Technische Universität Dortmund

Fakultät für Mathematik
Lehrstuhl III: Angewandte Mathematik und Numerik

Genetische Algorithmen

Wirtschaftsmathematisches Projekt zur Numerik im WS 2013/14

Qendresa Mehmeti, 133329
Vladimir Amrein, 136809
Marius Kulms, 145892
Nick Macin, 133518
Christian Loenser, 133286
Dmitri Bogonos, 133382
Manuel Waidhas, 136739

Betreuung:
Prof. Dr. Stefan Turek
Dr. Andriy Sokolov

19.05.2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

1. Einleitung

Zur Lösung von Optimierungsproblemen und zur Extremwertsuche gibt es neben den analytischen Lösungsmethoden auch zahlreiche Algorithmen für die numerische Näherung. Viele Verfahren sind auf gewisse Rahmenbedingungen spezialisiert, woraus natürlich eine starke Limitierung auf bestimmte Probleme aber auch das Erlangen einer gewissen Effizienz folgt. Beispielsweise kann das Newton-Raphson-Verfahren zur Bestimmung eines Extremums nur auf zweimal stetig differenzierbare Funktionen angewendet werden, stellt dafür aber auch eine schnelle, wenn auch nicht allzu robuste Lösungsmethode dar.

Mit der Zeit wurden immer mehr Verfahren entwickelt, die sich auf gewisse Problemklassen spezialisieren. Die relative Anzahl an universell einsetzbaren Methoden nahm in der Folge immer weiter ab. Letztere können auf ein breites Feld von Problemen angewendet werden und bieten eine gewisse Robustheit mit entsprechenden Einbußen bei der Effizienz. Doch was spricht für den Einsatz universeller und/oder robuster Verfahren?

Ist die „Gutartigkeit“ eines Problems unbekannt oder ungewiss, verflüchtigen sich die Vorteile spezialisierter Verfahren recht schnell. Verfügt ein Verfahren über eine geringe Resistenz gegenüber Störungen, kann der ursprüngliche Zeitvorteil in einen -nachteil umschlagen - robuste Verfahren bieten dann eine bessere Eignung und stellen damit eine Alternative dar. In der Realität sieht man sich häufig Problemen mit Rauschen, der Suche nach globalen Lösungen oder zeitabhängigen Optima konfrontiert. Für solche Probleme eignen sich universelle Verfahren und im Speziellen genetische Algorithmen.

Genetische Algorithmen gehören zu der Klasse der evolutionären Algorithmen, die zu den universellen Verfahren zählen. Ihr Vorteil ist nicht nur der mögliche Einsatz bei den oben genannten Problemstellungen, sondern auch die prinzipielle Anwendung auf Probleme ohne großes Wissen bezüglich deren Eigenschaften. Ihre Suche basiert auf einer Gruppe von Punkten des Definitionsbereichs, auch Population genannt, wodurch sie im Stande sind lokale Optima zu überwinden und dadurch eine Eignung für multimodale, nichtlineare und diskontinuierliche Optimierungsprobleme erlangen. Zudem bestechen sie durch die recht einfache und damit zeitsparende Implementierung.

Die offensichtlichen Qualifikationen scheinen interessant und vielfältig zu sein, doch wie sehen die Einsatzmöglichkeiten für derartige Algorithmen in der Realität aus und

in welchen Branchen werden sie heutzutage verwendet? Existieren Bereiche, in denen einzig genetische Algorithmen Möglichkeiten zur Lösungsfindung bieten und wie können die Einsatzmöglichkeiten auf weitere Problemstellungen ausgeweitet werden?

Diesem Themengebiet und der Beantwortung der aufgeführten Fragen widmet sich diese Arbeit. Sie befasst sich mit genetischen Algorithmen, deren Theorie und Einsatzmöglichkeiten, gibt anhand numerischer Tests Ratschläge zur Nutzung und gibt einen Ausblick auf die Anwendungen der genetischen Programmierung, beispielsweise zur Lösung von Systemen partieller Differentialgleichungen.

Die ersten beiden Kapitel führen genetische Algorithmen und deren Theorie ein, gefolgt von einer Untersuchung zur Findung von Richtlinien für Anwendung und Wahl der einzelnen Instrumente und Parameter. Ziel ist eine Hilfestellung für eine Implementierung zur Lösung unterschiedlicher Probleme. Kapitel 5 und 7 geben einen Überblick über Anwendungsmöglichkeiten in der Wirtschaft anhand expliziter Beispiele.

Zu der Klasse der evolutionären Algorithmen zählt auch die genetische Programmierung. Sie erlaubt die Anwendung von biologischen Evolutionsstrategien auf Problemstellungen unterschiedlicher Arten, beispielsweise zur Lösung von Systemen partieller Differentialgleichungen. Kapitel 6 gibt einen Ausblick über diese Anwendungsmöglichkeiten und deren Nutzen.

2. Einführung in genetische Algorithmen

Qendresa Mehmeti

Genetische Algorithmen fassen die natürliche Evolution als Optimierungsverfahren auf, bei dem eine Population von Individuen an Umweltbedingungen optimal angepasst wird. Dabei basieren sie auf Mechanismen der biologischen Evolution nach der Theorie des britischen Naturforschers Charles Robert Darwin (1809-1882). Ihren Ursprung haben sie in den Sechziger- und Siebzigerjahren des 20. Jahrhunderts - an der Universität von Michigan war es John H. Hollands Intention, ein artifizielles Softwaresystem zu entwickeln, welches zum einen dabei hilft den Prozess von natürlichen Systemen zu verstehen und gleichzeitig die Robustheit der Natur beibehält.

Genetische Algorithmen gehören der Klasse der evolutionären Algorithmen an. Dieser Begriff beschreibt Optimierungsverfahren, die von biologischen Paradigmen, wie beispielsweise bei Tieren beobachtete Verhaltensweisen oder der Genetik, inspiriert sind. Evolutionäre Algorithmen lassen sich nach [19] in vier voneinander unabhängig entwickelte Strömungen unterteilen:

- genetische Algorithmen
- evolutionäre Programmierung
- Evolutionsstrategien
- genetische Programmierung

Von der evolutionären Programmierung wurde erstmals in [7] berichtet. Das Ziel bestand darin, eine künstliche Intelligenz durch Simulation der Evolution als einen Lernprozess zu erschaffen. Die Evolutionsstrategien gehen zurück auf Ingo Rechenberg und Hans-Paul Schwefel [30]. Ursprünglich ging es um die Entwicklung von Verfahren für die experimentelle Optimierung von verschiedenen physikalischen Problemen, auf die keine gradientenbasierten Methoden angewendet werden konnten. Die Genetischen Programmierung handelt von der Erstellung optimaler Programme zur Berechnung von

gegebenen Eingabe-Ausgabe-Paaren. Vorreiter auf diesem Gebiet war John Koza (siehe z.B. [22], vgl. [1]).

2.1. Mechanismen der natürlichen Evolution

Wie bereits erwähnt, formalisieren genetische Algorithmen den Prozess der natürlichen Evolution für die Anwendung auf mathematische Optimierungsprobleme. Aus diesem Grund werden im Rahmen derartiger Verfahren Begriffe aus der biologischen Evolution und Genetik mit Begriffen aus der mathematischen Optimierung identifiziert. Im Verlauf dieser Arbeit werden dieses Begriffe oftmals als Synonyme verwendet, weshalb eine Gegenüberstellung der Begriffe und ihrer Bedeutungen notwendig erscheint. In der Literatur finden sich Teils widersprüchliche Angaben, weshalb nur die nachstehenden Begriffe für diese Arbeit von Relevanz sind. Sie beziehen sich zur Vereinfachung auf die Binärkodierung; die Bedeutungen lassen sich bei Bedarf auch auf andere Kodierungen übertragen.

biologischer Begriff	Bedeutung
Phänotyp	Erscheinungsbild; dekodierte Variable
Genotyp	Erbbild; kodierte Variable
Population	Gruppe von Suchpunkten
Individuum	Suchpunkt aus der Population
Chromosom	als String kodierter Suchpunkt
Gen	Bit
Allel	Wert eines Bits
Lokus	Bitposition
Fitness	Zielfunktionswert
Generation	Population zu einem Zeitpunkt
Selektion	Auswahl
Rekombination	Kreuzungsoperator
Mutation	Mutationsoperator

Tabelle 2.1.: Begriffserklärung

Zufällige Variationen im genetischen Code können zu vorteilhaften Eigenschaften führen, welche durch natürliche Auslese an künftige Generationen weitergereicht werden. Individuen einer Population passen sich somit an ihre Umweltbedingungen durch Selektion, Rekombination und Mutation an. Organismen, die an die Umweltbedingungen besser angepasst sind oder über günstige Eigenschaften verfügen, werden begün-

stigt, ihre Erbinformation an die Nachkommen weiterzureichen, was den Fortbestand der Art sichert. Individuen, die eine geringere Anpassung aufweisen, sterben mit höherer Wahrscheinlichkeit aus. Umwelteinflüsse und Fehler bei der Rekombination lassen Mutationen am Erbgut auftreten, welche zu einer verbesserten Anpassung führen können.

Diese Prozesse wenden genetische Algorithmen in Form von Operatoren auf Werte des Definitionsbereichs, mit dem Ziel einer optimalen Anpassung an das zugrundeliegende Problem, an. Die Anpassung von Generation zu Generation, also Iteration zu Iteration, erfolgt durch Änderungen der Erbinformation, der genotypen Darstellung der Suchpunkte. Die genetischen Operatoren lassen sich nicht auf Werte in phänotyper Darstellung anwenden. Das Verfahren betrachtet nicht jeweils nur einen Suchpunkt, sondern ein sogenanntes "Multiset" an Suchpunkten, welche eine Population bilden - in der Biologie: eine Gemeinschaft von Individuen. So werden Punkte aus dem Defintionsbereich, welche Teil der Population sind, als Individuen, Chromosomen, oder in der Binärkodierung als Strings bezeichnet. Ein Chromosom/String besteht in der genotypen Darstellung aus Genen/Bits, welche gewisse Ausprägungen (Allele) haben. In der Binärkodierung besteht folglich ein Chromosom/String aus einer festgelegten Anzahl an Genen/Bits, welche die Werte 0 oder 1 (Allele) annehmen können. Allele stellen somit die einzelnen Elemente des zugrundeliegenden Alphabets dar, welches bei der Kodierung der Suchpunkte definiert wird. Durch die wiederholte Anwendung der genetischen Operatoren auf die Population gelangen genetische Algorithmen zu einer Anpassung der Population an die entsprechenden Umweltbedingungen.

2.2. Aufbau genetischer Algorithmen

Der restliche Teil dieses Kapitels widmet sich dem Aufbau genetischer Algorithmen. Es soll eine klare Vorstellung des Ablaufs vermittelt und die unterschiedlichen, zur Verfügung stehenden Methoden erläutert werden (vgl. [33] und [10]).

Um überhaupt den gesamten Optimierungsprozess starten zu können, bedarf es einer Kodierung der Punkte des Definitionsraumes. Nach der Bestimmung einer Kodierung wird das Verfahren durch die Generierung einer Startpopulation initialisiert. Es folgt eine Ausführung der Prozesse Selektion, Rekombination und Mutation für jedem Iterationsschritt, bis ein Abbruchkriterium erreicht wird. Dieses wird oftmals als eine maximale Anzahl an Iterationen definiert. Ausgegeben wird dann ein Chromosom mit der höchsten Fitness aus der Population des letzten Iterationsschrittes. Bei der Suche nach einem Minimum, kann, wie bei Optimierungsproblemen üblich, die Zielfunktion

negiert oder Veränderungen am Algorithmus selbst vorgenommen werden.

Bevor auf die einzelnen Prozesse im Detail eingegangen wird, soll der Ablauf anhand einer Grafik 2.1 verdeutlicht werden:

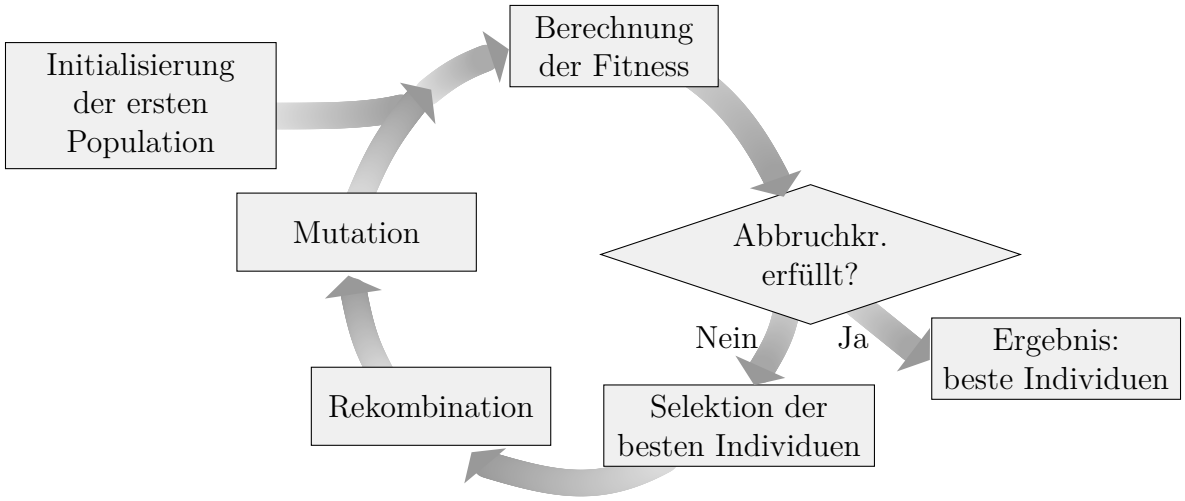


Abbildung 2.1.: Schematische Darstellung des Ablaufs
Quelle: A. Windisch [1]

2.2.1. Kodierung

Die Kodierung des Definitionsraumes ist problemspezifisch zu wählen. Abhängig von dem zugrundeliegenden Optimierungsproblem, kann eine Binärdarstellung oder beispielsweise eine Darstellung von Permutationen optimal sein. Letztere bietet sich beispielsweise bei der Suche nach kürzesten Wegen an, bei der jeder Wegpermutation durch die Fitnessfunktion die entsprechende Zeit zugeordnet wird.

Populär ist die Binärkodierung, welche u.a. für die Herleitung des Fundamentalsatzes der genetischen Algorithmen und auch in dieser Arbeit für die numerischen Tests verwendet wird. Mit einer festgelegten Bitlänge können recht einfach natürliche Zahlen dargestellt werden. Eine 5-Bit Kodierung ist in der Lage ganze Zahlen zwischen 0 und 31 darzustellen. Mithilfe von arithmetischen Operatoren können durch die Binärkodierung auch reelle Zahlen umgesetzt werden, was die Anwendungsmöglichkeiten deutlich erhöht. Eine genaue Erläuterung findet sich in Kapitel 5.

2.2.2. Initialisierung

Ein Durchlauf wird Initialisiert mit dem Erzeugen der Startpopulation. Dies geschieht zufallsbasiert und ist abhängig von der zugrundeliegenden Kodierung der Chromosomen.

Bei einer Populationsgröße von N müssen zwangsläufig N Chromosomen der Länge L in genotyper Darstellung erzeugt werden, auf die dann in jedem Iterationsschritt die nachfolgenden Operatoren angewendet werden.

2.2.3. Selektion

Jeder Iterationsschritt beginnt mit dem Prozess der Selektion. Aus der resultierenden Population des vorangegangenen Iterationsschrittes werden unter gewissen Kriterien Chromosomen für den nächsten Iterationsschritt ausgewählt. Die Gesamtanzahl der Chromosomen in der Population wird dabei unverändert gelassen, um die Unversehrtheit des grundlegenden Prinzips genetischer Algorithmen, der populationsbasierten Suche, zu gewährleisten. Allerdings ist es auch möglich die Populationsgröße mit jedem Iterationsschritt zu verkleinern. Welches Vorgehen gewählt wird bleibt dem Anwender überlassen.

Die Selektion dient der Verbesserung der durchschnittlichen Fitness der gesamten Population, durch die Auswahl der Chromosomen mit den höchsten Fitnesswerten. Sie ist somit das wichtigste Instrument für die Suche nach einer geeigneten Näherungslösung.

Es gibt mehrere verschiedene Selektionsvarianten - alle haben jedoch das gemeinsame Ziel der Auswahl der fitnessbezogen besten Chromosomen. Zu den bekannteren Varianten zählen die Roulette-Wheel-Selektion, die Tournament-Selektion und die Truncation-Selektion, welche nun näher erläutert werden sollen.

Roulette-Wheel-Selection

Die Roulette-Wheel-Selection, auch fitnessproportionale Selektion genannt, ist das bekannteste Verfahren zur Auswahl geeigneter Individuen. Diese Selektionsmethode war die erste, von John Holland zur Herleitung des Fundamentalsatzes genutzte Variante.

Jedem Individuum, also jedem Chromosom c_i mit $i = 1,...,N$ der Population der Größe N, wird eine Auswahlwahrscheinlichkeit zwischen 0 und 1 entsprechend dem Anteil an der Gesamtfitness zugeordnet. Dieser Anteil kann berechnet werden mit

$$\frac{f(c_i)}{\sum_{j=1}^{N} f(c_j)} \tag{2.1}$$

Mit einer höheren Wahrscheinlichkeit steigt auch die Chance für die nächste Generation, demnach den nächsten Iterationsschritt, ausgewählt zu werden. Nach der Bestimmung der Auswahlwahrscheinlichkeiten werden auf dem Intervall [0;1] allen Chromosomen Teilintervalle entsprechend ihren Wahrscheinlichkeiten p_{c_i} zugewiesen. An-

schließend werden gleichverteilt N Zufallszahlen auf dem Intervall [0;1] bestimmt, welche dann insgesamt N Chromosomen für die nächste Generation bestimmen. Hierbei kann ein Chromosom mehrfach (mehrere Kopien des Chromosoms) in die nächste Generation gelangen. Man kann sich diesen Vorgang mithilfe eines Rouletterads anschaulich erklären. Jedem Chromosom wird entsprechend seiner Auswahlwahrscheinlichkeit eine Fläche auf der Rad zugewiesen. Das Drehen des Rades startet das Zufallsexperiment; kommt das Rad zum Stillstand, wird das Chromosom ausgewählt auf dessen Fläche sich der Zeiger befindet.

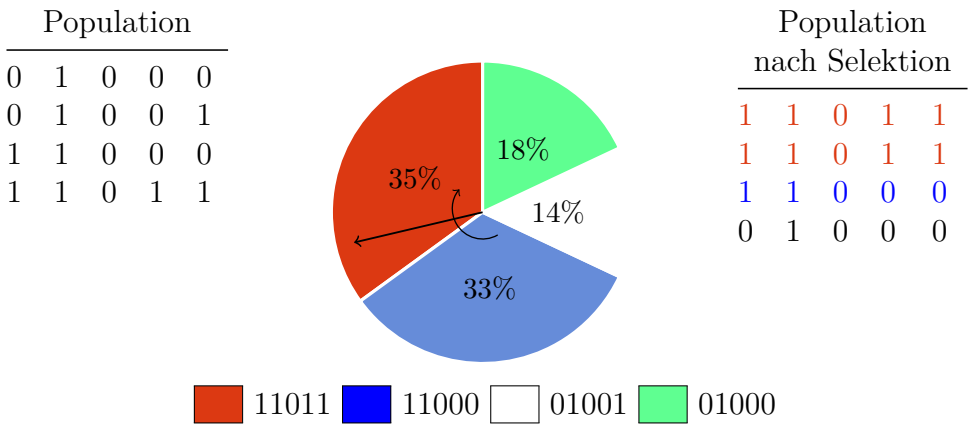


Abbildung 2.2.: Veranschaulichung der Roulette-Wheel-Selection

Bei der Roulette-Wheel-Selection besteht durchaus die Möglichkeit, wenn auch mit einer geringen Wahrscheinlichkeit, dass ausschließlich Chromosomen die mit den niedrigsten Fitnesswerten ausgewählt werden. Somit ist eine Selektion der fittesten Individuen unter dieser Selektionsmethode nicht garantiert.

Tournament Selection

Die Tournament-Selektion ist die nächste sehr bekannte Selektionsvariante. Wie der Name schon vermuten lässt, werden bei dieser Methode „Turniere“ abgehalten um die Individuen für die nächste Generation zu bestimmen. Für ein „Turnier“ werden mindestens zwei Chromosomen zufällig aus der aktuellen Population ausgewählt. Das Chromosom mit der höheren Fitness geht als Sieger aus dem „Turnier“ hervor und geht über in die neue Population.

Mehrere Implementierungsmöglichkeiten sind denkbar. Zum einen können bei einer Populationsgröße von N und Auswahl von jeweils zwei Chromosomen insgesamt N Turniere abgehalten werden. Hierbei können dann einige Chromosomen mehrfach in

die nächste Generation gelangen. Zum anderen können Turnierteilnehmer aus der Liste für nachfolgende Turniere gelöscht werden, was aber bei einer Wahl von zwei Chromosomen pro „Turnier“ in einer Populationsgröße von insgesamt $N/2$ Chromosomen für die nächste Generation resultiert. Kann hingegen jedes Chromosom zwei mal für „Turniere“ ausgewählt werden, führt dies zu einer gleichbleibenden Populationsgröße in den nachfolgenden Generationen.

Rank-based Selection

Als weitere nennenswerte Selektionsmethode kann die Truncation-Selection aufgeführt werden. Dem Grunde nach werden bei dieser Methode die stärksten Chromosomen für die nächste Generation ausgewählt. Diese Methode klingt einerseits sehr einfach, ist jedoch relativ zeitaufwendig. Jedem Chromosom wird ein Rang entsprechend seiner Fitness zugewiesen, was eine absteigende Sortierung aller Chromosomen nach ihrer Fitness erforderlich macht (Aufwand $N \log(N)$). Darauf folgt eine Definition der Wahrscheinlichkeitsverteilung der Rangskala; je geringer der Rang, umso höher die Wahrscheinlichkeit. Die Auswahl der Chromosomen für die nächste Generation erfolgt mithilfe eines Rouletterads anhand der Verteilung. Durch dieses Vorgehen kann das Dominanzproblem, bei dem der Fitnesswert die Auswahlwahrscheinlichkeit direkt beeinflusst, vermieden werden.

Elitist Selection

Bei dieser Form der Selektion wird eine gewisse, frei wählbare Anzahl von Chromosomen unverändert in die nächste Generation übernommen. Dies bedeutet, dass die Variationsoperatoren nicht auf diese, sondern auf die restlichen Chromosomen angewendet werden. Ersichtlich ist, dass dieses Verfahren eine deutliche Zeitersparnis gegenüber anderen Verfahren erwirtschaften kann, da sich der Selektionsprozess dem Grunde nach auf eine Sortierung der Chromosomen nach ihrer Fitness reduziert. Lokale Optima können aber schlechter überwunden werden.

Auf die selektierten Chromosomen können nun die beiden Variationsoperatoren Rekombination und Mutation angewendet werden. Diese gehören zu den sogenannten Zwei- bzw. Ein-Elter-Operatoren. An dieser Stelle sei darauf hingewiesen, dass auch Mehr-Elter-Operatoren, bei denen z.B. drei Chromosomen miteinander gekreuzt werden, möglich sind.

2.2.4. Rekombination

Beim Zwei-Elter-Operator Rekombination (engl. Crossover), findet eine Kreuzung zweier zufällig gewählter Chromsomen aus der Population an mindestens einem zufälligen Punkt statt. Die Rekombinationspunkte befinden sich zwischen den einzelnen Genen, demnach zwischen den einzelnen Bits in der Binärkodierung. Hinter dem Kreuzungspunkt erfolgt ein Tausch der nachfolgenden Abschnitte bzw. Bitfolgen der beiden Chromosomen. Dieser Operator ist somit stark abhängig von der Diversität der Population. Befinden sich alle Individuen der Startpopulation im Bereich eines lokalen Optimums, ist die Wahrscheinlichkeit eher gering, dass der Algorithmus diesen Bereich verlässt. Wie bei der Selektion gibt es viele unterschiedliche Rekombinationsvarianten. Im Folgenden wird auf 1-Punkt-, 2-Punkt- und uniformen Crossover eingegangen. Zu den weiteren Varianten zählen unter anderem der Cut and Splice Crossover und der Shuffle Crossover.

1-Punkt Crossover

Beim 1-Punkt Crossover, oder auch 1X-Crossover, erfolgt eine Rekombination an einem zufällig gewählten Punkt zwischen den einzelnen Bits. Ein Chromosom der Länge L bietet $L-1$ mögliche Kreuzungspunkte. Die Wahl der Schnittpunktes ist zufallsbasiert und wird auf die beiden zufällig gewählten Chromosomen aus der Population angewndet. Ab diesem Punkt erfolgt ein Austausch der Gensequenzen. Auf die Auswahl des Rekombinationspunktes können unterschiedliche Wahrscheinlichkeitsverteilungen angewendet werden (z.B. die Gleichverteilung oder auch die Normalverteilung).

2-Punkt Crossover

Im Gegensatz zum 1-Punkt Crossover werden beim 2-Punkt Crossover zwei Kreuzungspunkte für die Rekombination gewählt. Der Austausch der Gensequenzen erfolgt zwischen den beiden Kreuzungspunkten.

Uniformer Crossover

Die Wahl der Anzahl der Rekombinationspunkte kann weitergeführt werden bis zum L-Punkt Crossover, bei dem jedes zweite Gen getauscht wird. Daraus entstand die Idee des uniformen Crossover, bei dem mit einer Wahrscheinlichkeit p_x für jedes Gen entschieden wird, ob es getauscht wird oder nicht.

2.2.5. Mutation

Ergänzend zur Rekombination kann der Variationsoperator Mutation betrachtet werden. Mit diesem Operator können geringfügige Änderungen an den Chromosomen vorgenommen werden mit dem Ziel einer stichprobenartigen Erkundung des Suchraumes. Dies verleiht die Möglichkeit, unabhängig von der erzeugten Startpopulation, neue Chromosomen erzeugen zu können. Abhängig von der gewählten Kopdierung wird bei diesem Prozess ein Gen mit einer festgelegten, frei wählbaren Wahrscheinlichkeit p_m geändert (mutiert). In der Binärkodierung erfolgt ein einfacher Allel- bzw. Bitwechsel von einer 0 zu einer 1 und umgekehrt. In der Praxis wird für Bitstrings der Länge L eine Mutationswahrscheinlichkeit von $p_m = 1/L$ gewählt. Zur Verbesserung der Konvergenz gibt es außerdem die Idee der Verwendung von steigenden Mutationswahrscheinlichkeiten für rechtsliegende Bits. Diese Technik soll ein Phänomen umgehen, bei dem rechtsstehende Bits von der Rekombination unbeeinflusst bleiben.

Anstelle der gewöhnlichen Standardmutation können auch die Allele von zwei Genen innerhalb des Chromosoms vertauscht werden (Zweiertausch). Denkbar wäre auch eine Verschiebung, Permutierung oder Invertierung eines zufällig gewählten Teilstücks eines zufällig gewählten Chromosoms. Letztendlich ist es dem Anwender überlassen, welche Möglichkeit er für die Mutation wählt.

2.2.6. Beispiel Variationsoperatoren

Der Ablauf und das Zusammenspiel der Variationsoperatoren sollen nun an einem einfachen Beispiel in der Tabelle 2.2 verdeutlicht werden. Zu sehen sind der 1-Punkt Crossover sowie die Standardmutation.

