10 zeigt den Verlauf der gemeinsamen Optimierung beider Szenarien, welche deutlich mehr Evaluationen benötigt als bei Betrachtung der einzelnen Szenarien. Dies könnte auf einen Zusammenhang zwischen der Komplexität der optimierten *ConstantHeadway*-Funktion und der Anzahl optimierter Szenarien hinweisen.

Abbildung 4.11 zeigt die abgetasteten Werte der Zielfunktion. Dabei wird Aufgrund der hohen Anzahl an Evaluationen für bessere Übersichtlichkeit nur jeder dritte Wert dargestellt. Da es sich nach der Definition von *ConstantHeadway* bei der abgetasteten Zielfunktion $f_{both} = \frac{f_{sinus}+f_{brake}}{2}$ um das arithmetische Mittel der Funk-

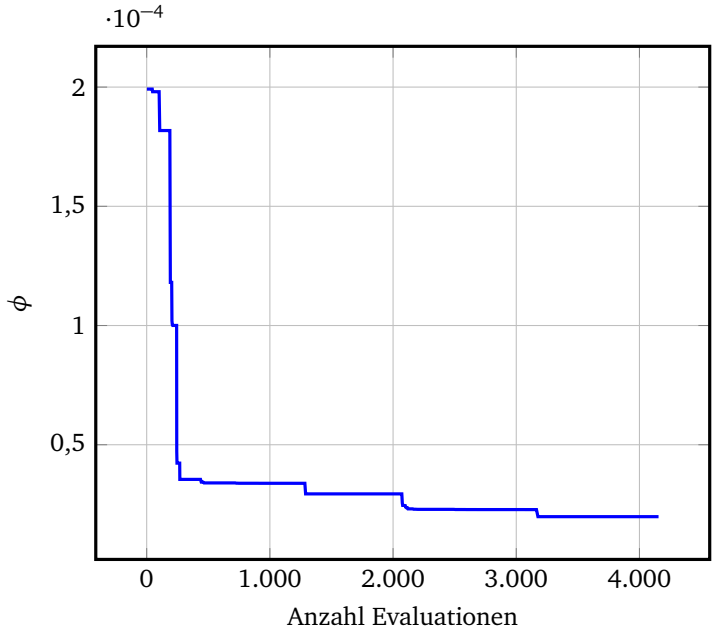


Abbildung 4.10 – Verlauf des besten Ergebnisses beider Szenarien

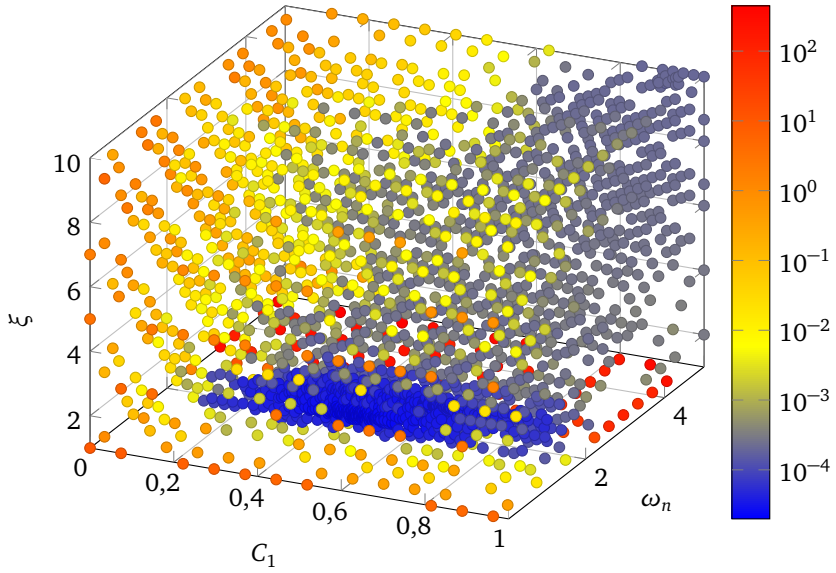


Abbildung 4.11 – Abtastung bei Optimierung beider Szenarien

tionen des „Sinusoidal“- (f_{sinus}) und des „Braking“-Szenarios (f_{brake}) handelt, weist sie deutliche Ähnlichkeiten zu diesen auf. Beispielsweise sind Kombinationen mit $C_1 \leq 0{,}2$ auch bei der gemeinsamen Optimierung beider Szenarien relativ schlecht bewertet, wie es auch bei den Funktionen der einzelnen Szenarien der Fall ist. Zudem befinden sich die meisten Werte, die kleiner als 10^{-3} sind, in einem Bereich, der eine Schnittmenge der in Abbildung 4.8 gezeigten optimalen Werte mit dem Bereich darstellt, der in Abbildung 4.5 die meisten Werte kleiner als 10^{-3} enthält.

