Implementierung der „feedback control “- Optimierung des Alternate Gradient Decelerators

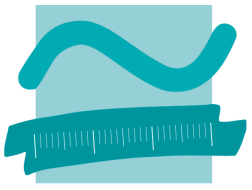
Diplomarbeit

am Fritz-Haber-Institut
der Max-Planck-Gesellschaft
von

Alexander Stanik
Matrikel-Nr.: 721519

Betreuer:
Prof. Dr. B. Buchholz
Dipl. Ing. H. Junkes

Beginn der Bearbeitung:	21. Mai 2007
Tag der Abgabe:	21. August 2007



TECHNISCHE
FACHHOCHSCHULE
BERLIN
University of Applied Sciences

Technische Fachhochschule Berlin
Fachbereich VI
Luxemburger Strasse 10
13353 Berlin

Fritz-Haber-Institut der
Max-Planck-Gesellschaft
Faradayweg 4 - 6
14195 Berlin

Kurzbeschreibung

Diese Diplomarbeit wurde am Fritz-Haber-Institut (FHI) der Max-Planck-Gesellschaft (MPG) in der Abteilung Molekülphysik ausgearbeitet. Der Forschungsschwerpunkt liegt in der Untersuchung der Eigenschaften von Molekülen. Ein Hauptthemenbereich der Abteilung ist das Forschungsgebiet der "Kalten Moleküle". Die Projektgruppe "Deceleration and trapping of large (bio-)molecules", unter der Leitung von Dr. Küpper, entwickelte die Datenerfassungs- und Steuerungssoftware KouDA, mit der es möglich ist, die bei den Experimenten anfallenden Daten aufzunehmen und erste Auswertungen durchzuführen. Im Rahmen dieser Diplomarbeit wurden Optimierungsverfahren basierend auf Evolutionären Strategien in das Softwarepaket KouDA implementiert. Durch den Einsatz dieser Optimierungsverfahren, werden die Experimente effizienter und die Moleküle besser kontrollierbar.

Abstract

The work for this diploma thesis was performed at the Fritz-Haber-Institute (FHI) of the Max-Planck-Society (MPG) in the department of Molecular Physics. The department's main subject of research is "Cold Molecules", with a primary focus on characterizing molecules with a low energy. The research group "deceleration and trapping of large (bio-) molecules", under the direction of Dr. Jochen Küpper, has developed and currently employs a data acquisition and controlling software known as KouDA. This software is used, in conjunction with an experiment, to both collect data and to make a first analysis of that data. The content of this diploma thesis addresses the implementation of optimization procedures in the software package KouDA that are based on evolutionary strategies. Implementing these optimization procedures improves the efficiency of the experiments and allows for more precise control over the experimental parameter, and then in turn, over the molecules.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1. Einleitung

Seit Anfang des 20.Jahrhunderts wurden eine Menge Erkenntnisse über die Abbremsung von Atomen gewonnen. Die Geschwindigkeit eines Teilchens ist entscheidend für dessen Temperatur. Physiker haben Atome auf extrem niedrige Temperaturen, dicht am Nullpunkt im Millikelvin- bis hin zum Nanokelvinbereich, abgekühlt. Durch die immer verbesserten Techniken, erhielt man auch eine immer bessere Kontrolle über die Atome.

Am Fritz-Haber-Institut (FHI) der Max-Planck-Gesellschaft (MPG) wird seit Jahren physikalische Grundlagenforschung betrieben. Die Abteilung Molekülphysik beschäftigt sich mit dem Forschungsgebiet "Kalte Moleküle". Die bereits erlangten Techniken der Atomabbremsung funktionieren bei Molekülen nicht, darum werden am FHI neue Techniken entwickelt, um auch Moleküle einzufrieren. Das Ziel der verschiedenen Projektgruppen ist das Verständnis von Moleküleigenschaften auf mikroskopischem Niveau zu stärken bzw. zu erweitern. Die Projektgruppe "Deceleration and trapping of large (bio-)molecules", unter der Leitung von Dr. Küpper, nutzt zur Abbremsung ein Verfahren mit geschalteten elektrischen Feldern.

Die Schaltzeitpunkte der elektrischen Felder können mathematisch berechnet werden, jedoch existieren beim Aufbau des Experiments, aufgrund der Materialbeschaffenheiten, kleine Ungenauigkeiten. Selbst eine geringe Abweichung von 0,01mm hat eine gravierende Verschlechterung der Effizienz eines Experiments zur Folge. Bei einer Optimierung der Schaltzeitpunkte zur Effizienzsteigerung, stellen die Schaltzeitpunkte einen sehr großen Parameterraum dar, bei dem die Fitnessfunktion sehr verrauscht ist und mehrere Maxima besitzt. Ein anderer wesentlicher Faktor bei der Abbremsung ist die angelegte Hochspannung an den Elektroden. Sie kann nur von Hand eingestellt werden und ist somit ein Parametersatz, der nur in speziellen Fällen geändert werden darf.

Bei der Abbremsung großer Moleküle sind insbesondere die genauen Zeitsequenzen der eingesetzten Hochspannungsschalter besonders wichtig. Diese Zeitsequenzen sollen mittels des Datenerfassungs- und Steuerungsprogramms KouDA (niederländische Abk. für *Koude molekuulen Data Acqisition*) unter Verwendung von Evolutionären Strategien (ES) optimiert werden. Im Rahmen dieser Diplomarbeit sollen fortgeschrittene Evolutionäre Algorithmen (EA) in das Softwarepaket KouDA implementiert und an einem komplexen Experiment getestet werden.

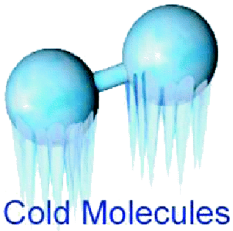
2. Grundlagen

In diesem Kapitel werden die drei Themenschwerpunkte dieser Diplomarbeit erklärt:

- **Abbremsung von Molekülen**
- **Datenerfassungs- und Steuerungssoftware KouDA**
- **Evolutionäre Algorithmen**

Es wird im weiteren Verlauf der Diplomarbeit das errungene Wissen des Studiums der Technischen Informatik vorausgesetzt und darauf aufgebaut.

2.1 Abbremsung von Molekülen



[F.M]

Am Fritz-Haber-Institut für Molekülphysik werden Verfahren entwickelt, um Moleküle mit starken elektrischen Feldern abzubremsen und sie somit abzukühlen.

2.1.1 Molekülverhalten im elektrischen Feld

Betrachtet werden im folgenden Moleküle, die nach außen elektrisch neutral sind. Ein Molekül mit einer polaren Bindung besitzt einen Dipol, weil der negative und der positive Ladungsschwerpunkt nicht in einem Punkt zusammenfallen [HMS02]. Betrachtet man ein polares Molekül in einem elektrischen Feld, so ändert sich seine potentielle Energie aufgrund des Starkeffekts [FHK$^+$03].

Da das homogene elektrische Feld (Abbildung 2.1 [KÖ5, LV1]), jedoch überall die gleiche Stärke hat, ist auch die Starkenergie des Moleküls konstant. Es erfährt keine Kraft und bleibt z.B. still im Raum stehen. Ein Molekül in einem inhomogenen elektrischen Feld bewegt sich in Abhängigkeit der Starkverschiebung seines Quantenzustands. Das in Abbildung 2.2 [KÖ5, LV1] gezeigte Molekül besitzt einen hochfeldsuchenden Zustand (hfs) und bewegt sich in Richtung der größeren Feldstärke. Ein Molekül mit einen tieffeldsuchenden Zustand (tfs) bewegt sich in Richtung der niedrigeren Feldstärke [BTK+06].

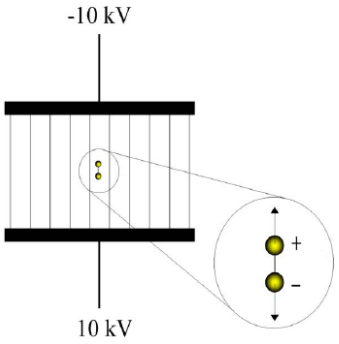


Abbildung 2.1: Molekül in einem homogenen elektrischen Feld

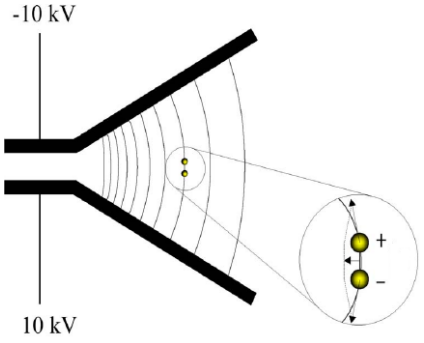


Abbildung 2.2: Molekül in einem inhomogenen elektrischen Feld

2.1.2 Stark Decelerator

Der Stark Decelerator, der nach dem Entdecker des Stark Effekts benannt ist, nutzt die Wechselwirkung polarer Moleküle mit inhomogenen elektrischen Feldern. An eine parallele Anordnung von Elektroden wird eine Hochspannung zwischen ±15 kV zu ±300V geschaltet. Die daraus resultierenden Felder bremsen die Moleküle ab. Das Beispiel in Abbildung 2.3 [KÖ5, LV1] zeigt das Verhalten eines Moleküls in einem tieffeldsuchenden Zustand (tfs). Beim Hineinfliegen in das elektrische Feld wird die kinetische Energie des Moleküls in potentielle Energie umgewandelt, sodass sich die Geschwindigkeit verringert. Der Zuwachs an potentieller Energie ist proportional zur Feldstärke. In diesem Fall wird das Feld ausgeschaltet, sobald das Molekül den Mittelpunkt zwischen den Elektroden erreicht hat. Das Molekül behält die erreichte Geschwindigkeit bei. Ohne einen Abbau des Feldes würde die Kraft entgegengesetzt wirken und das Molekül wieder auf die Ausgangsgeschwindigkeit beschleunigen.

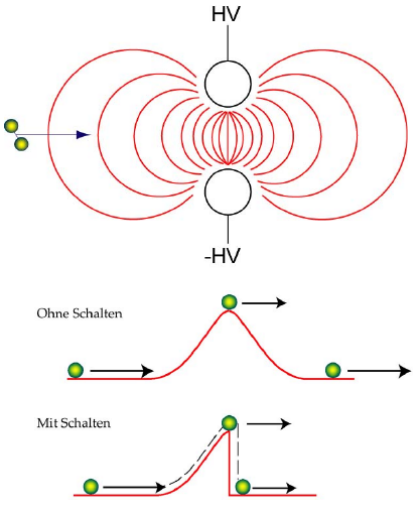


Abbildung 2.3: Molekül in einem elektrischen Feld des Stark Decelerators

Die typische Änderung der kinetischen Energie durch das elektrische Feld eines Elektrodenpaares entspricht 0,1 meV, dies ist ca. 1% der ursprünglichen kinetischen Energie [KÖ5]. Um die Moleküle auf eine sehr geringe Geschwindigkeit abzubremsen, müssen entsprechend viele Elektroden hintereinander geschaltet werden, die den Bremsvorgang wiederholen. Die Abbildung 2.4 [vdM05] zeigt den schematischen Aufbau des Stark Decelerators.

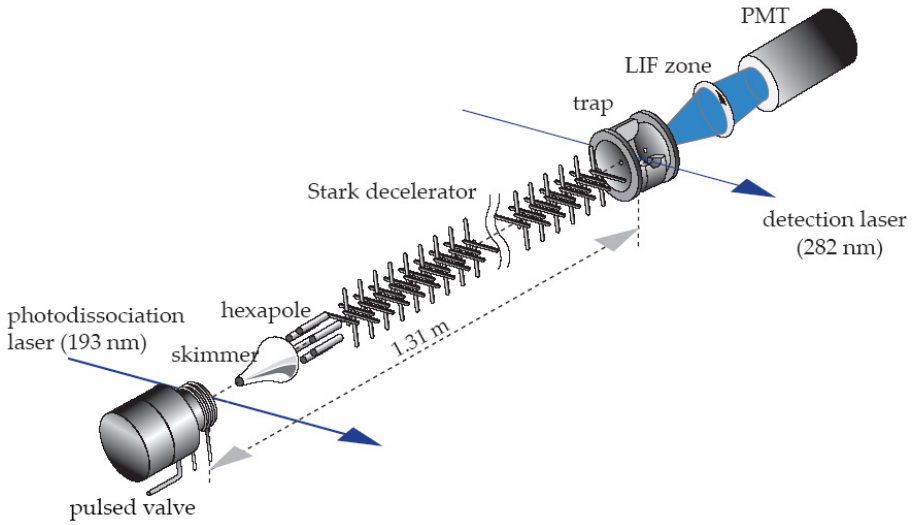


Abbildung 2.4: Schematischer Aufbau des Stark Decelerators

Wenn ein Molekülpaket zum Quasistillstand abgebremst ist, kann es in eine Falle, die sich am Ende der Abbremsstrecke befindet, geladen werden. Die Falle erzeugt ein elektrisches Feld, das alle Moleküle im Mittelpunkt konzentriert. Die Moleküle haben noch eine sehr geringe Restgeschwindigkeit, mit der sie innerhalb der Falle um den Mittelpunkt hin und her schwingen.

2.1.3 Alternate Gradient Decelerator

Der Alternate Gradient Decelerator ist dem Stark Decelerator sehr ähnlich, jedoch sind hier die Elektroden, wie in Abbildung 2.5 [BTK+06] dargestellt, entlang des Molekülstrahls parallel zu der z-Achse ausgerichtet. Ebenfals zu sehen ist die Energieaufspaltung eines Moleküls, aufgrund des Stark Effekts, im elektrischen Feld.

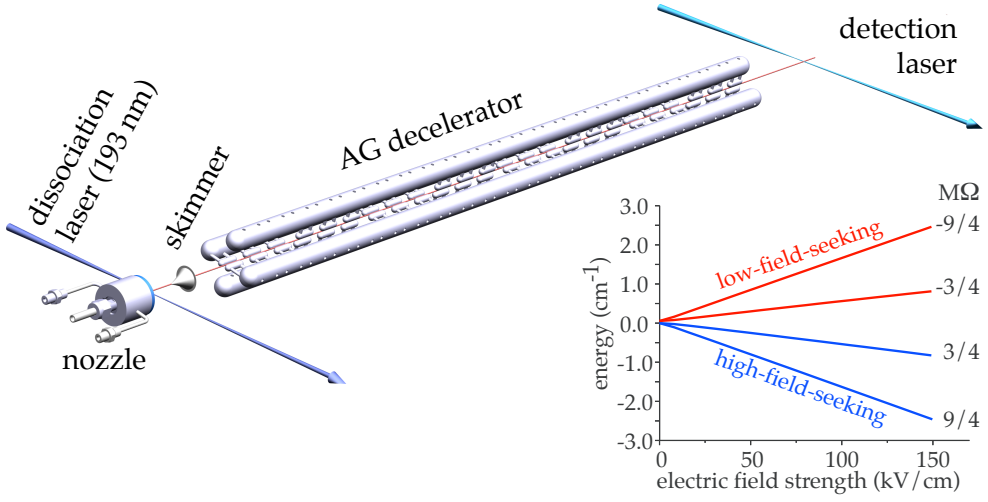


Abbildung 2.5: Schematischer Aufbau des Alternate Gradient Decelerator

Der Flug eines Moleküls durch das Elektrodenpaar, dass in Abbildung 2.6 [WFG+07] dargestellt ist, wird in zwei Schritte untergliedert:

Abbremsvorgang: Der Abbremsvorgang beginnt für Moleküle im tieffeldsuchenden Zustand beim Eintreten in das elekrische Feld bzw. für Moleküle im hochfeldsuchenden Zustand beim Austreten aus dem parallel zueinander stehenden Elektrodenpaar.

Fokussierung: Die Fokussierung findet innerhalb des annähernd homogenen elektrischen Feldes zwischen dem Elektrodenpaar statt. Die Fokussierung zentriert das Molekül auf der z-Achse, wodurch weniger Moleküle in der Abbremsstufe verloren gehen.

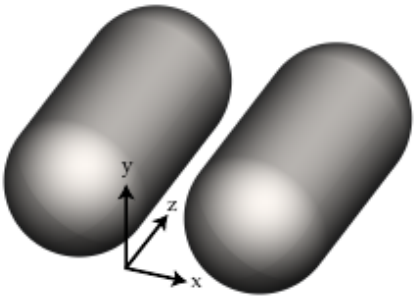


Abbildung 2.6: Elektrodenpaar des Alternate Gradient Decelerator

Es gibt verschiedene Schaltsysteme, mit denen die elektrischen Felder geschaltet werden. Die zwei am häufigsten benutzten Systeme sind:

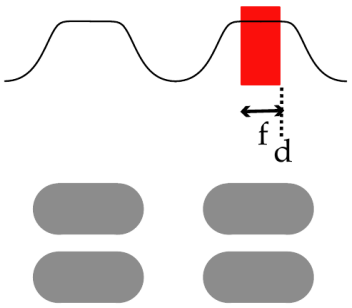


Abbildung 2.7: *single switching*

single switching [WFG+07]**:** Das elektrische Feld zwischen den Elektroden wird nur einmal geschaltet. Die Abbremsung d und Fokussierung f finden innerhalb eines Schaltimpulses statt. Das Schaltsystem wird für Moleküle im hochfeldsuchenden Zustand verwendet.

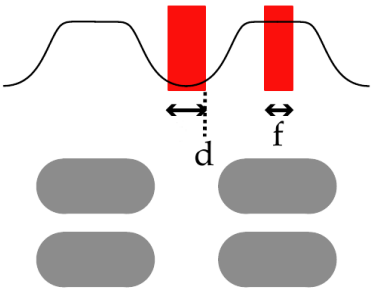
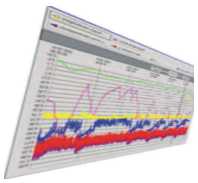


Abbildung 2.8: *double switching*

double switching [WFG+07]**:** Das elektrische Feld zwischen den Elektroden wird während eines Durchflugs zweimal geschaltet. Der Abbremsvorgang d und die Fokussierung f finden nacheinander in zwei seperaten Schaltimpulsen statt. Das Schaltsystem wird für Moleküle im tieffeldsuchenden Zustand verwendet.

2.2 KouDA



KouDA (niederländische Abkürzung für *Koude molekuulen Data Acqisition*) ist eine Datenerfassungs- und Steuerungssoftware, die am Fritz-Haber-Institut entwickelt wird. Der Hauptverwendungszweck von KouDA ist die Ansteuerung der Hardware, die für die Experimente mit kalten Molekülen notwendig ist. Das Programm muss in jeder Hinsicht stabil und fehlerfrei funktionieren, da die Experimente mit sehr hohen Spannungspegeln arbeiten und eine Fehlfunktion gravierende Folgen haben könnte. Bei einem Ausfall wäre der Datenverlust ein enormer und zusätzlicher Aufwand, da dessen Ermittlung sehr viel Zeit in Anspruch genommen hat.

Geräte können über verschiedene Schnittstellen mit dem KouDA-System-Rechner verbunden sein. Die am häufigsten verwendete Schnittstelle ist das lokale Netzwerk. Der KouDA-System-Rechner kommuniziert dabei mit einem VxWorks-Echtzeitsystem. Das Echtzeitsystem hat CompactPCI Slots und eine CPU, auf der das Echtzeitbetriebssystem VxWorks läuft. KouDA spricht diese Geräte über ein selbst entwickeltes Protokoll an. Alle anderen Geräte, die über GPIB, RS232, IEEE 1284(Parallel-Port) oder Ethernet-Direktverbindung mit dem KouDA-System verbunden sind, verwenden das Hersteller spezifische Protokoll. Die Abbildung 2.9 zeigt einen möglichen Verbindungsaufbau des KouDA-Systems mit einigen häufig verwendeten Geräten, die in Abschnitt 2.2.1 näher erleutert werden.

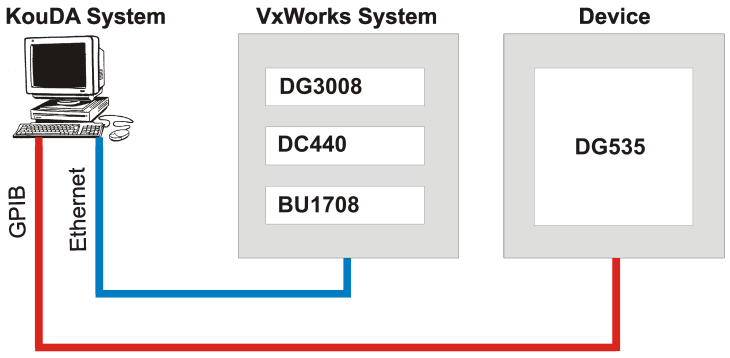


Abbildung 2.9: Mögliche Verbindung des KouDA-Systems mit der Hardware

Die Geräte sind in zwei Kategorien eingeordnet: Steuergeräte, die als *Controller* bezeichnet werden, und die Mess- bzw. Datenerfassungsgeräte, die von den KouDA Entwicklern als *Digitizer* bezeichnet werden.

2.2.1 Hardwareansteuerung

KouDA implementiert alle für die Experimente verwendeten Gerätetreiber als abgeleitete Klassen der Basisklasse *Device*. Diese Klassen stellen alle möglichen Eigenschaften und Konfigurationsoptionen als Funktionsschnittstelle zur Verfügung. Innerhalb der Klasse erfolgt die Verarbeitung der gewählten Option. Das Gerät wird über das entsprechende Protokoll und seine Schnittstelle angesprochen und somit die Konfiguration umgesetzt. Alle erzeugten Objekte werden in einer speziellen Containerklasse, die als Singleton Pattern realisiert ist, gespeichert [Kuh06]. Die *Controller* und *Digitizer* stellen die eigentlichen Treiber dar und jede Instanz von *Device* spiegelt die reelle Hardware wieder.

Es folgt eine Erklärung der wichtigsten *Controller* und *Digitizer*, die für die Experimente und die *Feedback Control* Optimierung relevant sind.

2.2.1.1 Controller

BU1708 Die Burstunit BU1708 ist eine Eigenentwicklung des FHI und ist als CompactPCI Karte ausgelegt. Sie kann extern oder intern per Software getriggert werden und hat vier TTL Ausgangskanäle. Die Kanäle werden nach einem Bitmuster geschaltet. Die Parametrisierung der Burstunit erfolgt durch Angabe eines Zeitpunkts relativ zum Trigger und einem Binärcode. Sie schaltet zum entsprechenden Zeitpunkt die Kanäle, bei Binär 1 ein und bei 0 aus.

In KouDA wird eine solche Schaltsequenz durch eine Burstdatei beschrieben:

```
1 # Kommentar
2 [1]
3 1000000 0x0001
4 2000000 0x0002
5 3050000 0x0000
```

Das Beispiel zeigt eine Schaltsequenz, bei der der Kanal 1 nach 1ms eingeschaltet und nach 2ms wieder ausgeschaltet wird. Der Kanal 2 wird nach 2ms eingeschaltet und nach 3,05ms wieder ausgeschaltet. Eine solche Datei kann in KouDA geladen und zur Burstunit übertragen werden.

DG3008 Der Delay-Generator DG3008 ist eine Eigenentwicklung des FHI und ist als CompactPCI Karte ausgelegt. Er bietet vier separate Delay-Ausgangskanäle. Diese können extern oder intern getriggert werden. Intern arbeitet er mit einer Frequenz von 1 mHz bis zu 10 kHz. Die Kanäle können zu relativen Zeitpunkten ein- und ausgeschaltet werden.

LaserRDLControl Der Ringfarbstofflaserkontroller (RDL für *Ring Dye Laser*) ist ein System zur Stabilisierung der Laserfrequenz auf VxWorks Basis, dass am FHI entwickelt wurde [FK08b].

DG535 Der Delay-Generator DG535 der Firma Stanford Research Systems ist zu dem DG3008 von der Funktionsweise ähnlich. Der Hauptunterschied ist die GPIB Schnittstelle über die er angeschlossen wird.

2.2.1.2 Digitizer

ADC M34: Der ADC M34 ist ein einfacher A/D-Wandler mit 16 Analog-Eingängen.

DC440: Der DC440 ist ein 12-Bit A/D-Wandler der Firma Acqiris, der in KouDA als Oszilloskop implementiert ist. Sie besitzt 2 Kanäle mit eine Abtastrate von 400 MS/s sowie Eingänge für externe Trigger.

DL4100: Das digitale Oszilloskop DL4100 der Firma Yokogawa hat zwei Kanäle mit einer 10-Bit Auflösung. Die maximale Wandlungsrate ist 100 MS/s mit einer effektiven Speicherfrequenz bis zu 150 Hz. Das Gerät wird über den GPIB Bus mit dem KouDA-System verbunden.

2.2.2 Datenauswertung

Neben der Steuerung von Experimenten ermöglicht KouDA auch die Berechnung und Darstellung von komplexen Signalen. Unter Anwendung von Messdaten werden Signale über definierte *Gates* berechnet.

2.2.2.1 Gates

Gates stellen einen Bereich über Messpunkte einer Spur eines *Digitizers* dar, der durch einen Start- und einen Endpunkt definiert wird. Einem Kanal eines *Digitizers* wird ein *Gate* zugeordnet. Der Endpunkt darf nicht kleiner oder gleich dem Startpunkt sein. In der Abbildung 2.10 [Kuh06] ist ein Beispiel für ein *Gate* dargestellt. In einer Reihe von zehn Messpunkten wird ein *Gate* G1 bestimmt. Der Startpunkt von G1 wird als Messpunkt 2 und der Endpunkt von G1 wird als Messpunkt 4 definiert. Das *Gate* G1 besteht aus drei Messpunkten: 2, 3 und 4.

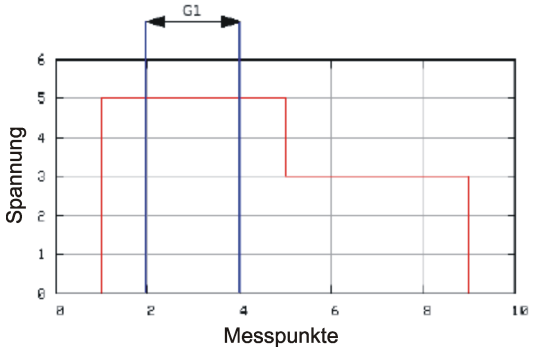


Abbildung 2.10: Beispiel eines Gates aus drei Messpunkten

2.2.2.2 Signale

Ein Signal ist eine Gleitkommazahl, die anhand der *Gates* über mathematische Formeln (max, min, avg, sum, etc.) und arithmetische Operatoren (+, -, ... etc.) berechnet wird. Der Wert eines *Gate* wird durch die Mittelwertbildung aller Messpunkte gemittelt. Um die Berechnung eines Signals zu verdeutlichen, werden zwei *Gates*, wie in Abbildung 2.11 [Kuh06] gezeigt, definiert.

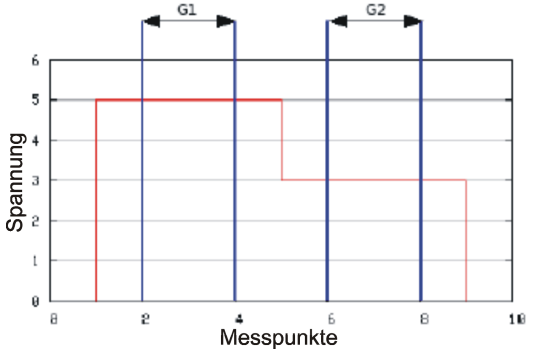


Abbildung 2.11: Beispiel eines Signals aus zwei Gates

In einer Reihe von zehn Messpunkten wurden zwei *Gates* bestimmt. *Gate* G1 beinhaltet die Messpunkte 2, 3, 4 und das Gate G2 die Messpunkte 6, 7, 8. Jedes *Gate* hat somit drei Messpunkte, aus denen sich folgende Berechnung ergibt:

$$\text{Signalexpression: } S1 = G1 + G2$$

$$\text{Wert: } S1 = \frac{5+5+5}{3} + \frac{3+3+3}{3} = 8$$

2.2.3 Praktische Anwendung

Die Burstunit wird verwendet, um die Hochspannungsschalter anzusteuern. Der Delay-Generator dient als Trigger für die Hardware der Experimente. Es wird zum Beispiel ein Laser geschaltet, der die aus der Düse kommenden Moleküle trennt. Ein Kanal wird oft als Bursttrigger verwendet. Dazu wird ein Puls mit der Pulsbreite einer Burstsequenz erzeugt. Die Zeit, die nach der Trennung der Moleküle vergeht, bis der Trigger die Burstsequenz startet, nennt man Burstdelay. Je nach Experiment wird ein Ionen- oder Photonen-Detektor eingesetzt, um Moleküle zu detektieren. Diese Detektoren sind evtl. über einen Vorverstärker mit einem *Digitizer* verbunden und digitalisieren die Messwerte für die KouDA-Software.

2.2.4 Entwicklungsumgebung Qt4

KouDA benutzt zur grafischen Darstellung die Qt4 und Qwt Bibliotheken. Qt4 ist eine plattformunabhängige Bibliothek, die unter anderem für Windows, Linux und Mac OS verfügbar ist. [lP]

Qt4 bietet, ausser den grafischen Objectklassen, noch Standard Template Libary (STL) vergleichbare Templateklassen. Diese Templateklassen basieren auf der STL Bibliothek und stellen zu dem noch weitere Funktionen zur Verfügung. Durch die Benutzung der Qt4 Templates wird die Programmierung wesentlich einfacher und komfortabler. Die Plattformunabhängigkeit der STL ist somit ebenfals gewährleistet [BS07].

Die grafische Gestaltung der Benutzeroberfläche (GUI englische Abkürzung für *Graphical User Interface*) ist durch den bei Qt mitgelieferten GUI Designer leicht zu handhaben. Man kann per Drag and Drop die grafischen Objekte direkt im dargestellten Fenster anordnen, sogar die spezifischen Eigenschaften der Objekte können verändert und angepasst werden. Der Designer erzeugt aus der entworfenen Oberfläche eine XML Datei mit der Dateierweiterung ”.ui”. Ein User-Interface-Compiler namens ”uic” erzeugt aus dem XML Format eine C++ Headerdatei, die anschließend mit einem Meta-Object-Compiler namens ”moc” übersetzt werden muss, bevor sie dem Benutzer als C++ Headerdatei zur Verfügung steht.

Eine Qt4 GUI - Anwendung läuft für gewöhnlich in nur einem Thread. Während einer zeitraubenden Ereignisverarbeitung wird die GUI eingefroren. Das führt oft zu Problemen, da der Benutzer trotz des Zustands weitere *Events* in die Warteschlange hinzufügen kann. Die Lösung solcher Probleme ist, die zeitraubende Verarbeitung in einen weiteren Thread auszulagern. Qt stellt Multithreading Klassen zur Verfügung [BS07]. Somit ist es möglich, mit der Qt Bibliothek, auch Multithreading - Anwendungen zu entwickeln.

2.3 Evolutionäre Algorithmen


[Lun]

Evolutionäre Algorithmen sind Optimierungsalgorithmen, die biologischen Evolutionsprozessen nachempfunden sind. Die Evolutionstheorie zeigt, dass sich Individuen ihrer Umgebung anpassen und ihre Fähigkeiten verbessern, um ihr Überleben zu sichern. Die exakte Nachbildung der Komplexität ist dabei nicht relevant, da nur das Konzept in einem sehr abstrakten Sinn nachgebildet wird.

Durch den Einsatz von evolutionären Algorithmen ist eine schnelle und sich anpassende Lösung möglich. Das Verfahren wird seit vielen Jahren in Industrie, Wirtschaft und Technik eingesetzt, um Abläufe zu optimieren und Lösungen zu finden [GKK04].

2.3.1 Evolution eines Rohrkrümmers

Ein Beispiel, eines von Rechenberg durchgeführten Experiments (Abbildung 2.12 [Rec07]), unter Verwendung von evolutionären Strategien, soll den Nutzen der evolutionären Algorithmen verdeutlichen. Dabei galt es bei einem 90°-Krümmer den Energieverlust beim Durchströmen von Pressluft zu minimieren.

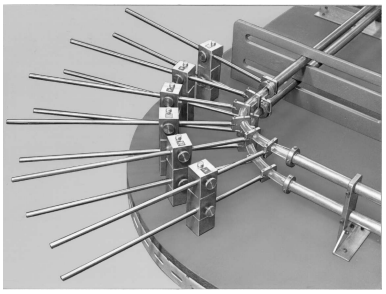


Abbildung 2.12: Beispiel eines 90°-Krümmers

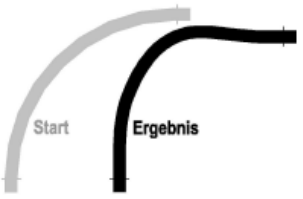


Abbildung 2.13: Krümmerkurve

Die Abbildung 2.13 [Rec07] zeigt die Start- und Endeinstellung des Krümmers. Am nicht so einfach vorhersehbaren Evolutionsergebnis ist zu erkennen, dass die Krümmung von der Geraden an stetig zunimmt und am Auslauf eine Krümmungsumkehr auftritt. Im Vergleich mit der Ausgangsform besitzt die Endform eine Energieersparnis von 2%.