Population nach Selektion	Rekombination	Population nach Rekombination	Population nach Mutation
1 1 0 1 1	1 1 | 0 1 1	1 1 0 1 0	1 1 0 1 0
1 1 0 1 0	1 1 | 0 1 0	1 1 0 1 1	1 1 1 1 1
1 1 0 0 0	1 | 1 0 0 0	1 1 0 0 1	1 1 0 0 1
0 1 0 0 1	0 | 1 0 0 1	0 1 0 0 0	1 1 0 0 0

Tabelle 2.2.: Beispiel Variationsoperatoren

Die Populationsgröße beträgt $N = 4$ und die Bitlänge $L = 5$. Somit kann die Population in jeder Iteration als eine 4x5-Matrix aus Nullen und Einsen dargestellt werden. Für die Rekombination werden zwei Paare gebildet und jeweils ein Kreuzungspunkt für

jedes Paar ausgewählt. Anschließend werden mit der Mutationswahrscheinlichkeit Bits durch die Mutation verändert, womit der Iterationsschritt beendet wird. Diese letzte Matrix repräsentiert die Population, auf die im nächsten Iterationsschritt wieder Selektion, Rekombination und Mutation angewendet werden.

3. Das Schema-Theorem

Vladimir Amrein

Nachdem nun die Grundlagen für das Verständnis des Aufbaus genetischer Algorithmen gelegt worden sind, stellt sich die Frage nach der Funktionsweise dieser. Wie gelangt ein derartiger Algorithmus zu einem hinreichend guten Ergebnis? Dieser Thematik nähert sich dieses Kapitel unter Zuhilfenahme von Hollands Schema-Theorem an. Es wurde im Jahre 1975 von John Holland [14] (vgl. auch [33] und [10]) und seinen Studenten hergeleitet und legte das mathematische Fundament für genetische Algorithmen . Das Theorem leitet seine Aussagen aus der Betrachtung der Vervielfältigung von Chromosomenschemata über Generationen hinweg her, wobei ein Schema als eine Klasse von Chromosomen betrachtet werden kann. Die Zusammenfassung von Chromosomen zu Schemata steht in Verbindung mit der Notwendigkeit einer populationsübergreifenden Betrachtung, da eine Verfolgung der Entwicklung einzelner Chromosomen wenig sinnvoll erscheint. Vereinfachend werden in der Herleitung ausschließlich fitnessproportionale Selektion, 1-Punkt Crossover und bitweise Mutation betrachtet. Zum besseren Verständnis erfolgt ergänzend eine anschauliche Darstellung der Implikationen des Theorems mittels Hyperwürfeln.

3.1. Herleitung

Für die Herleitung des Theorems sind einige grundlegende Definitionen notwendig, welche im Folgenden aufgeführt werden.

Definition 3.1 (Schema)
Ein Schema H ist ein Bitstring der Länge L über das erweiterte binäre Alphabet $\{0, 1, *\}$, dessen Bits nicht ausschließlich mit $*$ besetzt sind $H \in \{0, 1, *\}^L \backslash \{*\}^L$.

Dabei dient $*$ als Platzhalter, auch „Don´t Care“-Symbol genannt, für eine 0 oder eine 1 im Zeichen $* \in \{0, 1\}$, um einen Satz von Lösungen darstellen zu können. Demnach wird beispielsweise das Chromosom 01101 u.a. durch das Schema $01*0*$ vertreten. Insgesamt existieren $3^L - 1$ denkbare Schemata, da jede Bitposition drei Zustände annehmen

kann und das suchraumaufspannende Schema $\{*\}^L$ per Defintion nicht als solches gewertet wird.

Definition 3.2 (Ordnung)
Die Ordnung $o(H)$ eines Schemas H gibt die Anzahl der von $*$ verschienen Einträge wider

Somit hat das obige Schema $H = 01*0*$ die Ordnung $o(H) = 3$. Mit der Einführung der Ordnung für Schemata lässt sich folgern, dass ein Schema insgesamt $2^{1-o(H)}$ unterschiedliche Chromosomen darstellen kann. Außerdem gilt:

$$\sum_{i=1}^{L} 2^i \binom{L}{i} = 3^L - 1$$

denn für die Anzahl von Schemata der Ordnung i gilt:

$$2^i \binom{L}{i}$$

Dies ist einleuchtend, da es für ein Schema i-ter Ordnung $\binom{L}{i}$ Möglichkeiten gibt i Bits festzusetzen und 2^i Möglichkeiten diesen Werte zuzuweisen. Ergo haben 2^L der $3^L - 1$ Schemata volle Ordnung L und stellen die einzelnen Lösungen des Suchraums dar.

Definition 3.3 (definierende Länge)
Die definierende Länge eines Schemas H ist die Differenz zwischen dem letzten und ersten mit 0 oder 1 besetzten Lokus.

Obiges Schema $H = 01*0*$ hat folglich die definierende Länge $d(H) = 4 - 1 = 3$.

Zur Veranschaulichung lassen sich Schemata mithilfe von Hyperebenen in Hyperwürfeln darstellen. Betrachtet man einen Bitstring der Länge 3, so enthält der dazugehörige Suchraum $2^3 = 8$ denkbare Lösungen. Dieser Suchraum lässt sich mit einem Würfel visualisieren, wobei jeder Ecke eine Lösung zugewiesen wird. Ein Schema stellt dann eine Hyperebene des Würfels dar; so beschreibt das Schema $0**$ zum Beispiel die Frontebene des Würfels (siehe Abbildung 3.1). In einer derartigen Darstellung von Bits der Länge 3 stellen Schemata voller Ordnung Ecken, Schemata der Ordnung 2 Kanten und Schemata der Ordnung 1 Ebenen des Würfels dar. Es ist ersichtlich, dass jede Lösung, folglich jedes Chromosom, zu $2^L - 1$ Schemata bzw. Hyperebenen passt, denn jede Lösung gehört, bezogen auf das Beispiel, zu drei Ebenen, drei Kanten und einer Ecke des Würfels ($2^3 - 1 = 7 = 3 + 3 + 1$). Oder allgemein:

$$\sum_{i=1}^{L} \binom{L}{i} = 2^L - 1$$

Somit tragen die erzeugten Chromosomen in der Startpopulation Informationen zu einer Vielzahl an Hyperebenen. Es ist plausibel, dass wesentlich mehr Hyperebenen als Chromosomen während des Suchprozesses durchlaufen werden, woraus sich die Hypothese des impliziten Parallelismus folgern lässt.

Dank der Verwendung eines binären Alphabets verdoppelt sich die Anzahl der Lösungen für jedes zusätzlich kodierte Bit im String, was bei einem Bitstring der Länge 4 zu insgesamt $2^4 = 16$, also doppelt so vielen möglichen Lösungen führt. Der resultierende Suchraum lässt sich ohne großen Aufwand durch zwei ineinander geschachtelte Würfel anschaulich wie folgt darstellen:

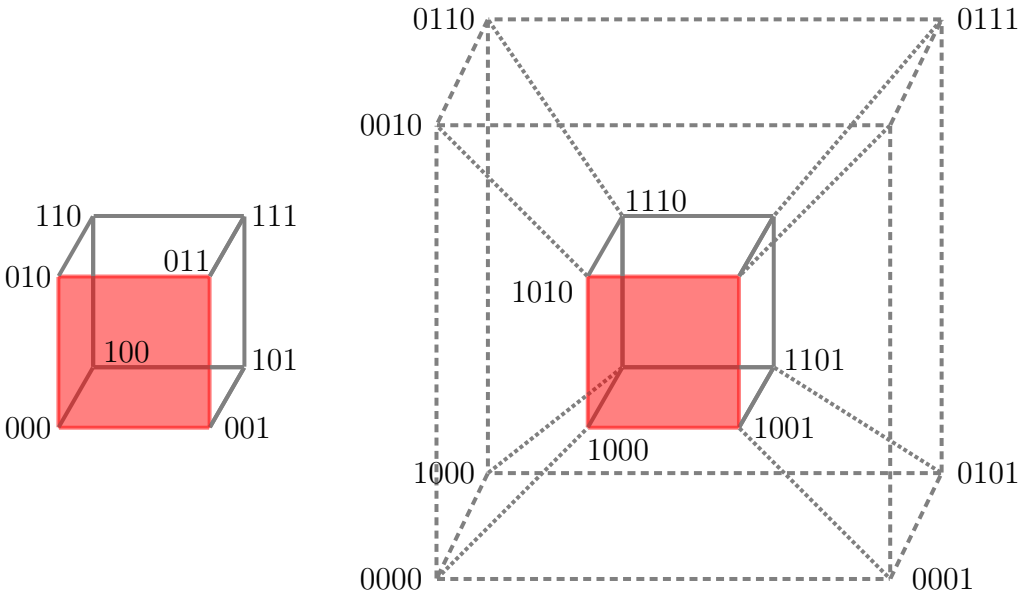


Abbildung 3.1.: Anschauliche Darstellung des Suchraumes mit Hyperwürfeln
Quelle: Whitley [33]

Es erfolgt eine Erweiterung des dreidimensionalen Falls, indem der äußere Würfel das Präbit 1 und der innere das Präbit 0 erhält. Jeder Würfel stellt weiterhin acht Lösungen des Suchraums dar. Die vier zuvor betrachteten Lösungen des Schemas $0 * *$ werden nun in der 4-Bit Kodierung durch das Schema $10 * *$ beschrieben, gekennzeichnet durch die rote Fläche in Abbildung 3.1. Im späteren Verlauf dieses Kapitels soll das Schema-Theorem anhand von Hyperebenen im Hyperwürfel klar machen, wie ein genetischer

Algorithmus den Suchraum durchläuft.

Nachdem nun die grundlegenden Definitionen und auch Schemata anschaulich dargestellt worden sind, widmet sich der restliche Teil dieses Abschnitts der Herleitung des grundlegenden Theorems, beginnend mit der Formalisierung des Selektionsprozesses. Anschließend wird dieser um Crossover und Mutation erweitert, woraus dann das endgültige Theorem resultiert.

Sei $f(c_i;t)$ die Fitness des Chromosoms $c_i \in \{0;1\}^L$ in der Generation t; weiterhin $\bar{f}(t) = \frac{1}{N}\sum_{i=1}^{N} f(c_i)$ die mittlere Fitness der Population und $m(c_i;t)$ die Anzahl der Auftritte von c_i in der Population. Die Auswahlwahrscheinlichkeit eines Chromosoms c_i während der fitnessproportionalen Selektion, auch als Überlebenswahrscheinlichkeit bezeichnet, ist definiert durch:

$$p(c_i) = \frac{f(c_i)}{\sum_{j=1}^{n} f(c_j)} = \frac{f(c_i)}{N\bar{f}(t)}$$

Es lässt sich nun die erwartete Anzahl der Kopien von c_i in der Population nach der Selektion beschreiben:

$$E[m(c_i, t+\Delta t_s)] = \frac{f(c_i;t)}{\bar{f}(t)} m(c_i;t)$$

$t+\Delta t_s$ beschreibt den Zeitpunkt unmittelbar nach der Selektion; analog beziehen sich im späteren Verlauf Δt_c auf den Rekombinations– und Δt_m auf den Mutationsprozess. Aus der Gleichung wird klar, dass Chromosomen mit einer überdurchschnittlichen Fitness und/oder höherem $m(c_i;t)$ häufiger in der nächsten Generation vertreten sein werden. Dies wird durch folgende Proposition bestätigt:

Proposition 3.4 (Vervielfältigung)
Hat ein Chromosom c_i eine überdurchschnittliche Fitness, so vervielfältigt es sich in den nachfolgenden Generationen annähernd exponentiell

Beweis
Sei $f(c_i;t) = (1+\delta)\bar{f}(t)$ mit $\delta > 0$. Dann folgt:

$$E[m(c_i, t+\Delta t_s)] = \frac{f(c_i;t)}{\bar{f}(t)} m(c_i;t) = \frac{(1+\delta)\bar{f}(t)}{\bar{f}(t)} m(c_i;t) = (1+\delta)m(c_i;t)$$

Ist die Populationsgröße N ausreichend groß, so ist $m(c_i;t)$ relativ klein und δ

approximativ konstant. Dann gilt:

$$m(c_i; t) = (1 + \delta)^t m(c_i; 0)$$

□

Die letzte Gleichung impliziert ein annähernd exponentielles Wachstum der Anzahl der Kopien von c_i. Analog gilt dies auch für die Sterberate von Chromosomen unterdurchschnittlicher Fitness. Die Ausbreitung der Chromosomen ist jedoch begrenzt, da mit fortlaufen des Algorithmus die durchschnittliche Fitness $\bar{f}(t)$ durch die Dominanz fitterer Chromosomen in der Population zunimmt. Durch diese Entwicklung sinkt der Selektionsdruck, dem diese unterworfen sind. Darunter ist eine abnehmende Aussterbewahrscheinlichkeit für (erfolgreichere) Chromosomen und Schemata zu verstehen.

Sei $m(H, t)$ die Anzahl der Vertreter des Schemas H in der Population c_1, c_2, …, c_n der Generation t. Die mittlere Fitness des Schemas H für alle c_i, die durch H vertreten werden, lautet:

$$\bar{f}(H, t) = \frac{1}{m(H, t)} \sum_{c_i \in H} m(c_i; t) f(c_i, t)$$

Damit lässt sich die hergeleitete Formel (3.1) für die Anzahl der Kopien von Chromosomen auch auf Schemata anwenden:

$$E[m(H, t + \Delta t_s)] = \frac{\bar{f}(H; t)}{\bar{f}(t)} m(H; t)$$

Sie beschreibt die erwartete Anzahl der Vertreter von H in der Population nach der Selektion. Chromosomen mit überdurchschnittlicher Fitness haben einen positiven Einfluss auf $\bar{f}(H; t)$, wodurch eine Präsenz des Schemas in der nächsten Generation wahrscheinlicher ist. $\frac{\bar{f}(H;t)}{\bar{f}(t)}$ ist der Selektionsdruck, welcher auf dem Schema H lastet.

Mit dem Abschluss der Formalisierung des Selektionsprozesses ist der Grundstein für das Verständnis der Funktionsweise genetischer Algorithmen gelegt worden. Es folgt eine Erweiterung um die beiden Variationsoperatoren: Rekombination und Mutation. Es sei darauf hingewiesen, dass eine Verwendung dieser beiden Prozesse für das Erreichen einer Näherungslösung nicht zwingend erforderlich, aber oftmals ratsam ist. Schließlich können nur durch Variationsmethoden neue Punkte im Suchraum erschlossen werden; eine Fokussierung alleine auf die Selektion führt zu einer starken Abhängigkeit von der zufallserzeugten Startpopulation.

Für die Rekombination oder auch den Crossover wird zuerst die Auswahlwahrscheinlichkeit p_c eingeführt — je nach Wahl dieses Wertes müssen nicht alle Chromosomen in diesen Prozess miteinbezogen werden. Schemata der Länge L bieten insgesamt $L-1$ mögliche Rekombinationspunkte für 1-Punkt Crossover; eine Rekombination an $d(H)$ dieser Stellen kann das zugrundeliegende Schema stören. In der Folge können dann im Schema festgelegte Genmuster nicht mehr an die Nachkommen weitergereicht werden, wodurch ein aussterben des Schemas möglich ist. Zu beachten ist, dass eine Rekombination an den kritischen Stellen zu einer Störung führen kann, es jedoch nicht zwingend muss. Betrachtet man beispielsweise das Schema $H = 11 * * * *$ mit $L = 6$ und $o(H) = 2 - 1 = 1$, so führt eine Rekombination von $111010 \in H$ mit $010000 \notin H$ zwischen den ersten beiden Bits zu keinem Verlust für H, obwohl der Kreuzungspunkt innerhalb der kritischen Zone liegt. Eine Rekombination mit $100110 \notin H$ stört hingegen das Schema und führt zu einer Auslöschung des Genmusters 11 an den Loki 1 und 2.

Es ist auch möglich, dass eine Rekombination von Chromosomen anderer Schemata zu Nachkommen führt, die dem betrachteten Schema H entsprechen. Somit können im gesamten Crossoverprozess Passungen verloren und hinzugewonnen werden. Die folgende Definition hält dies fest:

Definition 3.5 (Passungen während der Rekombination)
p_{loss} beschreibt die Wahrscheinlichkeit, dass durch Rekombination die Passung eines Chromosoms zum Schema H verloren geht. $gains$ hingegen gibt gewinne an zu H passenden Chromosomen wider

Die erwartete Anzahl der Vertreter von H nach Selektion lässt sich nun um die Rekombination erweitern:

$$\begin{aligned} E[m(H, t + \Delta t_s + \Delta t_c)] = (1 - p_c) E[m(H, t + \Delta t_s)] \\ + p_c E[m(H, t + \Delta t_s)](1 - p_{loss}) \\ + gains \end{aligned}$$

Die rechte Seite der Gleichung besteht nun aus drei Summanden. Der erste beschreibt Chromosomen, die nicht von Crossover betroffen sind; diese werden in die nächste Generation direkt übernommen. Der zweite Summand beschreibt Chromosomen, die am Crossover teilnehmen, wodurch eventuell die Passung zum Schema verloren gehen kann. Ergänzt wird diese Berechnung um die Zugewinne an passenden Chromosomen.

Mit der Erkenntnis, dass die Passung verloren gehen kann, wenn ein zu H passendes Chromosom mit einem zu H nicht passenden Chromosom im kritischen Bereich $\frac{d(H)}{L-1}$

gekreuzt wird, lässt sich p_{loss} unter der Annahme einer strikten störenden Auswirkung abschätzen:

$$p_{loss} \leq \frac{d(H)}{L-1}\left(1 - \frac{E[m(H, t+\Delta t_s)]}{N}\right)$$

Durch das einsetzen dieser Abschätzung und ignorieren von *gains* kann die Anzahl der Vertreter weiter ausformuliert werden:

$$\begin{aligned}
E[m(H, t+\Delta t_s+\Delta t_c)] &\geq (1-p_c)E[m(H, t+\Delta t_s)] \\
&\quad + p_c E[m(H, t+\Delta t_s)]\left(1 - \frac{d(H)}{L-1}\left(1 - \frac{E[m(H, t+\Delta t_s)]}{N}\right)\right) \\
&= E[m(H, t+\Delta t_s)]\left(1 - p_c + p_c\left(1 - \frac{d(H)}{L-1}\left(1 - \frac{E[m(H, t+\Delta t_s)]}{N}\right)\right)\right) \\
&= E[m(H, t+\Delta t_s)]\left(1 - p_c\frac{d(H)}{L-1}\left(1 - \frac{E[m(H, t+\Delta t_s)]}{N}\right)\right) \\
&= \frac{\bar{f}(H,t)}{\bar{f}(t)}m(H,t)\left(1 - p_c\frac{d(H)}{L-1}\left(1 - \frac{\bar{f}(H,t)}{N\bar{f}(t)}m(H,t)\right)\right) \qquad (3.1)
\end{aligned}$$

Bevor die Herleitung ihren Abschluss findet, muss noch der letzte Prozess, die Mutation, in die Formalisierung integriert werden. Es erfolgt eine bitweise Anwendung der Mutation mit der Mutationswahrscheinlichkeit p_m auf die Schemata. Wird ein Bit für die Mutation ausgewählt, verursacht dies einen Wechsel des Allels, auch Bit-Flip genannt. Die Wahrscheinlichkeit, dass dabei ein Schema gestört wird beträgt:

$$(1-p_m)^{o(H)} \qquad (3.2)$$

$1-p_m$ gleicht der Wahrscheinlichkeit eines im Schema festgesetzten Bits, die Mutation zu überstehen; $o(H)$ ist die Ordnung des Schemas H und gibt die Anzahl festgesetzten Bits eines Schemas an.

Mit der Mutation findet die Herleitung des Theorems ihren Abschluss:

Theorem 3.6 (Holland)
Die erwartete Anzahl der Vertreter des Schemas H nach Selektion, Rekombination und Mutation in Periode t kann abgeschätzt werden durch

$$E[m(H, t+1)] \geq \frac{\bar{f}(H,t)}{\bar{f}(t)}m(H,t)\left(1 - p_c\frac{d(H)}{L-1}\left(1 - \frac{\bar{f}(H,t)}{N\bar{f}(t)}m(H,t)\right)\right)(1-p_m)^{o(H)}$$

Das Theorem folgt aus der Multiplikation der Mutationswahrscheinlichkeit für Schemata (3.2) mit der rechten Seite der Ungleichung aus der Herleitung des Rekombinationsprozesses (3.1), welche Selektion und Rekombination formal darstellt.

3.2. Implikationen

Aus der Herleitung und dem Theorem selbst ergeben sich einige unmittelbare Implikationen, welche in diesem Abschnitt erläutert werden sollen. Ersichtlich ist, dass ein Schema in der nachfolgenden Generation häufiger vertreten sein wird, wenn es eine überdurchschnittliche Fitness aufweist, von geringer Ordnung ist und eine kurze definierende Länge besitzt. Ein derartiges Schema wird im Selektionsprozess begünstigt und bietet wenige Möglichkeiten für Störungen durch Variationsmethoden, was zu einer Ausbreitung in der Population führt. Zu beachten ist aber auch die Möglichkeit einer überdurchschnittlich hohen Vertretung in der zufallserzeugten Startpopulation.

Schemata mit den eben beschriebenen Eigenschaften können als „Bausteine“ (engl. „building blocks“) bezeichnet werden, da sie durch Rekombination ihre charakteristischen Genkombinationen an ihre Nachkommen weiterreichen und somit zur Eingrenzung des Suchraums beitragen. Je größer die Startpopulation gewählt wird, umso wahrscheinlicher ist das Vorhandensein guter Genkombinationen für den raschen Aufbau guter Lösungen. Bezogen auf das Hyperwürfelmodell stellen „Bausteine“ Hypereben dar, welche eine Vielzahl an Ecken abdecken und die Suche durch Selektion, Rekombination und Mutation auf bestimmte Areale des Hyperwürfels verlagern, vorzugsweise auf diejenigen mit den fitnessbezogen besten Lösungen. Im Verlauf des Algorithmus nimmt die Anzahl der Schemata nach und nach ab und die durchschnittliche Fitness nach und nach zu. Die Suche konzentriert sich dann auf einige bestimmte Areale des Suchraums, bzw. des Hyperwürfels. Ob sich in diesen Arealen die optimale Lösung befindet ist zwar unklar, jedoch ist die Chance bei mehreren separaten Suchorten größer als bei einem einzelnen. Unter Einsatz hoher Mutationswahrscheinlichkeiten und aggressiver Rekombinationsmethoden, insbesondere dem uniformen Crossover, können zwar recht schnell weite Teile des Suchraums erschlossen werden, bei höherer Iterationsanzahl können sich jedoch diese wegen störender Auswirkungen auf Schemata durchaus kontraproduktiv auswirken. Dieser Umstand verlangt eine sorgfältige Abwägung der Selektionsmethode, der Variationsoperatoren und frei wählbaren Parameter, wie Populationsgröße und Iterationsanzahl, in Abhängigkeit des zu Lösenden Optimierungsproblems.

3.3. Kritik am Schema-Theorem

Einige Kritikpunkte schmälern die Aussagekraft des Schema-Theorems und dessen Implikationen ein. Zum einen erfolgt eine Einschränkung durch einige Annahmen zur Vereinfachung - welches Verhalten der Algorithmus bei Wahl anderer Methoden offenbart, wird nicht ersichtlich. Zum anderen erfolgt eine Abschätzung von p_{loss} und $gains$ wird gänzlich außer Acht gelassen. In der Praxis ist somit eine Abschätzung der Vertreter eines Schemas in einer bestimmten Generation häufig fehlerhaft, wodurch ein realer Einsatz nur in den ersten Generationen sinnvoll erscheint. In höheren Generationen wird die Abschätzung Zusehens schlechter. Eine exakte Berechnung von $E[m(H, t+1)]$ ist zwar möglich, jedoch relativ aufwändig.

Folglich kann die „Baustein"-Hypothese nur als Erklärung für die ersten Iterationen herangezogen werden, denn mit Fortlaufen des Algorithmus erfolgt eine präzisere Eingrenzung des Suchraums, wodurch $o(H)$ und $d(H)$ ansteigen. Bei höherer Ordnung und definierender Länge steigt allerdings die Wahrscheinlichkeit für Störungen, die Variationsmethoden an Schemata verursachen können. Die Hypothese ist nur bei hoher Populationsgröße und niedriger Iterationsanzahl sinngemäß.

Die durchschnittliche Fitness eines Schemas H kann sich in wenigen Generationen drastisch ändern, falls eine Konzentration auf einen bestimmten Bereich des Suchraums stattfindet. Somit ist auch die Betrachtung der durchschnittlichen Fitness nur in den ersten Iterationsschritten relevant. Danach entwickelt sich eine Abhängigkeit von anderen Schemata, welche vom Theorem gänzlich ignoriert wird. Außerdem sinkt der Selektionsdruck mit Zunahme der durchschnittlichen Fitness und Abnahme der absoluten Anzahl der Schemata, resultierend in einer Abnahme der impliziten Parallelisierung. Mit der Möglichkeit der Durchführung einer Skalierung der Fitnesswerte, z.B. nach Goldberg [12] oder der Wahl einer alternativen Selektionsmethode, lässt sich dieser Kritikpunkt allerdings ausmerzen.

Schlussendlich bleibt die Erkenntnis, dass der Erklärungsansatz des Schema-Theorems zum völligen Verständnis leider nicht ausreichend ist. In der Realität scheitert das Theorem an der Komplexität, der dynamischen Natur und der modularen Struktur genetischer Algorithmen. Für die Wahl der einzelnen verfügbaren Instrumente gibt es zudem keine geschlossene Theorie; hier ist man auf Erfahrungswerte aus der Literatur oder eigene Experimente und Tests angewiesen. Dies gilt ebenso für die Konvergenzdauer, die im Allgemeinen nicht fundiert ist. Trotz allem gelingt mit dem Theorem die Vermittlung einer Idee der Funktionsweise, welche als Hilfreich angesehen werden darf.

4. Implementierung und numerische Tests

Manuel Waidhas

Wie im vorangehenden Kapitel bereits dargestellt, gibt es keine geschlossene Theorie für einen optimalen genetischen Algorithmus, weshalb man für Vergleiche zwischen den einzelnen frei wählbaren Operatoren auf Ergebnisse aus der Literatur oder auf eigene Tests angewiesen ist. In diesem Kapitel wird dem Folge geleistet. Es werden mehrere Tests präsentiert mit dem Ziel einer Hilfestellung für die Wahl der verfügbaren Methoden. Zudem sollen Richtlinien für die Wahl der freien Parameter erarbeitet werden.

Vor der Präsentation der Testergebnisse erfolgt eine kurze Erläuterung des zu testenden Verfahrens, eine Darstellung der Implementierung in Pseudo–Code sowie eine der getesteten Selektions- und Rekombinationsmethoden. In den ersten Tests werden anhand einer einfachen eindimensionalen Funktion die unterschiedlichen Selektions– und Rekombinationsmethoden im Hinblick auf deren Erfolgsquote und zeitlichen Aufwand miteinander verglichen. Darauf folgend werden weitere Tests für mehrdimensionale Funktionen aufgeführt, die einen Überblick über die Zusammenhänge von Iterationsanzahl (Abbruchkriterium) und Populationsgröße verschaffen sollen, um nachvollziehen zu können, welche Genauigkeit in welcher Zeit unter welcher Kombination erreicht werden kann.

Abschließend wird auf die mögliche Kombination von genetischen Algorithmen mit anderen Verfahren, im speziellen dem Hill-Climbing Verfahren, eingegangen. Welche Vorteile hat eine Verbindung dieser Art zu bieten?

4.1. Algorithmus und Implementierung

4.1.1. Allgemeiner Aufbau

Es folgt eine kurze Darstellung des grundlegenden Aufbaus genetischer Algorithmen. Zudem wird auf die Frage eingegangen, welche Parameter frei wählbar sind, um den

Algorithmus 4.1 : Allgemeiner Ablauf eines GA

Data : Parameter
Result : Chromosom mit maximalem Fitnesswert

1 Initialisierung: Zufallsbasierte Erzeugung der ersten Generation
2 **while** *Abbruchkriterium nicht erfüllt (max. Iterationsanzahl)* **do**
3 Evaluation: Jedem Chromosom wird mit Hilfe der Zielfunktion ein Fitnesswert zugewiesen
4 Selektion: Auswahl der Chromosomen für die Rekombination
5 Rekombination: Erschaffung neuer Chromosomen durch Kreuzung der durch die Selektion ausgewählten Chromosomen
6 Mutation: zufällige Veränderung der Nachfahren
7 **end**

Rahmen für die Tests abstecken zu können.