Das Becken des globalen Optimums befindet sich an gleicher Stelle wie beim „Sinusoidal“-Szenario, ist allerdings kleiner als dort. Dass sich dieses Becken nicht wie beim „Braking“-Szenario in Richtung höherer ω_n-Werte erstreckt, weist darauf hin, dass die *ConstantHeadway*-Funktion indirekt die Optimierung des „Sinusoidal“-Szenarios präferiert. Dies ist darauf zurückzuführen, dass beim „Braking“-Szenario ein großer Teil des Parameterraumes sehr niedrige Werte hervorbringt. Da in diesem Teil $f_{brake} \approx 0$ gilt, folgt $f_{both} = \frac{f_{sinus}+f_{brake}}{2} \approx \frac{f_{sinus}}{2}$, weshalb dort implizit eher nach dem „Sinusoidal“-Szenario optimiert wird. Dies kommt der in Abschnitt 3.4.1 beschriebenen simplen Pareto-Optimierung gleich. Um einen höheren Fokus auf das Teilziel der Optimierung des „Braking“-Szenarios zu legen, könnte f_{brake} mit einer Konstante $c \in (1, \infty)$ multipliziert werden.

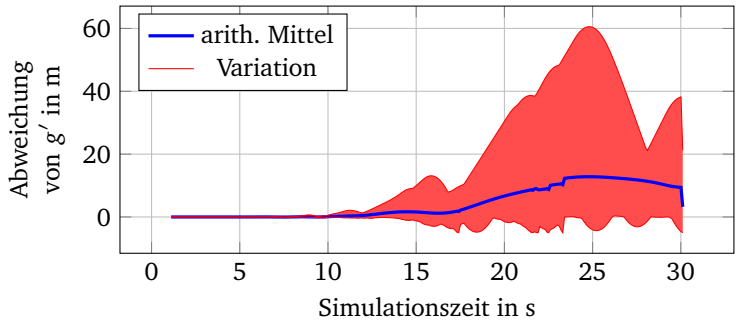
Tabelle 4.6 zeigt, wie die Szenarien im Einzelnen an den durch die gemeinsame Optimierung gefundenen Optima durch *ConstantHeadway* bewertet werden. Es ist ersichtlich, dass der Wert am Minimum beim „Sinusoidal“-Szenario bei gemeinsamer Optimierung beider Szenarien in der gleichen Größenordnung liegt, wie bei der alleinigen Optimierung des „Sinusoidal“-Szenarios. Die jeweiligen Zeitverläufe sind

Szenario	Optimum	Wert
Sinusoidal	Maximum	442,386
	Minimum	$4{,}117\,60 \cdot 10^{-5}$
Braking	Maximum	0,454 69
	Minimum	$1{,}502\,65 \cdot 10^{-5}$

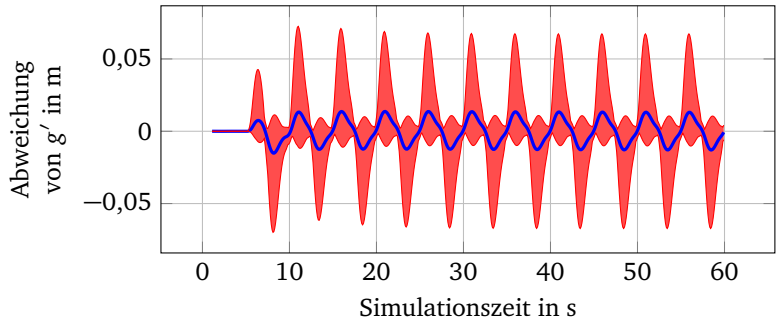
Tabelle 4.6 – Vergleich der Szenarien an den gefundenen Optima

für das „Sinusoidal“-Szenario in Abbildung 4.12, für das „Braking“-Szenario in Abbildung 4.13 dargestellt.