2.3.2 Evolutionäre Operatoren und Verfahren

Der Suchraum S ist die Menge aller möglichen Lösungen eines Optimierungsproblems. Ein Individuum ist ein Element des Suchraums S. Es besitzt eine gewisse Güte (Fitness), die es von einer über den Suchraum definierten Fitnessfunktion fit zugewiesen bekommt. Gesucht wird nach dem am besten angepassten Individuum mit der maximalen bzw. minimalen Güte [GKK04].

Eine Population P ist eine Menge von μ Individuen. Sie enthält alle nach der Selektion überlebenden Individuen, die sich wiederum vermehren und somit λ Nachkommen für eine nachfolgende Population P bilden. Es muss ein Abbruchkriterium erfüllt sein, sonst pflanzen die μ Individuen einer Population P sich endlos weiter fort. Das Erreichen einer bestimmten Fitness, eine maximale Anzahl t_{max} an Generationen $P(t)$ oder eine maximale Anzahl $eval_{max}$ an Bewertungen, die sich aus $eval_{max} = \mu + t_{max} \cdot \lambda$ ergibt, sind die häufigsten Abbruchkriterien. Eine Generation $P(t)$ ist eine Population P zu einem Zeitpunkt t der Evolution. Ein Chromosom c beschreibt die Eigenschaften eines Individuums. Es besitzt eine Menge n von Genen x, deren Position i im Chromosom c duch den Locus beschrieben wird. Ein Gen x enthält die Erbinformation in Form eines Werts x_i, der in der Biologie als Allel bezeichnet wird.

S	Suchraum ist die Menge aller möglichen Lösungen.
fit	Fitnessfunktion oder Kostenfunktion bewertet die Güte eines Individuums.
c	Chromosom legt den Bauplan bzw. Eigenschaften eines Individuums fest.
x	Gen ist ein einzelnes Teilstück eines Chromosoms.
n	Chromosomlänge ist die Anzahl der Gene eines Chromosoms.
x_i	Allel ist die Ausprägung bzw. der Wert eines Gens.
i	Locus ist der Ort bzw. die Position eines Gens im Chromosom.
P	Population ist eine Menge von Individuen.
μ	Populationsgröße ist die Anzahl der Individuen einer Population.
$P(t)$	Generation ist eine Population zu einem Zeitpunkt der Evolution.
λ	Nachkommenanzahl ist die Anzahl der Kinder, die aus den Individuen einer Population erzeugt werden.
σ	Mutationsschrittweite ist die Schrittweite mit der Gene mutieren.

Tabelle 2.1: Die wichtigsten Bezeichnungen der "Evolutionären Algorithmen"

2.3.2.1 Initialisierung

Die Initialisierung des Evolutionären Algorithmus ist sehr wichtig, weil sie entscheident für die Nachkommen bzw. das Verhalten der Konvergenz ist. Die Startpopulation ist eine Menge von λ Chromosomen und wird innerhalb der vom Benutzer angegebenen Grenzen erstellt. Die Chromosome und ihre Gene werden zufällig über den begrenzten Bereich verteilt. Durch einsetzten von bereits bekannten guten Genen mit einer guten Fitness, kann es zu einer vorzeitigen Konvergenz kommen, weil die Gene ungewollt in ein lokales Optimum laufen könnten. Andere Gene hätten keine Chance ihre Werte durchzusetzen, da sie von den stärkeren Genen verdrängt werden.

2.3.2.2 Fitnessfunktion

Die Fitnessfunktion (Kostenfunktion) entscheidet über die Güte eines Individuums. Sie ist entscheident für die Überlebenswahrscheinlichkeit jedes Chromosoms. Es ist eine Abbildung, die jedem Element des Suchraums S eine üblicherweise reele Zahl als Bewertung zuweist.

$$fit : S \rightarrow \mathbb{R} \tag{2.1}$$

Innerhalb einer Population gilt, dass die Stärkeren überleben. Individuen mit sehr guten und anpassungsfähigen Eigenschaften, bezüglich der Fitnessfunktion, haben eine höhere Überlebens- und Vermehrungschance. Im Falle einer Maximierung gilt, dass das Chromosom c_1 eine bessere Lösung repräsentiert, als das Chromosom c_2, wenn

$$fit(c_1) > fit(c_2)$$

zutrifft.

2.3.2.3 Selektion

Die Selektionsalgorithmen beschreiben die Kriterien, nach denen aus den vorhandenen λ Individuen eine neue Population mit μ Individuen erstellt wird.

Deterministischer Wettkampf (Deterministic Tournament(k)):
Beim deterministischen Wettkampf muss eine Wettkampfgröße k, die größer gleich 2 sein muss, angegeben werden. Es wird μ mal k zufällig gewählte Individuen aus der Menge der λ Individuen ausgesucht und deren Fitness verglichen. Das jeweils beste der k Individuen wird in die nächste Population übernommen.

Stochastischer Wettkampf (Stochastic Tournament(h)):
Beim stochastischen Wettkampf treten die Individuen paarweise gegen h Gegner an. h muss vor Beginn des Wettkampfes festgelegt werden und einen Wert zwischen $0.5\mu > h > 0.1\mu$ besitzen. Ein Individuum erringt einen Sieg, wenn sein Fitnesswert mindestens genau so gut wie der seines Gegners ist. Nach diesem Wettkampf werden die Individuen nach der Anzahl ihrer Siege sortiert, wobei die besten μ Individuen dieser Reihenfolge in die neue Population übernommen werden. Bei Punktegleichheit entscheidet der Fitnesswert. Obwohl der Fitnesswert nur zweitrangig ist und auch schlechtere Individuen eine Chance haben, ist sichergestellt, dass das beste Individuum nicht verlieren und das schlechteste nicht gewinnen kann.

Roulette:
Roulette Selektion ist ein Auswahlverfahren, bei dem die Selektionswahrscheinlichkeit mit der Fitness proportional wächst. Es wird für jedes Individuum die Selektionswahrscheinlichkeit in einer seperaten Tabelle berechnet. Dann wird eine standardnormalverteilte Zufallszahl zwischen 0 und 1 erwürfelt und die Tabelle in der Reihenfolge durchgegangen. Dabei wird der Zufallswert mit der errechneten Wahrscheinlichkeit verglichen. Sollte der Zufallswert kleiner als die Wahrscheinlichkeit sein, so wird das Individuum in die nächste Population übernommen. Anschliessend wird eine neue Zufallszahl bestimmt und die Tabelle wieder von vorne durchgegangen. Dies hat zur Folge, daß auch schlechte Individuen eine Chance haben, in die nächste Population übernommen zu werden.

Ranking(s,e):
Ranking Selektion ist ein Auswahlverfahren, bei dem der Rang eines Individuums über die Selektion entscheidet. Es werden alle Individuen nach ihrer Fitness in einer Tabelle eingeordnet. Anschließend wird eine Selektionswahrscheinlichkeit, die vom Rang des Individuums abhängt, errechnet. Es wird ein Erwartungswert s zwischen $1.0 < s < 2.0$ festgelegt, mit dem die Selektionswahrscheinlichkeit beim linearen Ranking nach einer linearen Funktion errechnet wird.

Um den Selektionsdruck der Individuen zu steigern, kann das exponentielle Rankingverfahren benutzt werden. Die exponentielle Komponente e, die im linearen Verhahren 1 ist, muss um den gewünschten Wert erhöht werden.

2.3.2.4 Mutation und Kreuzung

Durch die genetischen Operatoren Mutation und Kreuzung (*Crossover*) werden die ausgewählten Chromosome c der Generation $P(t)$ verändert, um Nachkommen für die folgende Population $P(t+1)$ zu erzeugen. Aus den existierenden Chromosomen sollen noch fittere Kinder entstehen. Dazu werden die Erbinformationen verändert, ausgetauscht und Gene mit sehr guten Eigenschaften vermehrt vererbt.

Der Operator **Kreuzung** oder **Crossover** dient der Kreuzung zweier Chromosome. Es werden, aus zwei Chromosomen, Teilketten miteinander vertauscht, wodurch zwei neue Chromosome entstehen. Welche Chromosome miteinander gekreuzt werden, wird nach einer standardnormalverteilten Zufallszahl bestimmt. Immer paarweise werden die erwürfelten Chromosome mit einer gewissen Kreuzungswahrscheinlichkeit (*Crossoverprobability*) p_c gekreuzt. Das geschieht solange, bis alle Chromosome einmal die Chance hatten sich mit einem anderen Chromosom zu kreuzen. Bei einer ungeraden Populationsgröße λ erhält das übriggebliebene Chromosom, keine Chance sich zu kreuzen.

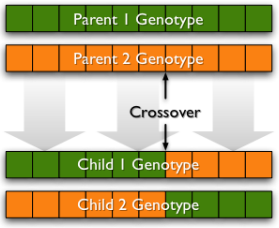


Abbildung 2.14: Kreuzung zweier Chromosome [Sau]

Die Kreuzung beginnt mit der zufälligen Ermittlung eines Kreuzungspunktes l. Dieser Kreuzungspunkt l muss sich innerhalb der Chromosomlänge n befinden.

$$l \in \{1, ..., n-1\} \tag{2.2}$$

Es werden alle Gene $[x_1, ..., x_l]$ bis zum Kreuzungspunkt l des ersten Chromosoms c_1 mit den Genen $[x_1, ..., x_l]$ bis zum Kreuzungspunkt des zweiten Chromosoms c_2 vertauscht. Die zwei neu entstandenen Chromosome c_1' und c_2' werden an den nächsten Operator weiter gereicht.

Der Operator **Mutation** dient der zufälligen Änderung eines oder mehrerer Gene im Chromosom. Es wird $n \cdot \lambda$ mal eine Zufallszahl generiert, die entscheidet ob ein Gen eines Chromosoms mutieren soll oder nicht. Die Mutationswahrscheinlichkeit (*Mutationprobability*) p_m wird zu Beginn der Optimierung festgelegt. Wenn ein Gen mutieren soll, dann wird für das zu erzeugende Allel x_i' das ursprüngliche Allel x_i mit einer standardnormalverteilten Zufallszahl modifiziert.

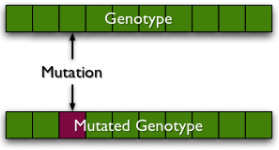


Abbildung 2.15: Mutation eines Chromosoms [Sau]

Die Entscheidung, in welche Richtung die Mutation gehen soll, hängt von einer weiteren Pseudozufallszahl z ab. Die Funktion $\triangle(x, y)$ ist eine mit x monoton fallende Funktion mit dem Wertebereich von $[0, ..., y]$. Ein Gen x mit dem Definitionsbereich von $[u_i, ..., o_i]$ mutiert in der unten angegebenen Form nur innerhalb seines zulässigen Difinitionsbereichs.

$$x_i' = \begin{cases} x_i + \triangle(t, o_i - x_i) & z = 0 \\ x_i - \triangle(t, x_i - u_i) & z = 1 \end{cases} \tag{2.3}$$

Die verschiedenen Evolutionsstrategien, nach denen ein Gen mutiert, werden im Abschnitt 2.3.3 näher erklärt.

2.3.2.5 Rekombination

Die Rekombination, wie der Name schon sagt, kombiniert zwei Chromosome miteinander. Das in Abbildung 2.14 beschriebene One-Point Crossover ist die einfachste Form der Rekombination. Es gibt sehr viele Crossover-Operatoren, jedoch beschränkt sich dieser Abschnitt auf die Operatoren, die von der *Feedback Control* zur Verwendung angeboten werden. Aus dieser Sichtweise kann man die Rekombination in drei generelle Kreuzungsmechanismen einordnen.

Die **Discrete Recombination** ermittelt k zufällige Kreuzungspunkte l_i. Diese Kreuzungspunkte l_i müssen sich innerhalb der Chromosomlänge n befinden und mindestens einen Abstand von 1 besitzten. Es werden alle Gene, bis zu dem ermittelten Kreuzungspunkt l_i, zwischen den zwei Chromosomen vertauscht.

$$c_1' = (x_1, ..., x_l, y_{l+1}, ..., y_n) \tag{2.4}$$

$$c_2' = (y_1, ..., y_l, x_{l+1}, ..., x_n) \tag{2.5}$$

Das Beispiel in Abbildung 2.16 [ES03] zeigt die Discrete Intermediate Recombination mit $k = 2$ Kreuzungspunkten bei $l_1 = 4$ und $l_2 = 6$.

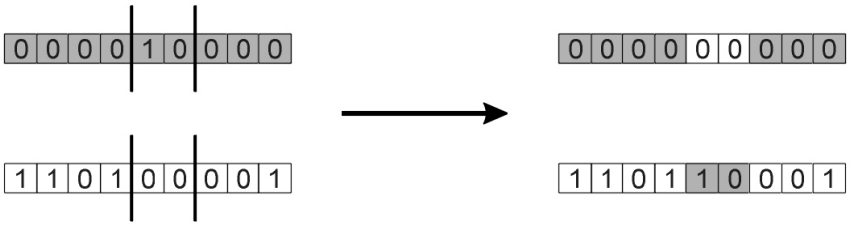


Abbildung 2.16: Discrete Recombination durch k-Point Crossover

Die **Standard Intermediate Recombination** ermittelt einen zufälligen Kreuzungspunkt l. Die beiden Teilketten $[x_{l+1}, ..., x_n]$ und $[y_{l+1,...,y_n}]$ der Chromosome $c_1(x)$ und $c_2(y)$, die durch den Kreuzungspunkt l getrennt werden, werden bei der Intermediate oder Arithmetischen Recombination nicht miteinander vertauscht, sondern mit einem Faktor α arithmetisch addiert. In der am häufigsten verbreiteten Variante wird als Faktor $\alpha = \frac{1}{2}$ gewählt, um den Mittelwert der beiden Genpaare zu erhalten.

$$c_1' = (x_1, ..., x_l, \alpha \cdot y_{l+1} + (1 - \alpha) \cdot x_{l+1}, ..., \alpha \cdot y_n + (1 - \alpha) \cdot x_n) \tag{2.6}$$

$$c_2' = (y_1, ..., y_l, \alpha \cdot x_{l+1} + (1 - \alpha) \cdot y_{l+1}, ..., \alpha \cdot x_n + (1 - \alpha) \cdot y_n) \tag{2.7}$$

Das Beispiel in Abbildung 2.17 [ES03] zeigt die Standard Intermediate Recombination mit einem Kreuzungspunkt $l = 6$ und einem Faktor $\alpha = \frac{1}{2}$.

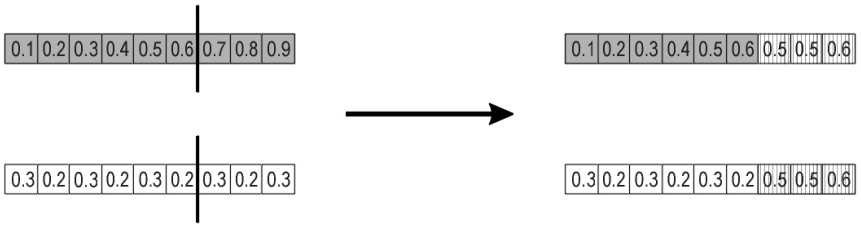


Abbildung 2.17: Standard Intermediate Recombination durch One-Point Crossover

Die **Global Intermediate Recombination** funktioniert genauso, wie die Standard Intermediate Recombination. Es muss hier allerdings kein Kreuzungspunkt ermittelt werden, da das komplette Chromosom $c_1(x)$ mit dem kompletten Chromosom $c_2(y)$ arithmetisch gekreutzt wird.

$$c_1' = \alpha \cdot \overrightarrow{x} + (1 - \alpha) \cdot \overrightarrow{y} \tag{2.8}$$

$$c_2' = \alpha \cdot \overrightarrow{y} + (1 - \alpha) \cdot \overrightarrow{x} \tag{2.9}$$

Das Beispiel in Abbildung 2.18 [ES03] zeigt die Global Intermediate Recombination mit einem Faktor $\alpha = \frac{1}{2}$.

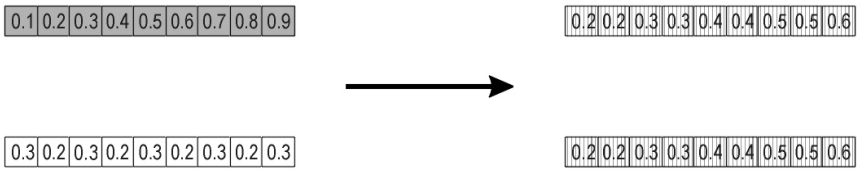


Abbildung 2.18: Global Intermediate Recombination durch One-Point Crossover

Ein Problem der Rekombination ist die geringe Variation von Chromosomen, da die Gene der Chromosomen nur vertauscht bzw. arithmetisch kombiniert werden. Die Kreuzungswahrscheinlichkeit (Crossoverprobability) p_c und der Suchraum S der Gene sollten dem Problem entsprechend angepasst sein [GKK04].

2.3.3 Evolutionsstrategien

(1 + 1) - ES: Jede Population umfasst nur ein Individuum. Es generiert ein neues Individuum aus dem Elter, wobei bei dieser Methode die Selektion entfällt. Nur durch Mutation wird der Nachkomme verändert. Um die Fitness zu berechnen, wendet man die Fitnessfunktion an, denn nur der Fittere der beiden Individuen geht in die nächste Generation über.

(μ + 1) - ES: Jede Generation umfasst μ Individuen. Nach dem Zufallsprinzip wird ein Elter ausgesucht und mit ihm ein neues Individuum generiert. Nach der Berechnung der Fitness wird das am wenigsten fitte Individuum entfernt und die nächste Generation wird aus den verbleibenden Individuen gebildet.

$(\mu + \lambda)$ - ES: Jede Generation umfasst μ Individuen. Aus den μ Eltern werden λ Kinder generiert, dabei muss $\lambda \geq \mu$ erfüllt sein. Von allen Kindern und Eltern berechnet man die Fitness, wobei die μ Fittesten aus Eltern und Kindern in die nächste Generation übergehen. Man bezeichnet den Quotient λ/μ als Selektionsdruck.

(μ, λ) - ES: Jede Generation umfasst μ Individuen. Aus den μ Eltern werden λ Kinder generiert, dabei muss $\lambda \geq \mu$ erfüllt sein. Für das Entstehen der nächsten Generation spielen die Eltern keine Rolle mehr, da sie gelöscht werden. Nach der Berechnung der Fitness von allen λ Kindern werden die μ Fittesten in die nächste Generation übernommen. Die Fitnesskurve ist nicht monoton steigend, da kein Individuum länger als eine Generation überlebt.

$(\mu', \lambda'(\mu, \lambda)\nu)$ - ES: μ' Elter erzeugen λ' Kinder und die Kinder werden für ν Generationen isoliert. In jeder der ν Generationen generiert jede isolierte Population λ Nachkommen. Die Fitness wird berechnet und die μ Fittesten der Nachkommen werden in die nächste Generation übernommen. Nach den ν isolierten Generationen wird die beste der λ' Populationen ausgewählt und erzeugt als nächste Elterpopulation wieder λ' Kinder. Wie auch durch $(\mu + \lambda)$ - ES bzw. (μ, λ) - ES lassen sich auch mit verschachtelten Evolutionsstrategien multimodale Optimierungsprobleme lösen [FR98].

2.3.3.1 Unkorrelierte Mutation mit nur einer Mutationsschrittweite

Die evolutionäre Strategie, der unkorrelierten Mutation mit nur einer Mutationsschrittweite σ, ist der einfachste Algorithmus. Es existiert für ein Chromosom c bestehend aus n Genen x nur eine Mutationsschrittweite σ.

$$c = (x_1, ..., x_n, \sigma) \tag{2.10}$$

Diese Schrittweite mutiert mit jeder Population für das zu erzeugende Individuum. Bei der Mutation wird σ mit einem Term aus Streuung τ_{local} und einer Zufallszahl $N(0, 1)$, der eine gaußschen Standardnormalverteilung zugrunde liegt, multipliziert.

$$\sigma' = \sigma \cdot e^{\tau_{local} \cdot N(0,1)} \tag{2.11}$$

Es wird eine untere Grenze $\varepsilon = \sigma_{min}$ festgelegt. Wenn diese Grenze unterschritten wird, so wird die Mutationsschrittweite auf die untere Grenze gesetzt.

$$\sigma' < \varepsilon \Rightarrow \sigma' = \varepsilon \tag{2.12}$$

Lässt man beliebig kleine Werte für σ zu, können Chromosome einen Selektionsvorteil erhalten, indem sie sich auf ein lokales Optimum konzentrieren, was zur vorzeitigen Konvergenz führt [Sch95].

Die Streuung τ_{local}, die auch als Lernrate bezeichnet wird, ist entscheident für die Änderung der Mutationsschrittweite σ. Sie wird zu Beginn des evolutionären Algorithmus festgelegt und bleibt konstant. Durch ein zu großes τ_{local} konvergiert das σ zum Erwartungswert 0 nur sehr langsam oder sehr schnell. Es werden dabei viele Zwischenschritte übersprungen oder gar nicht erreicht, dass zur frühzeitigen Konvergenz in ein lokales Optimum führen kann. Eiben und Smith empfehlen eine parameterabhängige Streuung [ES98]:

$$\tau_{local} \propto \frac{1}{\sqrt{n}} \tag{2.13}$$

Von den Entwicklern der EO Bibliothek wird die Streuung τ_{local} mit einem Zählerparameter $\tau_{init_{local}}$, der mit 1 initialisiert wird, berechnet.

$$\tau_{init_{local}} = 1 \tag{2.14}$$

$$\tau_{local} = \frac{\tau_{init_{local}}}{\sqrt{2n}} \tag{2.15}$$

Das mutierte Gen resultiert aus dem ursprünglichen Gen, summiert mit dem Produkt der Mutationsschrittweite und einer standardnormalverteilten Zufallszahl.

$$x_i' = x_i + \sigma' \cdot N_i(0,1) \tag{2.16}$$

Zu beachten ist, dass $N_i(0,1)$ für jedes Gen neu bestimmt wird.

2.3.3.2 Unkorrelierte Mutation mit n Mutationsschrittweiten

Die evolutionäre Strategie, der unkorrelierten Mutation mit n Mutationsschrittweiten, ist dem in Abschnitt 2.3.3.1 erklärten Algorithmus sehr ähnlich. Es existieren für ein Chromosom c bestehend aus n Genen x auch n Mutationsschrittweiten σ.

$$c = (x_1, ..., x_n, \sigma_1, ..., \sigma_n) \tag{2.17}$$

Die Schrittweite mutiert mit jeder Population für jedes Gen des zu erzeugenden Individuums. Die Mutation von σ erfolgt durch die Multiplikation mit einem Term aus einer lokalen Streuung τ_{local} und einer globalen Streuung τ_{global}. Die globale Streuung τ_{global} wird mit einer zu Beginn der Mutation erzeugten Zufallszahl $N(0,1)$,der eine gaußsche Standardnormalverteilung zugrunde liegt, multipliziert. Die lokale Streuung τ_{lokal} wird mit einer Zufallszahl $N_i(0,1)$, die für jedes Gen neu erzeugt wird, multipliziert.

$$\sigma_i' = \sigma_i \cdot e^{\tau_{global} \cdot N(0,1) + \tau_{local} \cdot N_i(0,1)} \tag{2.18}$$

Die Begrenzung, der Schrittweite durch $\varepsilon = \sigma_{min}$, ist identisch zur Mutation mit nur einer Mutationschrittweite.

$$\sigma_i' < \varepsilon \Rightarrow \sigma_i' = \varepsilon \tag{2.19}$$

Die empfohlenen Werte der globalen Streuung τ_{global} und lokalen Streuung τ_{local} sind nach Bäck [BS93]

$$\tau_{global} \propto \frac{1}{\sqrt{2n}} \tag{2.20}$$

$$\tau_{local} \propto \frac{1}{\sqrt{2\sqrt{n}}} \tag{2.21}$$

oder nach Nissen [Nis97]

$$\tau_{global} \approx 0.1 \tag{2.22}$$

$$\tau_{local} \approx 0.2 \tag{2.23}$$

Von den Entwicklern der EO Bibliothek werden die Streuungen τ_{local} und τ_{global} mit den Zählerparametern $\tau_{init_{local}}$ und $\tau_{init_{global}}$, die jeweils mit 1 initialisiert werden, berechnet.

$$\tau_{init_{global}} = 1 \qquad \tau_{global} = \frac{\tau_{init_{global}}}{\sqrt{2n}} \tag{2.24}$$

$$\tau_{init_{local}} = 1 \qquad \tau_{local} = \frac{\tau_{init_{local}}}{\sqrt{2\sqrt{n}}} \tag{2.25}$$

Die Mutation erfolgt somit aus dem ursprünglichen Gen und der ihm zugeordneten Mutationsschrittweite, sowie einer standardnormalverteilten Zufallszahl.

$$x_i' = x_i + \sigma_i' \cdot N_i(0,1) \tag{2.26}$$

Die zusätzlichen Parameter σ_i spielen bei der Berechnung der Kostenfunktion keine Rolle. Allerdings werden Chromosomen mit gut angepassten Mutationsschrittweiten im Mittel bessere Nachkommen erzeugen. Durch die zusätzlichen Parameter soll sich die Mutationsschrittweite automatisch anpassen. Man erhofft sich, dass Gene mit schlechten σ_i Werten durch die Selektion ausgetilgt werden. Die Lösung der Gene wird durch zu kleine σ_i Werte zu langsam vorangebracht und bei zu großen σ_i Werten zu leicht verschlechtert [GKK04].

2.3.3.3 Korrelierte Mutation

Die evolutionäre Strategie der korrelierten Mutation, versucht nicht wie die bisher beschriebenen Algorithmen aus Abschnitt 2.3.3.1 und 2.3.3.2 das Optimum, durch zufällige Änderung der einzelnen Parameter, unabhängig von einander zu finden, sondern Korrelationen zwischen den einzelnen Parametern zu berücksichtigen. Es wird dabei nicht nur in die vielversprechendste Richtung gesucht, sondern der Suchraum in der Nähe des Chromosoms, in alle Richtungen mit gleicher Wahrscheinlichkeit untersucht. Ein Chromosom c besteht aus n Genen x, n Schrittweiten σ und $\frac{n(n-1)}{2}$ Rotationswinkeln α.

$$c = (x_1, ..., x_n, \sigma_1, ..., \sigma_n, \alpha_1, ..., \alpha_{\frac{n(n-1)}{2}}) \tag{2.27}$$

Die Schrittweite mutiert, wie gehabt, mit jeder Population für jedes Gen des zu erzeugenden Individuums, jedoch fliesst sie nicht mehr als Addition direkt in die Mutation mit ein. Das σ ist nun die Schrittweite der Kovarianzmatrix.

$$\sigma_i' = \sigma_i \cdot e^{\tau_{global} \cdot N(0,1) + \tau_{local} \cdot N_i(0,1)} \tag{2.28}$$

$$\sigma_i' < \varepsilon \Rightarrow \sigma_i' = \varepsilon \tag{2.29}$$

Der Rotationswinkel α wird für jedes Chromosom neu adaptiert. Es wird der ursprüngliche Rotationswinkel mit dem Produkt einer Winkelstreuung β, die der Streuung τ der Mutationsschrittweite entspricht, und einer standardnormalverteilten Zufallszahl, summiert.

$$\alpha_j' = \alpha_j + \beta \cdot N_j(0,1) \tag{2.30}$$

Der resultierende Rotationswinkel α_j' muss sich innerhalb eines Bereichs befinden. Die Grenzen sind auf $[-\pi, \pi]$ festgelegt.

$$|\alpha_j'| > \pi \Rightarrow \alpha_j' = \alpha_j' - \pi \cdot \frac{\alpha_j'}{\pi} \tag{2.31}$$

Üblicherweise wird die Winkelstreuung β auf 5° festgelegt ($\beta = 0.0873$). Die Rotationsmatrix $R_{ij}(\phi)$ enthält in der i-ten und j-ten Zeile den $\cos(\phi)$ Eintrag. Alle außerhalb der Diagonalen nicht angegebenen Einträge sind 0.

$$R_{ij}(\phi) = \begin{Bmatrix} \sigma'_1 & & & & & & \\ & \ddots & & & & & \\ & & \cos(\phi) & & -\sin(\phi) & & \\ & & & \ddots & & & \\ & & & & \sigma'_i & & \\ & & & & & \ddots & \\ & & \sin(\phi) & & \cos(\phi) & & \\ & & & & & \ddots & \\ & & & & & & \sigma'_n \end{Bmatrix} \tag{2.32}$$

Bei der korrelierten Mutation berechnet man einen Vektor Δ, der einen Mutationsschritt repräsentiert.

$$\Delta_{korreliert} = (\prod_{i=1}^{n} \prod_{j=1}^{\frac{n(n-1)}{2}} R_{ij}(\alpha_{ij})) \tag{2.33}$$

Das durch die korrelierte Mutation resultierende Chromosom x'_i wird dann folgendermaßen berechnet.

$$x'_i = x_i + \Delta_{korreliert} \tag{2.34}$$

3. Aufgabenstellung

Thema:

Implementierung der „*feedback control* “-
Optimierung des Alternate Gradient Decelerators

Umfang:

Bei der Stark-Abbremsung großer Moleküle in einem Alternate Gradient Decelerator sind insbesondere die genauen Zeitsequenzen zur Ansteuerung der eingesetzten Hochspannungsschalter sehr wichtig. Diese Zeitsequenzen sollen mittels des Datenerfassungs- und Steuerungsprogramms KouDA unter Verwendung von evolutionären Strategien für ein Experiment optimiert werden.

Im Rahmen dieser Diplomarbeit sollen fortgeschrittene Algorithmen, wie z.B. CMA-ES (2.3.3.3) in KouDA unter Verwendung der EO Bibliothek implementiert und an dem komplexen AG-Decelerator-Experiment getestet werden.

Dies erfordert eine Neuimplementation des *feedback control* Programmcodes in KouDA-1.0. Der Programmcode soll modular gehalten werden, um die ständig wachsende Anzahl der Controller leichter zur Optimierung in die Struktur einzubinden. Die Berechnungen und Parametrisierungen der Geräte sollen optimiert werden. Der zeitliche Ablauf muss somit multithreadfähig gehalten und implementiert werden. Die Optimierungsparameter der Controller sollten variabel und dem Gerät spezifisch zugeordnet sein.

Es soll eine grafische Benutzeroberfläche geschaffen werden, die eine Handhabung der Evolutionsoptionen erleichtert. Die ermittelten Daten müssen ausgewertet und benutzerfreundlich dargestellt werden. Alle Optimierungsparameter mit ihren Werten, sowie die zum evolutionären Algorithmus genutzten Einstellungen, müssen in einem sinnvollen Format in Dateien auf der Festplatte gespeichert werden.

Eine Auswertung der Daten mit den genutzten Einstellungen des evolutionären Algorithmus soll dokumentiert und als Hilfe den Benutzern des Programms am Fritz-Haber-Instituts zur Verfügung gestellt werden.

4. Entwurf und Implementierung

Die Implementierung der *Feedback Control* Optimierung ist ein in sich geschlossener Programmteil. Es werden lediglich die bereits bestehenden *Controller* Klassen von KouDA zur Parametrisierung verwendet. Die *Scan* Klasse, die für die Erfassung der Daten benutzt wird, wurde modularisiert und angepasst, damit eine Verwendung möglich wurde.

Das Kapitel Entwurf und Implementierung zeigt viele Quellcode Ausschnitte, die jedoch nur den Programmierstil zeigen und zum Verständnis beitragen sollen. Alle in diesem Kapitel befindlichen Quellcode Ausschnitte enthalten nur wenig Kommentare. Der orginale Quellcode ist jedoch ausführlich kommentiert und kann zu jeder Zeit auf der beiliegenden CD im Anhang B eingesehen werden.

Die Klassen der folgenden Strukturen haben aufgrund der Klassenhirarchie sehr ähnliche Namen. Um das Verständnis der Struktur zu verdeutlichen, sind die wichtigsten Klassen farbig dargestellt. Dabei entsprechen die Farben der abgeleiteten Klassen den der Basisklasse. Für abgeleitete Klassen mit unterschiedlicher Verwendung, ist eine andere aussagekräftige Farbe gewählt worden. Die Klassendiagramme beinhalten nur die für den Abschnitt relevanten Funktionen und Attribute.

4.1 Strukturierung der Implementierung

Die Implementierung ist in vier Header - und vier Cpp - Datein aufgesplittet. Jede dieser Headerdatei repräsentiert eine Ebene der Struktur. Die Abbildung 4.1 zeigt die vier Ebenen der *Feedback Control* Hirarchie.