Daraus ergibt sich die freie Wählbarkeit folgender Parameter:

- Bitlänge (Anzahl der Stellen der Binärzahl)
- Populationsgröße
- Anzahl Iterationen (Abbruchkriterium)
- Methode für Selektion
- Methode für Rekombination
- Methode für Mutation

4.1.2. Bitlänge

Spätestens jetzt stellt sich die Frage, wie man mit Hilfe von Binärzahlen, welche, im Prinzip, Zahlen aus $\mathbb{N}_0$ darstellen können, ein reelles Intervall abdecken kann.

Sei B eine Binärzahl, l die Bitlänge und $I = [a, b]$ das gewünschte Intervall. Mit folgender Formel erfolgt eine Transformation auf das Intervall I:

$$x = \frac{B}{2^{l+1} - 1}(b - a) + a$$

Beispiel: Binärzahl B habe die Bitlänge 10, das gewünschte Intervall sei $I = [-5, 5]$. Binärzahlen mit einer Bitlänge von 10 können Zahlen von 0 bis 1023 darstellen. Normierung auf das Intervall $[0, 1] : x = \frac{B}{1023}$. Normierung auf das Intervall $[-5, 5] : x = x * 10 - 5$.

4.1.3. Initialisierung

Für die Initialisierung müssen die Individuen in der Population zufällig erzeugt werden. Erreicht wird dies mittels einer zufallsgenerierten Matrix $M \in \{0,1\}^{mxn}$ bestehend aus Nullen und Einsen. Hierbei entspricht m (Anzahl der Zeilen) der Populationsgröße bzw. der Anzahl der Individuen, welche als Binärzahlen gegeben sind. n (Anzahl der Spalten) entspricht der Bitlänge der Binärzahlen. Zur Evaluation, also der Berechnung der Fitness, muss jedesmal die Binärzahl in eine Dezimalzahl umgewandelt werden, bevor sie mithilfe der Zielfunktion ausgewertet werden kann.

4.1.4. Selektionsmethoden

Im Rahmen der numerischen Tests fanden die folgenden drei Selektionsmethoden Beachtung:

Roullette-Wheel Selection 4.2:
Diese Form der Selektion fand Verwendung während der Herleitung des Schema-Theorems.

Algorithmus 4.2 : Roulette-Wheel Selection

1. Berechne Fitness der Chromosomen f_i
2. Berechne Anteil an der Gesamtfitness $p_i = \frac{f_i}{\sum f_i}$
3. Jedes Chromosom bekommt auf dem "Rouletterad" einen Bereich der Größe p_i zugewiesen
4. Berechne m Zufallszahlen aus $(0,1)$ (Rouletterad drehen) und wähle die entsprechenden Chromosomen aus

Tournament Selection 4.3:
Für die Tournament Selection existieren mehrere unterschiedliche Ansätze. Meistens werden zufällig $1 \leq k \geq m$ Chromosomen aus der Population gewählt, von denen das mit der größten Fitness ausgewählt wird. Da es tendenziell sinnvoll erscheint, einige 'schwächere' Individuen auszuwählen, um im späteren Verlauf weiterhin qualitativ hochwertige Nachfahren zu erhalten und trotzdem sicher zu stellen, dass die stärksten Individuen nicht ausselektiert werden, fiel die Wahl auf die Implementierung im Algorithmus 4.3.

Truncation Selection/Elitist Selection 4.4:
Die folgende Methode ist eine Mischung aus der Elitist Selection, bei der Individuen 'ungealtert' mehrere Generationen überleben können, und der Truncation Selection, bei welcher die stärksten ausgewählt werden.

Algorithmus 4.3 : Tournament Selection

1 Erstelle Teilnehmerfeld in dem jedes Chromosom zwei mal präsent ist
2 Wähle zufällig zwei Chromosomen aus dem Teilnehmerfeld aus
3 Vergleiche die Fitness
4 Das Individuum mit der höheren Fitness wird ausgewählt
5 Lösche diese beiden Chromosomen aus dem Teilnehmerfeld
6 Fahre fort bis Teilnehmerfeld leer ist

Algorithmus 4.4 : Truncation Selection/Elitist Selection

1 Sortiere Chromosomen nach ihrer Fitness
2 Wähle 50 Prozent der 'stärkeren' für die neue Generation aus
3 Die anderen 50 Prozent entstehen aus Rekombinationen der 'stärkeren' Chromosomen

4.1.5. Rekombinationsmethoden

Es wurden die folgenden drei Rekombinationsvarianten getestet:

- 1-Punkt Crossover
- 2-Punkt Crossover
- Uniformer Crossover

4.1.6. Pseudocode

Zur Übersicht werden nun einige Teile des Verfahrens in Pseudocode 4.5, 4.6, 4.7, 4.8 aufgeführt. Der komplette Quellcode, MATLAB und C++, kann im Anhang eingesehen werden.

Algorithmus 4.5 : Pseudocode genetischer Algorithmus

Data : Zielfunktion, Populationsgröße, Bitlänge und Anzahl Iterationen
Result : max. Fitnesswert: $max_i f(M(i,:))$

1 Erzeuge zufällig eine Matrix $M \in \{0,1\}^{mxn}$
2 **while** *Abbruchkriterium nicht erfüllt* **do**
3 selection()
4 crossover()
5 mutation()
6 **end**

Algorithmus 4.6 : Pseudocode Tournament Selection

1 Erzeuge $T = \{1, 2, ..., m, 1, 2, ..., m\}$
2 **for** $k = 1, ..., m$ **do**
3 Wähle zufällig $i, j \in T, i \neq j$
4 Berechne Fitness von $i, j : f(M(i, :))$ und $f(M(j, :))$
5 Wähle stärkeres aus (bspw. i): $M_{neu}(k, :) = M(i, :)$
6 Lösche i, j aus T
7 **end**
8 Gebe M_{neu} zurück

Algorithmus 4.7 : Pseudocode Unifromer Crossover

1 Erzeuge $T = \{1, ..., m\}$
2 **while** T *nicht leer* **do**
3 Wähle zufällig $i, j \in T$
4 rschaffe 2 neue Chromosomen c_1, c_2 aus i (M(i,:)) und j (M(j,:)) durch uniformen Crossover
5 Übertrage c_1, c_2 in M_{neu}
6 Lösche i, j aus T
7 **end**
8 Gebe M_{neu} zurück

Algorithmus 4.8 : Pseudocode Mutation

1 Gehe jedes Element von M durch
2 Ändere mit vorgegebener Mutationswahrscheinlichkeit von 0 zu 1 bzw. von 1 zu 0

4.2. Numerische Tests (eindimensional)

Die folgenden Tests verschaffen einen Überblick über die Kombinationen von Iterationsanzahl, Populationsgröße, Selektionsmethode, Rekombinationsmethode und Mutation. Da bei diesem Algorithmus die Prozesse zufallsabhängig sind und somit ein einmaliges Anwenden des Verfahrens auf das Problem keine objektiven Ergebnisse liefern kann, erfolgt eine 100-malige Ausführung zur Bestimmung einer Erfolgsquote (Anzahl "richtiger Lösungen" auf 6 Nachkommastellen; erster Eintrag) sowie der durchschnittlichen Abweichung von der exakten Lösung (zweiter Eintrag). Die Bitlänge wurde auf $n = 10$ gesetzt. Die Berechnung dieser ersten Tests erfolgte unter Ubuntu 13.10 64-Bit mit GNU Octave 3.6.4 (Intel Pentium Dual-Core 2x3.2GHz, 4GB RAM). Alle Tests in

diesem Abschnitt beziehen sich auf nachfolgende eindimensionale Funktion. Die oberen Zahlen in den Tabellen sind die Erfolgsquoten, die unteren — die durchschnittlichen Abweichungen.

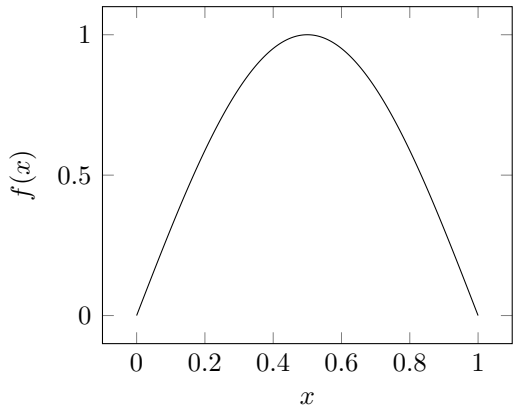


Abbildung 4.1.: $f(x) = \sin(\pi x)$ auf $[0, 1]$

Test 1: Populationsgröße 10, Anzahl Iterationen 10, Mutationswahrscheinlichkeit 1/50

	1-Punkt	2-Punkt	Uniform
Tournament	34% 0.0031938	35% 0.0024602	38% 0.0025656
Roulette Wheel	9% 0.024368	7% 0.027152	5% 0.034031
Elitist	8% 0.011757	14% 0.010879	7% 0.02124

Tabelle 4.1.: Quote und durchschnittliche Abweichung, Test 1

Test 2: Populationsgröße 10, Anzahl Iterationen 50, Mutationswahrscheinlichkeit 1/50

	1-Punkt	2-Punkt	Uniform
Tournament	62% 0.0035248	62% 0.0018656	58% 0.0028361
Roulette Wheel	32% 0.059202	25% 0.047554	29% 0.051855
Elitist	45% 0.011638	59% 0.011455	47% 0.013761

Tabelle 4.2.: Quote und durchschnittliche Abweichung, Test 2

Test 3: Populationsgröße 10, Anzahl Iterationen 100, Mutationswahrscheinlichkeit 1/50

	1-Punkt	2-Punkt	Uniform
Tournament	55% 0.0058105	59% 0.00078751	63% 0.0010315
Roulette Wheel	48% 0.06701	51% 0.061118	44% 0.068841
Elitist	44% 0.023241	57% 0.010784	59% 0.010829

Tabelle 4.3.: Quote und durchschnittliche Abweichung, Test 3

Test 4: Populationsgröße 50, Anzahl Iterationen 10, Mutationswahrscheinlichkeit 1/250

	1-Punkt	2-Punkt	Uniform
Tournament	91% $3.065e{-}06$	77% $4.8568e{-}06$	74% $1.6173e{-}05$
Roulette Wheel	26% 0.00060505	21% 0.00053886	21% 0.00065658
Elitist	20% 0.00049973	21% 0.00060522	7% 0.00072628

Tabelle 4.4.: Quote und durchschnittliche Abweichung, Test 4

Test 5: Populationsgröße 50, Anzahl Iterationen 50, Mutationswahrscheinlichkeit 1/250

	1-Punkt	2-Punkt	Uniform
Tournament	99% $1.2732e{-}06$	95% $2.4991e{-}06$	96% $4.8567e{-}06$
Roulette Wheel	45% 0.0017457	35% 0.0045101	42% 0.0026197
Elitist	50% 0.00086335	48% 0.00053838	52% 0.00049904

Tabelle 4.5.: Quote und durchschnittliche Abweichung, Test 5

Test 6: Populationsgröße 50, Anzahl Iterationen 50, Mutationswahrscheinlichkeit 1/50

	1-Punkt	2-Punkt	Uniform
Tournament	99% $1.4618e{-}06$	98% $1.3675e{-}06$	99% $1.2732e{-}06$
Roulette Wheel	70% 0.0036952	61% 0.0036904	62% 0.0051865
Elitist	62% 0.00039589	53% 0.00082888	65% 0.00038787

Tabelle 4.6.: Quote und durchschnittliche Abweichung, Test 6

Test 7: Populationsgröße 50, Anzahl Iterationen 50, Mutationswahrscheinlichkeit 1/10

	1-Punkt	2-Punkt	Uniform
Tournament	100% $1.1788e{-}06$	99% $3.8194e{-}06$	91% 0.00024578
Roulette Wheel	95% 0.0066455	95% 0.0036508	82% 0.0089702
Elitist	86% 0.0045138	79% 0.037691	86% 0.042401

Tabelle 4.7.: Quote und durchschnittliche Abweichung, Test 7

Benötigte Zeit: Populationsgröße: 50, Anzahl Iterationen: 50

Zeit in s	1-Punkt	2-Punkt	Uniform
Tournament	5.8844	6.3324	7.7333
Roulette Wheel	6.8087	7.2858	8.7748
Elitist	4.4421	4.4833	4.4698

Tabelle 4.8.: Benötigte Zeit, Test 7

Für den eindimensionalen Fall offenbaren die Tests einerseits große Unterschiede zwischen den Selektionsmethoden, andererseits eher marginale Unterschiede zwischen den Rekombinationsmethoden. Es lässt sich eindeutig feststellen, dass die Tournament-Selection die anderen Selektionsmethoden klar dominiert; in allen Tests erzielt sie die besten Resultate. Besonders profitiert sie von einer höheren Populationsgröße und Mutationswahrscheinlichkeit. Es können zudem recht schnell gute Ergebnisse in Verbindung mit 1-Punkt Crossover unter niedriger Populationsgröße und Iterationsanzahl erreicht

werden, was zu einer Verringerung des Rechenaufwands genutzt werden kann. Übersteigt aber die Iterationsanzahl einen gewissen Wert, verschlechtern sich die Ergebnisse, wenn man vom uniformen Crossover absieht. Somit spielen die anderen Selektionverfahren eine eher untergeordnete Rolle. Die Roulette-Wheel Selection verlangt nach hohen Populationsgrößen, Iterationszahlen und Mutationswahrscheinlichkeiten um vergleichbar zu erscheinen; die Elitist Selection erzielt einzig in Test 2 (2-Punkt Crossover) und 3 (2-Punkt und uniformer Crossover) vergleichsweise gute Ergebnisse, aber mit höheren durchschnittlichen Abweichungen. Zu beachten ist jeodch, dass die Elitist Selection den geringsten Zeitaufwand verursacht.

Bei den Rekombinationsvarianten kann sich unter Tournament Selection der 1-Punkt Crossover behaupten (4 bis 7). Auch unter den anderen Selektionsmethoden schneidet er meistens gut ab und kann den 2-Punkt Crossover in den Tests 4 bis 7 dominieren. Der uniforme Crosover brilliert in Relation zu den anderen Methoden vor allem bei kleineren Populationsgrößen; höhere Mutationswahrscheinlichkeiten dämpfen aber die Resultate. In Test 4 leistet er sich einen Ausreißer unter der Elitist Selection. Die Verbindung von uniformen Crossover und Elitist Selection verlangt anscheinend nach einer höheren Iterationsanzahl. Erwähnenswert ist zudem, dass eine höhere Mutationswahrscheinlichkeit häufig gute Ergebnisse bei unveränderter durchschnittlicher Abweichung erzielt.

Aus diesen Ergebnissen kann gefolgert werden, dass eine Erhöhung der Populationsgröße deutlich bessere Resultate erzielt. Eine Erhöhung der Iterationsanzahl führt in den meisten Fällen ebenfalls zu besseren Ergebnissen, allerdings nicht im gleichen Ausmaße. Da beide Möglichkeiten ungefähr den gleichen Aufwand verursachen, kann gefolgert werden, dass man zur Verbesserung der Ergebnisse tendenziell eher die Populationsgröße anstelle der Iterationsanzahl erhöhen sollte. In einigen Fällen vollzieht sich nach einer gewissen Anzahl an Iterationen eine Verschlechterung der Resultate. Demnach besteht die Notwendigkeit einer dem Problem angemessenen Wahl der Iterationszahl - mehr Iterationsschritte führen nicht zwangsläufig zu einer Verbesserung.

4.3. Numerische Tests (mehrdimensional)

Nun wird der genetische Algorithmus auf mehrdimensionale Funktionen $f : D \subset \mathbb{R}^2 \to \mathbb{R}$ angewendet. Die Bitlänge wird auf $n = 30$ festgesetzt, als Selektionsvariante Tournament Selection, als Rekombinationsmethode der uniforme Crossover und eine Mutationswahrscheinlichkeit von $\frac{1}{10n}$ gewählt. Die folgenden Tests wurden mit GNU C++ unter Ubuntu 13.10 bei unveränderter Hardware kompiliert. Auch hier wird der Algorithmus

jeweils 100 mal durchgeführt. Die nachstehenden Tabellen zeigen die durchschnittliche Abweichung (oben) von der exakten Lösung sowie die benötigte Zeit (unten), um einen Vergleich zwischen Populationsgröße und Iterationsanzahl zu geben. Beschriftungen in der linken Spalte stellen die Populationsgröße und Beschriftungen in der obersten Zeile die Iterationsanzahl dar.

Test 8:

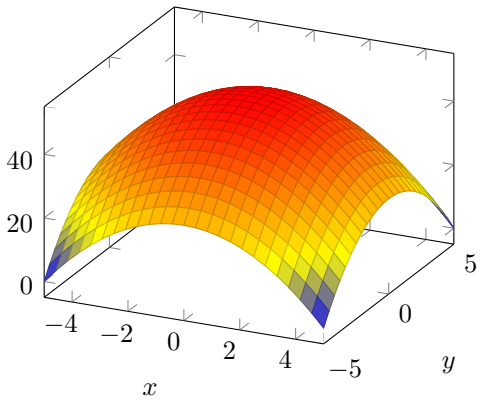
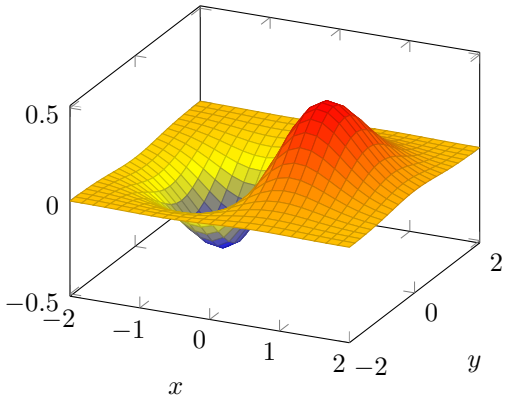


Abbildung 4.2.: $f(x, y) = 50 - x^2 - y^2$ auf $[-5, 5] \times [-5, 5]$

Population / Iterationen	10	50	100	500
10	0.85861933 $0.0008s$	0.19815222 $0.0039s$	0.20910056 $0.0077s$	1.1546998 $0.0387s$
50	0.20170965 $0.0034s$	0.00029872 $0.0169s$	0.00014079 $0.0337s$	$6.21848e-05$ $0.1682s$
100	0.00028372 $0.0069s$	$3.59381e-06$ $0.0337s$	$8.79642e-07$ $0.0666s$	$1.23103e-05$ $0.3377s$
500	0.00688221 $0.0344s$	$3.13665e-11$ $0.1687s$	0 $0.3329s$	$7.10542e-17$ $1.667s$

Tabelle 4.9.: Durchschnittliche Abweichung und die benötigte Zeit, Test 8

Test 9:



Quelle: eigene Darstellung

Abbildung 4.3.: $f(x, y) = xexp(-x^2 - y^2)$ auf $[-2, 2] \times [-2, 2]$

Population / Iterationen	10	50	100	500
10	0.00463929 $0.0008s$	0.03499361 $0.0039s$	0.03509808 $0.0079s$	0.04269244 $0.0041s$
50	0.00337451 $0.0035s$	0.00079542 $0.0178s$	0.00051921 $0.0352s$	0.00038902 $0.1743s$
100	0.00017032 $0.0071s$	0.00045263 $0.0349s$	0.00029578 $0.0707s$	0.00028805 $0.3425s$
500	0.00041027 $0.0353s$	$3.76329e-05$ $0.1837s$	$1.41898e-09$ $0.3445s$	$4.63462e-05$ $1.753s$

Tabelle 4.10.: Durchschnittliche Abweichung und die benötigte Zeit, Test 9

Test 10:

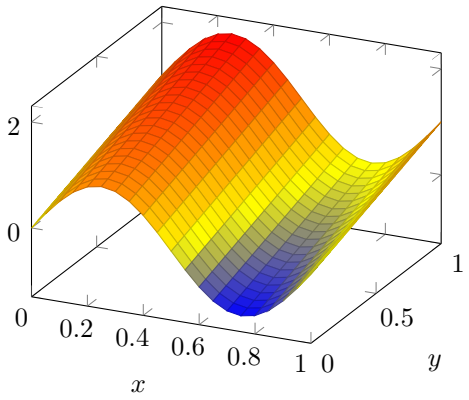


Abbildung 4.4.: $f(x, y) = \sin(2\pi x) + y$ auf $[0, 1] \times [0, 1]$

Population / Iterationen	10	50	100	500
10	0.10239198 $0.0008s$	0.09229848 $0.0038s$	0.20369982 $0.0078s$	0.04902680 $0.039s$
50	0.00632808 $0.0034s$	0.00030542 $0.0176s$	$2.75457e-06$ $0.0346s$	$2.0825e-05$ $0.1732s$
100	0.00197074 $0.007s$	$4.30026e-08$ $0.034s$	$3.05991e-07$ $0.0688s$	$9.02246e-06$ $0.3986s$
500	0.00070039 $0.035s$	$1.1235e-11$ $0.1746s$	0 $0.3446s$	0 $1.719s$

Tabelle 4.11.: Durchschnittliche Abweichung und die benötigte Zeit, Test 10

Die hier offerierten Tests zeigen, dass die optimale Populationsgröße sowie die Anzahl der Iterationen von der Zielfunktion abhängen. Bei allen Tests ist eine gewisse Kombination beider Größen für eine bestimmte Genauigkeit erforderlich. Für eine Genauigkeit auf 5 Stellen sind bei Test 8 mindestens 50 Iterationen und eine Populationsgröße von 100 vonnöten, bei Test 9 hingegen eine Populationsgröße von 500. Test 10 verlangt für die gleiche Genauigkeit moderate 100 Iterationen und eine Populationsgröße von 50.

Allgemeine Richtlinien lassen sich infolgedessen nur schwer angeben; feststellen kann man aber, dass zur Verbesserung der Lösung wieder die Populationsgröße anstelle der Iterationsanzahl angehoben werden sollte. Eine „zu hohe“ Anzahl an Iterationen führt auch hier zu einer Zunahme des Fehlers.

4.4. Verbindung mit dem Hill-Climbing Verfahren

In diesem Abnschitt soll darauf eingegangen werden, wie sich der genetische Algorithmus sinnvoll mit anderen Verfahren kombinieren lässt und welche Vorteile dies mit sich bringt. Mit genetischen Algorithmen kann relativ schnell und einfach mittels kleiner Population und Iterationsanzahl eine gute Approximation der gesuchten Lösung bestimmt werden. Um eine hohe Genauigkeit zu erreichen wird allerdings eine deutlich höhere Populationsgröße und/oder Anzahl an Iterationen benötigt.

Aufgrund dieser Erkenntnis kann es als sinnvoll erachtet werden, eine Näherungslösung mit einer kleinen Population und geringer Iterationsanzahl zu bestimmen, welche als Startwert für ein anderes Verfahren, wie beispielsweise das Gradientenverfahren dient. Da aber das Gradientenverfahren, anders als die genetischen Algorithmen, nur auf differentierbare und stetige Funktionen anwendbar ist, fiel die Wahl auf eine Kombination mit dem Hill-Climbing Verfahren. Damit ist weiterhin ein breiter Anwendungsbereich sichergestellt.

Um die Funktionsweise klar zu machen, folgt eine kurze Darstellung des Hill-Climbing Verfahrens für den eindimensionalen Fall:

Algorithmus 4.9 : Ablauf des Hill-Climbing Verfahrens

Data : Startwert x_0, Schrittweite h

Berechne $v_1 = f(x_0 + h), v_2 = f(x_0 - h)$
if $x_0 > v_1$ *und* $x_0 > v_2$ **then**
    x_0 ist die Lösung
else if $x_0 < v_1$ *und* $v_1 > v_2$ **then**
    $x_1 = v_1$
else if $x_0 < v_2$ *und* $v_2 > v_1$ **then**
    $x_1 = v_2$
etc

Dieses Verfahren lässt sich zwar auch problemlos auf mehrdimensionale Funktionen anwenden, wodurch jedoch der Aufwand deutlich ansteigt. Dies hängt damit zusammen, dass wesentlich mehr Richtungen 'abgetastet' werden müssen. Es folgen zwei Tests zum kombinierten Verfahren:

Test 11: $h = 10^{-4}$

Population / Iterationen	10	50	100	500
10	$4.50583e-05$ $0.0015s$	$8.35800e-05$ $0.0039s$	$6.11396e-05$ $0.0114s$	$4.43963e-05$ $0.0397s$
50	$3.72960e-05$ $0.0036s$	$2.63988e-05$ $0.0172s$	$2.38828e-05$ $0.0372s$	$3.31514e-05$ $0.1723s$
100	$6.31381e-05$ $0.007s$	$1.52321e-06$ $0.037s$	$5.9032e-09$ $0.069s$	$9.5042e-06$ $0.3412s$

Tabelle 4.12.: Durchschnittliche Abweichung und die benötigte Zeit, Test 11

Test 12: $h = 10^{-6}$

Population / Iterationen	10	50	100
10	$5.88087e-07$ $0.0442s$	$6.3011e-07$ $0.0426s$	$8.0705e-07$ $0.0314s$
50	$2.76183e-07$ $0.0078s$	$4.64412e-07$ $0.017s$	$1.79602e-07$ $0.0364s$

Tabelle 4.13.: Durchschnittliche Abweichung und die benötigte Zeit, Test 12

Bei Betrachtung der Testergebnisse fällt sofort auf, dass das kombinierte Verfahren schon bei Wahl niedriger Parameterwerte hohe Genauigkeiten erzielt. Die Wahl niedrigerer

Werte resultiert in einer kürzeren Prozessdauer. Hierbei spielt zusätzlich auch die Wahl der Schrittweite eine entscheidende Rolle. Eine kleinere Schrittweite führt zu einer höheren Genauigkeit, aber auch zu höherem Rechenaufwand.

Im Vergleich mit den Ergebnissen des vorigen Abschnitts kann die Kombination von genetischen Algorithmen mit dem Hill-Climbing Verfahren die Performance teilweise verbessern. Tendenziell ist es sinnvoll, eine sehr kleine Populationsgröße und Iterationsanzahl zu verwerden, um eine Maximierung der Leistungsfähigkeit zu erreichen. Höhere Genauigkeiten verlangen höhere Parameterwerte, da ansonsten dass Hill-Climbing Verfahren zu viele Iterationsschritte benötigt. Bei kluger Wahl aller Parameterwerte kann dieses Verfahren eine Zeitersparnis gegenüber dem gewöhnlichen genetischen Algorithmus erzielen.

4.5. Zusammenfassung

Die Tests dieses Kapitels geben einige Einblicke in die praktische Anwendung von genetischen Algorithmen. Sie zeigen deutlich, dass die Wahl der einzelnen Methoden und Parameter von dem zugrundeliegenden Problem abhängig ist. Die markanteste Erkenntnis dieses Kapitels ist die übergeordnete Bedeutung der Selektionsmethode und der Populationsgröße für die Erreichung der gesteckten Näherung. Unter Wahl einer großen Population und der Tournament Selection kann recht schnell ein akzeptables Ergebnis erzielt werden - die weiteren Parameter spielen dann eine eher untergeordnete Rolle. Bei den Rekombinationsvarianten sind die Unterschiede nur moderat, woraus sich nicht folgern lässt, welche Methode bevorzugt werden sollte. In Kombination mit der Tounament Selection erzielt jedoch der 1-Punkt Crossover die besten Ergebnisse im gesamten Testdurchlauf. Entgegen der Theorie aus Kapitel 3 trägt eine höhere Mutationswahrscheinlichkeit in den meisten Fällen zu einer Verbesserung die Lösungsquote bei. Auch der uniforme Crossover scheint bei hohen Iterationszahlen nicht so destruktiv zu sein wie prognostiziert.