Abbildung 4.12 zeigt dasselbe Verhalten wie Abbildung 4.6. Der einzige Unterschied in Abbildung 4.12b ist, dass die Amplitude der periodischen Abweichung höher ist, die Fahrzeuge also wenige Zentimeter mehr abweichen als in Abbildung 4.6b, was den höheren Wert von *ConstantHeadway* erklärt. In Abbildung 4.12a ist das Verhalten ebenfalls ähnlich zu Abbildung 4.6a, allerdings fällt auf, dass für Zeitpunkte nach $t = 30{,}1\,\mathrm{s}$ keine Werte vorliegen. Dies liegt darin begründet, dass zum Zeitpunkt t in jeder Wiederholung der Simulation Fahrzeuge zusammenstoßen, was den jeweiligen Simulationslauf sofort beendet. Dass für die Zeit nach einem Zusammenstoß



(a) $C_1 = \frac{4}{27}, \omega_n = \frac{49}{12}, \xi = 1$

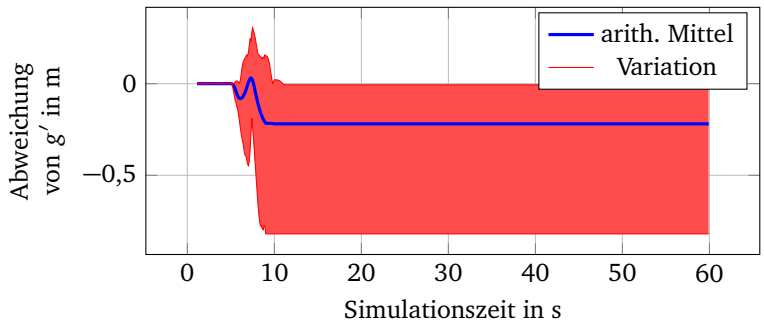


(b) $C_1 = \frac{3693}{6561}, \omega_n = \frac{673}{810}, \xi = \frac{604}{243}$

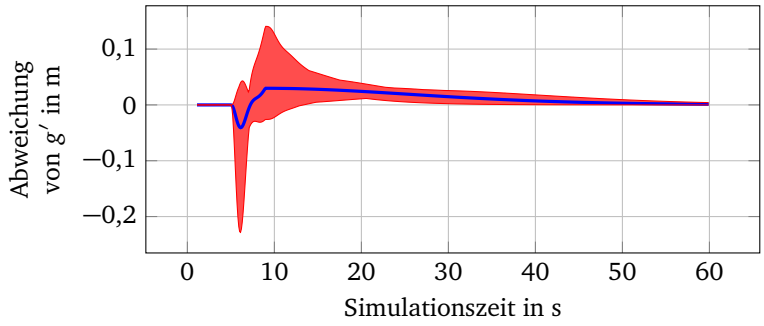
Abbildung 4.12 – Abweichung vom Sollabstand g' beim „Sinusoidal“-Szenario

keine Daten vorhanden sind, ist ein Sachverhalt, mit dem Auswertungsfunktionen wie *ConstantHeadway* umgehen können müssen, wenn sie in Verbindung mit dem *PlexeSimulationRunner* verwendet werden. In diesem Fall sollte ein möglichst hoher Wert vergeben werden, damit Zusammenstöße verhindert werden. *ConstantHeadway* tut dies implizit schon, da für einen Zusammenstoß eine hohe Abweichung vom Sollabstand auftreten muss, die per Definition der Auswertungsfunktion als schlecht bewertet wird.

Abbildung 4.13a zeigt ein ähnliches Problem wie Abbildung 4.9a, da die Fahrzeuge auch dort anfangs zu schwach abbremsen, den Sollabstand unterschreiten und in dieser suboptimalen Position ab $t = 11{,}2\,\mathrm{s}$ stehen bleiben. Die Abweichung ist hier allerdings deutlich geringer als in Abbildung 4.9a und beträgt maximal 82,0 cm, im Mittel nur 22,0 cm. Der optimale Fall in Abbildung 4.13b zeigt ähnlich zu Abbildung 4.9b eine Konvergenz der Fahrzeuge gegen den Sollabstand, nachdem sie im Mittel stärker abbremsen als der Platoon-Führer. Dabei fällt allerdings auf, dass sie dies erst nach einer Verzögerung tun, in der die Istabstände den Sollabstand bis zu 22,8 cm unterschreiten. Die maximale positive Abweichung ist in Abbildung 4.13b geringer als in Abbildung 4.9b, allerdings dauert die Konvergenz der Fahrzeuge