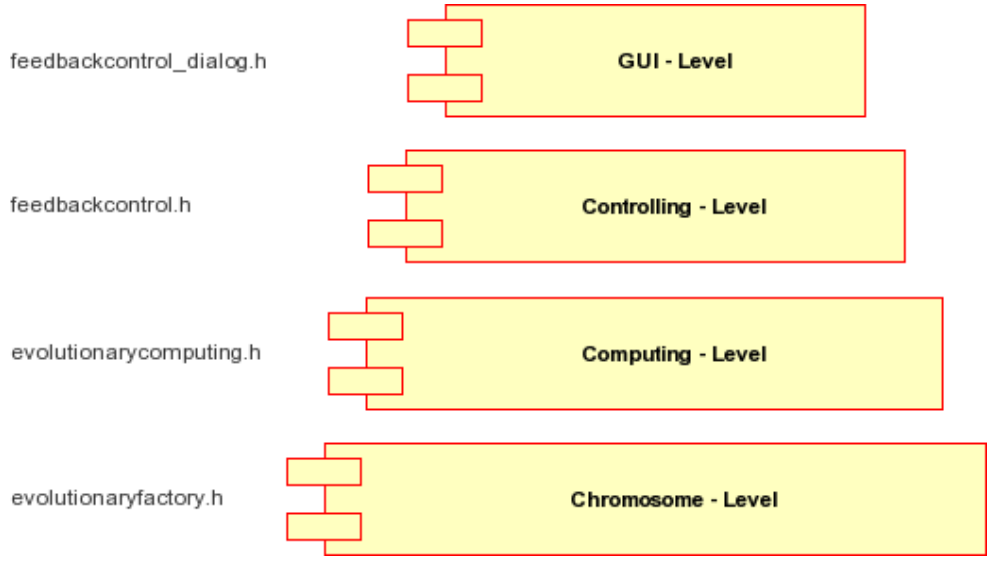


Abbildung 4.1: *Feedback Control* Hirarchie

- **Chromosome - Level:** Die Chromosom - Ebene kapselt alle Gene mit ihren Eigenschaften sowie die verfügbaren *Controller* und Pseudotypen[1].
- **Computing - Level:** Die Berechnungs - Ebene speichert die Evolutionsparameter. In dieser Ebene findet die Abarbeitung des Evolutionären Algorithmus statt.
- **Controlling - Level:** In der *Feedback Control* - Ebene werden die berechneten Werte der Berechnungs - Ebene auf die Chromosom - Ebene adaptiert. Hier erfolgt die Auswertung der Daten bzw. die Bewertung der Individuen.
- **GUI - Level:** In der grafischen Darstellungs - Ebene wird dem Benutzer die Möglichkeit gegeben die Evolutionsparameter zu verändern. Hier erfolgt die Auswahl der Gene sowie die Darstellung des Evolutionsfortschritts.

[1] Ein Pseudotyp ist ein Teilgenstrang im Chromosom. Die deteilierte Erleuterung erfolgt in Abschnitt 4.2.1.5.

4.2 Chromosomstruktur

Ein Chromosom legt den Bauplan und die Eigenschaften eines Individuums fest. Ein Gen ist ein Teilstück des Chromosoms. Es besitzt Teileigenschaften wie z.B. einen Suchraum oder das aktuelle Allel. Das Allel ist der vom Gen angenommene Wert zu einem Zeitpunkt. Der Suchraum eines Gens wird vom Benutzer zu Beginn der *Feedback Control* Optimierung festgelegt. Ebenso muss der Suchraum der Startpopulation festgelegt werden. Dieser sollte sich innerhalb des globalen Suchraums befinden. Abhängig vom verwendeten Algorithmus muss eventuell noch die Mutationsschrittweite angegeben werden.

Der Genotyp, der die formale Kodierung einer Lösung bzw. das Erscheinungsbild des Chromosoms definiert, kann durch den Benutzer beeinflusst werden. Das Erscheinungsbild ist abhängig von der Anzahl der Gene und ihren Eigenschaften. Jeder *Controller* stellt einen Genstrang des Chromosoms dar. Ein Chromosom kann viele Gene von verschiedenen *Controllern* und Pseudotypen beinhalten. Der Benutzer kann entscheiden, ob ein *Controller* mit seinen Pseudotypen in die *Feedback Control* Optimierung einbezogen wird. Das Chromosom hat das Wissen über die verfügbaren *Controller* und ihres derzeitigen Zustands. Eine Deaktivierung eines Genstrangs hat zur Folge, dass sich der Locus der Gene verändert. Der Locus bestimmt den Ort eines Gens im Chromosom. Aus diesem Grund ist der Locus auch nur dem Chromosom selber bekannt und wird nicht nach außen oder an irgendein Gen preisgegeben.

4.2.1 Inverses Observer Pattern

Die Gene müssen losgelöst von ihrem Locus und den *Controllern* sein. Das Allel eines Gens ist nur ein simpler double Wert eines Vektors. Das Gen ist ein Parameter eines *Controllers* ohne das Wissen, zu welchem *Controller* es gehört oder welchen Parameter des *Controllers* es repräsentiert.

Der Zweck eines Observer Pattern ist eine vereinfachte und schnellere Aktualisierung von Abhängigkeiten. Das bekannte Observer Pattern definiert eine 1 - zu - n Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objektes dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktuallisiert werden [GHJV04].

In der Chromosomstruktur existiert jedoch eine n - zu - 1 Abhängigkeit. Es existieren viele Gene, die von nur einem *Controller* oder einem Pseudotyp abhängig sind. Die Änderung des Allel eines Gens bewirkt eine Benachrichtigung des Genstrangs. In der angepassten Observerstruktur stellt ein Gen ein Subjekt und ein Genstrang einen Beobachter (Observer) dar. Wenn alle Gene eines Genstrangs ihren Zustand geändert haben, wird der *Controller* oder der Pseudotyp neu parametrisiert.

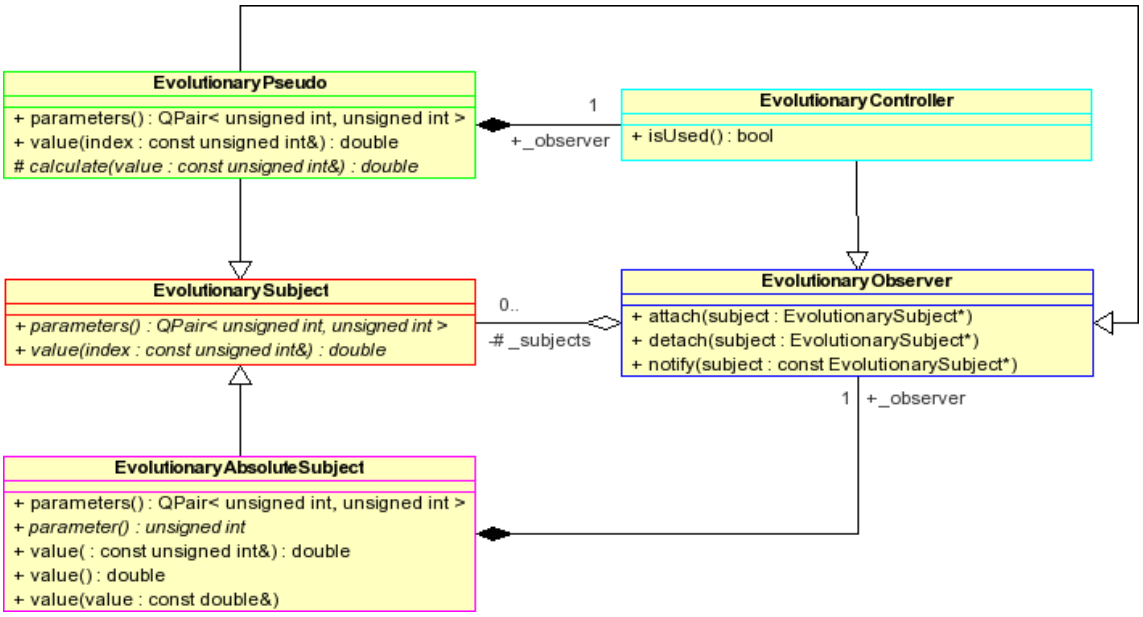


Abbildung 4.2: Klassendiagramm des “Inversen Observer Pattern”

Da die Idee auf dem Observer Pattern basiert, jedoch kein Observer Pattern mehr ist, erhielt die neu designte Struktur den Namen “Inverses Observer Pattern”. Das Klassendiagramm der Struktur ist in Abbildung 4.2 gezeigt.

4.2.1.1 *EvolutionarySubject*

Die abstrakte Klasse *EvolutionarySubject* deklariert die fundamentalen Eigenschaften einer Menge von aneinandergereihten Genen.

```
class EvolutionarySubject
{
public:
    virtual QPair<unsigned int, unsigned int> parameters() const throw() = 0;

    virtual double value(const unsigned int& index = 0) const = 0;

}; // EvolutionarySubject
```

Quellcode 4.1: *EvolutionarySubject* Klasse

Die zwei abstrakten Funktionen definieren die Schnittstelle des Teilgenstrangs. Die *parameters()* Funktion gibt den Bereich eines Teilgenstrangs zurück. Mit der *value()* Funktion kann das Allel eines Gens, an der indizierten Position innerhalb des Bereichs, abgefragt werden.

4.2.1.2 *EvolutionaryObserver*

Die abstrakte Klasse *EvolutionaryObserver* deklariert die fundamentalen Eigenschaften eines Genstrangs. Diese Klasse erbt von *QThread*[2], damit eine selbstständige Aktualisierung erfolgen kann. Sobald alle Gene des Genstrangs einen neuen Allel erhalten haben, wird die Aktualisierung durchgefürt.

[2] QThread ist eine Threadklasse der Qt Bibliothek. Sie ermöglicht eine objektorientierte Implementierung von Threads.

```
class EvolutionaryObserver : protected QThread
{
public:
    virtual void attach(EvolutionarySubject* subject) const throw();

    virtual void detach(EvolutionarySubject* subject) const throw();

    virtual void notify(const EvolutionarySubject* subject) const throw();

protected:
    mutable QList<EvolutionarySubject*> _subjects;
```

Quellcode 4.2: Registrierung der *EvolutionaryObserver* Klasse

Die Registrierung der Subjekte erfolgt durch Aufruf der *attach()* Funktion. Es muss der Zeiger des zu registrierenden Subjekts übergeben werden. Der Zeiger wird in eine geschützte Liste eingetragen, wodurch das Subjekt ein Teil des Genstrangs wird. Wenn das Subjekt seinen Zustand ändert, muss es seinen Beobachter von der Zustandsänderung benachrichtigen. Dazu wird die *notify()* Funktion aufgerufen und der Zeiger auf sich selbst zur Identifikation übergeben. Die Objekte bleiben solange Teilgenstränge des beobachtenden Genstrangs bis die *detach()* Funktion ausgefürt wird und sich das Subjekt somit wieder abmeldet.

```
public:
    virtual ~EvolutionaryObserver()
    {
        {
            QMutexLocker locker(&_mutex);
            _abort = true;
        }
        _semaphore.release();
        this->wait();
    };

protected:
    EvolutionaryObserver()
        : QThread(), _semaphore(), _mutex(), _abort(false)
    {
        this->start();
    };

    virtual void run();

private:
    QSemaphore _semaphore;

    QMutex _mutex;

    volatile bool _abort;
```

Quellcode 4.3: Aktualisierung der *EvolutionaryObserver* Klasse

Bei der Instantiierung der Klasse, werden die Synchronisationsobjekte erzeugt und initialisiert. Der Konstruktor ruft die *start()* Funktion der Basisklasse auf, wodurch ein Thread erzeugt wird. Die *QThread* Klasse ist so implementiert, dass die *run()* Methode des Objekts als Threadfunktion übergeben wird.

```
void EvolutionaryObserver::run()
{
    forever {
        // wait for the notify of all subjects
        _semaphore.acquire();
        // block any other thread to lock the mutex
        QMutexLocker locker(&_mutex);
        // check the thread abort
        if(_abort)
            break;
        // check the mode
        if(_storagePath == "") {
            this->update();
        } else {
            this->store(_storagePath);
            // switch back to update mode
            _storagePath = "";
        }
    }
}
```

Quellcode 4.4: Threadfunktion der *EvolutionaryObserver* Klasse

Qt stellt einige zusätzliche Schlüsselwörter, wie z.B. *forever*, das ausschließlich für Threadobjekte gedacht ist [BS07], zur Verfügung. Es ist eine Endlosschleife, die keine Abbruchbedingung hat und nur durch ein *break* beendet werden kann. Nach der Erzeugung des Threads, blockiert das Semaphore, dass mit dem Wert 0 initialisiert wurde, die Threadfunktion. Erst wenn das Semaphore inkrementiert und somit freigegeben wird, kann die Threadfunktion ausgeführt werden. Das Objekt kann den Zustand aktualisierend oder speichernd annehmen. Das *_storagePath* Attribut entscheidet über den momentanen Zustand. Je nach Zustand wird nach der Freigabe des Semaphore, entweder die abstrakte Funktion *update()* oder *store()* ausgeführt. Die abstrakt deklarierten Funktionen müssen in den abgeleiteten Klassen, für den Genstrang spezifischen Fall, implementiert werden.

```
signals:
    void updateFinished(bool success);

protected:
    /* Exception */
    class EvolutionaryParameterizationError : public EvolutionaryFactoryError
    { ... };

    /* Exception */
    class EvolutionaryControllerError : public EvolutionaryFactoryError
    { ... };

    virtual void update() throw(EvolutionaryParameterizationError,
                                EvolutionaryControllerError,
                                std::bad_exception) = 0;
```

Quellcode 4.5: Aktualisierung der *EvolutionaryObserver* Klasse

Wenn die *update()* Funktion erfolgreich abgearbeitet werden konnte, dann wird das *updateFinished()* Signal mit dem Erfolgszustand emittiert. Die emittierten *updateFinished()* Signale aller Genstränge werden über die *Application Message Queue*[3] in der *EvolutionaryChromosom* Klasse, die in Kapitel 4.2.4 näher beschrieben wird, ausgewertet. Das

[3] Die *Application Message Queue* ist ein von Qt zur Verfügung gestellte Kernstruktur, die eine Interprozesskommunikation durch eine *Message Queue* umsetzt.

Qt-Signal-Slot-Konzept[4] wird mit einer *Application Message Queue* Verbindung verwendet, um den sicheren Datenaustausch zwischen den Threads zu gewährleisten ohne einen Threads zu blockieren.

Die *update()* Funktion ist für zwei verschiedene Fehlerfälle spezifiziert:

- ***EvolutionaryParameterizationError*** ist eine geschachtelte *Exceptionklasse* innerhalb der *EvolutionaryObserver* Klasse. Sie wird geworfen, wenn es zu einer ungültigen Parametrisierung kommt. Dieser Fehlerfall ist dem evolutionären Algorithmus anzurechnen, da die EO Bibliothek (4.3) Werte erzeugt, die sich nicht zwangsläufig in jedem *Controller* als Parameter setzten lassen.
- ***EvolutionaryControllerError*** ist eine geschachtelte *Exceptionklasse* innerhalb der *EvolutionaryObserver* Klasse. Sie wird geworfen, wenn es sich um einen unvorhersehbaren Fehler innerhalb einer *Device* Klasse handelt. Dieser Fehlerfall sollte normalerweise nicht auftreten.

```
public:
    void storage(const QString& path);

protected:
    virtual void store(const QString& path) throw() = 0;

private:
    QString _storagePath;

}; // EvolutionaryObserver
```

Quellcode 4.6: Speicherung der *EvolutionaryObserver* Klasse

Ein *EvolutionaryObserver* Objekt befindet sich im aktualisierenden Zustand als Grundzustand. Wenn die *storage()* Funktion ausgeführt wird, wechselt das *EvolutionaryObserver* Objekt in den speichernden Zustand. Dazu wird ein Pfad sowie der Beginn eines Dateinamens als Parameter übergeben. Diese Zeichenkette wird um einen Genstrangspezifizierer erweitert und in dem *_storagePath* Attribut abgelegt. Ein möglicher Pfad mit Dateinamen könnte so aussehen: *"/tmp/eoStorage/popXXindiXX"*. Wenn nun die Teilgensträng bzw. die Gene (Subjekte) ihren Zustand ändern und das Semaphore freigegeben ist, dann wird statt der *update()* Funktion die *store()* Funktion ausgeführt. Jeder Genstrang speichert die Parameter in einem anderen Dateiformat auf die Festplatte. Die *store()* Funktion darf keine *Exception* werfen, da dies die eigentliche Abarbeitung der *Feedback Control* Optimierung behindern würde. Jede Fehlerbehandlung muss innerhalb der abgeleiteten Funktion geschehen.

4.2.1.3 *EvolutionaryAbsoluteSubject* (Gen)

Die *EvolutionaryAbsoluteSubject* Klasse erbt von *EvolutionarySubject* und repräsentiert genau ein Gen. Im Standard Observer Pattern sollte das abstrakt deklarierte Subjekt in der abgeleiteten Klasse definiert werden. In gewisser Weise geschieht dies auch, jedoch wird aus einer Genmenge, ein einzelnes Gen. Die Klasse implementiert die von der Basisklasse deklarierten Funktionen. Diese adaptieren auf neue Funktionsdeklarationen, die das *EvolutionaryAbsoluteSubject* wiederum zu einer abstrakten Klasse machen.

[4] Das Qt-Signal-Slot-Konzept ist ein makrobasiertes Konzept von Qt, dass von dem in Abschnitt 2.2.4 beschriebenen mmocCompiler umgesetzt wird.

```
class EvolutionaryAbsoluteSubject : protected QObject,
                                    public EvolutionarySubject
{
public:
    virtual ~EvolutionaryAbsoluteSubject()
    {
        _observer->detach(this);
    };

protected:
    EvolutionaryAbsoluteSubject(EvolutionaryObserver& observer)
        : QObject(), EvolutionarySubject()
    {
        _observer = &observer;
        _observer->attach(this);
    };

private:
    EvolutionaryObserver* _observer;
```

Quellcode 4.7: Registrierung der *EvolutionaryAbsoluteSubject* Klasse

Das konkrete Subjekt besitzt einen geschützten Konstruktor, damit nur abgeleitete Klassen ihn erreichen können. Der Konstruktor erhält als Parameter eine Instanz eines beobachtenden Genstrangs oder Teilgenstrangs. Der Zeiger auf den Beobachter wird in einem geschützten Attribut gespeichert. Da kein Gen zu einem undefinierten Genstrang gehören darf, ist der Konstruktor dafür verantwortlich, das Gen bei einem beobachtenden Genstrang zu registrieren. Der Destruktor muss das Gen während dessen Zerstörung bei dem beobachtenden Genstrang oder Teilgenstrang wieder abmelden, damit Zombies und falsche Adressierung vermieden werden.

```
public:
    virtual QPair<unsigned int, unsigned int> parameters() const throw()
        { return qMakePair(this->parameter(), this->parameter()); };

    virtual unsigned int parameter() const throw() = 0;

    virtual double value(const unsigned int&) const throw()
        { return this->value(); };

    virtual double value() const throw()
        { return _value; };

private:
    double _value;
```

Quellcode 4.8: Adaption der *EvolutionaryAbsoluteSubject* Klasse

Wie bereits erwähnt, repräsentiert die *EvolutionaryAbsoluteSubject* Klasse genau ein Gen und adaptiert die von der Basisklasse deklarierten Funktionen einer Genmenge. Die *parameters()* Funktion, die den Bereich eines Teilgenstrangs definiert, erhält für den Start- und Endwert die exakte Position des Gens innerhalb der Genmenge. Die Darstellung der Position kann je nach Gentyp sehr unterschiedlich ausfallen. Es wird eine weitere Funktion *parameter()* deklariert, die die exakte Position des Gens bzw. die Art des Parameters zurück gibt. Es wird dem spezifischen Gen überlassen, wie oder wodurch sich das Gen identifiziert oder beschreibt. Da es sich ausschließlich nur um ein Gen handelt besitzt es nur ein Allel. Der Wert des Allel wird in einem privaten Attribut gehalten und kann mit der Funktion *value()* von außen erhalten werden.

```
public:
    virtual void value(const double& value) throw()
    {
        _value = value;
        _observer->notify(this);
    };
```

Quellcode 4.9: Allelzuweisung der *EvolutionaryAbsoluteSubject* Klasse

Wird dem Gen ein neuer Allelwert zugewiesen, so muss der beobachtende Genstrang über die Zustandsänderung informiert werden. Das *EvolutionaryAbsoluteSubject* Objekt ruft dazu die *notify()* Funktion des übergeordneten Genstrangs oder Teilgenstrangs auf. Es muss ein Zeiger zur Identifizierung auf das Objekt selbst übergeben werden.

```
public:
    inline Controller::controller_t type() const throw()
    {
        return _observer->type();
    };

    inline bool isUsed() const throw()
    {
        return _observer->isUsed();
    };

}; // class EvolutionaryAbsoluteSubject
```

Quellcode 4.10: Rekursion der *EvolutionaryAbsoluteSubject* Klasse

Jedes Gen hat gewisse Eigenschaften, die zu dem Genstrang, zu dem es gehört, identisch sind. Wenn ein Genstrang vom Benutzter deaktiviert wird, weil dieser bei dem evolutionären Algorithmus nicht berücksichtigt werden soll, dann betrifft dies selbstverständlich auch das Gen. Zwei wichtige Eigenschaften werden durch die Funktionen *type()* und *isUsed()* vom Genstrang abgefragt. Da die Funktionen als *inline* definiert sind, ersetzt der Compiler die Funktionen durch die der *EvolutionaryObserver* Klasse.

4.2.1.4 *EvolutionaryController* (Genstrang)

Die *EvolutionaryController* Klasse erbt von *EvolutionaryObserver* und repräsentiert einen Genstrang eines eigenen Typs. Der Genstrang ist nicht nur eine Ansammlung von Genen und Teilgensträngen, sondern bietet zusätzlich das rekursive Verhalten von Eigenschaft an. Diese Klasse ist weiterhin abstrakt und hat nur indirekt einen spezifizierten Typ.

```
class EvolutionaryController : public EvolutionaryObserver
{
public:
    virtual QWidget* widget() throw();

    bool isUsed() const throw();

protected:
    EvolutionaryController(const Controller::controller_t& type);

    QCheckBox _usageBox;

}; // class EvolutionaryController
```

Quellcode 4.11: *EvolutionaryController* Klasse

Der Genstrang besitzt eine *QCheckBox*[5] als Attribut. Im Konstruktor wird das *QCheckBox* Objekt mit einem controllerspezifischen Namen, der als Parameter übergeben wird, initialisiert. Dieses Qt GUI Element, das mit der Funktion *widget()* von außen zu erreichen ist, kann zwei Zustände besitzen: *true* und *false*. Durch die Wahl einer grafischen Klasse, kann das Objekt dynamisch in eine Tabelle innerhalb eines Fensters geladen werden. Der Benutzer kann durch Änderung des Zustands direkt den gesamten Genstrang aktivieren oder deaktivieren. Die Funktion *isUsed()* gibt den aktuellen Zustand des Genstrangs zurück.

Alle abgeleiteten Klassen von *EvolutionaryController* spezifizieren einen diskreten *Controller*. Zur Instantiierung einer diskreten *EvolutionaryController* Klasse, muss ein Zeiger auf das entsprechende *Device* Treiberobjekt übergeben werden, da jeder *Controller* eine unterschiedliche Funktionsschnittstelle zur Parametrisierung besitzt. Die in Tabelle 4.1 zeigt eine Auflistung aller optimierbaren *Controller* mit den möglichen Optimierungsparametern.

Controller	**Absolut Parameter**	**Polynomiale Parameter**[6]
AG33220	frequency, amplitude	
BU1708	timing	C0...Cn, phi, v, f0, d0 f, f', f", d, d', d"
DG3008	timing	
DG535	timing	
IselCNC	position	
LaserRDLControl	RDL pos[7], P[8], I[8], D[8] HeNe pos[7], P[8], I[8], D[8]	
Ramp	voltage	
SMR20	frequency, power	

Tabelle 4.1: Optimierungsparameter der *Controller*

[5] Die *QCheckBox* ist eine grafische Klasse, die von der Qt Bibliothek zur Verfügung gestellt wird.
[6] Die polynomialen Parameter sind Pseudoparameter, die in Abschnitt 4.2.3 näher erleutert werden.
[7] RDL und HeNe sind Lasertypen, deren Frequenz durch den Parameter "pos" optimiert werden kann.
[8] P,I,D sind die Optimierungsparameter einer PID-Regelung, die die Laserfrequenz des RDL und HeNe stabilisiert.

4.2.1.5 *EvolutionaryPseudo* (Teilgenstrang)

Die *EvolutionaryPseudo* Klasse ist ein *EvolutionaryObserver* als auch ein *EvolutionarySubject* und repräsentiert einen Teil eines Genstranges. Der Teilgenstrang muss sich, wie ein einzelnes Gen, bei einem übergeordneten Genstrang registrieren. Gene, die sich dem Teilgenstrang anschliessen möchten, müssen sich wiederum bei diesem anmelden.

```
class EvolutionaryPseudo : public EvolutionaryObserver,
                           public EvolutionarySubject
{
protected:
    EvolutionaryPseudo(EvolutionaryController& observer,
                       const PseudoWidget::pseudo_t& kind);

    EvolutionaryController* _observer;
```

Quellcode 4.12: Instantiierung der *EvolutionaryPseudo* Klasse

Der Konstruktor erhält nicht nur eine Instanz von einem *EvolutionaryObserver* Objekt, sondern von einem *EvolutionaryController* Objekt. Somit ist sichergestellt, dass der übergeordnete Genstrang kein weiter Teilgenstrang ist. Die Registrierung beim beobachtenden Genstrang sowie das Weiterleiten von genstrangspezifischen Eigenschaften sind identisch zur *EvolutionaryAbsoluteSubject* Klasse und wurde bereits in Abschnitt 4.2.1.3 erklärt. Der zweite Parameter des Konstruktors beschreibt den Typ des Teilgenstrangs.

```
public:
    int order() const throw()
        { return ( _coefficients.size() - 1 ); };

    QVector<EvolutionaryFactory*> coefficients() const throw()
        { return _coefficients; };

protected:
    EvolutionaryFactory* add(const int& order);

    unsigned int remove(const int& order);

    void clear()
        { this->remove(-1); };

    QVector<EvolutionaryFactory*> _coefficients;
```

Quellcode 4.13: Gene der *EvolutionaryPseudo* Klasse

Die Gene eines Teilgenstrangs werden in dem *_coefficients* Klassenattribut vom Type *QVector*[9] aufbewahrt. Die Namensgebung erfolgte in Anlehnung an ein Polynom, deswegen erhält man von der Funktion *order()* auch den höchsten Index eines Gens. Mit den Funktionen *add()* und *remove()* werden Gene zu der angegebenen Ordnung erzeugt oder entfernt.

[9] *QVector* ist eine Templateklasse der Qt Bibliothek. Sie basiert aus der STL und wurde bereits in 2.2.4 erwähnt.

```
public:
    virtual double value(const unsigned int& index) const
                    throw(EvolutionaryControllerError)
    {
        if( !inRange( index ) )
            throw EvolutionaryControllerError("out of parameter range");
        return calculate( index );
    };

protected:
    virtual bool inRange(const unsigned int& value) const throw();

    virtual double calculate(const unsigned int& value) const = 0;
```

Quellcode 4.14: Werteberechnung der *EvolutionaryPseudo* Klasse

In der Implementierung der abstrakten Funktion *value()* der Basisklasse, wird aus Sicherheitsgründen der Definitionsbereich überprüft. Es wird ein Wert aus allen Genen des Teilgenstrangs berechnet. Dazu wird die abstrakt deklarierte Funktion *calculate()*, die in der abgeleiteten Klasse für den spezifischen Fall implementiert ist, verwendet.

```
public:
    QWidget* widget() const throw()
        { return _widget; };

    virtual QPair<unsigned int, unsigned int> parameters() const throw()
        { return _widget->bounds(); };

protected:
    PseudoWidget* _widget;

}; // class EvolutionaryPseudo
```

Quellcode 4.15: *PseudoWidget* der *EvolutionaryPseudo* Klasse

Die Klasse *PseudoWidget* ist ein mit dem Qt Designer erstelltes Grafikelement der Klasse *QWidget*[10]. Sie stellt entsprechende Funktionen zur Verfügung, um sie als geschlossenes System zu betrachten. Die Verarbeitung und Darstellung aller Gene des Teilgenstrangs findet innerhalb des GUI Elements statt. Der Vorteil dieser Herangehensweise ist die dynamische Erweiterung eines Fensters um einen ganzen Teilgenstrang. Wenn Gene hinzugefügt werden, dann erfolgt die Verarbeitung nicht in der Hauptfensterklasse, sondern geschieht innerhalb der abgeleiteten *QWidget* Klasse. Viele Operationen in der *EvolutionaryPseudo* Klasse beziehen ihre Parameter direkt von dem grafischen *PseudoWidget* Objekt.

[10] *QWidget* ist eine Grafikklasse der Qt Bibliothek. Sie ist die Basisklasse aller grafischen Elemente.

4.2.2 Genfabrik

In der Chromosomstruktur existiert eine Vielzahl von Gentypen. Durch den Einsatz eines Factory Pattern können neue Gentypen modular implementiert werden. Änderungen an den speziellen Gentypen können, ohne Auswirkung auf den restlichen Programmcode, vorgenommen werden. Die Erzeugung eines Gens erfolgt ohne eine Spezifizierung des Gentyps [Ker05]. Der Fabrik wird lediglich ein beobachtendes Genstrangobjekt übergeben. Das Resultat ist ein Gen, das sich im entsprechenden Genstrang oder Teilgenstrang eingeordnet hat.

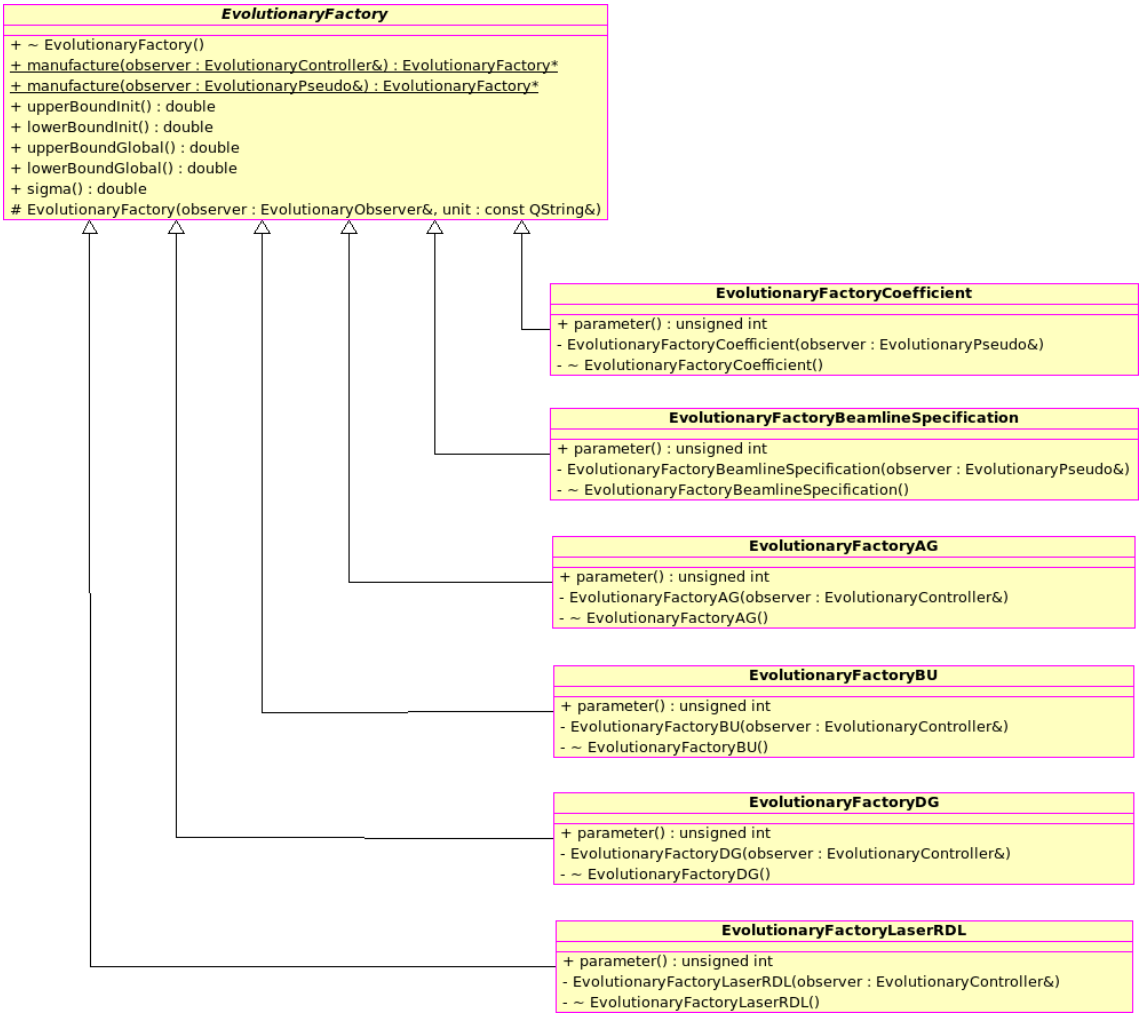


Abbildung 4.3: Klassendiagramm des Factory Pattern

Die abstrakte Fabrik Klasse deklariert und definiert die Geneigenschaften, die vom Benutzer verändert werden können. Jede abgeleitete Klasse spezifiziert einen speziellen Gentyp. Die Abbildung 4.3 zeigt das Klassendiagramm des Factory Pattern. Es sind nur die wichtigsten, spezifischen Gentypen dargestellt. Die Klassen haben alle die Farbe der *EvolutionaryAbsoluteSubject* Klasse, weil jede dieser Klassen ein einzelnes Gen repräsentiert.

```
class EvolutionaryFactory : public EvolutionaryAbsoluteSubject
{
public:
    virtual ~EvolutionaryFactory();

    static EvolutionaryFactory* manufacture(EvolutionaryController& observer)
                                        throw(EvolutionaryFactoryError);

    static EvolutionaryFactory* manufacture(EvolutionaryPseudo& observer)
                                        throw(EvolutionaryFactoryError);

protected:
    EvolutionaryFactory(EvolutionaryObserver& observer, const QString& unit);

}; // EvolutionaryFactory
```

Quellcode 4.16: *EvolutionaryFactory* Klasse

Der Konstruktor der *EvolutionaryFactory* Klasse sowie die Konstruktoren der abgeleiteten Klassen sind geschützt. Jedoch ist die *EvolutionaryFactory* Fabrikklasse in den abgeleiteten Klassen als *friend* definiert. Das heißt, dass nur die *EvolutionaryFactory* Klasse eine Instanz aller von ihr abgeleiteten Klassen erzeugen kann. Die Fabrikklasse besitzt zwei statische *manufacture()* Funktionen. Diese Funktionen wird entweder ein beobachtender Genstrang vom Typ *EvolutionaryController* oder ein Teilgenstrang vom Typ *EvolutionaryPseudo* übergeben. Der Rückgabewert ist ein Zeiger auf das produzierte Genobjekt.