Unter der Kombination des genetischen Algorithmus mit dem Hill-Climbing Verfahren kann eine Zeitersparnis erreicht werden, was jedoch eine smarte Wahl der Parameter und der Schrittweite voraussetzt.

Für weiterführende Aussagen und zur Festigung der hier erzielten Resultate sind weitere Tests für eine Vielzahl an Funktionen und Problemstellungen erforderlich. Eine derartige Untersuchung würde aber den Rahmen dieser Arbeit sprengen und so bleibt an dieser Stelle nur der Verweis auf die Notwendigkeit weiterer Untersuchungen.

5. Das Problem Economic Dispatch

Dmitri Bogonos

Der Anwendungsbereich der genetischen Algorithmen in der Wirtschaft ist sehr groß. In diesem Kapitel wird eine sehr wichtiges Optimierungsproblem, das bei allen Energieunternehmen auftritt, mithilfe der GA gelöst. Das Problem *Economic Dispatch* beschreibt Bestimmung der Menge an Strom, die an mehreren Anlagen zu möglichst geringen Kosten produziert werden muss, um die Nachfrage der Verbraucher zu decken. Nach R. Ouiddir, M. Rahli und L. Abdelhakem-Koridak [25] muss die Balance zwischen dem Stromerzeugnis und der variablen Auslastung des Stromnetzes (bedingt durch die variable Nachfrage) gefunden werden. Dabei ist wichtig, welche Anlage wie viel Strom produziert. Diese Entscheidung muss alle wenige Stunden bis alle wenige Minuten getroffen werden.

5.1. Formulierung

Das Problem Economic Dispatch wird nach [25] und P. Chen, H-C. Chang [27] wie folgt formuliert:

$$\begin{aligned} \min F &= \sum_{i=1}^{n} f_i\left(P_i\right) \\ s.t. \quad \sum_{i=1}^{n} P_i &= P_D + P_{loss}, \\ P_i^{min} &\leq P_i \leq P_i^{max}. \end{aligned} \tag{ED}$$

Dabei bezeichnen F die gesamten Kosten der Stromerzeugung, P_i die Erzeugung an der Anlage i, P_i^{min} und P_i^{max} die minimale bzw. die maximale Erzeugung der Anlage i, n die Anzahl der Anlagen, $f_i\left(P_i\right)$ die Produktionskosten der Anlage i, P_D die Auslastung des Systems und P_{loss} die Übertragungsverluste.

Die Produktionskosten $f_i\left(P_i\right)$ werden als quadratische Funktion dargestellt:

$$f_i\left(P_i\right) = a_i P_i^2 + b_i P_i + c_i, \tag{5.1}$$

wobei a_i, b_i und c_i Konstanten sind.

Die Übertragungsverluste P_{loss} haben geographische und/oder physikalisch-technische Gründe und können z.B. als abgegebene Wärme aufgefasst werden. Ihre Bestimmung kann auf zwei Wegen erfolgen:

- mit den Penalty-Faktoren (dazu siehe [28]),
- mit den B-Koeffizienten.

Die letzte Methode wird hier verwendet. L. K. Kirchmayer, G. W. Stagg [21], A. F. Glimn, L. K. Kirchmayer, J. J. Skiles [11] und E. E. George [9] unterstellen, dass die Verluste auf gegenseitige Eigenschaften der Stromlinien (Entfernungen zwischen Kraftwerken und Umspannwerken, Material der Stromleitungen) und der Sammelschienen der Umspannwerke zurückzuführen sind. Deswegen werden Übertragungsverluste als quadratische Funktion in den Produktionsmengen betrachtet:

$$P_{loss} = \sum_{i=1}^{n} \sum_{j=1}^{n} P_i B_{ij} P_j. \tag{5.2}$$

Die Koeffizienten B_{ij} beinhalten mehrere charakteristische Größen der Stromnetze: Spannungen, Widerstände der Linien, Entfernungen usw.

Die Betriebsbereiche der einzelnen Kraftwerke können durch kraftwerkspezifische Grenzen limitiert werden. Ebenso spezifisch ist der Ausmaß der Anpassung (*ramp rate limits*) der zu produzierenden Menge an Strom begrenzt. Wenn am Kraftwerk i heute P_i^0 MWh erzeugt werden, und morgen soll die Erzeugung auf P_i erhöht werden, dann kann man sie maximal um UR_i erhöhen. Analog, wenn morgen weniger produziert werden soll, kann man die Produktion höchstens um DR_i herabsenken. Also:

- $P_i - P_i^0 \leq UR_i$, falls die Produktion erhöht werden soll,
- $P_i^0 - P_i \leq DR_i$, falls die Produktion gesenkt werden soll.

Die folgende Abbildung 5.1 zeigt drei mögliche Situationen eines Kraftwerks.

Die Betriebsbereiche der Kraftwerke können auch durch s.g. *prohibited zones* — verbotene Zonen eingeschränkt werden. Diese Zonen sind bestimmte Leistungen, die ein Kraftwerk aufgrund seiner bautechnischen Besonderheiten nicht erzeugen kann. Diese können z.B. Vibrationen in den Maschinen sein. Die Produktion in den verbotenen Zonen kann zum Ausfall einzelner Anlagen oder sogar ganzer Kraftwerke führen oder der Verlauf der Input-Output-Kurven ist in diesen Zonen unbekannt, nicht eindeutig oder zu komplex. Nach [27] ist die optimale produzierte Leistung diejenige, die außerhalb

dieser Zonen liegt. Um diese Zonen umzugehen, werden Heuristiken angewendet. Sollte $P_i \leq P_i^0$ gelten und P_i liegt in einer verbotenen Zone, wird $P_i = P_{pz}^+$ gesetzt. Andererseits, gilt $P_i^0 \leq P_i$ und P_i liegt in einer verbotenen Zone, wird $P_i = P_{pz}^-$ gesetzt (siehe Abbildung 5.2).

Zusammenfassend, das Problem (ED) kann reformuliert werden:

$$\begin{aligned} \min F &= \sum_{i=1}^{n} f_i\left(P_i\right) \\ s.t. \quad &\sum_{i=1}^{n} P_i = P_D + P_{loss}, \qquad \text{(ED*)} \\ \max\left(P_i^{min}, P_i^0 - DR_i\right) &\leq P_i \leq \min\left(P_i^{max}, P_i^0 + UR_i\right), \end{aligned}$$

Somit ergibt sich, dass die Lösungsmenge des Problems aus mehreren Intervallen in $\mathbb{R}$ besteht (roter Bereich in der Abbildung 5.2) und damit nicht konvex ist. Die konvexen Optimierungsverfahren kommen damit für die Lösung des Problems nicht infrage. Es bietet sich ein genetischer Algorithmus an.

5.2. Implementierung

Die einzige Variable, die kodiert wird, ist λ^{nm} mit $0 \leq \lambda^{nm} \leq 1$, das normierte Zusatzkosten beschreibt, die in diesem Fall mit Grenzkosten der Produktion übereinstimmen[1]. Damit ist das Problem unabhängig von der Anzahl der Kraftwerke, was die Lösungsmethode auch für große Stromnetze geeignet macht. Es wird behauptet, dass durch diese Wahl

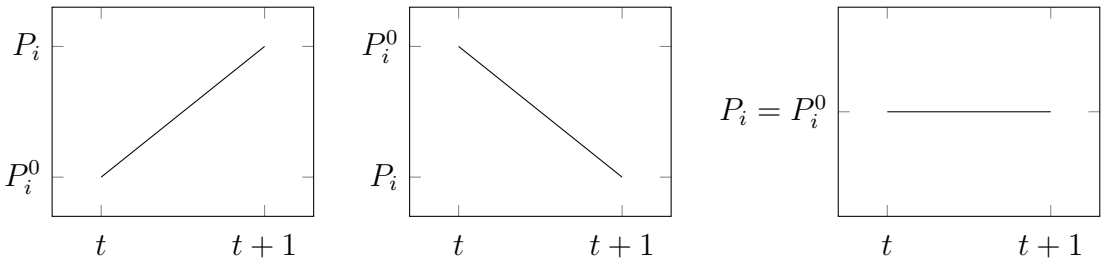
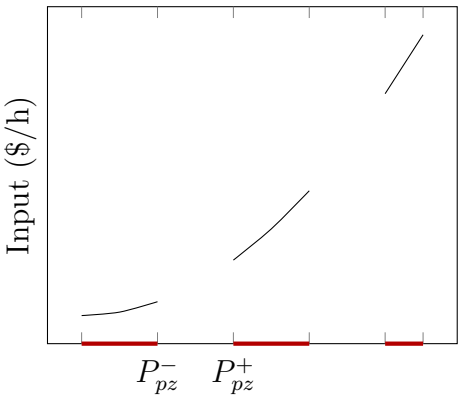


Abbildung 5.1.: Änderung der Stromerzeugung
Quelle: P. Chen, H-C. Chang [27]

[1] Grenzkosten der Produktion sind Kosten, die bei der Produktion einer zusätzlichen Einheit Output entstehen. Im mathematischen Sinne entsprechen sie der Ableitung der Kostenfunktion nach der Variable, die die Anzahl der produzierten Einheiten darstellt, also P_i. Die Zusatzkosten beschreiben Kosten, die durch Erhöhung oder Hinzunahme neuer Kostenfaktoren, z.B. neuer Produktionsfaktoren entstehen. Hier gilt Grenzkosten = Zusatzkosten, da die Kostenfunktion (5.1) keine anderen Kostenfaktoren außer Produktionseinheiten als Variablen hat.



Quelle: nach P. Chen, H-C. Chang [27]

Abbildung 5.2.: Input-Output-Kurve

der Kodierung die Lösungszeit linear mit der Anzahl der Kraftwerke steigt.

Die Autoren benutzen 10 Bits für die Kodierung. Sie behaupten, dass die Genauigkeit der Lösung mit der Anzahl der Bits zunimmt, wobei gleichzeitig die Konvergenz langsamer wird. Die Länge von 10 Bits wird hier wie folgt benutzt:

$$\begin{array}{cccccccccc} d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & d_{10} \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} & 2^{-5} & 2^{-6} & 2^{-7} & 2^{-8} & 2^{-9} & 2^{-10} \end{array}$$

wobei $d_i \in \{0, 1\}$, i = 1,2, ... 10. Die Dekodierung erfolgt gemäß

$$\lambda^{nm} = \sum_{i=1}^{10} d_i 2^{-i}, \quad d_i \in \{0, 1\}. \tag{5.3}$$

Die Beziehung zwischen den tatsächlichen und normierten Zusatzkosten ist gegeben durch

$$\lambda^{act} = \lambda_{sys}^{min} + \lambda^{nm} \left(\lambda_{sys}^{max} - \lambda_{sys}^{min} \right), \tag{5.4}$$

wobei $\lambda_{sys}^{min} = min_i \left\{ \frac{\partial f_i(P_i^{min})}{\partial P_i} \right\}$ und $\lambda_{sys}^{min} = max_i \left\{ \frac{\partial f_i(P_i^{max})}{\partial P_i} \right\}$ gilt.

Mit der Annahme, dass die Lösungsmenge des Problems konvex ist, kann man (ED*)

mit dem Lagrangeansatz lösen:

$$\begin{aligned} &Pf_i\,(2a_iP_i + b_i) = \lambda^{act}, \quad \max\left(P_i^{min}, P_i^0 - DR_i\right) \le P_i \le \min\left(P_i^{max}, P_i^0 - UR_i\right) \\ &Pf_i\,(2a_iP_i + b_i) \le \lambda^{act}, \quad P_i = \min\left(P_i^{max}, P_i^0 - UR_i\right) \qquad (*) \\ &Pf_i\,(2a_iP_i + b_i) \ge \lambda^{act}, \quad P_i = \max\left(P_i^{min}, P_i^0 - DR_i\right), \end{aligned}$$

wobei $Pf_i = \frac{1}{1+\frac{\partial P_{loss}}{\partial P_i}}$.

Es wird

$$\varepsilon := \left|\sum_{i=1}^{n} P_i - P_D + P_{loss}\right| \tag{5.5}$$

definiert. Das Abbruchkriterium ist erfüllt, wenn ε unter einem bestimmten Wert liegt. Die normierte Fitnessfunktion wird wie folgt definiert:

$$F = \frac{1}{1 + k\left(\frac{\varepsilon}{P_D}\right)} \quad \text{mit Normierungsvariable k = 200.} \tag{5.6}$$

Die Selektion wird mit der Regel *Roulette wheel* simuliert. Folgende Werte werden gewählt: Populationsgröße = 16, Crossoverwahrscheinlichkeit = 0.8, Mutationswahrscheinlichkeit = 0.1.

Der Ablauf ist im Algorithmus 5.1 beschrieben. Schritt 2 von 5.1 ist im Algorithmus 5.2 beschrieben.

Algorithmus 5.1 : Ablauf des GA

Data : Anlagen, Nachfrage ...
Result : Output

1 Initialisiere Startpopulation
2 Evaluiere jedes Chromosom
3 Ordne alle Chromosomen nach ihrer Fitness
4 Wähle beste Eltern für die Reproduktion
5 Wende Crossover und Mutation an
6 Evaluiere die neue Generation und ersetze die schwächsten durch die stärksten
7 **if** *Konvergenzkriterium (5.5) erfüllt* **then**
8 Gebe die Population aus
9 **else**
10 Gehe zu 4

Algorithmus 5.2 : Auswertung der Individuen

1 Bestimme λ^{nm} mit (5.3)
2 Bestimme λ^{act} mit (5.4)
3 $i = 1$
4 Berechne P_i mit $(*)$
5 **if** *P_i in einer verbotenen Zone* **then**
6 **if** *Phase der steigenden Auslastung* **then**
7 $P_i = P_{pz}^{+}$
8 **else**
9 $P_i = P_{pz}^{-}$
10 **if** *i die letzte Anlage* **then**
11 Berechne P_{loss} mit (5.2)
12 **else**
13 $i = i + 1$ und gehe zu 4
14 Berechne die Fitness mit (5.5) und (5.6)

5.3. Zahlenbeispiel

Die Autoren von [27] konstruieren ein Stromnetz aus 3 Kraftwerken mit Eigenschaften, die in Tabellen 5.1 und 5.2 zusammengefasst sind.

Anlage	P_i^{min}	P_i^{max}	a_i in $\$/MW^2$	b_i in $\$/MW^2$	c_i in \$
1	50	250	0.00525	8.663	328.13
2	5	150	0.00609	10.04	136.91
3	15	100	0.00592	9.76	59.16

Tabelle 5.1.: Koeffizienten und Leistungsgrenzen

Anlage	P_i^0	$UR_i\left(MW/h\right)$	$DR_i\left(MW/h\right)$	Prohibited zones (MW)
1	215	55	95	[105,117],[165,177]
2	72	55	78	[50,60],[92,102]
3	98	45	64	[25,32],[60,67]

Tabelle 5.2.: Ramp rates und prohibited zones

Die Matrix B_{ij} gibt die Verlustkoeffizienten an:

$$B_{i,j} = \begin{pmatrix} 0.000136 & 0.0000175 & 0.000184 \\ 0.0000175 & 0.000154 & 0.000283 \\ 0.000184 & 0.000283 & 0.00161 \end{pmatrix}.$$

Die Nachfrage beträgt 300 MW, als Konvergenzkriterium wird $\varepsilon < 0.3MW(0.001 * Nachfrage)$ gewählt.

Die Berechnungen wurden am Rechner mit dem 486-33 Prozessor durchgeführt. In der zweiten Generation ist das Konvergenzkriterium erfüllt und die optimale Lösung ist gefunden (nur die ersten drei Individuen werden aufgelistet):

Rank	Chromosom	λ^{nm}	$\varepsilon(MW)$	Fitness
1	1000111010	0.556641	3.4625	0.8125
2	1001010101	0.583008	6.5155	0.6972
3	1000101011	0.541992	9.7654	0.6057

Tabelle 5.3.: Initialisation

Rank	Chromosom	λ^{nm}	$\varepsilon(MW)$	Fitness
1	1001000000	0.562500	1.2371	0.9241
2	1001001000	0.570313	1.7308	0.8966
3	1000111010	0.556641	3.4625	0.8125

Tabelle 5.4.: 1. Iteration

Rank	Chromosom	λ^{nm}	$\varepsilon(MW)$	Fitness
1	1001000011	0.565430	0.1192	0.9921
2	1001000010	0.564453	0.4898	0.9684
3	1001000000	0.562500	1.2317	0.9241

Tabelle 5.5.: 2. Iteration

Hier sind die Ergebnisse[2]:

[2]Rundungsfehler möglich

- $P_1 = 194.265$ MW, $P_2 = 49.999$ MW, $P_3 = 79.627$ MW
- $P_{loss} = 24.011$ MW
- $\lambda^{act} = 10.7028$ \$/MWh
- Zeit $= 0.016$s.

5.4. Anwendung auf das westalgerische Stromnetz

Die Autoren von [25] wenden diesen Algorithmus auf das westalgerische Stromnetz mit zwei Kraftwerken:

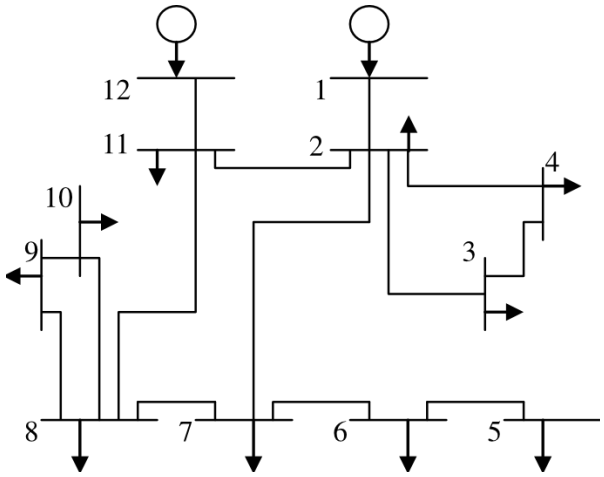


Abbildung 5.3.: Stromnetz von Westalgerien

Die Fitnessfunktion ist gegeben durch

$$F = \frac{1}{1 + \frac{1}{\sum_i f_i}}.$$

Die Parameter werden wie folgt gewählt: Populationsgöße $= 10$, Crossoverwahrscheinlichkeit $= 0.88$, Mutationswahrscheinlichkeit $= 0.1$. Die Kostenfunktionen lauten:

$$f_1(P_1) = 0.85P_1^2 + 150P_1 + 2000,$$
$$f_2(P_2) = 1.7P_2^2 + 250P_2 + 3000,$$

mit

$$30 \leq P_1 \leq 510\ MW,$$
$$10 \leq P_2 \leq 70\ MW,$$
$$P_D = 505\ MW,$$
$$\text{Fall 1: } P_{loss} = 15.94,$$
$$\text{Fall 2: } P_{loss} = 0.0189P_1 + 0.0924P_2.$$

Zur Vereinfachung wird hier unterstellt, dass es keine verbotenen Zonen gibt. Im ersten Fall werden die Übertragungsverluste als konstant angenommen mit $P_{loss} = 15.94MW$. Die Ergebnisse sind in der Tabelle 5.6 zusammengefasst. Im zweiten Fall werden die Verluste als lineare Funktion in $P_i, i = 1, 2$ angenommen, mit $P_{loss} = 0.0189P_1 + 0.0924P_2$. Die Ergebnisse sind in der Tabelle 5.7 zu sehen. Die Berechnungen werden mit konvexen Lösungsmethoden und den tatsächlichen Entscheidungen des Netzbetreibers verglichen.

	GA	Fletcher-Reeves[a]	Fletcher[b]	Sonelgaz[c]
P_1	450.95	466.64	469.93	465.94
P_2	69	54.25	49.98	55
P_{loss}	15.94	15.94	15.94	15.94
Kosten (Nm^3/h)	241764	278649	279940	278319
Berechnungszeit	0.05	0.01	0.01	/
Generation	304	8	3	/

Tabelle 5.6.: Fall 1

Im ersten Fall mit konstanten Verlusten liefert der genetische Algorithmus im Vergleich zu konvexen Methoden und zu Sonelgaz bei vergleichbarer Laufzeit deutlich niedrigere Kosten mit Jahresersparnis von über 2%. Im zweiten Fall können dank der linearen Form die Verluste deutlich gesenkt werden, obwohl sie höher sind als bei den konvexen Methoden. Die Kosten sind dagegen im GA niedriger und die Jahresersparnis beläuft sich auf ca. 3%. Somit hat der genetische Algorithmus den entscheidenden Vorteil gegenüber den konvexen Optimierungsmethoden.

Als Alternative zu üblichen innerbetrieblichen Lösungen des Problems Economic Dispatch kann die Verwendung der auf genetischem Algorithmus basierten Methode bessere

[a] Nichtlineare konjugierte Gradienten Methode
[b] BFGS-Verfahren bzw. DFP-Formel
[c] Daten des algerischen Netzbetreibers Sonelgaz

	GA	Fletcher-Reeves	Fletcher	Sonelgaz
P_1	449.98	465.374	468.92	465.94
P_2	70	53.36	49.5	55
P_{loss}	14.97	13.72	13.43	15.94
Kosten (Nm^3/h)	270444	277067	278779	278319
Berechnungszeit	0.1	0.05	0.01	/
Generation	76	25	3	/

Tabelle 5.7.: Fall 2

Ergebnisse liefern. P. Chen, H-C. Chang [27] zeigen, dass die vorgestellte GA-basierte Methode insbesondere für große Stromnetze bessere Ergebnisse liefert und schneller ist, als die weit verbreitete Lambda-Iterationsmethode. Sie behaupten außerdem, dass die letzte solche Besonderheiten wie ramp-rate limits, verbotene Zonen und Übertragungsverluste nicht berücksichtigen kann.

Darüber hinaus ist die Prognose der Energienachfrage mithilfe genetischer Algorithmen möglich (siehe z.B. H. Ceylan und H. K. Ozturk [5]). Es ist damit möglich, beide Ansätze zur Optimierung der Energieproduktion zu verwenden.

6. Weitere Ansätze und Anwendungen auf die numerische Lösung von Differentialgleichungen

Marius Kulms, Christian Loenser, Nick Macin

6.1. Grammatical Evolution

Marius Kulms

Für die folgende Theorie zur numerischen Lösung von Differentialgleichungen wird nun eine weitere Verfeinerung der *Evolutionary Computing*–Familie eingeführt: den Ansatz der sogenannten *Grammatical Evolution* — im Folgenden als GE bezeichnet.

Diese weitere Verfeinerung liegt in der formulierten Zielsetzung der Lösung von Differentialgleichungen bzw. dynamischen Systemen begründet: Zwar ist es bisher möglich, Iterationslösungen von GAs sinnvoll mit Hilfe eines Alphabets zu kodieren, für die Darstellung von Differentialgleichungen ist es aber darüber hinaus erforderlich, Funktionen und ihre Ableitungen darzustellen. Eine hilfreiche Lösung dieses Problems bietet das Instrument der GE, erstmals vorgestellt von Koza [22]); das im Folgenden vorgestellte Konzept basiert auf einer Arbeit der Autoren Tsoulos und Lagaris [32]. Der Grundbaustein von GE ist eine kontextfreie Grammatik in Backus–Naur–Form, d.h. ein Quadrupel $\{N, T, P, S\}$, wobei N eine Menge von Nichtterminalsymbolen, T eine Menge von Terminalsymbolen, P eine Menge von Produktions– bzw. Ableitungsregeln sowie $S \in N$ ein Startsymbol bezeichne. Die Individuen sind nun wie bei GAs als Vektoren von Integers kodiert, wobei jedes Integer für eine Ableitungsregel stehe. Die Auswahl der Ableitungsregel erfolgt in zwei Schritten: zunächst wird von links nach rechts des ungeordneten Chromosomenvektors ein Integer nach dem anderen eingelesen und ihm

sein tatsächlicher Wert V zugewiesen. Im Anschluss erfolgt die Auswahl nach folgender Rechnung:

$$Rule = V \ mod \ NR, \tag{6.1}$$

wobei NR die Anzahl der möglichen Ableitungsregeln je Nichtterminale bezeichne. Dieses technische Vorgehen soll nun anhand eines Beispiels verdeutlicht werden.

Beispiel 6.1
Sei der Vektor $j = [16, 3, 7, 4, 10, 28, 24, 1, 2, 4]$ gegeben.

Behauptung 6.2
j kodiert die Funktion $M_i(x) = log(x^2)$.

Beweis
Anwenden der Regel (6.1) zeigt diese Behauptung: die Sequenz, die j erzeugt, ist in der Tabelle 6.2 zusammengefasst, wobei die Tabelle 6.1 die im Folgenden verwendete Grammatik visualisiert.

Zum leichteren Verständnis werden nun die ersten Folgenglieder näher erläutert: Die Nichtterminalsymbole dieser speziellen Grammatik lauten <expr> (dieses ist zugleich das Startsymbol und steht zunächst für einen beliebigen mathematischen Ausdruck), <op> (oder Operation), <func> (eine mathematische Funktion aus einer gegeben Funktionenmenge) sowie <digit> (eine Zahl zwischen 0 und 9). Jedes dieser Nichtterminalsymbole kann auf eine gegebene Menge Nichtterminalen oder Terminalen abgebildet werden; diese Menge steht jeweils zur rechten der Nichtterminalen. Bspw. kann <func> abgebildet werden auf einer der vier Funktionen sin, cos, exp oder log; diese vier Funktionen definieren das gegebene Funktionenrepertoire dieser Grammatik. Die Elemente der jeweiligen Wertemengen sind aufsteigend nummeriert: Die Operation + ist offensichtlich Element 0, die Operation ist Element 1 etc. Die Kardinalität dieser Wertemenge ist die Variable NR aus Gleichung (6.1). Die Grammatik hat vier Funktionen im Repertoire, d.h. $NR(<\text{func}>) = 4$. Mit diesem Wissen kann j nun dekodiert werden: Das erste Integer von j ist 16 mit $V = 16$. Gleichung (6.1) gibt nun folgende Rechnung vor: $16 \ mod \ 7 = 2$, d.h. das Startsymbol <expr> bildet ab auf das Element 2 der Wertemenge, also <func>(<expr>). Nun wird der nächste Eintrag des Vektors betrachtet, d.h. die 3 und man wendet wieder Gleichung (6.1) an. Der String wird dabei von links nach rechts gelesen, d.h. es wird zuerst das Symbol <func> abgebildet. Die Wertemenge von <func> hat Kardinalität $NR = 4$, also: $3 \ mod \ 4 = 3$ und der Operator <func> bildet ab auf die Logarithmusfunktion. Nun benötigen wir noch ein Argument für den Logarithmus: das nächste Integer ist 7 und für <expr> gilt: $NR = 7$, also

S::=	<expr>	(0)
<expr>::=	<expr> <op> <expr>	(0)
	|(<expr>)	(1)
	|<func>(<expr>)	(2)
	|<digit>	(3)
	|x	(4)
	|y	(5)
	|z	(6)
<op>::=	+	(0)
	|-	(1)
	|×	(2)
	|/	(3)
<func>::=	+	(0)
	sin	(0)
	|cos	(1)
	|exp	(2)
	|log	(3)
<digit>::=	0	(0)
	|1	(1)
	|2	(2)
	|3	(3)
	|4	(4)
	|5	(5)
	|6	(6)
	|7	(7)
	|8	(8)
	|9	(9)

Tabelle 6.1.: Die Grammatik aus Beispiel 6.1

$7 \; mod \; 7 = 0$, d.h. <expr> bildet ab auf <expr><op><expr> etc. An dieser Stelle soll nicht die gesamte Sequenz erläutert werden, laut Tabelle 6.2 ist j tatsächlich die Kodierung einer analytischen Funktion ist, von $M_i(x) = log(x^2)$. In der Sprechweise von GAs: Die Genotyp–Repräsentation kodiert also den Phänotyp–Ausdruck $log(x^2)$. Es ist zu beachten, dass j ganz im Sinne des Ansatzes von Evolutionärer Programmierung als stochastische Optimierungsmethode ein mit einem Pseudozufallszahlgenerator erstellter willkürlicher Vektor ist. □

Es ist leicht einzusehen, dass die Anwendung genetischer Operatoren auf dieses Verfahren leicht möglich ist. Gemäß der Definition der Operatoren werden beim Crossover zwei Integervektoren k und l zufällig ausgewählt und Teile beider Chromosomen ausgetauscht, wohingegen beim Mutationsoperator nur ein Vektor stochastisch modifiziert

wird. Im Folgenden werden die durch die Grammatik erstellten analytischen Funktionen *Modelle* genannt und es gelte die Schreibweise $M_i(x)$.