(a) $C_1 = \frac{4}{27}, \omega_n = \frac{49}{12}, \xi = 1$



(b) $C_1 = \frac{3693}{6561}, \omega_n = \frac{673}{810}, \xi = \frac{604}{243}$

Abbildung 4.13 – Abweichung vom Sollabstand g' beim „Braking“-Szenario

gegen den Sollabstand deutlich länger und ist zum Ende der Simulation bei $t = 60\,\mathrm{s}$ noch nicht abgeschlossen.

4.2.3 Auswirkung der Anzahl an Wiederholungen

Um die Auswirkung der Anzahl an Wiederholungen r auf den Optimierungsvorgang zu untersuchen, wurde die Optimierung des „Sinusoidal“-Szenarios für alle $r \in \{1, \ldots, 6\}$ durchgeführt. Dabei wurde jeweils optimiert, bis 1000 Evaluationen durchgeführt wurden, um vergleichbare Ergebnisse zu erhalten. Die jeweils gefundenen Minima sind in Tabelle 4.7 aufgelistet. Um die Werte an den gefundenen Minima möglichst unabhängig von statistischem Rauschen vergleichen zu können, wurden die jeweiligen Parameterkombinationen nochmals mit $r = 16$ unterschiedlichen Seeds simuliert und durch *ConstantHeadway* ausgewertet (letzte Spalte).

In Tabelle 4.7 ist ersichtlich, dass die gefundenen Minima unabhängig von der Wiederholungsrate r sehr nah beieinander liegen. Bei $r \in \{1, 2, 3\}$ und $r \in \{4, 6\}$ wurde sogar jeweils dasselbe Minimum gefunden. Hier sei darauf hingewiesen, dass die Anzahl an Evaluationen zwar jeweils gleich ist, die Rechenzeit jedoch mit höheren Wiederholungsraten r steigt. Dies liegt daran, dass für jede Evaluation $s \cdot r$ Simulationsläufe durchgeführt werden müssen, wobei die Anzahl der Szenarien s in diesem Fall gleich 1 ist. Demnach müssen 6000 Simulationsläufe bei $r = 6$ und nur 4000 Simulationsläufe bei $r = 4$ durchgeführt werden, um dasselbe Ergebnis zu erzielen. Da Simulationsläufe durch den *PlexeSimulationRunner* parallelisiert werden, steigt die Rechenzeit dabei allerdings nicht linear mit der Anzahl der Wiederholungen r. Sei t die Anzahl an Threads, die echt parallel ausgeführt werden können, dann lässt sich die Anzahl der parallel evaluierten Parameterkombinationen e wie folgt berechnen:

$$e = \frac{t}{s \cdot r} \tag{4.2}$$

Aus dem Verhältnis von Iterationen zu Evaluationen in Tabelle 4.5 ist ersichtlich, dass je Iterationsschritt durchschnittlich nur $e = 763 : 727 \approx 1{,}05$ Evaluationen vorgenommen werden. Bei der Optimierung des „Sinusoidal“-Szenarios werden zwar

r	C_1	ω_n	ξ	Wert bei $r = 16$
6	$\frac{25}{27}$	$\frac{3}{5}$	$\frac{16}{3}$	$3{,}212\,27 \cdot 10^{-5}$
5	$\frac{221}{243}$	$\frac{3}{5}$	$\frac{14}{3}$	$3{,}215\,62 \cdot 10^{-5}$
4	$\frac{25}{27}$	$\frac{3}{5}$	$\frac{16}{3}$	$3{,}212\,27 \cdot 10^{-5}$
1-3	$\frac{77}{81}$	$\frac{3}{5}$	$\frac{16}{3}$	$3{,}195\,44 \cdot 10^{-5}$

Tabelle 4.7 – Minima bei Unterschiedlicher Anzahl Wiederholungen r

bis zu 11 Evaluationen in einem Iterationsschritt durchgeführt, meist sind es aber weniger als 3. Demnach können ohne große Verlängerung der Rechenzeit größere Wiederholungsraten gewählt werden, wenn mehrere Threads echt parallel arbeiten können.