Der Destruktor aller Klassen der Hirarchie ist von außen erreichbar, damit jedes Objekt auch ohne die Fabrikklasse zerstört werden kann. Das ist ein grosser und wesentlicher Nachteil des Factory Pattern, weil keine Kontrolle über die bestehenden Objekte existiert. Wenn ein Zeiger auf eines dieser Objekte verloren geht oder nicht gespeichert wird, dann entstehen sogenannte *resource leaks*. In Abschnitt 4.2.4 wird die implementierte Vermeidung solcher Fehler näher beschrieben.

```
EvolutionaryFactory* EvolutionaryFactory::manufacture(
                                EvolutionaryController& observer)
                                throw(EvolutionaryFactoryError)
{
    // pointer to the product
    EvolutionaryFactory* product(NULL);
    // manufacture the specified product
    switch( observer.type() )
    {
    case Controller::AG33220:
        product = new EvolutionaryFactoryAG(observer);
        break;
    case Controller::BU1708:
        product = new EvolutionaryFactoryBU(observer);
        break;
    case Controller::DG3008:
        product = new EvolutionaryFactoryDG(observer);
        break;
    case Controller::LaserRDLControl:
        product = new EvolutionaryFactoryLaserRDL(observer);
        break;
    ...
    }
    return product;
}
```

Quellcode 4.17: Genproduktion der *EvolutionaryFactory* Klasse

Der Typ des Genstrangs ist entscheidet für das Erscheinungsbild eines Gens. Jedes Gen ist ein *EvolutionaryAbsoluteSubject* und bekommt einen beobachtenden Genstrang im Konstruktor übergeben. Die überladene *manufacture()* Funktion für die *EvolutionaryPseudo* Klasse verhält sich ähnlich, jedoch wird dort nicht nach dem Typ, sondern nach der Art unterschieden.

4.2.3 Pseudotypen

Wie bereits in Abschnitt 4.2.1.5 beschrieben ist ein Pseudotyp ein Teilgenstrang eines *Controllers*. Der Sinn einer solchen Teilgenstrangstruktur ist die Reduktion der zu optimierenden Parameter. Eine Optimierung eines kleineren Parameterraumes hat eine gewaltige Zeitersparnis zur Folge, da die optimalen Parameter durch den evolutionären Algorithmus viel schneller gefunden werden können.

4.2.3.1 Polynom

Die erste Herangehensweise der Parameterreduktion ist die Beschreibung eines Werteverlaufs durch ein Polynom. Ein zyklisches Signal, wie es z.B. von einem Rechteckgenerator erzeugt wird, kann durch ein Polynom 1.Ordnung dargestellt werden.

$$f(t) = c_1 \cdot t + c_0 \qquad \text{für } t \in \mathbb{N} \tag{4.1}$$

Die Funktion 1.Ordnung enthält zwei Optimierungsparameter: c_0 und c_1. Will man einen Signalverlauf der gleichen Funktion $f(t)$ mit absoluten Parametern beschreiben, so braucht man alle möglichen Werte von t des Definitionsbereichs.

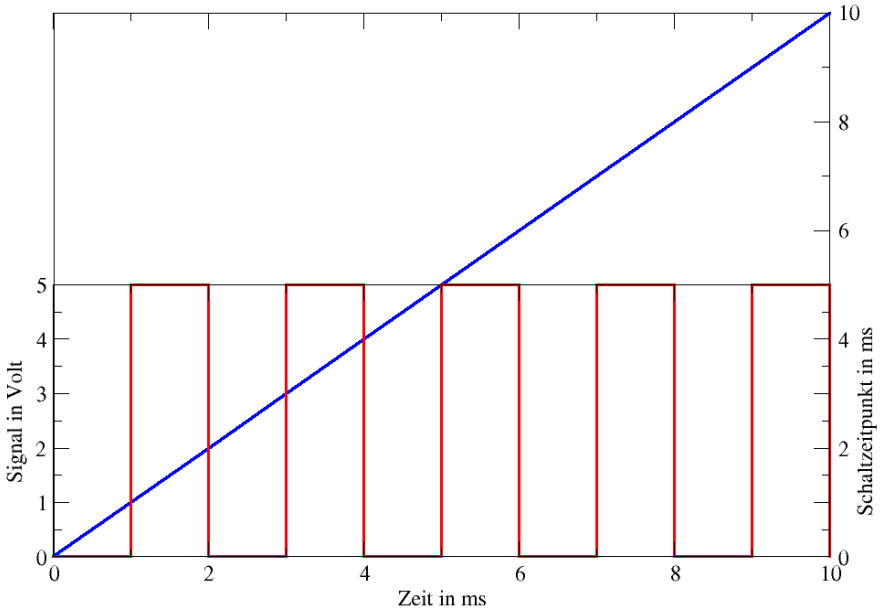


Abbildung 4.4: Polynomialer Verlauf eines Signals

Um den rot dargestellten Signalverlauf in Abbildung 4.4 mit absoluten Parametern zu beschreiben, braucht man zehn Optimierungsparameter. Die blau dargestellte Gerade zeigt den polynomialen Verlauf mit zwei Koeffizienten.

```
double EvolutionaryPolynomial::calculate(const unsigned int& value) const
{
    double result(0);
    for(int i=0; i < _subjects.size(); ++i) {
        if(i == 0)
            result += _subjects.at(i)->value();
        else
            result += _subjects.at(i)->value()
                    * pow(static_cast<double>(value), i);
    }
    return result;
}
```

Quellcode 4.18: Berechnung der *EvolutionaryPolynomial* Klasse

Die *EvolutionaryPolynomial* Klasse implementiert die abstrakt deklarierte Funktion *calculate()* der Basisklasse. Die Gene sind die Koeffizienten. Die Ordnung des Koeffizienten entspricht dem Index der Subjekte. Die Funktion *calculate()* berechnet den Wert eines Parameters, anhand der für den Teilgenstrang registrierten Gene.

4.2.3.2 Beamline Calculator

Die Klasse *EvolutionaryBeamlineCalculator* ist ein Teilgenstrang zur Berechnung einer Burstsequenz. Die Gene dieses Teilgenstrangs repräsentieren die zu optimierenden Parameter einer Konfigurationsdatei. Zur Berechnung der Burstsequenz wird das Programm *coldmol_timesequence*[11] mit einer generierten Konfigurationsdatei ausgeführt. Jede Konfigurationsdatei enthält unterschiedliche Optionen bzw. Variablen. Deswegen muss eine Templatedatei vorher definiert werden, aus der die resultierende Konfigurationsdatei generiert wird. Die Datei muss gewisse Schlüsselwörter, die durch die Optimierungsparameter ersetzt werden, enthalten:

Das fundamentale Makro - Schlüsselwort:

- **PARAMETER_BURSTFILE:** Das Schlüsselwort ist in jeder Templatedatei zwingend notwendig. Es wird durch den temporären Konfigurationsdateinamen ersetzt. Ohne dieses Schlüsselwort kann das erzeugte Burstfile nicht lokalisiert werden.

[11] Das *coldmol_timesequence* Programm ist eine Konsolenanwendung, die von der Projektgruppe "Deceleration and trapping of large (bio-)molecules" entwickelt wurde. Das Programm gehört zu der "coldmol" Bibliothek und wird zur Berechnung der "beamline" in der Abbremsstufe verwendet. Das Resultat der Berechnung ist ein *Burstfiles* mit den errechneten Zeitsequenzen.

Die optionalen Makro - Schlüsselwörter:

- **PARAMETER_PHI:** Zur Optimierung des Phasenwinkel ϕ der Schaltmomente beim Stark Decelerators.
- **PARAMETER_V_:** Zur Optimierung der Anfangsgeschwindigkeit v eines Molekülpackets.
- **PARAMETER_D0:** Zur Optimierung des Mittelpunkts $d0$ der Abbremsstrecke beim Alternate Gradient Decelerators.
- **PARAMETER_D_:** Zur Optimierung der Länge d der Abbremsstrecke beim Alternate Gradient Decelerators.
- **PARAMETER_F0:** Zur Optimierung des Mittelpunkts $f0$ der Fokussierstrecke beim Alternate Gradient Decelerators.
- **PARAMETER_F_:** Zur Optimierung der Länge f der Fokussierstrecke beim Alternate Gradient Decelerators.

Der wesentliche Vorteil dieser Optimierung ist die starke Reduktion der zu optimierenden Parameter. Oft soll zur Signalverbesserung die Intensität der Moleküle gesteigert werden. Dazu muss die Länge der Fokussierstrecke variieren. Durch die Optimierung von f wird somit eine Burstsequenz erstellt, bei der indirekt, je nach Schaltsystem, alle Schaltzeitpunkte verändert werden.

Die Werte von f werden durch ein Polynom beschrieben, da sich die Länge der Fokussierstrecke mit der Geschwindigkeit des Moleküls verändert.

$$f(i) = f + i \cdot f' + i^2 \cdot f'' \qquad \text{für } i \in \mathbb{N} \tag{4.2}$$

Um eine optimale Fokussierung zu erzielen, müssen f, f' und f'' optimiert werden. Es ist jedoch nicht bei jedem Experiment zwangsläufig notwendig alle f Parameter zu optimieren. Wenn einer dieser f Parameter zur Optimierung nicht freigegeben wurde, wird dieser auf 0 gesetzt.

$$f(i) = f \qquad \text{für } f \in \mathbb{R};\ f' = 0\ ;\ f'' = 0 \tag{4.3}$$

$$f(i) = i \cdot f' \qquad \text{für } f' \in \mathbb{R}; i \neq 0;\ f = 0\ ;\ f'' = 0 \tag{4.4}$$

Die *EvolutionaryBeamlineCalculator* Optimierung von $f(i)$ besitzt noch eine weitere Eigenschaft, die vom Benutzer aktiviert werden kann. Das *PseudoWidget* enthält eine *QCheckBox* mit der negative Werte von $f(i)$ verhindert werden können.

```
QString EvolutionaryBeamlineCalculator::calculate(const unsigned int& i,
                                                  const double& f,
                                                  const double& firstDerivative,
                                                  const double& secondDerivative,
                                                  bool negativeAllowed ) const
{
    QString str("1*");
    const double value( f + (i * firstDerivative)
                          + (pow( static_cast<double>(i), 2.)
                            * secondDerivative) );
    if( !negativeAllowed && (value < 0) )
        str.append("0.0");
    else
        str.append( QString().setNum( value ) );
    return str;
}
```

Quellcode 4.19: Berechnung der *EvolutionaryBeamlineCalculator* Klasse

Die Funktion *calculate()* berechnet die Werte von $f(i)$ und schreibt diese in eine Zeichenkette. Die erzeugte Zeichenkette wird als Resultat der Berechnung zurückgegeben und in die Konfigurationsdatei eingetragen.

Das erleuterte Optimierungsbeispiel der Länge der Fokussierstrecke $f(i)$ entspricht ebenso der Optimierung der Länge der Abbremsstrecke $d(i)$.

4.2.4 Gencontainer (Chromosom)

Der Gencontainer repräsentiert ein Chromosom und ist die eigentliche Schnittstelle der gesamten Ebene. Sie bietet alle Funktionen um Gene zu produzieren, zu zerstören, Genstränge abzufragen und die Gene zu indizieren. Sie berechnet bzw. verwaltet den Locus der Gene und steuert das Aussehen sowie das Verhalten aller Gene, Teilgenstränge und Genstränge.

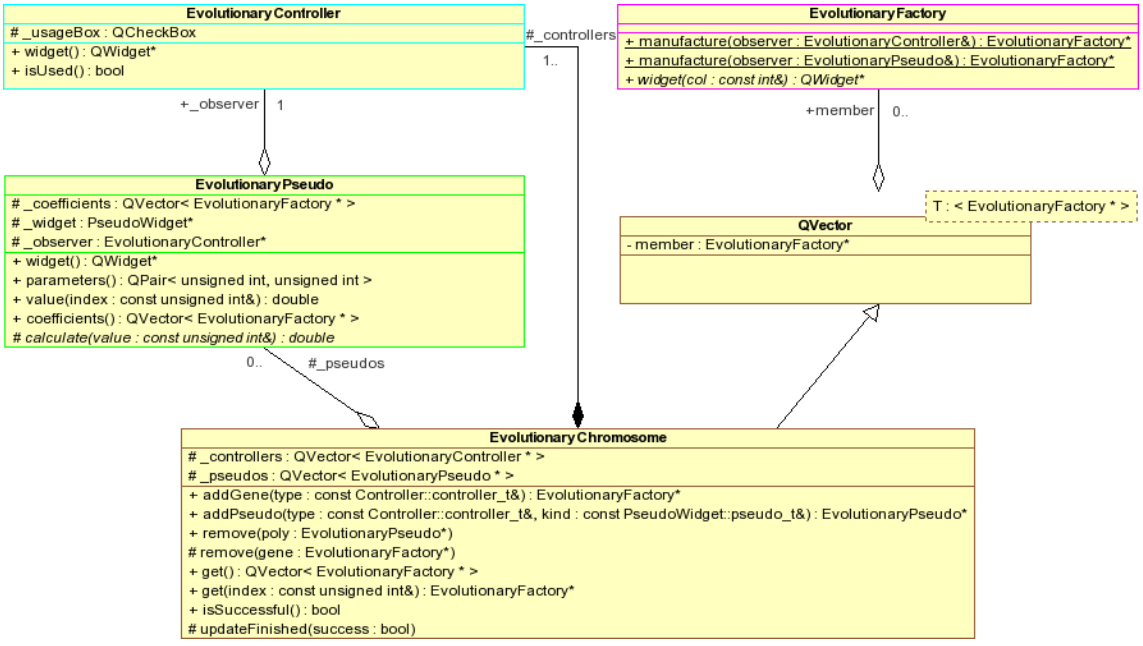


Abbildung 4.5: Klassendiagramm des Gencontainer (Chromosom)

```
class EvolutionaryChromosome : protected QObject,
                               protected QVector<EvolutionaryFactory *>
{
    friend class EvolutionaryComputingParameters;

private:
    EvolutionaryChromosome();

    virtual ~EvolutionaryChromosome();
```

Quellcode 4.20: Instantiierung der *EvolutionaryChromosome* Klasse

Der Konstruktor als auch der Destruktor der *EvolutionaryChromosome* Klasse sind von außen nicht zu erreichen. Allerdings ist die Klasse als *friend* in der *EvolutionaryComputingParameters* Klasse definierte. Die *EvolutionaryComputingParameters* Klasse ist als Singleton Pattern realisiert und wird in Abschnitt 4.3.2 beschrieben. Indirekt ist somit auch die *EvolutionaryChromosome* Klasse ein Singleton, weil nur eine einzige Instanz von ihr erzeugt werden kann.

```
public:
    EvolutionaryFactory* addGene( const Controller::controller_t& type)
                          throw( EvolutionaryChromosomeError);

    EvolutionaryPseudo* addPseudo(const Controller::controller_t& type,
                                  const PseudoWidget::pseudo_t& kind)
                          throw( EvolutionaryChromosomeError);

public slots:
    void remove(QWidget *widget);

    void remove(EvolutionaryPseudo *poly);


protected:
    QVector<EvolutionaryController *> _controllers;

    QVector<EvolutionaryPseudo *> _pseudos;
```

Quellcode 4.21: Genverwaltung der *EvolutionaryChromosome* Klasse

Die *EvolutionaryChromosome* Klasse ist ein *QVector* aus Genen. Sie erbt allerdings nur *protected* die Eigenschaften der Templateklasse, damit der Zugriff auf die Gene von außen nicht möglich ist. Um ein Gen hinzuzufügen oder zu entfernen, muss eine der *add()* oder *remove()* Funktionen benutzt werden.

Die zwei geschützten Attribute *_controllers* und *_pseudos* vom Typ *QVector* speichern die Genstränge und Teilgenstränge. Alle Gene, die in dem Klassenvektor gespeichert werden, erhalten einen *EvolutionaryController* als beobachtenden Genstrang. Die Teilgenstränge, die vom Typ *EvolutionaryPseudo* sind, speichern und verwalten innerhalb ihrer Instanz die ihnen angehörenden Gene.

Die *add()* Funktion nutzt die statische *manufacture()* Schnittstelle der Genfabrik und speichert die produzierten Gene in dem Klassenvektor. Es wird der Genstrang des entsprechenden Typs lokalisiert und an die Fabrik weitergereicht. Die Teilgenstränge besitzen ihre eigenen *add()* und *remove()* Funktionen, um die Gene mit der Fabrik zu produzieren. Da die Produktion und Destruktion nur über die definierten Funktionsschnittstellen veranlasst werden kann, ist die Problematik der *resource leaks*, wie sie in Abschnitt 4.2.2 erwähnt wurden, gelöst.

```
EvolutionaryFactory* EvolutionaryChromosome::get(const unsigned int& index)
                                        const throw(EvolutionaryChromosomeError)
{
   unsigned int idx(0);
   // search in myself (absolute genes)
   for(EvolutionaryChromosome::ConstIterator iter =  this->constBegin();
           iter != this->constEnd(); ++iter)
   {
      if( (*iter)->isUsed())
         if(idx == index)
            return (*iter);
         else
            ++idx;
   }
   // search in pseudos
   for(QVector<EvolutionaryPseudo*>::ConstIterator
           iter  = _pseudos.constBegin();
           iter != _pseudos.constEnd(); ++iter)
   {
      if( (*iter)->isUsed())
      {
         // get the pseudo coefficients
         QVector<EvolutionaryFactory*> coefficients((*iter)->coefficients());
         // search after it
         for(QVector<EvolutionaryFactory*>::ConstIterator
                 cIter = coefficients.constBegin();
                 cIter != coefficients.constEnd(); ++cIter)
         {
            if(idx == index)
               return (*cIter);
            else
               ++idx;
         }
      }
   }
   return NULL;
}
```

Quellcode 4.22: Locusbestimmung der *EvolutionaryChromosome* Klasse

Die *get()* Funktion gibt ein Gen für den als Parameter übergebenen Locus zurück. Es werden alle Gene der Genstränge und Teilgenstränge nach dem Gen mit dem gesuchten Locus durchsucht. Der Locus jedes Gens wird bei jedem Aufruf neu berechnet. Da die Gene ihren Index innerhalb der Vektoren nicht verändern, bleibt der Locus erhalten. Nur Gene eines aktiven Genstrangs besitzten einen Locus und werden bei der Lokalisierung berücksichtigt.

```
void EvolutionaryChromosome::writeGenes(Settings& setting)
{
    setting.beginWriteArray("Genes");
    // write the absolute genes
    for (int i = 0; i < QVector<EvolutionaryFactory *>::size(); ++i) {
        setting.setArrayIndex(i);
        // write the controller type to settings
        setting.setValue("type", this->at(i)->type() );
        // let the gene write its settings
        this->at(i)->write( setting );
    }
    setting.endArray();
}
```

Quellcode 4.23: Rekursivverhalten der *EvolutionaryChromosome* Klasse

Die *EvolutionaryChromosome* Klasse stellt eine Menge an Funktionen, die sich auf die Gene auswirkt, zur Verfügung. Das Rekursivverhalten der Chromosomstruktur wird an einem Beispiel, das in Quellcode 4.23 gezeigt wird, beschrieben. Die Funktion *writeGenes()* speichert alle Einstellungen der Gene in einer Konfigurationsdatei. Die Klasse *Settings*[12] erleichtert die Speicherung durch entsprechende Funktionalität. Jedes Gen wird einzelnd angesprochen, um seine genspezifische Speicherung der Einstellungen zu veranlassen. Das Gleiche geschieht für die Teilgenstränge, die den Speicherungsprozess an ihre Gene weiterleiten.

4.3 *Evolving Object* Bibliothek (EO)

Die *Evolving Object*[13] (EO) Templatebibliothek implementiert die evolutionären Algorithmen. Es wird zur Implementierung der *Feedback Control* Optimierung verwendet.

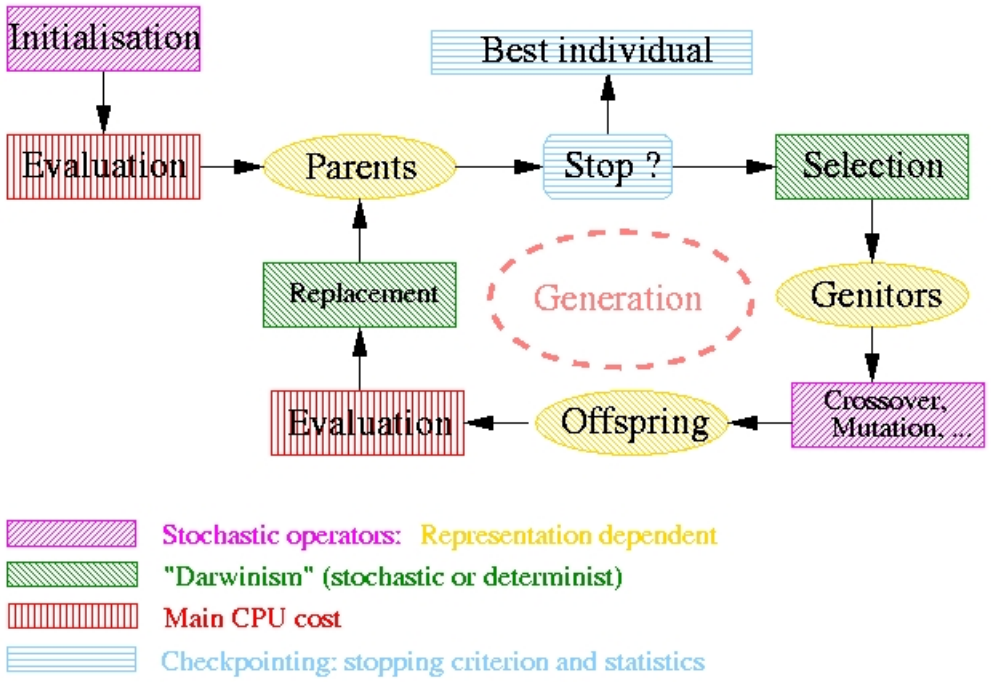


Abbildung 4.6: Schematischer Ablauf des “Evolutionären Algorithmus”

Der schematische Ablauf des evolutionären Algorithmus ist im Bild 4.6 [KMRS01] dargestellt. Der zeitraubenste Teil ist die Berechnung der Fitness. Dazu muss ein Funktor der Templateklasse *eoEvalFunc<EOT>* implementiert werden. Der hellblau gekennzeichnete Teil prüft das Abbruchkriterium und arbeitet mit der berechneten Population die registrierten Checkpoints ab. Ein Checkpoint ist eine Templateklasse vom Typ *eoUpdater*, die ebenso als Funktor implementiert werden muss. Jede dieser Klassen wird vor Beginn der evolutionären Algorithmusprozedur instanziiert.

[12] Die *Settings* Klasse ist eine *QSettings* Klasse der Qt Bibliothek. Sie erleichtert die Speicherung von Einstellungen.

[13] Die *Evolving Object* (EO) Templatebibliothek ist ein kostenloses Softwarepaket, das im Internet heruntergeladen werden kann. Es wurde unter der Leitung von Juan Julián Merelo und dem Team Geneura an der University of Granada entwickelt [toE].

4.3.1 Evolutionäres Threadobjekt

Das evolutionäre Threadobjekt ist eine abstrakte Klassendeklaration, die die Basisklasse des *Feedback Control* Kerns ist. Sie ist ein *QThread*, dessen Bedeutung in Abschnitt 4.4.2 deteilierter erleutert wird.

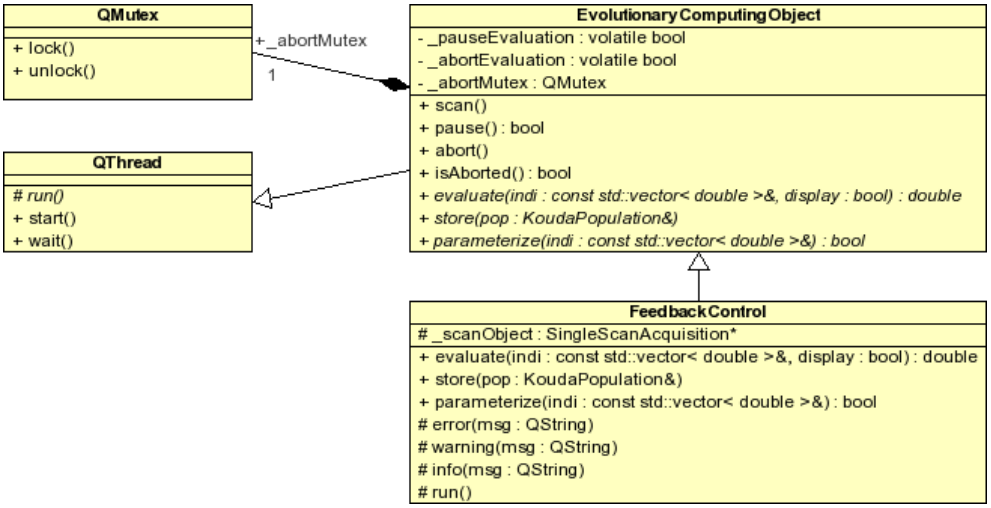


Abbildung 4.7: Klassendiagramm der Hauptthreadhirarchie

```
class EvolutionaryComputingObject : protected QThread
{
public:
    typedef eoEsSimple<double> Individual_one_Sigma;

    typedef eoEsStdev<double> Individual_n_Sigma;

    typedef eoEsFull<double> Individual_CMA;
```

Quellcode 4.24: Algorithmusdefinitionen der *EvolutionaryComputingObject* Klasse

Es werden drei Typen definiert, die die verfügbaren Algorithmen darstellen. Mit ihnen wird die *compute()* Templatefunktion der *EvolutionaryComputingEvolvingObject* Klasse spezifiziert. Die Definition erfolgt hier in der Berechnungs - Ebene, weil die abgeleitete Klasse in der *Feedback Control* - Ebene implementiert ist und dort keine Elemente der EO Bibliothek verwendet werden.

```
public:
    bool isAborted()
    {
        QMutexLocker locker(&_abortMutex);
        return _abortEvaluation;
    };

private:
    volatile bool _pauseEvaluation;

    volatile bool _abortEvaluation;

    QMutex _abortMutex;

}; // class EvolutionaryComputingObject
```

Quellcode 4.25: Threadattribute der *EvolutionaryComputingObject* Klasse

Die *EvolutionaryComputingObject* Klasse besitzt drei wesentliche Threadattribute. Die zwei *_pauseEvaluation* und *_abortEvaluation* Attribute sind vom Typ *volatile bool*. Das sind Zustandsvariablen die bei jedem Zugriff neu aus dem Speicher geladen werden und ein Abbruch- oder Pausezustand signalisieren. Sie werden durch ein drittes Synchronisationsobjekt, vom Typ *QMutex*[14] geschützt. Das Mutex ist ein Threadsynchronisationsobjekt und kann nur die zwei Zustände "blockierend" und "nicht blockierend" annehmen. Alle drei Attribute sind *private* und somit nicht einmal für eine abgeleitete Klasse erreichbar. Deswegen muss der Abbruchzustand durch eine *isAborted()* Funktion abgefragt werden. Das geschieht zyklisch in der *run()* Threadfunktion, um den Thread sauber zu beenden.

```
void EvolutionaryComputingObject::scan()
{
    // clear the abort flag
    QMutexLocker locker(&_abortMutex);
    _abortEvaluation = false;
    // start the thread
    this->start(QThread::TimeCriticalPriority);
}
```

Quellcode 4.26: Threadstart der *EvolutionaryComputingObject* Klasse

Die *scan()* Funktion startet den Thread. Dazu muss die Abbruchzustandsvariable zurück gesetzt werden. Die *QMutexLocker* Klasse wird von der Qt Bibiothek zur Verfügung gestellt. Sie macht die Benutzung des Mutex sicherer [BS07]. Im Konstruktor der *QMutexLocker* Klasse wird das Mutex in den Zustand "blockierend" versetzt und im Destruktor wieder zurückgesetzt. Das Mutex ist während der gesamten Funktionsabarbeitung von dem *locker* Objekt blockiert. Es kann bedenkungslos auf die Zustandsvariablen zugegriffen und der Thread gestartet werden.

```
void EvolutionaryComputingObject::abort()
{
    { // set the abort flag
        QMutexLocker locker(&_abortMutex);
        _abortEvaluation = true;
    }
    // wait for the threads end
    this->wait();
}
```

Quellcode 4.27: Threadabbruch der *EvolutionaryComputingObject* Klasse

Die *abort()* Funktion initiiert einen Threadabbruch durch setzten des Abbruchzustands. Das Mutex muss dafür blockiert werden. Durch die geschweiften Klammern beschränkt sich die Lebensdauer des *locker* Objekt vom Typ *QMutexLocker* auf den begrenzten Bereich. Anschließend wartet der aufrufende Thread bis die *run()* Threadfunktion verlassen wird.

[14] *QMutex* ist eine Threadsynchonisationsklasse der Qt Bibliothek. Es ist die objektorientierte Implementierung eines Mutex.

```
bool EvolutionaryComputingObject::pause()
{
    // check the abort flag
    if(_pauseEvaluation) {
        // resume run
        _pauseEvaluation = false;
        _abortMutex.unlock();
    } else {
        // take a break
        _pauseEvaluation = true;
        _abortMutex.lock();
    }
    // return pause state
    return _pauseEvaluation;
}
```

Quellcode 4.28: Verarbeitungspause der *EvolutionaryComputingObject* Klasse

Die *pause()* Funktion hält den laufenden Thread an und legt ihn solange schlafen bis die *pause()* Funktion ein weiteres mal aufgerufen wird. Die Pausezustandsvariable wird nur in dieser einzigen Funktion benutzt und muss deswegen auch nicht vor mehreren Threadzugriffen geschützt werden. Je nach Zustand des Pauseattributs, wird das Mutex in den "blockierenden" oder "nicht blockierenden" Zustand versetzt. Da das Abbruchflag durch das Mutex geschützt und zyklisch in der *run()* Threadfunktion abgefragt wird, bleibt der Thread solange blockiert bis das Mutex von der *pause()* Funktion wieder freigegeben wird. Die Zugriffe auf das *QMutex* erfolgen durch Klassenfunktionen ohne eine Behelfsklasse.