String	Chromosom	Operation
<expr>	16,3,7,4,10,28,24,1,2,4	$16 \equiv 2 \mod 7$
<func>(<expr>)	3,7,4,10,28,24,1,2,4	$3 \equiv 3 \mod 4$
log(<expr>)	7,4,10,28,24,1,2,4	$7 \equiv 0 \mod 7$
log(<expr><op><expr>)	4,10,28,24,1,2,4	$4 \equiv 4 \mod 7$
log(x<op><expr>)	10,28,24,1,2,4	$10 \equiv 2 \mod 4$
log(x^*<expr>)	28,24,1,2,4	$28 \equiv 0 \mod 7$
log(x^*<expr><op><expr>)	24,1,2,4	$24 \equiv 3 \mod 7$
log(x^*<digit><op><expr>)	1,2,4	$1 \equiv 1 \mod 10$
log(x^*1<op><expr>)	2,4	$2 \equiv 2 \mod 4$
log(x^*1^*<expr>)	4	$4 \equiv 4 \mod 7$
log(x^*1^*x)		

Tabelle 6.2.: Die Dekodierungssequenz aus Beispiel 6.1

Nun mag die Frage aufkommen, was passiert, wenn ein ungültiges Modell erstellt wurde. Ein ungültiges Modell wäre bspw. eine Funktion, die abhängt von mehr als einer Variablen und die Kodierung einer Differentialgleichung sein soll, die nur von einer Variablen abhängt (dies ist durchaus möglich, schließlich enthält obige Grammatik die Ausdrücke x, y und z). Für diesen und weitere Fälle wird das Chromosom verworfen und ein neues generiert, solange ein wohldefiniertes Modell erstellt wurde. Des Weiteren können sogenannte *wrapping events* implementiert werden. Wrapping events kehren zurück zum ersten Integer des Chromosoms und haben zwei sinnvolle Anwendungen: Einerseits falls das Ende des Chromosoms erreicht ist, aber noch nicht alle Nichtterminalsymbole auf ein Terminalsymbol abgebildet wurden; und andererseits zollen sie dem fundamentalen Grundgedanken evolutionärer Programmierung Rechnung: der Erhöhung der Fitness durch ein Ausschöpfen eines möglichst großen Bereichs der Definitionsmenge, also der Menge aller realisierbaren Modelle gegeben diese Grammatik. Durch wrapping events wird also die Wahrscheinlichkeit erhöht, eine möglichst große Anzahl verschiedener Modelle zu erzeugen, um damit eine gute Fitness der Lösung des gesamten dynamischen Systems zu erzielen.
Nach diesem einführenden Beispiel in das Konzept der GE soll nun gezeigt werden, wie dieses Konzept dazu verwendet werden kann, Differentialgleichungen zu lösen.

6.2. Anwendung: Lösen von Differentialgleichungen

Im Folgenden soll ein Verfahren vorgestellt werden, das nach Tsoulos und Lagaris [32] eine erfolgreiche Methode zur Lösung von sowohl gewöhnlichen (ODEs) als auch partiellen Differentialgleichungen (PDEs) sein soll. Die einfachste Form des Verfahrens dient der Lösung von linearen ODEs. Um eine gegebene ODE zu lösen, arbeitet die Methode wie ein GA: zunächst werden im Rahmen der Initialisierung die Grammatik, d.h. insbesondere das zur Verfügung stehenden Repertoire an Funktionen bestimmt sowie die Startpopulation generiert; im Anschluss kommt es zur Auswahl von Eltern–Vektoren, zur Anwendung von genetischen Operatoren und zur Fitnessauswertung; diese Schleifen werden ausgeführt bis ein Abbruchkriterium erfüllt ist und die ODE im besten Fall gelöst ist. Da in den vorangegangenen Kapiteln bisher ausschließlich Probleme betrachtet wurden, bei denen Funktionen an bestimmten Iterationslösungen ausgewertet werden, jetzt aber Funktionen selbst Iterationslösungen darstellen, liegt besonderes Augenmerk auf der Fitnessauswertung. Eine eindeutig lösbare ODE besteht aus zwei Teilen: der Differentialgleichung selbst sowie den Anfangs– bzw. Randbedingungen. Dementsprechend besteht die Auswertung des Fitnesswertes ebenso aus zwei Teilen: Es sei eine ODE in folgender allgemeiner Form

$$f\left(x, y, y', y'', \ldots, y^{(n-1)}, y^{(n)}\right) = 0,\ x \in [a, b]$$

mit Anfangsbedingungen

$$g_i\left(x, y, y', y'', \ldots, y^{(n-1)}\right)\Big|_{x=t_i} = 0, \quad i = 1, \ldots, n, \quad t_i \in \{a, b\}$$

gegeben.

Die Fitnessauswertung arbeitet mit den Schritten wie im Algorithmus 6.1.

Eine genauere Analyse dieses Algorithmus zeigt: offensichtlich besteht der letztendliche Fitnesswert v_i aus zwei Teilen, E und P. Dieser Aspekt rührt daher, dass die gegebene ODE ebenso aus zwei Komponenten besteht. Der Ausdruck

$$E(M_i) = \sum_{j=0}^{N-1} \left(f\left(x_j, M_i(x_j), M_i'(x_j), \ldots, M_i^{(n)}(x_j)\right)\right)^2$$

steht für die gegebene ODE selbst, nur dass die vorher erstellten Modelle M_i und ihre Ableitungen die Ableitungen der gesuchten Lösung in der ODE ersetzen und die Modelle an den Stützstellen ausgewertet und im Anschluss über alle N Stützstellen aufsummiert werden. Des Weiteren erfolgt im Anschluss eine Quadrierung dieser mit

Algorithmus 6.1 : Algorithmus zur Fitnessauswertung

1 Wähle N äquidistante Stützstellen
2 **for** *alle Chromosomen* i **do**
3 Konstruiere die jeweiligen $M_i(x)$
4 Berechne $E(M_i) = \sum_{j=0}^{N-1} \left(f\left(x_j, M_i(x_j), M_i'(x_j), \ldots, M_i^{(n)}(x_j)\right)\right)^2$
5 Berechne die sogenannte zugehörige Zielfunktion:
$P(M_i) = \lambda \sum_{k=1}^{n} g_k^2 \left(x, M_i, M_i', \ldots, M_i^{(n-1)}\right)\Big|_{x=t_k}, \quad \lambda > 0$
6 Berechne den Fitnesswert von i wie folgt: $v_i = E(M_i) + P(M_i)$
7 **end**

Hilfe der Modelle modifizierten ODE. Hier ist es wichtig darauf hinzuweisen, dass die gegebene ODE f in homogener Form gegeben ist. Implizit misst der Ausdruck $E(M_i)$ damit den quadratischen Abstand der Iterations– zur tatsächlichen Lösung. Analog dazu misst die Zielfunktion

$$P(M_i) = \lambda \sum_{k=1}^{n} g_k^2 \left(x, M_i, M_i', \ldots, M_i^{(n-1)}\right)\Big|_{x=t_k}, \quad \lambda > 0,$$

wie gut die Iterationslösung die Anfangsbedingungen erfüllt: Auch hier werden diese ersetzt durch die erstellen Modelle sowie ihre Ableitungen, dann quadriert und über die Anfangsbedingungen aufsummiert. Des Weiteren wird dieser Wert mit einem positiven Parameter λ gewichtet. Dieser liegt im Ermessen des Programmierers und spiegelt die Relevanz wider, die den Anfangsbedingungen bei der Lösung des Problems beigemessen wird. So stelle man sich eine einfache lineare Funktion durch eine einzige gegebene Anfangsbedingung vor; diese wäre eine perfekte Approximation an jene Anfangsbedingung mit quadratischem Abstand von Null. Daher erscheint es sinnvoll, dem Term $E(M_i)$ eine größere Relevanz zuzuordnen; Koza bspw. schlägt einen Wert $\lambda = 0,25$ vor, d.h. er misst dem Term E eine Relevanz von 75 Prozent und dem Term P eine Relevanz von 25 Prozent bei der Lösung einer ODE bei [22]. Der letztendliche Fitnesswert ergibt sich dann als Summe beider Komponenten. Durch die Messung der quadratischen Abstände ist ersichtlich: eine Funktion mit geringerem Wert v ist im Sinne des Algorithmus fitter als eine Funktion mit großem v.

Im Folgenden soll ein Beispiel zur Anwendung der Methode betrachtet werden. Es sei folgende ODE mit dazugehörigen Anfangswerten gegeben:

$$y'' + 100y = 0, \quad x \in [0,1], \quad y(0) = 0, \quad y'(0) = 10.$$

Das Verfahren soll mit 10 Stützstellen im Intervall $[0, 1]$ arbeiten und es sei der Beispielvektor $c = [7, 2, 10, 4, 4, 2, 12, 20, 30, 5]$ gegeben. Dieser Vektor ist die Kodierung für folgende Modellfunktion

$$M_c(x) = \exp(x) + \sin(x)$$

mit den ersten beiden Ableitungen

$$M_c'(x) = \exp(x) + \cos(x), \;\; M_c''(x) = \exp(x) - \sin(x).$$

Mit diesen Informationen lässt sich der Fitnesswert des Vektors c leicht berechnen:

$$\begin{aligned} E(M_c) &= \sum_{j=0}^{9} \left(101 \exp(x_j) + 99 \sin(x_j)\right)^2 \\ P(M_c) &= \lambda \left(\left(M_c(0) - y(0)\right)^2 + \lambda \left(M_c'(0) - y'(0)\right)^2\right. \\ &= \lambda \left(\left(\exp(0) + \sin(0) - 0\right)^2 + \left(\exp(0) + \cos(0) - 10\right)^2\right) = 65\lambda \\ \Rightarrow v_c &= \sum_{j=0}^{9} \left(101 \exp(x_j) + 99 \sin(x_j)\right)^2 + 65\lambda \end{aligned}$$

Dies ist eine sehr große Zahl und damit eine schlechte Iterationslösung, allerdings dient diese Rechnung auch nur als Beispiel und ist natürlich kein vollständiger Durchlauf des Algorithmus. Vielmehr wurde hier exemplarisch eine Auswertung des Fitnesswertes vorgenommen, aber keinerlei genetische Operatoren angewendet. Tsoulos und Lagaris haben dieses Verfahren mehrmalig auf verschiedene ODEs angewendet und experimentell untersucht. Dabei wurde das Verfahren auf neun verschiedene ODEs erster und zweiter Ordnung jeweils 30 Mal mit der folgenden Initialisierung angewendet: Mutationsrate: 5 Prozent, Replikationsrate: 10 Prozent, Population: 1000, Chromosomenlänge: 50, maximale Anzahl Generationen: 2000, Abbruchkriterium: Zielfitness von 10^{-7}. Eine dieser neun ODEs wird nun genauer betrachtet: sei $y'' = -100y, \;\; y(0) = 0, \;\; y'(0) = 10, \;\; x \in [0, 1]$ gegeben. Die analytische Lösung dieses Problems lautet $y(x) = sin(10x)$. Die beiden Autoren haben eine Iterationslösung

$$y_{22}(x) = sin((sin(-log(4)x((-cos(cos(exp(7)))exp(cos(6))) - 5))))$$

mit einem Fitnesswert von $v = 4200, 5$ in Generation 22 dokumentiert. In Generation 27 hat sich der Fitnesswert bereits auf $v = 517, 7$ erheblich verbessert; und in Generation 59 hat der Algorithmus die exakte Lösung gefunden. Ihre experimentellen Resultate für ODEs haben die beiden Autoren in Tabelle 6.3 zusammengefasst.

ODE	min	max	avg
ODE1	8	1453	653
ODE2	52	1816	742
ODE3	23	1598	705
ODE4	14	1158	714
ODE5	89	1189	441
ODE6	37	1806	451
ODE7	42	1242	444
ODE8	3	702	66
ODE9	59	1050	411

Tabelle 6.3.: Experimentelle Resultate für ODEs

Hierbei geben die Spalten die minimale, die maximale sowie die durchschnittliche Anzahl an Generationen, d.h. an Durchläufen des Algorithmus an. Es ist bemerkenswert, dass die maximale Anzahl in jedem Fall die Zahl 2000 unterschreitet, d.h., dass das Verfahren die korrekte analytische Lösung in allen Fällen gefunden hat.

6.3. Die Methode zur Lösung von PDEs

Wie bereits erwähnt, lässt sich dieses Verfahren mit kleinen Modifikationen leicht auf die Lösung von PDEs übertragen. Dieses modifizierte Verfahren soll nun im Folgenden kurz vorgestellt werden. Da das Verfahren für PDEs große Ähnlichkeiten zu dem Ansatz für lineare ODEs aufweist, sei hier nur auf den grundätzlichen Algorithmus 6.2 verwiesen; dieser arbeitet wie nachfolgend beschrieben.

Laut Tsoulos und Lagaris [32] ist die Übertragung auf PDEs höherer Ordnung mit anderen Randbedingungen leicht möglich. Neben der Anwendung von GE zur Lösung von linearen ODEs sowie PDEs, lässt sich das Konzept ebenso leicht auf nichtlineare ODEs sowie auf Systeme von Differentialgleichungen übertragen. Die Experimente der Autoren zeigen, dass — falls eine Lösung in geschlossener Form existiert und falls die verfügbare Grammatik über ein ausreichendes Repertoire an Funktionen verfügt — die Methode diese Lösung findet. Bemerkenswert ist weiterhin, dass für den Fall, dass keine Lösung analytisch anzugeben ist, diese Methode eine Approximationslösung in geschlossener Form angibt. Nicht weniger beachtlich ist, dass das Verfahren bereits erfolgreich in verschiedenen Anwendungsgebieten wie in der Symbolischen Regression, bei der Vorhersage von Finanzdaten (siehe Kapitel 7) sowie in der Robotik eingesetzt wurde. Darüber hinaus forschen die Autoren an weiteren PDEs mit modifizierten Grammatiken.

Algorithmus 6.2 : Lösung von PDEs

Data : zur Vereinfachung nur elliptische PDEs in zwei und drei Variablen mit Dirichlet-Randbedingungen:
$f\left(x, y, \Psi(x,y), \frac{\partial}{\partial x}\Psi(x,y), \frac{\partial}{\partial y}\Psi(x,y), \frac{\partial^2}{\partial x^2}\Psi(x,y), \frac{\partial^2}{\partial y^2}\Psi(x,y)\right) = 0$, wobei $x \in [x_0, x_1], y \in [y_0, y_1]$ und Dirichlet-Randbedingungen
$\Psi(x_0, y) = f_0(y), \Psi(x_1, y) = f_1(y), \Psi(x, y_0) = g_0(x), \Psi(x, y_1) = g_1(x)$

1 Wähle N^2 äquidistante Stützstellen aus dem Rechteck $[x_0, x_1] \times [y_0, y_1]$, davon N_x äquidistante Punkte auf dem Rand in $x = x_0$ und in $x = x_1$ sowie N_y Punkte auf dem Rand in $y = y_0$ und in $y = y_1$.
2 **for** *alle Chromosomen* i **do**
3 Erstelle ein Modell $M_i(x, y)$
4 Berechne die Größe:

$$E(M_i) = \sum_{j=1}^{N^2} f\Big(x_j, y_j, M_i(x_j, y_j), \frac{\partial}{\partial x} M_i(x_j, y_j), \frac{\partial}{\partial y} M_i(x_j, y_j), \frac{\partial^2}{\partial x^2} M_i(x_j, y_j), \frac{\partial^2}{\partial y^2} M_i(x_j, y_j)\Big)^2 \tag{6.2}$$

5 Berechne die zugehörigen Zielfunktionen:
$P_1(M_i) = \sum_{j=1}^{N_x} \left(M_i(x_0, y_j) - f_0(y_j)\right)^2$, $P_2(M_i) = \sum_{j=1}^{N_x} \left(M_i(x_1, y_j) - f_1(y_j)\right)^2$,
$P_3(M_i) = \sum_{j=1}^{N_y} \left(M_i(x_j, y_0) - g_0(x_j)\right)^2$, $P_4(M_i) = \sum_{j=1}^{N_y} \left(M_i(x_j, y_1) - g_1(x_j)\right)^2$
6 Berechne den Fitnesswert von i wie folgt: $v_i = E(M_i) + \lambda \sum_{l=1}^{4} P_l(M_i)$.
7 **end**

6.4. Genetische Programmierung

Wie bereits im ersten Abschnitt erwähnt, sind Genetische Algorithmen als ein Teil dessen zu sehen, was man unter der Terminologie *Evolutionary Computing* subsumieren kann; neben den bereits bekannten GAs ist ein weiteres Mitglied dieser Familie das sogenannte Genetic Programming, im Folgenden GP genannt. In den folgenden Abschnitten soll gezeigt werden, dass es sinnvoll ist, den Fokus weg von GAs hin auf diesen neuen Ansatz zu lenken, da mit letzterem die numerische Lösung von dynamischen Systemen bedeutend einfacher sein kann. Zwischen den beiden Konzepten GA und GP existieren zwei fundamentale Unterschiede, wobei sich der hohe Verwandtschaftsgrad zwischen GAs und GP im ersten technischen Unterschied wiederspiegelt: Genetische Programmierung ist aus der Implementierungsperspektive betrachtet prinzipiell ein GA, mit dem Unterschied, dass die Kodierung nicht mit Hilfe von Bit Strings, sondern mit Hilfe von *Parse Trees* oder auch *Syntaxbäumen* erfolgt. Ein Beispiel im folgenden Abschnitt wird diese

technische Feinheit genauer beleuchten. Der zweite fundamentale Unterschied zwischen GP und GA betrifft weniger die technischen Attribute als die Art der Probleme, zu deren Lösung beide Konzepte sinnvoll eingesetzt werden: Wie bereits in der einleitenden Motivation dieser Arbeit dargestellt, werden GAs generell dazu eingesetzt, eine gegebene Funktion numerisch zu optimieren. Jene Funktion kann durchaus beliebig komplex in Bezug auf ihre Optimierbarkeit sein; d.h. sie muss weder differenzierbar, noch stetig oder in analytischer Form gegeben sein. Sie kann sogar einen diskreten Definitionsbereich besitzen, wie es beim Travelling Salesman Problem der Fall ist. In all diesen Fällen jedoch ist eine Funktion, d.h. ein gegebenes Problem vorhanden, das mittels eines GAs numerisch optimiert werden soll. GP verwendet dagegen einen anderen Ansatz: Generell wird dieses Konzept dazu eingesetzt, ein bestimmtes abstraktes Modell, z.B. eine Funktion oder eine Formel zu finden, die eine maximale Fitness aufweisen. Vor dem Hintergrund, dass GP dazu eingesetzt werden soll, dynamische Systeme zu lösen, erscheint dieser neue Ansatz damit durchaus vielversprechend; schließlich ist eine Differentialgleichung eine Gleichung, deren Lösungen unbekannte Funktionen sind. Ein Beispiel soll diese durchaus abstrakte Grundüberlegung konkretisieren.

Beispiel 6.3 (Bonitätsproblem)

Um Kreditausfallrisiken zu minimieren, müssen Banken die Zahlungszuverlässigkeit ihrer Kunden, d.h. der Kreditnehmer untersuchen und pflegen. Diese Information soll dazu genutzt werden, um ein Modell erstellen, das sogenannte *gute* von *schlechten Kunden* abgrenzt. In einem weiteren Schritt kann das Modell dazu genutzt werden, die zukünftige Zahlungsmoral von Bestands– und Neukunden vorauszusagen. Das generierte Modell kann verschiedene, zur Vorhersage von Zahlungsausfällen nützliche Inputdaten verwenden, bspw. das jährliche Gehalt, die Anzahl Kinder des Kunden, den Familienstand etc. Das simple Modell soll einen binären Output bestimmen, wobei 0 eine schlechte und 1 eine gute Zahlungsmoral kodiert. Selbstverständlich arbeiten Kreditinstitute und Ratingagenturen in der Praxis mit weitaus mehr Inputvariablen und einer größeren Klassifizierungsmenge — vergleiche z.B. die Bandbreite an Bonitätsnoten von D bis AAA der größten Ratingagentur Standard and Poors — das einfache Beispiel hier erfüllt dennoch seinen Zweck, den Begriff *Modell* im Kontext von GP zu erklären. Tabelle 6.4 fasst dieses konstruierte Beispiel zusammen.

Kundennr	Janresgehalt brutto	Anzahl Kinder	Familinstand	Bonität
00001	45000	2	verheiratet	0
00002	30000	0	ledig	1
00003	40000	1	verheiratet	1
00004	60000	2	geschieden	1
⋮	⋮	⋮	⋮	⋮
10000	50000	2	verheiratet	1

Tabelle 6.4.: Beispieldaten für das Bonitätsproblem

Ein mögliches Modell, das das Problem der Bonitätsvorhersage löst, ist z.B. folgendes:

$$IF(Anzahl\ Kinder = 2)AND(Jahresgehalt > 80.000)THEN\ gut\ ELSE\ schlecht.$$

Abstrakter kann man das allgemeine Modell zur Lösung des Problems wie folgt beschreiben:

$$IF\ Formel\ THEN\ gut\ ELSE\ schlecht.$$

Die Variable *Formel* ist die einzige Unbekannte in diesem Modell. Das Ziel in diesem Problem ist es also, die optimale Formel zu finden; diese wiederum bestimmt das optimale Modell und damit die Lösung des Bonitätsproblems. Technisch ist das Bonitätsproblem als Suchproblem über den Raum aller möglichen Formeln definiert. Eine genaue Definition dieses Raumes soll hier außer Acht gelassen werden; vielmehr steht hier die motivierende Beschreibung des Problems im Mittelpunkt.

Ausführungen zur Kodierung folgen im nächsten Abschnitt. Jedoch kann man die Fitness dieses Modells beschreiben: Die *Qualität einer Formel* wird definiert als der Anteil der durch das obige Modell korrekt klassifizierten Kunden. Dieses Beispiel legt nahe, dass es auch vor dem Hintergrund von GP die bekannten Metaphern aus der Darwinschen Evolutionsbiologie gibt, schließlich liegen GP und GAs als Teile der Evolutionary Computing–Familie nicht allzu sehr auseinander. Tabelle 6.5 fasst diesen Aspekt kurz zusammen.

Kodierung	Syntaxbäume
Crossover	Austausch von Teilbäumen
Mutation	Zufallstausch im Syntaxbaum
Phänotyp	Formel
Chromosom, Gen	Syntaxbaum, Knoten

Tabelle 6.5.: Zusammenfassung der wichtigsten Attribute der GP

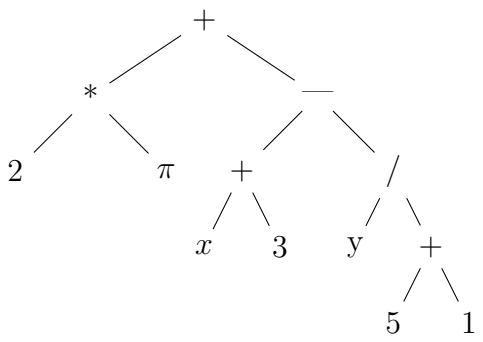


Abbildung 6.1.: Syntaxbaumkodierung eines arithmetischen Ausdrucks

Kodierung:
Wie bereits erwähnt erfolgt die Kodierung bei GP methodisch mit Hilfe von Syntaxbäumen. Prinzipiell ist die Kodierung von vielen verschiedenen Ausdrücken denkbar je nach Problemstellung und Syntax. Hervorzuheben hierbei sind drei zentralen Klassen von Ausdrücken: arithmetische Formeln, logische Formeln sowie Programmcode; diese Ausdrücke kommen besonders häufig zum Einsatz, weshalb deren Kodierung im Folgenden (teilweise) im Fokus stehen soll.

Betrachte folgenden einfachen arithmetischen Ausdruck:

$$(+\ (*\ 2\ \pi)\ -(+\ x\ 3)\ (/\ y\ (+\ 5\ 1))) \Rightarrow 2\pi + \left((x+3) - \tfrac{y}{5+1}\right)$$

Die erste Teil vor dem Implikationspfeil gibt den Ausdruck in sogenannter *symbolic expression* an; wie man anhand von Abbildung 6.1 erkennen kann, ermöglicht diese Notation einen leichten Zugang zur äquivalenten Kodierung als Syntaxbaum: Das + ist der globale Operator, daher thront er als Wurzel des Baumes ganz oben. Links und rechts verzweigen die beiden Summanden 2π bzw. $(x+3) - (y/(5+1))$ als Äste ab, wobei der erste Summand mit den beiden Blättern 2 und π beendet ist, wohingegen der rechte Ast am Knoten weiter verzweigt wird.

Die Kodierung von logischen Formeln sowie von Programmcode ist sehr ähnlich; für eine nähere Erklärung siehe (Eiben und Smith [6]).

Im Folgenden soll noch kurz auf die genetischen Operatoren Mutation sowie Crossover eingegangen werden. Analog zu GAs erfolgt der Mutationsvorgang über den zufälligen Austausch einer Teilmenge von Genen bei einem Exemplar der bestehenden Population. Der einzige Unterschied zum GA ist, dass die Gene hier nicht aus Strings, sondern aus Teilbäumen bestehen: Es wird zufällig ein Knoten des Syntaxbaumes, der die Elterngeneration repräsentiert, ausgewählt und durch einen wiederum zufällig generierten

neuen Teilbaum ersetzt. Abbildung 6.2 visualisiert diesen Prozess anhand des obigen arithmetischen Ausdrucks.

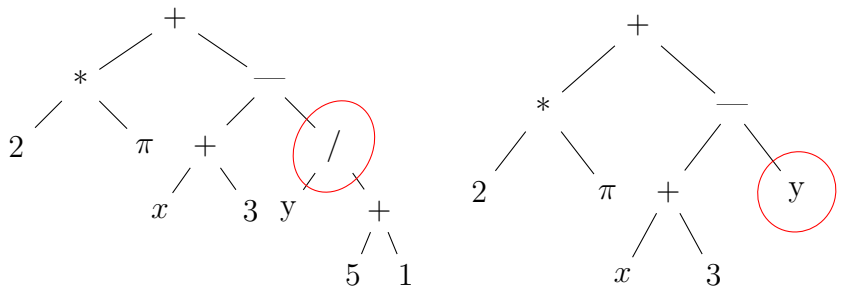


Abbildung 6.2.: Syntaxbaumkodierung und Mutation

Der Mutationsvorgang bei GP hat also zwei Parameter: einerseits die Mutationsrate, d.h. die Wahrscheinlichkeit, mit der ein Gen verändert wird sowie die Wahrscheinlichkeit, mit der ein Knoten des Syntaxbaumes als Wurzel des neuen Teilbaums ausgewählt wird. Darüber hinaus ist es erwähnenswert, dass damit die neue Teilbaumlänge die Länge des Elternbaumes übersteigen kann.