Allgemein scheint die Anzahl an Wiederholungen r beim untersuchten Szenario kaum Einfluss darauf zu haben, ob ein Optimum gefunden wird. In Abbildung 4.14 werden die Verläufe des aktuellen Minimums ϕ für die Optimierungen mit $r \in \{1, \ldots, 6\}$ dargestellt. Auch dort ist ersichtlich, dass die Wahl von r kaum Einfluss auf den Verlauf der Optimierung hat, da sich die Verläufe sehr ähneln – bei $r = 1$ und $r = 2$ läuft die Optimierung sogar exakt gleich ab. Mit steigender Anzahl an Evaluationen divergieren die Verläufe allerdings zunehmend und die Optimierungen mit höheren Wiederholungsraten r finden tendenziell bessere Werte. Dies könnte darin begründet sein, dass die Stetigkeit der Funktion – welche durch hohe r erreicht werden soll – erst bei feingranularer Abtastung relevant wird. Möglicherweise handelt es sich jedoch auch um einen Effekt, des in Abschnitt 4.2.2.2 beschriebenen Fehlers der Auswertungsfunktion.

Grund für den trotzdem geringen Einfluss von r ist vermutlich die in diesem Szenario überwiegend fehlerfreie Kommunikation der Fahrzeuge im Platoon. Diese hängt mit den gewählten Netzwerkparametern sowie dem relativ geringen Abstand von 5 m zwischen den Fahrzeugen zusammen. Je fehlerfreier die Kommunikation abläuft, desto weniger Einfluss hat Zufall auf die Simulation – bei störungsfreier Kommunikation wäre sie sogar deterministisch. Durch die größtenteils ungestörte

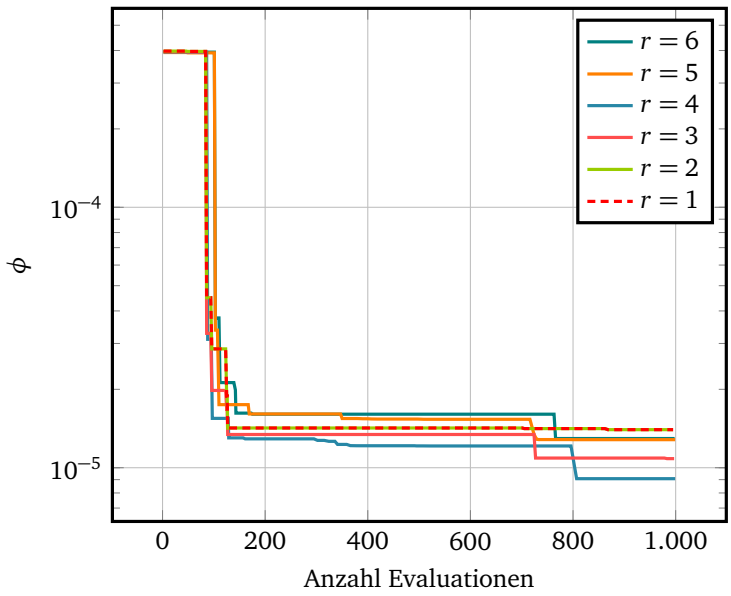


Abbildung 4.14 – Verlauf der Minima bei verschiedenen Wiederholungszahlen

Kommunikation ist bei den gegebenen Netzwerkparametern keine hohe Schwankung der Ergebnisse vorhanden. Für Szenarien mit hohem Paketverlust kann eine Erhöhung der Wiederholungsrate r jedoch nötig sein.

Kapitel 5

Fazit

Ziel dieser Arbeit war die Entwicklung eines modularen Frameworks zur Simulationsoptimierung und dessen Anwendung auf die Optimierung der Parameter von Platoon-Reglern. Zu diesem Zweck wurde eine neue Abwandlung des DIRECT-Algorithmus implementiert, die die Ansätze MRDIRECT und ADC vereint. Die experimentelle Evaluation des neuen Optimierers anhand ausgewählter Benchmark-Funktionen zeigte im Vergleich zum ursprünglichen DIRECT-Algorithmus und zu ADC in mehreren Fällen eine Verringerung der zur Konvergenz benötigten Evaluationen. Stärken des DIRECT-Algorithmus, wie dessen deterministisches, stark kausales Verhalten, dessen Balance zwischen lokaler und globaler Suche sowie dessen sehr gute Leistung in niedriger Dimensionalität konnten genutzt und erweitert werden. Schwächen, wie zu hohe Konzentration des DIRECT-Algorithmus auf lokale Optima kann die neue Implementierung teilweise ausgleichen, bei einer hohen Anzahl an Parametern liefert sie jedoch entsprechend des „curse of dimensionality“ hohe Konvergenzraten.