4.3.2 Implementierung der Parseroptionen

EO benutzt eine Parserdatei, die die relevanten Einstellungen der Evolutionsstrategie enthält. Zur Einlesung der Parseroptionen verwendet EO eine eigene Klasse namens *eoParser*. Es ist unumgänglich diese Klasse zu benutzten, da sie zur Initialisierung und Instantiierung von vielen EO Objekten benötigt wird.

In der *Feedback Control* werden die meisten Optionen dem Benutzer zur Änderung in der grafischen Oberfläche angeboten, somit ist die Erstellung einer Parserdatei von Hand mit einem Texteditor überflüssig. Die *EvolutionaryComputingParameters* Klasse ist eine *Entryklasse*, die die Informationen hält, speichert und daraus die *eoParser* Klasse erzeugt. Diese *Entryklasse* ist als Singleton Pattern implementiert, weil es nur ein einziges Objekt geben darf. Die Optionen werden durch einfache *get* und *set* Funktionen des Parameternamens zur Verfügung gestellt.

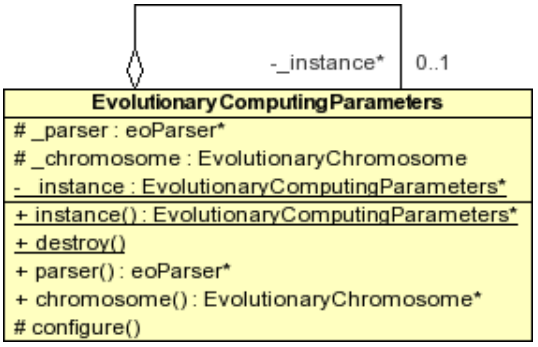


Abbildung 4.8: Klassendiagramm des Parser Singleton

```
class EvolutionaryComputingParameters
{
public:
    static EvolutionaryComputingParameters* instance() throw()
    {
        if(_instance == NULL)
            _instance = new EvolutionaryComputingParameters();
        return _instance;
    };

    static void destroy() throw()
    {
        if(_instance != NULL) {
            delete _instance;
            _instance = NULL;
        }
    };

private:
    EvolutionaryComputingParameters(const std::string& name =
                                    std::string("KouDA_Feedback_Control") );

    virtual ~EvolutionaryComputingParameters();

    EvolutionaryComputingParameters(const EvolutionaryComputingParameters&);

    EvolutionaryComputingParameters& operator=
                                        (EvolutionaryComputingParameters&);

    static EvolutionaryComputingParameters* _instance;
```

Quellcode 4.29: Singletoninstanz der *EvolutionaryComputingParameters* Klasse

Der Konstruktor, Kopierkonstruktor, Zuweisungskonstruktor und der Destruktor sind geschützte Elemente der *EvolutionaryComputingParameters* Klasse. Ein statischer Zeiger auf die Instanz hält und sichert die Existenz des Objekts. Die statische *instance()* Funktion gibt diesen Zeiger zurück oder erzeugt die Instanz, falls sie noch nicht existiert. Bei dieser Implementierung des Singleton Pattern ist es wichtig die Funktion *destroy()* aufzurufen, wenn das Objekt nicht weiter benötigt wird. Ohne den Aufruf von *destroy()* wird der Destruktor nicht ausgeführt und die Instanz bleibt erhalten. In der *EvolutionaryComputingParameters* Klasse sorgt der Destruktor für die Speicherung der Optionen.

```
protected:
    EvolutionaryChromosome _chromosome;

} // class EvolutionaryComputingParameters
```

Quellcode 4.30: Chromosominstanz der *EvolutionaryComputingParameters* Klasse

Das geschützte Attribut der Klasse *EvolutionaryChromosome* wird mit der Instantiierung erzeugt und hat die gleiche Lebensdauer wie das Singletonobjekt selbst. Wie in Absatz 4.2.4 bereits beschrieben, ist der Konstruktor und Destruktor des Chromosoms für die Speicherung und Wiederherstellung der Genstruktur verantwortlich. Ein weiterer Grund ist die Abhängigkeit zur GUI-Ebene. Da Elemente der Gene an die grafische Oberfläche weitergereicht worden sind, müssen diese auch vor der Zerstörung der Oberfläche, beseitigt werden. Die Lösung ist der Aufruf der *destroy()* Funktion im Destruktor der *DialogFeedbackControl* Klasse in der GUI-Ebene.

```
void EvolutionaryComputingParameters::configure()
     throw(EvolutionaryComputingError, EvolutionaryComputingWarning)
{
    // reset the parser
    delete _parser;
    _parser = new eoParser(_argc, _argv,
                           "KouDA feedback control optimization");

    // used local vars
    vector<double> lower_initBounds;
    vector<double> upper_initBounds;
    vector<double> init_sigmas;
    vector<double> lower_objectBounds;
    vector<double> upper_objectBounds;

    // create individual options
    QVector<EvolutionaryFactory*> chromosome = _chromosome.get();
    for(QVector<EvolutionaryFactory*>::Iterator iter=chromosome.begin();
        iter != chromosome.end(); iter++)
    {
        lower_initBounds.push_back(    (*iter)->lowerBoundInit()    );
        upper_initBounds.push_back(    (*iter)->upperBoundInit()    );
        if( _algorithm == uncorrelatedMutation_one_sigma )
            init_sigmas.push_back(     _objectSigma                 );
        else
            init_sigmas.push_back(     (*iter)->sigma()             );
        lower_objectBounds.push_back( (*iter)->lowerBoundGlobal() );
        upper_objectBounds.push_back( (*iter)->upperBoundGlobal() );
    }

    // set parser options
    ...
}
```

Quellcode 4.31: Parsererstellung der *EvolutionaryComputingParameters* Klasse

Das Parserobjekt wird für jede evolutionäre Algorithmusprozedur neu erstellt, damit die bereits existierenden Parseroptionen gelöscht werden. Anschließend werden für die zu optimierenden Parameter temporäre Vektoren erzeugt, die mit den existierenden Genparameter gefüllt werden. In der Schleife aus Quellcode 4.31 erhalten die Gene zum ersten Mal von der Chromosomstruktur ihren Locus (Abschnitt 4.2.4), der aus Sicht der Berechnungs - Ebene bis Ende des evolutionären Algorithmuns bestehen bleibt.

```
    _parser->setORcreateParam( unsigned(_maxGeneration), "maxGen",
                               "Maximum number of generations () = none)",
                               'G', "Stopping criterion");
```

Quellcode 4.32: Parseroptionen der *EvolutionaryComputingParameters* Klasse

Im Quellcode 4.32 wird schematisch an Hand eines Beispiels gezeigt, wie z.B. die maximale Anzahl an Generationen als Abbruchkriterium in dem Parserobjekt gesetzt bzw. erzeugt wird. Einige der Parseroptionen sind als Konstanten im Programm verankert. Die restlichen Parameter sind vom Benutzer gewählte Einstellungen. Die ausführliche Erzeugung und das resultierende Ergebnis der Parsereinstellungen ist im Anhang C einzusehen.

4.3.3 Fitnessfunktion (Evaluator)

Zur Bestimmung der Fitness muss eine Templateklasse von Typ *eoEvalFunc<EOT>* implementiert werden. Diese Klasse ist ein Funktionsobjekt oder Funktor. Sie wird genauso aufgerufen wie eine Funktion und hat die Eigenschaften eines Objekts. Es ist lediglich die Operatorfunktion *operator()* überladen. Die Fitnessfunktion gehört zu der Kategorie der Unären Funktionen mit einem Funktionsparameter *f(x)*.

```
template <typename EOT>
class EvolutionaryComputingEvolvingObject::eoKoudaFunc
                                        : public eoEvalFunc<EOT>
{
public:
    eoKoudaFunc(const EvolutionaryComputingEvolvingObject* eval)
        : eoEvalFunc<EOT>(), _eval(eval)
    { };

    virtual void operator()(EOT& _eo)
    {
        // evaluate
        if (_eo.invalid())
            _eo.fitness( _eval->evaluate( _eo, true) );
    };

private:
    const EvolutionaryComputingEvolvingObject* _eval;

}; // template class eoKoudaFunc
```

Quellcode 4.33: *eoKoudaFunc* Funktor

Der Konstruktor erhält bei der Instantiierung einen konstanten Zeiger auf ein *EvolutionaryComputingEvolvingObject* Objekt. Weil die *eoKoudaFunc* Templateklasse eine geschachtelte Klasse in der *EvolutionaryComputingEvolvingObject* Klasse ist, hat diese über den Zeiger auch vollen Zugriff auf die geschützten Elemente der übergeordneten Klasse. Somit ruft die Operatorfunktion, die ein Individuum übergeben bekommt, auch die geschützte *evaluate()* Funktion auf. Mit dem Aufruf der Funktion wird die Fitness eines Individuums ermittelt und dem Individuum zugewiesen.

```
template <typename EOT>
double EvolutionaryComputingEvolvingObject::evaluate(const EOT& indi,
                                                     bool display) const
{
    return _evalObject->evaluate(indi, display);
}
```

Quellcode 4.34: Adaption der *EvolutionaryComputingEvolvingObject* Klasse

Die geschützte *evaluate()* Templatefunktion adaptiert ein Individuum eines bestimmten Algorithmus in einen STL *std::vector<double>*. Die detailiertere Beschreibung der Berechnung der Fitness eines Individuums erfolgt in Abschnitt 4.4.2.3.

4.3.4 Aktualisierung (Updater)

Der Aktualisierungsfunktor *(Updater)* gehört zu der Kategorie der Generatoren ohne Funktionsparameter *f()*. Jeder Aktualisierungsfunktor muss von *eoUpdater* erben, um als Checkpoint registriert zu werden.

4.3.4.1 Reevaluation

Der *Reevaluation* Funktor berechnet alle Individuen einer Population neu.

```
template <typename EOT>
class EvolutionaryComputingEvolvingObject::eoKoudaUpdater : public eoUpdater
{
public:
    eoKoudaUpdater(const EvolutionaryComputingEvolvingObject* eval,
                   eoPop<EOT>& pop, unsigned gen=1)
        : eoUpdater(), _popCounter(0), _gen(gen), _pop(pop), _eval(eval)
    { };

    virtual void operator()()
    {
        if( (0 != _gen) && (0 == (++_popCounter % _gen)) )
            for(unsigned int i = 0; i < _pop.size(); ++i)
                _pop[i].fitness( _eval->evaluate(_pop[i], true) );
    };

protected:
    unsigned int _popCounter;

    const unsigned int _gen;

    eoPop<EOT> &_pop;

private:
    const EvolutionaryComputingEvolvingObject* _eval;

}; // template class eoKoudaUpdater
```

Quellcode 4.35: *eoKoudaUpdater* Funktor

Die Instantiierung der Objekte erfolgt analog zur Fitnessfunktion. Dem Konstruktor, wird ebenso als ersten Parameter ein konstanter Zeiger auf ein *EvolutionaryComputingEvolvingObject* Objekt, übergeben. Der zweite Parameter ist eine Instanz der berechneten Generation, die eine Aktualisierung erfordert.

4.3.4.2 Speichern der Parameter

Der *eoKoudaSaver* Funktor speichert jede Generation auf einem Speichermedium ab. Der Konstruktor und die Klassenattribute der *eoKoudaUpdater* Templateklasse, die im Quellcode 4.35 dargestellt sind, werden im weiteren Verlauf nicht nochmals gezeigt oder erklärt, weil sie in allen *Updater* Funktoren sehr ähnlich sind.

```
template <typename EOT>
class EvolutionaryComputingEvolvingObject::eoKoudaSaver : public eoUpdater
{
public:
    virtual void operator()()
    {
        _eval->store(_popCounter, _pop);
        ++_popCounter;
    };

}; // template class eoKoudaSaver
```

Quellcode 4.36: *eoKoudaSaver* Funktor

Die Operatorfunktion *operator()()* übergibt die gesamte Population der *store()* Funktion des *EvolutionaryComputingEvolvingObject* Objekts. Es wird für jede Population ein Zähler erhöht, der die Anzahl der Populationen zählt. Die *store()* Funktion versetzt die beobachtenden Genstränge in den speichernden Zustand. Anschließend wird jedem Gen der Chromosomstruktur ein neues Allel zugewiesen. Die Speicherung der Daten und Parameter erfolgt, wie in Abschnitt 4.2.1.2 bereits erleutert, in seperaten Threads.

4.3.4.3 Speichern des Populationsverlaufs

Der *eoKoudaDiagram* Funktor speichert die Fitness, sowie die dazu gehörigen Mutationsschrittweiten σ jedes Gens einer Population. Dieser *Updater* Funktor dient der Auswertung der evolutionären Algorithmusprozedur. Durch eine entsprechende Speicherung der Daten, kann die erzeugte Datei in xmGrace [14] importiert und analysiert werden.

```
template <typename EOT>
class EvolutionaryComputingEvolvingObject::eoKoudaDiagram : public eoUpdater
{
public:
    virtual void operator()()
    {
        ofstream fitFile( string(_dir + string("/")
                          + _fitFile).c_str(), ios::app );
        ofstream indiFile( string(_dir + string("/")
                           + _indiFile).c_str(), ios::app );
        if( fitFile && indiFile ) {
            for(unsigned int i = 0; i < _pop.size(); ++i) {
                fitFile << _indiCounter << "␣␣␣"
                        << _pop[i].fitness() << endl;
                indiFile << _indiCounter << "␣␣␣";
                _eval->diagram( _pop[i], indiFile );
                indiFile << endl;
                ++_indiCounter;
            }
        }
    };

}; // template class eoKoudaDiagram
```

Quellcode 4.37: *eoKoudaDiagram* Funktor

Im Konstruktor wird bei der Instantiierung ein erforderlicher Dateipfad übergeben und in einem Klassenattribut gespeichert. Dieser Pfad wird um einen entsprechenenden Dateinamen erweitert und in der Operatorfunktion *operator()()* verwendet.

[14] XmGrace ist ein Programm, mit dem es möglich ist, Daten grafisch darzustellen. Die Grafiken können durch eine Vielzahl von Optionen editiert werden.

4.3.5 Initialisierung

Die Initialisierung der EO Bibliothek erfolgt in einer Templatefunktion. Der Typ, der die Templatefunktion typirisiert, entscheident über den verwendeten Algorithmus. Die Typisierung bzw. der Aufruf dieser fundamentalen Templatefunktion wird in Abschnitt 4.4.3 erklärt und gehört in die darüberliegende *Feedback Control* Ebene.

```
template <typename EOT>
void EvolutionaryComputingEvolvingObject::compute()
                                                throw(EvolutionaryComputingError,
                                                      EvolutionaryComputingWarning,
                                                      AbortThreadError)
{
    // get the parser object
    eoParser& parser(*EvolutionaryComputingParameters::instance()->parser());
    // keep all things allocated
    eoState state;
    // add the evaluation function
    eoKoudaFunc<EOT> mainEval(this);
    // packaging it into the counter for output
    eoEvalFuncCounter<EOT> eval(mainEval);
    // init the genotype with a genotype initializer
    eoRealInitBounded<EOT>& init = make_genotype(parser, state, EOT());
    // variation operator (any seq/prop construct)
    eoGenOp<EOT>& op = make_op(parser, state, init);
    // create the population
    eoPop<EOT>& pop = make_pop(parser, state, init);
    // create stopping criteria
    eoContinue<EOT>& term = make_continue(parser, state, eval);
    // create the checkpoint object
    eoCheckPoint<EOT>& checkpoint=make_checkpoint(parser, state, eval, term);
    // registry the checkpoint for reevaluation of population
    eoKoudaUpdater<EOT> evaluator(this, pop,
                EvolutionaryComputingParameters::instance()->reevaluation());
    checkpoint.add(evaluator);
    // registry the checkpoints for output and displaying of population
    eoKoudaSaver<EOT> saver(this, pop);
    checkpoint.add(saver);
    // algorithm (need the operator!)
    eoAlgo<EOT>& ga = make_algo_scalar(parser, state, eval, checkpoint, op);
    // evaluate intial population AFTER help and status in case it takes time
    apply<EOT>(eval, pop);
    // run optimization
    run_ea(ga, pop);
}
```

Quellcode 4.38: Initialisierung des evolutionären Algorithmus

Es werden in der *compute()* Templatefunktion alle nötigen EO-Objekte mit der typisierten Algorithmusklasse instanziiert. Die EO Bibiothek stellt einige statische Behelfsfunktionen zur Verfügung. Sie sind mit *make_XXX* bezeichnet und geben eine erzeugt Instanz zurück. Alle weiteren Templateklassen müssen selbst erzeugt werden. Nachdem alle nötigen Objekte existieren, wird die *run_ea()* Funktion ausgeführt und die Kontrolle an die EO Bibliothek übergeben.

4.4 *Feedback Control* Kern (Core)

Der *Feedback Control* Kern bildet die *Feedback Control* - Ebene. Hier erfolgt die Adaptierung der von der EO Bibliothek erstellten Werte auf die Gene der Chromosomstruktur. Ebenso wird die Fitness jedes Individuums bestimmt und an die Berechnungs-Ebene übergeben.

4.4.1 *Feedback Control* Parameter

Eine *Entryklasse* der *Feedback Control* - Ebene speichert alle Steuerparameter, die nicht den evolutionären Algorithmus betreffen. Diese *Entryklasse* ist ähnlich wie die in Abschnitt 4.3.2 beschriebene Klasse als Singleton Pattern implementiert.

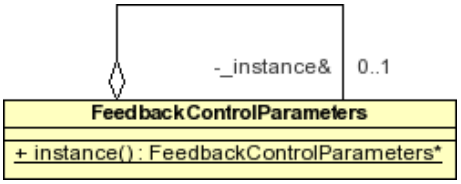


Abbildung 4.9: Klassendiagramm des *Feedback Control* Singleton

```
class FeedbackControlParameters
{
public:
    static FeedbackControlParameters* instance()
    {
        static FeedbackControlParameters _instance;
        return &_instance;
    };

}; // class FeedbackControlParameters
```

Quellcode 4.39: Singletoninstanz der *FeedbackControlParameters* Klasse

Die Singleton Pattern Variante der *FeedbackControlParameters* Klasse besitzt eine andere Lebensdauer als die Singletonimplementierung der *EvolutionaryComputingParameters* Klasse. Sie muss nicht durch eine *destroy()* Funktion zerstört werden. Die Instanz ist kein geschütztes Klassenattribut, sondern eine statische Funktionsvariable. Somit beginnt die Lebensdauer der Instanz mit dem ersten Aufruf der *instance()* Funktion und bleibt erhalten bis zur Beendung des Programms. Diese Art der Implementierung ist einfacher und sicherer. Es ist sichergestellt, dass die Instanz in jedem Fall zerstört wird. Die Variante des Singleton Pattern ist nur sinnvoll, wenn der Zeitpunkt der Zerstörung keine Rolle spielt.

4.4.2 Berechnung der Fitness

Die Berechnung der Fitness hat einen sequenziellen fest vorgegebenen, zyklischen Verlauf. Einige der Abschnitte sind sehr zeitraubend. In Abbildung 4.10 ist der sequenzielle Ablauf dargestellt, wobei die roten Elipsen den Zeitintensivsten Teil markieren. Die blauen Elipsen nehmen weniger Zeit in Anspruch. Ihre Verarbeitung findet in der KouDA Umgebung statt. Die grüne Elipse beansprucht ebenso wenig Zeit. Auf sie kann jedoch kein Einfluss genommen werden, da die Verarbeitung innerhalb der EO Bibliothek statt findet.

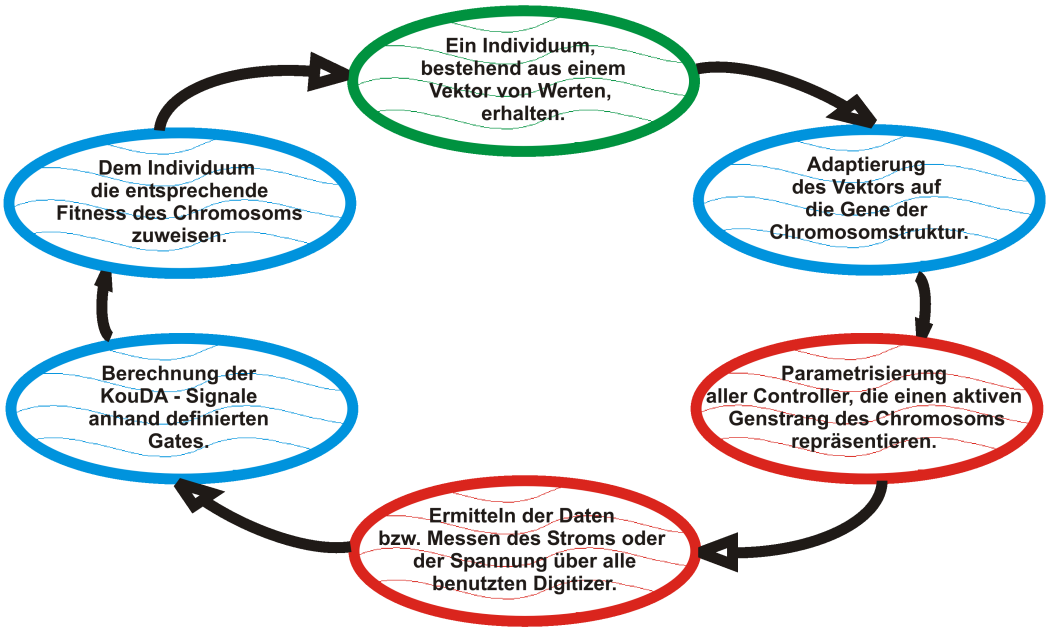


Abbildung 4.10: Zyklischer Verlauf der Parametrisierung

Der Teil, in dem ein Individuum von der EO Bibliothek an einen Funktor übergeben und an die *Feedback Control* Ebene weitergereicht wird, wurde bereits in Abschnitt 4.3.3 beschrieben.

```
double FeedbackControl::evaluate(const vector<double>& indi, bool display)
                        throw(AbortThreadError, FeedbackControlError)
{
    // get an own instance
    FeedbackControlParameters* dev( FeedbackControlParameters::instance() );
    // parameterize the devices
    bool error( ! this->parameterize(indi) );
    double result;
    // get data and display it
    try {
        result = _scanObject->acquire( dev->signal(), display );
    } catch (const ScanError& error) {
        throw FeedbackControlError(string("evaluate(): ") + error.what());
    }
    // check the errors
    if( error )
        // return worst fitness
        return -std::numeric_limits<double>::max();
    // for minimization
    if(dev->toward() == FeedbackControlParameters::minimization)
        result *= -1.0;
    // return fitness
    return result;
}
```

Quellcode 4.40: Fitnessberechnung der *FeedbackControl* Klasse

In der *FeedbackControl* Klasse erfolgt die Implementierung, der in *EvolutionaryComputingObject* deklarierten Funktion *evaluate()*. In ihr sind alle Schritte zur Berechnung der Fitness zusammengefasst. Die Funktion *parameterize()* adaptiert einen Vektor aus Werten zu den Genen der Chromosomstruktur, wodurch die *Controller* parameterisiert werden. Der Aufruf von *acquire()* des Scanobjekts ermittelt die Daten und berechnet die KouDA-Signale.

Die Fitness kann auf drei verschiedene Arten zurückgegeben werden. Sollte während der Parametrisierung der *Controller* ein Fehler auftreten, weil Schaltzeiten oder andere Parameter nicht zulässig sind, dann wird die mögliche, schlechteste Fitness zurück gegeben. Bei einer Optimierung auf ein Minimum, wird die ermittelte Fitness invertiert zurückgegeben, weil die Implementierung des Funktors immer eine Optimierung auf ein Maximum durchführt. Bei der dritten Variante, die der Normalfall sein sollte, wird die ermittelte Fitness des berechneten KouDA-Signals zurückgegeben.

4.4.2.1 Genadaptierung

Die Genadaptierung erfolgt durch einfache Indizierung und Wertzuweisung. Der Locus des Gens wird von der Chromosomstruktur verwaltet und wie in Abschnitt 4.2.4 umgesetzt.

```
bool FeedbackControl::parameterize(const std::vector<double>& indi)
                                throw(FeedbackControlError)
{
    EvolutionaryChromosome* param(NULL);
    try {
        // get an own instance
        param = EvolutionaryComputingParameters::instance()->chromosome();

        // walk through the chromosome
        for(unsigned int idx = 0; idx < indi.size(); idx++) {
            // get and set the next parameter
            param->get(idx)->value( indi[idx] );
        }
    } catch (const EvolutionaryChromosomeError& error) {
        throw FeedbackControlError( string("FeedbackControl::parameterize()")
                                    + error.what() );
    }
    // check and wait for the parameterization
    return param->isSuccessful();
}
```

Quellcode 4.41: Genadaptierung der *FeedbackControl* Klasse

Der Funktion *parameterize()* wird ein STL *std::vector<double>* übergeben. Der Vektor enthält die Allele der Gene in der Reihenfolge, der im Parserfile geschriebenen Suchraum. Das *EvolutionaryChromosome* Objekt hat die Reihenfolge zu Beginn des Algorithmus festgelegt. Die Werte können durch einfache Indizierung der Gene auf die Chromosomstruktur abgebildet werden. Nach Aussen erscheint es, als hätten die Werte des Vektors und die Gene der Chromosomstruktur den gleichen Locus.

4.4.2.2 Parametrisierung der *Controller*

Das Parametrisieren aller *Controller*, die einen aktiven Genstrang des Chromosoms repräsentieren, ist theoretisch die zeitraubendste Verarbeitung. Die *Controller* werden nach einem definierten Protokoll über das Netzwerk oder einer anderen Schnittstelle angesprochen. Schnittstellenzugriffe sind die langsamste Interaktion eines Computers. Um den Datendurchsatz zu steigern, werden zur *Controllerparametrisierung* Threads verwendet. Die Implementierung der Threadobjekte ist in Abschnitt 4.2.1 bereits erklärt worden. Dieser Abschnitt beschäftigt sich mit dem globalen Verlauf der Threads.

Die Abbildung 4.11 zeigt ein Beispiel einer Parametrisierung anhand eines Petrinetzes. In diesem Beispiel wird der DG3008 sowie die BU1708 als zu parametrisierende *Controller* verwendet. Die Schaltzeitpunkte der Burstunit werden mit einem Polynom beschrieben.

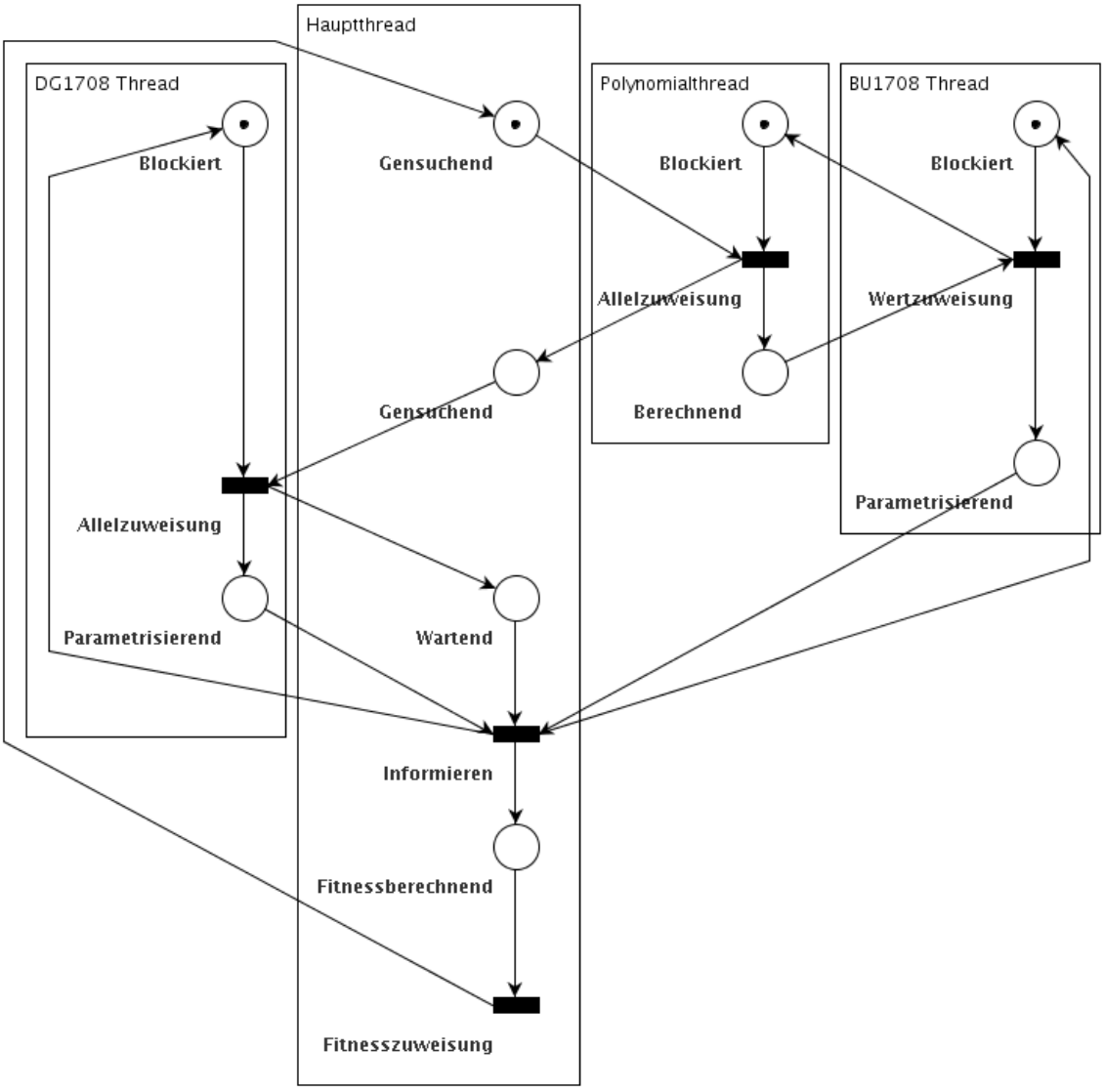


Abbildung 4.11: Petrinetz der Parametrisierung

Der Hauptthread lokalisiert die Gene und weist ihnen ein neues Allel zu. Haben alle Gene eines aktiven Genstrangs oder Teilgenstrangs der Chromosomstruktur ein neues Allel erhalten, so wird der *Controller* parametrisiert. Anschließend muss der wartende Hauptthread über den Zustand der Parametrisierung informiert werden. Nach dem dies geschehen ist werden alle Threads der Genstränge und Teilgenstränge wieder blockiert. Der Hauptthread kann die Berechnung der Fitness starten und sie dem entsprechenden Individuum zuweisen.

Um die Interaktion zwischen den Ojekten zu verdeutlichen, soll das in Abbildung 4.12 gezeigte Sequenzdiagramm helfen. Jede unterschiedliche Farbe eines Objekts steht für einen anderen Thread. Der erste Aufruf im Diagramm, der Funktion *isSuccessful()*, entspricht dem im Quellcode 4.41 gezeigten Rückgabewert. Zu diesem Zeitpunkt haben die Gene ihr neues Allel erhalten. Das Diagramm ist nur eine mögliche schematische Darstellung des Ablaufs, weil es nicht vorhersehbar ist, wann die Threads vom Scheduler ihre Zeitscheibe bekommen und wieder rechnend gemacht werden.