Im Folgenden soll noch ein bestimmter Rekombinationsvorgang betrachtet werden, das sogenannte *Subtree Crossover*. Auch hier läuft der Vorgang prinzipiell analog zum Crossover bei GAs ab: Es werden im Rahmen der Vorauswahl zwei Elternbäume ausgewählt und Teile beider Individuen werden ab einem zufällig ausgewählten Knoten getauscht, wie Abbildung 6.3 unterstreicht.

$$2\pi + \left((x+3) - \tfrac{y}{5+1}\right) \quad (Parent1) \Rightarrow 2\pi + ((x+3) - 3a) \quad (Child1)$$
$$3a(3 + (y+12)) \quad (Parent2) \Rightarrow \left(\tfrac{y}{5+1}\right)(3 + (y+12)) \quad (Child2)$$

Auch der Rekombinationsvorgang beim GP besteht damit aus zwei Parametern: der Replikationsrate, wobei diese die Wahrscheinlichkeit angibt, mit der Crossover auf ein Chromosomen angewendet wird; und der Wahrscheinlichkeit, mit der ein innerer Knoten jedes Elternbaumes als Crossover-Punkt ausgewählt wird. Es gilt auch hier, dass die neue Teilbaumlänge die der Elternbäume übersteigen kann. Es wurde also gezeigt, dass GP und GA tatsächlich zwei sehr ähnliche Konzepte darstellen. Sieht man von der unterschiedlichen Datenstruktur beider Verfahren ab, so ist der einzige technische fundamentale Unterschied, dass bei GP die Chromosomen in ihrer Länge — gemessen anhand der Anzahl an Knoten - variieren können, wohingegen die Stringlänge bei GAs in der Regel fix ist (Eiben und Smith [6]).

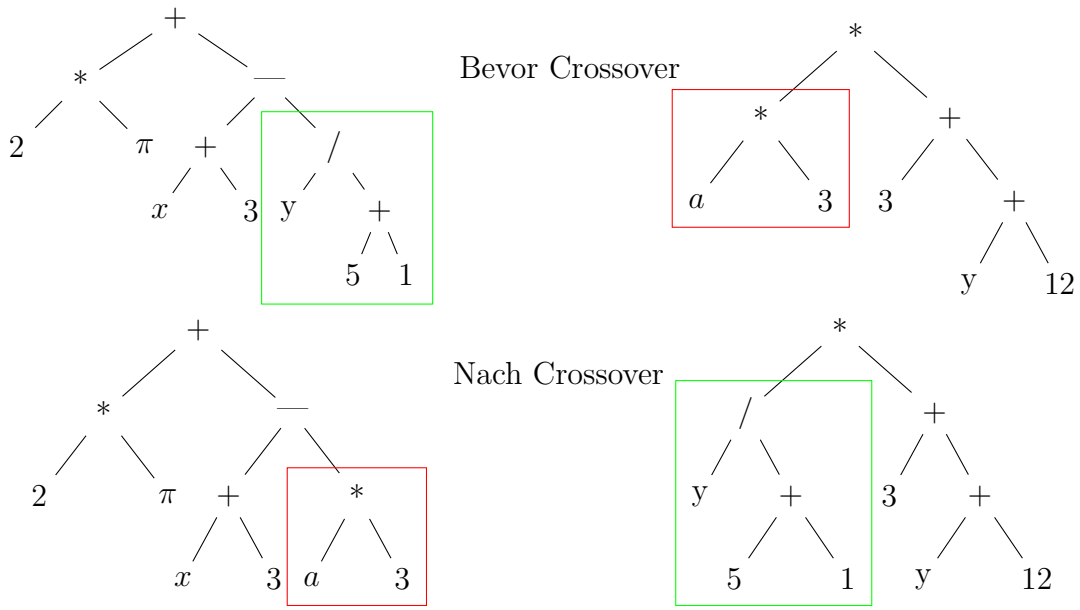


Abbildung 6.3.: Syntaxbaumkodierung und Subtree Crossover

Diese intendiert kurze Einführung in die GP und ihren Instrumenten bildet die Grundlage zum besseren Verständnis für die folgenden Anwendungen auf numerische Lösungsmethoden von Konvektions-Diffusions-Gleichungen.

6.5. GP zur Lösung der Konvektion–Diffusion–Gl.

Christian Loenser

Genetic Programming (GP) soll nun dazu verwendet werden, die Konvektion–Diffusion–Gleichung (KDG) in einer Variablen zu lösen. Der Zweck der Methoden, die im weiteren Verlauf vorgestellt werden, soll es nicht sein, eine eindimensionale KDG zu lösen, da deren Lösung einfach zu bestimmen ist. Die Methoden sollen stattdessen komplexere (partielle) Differentialgleichungen lösen, werden aber zunächst an der KDG getestet. Die KDG, die hier gelöst wird, lautet

$$\begin{aligned} \frac{\partial^2 T}{\partial x^2} - Pe\frac{\partial T}{\partial x} &= 0 \qquad (6.3)\\ x &\in [0,1]\\ T(0) &= 1\\ T(1) &= 0 \end{aligned}$$

mit der Peclet–Zahl Pe. Die analytische Lösung T ist gegeben durch

$$T(x) = \frac{\exp(xPe) - \exp(Pe)}{1 - \exp(Pe)} \tag{6.4}$$

Die folgende Abbildung 6.4 zeigt die Lösungen der KDG für verschiedene Peclet–Zahlen.

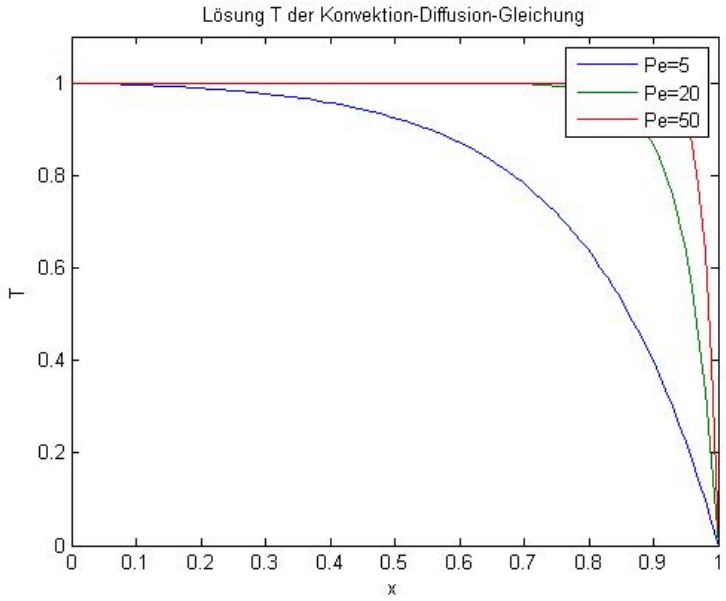


Abbildung 6.4.: Analytische Lösung der KDG

In den nächsten drei Abschnitten werden drei Methoden beschrieben, wie man eine (zumindest näherungsweise) Lösung der obigen KDG mittels GP berechnen kann. Die drei Methoden basieren auf einer Approximation der analytischen Lösung durch mathematische Funktionen wie z.B. Polynome, Sinus–, oder Exponentialfunktionen. Die erste Methode ist von Howard und Roberts [16] und legt bereits zu Beginn die Art der approximierenden Funktion als Polynom mit variabler Länge fest, das die Randwertbedingungen a priori erfüllt. Aufgabe der GP–Methode ist es dann, die Koeffizienten und somit den Grad eines Polynoms so zu bestimmen, dass das Polynom eine möglichst gute Approximation darstellt. Die zweite Methode, die von Koza [22] stammt, legt die approximierende Funktion nicht so starr fest. Diese GP–Methode findet eine Komposition von typischen mathematischen Funktionen (+, -, exp, sin, ...) mit der Variablen x als approximierende Lösung. Die Randwertbedingungen sind in dieser Variante nicht a priori erfüllt. In der dritten Methode, die aus Howard und Kolibal [15] ist, wird die GP–Methode mit stochastischer Bernstein Interpolation durchgeführt. Im weiteren Verlauf

werden die drei Methoden genauer vorgestellt.

6.5.1. Methode nach Howard und Roberts

Howard und Roberts [16] legen die approximierende Funktion $\hat{T}$ a priori durch

$$\hat{T}(x) = x(1-x)p(x) + (1-x) \tag{6.5}$$

mit

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \ldots$$

fest, wobei p ein Polynom variabler Länge ist. Die Wahl dieser Definition hat zwei Vorteile. Die Randwertbedingungen sind durch diese Definition a priori erfüllt, da

$$\hat{T}(0) = 0(1-0)p(0) + (1-0) = 1$$
$$\hat{T}(1) = 1(1-1)p(1) + (1-1) = 0$$

ist. Da (6.5) ein Polynom ist, sind die ersten beiden Ableitungen ebenfalls Polynome:

$$\frac{\partial \hat{T}}{\partial x} = x(1-x)\frac{\partial p}{\partial x} + (1-2x)p - 1$$
$$\frac{\partial^2 \hat{T}}{\partial x^2} = x(1-x)\frac{\partial^2 p}{\partial x^2} + 2(1-2x)\frac{\partial p}{\partial x} - 2p$$

Diese Tatsache wird sich bei der Bestimmung der Fitness noch als nützlich erweisen. Die Aufgabe der GP-Methode ist es nun, Koeffizienten

$$[a_0, a_1, a_2, \ldots, a_n]$$

und somit ein Polynom p vom Grad $n \in \mathbb{N}$ mit

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

zu finden, so dass die entsprechende Funktion $\hat{T}$ eine gute approximative Lösung der KDG ist.

Die Methode von Howard und Roberts nutzt GP, um die KDG zu lösen. Man könnte das Prinzip dieser Methode jedoch auch nutzen, um die KDG mit GA zu lösen. Da

das Polynom von variabler Länge ist und die Methode selbst einen Grad festlegen soll, muss man eine GP–Methode anwenden. Wenn aber der Grad und somit die Anzahl der Koeffizienten vorab festgelegt wird, kann man auch GA anwenden, um die KDG zu lösen.

Im weiteren Verlauf wird die genaue Durchführung der GP–Methode beschrieben. Die approximierenden Funktionen werden, wie es in GP üblich ist, durch Bäume dargestellt. Jeder Baum repräsentiert also jeweils einen Lösungsvorschlag $\hat{T}$, so dass es das Ziel der GP–Methode ist, den Baum mit der besten Fitness zu finden. Für die Methode von Howard und Roberts bedeutet dies, dass jeder Baum einen Koeffizientenvektor repräsentiert. Die GP–Methode durchläuft folgende standardmäßige Schritte 6.3:

Algorithmus 6.3 : Ablauf der GP

1 Setzte $k = 1$ und bestimme max. Anzahl von Durchläufen k_{max}
2 Erstelle eine Startpopulation von Bäumen
3 Berechne Fitness der Bäume der Startpopulation und speichere den fittesten in $\hat{T}$
4 **while** $\hat{T}$ *keine ausreichende Approximation und* $k < k_{max}$ **do**
5 GP–Operatoren erzeugen neue Generation von Bäumen
6 Berechne Fitness jedes Baumes der Generation und speichere den fittesten in $\hat{T}$
7 Setzte k=k+1
8 **end**

Die einzelnen Schritte werden nun von uns näher erläutert.

Schritt 1 und 4: Abbruchkriterium

Die obige GP Methode berechnet Approximationen, bis eine von ihnen gut genug oder eine Höchstzahl an Durchläufen erreicht worden ist. Howard und Roberts geben in ihrer Arbeit jedoch nicht an, welche Höchstzahl sie verwendet haben.

Schritt 2: Startpopulation

Bevor die Startpopulation erstellt wird, müssen einige Parameter der Methode bestimmt werden. Die folgende Tabelle 6.6 zeigt die Wahl von Howard und Roberts.

Parameter	Wahl
Population	8000
Blätter (terminals)	$\left\{0, \frac{1}{255}, \frac{2}{255}, \ldots, 1\right\}$
Funktionen	+, -, *, / (x/0:=1)
	ADD, WRITE, BACK
	Wm_1, Wm_2, Rm_1, Rm_2, m_1, m_2
max. Baumgröße	1000

Tabelle 6.6.: Parameter

Howard und Roberts wählen eine Größe der Startpopulation von 8000 Bäumen, die jeweils eine approximative Lösung $\hat{T}$ darstellen. Die Blätter (terminals) ihrer Bäume sind numerische Konstanten aus der Menge $\left\{0, \frac{1}{255}, \frac{2}{255}, \ldots, 1\right\}$. Die restlichen Knoten stellen Funktionen dar, die die Konstanten verändern (mathematische Grundfunktionen +, -, *, / mit x/0:=1) und sie in den Vektor des jeweiligen Baumes schreiben (übrige Funktionen). Zwei Zeiger C und L, die auf Stellen des Vektors zeigen und durch die Funktionen verschoben werden können, bestimmen, an welche Stellen die Werte geschrieben werden. Ein festzulegender Wert L_{max} begrenzt die Länge des Koeffizientenvektors.
Als nächstes werden die Funktionen erklärt, die keine mathematischen Grundfunktionen sind. Die folgende Graphik zeigt ein Beispiel eines Baumes, mit dessen Hilfe zwei Funktionen anschaulich erklärt werden, wobei die Zeiger zu Beginn $L = C = 0$ sind und $L_{max} > 1$ beliebig ist.

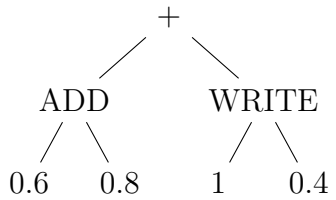


Abbildung 6.5.: Beispiel eines Baumes

Die Funktion ADD hat zwei Argumente, übergibt ihr zweites und schreibt ihr erstes Argument an die Stelle des Vektors, auf den der Zeiger L zeigt. Anschließend wird L erhöht, sofern $L < L_{max}$ gilt, und die Zeiger gleichgesetzt ($C = L$). Dies bedeutet für das Beispiel, dass ADD 0.8 an die Funktion + übergibt, den Wert 0.6 an die erste Stelle des noch leeren Koeffizientenvektors schreibt ($L = 0$) und $C = L = 1$ setzt. Die Funktion WRITE verhält sich ähnlich. Sie hat zwei Argumente, übergibt ihr zweites und schreibt ihr erstes Argument an die Stelle C des Vektors. Im Fall $C < L_{max}$ wird

C und im Fall $C > L$ zusätzlich noch der Zeiger L um eine Einheit erhöht. WRITE übergibt im Beispiel also 0.4 an die Funktion +, schreibt den Wert 1 an die zweite Stelle des Koeffizientenvektors ($C = 1$) und erhöht die Zeiger auf $C = L = 2$. Der Koeffzientenvektor des Baumes ist also $[0.6, 1]$ und die zugehörige Approximation somit $\hat{T}(x) = x(1-x)(0.6+1x) + (1-x)$ mit $p(x) = 0.6 + 1x$.

Die Funktion BACK hat ebenfalls zwei Argumente. Im Gegensatz zu den vorherigen beiden Funktionen schreibt BACK jedoch keines ihrer Argumente in den Koeffizientenvektor, sondern übergibt lediglich ihr erstes Argument und verringert den Zeiger C um eine Einheit ($C > 0$ vorausgesetzt). Die übrigen Funktionen haben einen anderen Zweck. Die Funktion Wm_1 bzw. Wm_2 hat zwei Argumente, übergibt ihr erstes und speichert ihr zweites Argument in m_1 bzw. m_2. Der gespeicherte Wert wird von der Funktion Rm_1 bzw. Rm_2 abgerufen, die ihrerseits ihre Argumente ignoriert.

Als nächstes wird noch erklärt, wie ein Baum genau entsteht. Howard und Roberts legen eine maximale Baumgröße von 1000 Knoten fest. In der Literatur findet man häufig zwei Methoden, um einen Baum aufzubauen (Koza [22]). Eine Möglichkeit ist es, so lange gleichmäßig Knoten an einen Baum anzufügen, bis dieser die maximale Größe erreicht hat. In dieser Full Method genannten Variante haben alle Bäume die maximale Größe. Die zweite Möglichkeit wird Grow Method genannt und lässt Bäume unterschiedlicher Größe zu. Es wird zunächst eine Funktion als Wurzel gewählt. Der nächste Knoten ist dann mit einer zu wählenden Wahrscheinlichkeit P ein Knoten, der kein Blatt ist (also eine Funktion), und mit einer Wahrscheinlichkeit 1-P ein Blatt (also eine Konstante). Dieses Vorgehen wird durchgeführt, bis nur noch Blätter am Ende des Baumes vorzufinden sind. Da die durchschnittliche Baumgröße, wie man unten bei der Darstellung der Ergebnisse sehen kann, bei den verschiedenen Durchläufen von Howard und Roberts variieren, verwenden beide nicht die Full Method, um ihre Bäume aufzustellen.

Schritt 3 und 6: Fitnessfunktion

Die GP–Methode soll einen Baum bzw. eine Approximation finden, die die KDG möglichst gut löst. Die Güte einer Approximation wird durch die Fitnessfunktion bestimmt. Howard und Roberts wählen dafür folgende Funktion:

$$F = -\int_0^1 \left(\frac{\partial^2 \hat{T}}{\partial x^2} - Pe \frac{\partial T}{\partial x} \right)^2 dx \tag{6.6}$$

Der Integrand entspricht dem quadratischen Fehler (im Vergleich zur rechten Seite der KDG), den die Approximation $\hat{T}$ beim Einsetzen in die KDG verursacht. Ein höherer

Fitnesswert bedeutet in diesem Fall eine bessere Approximation. Die analytische Lösung der KDG ist die Maximalstelle der Fitnessfunktion und führt zu $F = 0$. Eine Betrachtung der Definition der Fitnessfunktion macht deutlich, warum Howard und Roberts die obige Darstellung der Approximation $\hat{T}$ gewählt haben. Da $\hat{T}$ ein Polynom ist, sind die Ableitungen ebenfalls Polynome und die Fitnessfunktion ist somit analytisch berechenbar.

Schritt 5: GP–Operatoren

Wenn innerhalb einer Generation keine ausreichend gute Approxmation gefunden worden ist, wird mittels GP–Operatoren eine neue Generation erstellt. Die Details zu den GP–Operatoren sind in der folgenden Tabelle dargestellt.

Parameter	Wahl
Operatoren	80% Crossover, 20% Clone
Kill Tournament	Größe: 2
Breed Tournament	Größe: 4

Tabelle 6.7.: Parameter zu GP-Operatoren

Howard und Roberts verwenden den Crossover– und den Clone Operator. Bei Verwendung des Crossover Operators wird in zwei Bäumen jeweils ein Knoten zufällig gewählt und die darunterliegenden Subbäume getauscht. Beim Clone Operator wird ein gewählter Baum direkt in die neue Generation übernommen. Die Auswahl der Bäume für diese Operatoren erfolgt durch ein Breed Tournament der Größe vier. Es werden in diesem Fall vier Bäume zufällig aus der aktuellen Generation gezogen, wobei der fitteste von ihnen für den Operator gewählt wird. In der klassischen Variante des GP kommen die gewählten Bäume in eine zunächst leere Generation, bis diese so viele Bäume wie die vorherige Generation enthält. In der Steady–State Variante, die Howard und Roberts verwenden, werden die Bäume wieder in die alte Generation zurückgesetzt, wobei jeder zurückeingesetzte Baum einen Baum aus der alten Generation verdrängt. In einem Kill Tournament werden zwei Bäume aus der alten Generation gezogen und der schwächste Baum wird ersetzt. Eine neue Generation ist dann entstanden, wenn die Methode genau so viele Bäume eingesetzt hat, wie in der alten Generation gewesen sind (siehe z.B. Kinnear [20]).

Ergebnisse

Howard und Roberts haben die Methode mit den obigen Einstellungen verwendet, um eine Lösung der KDG für verschiedene Peclet–Zahlen Pe zu finden. Die Ergebnisse der

erfolgreichen Durchläufe sind in den drei folgenden Tabellen 6.8 — 6.10 dargestellt.

Pe	pop	gens	best F	avg tree	mins
5	8000	42	-0.000175	55	24.23
20	8000	80	-0.003145	348	72.11
50	8000	80	-0.042962	830	139.61
100	2000	300	-0.258476	958	228.86

Tabelle 6.8.: Informationen

I	a_{I+0}	a_{I+1}	a_{I+2}	a_{I+3}
a_0	0.964706	0.882353	0.713725	0.717647
a_4	0.170588	0.137255	0.447377	

Tabelle 6.9.: **Pe=5**, 7 Koeffizienten

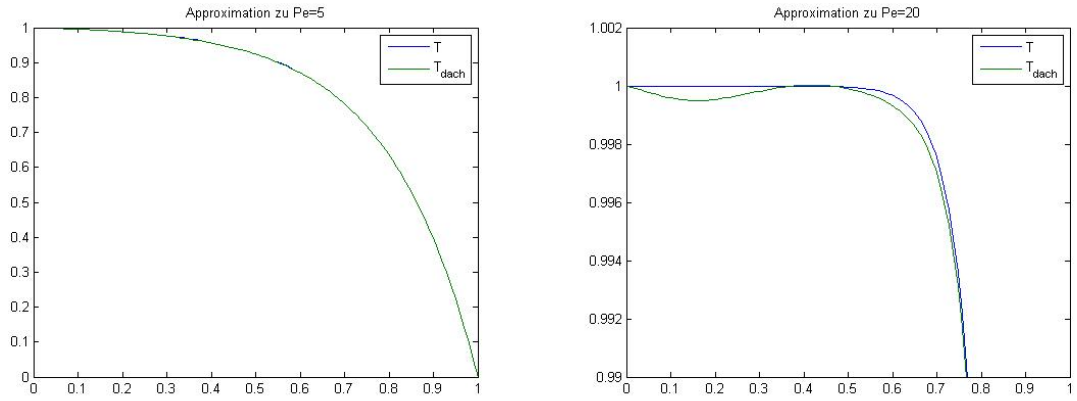
I	a_{I+0}	a_{I+1}	a_{I+2}	a_{I+3}
a_0	0.996078	0.972549	1.24314	0.615686
a_4	0.752941	1.77528	0.980392	0.588235
a_8	0.745098	0.611765	2.24359	0.931927
a_{12}	0.586275	0.858824	0.462745	0.92549
a_{16}	0.74902	0.374761	0.374761	0.733333
a_{20}	0.374761	0.410748	0.0313725	0.588235
a_{24}	0.0313725	0.0313725		

Tabelle 6.10.: **Pe=20**, 26 Koeffizienten

Die erste Tabelle zeigt die gewählte Populationsgröße, die Anzahl der benötigten Generationen, den höchsten erzielten Fitnesswert, die durchschnittliche Baumgröße und die Dauer eines Durchlaufes für die verschiedenen Peclet–Zahlen. Eine Approximation der analytischen Lösung durch Polynome wird, wie man an der obigen graphischen Darstellung sieht, mit steigender Peclet–Zahl immer schwieriger, da die Steigung in der Nähe von $x = 1$ vom Betrag her immer größer wird und im Bereich vorher gegen Null geht. Dieses Problem spiegelt sich in den Ergebnissen der GP–Methode wider. Wenn die Peclet–Zahl steigt, sinkt die Fitness der besten Approximation eines Durchlaufes. Die

analytische Lösung kann also immer schlechter durch die Polynome der GP–Methode approximiert werden. Neben der Abweichung von der analytischen Lösung steigt auch der Aufwand der Methode mit zunehmender Peclet–Zahl. Um eine ausreichende Approximation zu erlangen, werden mehr Generationen und Zeit benötigt. Der Grund für diesen Anstieg ist, dass mit zunehmendem Pe Polynome höheren Grades benötigt werden, um eine ausreichende approximative Lösung zu erhalten.

Die beiden anderen Tabellen zeigen jeweils die Koeffizienten für einen Durchlauf mit $Pe = 5$ und $Pe = 20$. Werden für $Pe = 5$ nur sieben Koeffizienten benötigt, sind es bei $Pe = 20$ schon 26. Obwohl die Anzahl der Koeffizienten steigt, sinkt die Fitness der besten Approximation, da die höhere Peclet–Zahl die Approximation erschwert. Wie erreicht die Methode, dass der Grad der Polynome mit zunehmendem Pe steigt? Ein Blick in die erste Tabelle zeigt, dass die durchschnittliche Größe eines Baumes mit zunehmendem Pe steigt. Der Grund dafür ist, dass größere Bäume während des Durchlaufes eine höhere Fitness aufweisen bei größerem Pe. Deshalb werden bei der Wahl für die GP–Operatoren größere Bäume tendenziell bevorzugt mit der Konsequenz, dass die durchschnittliche Größe in einem Durchlauf steigt. Die folgende Abbildung 6.6 vergleicht die beste gefundene Approximation zu $Pe = 5$ bzw. $Pe = 20$ mit der analytischen Lösung. Obwohl die Güte der Approximation mit zunehmendem Pe sinkt, liefert die GP–Methode nach Howard und Roberts trotzdem eine relativ gute Approximation der analytischen Lösung für $Pe \leq 20$, wie man in der Abbildung zu $Pe = 5$ und an der Skalierung der Achsen in der Abbildung für $Pe = 20$ sehen kann.



(a) $Pe = 5$ (b) $Pe = 20$

Abbildung 6.6.: Approximationen

Modifikationen

Da die oben dargestellte Basismethode nicht geeignet ist, um eine vernünftige Approximation für relativ große Peclet–Zahlen (z.B. $Pe > 50$) zu erhalten, erwähnen Howard und Roberts einige Möglichkeiten, um dieses Problem zu beheben. Eine Möglichkeit basiert auf der folgenden Idee. Zunächst wendet man die GP–Methode auf die KDG mit einer kleineren Peclet–Zahl als eigentlich vorgesehen an. Die letzte Population dieses Durchlaufes wird dann als Startpopulation für die KDG mit der eigentlich hohen Peclet–Zahl verwendet. Howard und Roberts überprüfen außerdem, ob die Wahl anderer numerischer Konstanten bessere Approximationen liefert. In der Basismethode stammen die Konstanten aus dem Intervall $[0, 1]$. Durchführungen der GP–Methoden mit den Intervallen $[0, 0.001]$ und $[-1, 1]$ führen zu einer Veränderung der Größe der einzelnen Koeffizienten, ohne jedoch eine Verbesserung der Güte oder des Aufwandes zu erreichen. Die Performance der Methode zur Lösung der KDG könnte eventuell dadurch verbessert werden, dass man anstatt der obigen Polynome zum Beispiel Chebyshev oder Legendre Polynome verwendet, die ebenfalls leicht zu differenzieren und intergrieren sind.

Die obige GP–Methode ist verwendet worden, um eine eindimensionale KDG zu lösen. Wünschenswerter wäre jedoch eine Methode, die partielle Differentialgleichungen lösen kann. Howard und Roberts machen einen Vorschlag, wie man mit ihrer Methode folgende KDG in zwei Variablen lösen könnte:

$$\frac{\partial^2 \hat{T}}{\partial x^2} + \frac{\partial^2 \hat{T}}{\partial y^2} - Pe\left(\frac{\partial T}{\partial x} + \frac{\partial T}{\partial y}\right) = 0$$

$$T(x = 0) = 1$$
$$T(x = 1) = 0$$
$$T(y = 0) = 0$$
$$T(y = 1) = 0$$

Als Approximation kann man in diesem Fall dann

$$\hat{T} = xy(1 - x)(1 - y)p + exp(-\Gamma x^2)$$

wählen, wobei p nun ein Polynom in $x^i y^j$ ist und Γ relativ groß gewählt werden muss (z.B. $\Gamma > 10^4$), damit die Randbedingungen $\hat{T}(x = 1) = \hat{T}(y = 0) = \hat{T}(y = 1) = 0$ zumindest näherungsweise erfüllt sind.

Insgesamt ist die GP–Methode von Howard und Roberts gut geeignet, um eine eindimensionale KDG zu lösen. Um jedoch kompliziertere (partielle) Differentialgleichungen

zu lösen, benötigt man eine Weiterentwicklung dieser vorgestellten Methode oder einen anderen Ansatz.