Des Weiteren wurden die automatisierte Ausführung von Platooning-Simulationen mithilfe des PLEXE-Frameworks und deren Auswertung implementiert. Dabei wurden Maßnahmen zur Parallelisierung ergriffen, um die Ausführung und Auswertung mehrerer Simulationen zu beschleunigen. Zudem können beliebige zu simulierende Szenarien konfiguriert und ausgewertet werden. Außerdem ist es möglich, die Wiederholungsrate jedes Simulationslaufs zu erhöhen, um die erhobenen Daten unabhängig von statistischem Rauschen zu bewerten.

Die Optimierung eines Platoon-Reglers wurde am Beispiel mehrerer simulierter Szenarien demonstriert. Dabei konnte eine klare Differenz zwischen der Leistung bei Nutzung der gefundenen, optimalen Parameter und der Nutzung schlechterer Belegungen festgestellt werden. Die gemeinsame Optimierung mehrerer simulierter Szenarien konvergiert zudem gegen eine Parameterbelegung, welche einen Kompromiss zwischen den Optima der einzelnen Szenarien darstellt. Untersuchungen zu unterschiedlichen Wiederholungsraten zeigten nur geringe Auswirkungen selbiger.

Da die entsprechenden Experimente allerdings nur für ein einziges Szenario mit relativ geringer Zufälligkeit durchgeführt wurden, sind größere Auswirkungen der Wiederholungsrate bei anderen Szenarien nicht auszuschließen.

Zukünftige Arbeiten könnten den Effekt der Wiederholungsrate bei verschiedenen Szenarien, Netzwerkparametern und Platoon-Reglern näher untersuchen. Zudem sind Erweiterungen des *Simopticon*-Frameworks um zusätzliche Optimierungsstrategien, Schnittstellen zu anderen Simulationsframeworks und Auswertungsfunktionen über den Simulationsdaten möglich. Des Weiteren kann *Simopticon* genutzt werden, um optimale Parameter für verschiedene Platoon-Regler zu finden, die infolgedessen qualitativ verglichen werden können.

Abkürzungsverzeichnis

ACC	Abstandsregeltempomat (Adaptive Cruise Control)
ADC	Adaptive Diagonale Kurven (Adaptive Diagonal Curves)
CACC	Kooperative Abstandsregeltempomaten (Cooperative Adaptive Cruise Control)
CC	Geschwindgkeitskontrolle (Cruise Control)
DIRECT	Dividing Rectangles
JSON	JavaScript Object Notation
MRDIRECT	Robustes Mehrebenen-DIRECT (Multilevel robust DIRECT)
PLEXE	Platooning Extension for VEINS
SA	Shuberts Algorithmus
SUMO	Simulation of Urban Mobility
VEINS	Vehicles in Network Simulation

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] S. E. Shladover, D. Su und X.-Y. Lu, „Impacts of Cooperative Adaptive Cruise Control on Freeway Traffic Flow“, *Transportation Research Record*, Jg. 2324, Nr. 1, S. 63–70, Jan. 2012, Publisher: SAGE Publications Inc. DOI: 10.3141/2324-08.

[2] H. Liu, X.-Y. Lu und S. E. Shladover, „Mobility and Energy Consumption Impacts of Cooperative Adaptive Cruise Control Vehicle Strings on Freeway Corridors“, *Transportation Research Record*, Jg. 2674, Nr. 9, S. 111–123, Sep. 2020, Publisher: SAGE Publications Inc. DOI: 10.1177/0361198120926997.

[3] D. Mitra und A. Mazumdar, „Pollution control by reduction of drag on cars and buses through platooning“, *International Journal of Environment and Pollution*, Jg. 30, Nr. 1, S. 90–96, Jan. 2007, Publisher: Inderscience Publishers. DOI: 10.1504/IJEP.2007.014504.