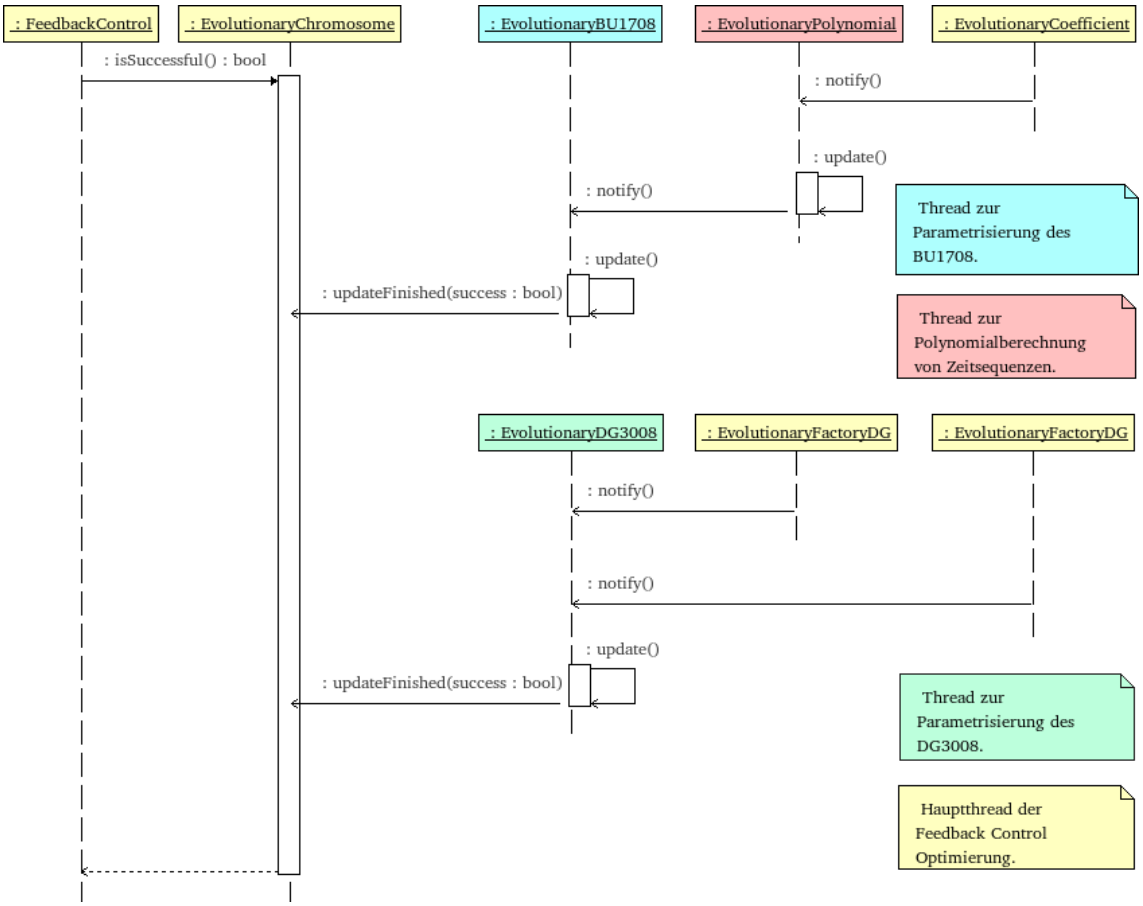


Abbildung 4.12: Parametrisieung der *Controller*

Sobald alle Gene eines beobachtenden Genstrangs oder Teilgenstrangs mit *notify()* einen neuen Zustand gemeldet haben, wird der entsprechende Thread wieder rechenwillig gemacht. Nun kommt es auf das Schedulingverfahren an, wann der Thread seine Zeitscheibe erhält. Ist der Thread rechnend, so wird die *update()* Funktion des abgeleiteten *EvolutionaryController* Objekts ausgeführt. Das Objekt parametrisiert das Gerät und informiert, durch ein emittiertes *dateFinished()* Qt - Signal, über den Erfolg oder Misserfolg der Parametrisierung. Wenn alle *Controller* parametrisiert wurden bzw. alle beobachtenden Threads das *dateFinished()* Qt - Signal emittiert haben, wird das Semaphore in der *isSuccessful()* Funktion wieder freigegeben und der Hauptthread kann seine Verarbeitung fortführen.

```
bool EvolutionaryChromosome::isSuccessful() throw()
{
    // block until all updates were ready
    _updateFinished.acquire();
    // get the state of the updates
    bool success( _success );
    // reset the success state
    _success = true;
    return success;
}
```

Quellcode 4.42: Zustandsabfrage der *EvolutionaryChromosome* Klasse

Wie bereits erwähnt ist der Aufruf der *isSuccessful()* Funktion blockierend. Der fragende Thread wird durch ein Semaphore blockiert und gibt seine Rechenzeit ab. Haben alle aktiven Genstränge der Chromosomstruktur eine erfolgreiche oder nicht erfolgreiche Parametrisierung gemeldet, so wird das Semaphore wieder freigegeben und der fragende Thread wieder rechenwillig gemacht.

4.4.2.3 Akquirierung von Daten

Die Datenerfassung ist ebenso ein zeitraubender Vorgang, wie das Parametrisieren der *Controller*. Die Zeitintensivität hängt jedoch sehr stark von den Einstellungen des Benutzers ab. Die Gegebenheiten des Experiments sind entscheident für die Einstellungen des *Digitizers*. Einige Experimente erzeugen sehr stark verrauschte Signale. Um diese Signale erkennbar darzustellen, wird der Durchschnitt der Datenpunkte über mehrere Messungen gebildet. Bei der Aufnahme von Daten, mit 40Hz Zyklusfrequenz f und einem *Averaging* von 2000, beträgt die Dauer einer Messung:

$$t = \frac{1}{f} \cdot average = \frac{1}{40Hz} \cdot 2000 = 50s \tag{4.5}$$

In KouDA ist bereits eine *Scan* Klasse als *QThread* implementiert. Der *Controller* kann bei üblichen KouDA - Scans sofort nach der Datenerfassung neu parametrisiert werden. In der *Feedback Control* Optimierung müssen jedoch die Daten zuerst ausgewertet werden. Die Werte der Gates müssen errechnet und die KouDA - Signale berechnet werden. Das Ergebnis ist die Fitness eines Individuums.

4.4.3 Starten der Optimierung

Nachdem der Benutzer den Start der *Feedback Control* Optimierung veranlasst hat, wird ein neuer Thread erzeugt. Durch die Benutzung eines weiteren Thread, wird die grafische Oberfläche nicht blockiert. Dieser neue Thread ist der Hauptthread der gesamten *Feedback Control* Optimierung. In ihm findet die Datenerfassung, die Genadaptierung und die Verarbeitung des evolutionären Algorithmus statt.

```
void FeedbackControl::run()
{
    // create the evaluater with myself
    EvolutionaryComputingEvolvingObject evaluator(this);
    // create for acquisition and display the scan object
    unsigned int steps = EvolutionaryComputingParameters::instance()
                                                    ->generalEvaluation();
    ScanAcquisition::timing_t t(1, steps, 1, steps);
    _scanObject = new SingleScanAcquisition(t, true);
    // set output directory
    EvolutionaryComputingParameters::instance()->outputFile(
                        _scanObject->outputDirectory().c_str() );

    // compute with the selected algorithm
    switch( (EvolutionaryComputingParameters::instance())->algorithm() )
    {
        case EvolutionaryComputingParameters::uncorrelatedMutation_one_sigma:
            evaluator.compute<Individual_one_Sigma>();
            break;
        case EvolutionaryComputingParameters::uncorrelatedMutation_n_sigma:
            evaluator.compute<Individual_n_Sigma>();
            break;
        case EvolutionaryComputingParameters::correlatedMutation:
            evaluator.compute<Individual_CMA>();
            break;
        default:
            throw coldmol::ImplementationError("wrong algorithm type");
    }
    delete _scanObject;
    _scanObject = NULL;
}
```

Quellcode 4.43: Threadfunktion der *FeedbackControl* Klasse

In der Threadfunktion wird ein funktionslokales Objekt von Typ *EvolutionaryComputingEvolvingObject* erzeugt. Das Scanobjekt wird instanziiert und zur Datenerfassung vorbereitet. Anschließend wird je nach gewählten Algorithmus die *compute()* Templatefunktion typisiert und ausgeführt.

Der gesamte Threadfunktionsrumpf ist mit einer Fehlerbehandlung umschlossen und wird im folgenden Abschnitt 4.4.4 behandelt.

4.4.4 Fehlerbehandlung der Threads

Die *Feedback Control* Klasse benötigt eine eigene Fehlerbehandlung, da die Threadfunktion keine *Exception* werfen darf. Es würde zu einem Abstruz der Anwendung führen, da eine undefinierte *Exception* nicht gefangen werden kann.

```
class FeedbackControl :  public EvolutionaryComputingObject
{
signals:
    void error(QString msg);

    void warning(QString msg);

    void info(QString msg);

}; // class FeedbackControl
```

Quellcode 4.44: Fehlersignale der *FeedbackControl* Klasse

Ein *QThread* erbt von *QObject* und verfügt über die entsprechenden Makros des Qt-Signal-Slot-Konzepts. Aus diesem Grund wird zur Weiterleitung der Fehler ein Signal emittiert, dass den Fehler spezifiziert. Das Quellcode 4.44 zeigt die drei möglichen Fehlerfälle der Threadfunktion der *FeedbackControl* Klasse.

```
void FeedbackControl::run()
{
    try {
        // create the evaluater with myself
        ...
        // compute with the selected algorithm
        ...
    } catch (const EvolutionaryComputingError& err) {
        emit error(err.what());
    } catch (const EvolutionaryComputingWarning& err) {
        emit warning(err.what());
    } catch (const AbortThreadError& err) {
        emit info("The feedback control scan was aborted!!!");
    } catch (...) {
        emit error("An unknown super exception was thrown.");
    }
}
```

Quellcode 4.45: Fehlerbehandlung der *FeedbackControl* Klasse

Wie in Abschnitt 4.4.3 erwähnt ist der Threadfunktionsrumpf mit einer Fehlerbehandlung umschlossen. Es werden alle drei möglichen Fehlerfälle, die von der *compute()* Templatefunktion spezifiziert sind, abgefangen und durch ein Qt-Signal emittiert.

4.5 Grafische Benutzeroberfläche (GUI)

Die grafische Benutzeroberfläche der *Feedback Control* Optimierung wurde mit dem Qt Designer erstellt und ist vom Typ *QMainWindow*[15]. Die Klasse *DialogFeedbackControl* erbt von *KoudaDialog*, die ein *QMainWindow* ist. Die Basisklasse *KoudaDialog* definiert KoudA standardisierte Funktionen und sollte die Basisklasse aller Dialoge des Softwarepackets sein.

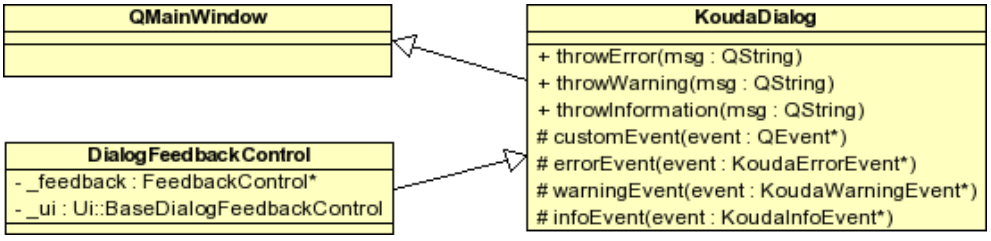


Abbildung 4.13: Klassendiagramm der *Feedback Control* Dialog Klassen

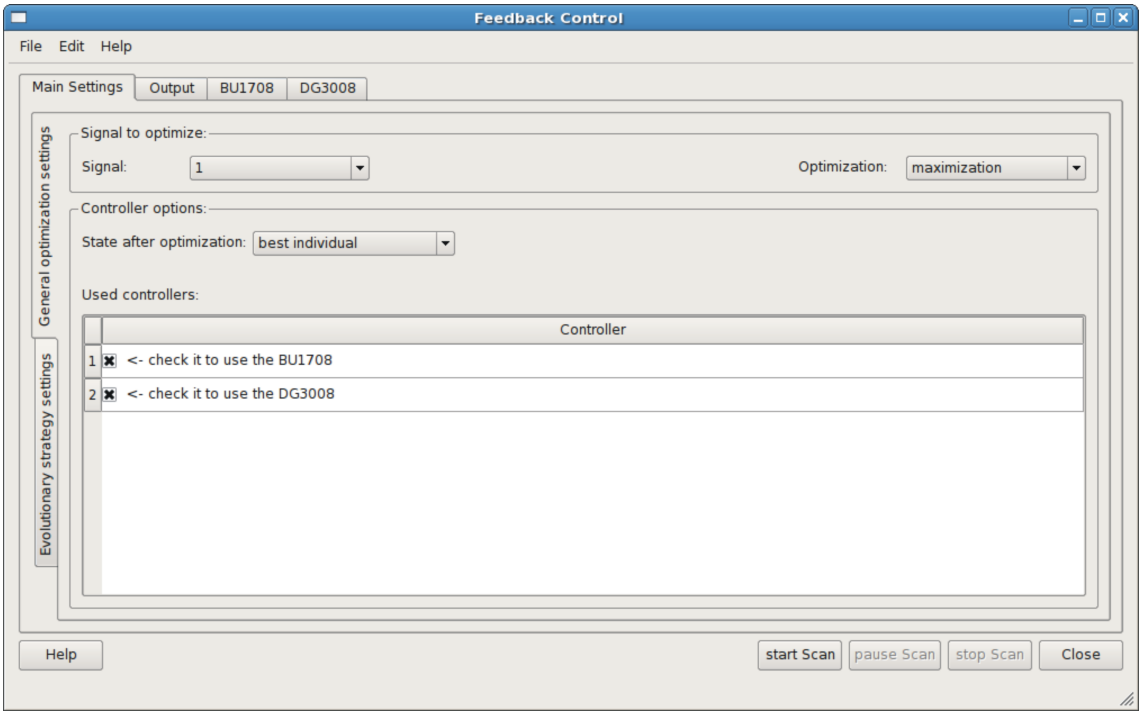


Abbildung 4.14: *Controller* im *Feedback Control* Fenster

Die Abbildung 4.14 zeigt das *Feedback Control* Fenster nach der Instantiierung. Die dargestellte Tabelle beinhaltet die verfügbaren Genstränge. Durch die Zustandsänderung der *QCheckBoxen* in der Tabelle, können die gewünschten Genstränge, wie in Abschnitt 4.2.1.4 beschrieben, aktiviert oder deaktiviert werden.

[15] *QMainWindow* ist eine grafische Klasse der Qt Bibliothek, die für Hauptfenster verwendet wird.

4.5.1 Fehlerbehandlung der GUI

Qt besitzt einen eigenen Eventhändler, der die *Events* der GUI Elemente in die entsprechenden Slots führt. Aus einer Eventabarbeitung bzw. einer Slotfunktion darf keine *Exception* geworfen werden, weil der Eventhändler keine ensprechende Fehlerbehandlung besitzt bzw. die *Exception* nicht fangen kann. Es muss sichergestellt sein, dass jede *Exception* in einer Slotfunktion gefangen wird. Um eine universelle Verarbeitung des Fehlers zu implementieren, fiel die Entscheidung auf die Konvertierung von *Exceptions* zu *QEvents*[16]. Dabei sind *KoudaErrorEvent*, *KoudaWarningEvent* und *KoudaInfoEvent* als die drei wesentlichen KouDA-Events implementiert.

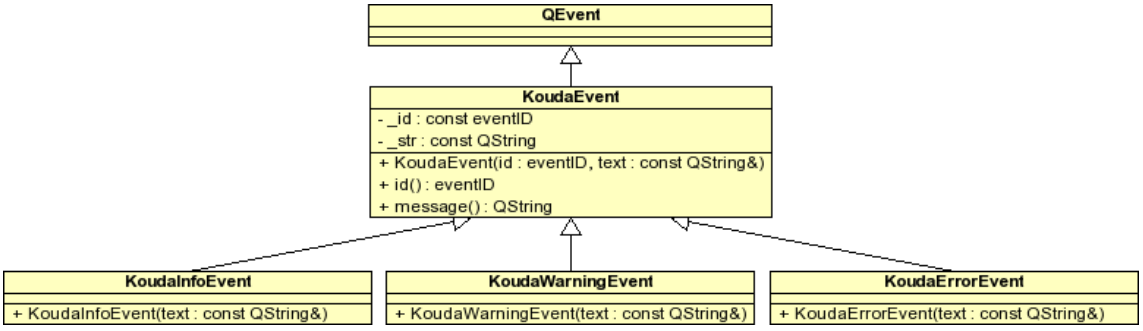


Abbildung 4.15: Klassendiagramm der *Feedback Control Events*

```
class KoudaDialog : public QMainWindow
{
public slots:
    virtual void throwError(QString msg);

    virtual void throwWarning(QString msg);

    virtual void throwInformation(QString msg);

}; // KoudaDialog
```

Quellcode 4.46: *Exception-Event-Slotfunktionen* der *KoudaDialog* Klasse

Jedes dieser *Events* kann durch die entsprechende Funktion eingeleitet werden. Da diese *Events* analog zu *Exceptions* sein sollen, erhielten die Funktionen die Namen *throwError()*, *throwWarning()* und *throwInformation()*. Diese Funktionen sind als Qt-Slotfunktionen definiert. Man hat somit die Möglichkeit *Events* automatisiert auszulösen.

```
    // connect the exception signal of feedback control
    connect( _feedback, SIGNAL( error(QString) ),
             this, SLOT( throwError(QString) ), Qt::QueuedConnection );
    connect( _feedback, SIGNAL( warning(QString) ),
             this, SLOT( throwWarning(QString) ), Qt::QueuedConnection );
    connect( _feedback, SIGNAL( info(QString) ),
             this, SLOT( throwInformation(QString) ), Qt::QueuedConnection );
```

Quellcode 4.47: Threadverbindungen der *DialogFeedbackControl* Klasse

Die in Abschnitt 4.4.4 beschriebenen Threadsignale werden durch das *Qt-Signal-Slot-Konzept* mit den *Exception-Event-Slotfunktionen* verbunden. Die Funktionsaufrufe erfolgen über die *Applikation Message Queue*. Somit ist eine automatisierte Interprozesskommunikation zwischen den Threads realisiert.

[16] *QEvent* ist eine Eventklasse der Qt Bibliothek. Ein Eventobjekt wird aufgrund von Benutzeraktionen erzeugt und von einem Eventhändler verarbeitet.

4.5.2 Veränderung der Parseroptionen

Die GUI Elemente für die Parseroptionen sind dauerhaft im Dialog verankert. Jede Änderung eines Wertes führt zu einer Funktionsabarbeitung, bei der die Änderung sofort in die *EvolutionaryComputingParameters Entryklasse* aus Abschnitt 4.3.2 übernommen wird.

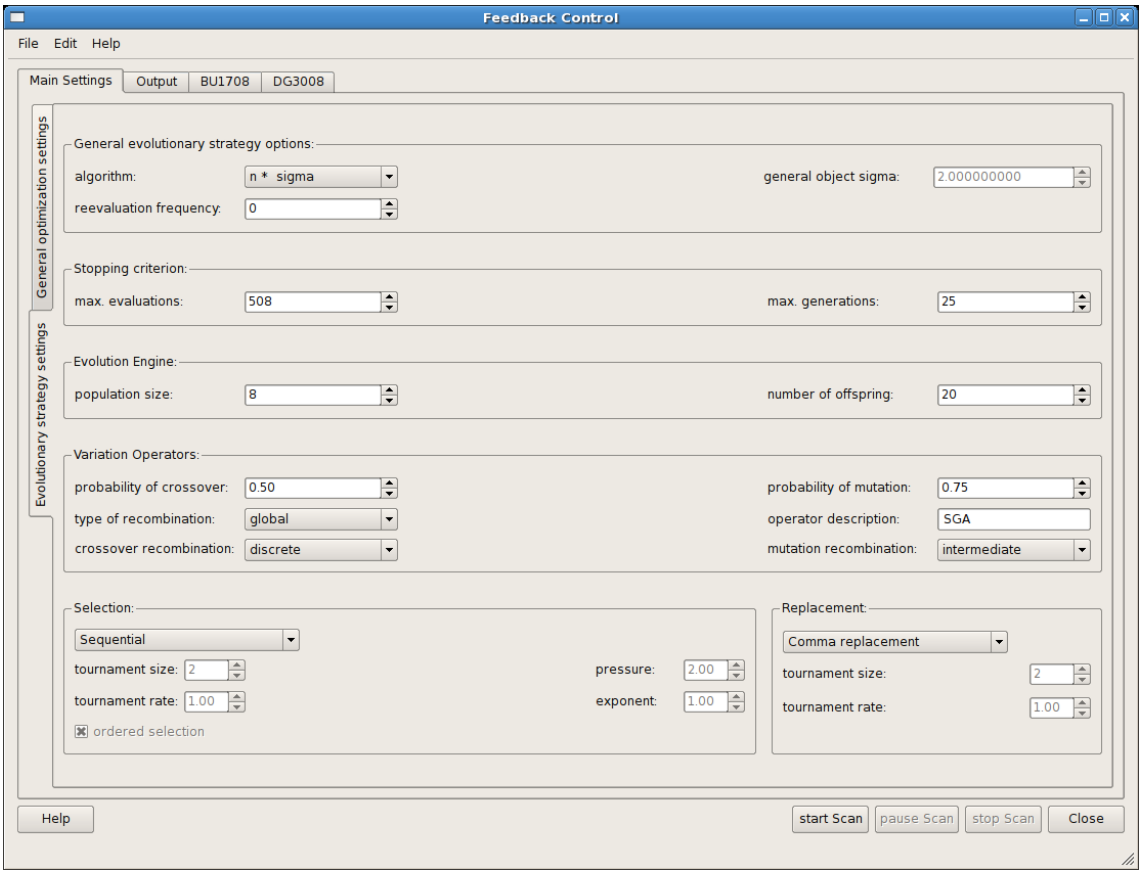


Abbildung 4.16: Parseroption im *Feedback Control* Fenster

Es gibt gewisse Abhängigkeiten zwischen den Optionen, die bei der Implementierung zu beachten sind. Der Wert der maximalen Berechnungen hängt von folgenden Faktoren ab:

$$maxEval = maxGen \cdot numberOffspring + popSize + reevalFreq \cdot popSize \qquad (4.6)$$

Somit führt eine Änderung eines abhängigen Wertes zu einer Aktualisierung der grafischen Darstellung der maximalen Berechnungen.

4.5.3 Genparameter Tabellen

Für jeden *Controller*, der in der *Feedback Control* Optimierung implementiert und in den Hardwareeinstellungen von KouDA eingetragen ist, existiert ein eigener Tab. Jedes dieser Tabs muss mit dem Qt Designer in der *Tabcontrol* eingetragen sein.

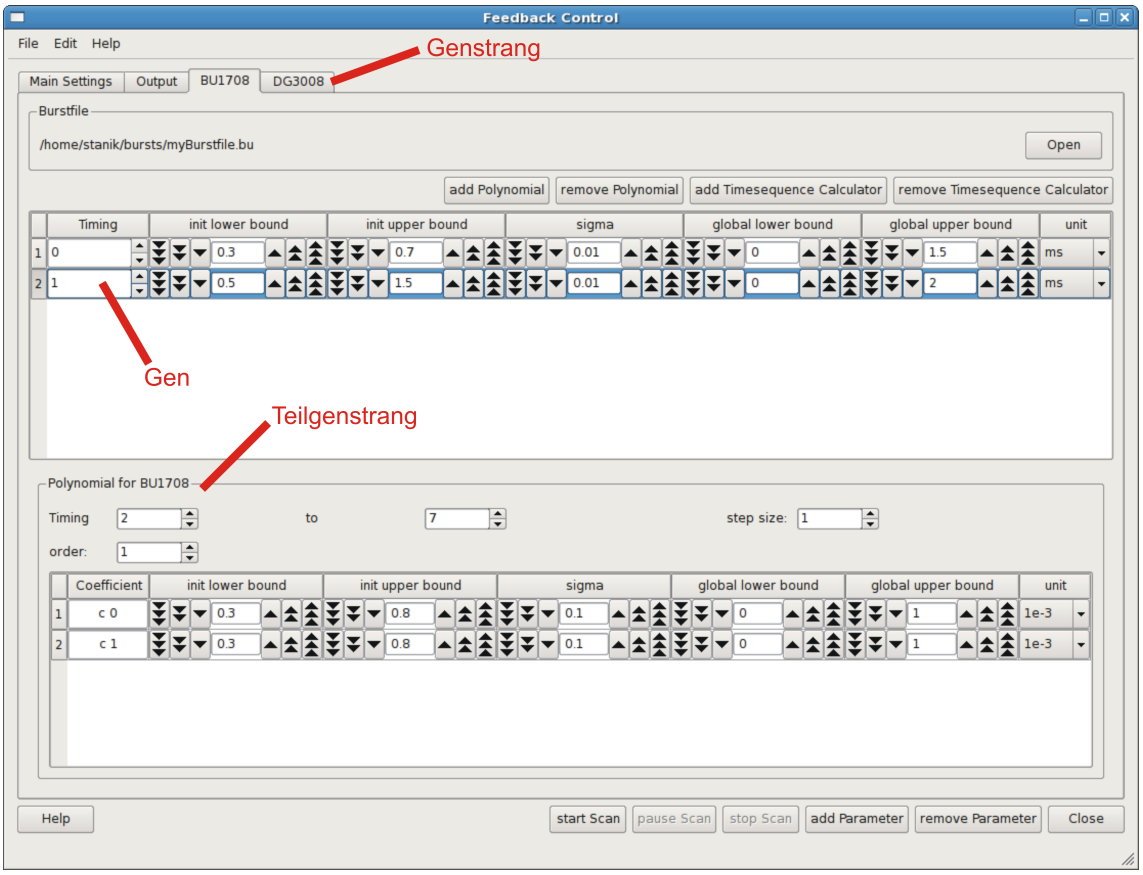


Abbildung 4.17: BU1708 Tab im *Feedback Control* Fenster

Die Gene in der Tabelle oder die Teilgenstränge unter der Tabelle werden mit der Instantiierung oder einer Benutzeraktion dynamisch in die *Tabcontrol* hineingeladen.

4.6 Zusammenfassung

Ein großer Vorteil der Design Pattern ist die frühzeitige Erkennung von Fehlern. Die Pattern sind so konstruiert, dass Fehler bereits zur Compilezeit deutlich werden. Des weiteren ist eine sehr hohe Modularität gegeben. Eine einfache und schnelle Erweiterung oder Änderung der Gene, Teilgenstränge oder Genstränge ist somit möglich.

5. Analyse der Funktionalität

In diesem Kapitel erfolgt die Auswertung der entwickelten Konzepte und ermittelten Daten. Die Analyse der Funktionalität ist in drei Experimente gegliedert:

Optimierung eines unverrauschten Signals: Das Experiment zeigt die Stabilität und Funktionalität der Implementierung.

Optimierung eines verrauschten Signals: Das Experiment zeigt die Anpassungsfähigkeit und Funktionalität der evolutionären Algorithmen.

OH - Abbremsung mit dem AG - Decelerator: Das Abschlussexperiment wird den Nutzen und das zukünftige Einsatzgebiet der Optimierungsalgorithmen präsentieren.

5.1 Optimierung eines unverrauschten Signals

Auf einem Testsystem wurde ein Versuchsaufbau realisiert, bei dem die wichtigsten *Controller* und *Digitizer* zum Einsatz kommen. Anhand des Testsystems soll der Nachweis erbracht werden, dass das in Kapitel 4 entworfene Konzept funktioniert. Der Versuchsaufbau ist so konstruiert, dass das Signal kaum verrauscht ist, damit die Konvergenz der Parameter ohne Nebeneffekte nachgewiesen werden kann. Ein weiterer Aspekt für die Wahl eines unverrauschten Signals ist, dass eine Mittelung über mehrere Datenspuren nicht notwendig ist.

5.1.1 Versuchsaufbau

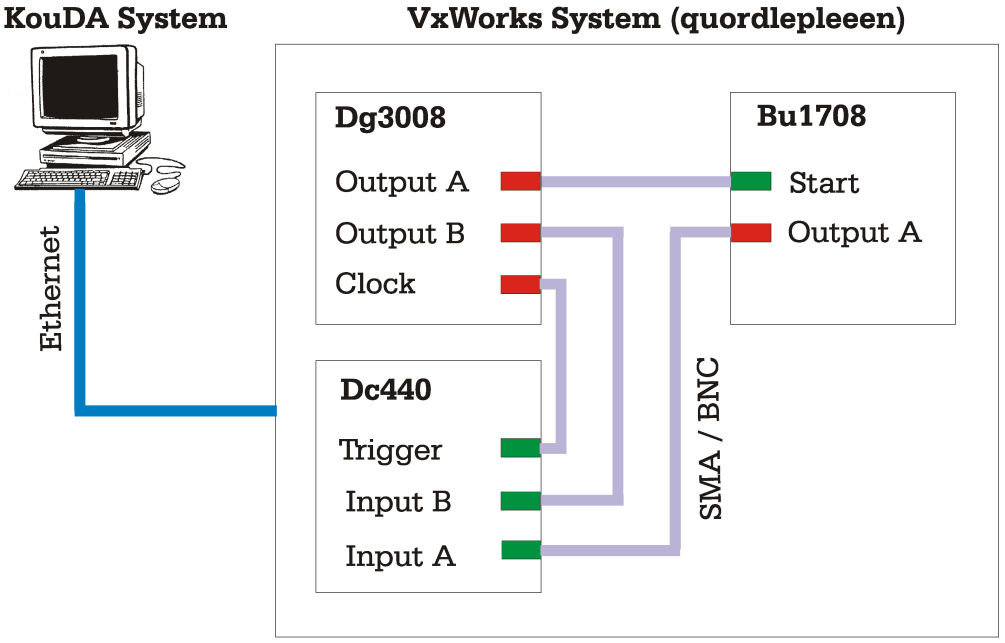


Abbildung 5.1: Versuchsaufbau zum Testen der Implementierung

Der DG3008 und die BU1708 sind die am häufigsten benutzten *Controller* am Fritz-Haber-Institut. Es werden die ersten zwei Kanäle des Delay - Generators benutzt, um ein Burstdelay und einen zu optimierenden Puls zu realisieren. Das Burstdelay ist die Zeit, die vom Maintrigger bis zum Burstunittrigger vergeht. Der Kanal 1, der das Burstdelay bestimmt, ist nicht zur Optimierung freigegeben. Der DC440 misst die erzeugten Signale der *Controller*. Der erste Kanal des *Digitizers* ermittelt die Daten der Burstunit und der zweite die Daten des Delay - Generators.

DG3008 Einstellungen

Startzeitpunkt Kanal 1: 1ms relativ zum Maintrigger
Endezeitpunkt Kanal 1: 15ms relativ zum Start Kanal 1
Startzeitpunkt Kanal 2: 5ms relativ zum Maintrigger
Endezeitpunkt Kanal 2: 10ms relativ zum Maintrigger

BU1708 Burstfile

1000000 0x1
1500000 0x0
2000000 0x1
2500000 0x0
3000000 0x1
3500000 0x0
4000000 0x1
4500000 0x0
5000000 0x1
6000000 0x0
8000000 0x1
10000000 0x0

DC440 Einstellungen
Abtastzeit: 20ms
Anzahl der Datenpunkte: 2000 Punkte
Anzahl der Mittelungen: 1

In Abbildung 5.2 werden die gemessenen Daten des *Digitizers* gezeigt. Es wurden *Gates* festgelegt, die dem Flankenwechsel des Signals entsprechen und in der Grafik rot dargestellt sind.

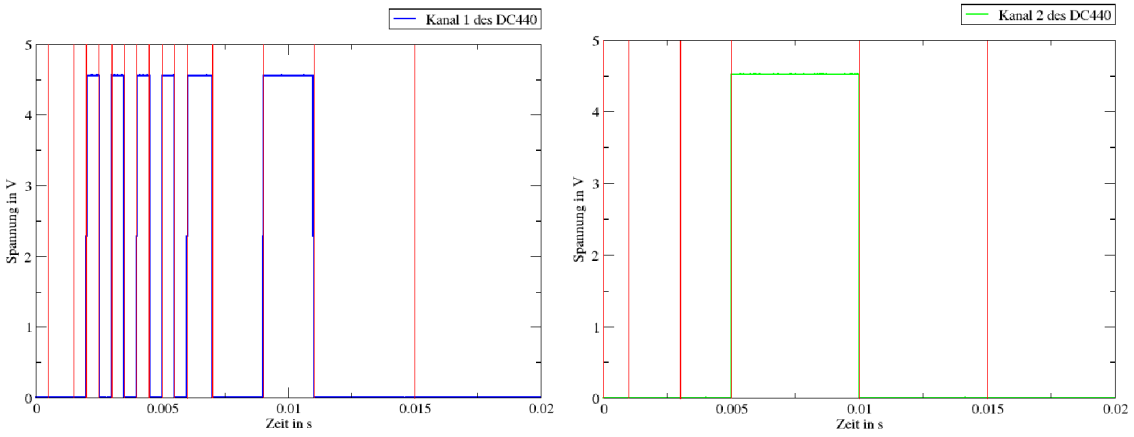


Abbildung 5.2: Ermittelte Daten mit den gesetzten *Gates* des DC440

Die Tabelle 5.1 listet die ausführlichen Definitionen der *Gates* auf.