6.5.2. Methode nach Koza

In diesem Abschnitt wird die Methode aus Koza [22] vorgestellt, mit der man mit Hilfe von GP die KDG lösen kann. Das grundlegende Prinzip ist identisch zur Methode von Howard und Roberts. Zunächst erstellt man eine Startpopulation von Bäumen, die jeweils eine approximative Lösung der KDG darstellen. Mit Hilfe von GP–Operatoren werden dann so lange neue Generationen erstellt, bis einer der Bäume eine ausreichende Fitness hat. Der Unterschied zwischen beiden Methoden liegt in der Darstellung der Bäume und in der Berechnung der Fitness.

Zunächst wird erklärt, wie die Bäume bei Koza aussehen. Koza legt die approximative Lösung nicht a priori (z.B. als Polynom) fest, sondern lässt die GP–Methode aus den vorgegebenen mathematischen Funktionen selber Approximationen erstellen. Ein Baum repräsentiert nun nicht mehr einen Koeffizientenvektor für ein Polynom sondern direkt eine mathematische Funktion. Die folgende Tabelle 6.11 zeigt, welche Funktionen in der Methode von Koza verwendet werden könnten.

Parameter	Wahl
Blätter (terminals)	[0,1], x
Funktionen	+, -, *, sin, cos, exp

Tabelle 6.11.: Parameter, Methode nach Koza

Damit ein Baum insgesamt eine Funktion darstellen kann, wird die Menge, aus der die Blätter bestimmt werden, um die Variable x erweitert. Das folgende Beispiel liefert einen Baum, der in der Methode nach Koza gemäß der obigen Tabelle vorkommen kann.

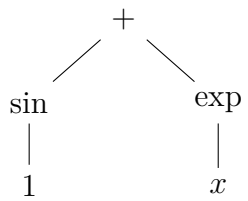


Abbildung 6.7.: Beispiel eines Baumes

Dieser Baum liefert als approximierende Funktion $\hat{T}$ mit $\hat{T}(x) = \sin(1) + \exp(x)$.
Der zweite Unterschied besteht in der Berechnung der Fitness. In dieser Methode wird

nun nicht die erste und zweite Ableitung der Funktion analytisch bestimmt, um dann die Fitnessfunktion (6.6) anzuwenden. Koza bestimmt stattdessen in dem Intervall [0,1] aus der obigen Tabelle 6.11 200 Stellen x_i mit $i \in 0, \ldots, 199$ und $x_i < x_{i+1}$. Durch numerisches Differenzieren erhält man jeweils die 200 Punkte $\left(x_i, \left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_i}\right)$ und $\left(x_i, \left.\frac{\partial^2 \hat{T}}{\partial x^2}\right|_{x=x_i}\right)$. Damit werden dann die 200 Punkte $\left(x_i, \left.\frac{\partial^2 \hat{T}}{\partial x^2}\right|_{x=x_i} - Pe \left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_i}\right)$ berechnet. Wenn $\hat{T}$ die exakte Lösung der KDG ist, sind die zweiten Koordinaten der Punkte Null. Falls $\hat{T}$ es nicht ist, sind sie ungleich Null. Koza wählt deshalb folgende Fitnessfunktion:

$$F = 0.75 \sum_{j=0}^{199} \left| \left.\frac{\partial^2 \hat{T}}{\partial x^2}\right|_{x=x_i} - Pe \left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_i} \right| + 0.125 * 200 |T(0) - \hat{T}(0)| + 0.125 * 200 |T(1) - \hat{T}(1)| \tag{6.7}$$

Die letzten beiden Summanden bestrafen die Fehler bzgl. der Randwertbedingungen, die bei Koza nicht a priori gesichert sind. Der erste Summand bestraft die Fehler beim Lösen der KDG in den einzelnen Punkten. Es werden folglich die betragsmäßigen Abweichungen von Null in den 200 Punkten aufaddiert. In diesem Fall bedeutet ein kleinerer Fitnesswert eine bessere Approximation, wobei das Minimum bei F=0 liegt. Das Prinzip der Größe der Startpopulation, der maximalen Größe eines Baumes, der GP–Operatoren und der Tournaments ändert sich nicht im Vergleich zur Methode nach Howard und Roberts.

Es bleibt nur noch die Frage, wie das numerische Differenzieren genau durchgeführt wird. Zunächst werden die entsprechenden Funktionswerte $\left(x_i, \hat{T}(x_i)\right)$ bestimmt. Der Wert $\left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_i}$ ist dann für alle x_i außer den Randwerten x_0 und x_{199} der Durchschnitt der Sekantensteigungen von $\left(x_{i-1}, \hat{T}(x_{i-1})\right)$ nach $\left(x_i, \hat{T}(x_i)\right)$ und $\left(x_i, \hat{T}(x_i)\right)$ nach $\left(x_{i+1}, \hat{T}(x_{i+1})\right)$. Den Wert $\left.\frac{\partial^2 \hat{T}}{\partial x^2}\right|_{x=x_i}$ erhält man analog, indem man die Steigungen der Sekanten durch die Punkte $\left(x_{i-1}, \left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_{i-1}}\right)$ und $\left(x_i, \left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_i}\right)$ bzw. $\left(x_i, \left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_i}\right)$ und $\left(x_{i+1}, \left.\frac{\partial \hat{T}}{\partial x}\right|_{x=x_{i+1}}\right)$ berechnet. Für die Randwerte x_0 und x_{199} nimmt man einfach die Steigungen zu x_1 bzw. x_{198}. Damit ist die grobe Struktur der Methode von Koza

erklärt.

Durchführungen dieser Methode zur Lösung von Differentialgleichungen konvergieren jedoch oftmals nur langsam, da die Fitnessberechnung einen relativ hohen Aufwand erfordert. Die Methode von Koza muss also ebenfalls weiterentwickelt werden, um komplexere (partielle) Differentialgleichungen lösen zu können.

6.5.3. Methode nach Howard und Kolibal

Nick Macin

In diesem Abschnitt wird eine dritte Methode zur Lösung von DGL und PDGL vorgestellt. Das GP nach Howard und Roberts (Abschnitt 6.5.1) wird mit der stochastischen Bernstein Interpolation (SBI), die von Kolibal entwickelt und patentiert wurde, kombiniert. SBI ist eine neue noch nicht komplett erforschte Methode, die aber einen großen Anwendungsbereich hat. Man kann sie für Daten-Approximation oder -Interpolation, bei Wiederherstellung der Daten, bei der Filtration der Datenfehler oder bei Daten/-Bild Entfaltungen benutzen. Das ist aufgrund einiger Eigenschaften von SBI möglich. Diese Technik ist robust, man bekommt immer eine numerische Konvergenz, hierarchisch aufgebaut, d.h. Daten können zu beliebigem Zeitpunkt hinzugefügt werden, um besseres Resultat zu bekommen, und sie ist nicht harmonisch, sondern spektral, d.h. es ist möglich, eine Folge von Komponenten zu konstruieren, deren Summe zu dem gleichen Resultat konvergiert. Diese neue Methode nach Howard und Kolibal [15] wird auf eindimensionale KDG (Gleichung (6.3)) angewandt — dadurch steigt das Konvergenzverhalten von $O(h^4)$ auf $O(h^6)$ im Inneren des Intervalls und zum Rande hin sinkt sie auf $O(h^1)$. Die Methode nach Howard und Kolibal lässt sich auch auf mehrdimensionale Probleme übertragen.

Vorbereitungen für den Algorithmus

Für den Algorithmus werden die Bernstein–Funktionen benötigt. Diese Funktionen sind eine Erweiterung der Bernsteinpolynome, wobei die Binomialverteilung zu der Standardnormalverteilung zusammen mit der entsprechenden Änderung der Variablen wird. Dadurch teilen diese Funktionen die Approximationseigenschaften der Bernsteinpolynome.

Sei die Funktion $f(x)$ gegeben für $x_k \in [0,1]$, so dass gilt $f(x_k) = y_k$ für $k = 1, \ldots, n$.

Bernstein Funktionen $K_n(x)$ sind gegeben durch:

$$K_n(x) = \sum_{k=0}^{n} \frac{y_k}{2} \left[erf\left(\frac{z_{k+1} - x}{\sqrt{\sigma(x)}} \right) + erf\left(\frac{x - z_k}{\sqrt{\sigma(x)}} \right) \right],$$

wobei f stückweise konstant auf (z_{k-1}, z_k) mit dem Wert y_k und $z_0 = -\infty$, $z_k = (x_{k+1} + x_k)/2$ für $k = 1, \ldots, n-1$, und $z_n = \infty$ ist. $\sigma(x)$ ist der freie Parameter oder Eigenglättungsparameter, der das Verhalten der Funktion zwischen den Stützstellen bestimmt. Im Bezug auf KDG bestimmt σ die Geschwindigkeit, mit der die Lösung entwickelt wird. Große Diffusionswerte verursachen große Änderung, und kleine Diffusionswerte — kleine Änderung. $erf(x)$ ist die Gaußsche Fehlerfunktion:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\tau^2} d\tau$$

Weiterhin wird eine quadratische $(n \times n)$–Matrix $A_{nn} = (a_{jk})$, wobei für die Koeffizienten gilt:

$$a_{jk} = \frac{1}{2} \left[erf\left(\frac{z_{k+1} - x_j}{\sqrt{\sigma(x)}} \right) + erf\left(\frac{x_j - z_k}{\sqrt{\sigma(x)}} \right) \right] \tag{6.8}$$

Aufgrund der Fehlerfunktion in der Definition sind die Interpolanten über das ganze Intervall unendlich oft differenzierbar und glatter, als z.B. die trigonometrischen Funktionen.

Algorithmus 6.4 : Algorithmus zur Lösung der KDG nach Howard und Kolibal

Data : GP entwickelt die Inputdaten $\{(x_i, y_i)\}, i = 0, \ldots, n-1$
1 Konvertiere Daten auf das Intervall $[0, 1]$
2 Konstruiere Matrix A_{nn} (hängt von x_i und freiem Parameter ab) durch die Bernsteinfunktion, gemäß (Gleichung (6.8))
3 Berechne A_{nn}^{-1}
4 Konstruiere augmentierende Matrix $\tilde{A_{mn}}$ durch Bernsteinfunktion für $m > n$, (Gleichung (6.8))
5 Berechne $\tilde{A_{mn}} A_{nn}^{-1} y$, um Outputdaten $\{y_i\}, i = 0, \ldots, m-1$ zu erhalten
6 Konvertiere Outputdaten $\{y_i\}$ zurück (Lösung der KDG)
7 Berechne Ableitungen
8 Berechne Fitnessfunktion

Die Methode arbeitet mit verschiedenen Längen der Input- und Outputdaten und ist zeitaufwendiger, als die Methode nach Howard und Roberts.

Lösung der KDG

Im ersten Schritt wird GP nach Howard und Roberts angewendet mit identischen Parametern aus Abschnitt 6.5.1 und liefert einen Datensatz $\{(x_i, y_i)\}$. GP verteilt gleichmäßig die berechneten Punkte x_i mit Werten y_i auf dem Intervall $(0, 1)$ und fügt die beiden Randbedienungen $(0, 1)$ und $(1, 0)$ hinzu. Weiterhin wird beim Experimentieren festgestellt, dass GP nach Howard und Roberts die Freiheit für Position der Punkte x_i schätzt und SBI Verfahren von den Pufferzonen außerhalb des Lösungsintervalls profitiert. Deswegen wird folgende Modifikation implementiert. Intervall der Lösung wurde von $(0, 1)$ auf $(0.2, 0.8)$ umgebaut und der GP-Methode wurde erlaubt, die Position der Punkte überall (unter bestimmten Einschränkungen) im Intervall $(0, 1)$ zu wählen. Der Algorithmus für diese Informationen wird folgend implementiert:

Algorithmus 6.5 : Modifikation des Algorithmus 6.4

1 Auswertung des Individuums
2 **if** *Länge des resultierenden Vektors ungerade* **then**
3 verwerfe seinen letzten Term
4 $x_0 = 0$ ist fix und y_0 wird auf das erste Element des resultierenden Vektors gesetzt
5 $x_n = 1$ ist fix und y_n wird auf das zweite Element des resultierenden Vektors gesetzt
6 Das Tupel des Vektors $v_i v_{i+1}$ nimmt zugehörige x_i und y_i im Intervall $(0, 1)$
7 Das Intervall $(0, 1)$ wird in 97 Abschnitte unterteilt, damit die Differenz zwischen Stützstellen $x_i - x_{i+1}$ nicht mehr als 0.01 beträgt
8 Die Position dieser Abschnitte berücksichtigt die neuen Randbedingungen $(0.2, 1.0)$ und $(0.8, 0.0)$

KDG wird mit 0.6 skaliert und somit lautet die Gleichung (6.3) wie folgt:

$$0.6\frac{\partial^2 T}{\partial x^2} - Pe\frac{\partial T}{\partial x} = 0 \; x \in [0, 1]$$
$$T(0.2) = 1$$
$$T(0.8) = 0$$

Die Fitness wird über die Outputdaten $\{y_i\}, i = 0, \ldots, m-1$ berechnet. Eine Menge von 241 Punkten wird durch proportionale Verteilung der Punkte im Intervall $(0.2, 0.8)$ variiert. Es wird mit zwei Fitnessfunktionen experimentiert:

1. Identisch mit der Fitnessfunktion aus GP–Methode nach Howard und Roberts

(siehe 6.6)

$$F = -\int_{0.2}^{0.8} \left(0.6\frac{\partial^2 T}{\partial x^2} - Pe\frac{\partial T}{\partial x}\right)^2 dx$$

2. Unterteile das Intervall $(0.2, 0.8)$ in 60 Abschnitte. Jeder Abschnitt hat 4 gleichmäßige Lücken und beinhaltet 5 Ausgangspunkte. Das Quadrat von $\left(0.6\frac{\partial^2 T}{\partial x^2} - Pe\frac{\partial T}{\partial x}\right)$ kann mit Hilfe der Fünf–Punkte Boole's rule integriert werden

Die Ableitungen $\left\{\frac{\partial y_i}{\partial x}\right\}$ und $\left\{\frac{\partial^2 y_i}{\partial x^2}\right\}$ für $i = 0, \ldots, m-1$ kann man durch die Ableitung der augmetierenden Matrix $\tilde{A_{mn}}$ bestimmen, z.B. durch Gradientenversion einer Matrix. Eine alternative Methode ist gegeben, durch berechnen der Ableitungen mit Hilfe Vorwärts–, Rückwärts oder zentraler Differenzenquotienten.

Ergebnisse

Beim Experimentieren stellen sich zwei typische Ergebnisse heraus. Die Abbildung

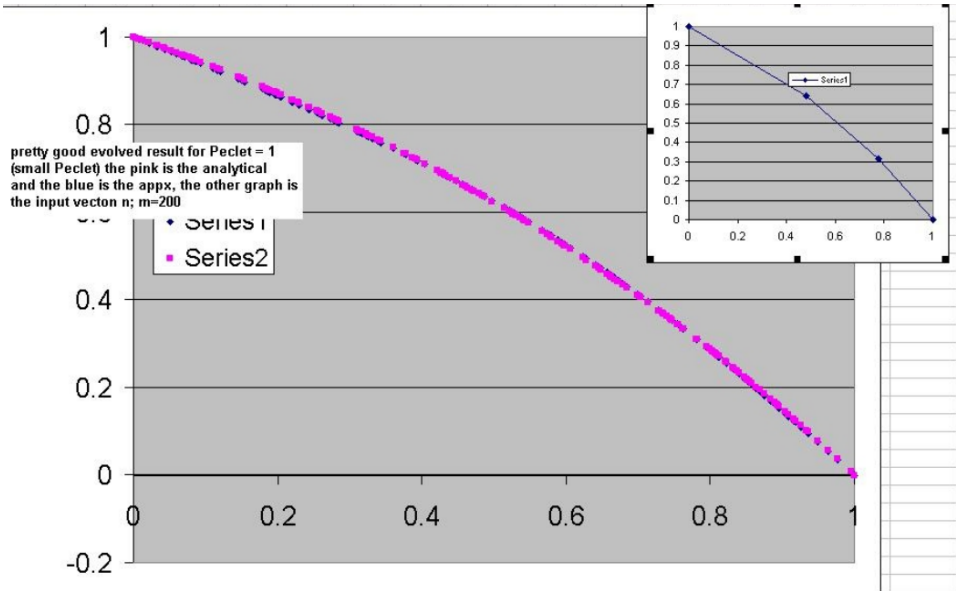


Abbildung 6.8.: GP–SBI–Methode, Pe=1
Quelle: D. Howard, J. Kolibal [15]

6.8 zeigt Ergebnis der Outputdaten y_i durch numerische Approximation der GP–SBI–Methode für eine sehr niedrige Peclet–Zahl ($Pe = 1$), wobei das Verfahren noch nicht modifiziert wurde. Die magentafarbige Linie ist die analytische Lösung und die blaue Linie ist SBI–Approximation. Der eingefügte Graph zeigt die Position und den Wert

der Interpolationspunkte. Es wird zu den Randbedingungen zwei interne Punkte durch GP–Methode entwickelt. Die Outputdaten werden bei 200 zufälligen Orten erzielt. σ wird konstant auf 1 gehalten.

In den Abbildungen 6.9 und 6.10 ist $Pe = 4$, $\sigma = 1$ und das Intervall ist modifiziert. Die

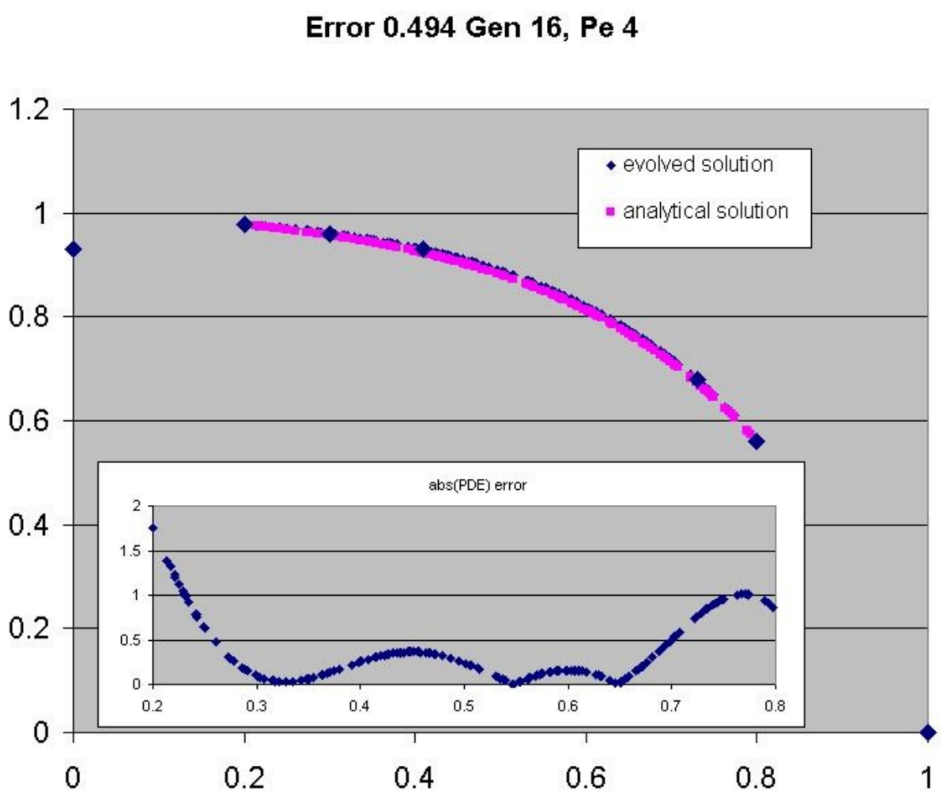


Abbildung 6.9.: GP–SBI–Methode, Pe=4
Quelle: D. Howard, J. Kolibal [15]

Abbildung 6.9 veranschaulicht auch die von GP–Methode entwickelte Stützstellen bei $x = 0$ und $x = 1$ und drei weitere entwickelte Punkte innerhalb des Intervalls $(0.2, 0.8)$, die modifizierten Randbedingungen sind konstante Werte, die von Evolution nicht betroffen sind. Der eingefügte Graph zeigt die Abweichung der Werte vom absolutem Wert der KDG an jedem Outputpunkt. Dabei werden andere Randbedingungen, als 0 bzw. 0.2 und 1 bzw. 0.8 gewählt. Das zeigt, dass beliebige Randbedingungen sich mit diesem Verfahren durchführen lassen werden. Die Abbildung 6.10 zeigt den Graph der Ableitung durch SBI–Approximation (blau) im Vergleich zu der analytischen Lösung der Ableitungen (magenta). Diagramm auf der linken Seite ist die erste und auf der rechten Seite — die zweite Ableitung. Die durch GP–Methode formulierte Approximation der ersten Ableitung ist näher zu der analytischen Lösung, als die zweite Ableitung. Der

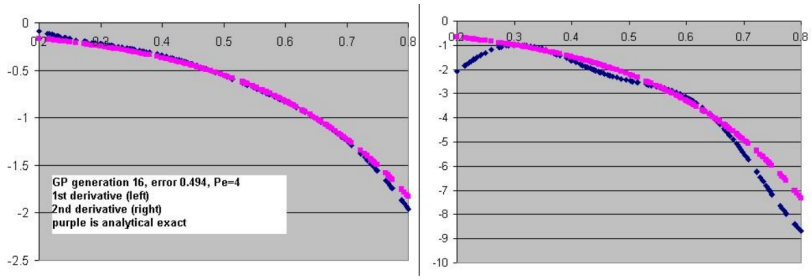


Abbildung 6.10.: GP–SBI Ableitungen
Quelle: D. Howard, J. Kolibal [15]

Fehler in den Endbereichen ist mehr ausgeprägt.

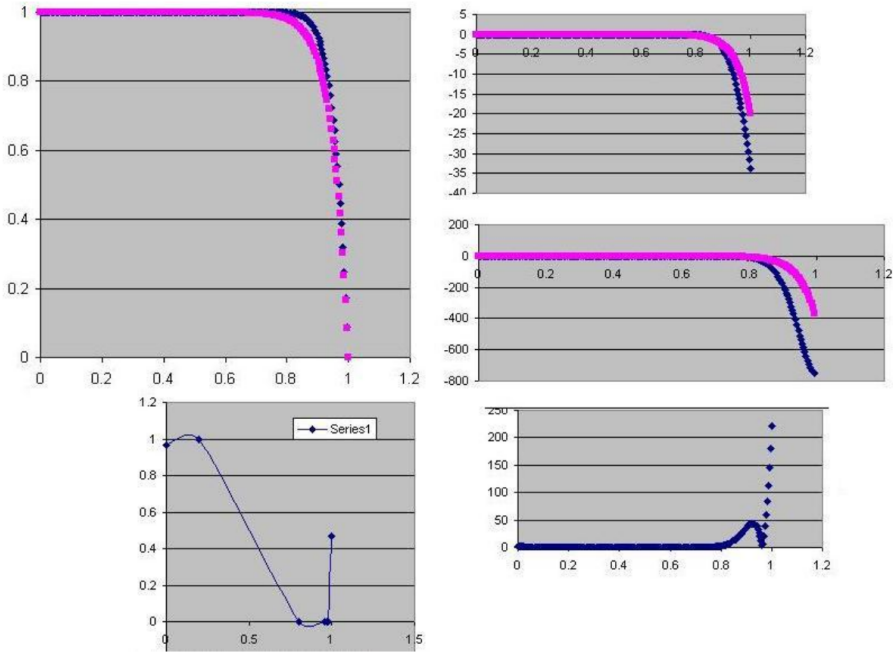


Abbildung 6.11.: GP–SBI–Methode, Pe=20
Quelle: D. Howard, J. Kolibal [15]

In den Abbildungen zuvor wird σ konstant gehalten, da sonst die Punkte nicht interpoliert werden und die Randbedingungen nicht erfüllt werden können. Aber ab Pe>10 ist es notwendig, den Wert von σ zu verkleinern. Ein Wert von $\sigma = 1$ führt zu keinen guten Konvergenzeigenschaften bei höheren Peclet–Zahlen. Die Verringerung des Wertes von σ bringt signifikante Ergebnisse hervor, die nahe an analytischen Lösungen bei hohem Wert von Pe sind. Die Abbildung 6.11 zeigt die Ergebnisse eines erfolgreichen Testlaufes bei einer hohen Peclet–Zahl ($Pe = 20$). Der blaue Graph ist die GP–SBI–

Methode und magentafarbiger Graph ist die analytische Lösung. Im Uhrzeigersinn von oben links: Lösung; erste Ableitung; zweite Ableitung; absoluter Fehler; entwickelte und vorgeschriebene Stützstellen. Die Ergebnisse werden durch eine geeignete Koordinatentransformation vom Intervall $(0.2, 0.8)$ auf $(0, 1)$ zurück skaliert. Diese Abbildung zeigt zwei wichtige Beobachtungen. Im Graph mit dem absoluten Fehler sieht man, wie das GP–Methode den Pufferraum $(0.8, 1)$ verwendet um die Funktion nach oben zu drehen. GP–Methode entwickelt mehr Punkte als erforderlich und einfach häufte sie in die Region nahe bei 1.0 an, um so effektiv diese in einen einzigen Punkt zu drehen. In Testläufen bei $Pe = 100$ und GP–Population von 10000 ergibt sich ein sehr niedriger Wert für σ ($\sigma = 0.01$). Obwohl der Fehler recht hoch ist, zeigen die resultierenden Werte richtige Form sowohl für die Lösung, als auch für die Ableitungen.

Die durch SBI berechnete Ableitungen sind in allen Testläufen glatter, als die mit finiten Differenzen gewonnenen Ableitungen. Ableitungen, die durch die finite Differenzen berechnet werden, weisen eine gewisse Schwingung auf, die als Quelle der Differenzen zur Fitnessfunktion übersetzt werden können. Der Grund liegt in der Tatsache, dass durch SBI berechnete Ableitungen in fast allen Experimenten verwendet werden, und somit glatter sind, als die lokalisierten Ableitungen der finiten Differenzen. Die GP–Methode ist auch mit aus den finiten Differenzen gewonnenen Ableitungen erfolgreich.

7. Symbolische Regression in der Zeitreihenanalyse

Dmitri Bogonos

Bei der Regressionsanalyse besteht die Aufgabe darin, eine abhängige Variable durch eine oder mehrere unabhängige Variablen zu beschreiben. Bei der linearen Regression ist die Methode der kleinsten Quadrate das Standardverfahren. In den Wirtschaftswissenschaften können viele Prozesse nur schlecht durch lineare Zusammenhänge erklärt werden. Oft liegen vielen Tatsachen komplexere Beziehungen zugrunde. Die linearen Modelle sind in diesen Fällen hilfslos. Ein Ansatz, der nichtlineare Zusammenhänge konstruiert, ist die *symbolische Regression*. Sie ermöglicht es, die abhängige Variable durch beliebige Funktionen zu erklären, z.B. $y = \frac{1}{8}x^2 - \sin^3(x)$ o.a. Ein wesentlicher Vorteil der symbolischen Regression ist, dass man keine Theorien für die Erklärung der Zusammenhänge zwischen den Variablen unterstellen muss. Sie findet die Struktur der Zusammenhänge rein numerisch.

Nach I. Zelinka, L. Nolle, Z. Oplatkova [17] existieren heutzutage drei Methoden, die für die symbolische Regression verwendet werden können: GP, grammatikalische Evolution ud analytische Programmierung. Die beiden letzten Methoden basieren auf GP, wobei GP für die symbolische Regression am häufigsten verwendet wird. John R. Koza [23] hat als erster vorgeschlagen, die Aufgaben der symbolischen Regression mithilfe genetischer Programmierung zu lösen. Im Prinzip, ist die genetische Programmierung genau die symbolische Regression.