[4] M. Segata, R. Lo Cigno, T. Hardes, J. Heinovski, M. Schettler, B. Bloessl, C. Sommer und F. Dressler, „Multi-Technology Cooperative Driving: An Analysis Based on PLEXE“, *IEEE Transactions on Mobile Computing (TMC)*, Feb. 2022, Publisher: IEEE. DOI: 10.1109/TMC.2022.3154643.

[5] S. Amaran, N. V. Sahinidis, B. Sharda und S. J. Bury, „Simulation optimization: a review of algorithms and applications“, *Annals of Operations Research*, Jg. 240, Nr. 1, S. 351–380, Mai 2016. DOI: 10.1007/s10479-015-2019-x.

[6] B. O. Shubert, „A Sequential Method Seeking the Global Maximum of a Function“, *SIAM Journal on Numerical Analysis*, Jg. 9, Nr. 3, S. 379–388, Sep. 1972, Publisher: Society for Industrial and Applied Mathematics. DOI: 10.1137/0709036.

[7] D. R. Jones, C. D. Perttunen und B. E. Stuckman, „Lipschitzian optimization without the Lipschitz constant“, *Journal of Optimization Theory and Applications*, Jg. 79, Nr. 1, S. 157–181, Okt. 1993. DOI: 10.1007/BF00941892.

[8] R. Lipschitz, „De explicatione per series trigonometricas instituenda functionum unius variabilis arbitrariarum, et praecipue earum, quae per variabilis spatium finitum valorum maximourm et minimorum numerum habent infinitum,disquisitio.“, Jg. 1864, Nr. 63, S. 296–308, Jan. 1864, Publisher: De Gruyter Section: Journal für die reine und angewandte Mathematik. DOI: 10.1515/crll.1864.63.296.

[9] R. M. Dudley, Real Analysis and Probability (The Wadsworth & Brooks Cole mathematics series). Pacific Grove: Wadsworth & Brooks Cole Publ. Co., 1989.

[10] K. Eriksson, D. Estep und C. Johnson, „Lipschitz Continuity“, in *Applied Mathematics: Body and Soul: Volume 1: Derivatives and Geometry in IR3*, K. Eriksson, D. Estep und C. Johnson, Hrsg., Berlin, Heidelberg: Springer, 2004, S. 149–164. DOI: 10.1007/978-3-662-05796-4_12.

[11] E. A. Galperin, „The cubic algorithm“, *Journal of Mathematical Analysis and Applications*, Jg. 112, Nr. 2, S. 635–640, Dez. 1985. DOI: 10.1016/0022-247X(85)90268-9.

[12] J. Pintér, „Globally convergent methods for n-dimensional multiextremal optimization“, *Optimization*, Jg. 17, Nr. 2, S. 187–202, Jan. 1986, Publisher: Taylor & Francis. DOI: 10.1080/02331938608843118.

[13] R. L. Graham, „An efficient algorith for determining the convex hull of a finite planar set“, *Information Processing Letters*, Jg. 1, Nr. 4, S. 132–133, Juni 1972. DOI: 10.1016/0020-0190(72)90045-2.

[14] D. R. Jones und J. R. R. A. Martins, „The DIRECT algorithm: 25 years Later“, *Journal of Global Optimization*, Jg. 79, Nr. 3, S. 521–566, März 2021. DOI: 10.1007/s10898-020-00952-6.

[15] D. R. Jones, „Direct global optimization algorithm“, in *Encyclopedia of Optimization*, C. A. Floudas und P. M. Pardalos, Hrsg., Boston, MA: Springer US, 2001, S. 431–440. DOI: 10.1007/0-306-48332-7_93.

[16] G. Liuzzi, S. Lucidi und V. Piccialli, „A DIRECT-based approach exploiting local minimizations for the solution of large-scale global optimization problems“, *Computational Optimization and Applications*, Jg. 45, Nr. 2, S. 353–375, März 2010. DOI: 10.1007/s10589-008-9217-2.

[17] R. Paulavičius, Y. D. Sergeyev, D. E. Kvasov und J. Žilinskas, „Globally-biased BIRECT algorithm with local accelerators for expensive global optimization“, *Expert Systems with Applications*, Jg. 144, S. 113 052, Apr. 2020. DOI: 10.1016/j.eswa.2019.113052.