Gate	Eingang des DC440	Start- und Endpunkt	Start- und Endzeit in ms
G1	Kanal1	[150 , 200]	[1.5 , 2]
G2	Kanal1	[50 , 150]	[0.5 , 1.5]
G3	Kanal1	[200 , 250]	[2.0 , 2.5]
G4	Kanal1	[250 , 300]	[2.5 , 3]
G5	Kanal1	[300 , 350]	[3.0 , 3.5]
G6	Kanal1	[350 , 400]	[3.5 , 4]
G7	Kanal1	[400 , 450]	[4.0 , 4.5]
G8	Kanal1	[450 , 500]	[4.5 , 5]
G9	Kanal1	[500 , 550]	[5 , 5.5]
G10	Kanal1	[550 , 600]	[5.5 , 6]
G11	Kanal1	[600 , 700]	[6 , 7]
G12	Kanal1	[700 , 900]	[7 , 9]
G13	Kanal1	[900 , 1100]	[0 , 11]
G14	Kanal1	[1100 , 1500]	[11 , 15]
G15	Kanal2	[0 , 100]	[0 , 1]
G16	Kanal2	[100 , 300]	[1 , 3]
G17	Kanal2	[300 , 500]	[3 , 5]
G18	Kanal2	[500 , 1000]	[5 , 10]
G19	Kanal2	[1000 , 1500]	[10 , 15]

Tabelle 5.1: *Gates* der zu optimierenden Signale des Testsystems

5.1.2 Optimierung

Das Ziel der Optimierung ist eine Verschiebung der Flanken der in Abbildung 5.2 dargestellten Signale. Um eine möglichst umfangreiche Optimierung durchzuführen, die das gesamte Konzept der Implementierung durchläuft, wurden drei vereinte Optimierungsvarianten verwendet.

5.1.2.1 Delay-Generator-Puls

Der Delay-Generator-Puls, der im rechten Bild der Abbildung 5.2 gezeigt wird, soll in das *Gate 16* optimiert werden. Die Delay-Generator-Parametrisierung stellt einen ganzen Genstrang in der Implementierung aus Abschnitt 4.2.1.4 dar. Die Verarbeitung läuft innerhalb eines Threads ab.

Um eine Bewertung für die Güte der Optimierungsparameter des Genstrangs zu finden, muss ein Signal definiert werden. Das Signal muss das *Gate 16* positiv und alle anderen Werte ausserhalb von *Gate 16* negativ in die Bewertung einfließen lassen.

$$S1 = G16 - G15 - G17 - G18 - G19 \tag{5.1}$$

Die zwei zu optimierenden Parameter sind die Zeitpunkte der positiven und negativen Flanke des Delay-Generator-Pulses. Die Erwartungswerte der Zeitpunkte sind:

$$\textbf{Startzeitpunkt Kanal 2} = 1ms$$

$$\textbf{Endezeitpunkt Kanal 2} = 3ms$$

5.1.2.2 Absolute Schaltzeitpunkte

Die Parametrisierung von absoluten Schaltzeitpunkten der Burstunit ist ein weiterer Genstrang der Optimierung. Es soll der erste Puls aus dem linken Bild in Abbildung 5.2 um 0,5ms vorgezogen werden. Das *Gate 1* begrenzt den optimalen Bereich. Das Signal wird nach dem gleichen Prinzip, wie beim Delay-Generator-Puls, definiert.

$$S2 = G1 - G2 - G3 \tag{5.2}$$

Die zu optimierenden Parmeter der Burstunit, sind der erste und zweite Schaltzeitpunkt der Burstsequenz. Die Erwartungswerte sind:

$$\textbf{1. Schaltzeitpunkt} = 0,5ms$$

$$\textbf{2. Schaltzeitpunkt} = 1ms$$

Die Erwartungswerte der beiden Schaltzeitpunkte sind um $1ms$ kleiner als die Grenzen des *Gate 1*, da das Burstdelay $1ms$ beträgt.

5.1.2.3 Polynom

Das Polynom ist ein Teilgenstrang des übergeordneten BU1708 Genstrangs. Die Berechnung des Teilgenstrangs, dessen Implementierung in Abschnitt 4.2.1.5 und 4.2.3.1 beschrieben wurde, läuft ebenso in einem separater Thread ab. Das Polynom soll durch die Optimierung der Koeffizienten eine Verschiebung von drei Pulsen der Burstunit erreichen. Die Definition des Signals muss dafür alle betroffenen *Gates* beinhalten.

$$S3 = G4 - G5 + G6 - G7 + G8 - G9 + G11 - G12 + G13 - G14 \tag{5.3}$$

Die zu optimierenden Parameter sind die Schaltzeitpunkte 3 bis 8 der Burstsequenz. Sie werden durch ein Polynom 1.Ordnung mit zwei Optimierungsparametern beschrieben.

$$f(x) = c_1 \cdot x + c_0 \tag{5.4}$$

Die Erwartungswerte der Koeffizienten betragen:

$$c_0 = 0,5$$
$$c_1 = 0,5$$

Da das Polynom die absoluten Schaltzeitpunkte der Burstsequenz beschreibt und das Burstdelay $1ms$ beträgt, ist der Erwartungswert von c_0 um 1 kleiner.

5.1.2.4 Fitnessfunktion

Die Optimierung aller sechs Optimierungsparameter soll gleichzeitig erfolgen. Die Summe aller Bewertungen der Genstränge und Teilgenstränge ergibt die Fitnessfunktion:

$$fit : S5 = S1 + S2 + S3 \tag{5.5}$$

$$KouDA : S5 = \begin{pmatrix} & G1 - G2 - G3 \\ + & G16 - G15 - G17 - G18 - G19 \\ + & G4 - G5 + G6 - G7 + G8 - G9 + G11 - G12 + G13 - G14 \end{pmatrix}$$

5.1.3 Datenauswertung

Die Abbildungen 5.3 bis 5.5 stellen den zeitlichen Verlauf der Optimierungsparameter dar. Die ausgewerteten Daten beziehen sich ausschließlich auf die Parameter einer Populationen. Der rot und magenta makierte Bereich ist die Mutationsschrittweite σ der Gene.

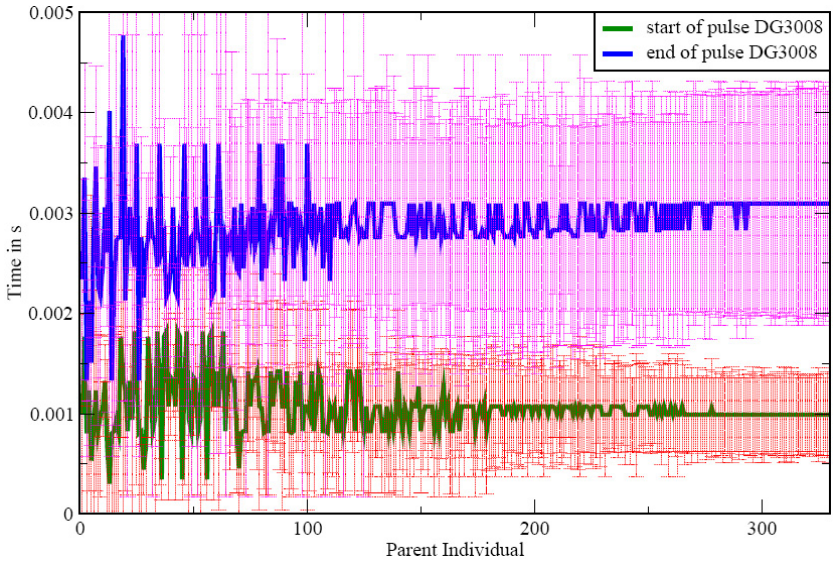


Abbildung 5.3: Optimierungsparameter des DG3008

Die Abbildung 5.3 zeigt die Optimierungsparameter des Delay-Generator-Pulses. Die Parameter sind zu Beginn der Optimierung sehr weit über den Suchraum verteilt. Es ist deutlich zu sehen, dass die Parameter im laufe der Populationen immer stärker zum Optimum konvergieren. Der Startzeitpunkte hat am Ende der Optimierung den Erwartungswert erreicht. Der Endzeitpunkt schwankt während der Evolution um das Optimum. Am Ende konvergiert der Wert, jedoch weicht das Ergebnis leicht vom Optimum ab.

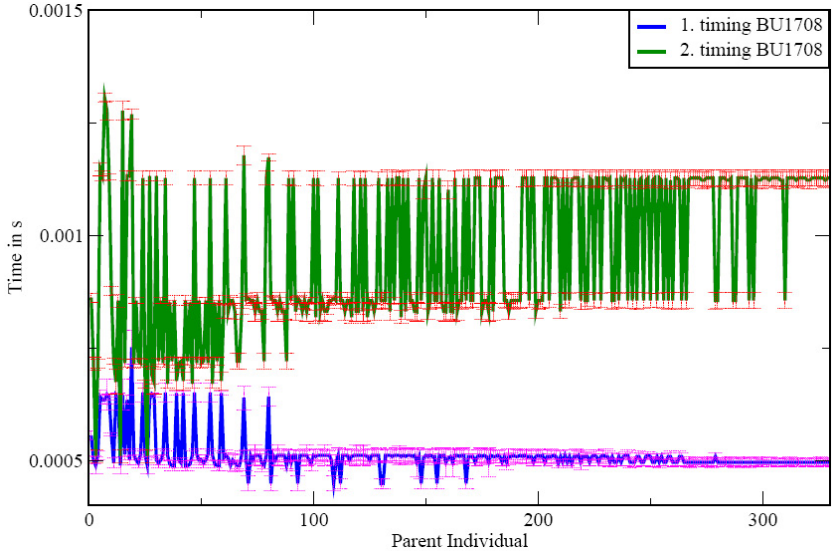


Abbildung 5.4: Optimierungsparameter des BU1708

Die Abbildung 5.4 zeigt die Optimierungswerte der absoluten Schaltzeitpunkte der Burstunit. Die Individuen haben sehr früh den optimalen Wert des ersten Schaltzeitpunktes gefunden. Durch die kleine Mutationsschrittweite der Gene sind die Individuen mit nicht optimalen Werten recht schnell ausgestorben. Der zweite Schaltzeitpunkt ist jedoch sehr stark über den Suchraum verteilt. Es setzen sich während der Evolution zwei Gene durch. Das gesuchte Optimum liegt genau zwischen dem Allel der Gene, deswegen haben die beiden Gene auch die gleiche Güte. Am Ende konvergiert der zweite Schaltzeitpunkt zu einem schlechten Wert.

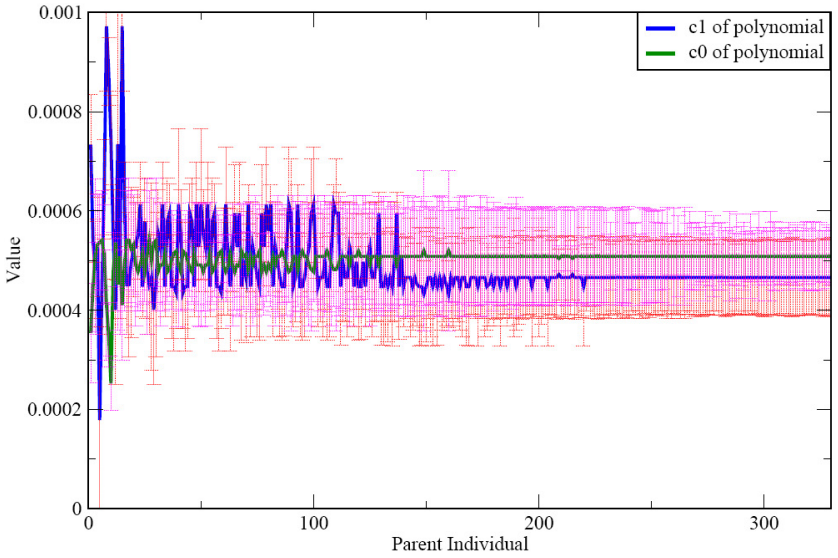


Abbildung 5.5: Optimierungsparameter des Polynoms

Die Abbildung 5.5 zeigt die Optimierungswerte der Koeffizienten des Polynoms. Beide Werte konvergieren sehr schnell zum gesuchten Optimum. Die schnelle Konvergenz der Koeffizienten ist auf den stärken Einfluss auf die Gesamtfitness zurückzuführen. Eine Änderung eines Koeffizienten wirkt sich auf sechs Schaltzeitpunkte aus. Der Koeffizient c_0 hat sein Optimum noch schneller gefunden als der Koeffizient c_1, weil er innerhalb des Polynoms die stärkeren Einfluss auf den polynomialen Verlauf der Schaltzeitpunkte hat. Der Koeffizient c_1 hat nicht sein genaues Optimum erreicht. Eine leichte Abweichung der Steigung ist, in Hinsicht auf die resultierende Größenordnung, zu verkraften.

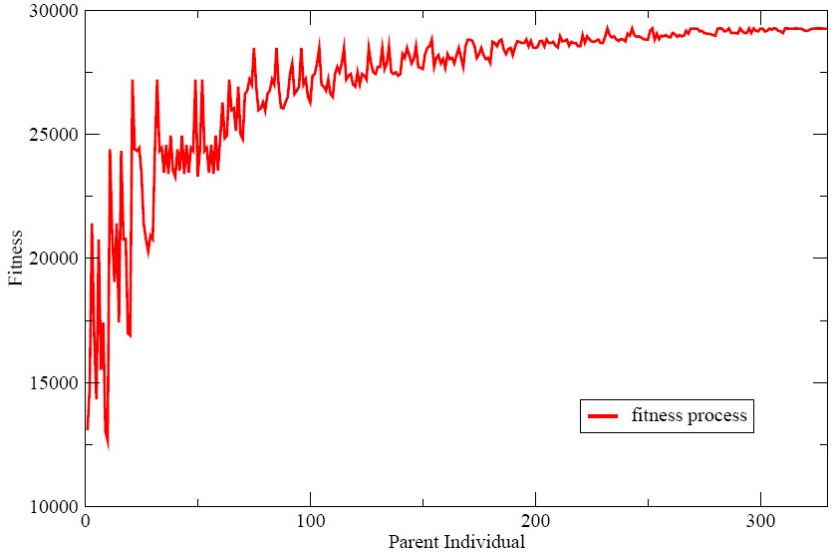


Abbildung 5.6: Fitness der Optimierung eines unverrauschten Signals

Der in Abbildung 5.6 dargestellte Verlauf der Fitness zeigt, dass die Fitness zu Beginn steil ansteigt und im Laufe der Evolution abflacht, da die Parameter sich im Evolutionsverlauf immer langsamer dem Optimum nähern. Die bestmögliche Fitness ist 30000. Durch die hohe Anzahl an Parametern, die nicht alle ihr Optimum gefunden haben, ist das Ergebnis dennoch zufriedenstellend.

5.1.4 Resümee

Anhand des Evolutionsergebnis ist gezeigt, dass die Parametrisierung der *Controller* in den seperaten Threads sowie die Synchronisation mit dem *Feedback Control* Thread funktioniert. Es wurden optimale Werte der Parameter gefunden und eine sehr gute Fitness erreicht. Die Funktionalität der Implementierung ist somit bewiesen.

5.2 Optimierung eines verrauschten Signals

Das Experiment "Conformer-selection of (bio-) molecules" ist der Forschungsschwerpunkt von Frank Filsinger [FK08a]. Alle Quantenzustände von großen Moleüklen sind hochfeldsuchend. Das Experiment selektiert große Biomoleküle unter Anwendung der Alternate Gradient Fokusierung aus Abschnitt 2.1.3. Das Prinzip entspricht dem Alternate Gradient Decelerator, jedoch findet in diesem Experiment keine Abbremsung statt.

Der Versuch soll die Anpassungsfähigkeit der evolutionären Algorithmen zeigen. Das Signal der Moleküldetektion ist sehr verrauscht. Die Laserfrequenz des Detektionslasers ist konstant.

5.2.1 Versuchsaufbau

Die Abbildung 5.7 zeigt eine schematische Darstellung eines Frequenzscans der Wechselhochspannung. Es wurde ein *Gate* definiert, das die zwei relevanten Maxima beinhaltet. Jedes der Maxima stellt die Güte der Fokusierung dar, dabei ist die Lage des Moleküls entscheidet zu welchem Maximum es gehört.

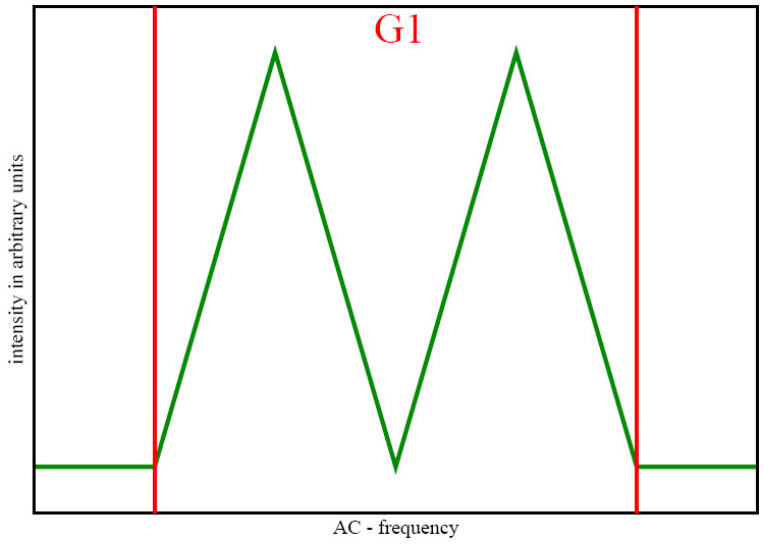


Abbildung 5.7: Schematische Darstellung eines Frequenzscans

5.2.2 Optimierung

Das Ziel der Optimierung ist die optimale Fokusierung zu finden. Es sollen zwei von einander Abhängige Parameter optimiert werden.

5.2.2.1 Burstdelay

Das Burstdelay bestimmt den Startzeitpunkt, zu dem die Wechselspannung an den Elektroden geschaltet wird. Als Optimierungsparameter wird der Startzeitpunkt des Pulses vom Delay-Generator verwendet.

Der Suchraum S des Startzeitpunktes ist folgendermaßen definiert:

$$\textbf{Burstdelay: } S = [0.0009ms, 0.0013ms]$$

Der Suchraum S der Startpopulation ist folgendermaßen definiert:

$$\textbf{Startpopulation: } S = [0.001ms, 0.0012ms]$$

Die Mutationsschrittweite σ der Startpopulation ist folgendermaßen definiert:

$$\textbf{Mutationsschrittweiten: } \sigma = 2 \cdot 10^{-5}$$

5.2.2.2 Frequenz der Wechselhochspannung

Die Frequenz der Wechselhochspannung ist der eigentliche zu optimierende Parameter. Die Frequenz ist entscheident für die Güte der Fokussierung. Das *coldmol_timesequence* Programm wurde speziell für diese Optimierung angepasst. Es wird die *EvolutionaryBeamlineCalculator* Klasse benutzt, die das externe *coldmol_timesequence* Programm ausführt.

Der Suchraum S der Frequenz ist folgendermaßen definiert:

$$\textbf{Frequenz: } S = [2200Hz, 3200Hz]$$

Der Suchraum S der Startpopulation ist folgendermaßen definiert:

$$\textbf{Startpopulation: } S = [2500Hz, 2800Hz]$$

Die Mutationsschrittweite σ der Startpopulation ist folgendermaßen definiert:

$$\textbf{Mutationsschrittweiten: } \sigma = 30$$

5.2.2.3 Evolutionsparameter

Die Tabelle 5.2 zeigt die verwendeten Einstellungen der Evolutionsstrategie. Als evolutionären Algorithmus wurde die unkorrelierte Mutation mit n Mutationsschrittweiten σ benutzt.

Parameter	**Wert**
Populationsgröße	5
Selektionsverfahren	Deterministischer Wettkampf (2)
Anzahl der Nachkommen	15
Evolutionsstrategie	(μ , λ) - ES
Anzahl der Generationen	50
Mutationswahrscheinlichkeit	75%
Kreuzungswahrscheinlichkeit	50%
Rekombination	Discrete Recombination

Tabelle 5.2: Evolutionsparameter zur Optimierung der Frequenz

5.2.3 Datenauswertung

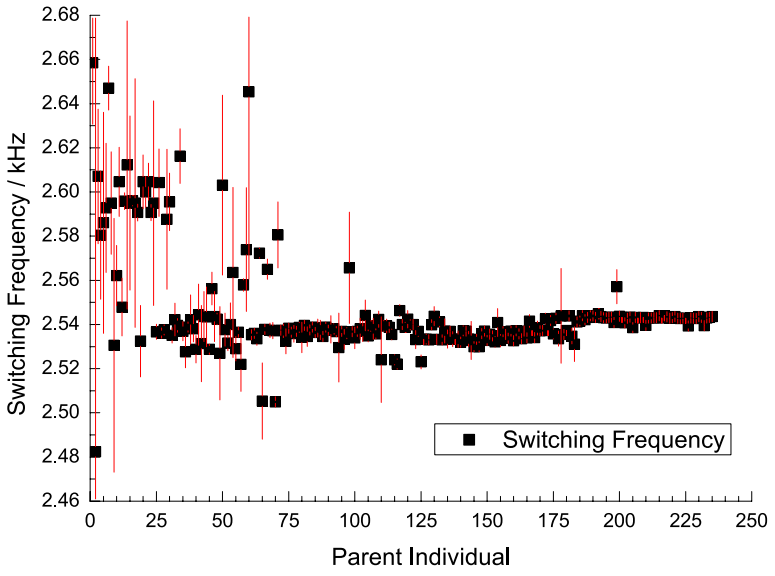


Abbildung 5.8: Optimierungsparameter der Frequenz

Die Abbildung 5.8 [SF07] zeigt den Verlauf der Frequenz. Zu Beginn der Evolution ist deutlich zu sehen, dass die Individuen sich über den gesamten Suchraum verteilen. Nach der zweiten Population laufen die Werte der Individuen in die beiden Maxima, die in Abbildung 5.7 schematisch dargestellt sind. Es überwiegt die Anzahl der Individuen mit ca. $2,6kHz$. Sie besitzen jedoch noch eine relativ hohe Mutationsschrittweite, wodurch die Werte der Nachkommen stärker variieren. In der sechsten Generation, die aus den Individuen 25 bis 30 besteht, sind die Individuen gleichermaßen auf den Suchraum in den beiden Maxima verteilt. Die Individuen mit ca. $2,53kHz$ besitzen zu diesem Zeitpunkt eine wesentlich kleinere Mutationsschrittweite. Nach der zehnten Generation kann man die Werte der Individuen als konvergiert betrachten. Es wurde ein Optimum von $2,54kHz$ gefunden.

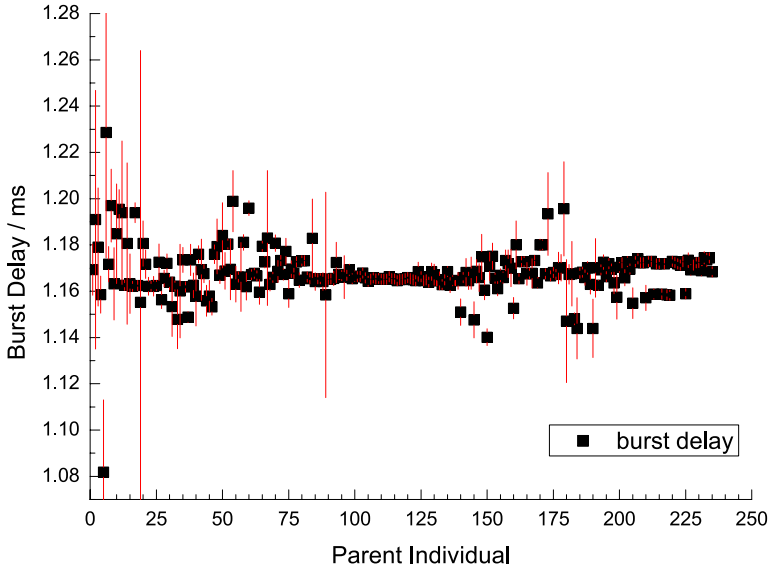


Abbildung 5.9: Optimierungsparameter des Burstdelay

Die Abbildung 5.9 [SF07] zeigt den Werteverlauf vom Burstdelay. Die Individuen konzentrieren sich von Beginn an im oberen Drittel des Suchraums der Startpopulation. Durch den Verlauf der ersten 25 Generationen ist die in Abbschnitt 2.3.3.2 genannte Aussage bestätigt worden. Die Individuen mit gut angepassten Mutationsschrittweiten erzeugen im Mittel bessere Nachkommen. Die Mutationsschrittweite hat sich automatisch angepasst. Individuen mit schlechten Mutationsschrittweiten wurden durch die Selektion ausgetilgt.

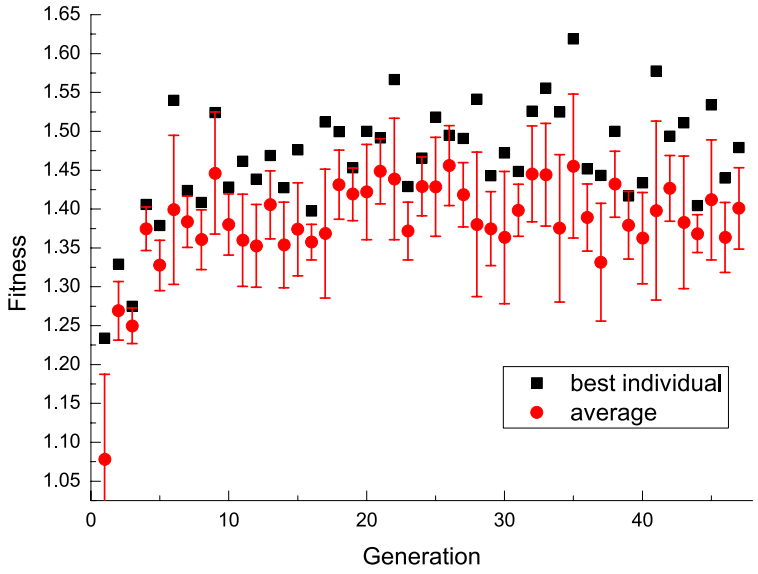


Abbildung 5.10: Fitness der Optimierung eines verrauschten Signals

Der Verlauf der Fitness, der in Abbildung 5.10 [SF07] dargestellt ist, zeigt eine ständige Verbesserung bis zur 25. Generation. Der Mittelwert einer Generation hat dort die kleinste Streuung, da die Optimierungsparameter zu einem Optimum in der entsprechenden Generation konvergiert sind. Die Streuung der Fitness ist in dieser und den Folgegenerationen dennoch existent, da das Signal stark verrauscht ist und sich in der Fitnessfunktion wiederspiegelt.

5.2.4 Resümee

Das Evolutionsergebnis zeigt, dass die evolutionären Algorithmen auch bei einem stark verrauschten Signal ein Optimum gefunden haben. Die Anpassungsfähigkeit durch die unkorrelierte Mutation mit n Mutationsschrittweiten ist mit der Datenauswertung gezeigt worden.

Es kann leider keine konkrete Aussage über das gefundene Optimum gemacht werden. Ein Frequenzscan, wie er in Abbildung 5.7 schematisch dargestellt ist, beansprucht sehr viel Zeit. Das gefundene Optimum muss nicht das globale Maximum sein, da die beiden existierenden Maxima nicht gleich groß sind. Da die zwei Optimierungsparameter stark von einander abhängen und beide konvergiert sind, ist höchstwarscheinlich das globale Optimum gefunden worden. Frank Filsinger hält die resultierenden Parameter für sinnvolle Werte [SF07].

5.3 OH-Abbremsung von 355m/s auf 307m/s

Beim Abschlussexperiment sollen OH - Moleküle mit Hilfe des Alternate Gradient Decelerator von einer Anfangsgeschwindigkeit von $355\frac{m}{s}$ auf $307\frac{m}{s}$ abgebremst werden. Die Funktionsweise und das Aussehen des Alternate Gradient Decelerators wurde in Abschnitt 2.1.3 beschrieben.

Anhand des Abschlussexperiment soll der Nutzen und das zukünftige Einsatzgebiet der Optimierungsalgorithmen gezeigt werden. Das Experiment ist ein wichtiger Bestandteil der Diplomarbeit, der vor der Ausarbeitung als auch in der Aufgabenstellung festgelegt wurde.

5.3.1 Versuchsaufbau

Die Abbildung 5.11 zeigt eine Flugzeit (*time of flight*) Messung, bei der die Burstsequenz mit den theoretisch besten Werten berechnet wurde. Diese Messung wurde vor Beginn der *Feedback Control* Optimierung durchgeführt, um die Spitze (Peak) der das gesuchte Molekülpacket darstellt, zu finden. Die dargestellten Daten wurden mit 2000 Scanmittelungen aufgenommen, d.h. es wurde 2000 mal eine Datenspur mit dem Digitizer gemessen und der Durchschnitt aller Spuren zur dargestellten Kurve gemittelt. Die hohe Mittelung der Datenspuren ist notwendig, da das Signal sehr stark verrauscht ist. Die Laserfrequenz wird über mehrere Stationen modeliert und mit der *RDL Control* stabilisiert. Dennoch ist die Laserfrequenz über einen langen Zeitraum nicht konstant.

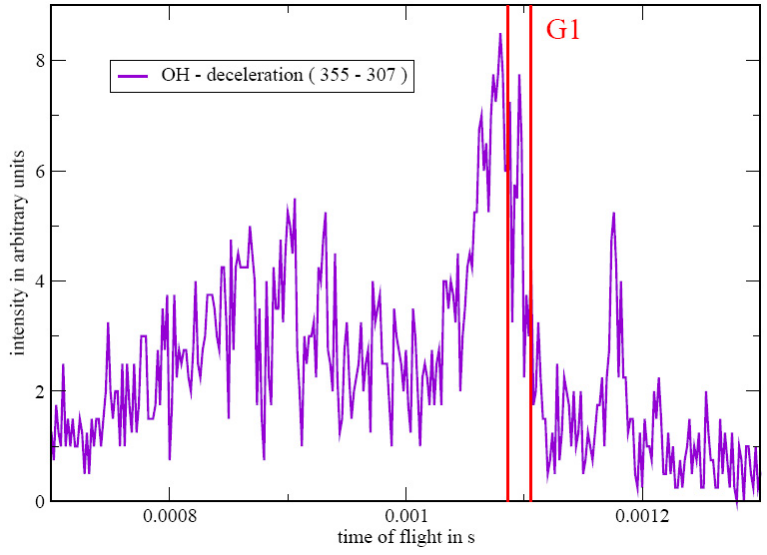


Abbildung 5.11: *Gate* zur OH-Abbremsung von $355\frac{m}{s}$ auf $307\frac{m}{s}$

Die in Abbildung 5.11 dargestellte Spitze (*peak*), innerhalb des definierten *Gates*, ist das gesuchte Molekülpaket.

5.3.2 Optimierung

Das Ziel der Optimierung ist eine bessere Fokussierung zu erreichen, dazu ist eine feste Anfangs- und Endgeschwindigkeit vorgegeben. Zur Optimierung wird die *Evolutionary-BeamlineCalculator* Klasse benutzt, die das externe *coldmol_timesequence* Programm ausführt. Die Abbremsstrecke zwischen den Elektroden ist auf $d = 0.00525m$ festgelegt. Es soll die Fokussierstrecke f optimiert werden, um den in Abbildung 5.11 dargestellten Peak zu maximieren.

Es ist eine Templatedatei zur Optimierung der Fokussierstrecke erstellt worden. Sie beinhaltet die zur Berechnung der Burstsequenz notwendigen Eigenschaften der Maschine und des Moleküls. Die Templatedatei ist folgendermaßen definiert:

```
[machine]
beamline = AG2
configuration = 15kV
field_scaling = 1.0
stages = 1*1;8*3;1*2;

[molecule]
species = OH
state = 1.5:1.5:1.5:e

[timesequence]
v=355.
system = pos_single_old
PARAMETER_BURSTFILE
PARAMETER_F_
d = 27*0.00525
```

5.3.2.1 Optimierungsparameter

Es wird an 27 Elektrodenpaaren eine Fokussierung veranlasst. Das Entspricht einer Burstsequenz mit 54 Schaltzeitpunkten i. Die Werte der Fokussierstrecke $f(i)$ werden durch das Polynom

$$f(i) = f + i \cdot f' \qquad \text{für } i \in \mathbb{N};\ i \geq 0;\ i \leq 53$$

beschrieben.

Der Suchraum S von dem Koeffizienten f ist folgendermaßen definiert:

$$\textbf{Koeffizienten } f\textbf{:}\ S = [-1.75mm, -0.75mm]$$

Der Suchraum S des Koeffizienten f der Startpopulation ist:

$$\textbf{Startpopulation:}\ S = [-1.5mm, -1mm]$$

Die Mutationsschrittweite σ des Koeffizienten f der Startpopulation ist:

$$\textbf{Mutationsschrittweiten:}\ \sigma = 0.1mm$$

Der Suchraum S von dem Koeffizienten f' ist folgendermaßen definiert:

$$\textbf{Koeffizienten } f'\textbf{:}\ S = [-0.075mm, -0.035mm]$$

Der Suchraum S des Koeffizienten f' der Startpopulation ist:

$$\textbf{Startpopulation:}\ S = [-0.06mm, -0.04mm]$$

Die Mutationsschrittweite σ des Koeffizienten f der Startpopulation ist:

$$\textbf{Mutationsschrittweiten:}\ \sigma = 0.005mm$$

5.3.2.2 Evolutionsparameter

Die Tabelle 5.3 zeigt die verwendeten Einstellungen der Evolutionsstrategie. Als evolutionären Algorithmus wurde die unkorrelierte Mutation mit n Mutationsschrittweiten σ benutzt, da sie sich bereits als eine gute Evolutionsstrategie bewährt hat.

Parameter	**Wert**
Populationsgröße	5
Selektionsverfahren	Deterministischer Wettkampf (2)
Anzahl der Nachkommen	15
Evolutionsstrategie	(μ , λ) - ES
Anzahl der Generationen	50
Mutationswahrscheinlichkeit	75%
Kreuzungswahrscheinlichkeit	50%
Rekombination	Discrete Recombination

Tabelle 5.3: Evolutionsparameter zur Optimierung der Fokussierstrecke f

5.3.3 Datenauswertung

Aus den Abbildungen 5.12 und 5.13, die den zeitlichen Verlauf der Optimierungsparameter darstellen, geht hervor, dass nach 8 Populationen die *Feedback Control* Optimierung abgebrochen wurde. Das hat den Grund, dass die Messungen mit 500 Scanmittelungen durchgeführt wurden. Die Zyklusfrequenz beträgt $20Hz$, somit dauert die Messung einer Fitness:

$$t_{Messung} = \frac{1}{f} \cdot average = \frac{1}{20Hz} \cdot 500 = 25s$$

Durch die Verwendung des externen *coldmol_timesequence* Programms, das für die Erzeugung einer Burstsequenz viele Integrale berechnen muss, kommte eine Berechnungsdauer von ca. $t_{Berechnung} \approx 30s$ pro Individuum dazu:

$$t_{Individuum} = t_{Messung} + t_{Berechnung} = 25s + 30s \approx 55s$$

Dazu kommt eine pauschale Verarbeitungszeit der KouDA Software. Somit dauert die Messung bzw. Berechnung der Fitness einer ganzen Population:

$$t_{Population} = t_{Individuum} \cdot \lambda = 55s \cdot 15 \approx 15min$$

Bevor ein Signal am Alternate Gradient Decelerator gemessen werden kann, muss der Laser justiert, die Faserfrequenz stabilisiert und noch viele weitere Vorbereitungen getroffen werden. Alles in allem beansprucht die Aufnahme der Daten bzw. die Durchführung der *Feedback Control* Optimierung am Alternate Gradient Decelerator einen ganzen Tag.

Ein weiterer Grund für den Abbruch ist die sich dauerhaft verschlechternde Laserintensität. Die *Feedback Control* Optimierung konnte nicht unterbrochen werden, um den Laser neu auszurichten. Nach der Durchführung des Abschlussexperiments, wurde der in Abschnitt 4.3.1 beschriebene Pausemechanismus nachträglich implementiert. Eine weitere *Feedback Control* Optimierung zur Abbremsung von OH - Molekülen war zu einem späteren Zeitpunkt nicht mehr möglich, da erstens das Gas des Eximer Laser ausgegangen ist und zweitens das Experiment für andere Moleküle umgebaut wurde.

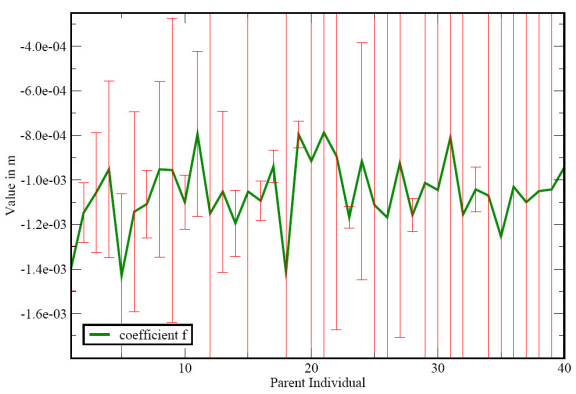


Abbildung 5.12: Koeffizienten f

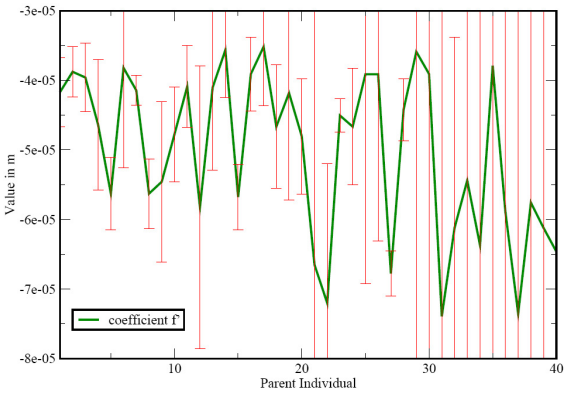


Abbildung 5.13: Koeffizienten f'

Die Abbildungen 5.13 und 5.12 zeigen den Werteverlauf der Koeffizienten. Die Mutationsschrittweiten sind zu Beginn in der Größenordnung der angegebenen Mutationsschrittweiten der Startpopulation. Im Laufe der Evolution nehmen sie sehr stark zu. Durch die großen Mutationsschrittweiten ist kein Konvergenzverhalten der Optimierungsparameter zu erkennen. Dieses Phänomen kann möglicherweise an der sich durchgehend verschlechternden Fitness liegen.

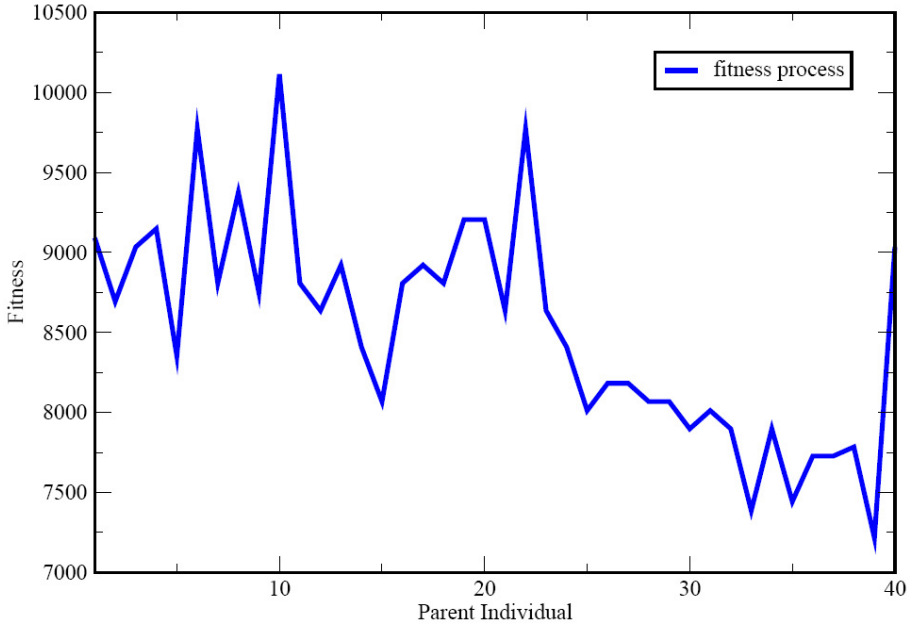


Abbildung 5.14: Fitness der Fokussierung von OH - Molekülen

Der Verlauf der Fitness, der in Abbildung 5.14 dargestellt ist, zeigt in der zweiten Population eine sehr stark schwankende Verbesserung im gegensatz zu der ersten Population. Der Fitnessdurchschnitt aller folgenden Populationen wird zunehmend schlechter. Die Tatsache, dass die Fitness eine Verschlechterung während der Evolution aufweist, hat nicht zu bedeuten, dass die Optimierungsparameter schlechter werden. Die Laserintensität nimmt während der Messungen zunehmend ab. Somit werden auch weniger Moleküle detektiert. Die Abbildung 5.14 zeigt also nur die Güte eines Individuums zu einem bestimmten Zeitpunkt.

5.3.4 Resümee

Um eine Aussage über das Ergebnis der *Feedback Control* Optimierung machen zu können, muss eine Refferenzmessung erfolgen. Physikerinnen und Physiker des Fritz-Haber-Instituts haben über Wochen hinweg Messungen und Optimierungen zur OH - Abbremsung durchgeführt. Die Optimierungen basierten auf Erfahrungswerte, Ausprobieren und Simulationen. Es wurden einige sehr gute Burstsequenzen erzeugt, die als Referenz verwendet werden können.

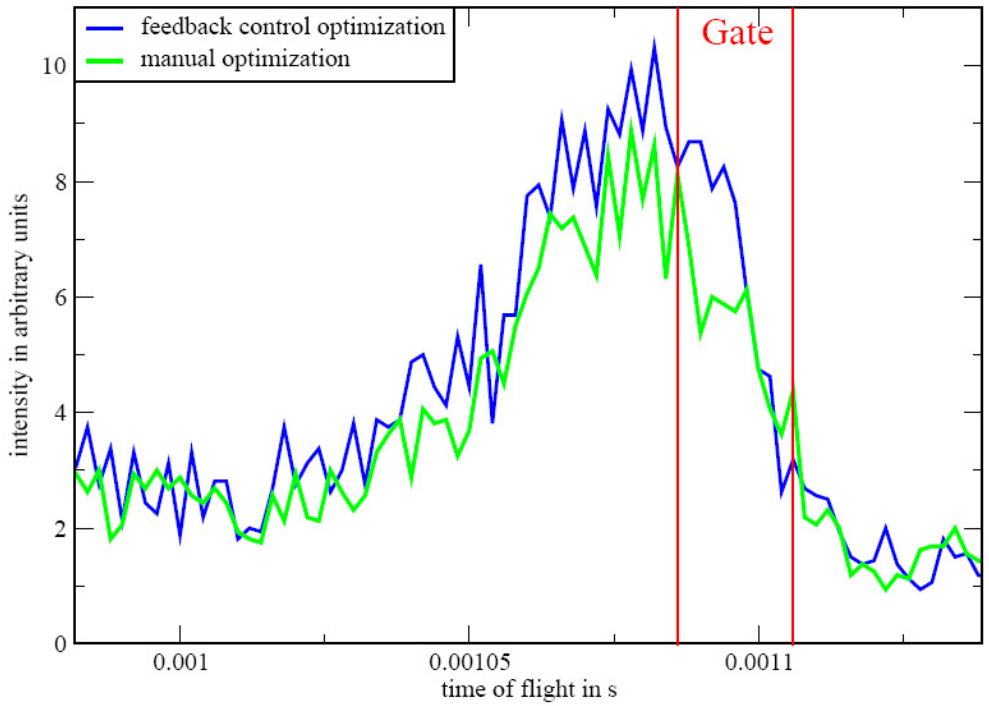


Abbildung 5.15: *Feedback Control Optimization vs. Manuel Optimization*

Die Abbildung 5.15 zeigt zwei über mehrere Messungen gemittelte Signalverläufe. Zur Erzeugung der Datenspur, die das Ergebnis der *Feedback Control* Optimierung präsentieren soll, wurde das fünfte Individuum der achten Generation verwendet. Es hat aus der letzten Generation die beste Fitness.

Das Ergebnis ist beeindruckend, wenn man bedenkt, dass die Referenz eine bereits optimierte Konfiguration ist. Die Abbildung 5.15 ist ein vergrößerter Ausschnitt aus Abbildung 5.11. Die unterschiedliche Intensität der Signale ist auf den Abfall der Laserintensität zurück zu führen, da die Daten aus Abbildung 5.11 vor der Optimierung und die Daten aus Abbildung 5.15 nach der Optimierung aufgenommen wurden.

5.4 Zusammenfassung

Die Auswertung der Experimente hat gezeigt, dass das entworfene Konzept der Implementierung funktioniert. Der Einsatz der evolutionären Algorithmen erwies sich als geeignete Lösungsstrategie zur Optimierung von Parametern. Die Algorithmen sind durch die Implementierung in KouDA problemunabhängig und vielseitig verwendbar.

6. Zusammenfassung und Ausblick

Gegenstand der Diplomarbeit war die Implementierung der *Feedback Control* Optimierung. Dazu wurden "Evolutionäre Algorithmen" verwendet und an dem komplexen Alternate Gradient Decelerator Experiment getestet. Das Ziel war eine Optimierung von Parametern zur Effizienzsteigerung der Experimente.

Die Implementierung der *Feedback Control* Optimierung, die in Kapitel 4 beschrieben wurde, ist in vier Ebenen gegliedert:

Die **Chromosom - Ebene** beinhaltet die Chomosomstruktur, die als "Inverses Observer Pattern" umgesetzt ist. Die Observer - Objekte stellen die Gen- und Teilgenstränge dar. Jedes dieser Objekte ist ein Thread und parametrisiert die *Controller* anhand der Gene innerhalb der Genstränge.

Die **Berechnungs - Ebene** nutzt die *Evolving Object* Bibliothek, um die evolutionären Algorithmen in die KouDA-Software zu implementieren. Eine Klasse, die als eine Variante des Singleton Pattern umgesetzt ist, speichert und verwaltet alle nötigen Evolutionseinstellungen.

Die ***Feedback Control* - Ebene** steuert die Interaktion der beiden unteren Ebenen. Sie ermittelt die Daten anhand von *Digitizern* und berechnet aus den definierten *Gates* die KouDA-Signale, die der Fitness der Individuen entsprechen.

Die grafische **Darstellungs - Ebene** bildet, durch eine grafische Oberfläche, die Schnittstelle zum Benutzer. Aus dieser Ebene werden die Evolutionseinstellungen sowie die Optimierungseigenschaften in den zwei *Entryklassen*, die als unterschiedliche Singleton Patternvarianten implementiert sind, verändert.

Die eingesetzten Design Pattern bieten eine hohe Modularität, durch die leicht weitere *Controller* zur Optimierung implementiert werden können.

Die in Kapitel 5 vorgestellten Auswertungen zeigen das enorme Potential, das die evolutionären Algorithmen bieten. Mit nur sehr rudimentären Erfahrungswerten lassen sich automatisiert Parametervariationen durchführen, um die Effizienz der Experimente zu steigern bzw. Ungenauigkeiten weg zu optimieren. Da die Anforderungen in jeder Hinsicht erfüllt sind, wird dieses Vorgehen in Zukunft weiterhin am Fritz-Haber-Institut eingesetzt. Besonders interessant wird die Anwendung von evolutionären Algorithmen bei komplett neuen Experimenten der Molekülabbremsung.

Es gibt einige Verbesserungsvorschläge, die vieleicht in naher Zukunft von den Entwicklern der KouDA-Software umgesetzt werden:

Verschmelzung der coldmolLib mit KouDA.
Durch das Zusammenführen der beiden Softwarepackete sind die installierten Programme auf den Messrechnern immer auf den neuesten Stand. Außerdem kann von dem *BeamlineCalculator* Klasse der *Feedback Control* Optimierung die *timesequence* Klasse der coldmolLib verwendet werden. Zur Zeit wird mit einem Systemcall das externe Programm *coldmol_timesequence* gestartet. Dieser Schritt würde das System, um einen wesentlichen Faktor, stabiler machen.

Verwendung von mehreren Sokets zur Kommunikation.
Die Verwendung von mehreren Sockets würde die Synchronisation der Devicethreads vereinfachen. Momentan kann immer nur ein Controller oder Digitizer die Netzwerkschnittstelle zum VxWorks-Echtzeitsystem zu einem bestimmten Zeitpunkt nutzen. Wenn jeder Gerätetreiber seinen eigenen Socket hätte, würde sich das Betriebssystem um die effizienteste Ausnutzung der Netzwerkschnittstelle kümmern.

Fehlerspezifikation der Funktionen.
Die Klassen in KouDA besitzen alle ihre eigene *Exceptionklasse*. Jedoch wird nicht mit Sicherheit diese *Exception* von den Funktionen der Klasse geworfen. Somit kann es vorkommen, dass einige *Exceptions* nicht rechzeitig gefangen werden können und bis in das Hauptprogramm *main()* hinunterfallen. Durch eine strukturierte Fehlerspezifikation sind auf einen Blick der Funktionsdeklaration die möglichen Fehlerfälle einer Funktion ersichtlich.

Datenerfassung in mehrere Threads aufteilen.
Das VxWorks-Echtzeitsystem bietet die Möglichkeit beliebig viele Anfragen zu stellen, bevor ein *Digitizer* seinen Trigger erhält. In KouDA jedoch laufen alle *Digitizertreiber* in nur einem Thread und können bisher mit maximal $50Hz$ die Daten an der Hardware erfassen. Die Erfassung der Daten sollte auf zwei Threads pro *Device* Klasse aufgeteilt werden. Ein Thread würde sich um die Anfrage der Aufträge kümmern, während der andere Thread die Verarbeitung der Daten vornimmt.

Redesign der *Controllerstruktur*.
Viele der *Controller* Klassen haben redundante bzw. veraltete Funktionen, die von keiner Klasse mehr genutzt werden. Diese Funktionen sollten entweder angepasst oder ganz aus den Quellcode herausgenommen werden. Alle abgeleiteten *Controller* Klassen spezifizieren eine Hardware. Durch eine Strukturänderung, dass z.B. eine Delay-Generator Klasse als Funktionsschnittstelle für alle Delay-Generatoren deklariert währe, würde den Quellcode transparenter und strukturierter machen.

7. Danksagung

Während der Ausarbeitung der Diplomarbeit und im Verlauf des Studiums, haben meine Freundin Jeanette und meine Eltern mich in allen möglichen Situationen mit Tat und Rat unterstützt. Ich will mich auf diesem Weg sehr herzlich bei Ihnen Bedanken.

Ich danke Heinz Junkes, dass er mir die Diplomandenstelle vermittelt hat. Er hat, als mein Zweitbetreuer und Abteilungsleiter von PP&B, mir bei der Beschaffung von Arbeitsmaterialien geholfen und den Einstieg in die Arbeitsumgebung erleichtert.

Ebenso danke ich Jochen Küpper, der als Projektleiter alle meine Lösungsansätzte und Aussagen hinterfragt hat. Er hat mich so dazu ermutigt die Ansätze und Implementierungen zu überdenken. Er forderte viel selbstständiges Erarbeiten von Kenntnissen ab, wodurch ich das Verständnis ausweiten bzw. vertiefen konnte. Er hat mir mit seinen Erfahrungen als Dozent das benötigte Wissen der physikalischen Grundlagen erklärt und mir stehts bei Fragen zur Seite gestanden.

Mein Dank geht auch an Herrn Buchholz, der sich bereiterklärt hat mich als Erstbetreuer auf dem Weg der Ausarbeitung zu begleiten.

Des weiteren möchte ich den Mittgliedern der Projektgruppe “Deceleration and trapping of large (bio-)molecules” und allen anderen danken, die mich in irgendeiner Form bei der Erstellung dieser Diplomarbeit unterstützt haben.

A. Eidesstattliche Erklärung

Hiermit versichere ich, Alexander Stanik, die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Literaturen und Hilfsmittel benutzt zu haben.

Berlin, 17. August 2007

_____

Alexander Stanik

B. Inhalt der CD

Auf der CD ist folgendes verfügbar:

Diplomarbeit in PDF Format:
CD/Diplomarbeit.pdf

Quellcode von KouDA:
CD/src/*

Klassendiagramm in PNG Format:
CD/Klassendiagramm.png

C. Parseroptionen

C.1 Setzen der Parseroptionen

Der folgende Auszug zeigt das Setzen der Parseroptionen in die *eoParser* Klasse.

```
// set the options in parser
// ######      Evolution Engine            ######
_parser->setORcreateParam(_population,"popSize",
                          "Population Size",
                          'P', "Evolution Engine");
_parser->setORcreateParam(eoParamParamType(_selection.parameter().c_str()),
                          "selection",
                          "Selection: DetTour(T), StochTour(t), Roulette,
                          Ranking(p,e) or Sequential(ordered/unordered)",
                          'S', "Evolution Engine");
_parser->setORcreateParam(eoHowMany(_offspring), "nbOffspring",
                          "Nb of offspring (percentage or absolute)",
                          'O', "Evolution Engine");
_parser->setORcreateParam(eoParamParamType(_replacement.parameter().c_str()),
                          "replacement",
                          "Replacement: Comma, Plus or EPTour(T),
                          SSGAWorst, SSGADet(T), SSGAStoch(t)",
                          'R', "Evolution Engine");
```

Quellcode C.1: Setzen der *Evolution Engine* als Parseroptionen

```
// ######    Genotype Initialization   ######
_parser->setORcreateParam(individualSize, "vecSize",
                          "The number of variables", 'n',
                          "Genotype Initialization");
try{
    eoRealVectorBounds all_initBounds(lower_initBounds, upper_initBounds);
    _parser->setORcreateParam(all_initBounds,"initBounds",
                          "Bounds for initialization (MUST be bounded)",'B',
                          "Genotype Initialization");
} catch (const logic_error& e){
    throw EvolutionaryComputingWarning(
          string( "false statement of initial bounds: ") + e.what() );
}
_parser->getORcreateParam(init_sigmas, "vecSigmaInit",
                          "Initial value for Sigma(s)",
                          'V', "Genotype Initialization");
{
    QString strSigma;
    strSigma.setNum(_objectSigma);
    _parser->getORcreateParam(strSigma.toStdString(), "sigmaInit",
                          "Initial value for Sigmas (with a
                          '%' -> scaled by the range of each variable)",
                          's', "Genotype Initialization");
}
```

Quellcode C.2: Setzen der *Genotype Initialization* als Parseroptionen

```
// ######    Output                    ######
_parser->setORcreateParam(true, "printBestStat",
                          "Print Best/avg/stdev every gen.",
                          '\0', "Output");
_parser->setORcreateParam(true, "printPop",
                          "Print sorted pop. every gen.",
                          '\0', "Output");
```

Quellcode C.3: Setzen des *Output* als Parseroptionen

```
// ######    Output - Disk             ######
_parser->setORcreateParam(_outputFile, "resDir",
                          "Directory to store DISK outputs",
                          '\0', "Output - Disk");
_parser->setORcreateParam(true, "eraseDir",
                          "erase files in dirName if any",
                          '\0', "Output - Disk");
_parser->setORcreateParam(true, "fileBestStat",
                          "Output bes/avg/std to file",
                          '\0', "Output - Disk");
```

Quellcode C.4: Setzen des *Output - Disk* als Parseroptionen

```
// ######    Output - Graphical        ######
_parser->setORcreateParam(true, "plotBestStat",
                          "Plot Best/avg Stat",
                          '\0', "Output - Graphical");
_parser->setORcreateParam(true, "plotHisto",
                          "Plot histogram of fitnesses",
                          '\0', "Output - Graphical");
```

Quellcode C.5: Setzen des *Output - Graphical* als Parseroptionen

```
// ######    Persistence                ######
_parser->setORcreateParam(unsigned(1), "saveFrequency",
                          "Save every F generation
                          (0 = only final state, absent = never)",
                          '\0', "Persistence" );
_parser->setORcreateParam(unsigned(0), "saveTimeInterval",
                          "Save every T seconds (0 or absent = never)",
                          '\0',"Persistence" );
```

Quellcode C.6: Setzen der *Persistence* als Parseroptionen

```
// ######    Stopping criterion         ######
_parser->setORcreateParam(static_cast<unsigned long>(_maxEvaluation),
                          "maxEval",
                          "Maximum number of evaluations (0 = none)",
                          'E', "Stopping criterion");
_parser->setORcreateParam(unsigned(_maxGeneration), "maxGen",
                          "Maximum number of generations () = none)",
                          'G',"Stopping criterion");
_parser->setORcreateParam(unsigned(_maxGeneration), "steadyGen",
                          "Number of generations with no improvement",
                          's', "Stopping criterion");
_parser->setORcreateParam(unsigned(_maxGeneration), "minGen",
                          "Minimum number of generations",
                          'g', "Stopping criterion");
```

Quellcode C.7: Setzen der *Stopping criterion* als Parseroptionen

```
// ######    Variation Operators        ######
try{
    eoRealVectorBounds all_objectBounds( lower_objectBounds,
                                         upper_objectBounds );
    _parser->setORcreateParam(all_objectBounds, "objectBounds",
                          "Bounds for variables",
                          'B', "Variation Operators");
} catch (const logic_error& e){
    throw EvolutionaryComputingWarning(
          string( "false statement of object bounds: ") + e.what() );
}
_parser->setORcreateParam(string("SGA"), "operator",
                          "Description of the operator (SGA only now)",
                          'o', "Variation Operators");
_parser->setORcreateParam(_crossoverProbability, "pCross",
                          "Probability of Crossover",
                          'C', "Variation Operators" );
_parser->setORcreateParam(_mutationProbability, "pMut",
                          "Probability of Mutation",
                          'M', "Variation Operators" );
_parser->setORcreateParam(this->toString(_crossoverType), "crossType",
                          "Type of ES recombination (global or standard)",
                          'C', "Variation Operators");
_parser->setORcreateParam(this->toString(_crossoverRecombination),
                          "crossObj",
                          "Recombination of object variables
                          (discrete, intermediate or none)",
                          'O', "Variation Operators");
_parser->setORcreateParam(this->toString(_mutationRecombination),
                          "crossStdev",
                          "Recombination of mutation strategy parameters
                          (intermediate, discrete or none)",
                          'S', "Variation Operators");
```

Quellcode C.8: Setzen der *Variation Operators* als Parseroptionen

```
// ######    ES mutation parameters    ######
_parser->setORcreateParam(1.0, "TauLoc",
                          "Local Tau (before normalization)",
                          'l', "ES mutation parameters");
_parser->setORcreateParam(1.0, "TauGlob",
                          "Global Tau (before normalization)",
                          'g', "ES mutation parameters");
_parser->setORcreateParam(0.0873, "Beta",
                          "Beta",
                          'b', "ES mutation parameters");
```

Quellcode C.9: Setzen der *ES mutation parameters* als Parseroptionen

C.2 Beispiel einer Parserdatei

Der folgende Auszug zeigt eine Parserdatei, wie sie vom Programm mit Hilfe der Bibliothek erzeugt wurde. Darin sind alle Parameter zum jeweiligen Optimierungslauf angegeben.

```
######                  General    ######
--help=0                      # -h : Prints this message
--stopOnUnknownParam=1        # Stop if unkown param entered
--seed=1184762402             # -S : Random number seed

######    ES mutation parameters  ######
--TauLoc=1                    # -l : Local Tau (before normalization)
--TauGlob=1                   # -g : Global Tau (before normalization)
--Beta=0.0873                 # -b : Beta

######    Evolution Engine        ######
--popSize=10                  # -P : Population Size
--selection=DetTour(2)        # -S : Selection: DetTour(T), StochTour(t),
                                     Roulette, Ranking(p,e) or
                                     Sequential(ordered/unordered)
--nbOffspring=30              # -O : Nb of offspring (percentage or absolute)
--replacement=Plus            # -R : Replacement: Comma, Plus or EPTour(T),
                                     SSGAWorst, SSGADet(T), SSGAStoch(t)

######    Genotype Initialization ######
--vecSize=6                   # -n : The number of variables
--initBounds=[0.0003,0.0007];[0.0005,0.0015];
             [0.0005,0.0015];[0.002,0.004];
             [0.0003,0.0008];[0.0003,0.0008]
                              # -B : Bounds for initialization
                                     (MUST be bounded)
--vecSigmaInit=6 1e-05 1e-05 0.001
                 0.001 0.0001 0.0001
                              # -V : Initial value for Sigma(s)
--sigmaInit=2                 # -s : Initial value for Sigmas
                                     (with a '%' -> scaled by
                                     the range of each variable)

######    Output                  ######
--printBestStat=1             # Print Best/avg/stdev every gen.
--printPop=1                  # Print sorted pop. every gen.
--useEval=1                   # Use nb of eval. as counter (vs nb of gen.)
--useTime=1                   # Display time (s) every generation
```

```
######    Output - Disk           ######
--resDir=/home/stanik/superTest/bla
                          # Directory to store DISK outputs
--eraseDir=1              # erase files in dirName if any
--fileBestStat=1          # Output bes/avg/std to file

######    Output - Graphical      ######
--plotBestStat=1          # Plot Best/avg Stat
--plotHisto=1             # Plot histogram of fitnesses

######    Persistence             ######
--saveFrequency=1         # Save every F generation
                            (0 = only final state, absent = never)
--saveTimeInterval=0      # Save every T seconds
                            (0 or absent = never)
--status=KouDA_Feedback_Control.status
                          # Status file

######    Stopping criterion      ######
--maxEval=910             # -E : Maximum number of evaluations (0 = none)
--maxGen=30               # -G : Maximum number of generations () = none)
--minGen=30               # -g : Minimum number of generations
--steadyGen=30            # -s : Number of generations with no improvement
--CtrlC=0                 # -C : Terminate current generation upon Ctrl C

######    Variation Operators     ######
--objectBounds=[0,0.0015];[0,0.002];
               [0,0.005];[0.001,0.01];
               [0,0.001];[0,0.001]
                          # -B : Bounds for variables
--operator=SGA            # -o : Description of the operator (SGA only now)
--pCross=0.5              # -C : Probability of Crossover
--pMut=0.75               # -M : Probability of Mutation
--crossType=global        # -C : Type of ES recombination
                                 (global or standard)
--crossObj=discrete       # -O : Recombination of object variables
                                 (discrete, intermediate or none)
--crossStdev=intermediate # -S : Recombination of mutation
                                 strategy parameters
                                 (intermediate, discrete or none)
```

Quellcode C.10: Erzeugte Parserdatei der *EO Bibliothek*

D. Klassendiagramm

Literatur

[BS93] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evol. Comput.*, 1(1):1–23, 1993.

[BS07] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programmierung mit Qt4*. Addison-Wesley, 2007.

[BTK+06] Hendrick L Bethlem, Michael R Tarbutt, Jochen Küpper, David Carty, Kirstin Wohlfart, Ed A Hinds, and Gerard Meijer. Alternating gradient focusing and deceleration of polar molecules. *J. Phys. B*, 39(16):R263–R291, 2006.

[ES98] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, New York, 1998.

[ES03] A. E. Eiben and J. E. Smith. Introduction to evolutionary computing. http://www.cems.uwe.ac.uk/~jsmith/ecbook/ecbook.html, 2003.

[FHK+03] Manfred Fink, Rolf-Dieter Heuer, Hans Kleinpoppen, Klaus-Peter Lieb, Nikolaus Risch, and Peter Schmüser. *Lehrbuch der Experimentalphysik*. Walter de Gruyter, 2003.

[FK08a] Frank Filsinger and Jochen Küpper. Conformer-selection of (bio-) molecules. to be published, 2008.

[FK08b] Frank Filsinger and Jochen Küpper. Frequency stabilization of a commercial ring-dye-laser system. to be published, 2008.

[F.M] F.M.H.Crompvoets. Deceleration and trapping of polar molecules.

[FR98] David B. Fogel and Ingo Rechenberg. *Evolutionary Computation - The Fossil Record*. John Wiley and Sons Inc, 1998.

[GHJV04] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster*. Addison-Wesley, 2004.

[GKK04] Ingrid Gerdes, Frank Klawonn, and Rudolf Kruse. *Evolutionäre Algorithmen*. Vieweg, Wiesbaden, 2004.

[HMS02] Ekbert Hering, Rolf Martin, and Martin Stohrer. *Physik für Ingenieure*. Springer, 2002.

[KÖ5] Jochen Küpper. Lehrveranstaltung atom und molekülphysik 2. Vorlesungsmaterial, FU Berlin, 2005.

[Ker05] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

[KMRS01] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: Yet another evolutionary computation library? http://eodev.sourceforge.net, 2001.

[Kuh06] Carol Kuhlisch. Reengineering einer anzeige- und steuerungssoftware zu einer geschwindigkeitsoptimierten und multithreadfähigen anwendung. Diplomarbeit, TFH Berlin, 2006.

[lP] My life and Project. Tuxmind. http://www.tuxmind.altervista.org/?p=35.

[Lun] Lundberg. Needless to say, to save these images just click on the images and save it! http://academy.asd20.org/kadets/lundberg/dnapic.html.

[Nis97] V. Nissen. *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997.

[Rec07] Ingo Rechenberg. Lehrveranstaltung evolutionsstrategie 1. Vorlesungsmaterial, TU Berlin, 2007.

[Sau] Rob Saunders. The evolutionary design system. http://people.arch.usyd.edu.au/~rob/applets/house/House2.html.

[Sch95] H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, 1995.

[SF07] Alexander Stanik and Frank Filsinger. Data evaluation of feedback control optimization. Korrespondenz, Fritz-Haber-Institut, 2007.

[toE] Developement team of EO. Eo evolutionary computation framework. http://eodev.sourceforge.net.

[vdM05] Sebastiaan Y. T. van de Meerakker. *Deceleration and Electrostatic Trapping of OH Radicals*. Thesis Radboud Universiteit Nijmegen, 2005.

[WFG+07] Kirstin Wohlfart, Frank Filsinger, Fabian Grätz, Henrik Haak, Jochen Küpper, and Gerard Meijer. Decoupling of longitudinal and transverse motion in the stark decelerator using alternate gradient focusing. to be published, 2007.