[18] Q. Liu, J. Zeng und G. Yang, „MrDIRECT: a multilevel robust DIRECT algorithm for global optimization problems“, *Journal of Global Optimization*, Jg. 62, Nr. 2, S. 205–227, Juni 2015. DOI: 10.1007/s10898-014-0241-8.

[19] Q. Liu, G. Yang, Z. Zhang und J. Zeng, „Improving the convergence rate of the DIRECT global optimization algorithm“, *Journal of Global Optimization*, Jg. 67, Nr. 4, S. 851–872, Apr. 2017. DOI: 10.1007/s10898-016-0447-z.

[20] Y. D. Sergeyev und D. E. Kvasov, „Global Search Based on Efficient Diagonal Partitions and a Set of Lipschitz Constants“, *SIAM Journal on Optimization*, Jg. 16, Nr. 3, S. 910–937, Jan. 2006, Publisher: Society for Industrial and Applied Mathematics. DOI: 10.1137/040621132.

[21] J. Gablonsky und C. Kelley, „A Locally-Biased form of the DIRECT Algorithm“, *Journal of Global Optimization*, Jg. 21, Nr. 1, S. 27–37, Sep. 2001. DOI: 10.1023/A:1017930332101.

[22] D. E. Finkel und C. T. Kelley, „Additive Scaling and the DIRECT Algorithm“, *Journal of Global Optimization*, Jg. 36, Nr. 4, S. 597–608, Dez. 2006. DOI: 10.1007/s10898-006-9029-9.

[23] L. Stripinis, R. Paulavičius und J. Žilinskas, „Improved scheme for selection of potentially optimal hyper-rectangles in DIRECT“, *Optimization Letters*, Jg. 12, Nr. 7, S. 1699–1712, Okt. 2018. DOI: 10.1007/s11590-017-1228-4.

[24] Q. Liu, „Linear scaling and the DIRECT algorithm“, *Journal of Global Optimization*, Jg. 56, Nr. 3, S. 1233–1245, Juli 2013. DOI: 10.1007/s10898-012-9952-x.

[25] P. Wesseling, „Introduction to multigrid methods“, Techn. Ber. NASA-CR-195045, Feb. 1995.

[26] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides und D. Riehle, Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software. München: Addison-Wesley, 2004.

[27] R. P. Garg und I. Sharapov, Techniques for Optimizing Applications: High Performance Computing. Prentice Hall Professional Technical Reference, Jan. 2002.

[28] C. Sommer, R. German und F. Dressler, „Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis“, *IEEE Transactions on Mobile Computing (TMC)*, Jg. 10, Nr. 1, S. 3–15, Jan. 2011, Publisher: IEEE. DOI: 10.1109/TMC.2010.133.

[29] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner und E. Wiessner, „Microscopic Traffic Simulation using SUMO“, in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, ISSN: 2153-0017, IEEE, Nov. 2018, S. 2575–2582. DOI: 10.1109/ITSC.2018.8569938.

[30] A. Varga und R. Hornig, „An Overview of the OMNeT++Simulation Environment“, in *1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, ICST, Mai 2010. DOI: 10.4108/ICST.SIMUTOOLS2008.3027.

[31] K. Miettinen, „Introduction to Multiobjective Optimization: Noninteractive Approaches“, in *Multiobjective Optimization: Interactive and Evolutionary Approaches*, Ser. Lecture Notes in Computer Science, J. Branke, K. Deb, K. Miettinen und R. Słowiński, Hrsg., Berlin, Heidelberg: Springer, 2008, S. 1–26. DOI: 10.1007/978-3-540-88908-3_1.

[32] D. Swaroop, J. Hedrick, C. C. Chien und P. Joannu, „A Comparision of Spacing and Headway Control Laws for Automatically Controlled Vehicles“, *Vehicle System Dynamics*, Jg. 23, Nr. 1, S. 597–625, Jan. 1994, Publisher: Taylor & Francis. DOI: 10.1080/00423119408969077.

[33] R. Rajamani, H.-S. Tan, B. K. Law und W.-B. Zhang, „Demonstration of integrated longitudinal and lateral control for the operation of automated vehicles in platoons“, *IEEE Transactions on Control Systems Technology*, Jg. 8, Nr. 4, S. 695–708, Juli 2000, Conference Name: IEEE Transactions on Control Systems Technology. DOI: 10.1109/87.852914.