Sehr oft wird die GP für die symbolische Regression für Anwendungen in der Finanzwirtschaft benutzt. M. Santini und A. Tettamazi [29], 2000, benutzen in Rahmen des Wettbewerbs „Dow Jones Prediction“ diese Kombination für die Vorhersage des Index *Dow Jones Industrial Average* und belegen Platz 1. Die Aufgabe bestand in der Prognose des Index am Tagesschluss für die Periode vom 19. Juni 2000 bis einschließlich 30. Juni 2000 (10 Tage). Die Bewertung der Ergebnisse erfolgte mit der Formel

$$\sum_{t=1}^{10} \frac{11-t}{10} |x_t - \hat{x}_t| \tag{7.1}$$

wobei x_t den Schlussstand am Tag t bezeichnet und $\hat{x}_t$ die Prognose.

7.1. Der Algorithmus

Die Idee ist, eine Funktion mit der symbolischen Regression aus 116 beobachteten Realisierungen des Index (vom 3. Januar 2000 bis 17. Juni 2000) zu konstruieren und mit ihr die 10 Prognosewerte zu bestimmen. Die Autoren konstruieren eine Datenmenge $\mathbf{D} = \{x^{(i)} | 0 \leq i \leq a\}$, wobei $a > 0$, $x^{(0)} = (x_1, \ldots, x_T)$ als die beobachtete Reihe und $x^{(i)} = \left(x_1^{(i)}, \ldots, x_T^{(i)}\right)$ für $1 \leq i \leq a$ als eine Hilfsreihe. Das Ziel ist ein Vektor $\mathbf{e} = (e_1, \ldots, e_h)$, wobei h die Horizontlänge beschreibt (hier $h = 10$), und $e_j : \mathbf{D} \times \{1, \ldots, T\} \to \mathbb{R}$, so dass $e_j(\mathbf{D}, T)$ Prognose für x_{T+j} mit $1 \leq j \leq h$ ist.

Es werden

- *Terminalsymbole* $= [-c, c] \subseteq \mathbb{R}$,
- *Function Set* $= \{+, -, \times, /, pow, sqrt, log, exp\} \cup \bigcup_{0 \leq i \leq a} \{data[i], diff[i], ave[i]\}$

definiert, wobei für $\alpha \in \mathbb{R}$ und $0 \leq t \leq T$, $\forall i$ gilt

$$
\begin{aligned}
data[i](\alpha, t) &= x^{(i)}_{t - \lfloor |\alpha| \rfloor}, \\
diff[i](\alpha, t) &= x^{(i)}_t - x^{(i)}_{t - \lfloor |\alpha| \rfloor}, \\
ave[i](\alpha, t) &= \frac{1}{1 + \lfloor |\alpha| \rfloor} \sum_{k = t - \lfloor |\alpha| \rfloor}^{t} x^{(i)}_k .
\end{aligned}
$$

Damit wird sicher gestellt, dass die Prognosen auf den Daten aus der Vergangenheit basieren. Jede Population besteht aus N Individuen, wobei jedes Individuum ein Vektor $\mathbf{e} = (e_1, \ldots, e_h)$ ist. Jedes $e_j, j = 1 \ldots h$, ist ein String in der umgekehrten polnischen Notation. Jedem Terminalsymbol und jedem Element des Function Sets wird eine Auswahlwahrscheinlichkeit zugewiesen, und jedes e_j wird aus diesen Elementen zusammengebaut.

Es wird eine Distanzfunktion $\delta \colon \mathbb{R}^2 \to \mathbb{R}^+$ mit $\delta(x, y) = |x - y|/|x|$ definiert. Weiter wird für δ der mittlere Fehler für $\mathbf{e}$ zur Zeit $0 \leq t \leq T$

$$
\varepsilon(\mathbf{e}, t) = \frac{1}{h} \sum_{j=1}^{h} \delta \left(x_{t+j}, e_j(\mathbf{D}, t) \right)
$$

und der mittlere Fehler für $\mathbf{e}$ über die ganze Zeit als

$$f_s(\mathbf{e}) = \frac{1}{T-h} \sum_{t=1}^{T-h} \varepsilon(\mathbf{e}, t) = \frac{1}{h(T-h)} \sum_{t=1}^{T-h} \sum_{j=1}^{h} \delta\left(x_{t+j}, e_j(\mathbf{D}, t)\right)$$

definiert. Ist $\delta > 0$, folgt $f_s(\mathbf{e}) > 0\ \forall \mathbf{e}$. $f_s(\mathbf{e})$ wird als die Fitness verwendet, $f_a(\mathbf{e}) = (1 + f_s(\mathbf{e}))^{-1}$ — als angepasste Fitness und $f(\mathbf{e}) = f_s(\mathbf{e}) / \sum_{\hat{\mathbf{e}}} f_s(\hat{\mathbf{e}})$ — als normierte Fitness.

Für die Selektion wird die Methode Truncation Selection verwendet, wobei $\lfloor N\rho_s \rfloor$–viele ($\rho_s = 0.1$) Individuen für die Erzeugung nächster Generation ausgewählt werden. Für Crossover werden zwei Individuen $\mathbf{e} = (e_1, \ldots, e_h)$ und $\mathbf{e}' = (e'_1, \ldots, e'_h)$ ausgesucht, und in jedem Paar e_j, e'_j für $1 \leq j \leq h$ werden e_j, e'_j mit Wahrscheinlichkeit $\rho_c = 0.6$ ausgetauscht. Die Mutation eines Individuums $\mathbf{e} = (e_1, \ldots, e_h)$ wird als Austausch zweier Werte $e_j, e_k, 1 \leq j, k \leq h, j \neq k$ durchgeführt, d.h. eine Art Crossover innerhalb eines Individuums. Die Werte werden mit Wahrscheinlichkeit $1/|\mathbf{e}|$ ausgewählt und mit Wahrscheinlichkeit $\rho_m = 0.2$ ausgetauscht. Da man nicht annehmen kann, dass die Zeitreihe stationär[1] ist, wird eine kurze Stichprobe der Werte vom 3. Januar 2000 bis 17. Juni 2000 ausgewählt. In der Tabelle 7.1 werden alle Parameter zusammengefasst:

Parameter	Bedeutung	Wert
T	Größe der Stichprobe	116
h	Prognosehorizont	10
N	Individuenanzahl	500
G	Generationenanzahl	20
m	Tiefe des Ausdrucks	5
δ	Abstandsfunktion	$\lvert x-y\rvert/\lvert x\rvert$
ρ_s	Selektionsverhältnis	0.1
ρ_c	Crossoverwahscheinlichkeit	0.6
ρ_m	Mutationswahrscheinlichkeit	0.1

Tabelle 7.1.: Parameterwerte für Dow Jones IA Prognose

7.2. Ergebnisse und Folgerungen

In der Tabelle 7.2 sind die Ergebnisse dargestellt. Die Lösung ist charakterisiert durch relativ niedrige Abweichung der Prognose vom tatsächlichen Wert. Die letzte Spalte wird mit der vorletzten nach der Gleichung 7.1 berechnet.

[1] grob gesprochen bedeutet Stationarität die Abwesenheit eines Trends in der Zeitreihe

Prognose	Tats. Wert	Fehler	diskontierter Fehler	Punkte
10449.300	10557.80	-108.5	108.5	108.5
10449.300	10435.20	14.1	12.69	121.19
10449.300	10497.70	-48.4	38.72	159.91
10417.374	10376.10	41.274	28.8918	188.8018
10476.400	10404.80	71.6	42.96	231.7618
10535.523	10542.99	-7.467	3.7335	235.4953
10435.246	10504.46	-69.214	27.6856	263.1809
10461.648	10527.79	-66.142	19.8426	283.0235
10418.607	10398.04	20.567	4.1134	287.1369
10421.874	10447.89	-26.016	2.6016	289.7385

Tabelle 7.2.: Ergebnisse des Verfahrens von M. Santini und A. Tettamazi

Leider werden die expliziten Lösungen der Autoren und ihrer Konkurrenten nicht veröffentlicht und man kann deswegen keinen direkten Vergleich unterschiedlicher Modelle machen. Allerdings hat diese Methode gegenüber anderen Verfahren den 1. Platz im Wettbewerb belegt. M. Santini und A. Tettamazi behaupten, dass weitere Verfeinerungen des Verfahrens mithilfe der Modellierung der Markov-Prozesse, ARCH und GARCH-Modellen, die Ergebnisse nicht verbessern konnten.

Miroslav Kĺúčik, Jana Juriová und Marian Kĺúčik [24] haben das Bündel symbolische Regression - genetische Programmierung mit einem verbreiteten makroökonomischen ARIMA-Modell[2] verglichen. Sie behaupten, dass die Weltwirtschaft strukturellen Änderungen bevorsteht und die konventionellen Modelle, wie z.B. das ARIMA-Modell, eventuell, somit nicht mehr in Lage sein werden, die Zusammenhänge zwischen volkswirtschaftlichen Größen zu bestimmen oder zu erklären. Sie sind in ihren Analysen zu dem Schluss gekommen, dass die symbolische Regression mit der genetischen Programmierung mindestens nicht schlechtere Ergebnisse, als ARIMA liefert, und sogar manchmal bessere. Dabei ist nicht zu vergessen, dass keine strukturellen Zusammenhänge zwischen den Variablen angenommen werden müssen. Das heißt, im Hinblick auf die möglichen Änderungen der Wirtschaftssysteme, kann die Hälfte der Arbeit erspart werden: man kann die beobachteten Daten sofort mit symbolischer Regression erklären, ohne eine Theorie unterstellen zu müssen Alles in allem lässt sich sagen, dass die symbolische Regression mithilfe der genetischen Programmierung ein sehr gutes Verfahren in den Wirtschaftswissenschaften sein kann, dass es vielversprechend ist und dass es viel an Bedeutung gewinnen kann.

[2]ARIMA = Autoregressive Integrated Moving Average. Ein lineares Modell, das eine Variable als Funktion in den Werten dieser Variable in der Vergangenheit und einem white-noise Prozess beschreibt.

8. Zusammenfassung und Schlussfolgerungen

Diese Arbeit behandelt Theorie und Anwendungsmöglichkeiten genetischer Algorithmen, welche ein universelles und robustes Optimierungsverfahren darstellen, wodurch eine Anwendung auf eine Vielzahl an Problemstellungen möglich ist. Des Weiteren werden die Möglichkeiten der genetischen Programmierung, beispielsweise zur Lösung von Systemen partieller Differentialgleichungen, aufgezeigt. Genetische Algorithmen abstrahieren die Terminologie der biologischen Evolution nach Charles Robert Darwin basierend auf der Erkenntnis, dass die Anpassung von Lebewesen an die vorherrschenden, oder auch sich ändernden Umweltbedingungen als Optimierungsprozess aufgefasst werden kann. Die Prozesse der natürlichen Evolution werden in Form von Operatoren auf Werte des Definitionsbereichs mit dem Ziel einer optimalen Anpassung an das zugrundeliegende Problem angewendet. Im Gegensatz zu vielen anderen Optimierungsverfahren behelfen sich genetische Algorithmen der populationsbasierten Suche, bei der eine festgelegte Anzahl an Startpunkten generiert wird, welche sich während des Durchlaufs dem Optimum annähern sollen. Die Population ist der Garant für die Robustheit des Verfahrens; außerdem resultiert daraus der Vorteil der Überwindung lokaler Extrema während eines Suchdurchlaufs. Zudem weisen genetische Algorithmen einen geringen Implementierungsaufwand auf.

Kapitel 2 gab eine Einführung in genetische Algorithmen, deren Hintergrund und den grundsätzlichen Aufbau. Außerdem wurden die zur Verfügung stehenden Instrumente näher erläutert. Kapitel 3 stellte die zugrundeliegende Theorie durch Herleitung des Theorems von John H. Holland vor. Damit wurde im Ansatz ersichtlich, wie ein genetischer Algorithmus den Suchraum durchläuft. Es wurde klar, dass das Theorem zum völligen Verständnis der Funktionsweise leider nicht ausreichend ist, da es in der Realität an der Komplexität, der dynamischen Natur und modularen Struktur genetischer Algorithmen scheitert. Des Weiteren gibt es für die Wahl der einzelnen verfügbaren Instrumente keine geschlossene Theorie, weshalb man entweder auf Erfahrungswerte aus der Literatur oder eigene Tests angewiesen ist.

Dieser Problemstellung widmete sich das 4. Kapitel, welches mithilfe numerischer Tests einige Erkenntnisse und Gesetzmäßigkeiten für die Anwendung erzielen sollte. Die Tests zu ein- und mehrdimensionalen Funktionen zeigten deutlich, dass die Wahl der einzelnen Methoden und Parameter von dem zugrundeliegenden Problem abhängig ist — was nicht weiter verwunderlich ist. Sie zeigten auch, dass die Wahl der Selektionsmethode und der Populationsgröße eine übergeordnete Rolle spielt. Unter einer großen Population und der Tournament Selection konnten in den Tests recht schnell akzeptable Ergebnisse erzielt werden — die Wahl der weiteren Parameter war dann eher nebensächlich. Entgegen der Theorie aus Kapitel 3 trug eine höhere Mutationswahrscheinlichkeit in den meisten Fällen zu einer Verbesserung der Lösungsquote bei, was wiederum durchaus verblüffend war. Unter der Kombination des genetischen Algorithmus mit dem Hill-Climbing Verfahren konnte zudem eine Zeitersparnis erreicht werden, was jedoch eine sorgsame Wahl der Parameter und der Schrittweite voraussetzt. Für weiterführende Aussagen und zur Festigung der erzielten Resultate sind weitere Tests für eine Vielzahl an Funktionen und Problemstellungen erforderlich.

Kapitel 5 zeigt eine Möglichkeit der Verwendung genetischer Algorithmen in der Wirtschaft mittels eines expliziten Beispiels auf. Ein genetischer Algorithmus wird für die Lösung des Problems Economic Dispatch entwickelt. Hier ist das ein nichtkonvexes Optimierungsproblem. Es müssen die Kosten der Stromproduktion an mehreren Kraftwerken minimiert werden. Zusätzliche Restriktionen (ramp rate limits, prohibited zones und Übertragungsverluste) machen konvexe Optimierungsverfahren und andere konventionellen Verfahren, wie z.B. Lambda-Iterations-Methode, unbrauchbar. Es stellt sich hieraus, dass auch unter Vereinfachungen des Problems durch Weglassen dieser Restriktionen der GA bessere Ergebnisse liefert, als andere Methoden. Die Kodierung des Problems macht das Verfahren vor allem für große Stromnetze attraktiv.

Im Anschluss wurde im Rahmen des Kapitel 6 eine Anwendungsmöglichkeit auf die numerische Lösung von Differentialgleichungen aufgezeigt. Hier lag zunächst der Fokus auf der Analyse zweier mit genetischen Algorithmen verwandter Konzepte: *Grammatical Evolution* sowie *Genetische Programmierung*. Das hier vorgestellte Verfahren ist eine effektive Möglichkeit zur Lösung verschiedener dynamischer Systeme, insbesondere zur Lösung von ODES in linearer und nichtlinearer Form sowie von PDEs. Der Grundgedanke des Verfahrens ist es, mit Hilfe einer gegebenen kontextfreien Grammatik Funktionen in geschlossener Form zu generieren, diese mit Hilfe genetischer Operatoren zu modifizieren und so ein gegebenes dynamisches System zu lösen. Auch wenn dieses Verfahren strukturelle Unterschiede zu einem genetischen Algorithmus aufweist, so zeigt sich die nahe Verwandtschaft beispielsweise anhand von Kodierung, der Durchführung von genetis-

chen Operatoren sowie dem Ziel der Fitnessmaximierung. GP arbeitet strukturell wie ein genetischer Algorithmus, nur dass die Kodierung mit Hilfe von parse trees realisiert wird. Der große Vorteil von GP ist dabei die Möglichkeit, sehr abstrakte Problemstellungen zu lösen, wie anhand des Bonitätsproblems 6.3 gezeigt wurde. Auf Basis dieser theoretischen Einführung wurde im Folgenden GP auf die numerischen Lösungspotenziale der *Konvektion–Diffusion–Gleichung* (KDG) angewendet. Hier wurden zunächst drei Methoden vorgestellt, mit denen man allgemeine PDEs lösen kann. Diese Methoden lösen die KDG relativ gut; um jedoch komplexere PDEs zu lösen, bedarf es weiterer Forschung und einer Weiterentwicklung dieser Ansätze.

In Kapitel 7 wird eine Anwendung der genetischen Programmierung und der symbolischen Regression in der Finanzwirtschaft vorgestellt. Es wird ein Verfahren entwickelt, das den zukünftigen Stand (10 Tage) des Aktienindex Dow Jones Industrial Average prognostiziert. Die Methode hat sich gegen viele andere durchgesetzt und erreichte 2000 den 1. Platz im Wettbewerb „Dow Jones Prediction“.

Alles in allem ist das Feld der Einsatzmöglichkeiten weitläufig. Dank ihrer Struktur und Robustheit können genetische Algorithmen auf eine Vielzahl von Problemstellungen erfolgreich angewendet werden. Anwendungsgebiete der GA sind z.B.: Finanzwirtschaft [26] und [3], Course Timetable [2], Spieltheorie [18], Behavioral Economics [4], Job scheduling [13], Ressourcenwirtschaft [8] und viele weitere. Andere Anwendungsgebiete der GP und der symbolischen Regression: Economics [31] und [24], Finanzwirtschaft [29]. Das Feld kann zudem durch Modifikationen am Algorithmus auf neue Probleme ausgeweitet werden, was ebenso für die genetische Programmierung gilt. Nachteilig sind natürlich der höhere Zeitaufwand und die benötigte Speicherkapazität, schließlich können die genetischen Operatoren nur auf eine multiple Anzahl von Definitionspunkten angewendet werden. Je größer die Populationsgröße, umso schneller terminiert der Algorithmus. Der große Vorteil genetischer Algorithmen und der genetischen Programmierung ist gleichzeitig auch der größte Nachteil: robuste und vielfältige Verfahren werden meist solange eingesetzt, bis sie von einem spezialisierten Verfahren abgelöst werden oder ihren Kostenvorteil verlieren. Existiert kein spezialisiertes Verfahren, wäre die Entwicklung mit Kosten verbunden, was die Nutzung eines bestehenden Verfahrens nahelegt. Existieren aber bereits spezialisierte Verfahren, haben genetische Algorithmen gegenüber diesen wegen ihrer geringeren Effizienz oftmals das Nachsehen, womöglich auch in der Kostenfrage (hierbei müssen natürlich wesentlich mehr Faktoren berücksichtigt werden). Durch die fortschreitende Entwicklung werden genetische Algorithmen mit der Zeit wohl nach und nach von spezialisierten Verfahren verdrängt werden, denn Speicherkapazität und Zeit sind wichtige Kostenfaktoren. Dem kann nur gegengesteuert

werden, indem neue Einsatzgebiete erschlossen werden oder der bestehende Algorithmus für gewisse Problemstellungen perfektioniert wird. Trotz ihrer Nachteile stellen genetische Algorithmen einen interessanten und alternativen Ansatz zur Lösung unterschiedlichster Problemstellungen dar und mit neuen Problemen entstehen auch neue Chancen.

A. Anhang

Listing A.1 : Bin2Dec.m

```
function [sol] = Bin2Dec(arg)

% ---------------------------------------
% Funktion, um eine Binaerzahl in eine
% Dezimalzahl umzuwandeln, um ihren
% Funktionswert berechnen zu koennen.
% ---------------------------------------

% Laenge der Binaerzahl
n = length(arg);

% Soluion gleich 0 setzen
sol = 0;

% Binaerzahl berechnen
for i=n:-1:1
  sol += 2^(n-i) * arg(i);
end

end
```

Listing A.2 : fkt.m

```
function [sol] = fkt(arg)

% ---------------------------------------
% Die Funktion, die maximiert werden soll.
% ---------------------------------------

%sol = (arg/1023)^2;
```

```
8
9  sol = sin(pi*arg/1023);
10
11 end
```

Listing A.3 : Mutation.m

```
1  function [M] = Mutation(M, method)
2
3  % ---------------------------------------
4  % Mutation:
5  % Gehe alle Bits durch und aendere
6  % es (0 zu 1 bzw. 1 zu 0) mit der
7  % Wahrscheinlichkeit 1/p (method==1).
8  % ---------------------------------------
9
10 % Groesse der Population bestimmen
11 populationSize = length( M(:,1) );
12
13 % Laenge der Binaerzahlen
14 n = length( M(1,:) );
15
16 % Wahrscheinlichkeit fuer Mutation -> 1/p
17 p = n; %* populationSize / (populationSize / 5);
18
19 % Gehe jedes Bit durch und aendere es mit der Wk 1/p
20 for i=1:populationSize
21
22   for j=1:n
23
24     % Mutationsmoegl 1:
25     % Zufallszahl aus {1,2,...,p}
26     if method == 1
27       rnd = randi(p);
28     end
29
30     % Mutationsmoegl 2:
31     % Zufallszahl aus {1,2,...,p*5/n}
32     if method == 2
33       rnd = randi( round( 2*p/j ) );
34     end
35
```

```
36      % falls rnd = 7 -> Mutation
37      if rnd == 7
38        if M(i,j) == 0
39          M(i,j) = 1;
40        end
41        if M(i,j) == 1
42          M(i,j) = 0;
43        end
44      end
45
46    end
47
48  end
49
50
51  end
```

Listing A.4 : TournamentSelection.m

```
1  function [M_new] = TournamentSelection(M)
2
3  % ---------------------------------------
4  % Erschaffe aus der alten Population eine
5  % neue. Erstelle ein Teilnehmerfeld, in
6  % dem jedes Individuum zweimal vorkommt.
7  % Waehle jeweils zufaellig zwei aus und lasse
8  % sie gegeneinander antreten (Tournament).
9  % Es setzt sich jeweils das Individuum mit
10 % der hoeheren Fitness durch und erhaelt
11 % einen Platz in der neuen Population.
12 % ---------------------------------------
13
14 % Groesse der Population bestimmen
15 populationSize = length( M(:,1) );
16
17 % Laenge der Binaerzahlen
18 n = length( M(1,:) );
19
20 % Fitness der einzelnen Individuen
21 for i=1:populationSize
22   fitness(i) = fkt( Bin2Dec( M(i,:) ) );
23 end
```

```
24
25 % Matrix fuer neue Population bestimmen
26 M_new = zeros(populationSize,n);
27
28 % participants enthaelt alle Indices der Individuen jeweils 2-mal
29 participants =  1:populationSize;
30 participants = [participants participants];
31
32 for i = 1 : populationSize
33
34   % Waehle 2 Individuen aus
35   i1 = randi( length(participants) );
36   i2 = randi( length(participants) );
37
38   % Die beiden Individuen sollen nicht gleich sein
39   while i1 == i2
40     i1 = randi( length(participants) );
41     i2 = randi( length(participants) );
42   end
43
44   % Das staerkere Individuum kommt in die neue Population
45   if fitness( participants(i1) ) >= fitness( participants(i2) )
46     M_new(i,:) = M( participants(i1) ,:);
47   end
48   if fitness( participants(i1) ) < fitness( participants(i2) )
49     M_new(i,:) = M( participants(i2) ,:);
50   end
51
52   % Die beiden Individuen aus Teilnehmerfeld loeschen
53   if i1 > i2
54     participants(i1) = [];
55     participants(i2) = [];
56   end
57   if i1 < i2
58     participants(i2) = [];
59     participants(i1) = [];
60   end
61
62 end
63
64 end
```

Listing A.5 : ElitistSelection.m

```
function [M_new] = ElitistSelection(M, crossover)

% ---------------------------------------
% selection:
% Berechne die Fitness jedes einzelnen
% Individuums und waehle die 50% der
% Population aus, welche eine hoehere
% Fitness haben und uebertrage sie
% in die neue Population.
% Diese sind gleichzeitig die parents
% fur das Crossover.
% crossover:
% Berechne nun die anderen 50% der neuen
% Population. Waehle zufaellig jeweils
% zwei der parents aus und kreuze
% sie (crossover).
% Nachdem alle parents gekreuzt wurden
% erhaelt man die neue Population, welche
% zu 50% aus den staerksten Individuen
% der alten Generation und zu 50% aus
% ihren Kreuzungen besteht.
% ---------------------------------------

% Groesse der Population bestimmen
populationSize = length( M(:,1) );

% Laenge der Binaerzahlen
n = length( M(1,:) );


% ----- selection -----


% Fitness der einzelnen Individuen
for i=1:populationSize
   fitness(i) = fkt( Bin2Dec( M(i,:) ) );
end

% Hilfsvektor, in den die Fitnesswerte uebertragen werden
hFitness = fitness;

% Bestimme die Indices der staerksten Individuen
```

```
43 fittestIndices = [];
44 for i=1:populationSize/2
45   [tempMax,tempIndex] = max(hFitness);
46   fittestIndices = [fittestIndices tempIndex];
47   hFitness(tempIndex) = [];
48 end
49
50 % erstelle neue Matrix fuer neue Population
51 M_new = zeros(populationSize,n);
52
53 % uebertrage staerkste Individuen in neue Population
54 for i=1:length(fittestIndices)
55   M_new(i,:) = M(fittestIndices(i),:);
56 end
57
58
59
60
61
62
63 % ----- crossover -----
64
65
66 % Hilfsvektor mit Indizes fuer Crossover
67 mate = zeros(populationSize/2,1);
68 for i=1:length(mate)
69   mate(i) = i;
70 end
71
72 index = populationSize+1;
73
74
75 while(index<populationSize)
76
77   % bestimme 2 Zufallszahlen
78   rnd1 = randi(length(mate));
79   rnd2 = randi(length(mate));
80
81   % duerfen nicht gleich sein
82   while rnd1 == rnd2
83     rnd1 = randi(length(mate));
84     rnd2 = randi(length(mate));
85   end
```

```
86
87  % OnePointCrossover
88  if crossover == 1
89
90    % Zufallszahl nach der fuer Crossover abgeschnitten wird
91    crossoverSite = randi(n-1)+1;
92
93    % Crossover (speichere Strings die zusammengesetzt werden sollen)
94    start1 = M( mate(rnd1), 1:crossoverSite );
95    start2 = M( mate(rnd2), 1:crossoverSite );
96    end1 = M( mate(rnd1), crossoverSite+1:n );
97    end2 = M( mate(rnd2), crossoverSite+1:n );
98
99    % Crossover (fuge sie der neuen Population hinzu)
100   M_new(index,:) = [start1 end1];
101   M_new(index+1,:) = [start2 end2];
102
103 end
104
105 % TwoPointCrossover
106 if crossover == 2
107
108   % 2 Zufallszahlen nach denen fuer Crossover abgeschnitten wird
109   crossoverSite1 = randi(n-2);
110   crossoverSite2 = randi(n-2)+1;
111
112   % crossoverSite2 muss groesser sein
113   if crossoverSite2 < crossoverSite1
114     temp = crossoverSite1;
115     crossoverSite1 = crossoverSite2;
116     crossoverSite2 = temp;
117   end
118
119   % duerfen nicht gleich sein
120   while crossoverSite1+1 > crossoverSite2
121     crossoverSite1 = randi(n-2);
122     crossoverSite2 = randi(n-2)+1;
123   end
124
125   % Crossover
126   string11 = M( mate(rnd1), 1:crossoverSite1 );
127   string12 = M( mate(rnd1), crossoverSite1+1:crossoverSite2 );
128   string13 = M( mate(rnd1), crossoverSite2+1:n );
```

```
129
130     string21 = M( mate(rnd2), 1:crossoverSite1 );
131     string22 = M( mate(rnd2), crossoverSite1+1:crossoverSite2 );
132     string23 = M( mate(rnd2), crossoverSite2+1:n );
133
134     M_new( index,:) = [string11 string22 string13];
135     M_new( index+1,:) = [string21 string12 string23];
136
137   end
138
139   % UniformCrossover
140   if crossover == 3
141
142
143   % bestimme 2 Zufallszahlen
144   rnd1 = randi(length(mate));
145   rnd2 = randi(length(mate));
146
147   % Zufallszahlen fuer Crossover
148   for i=1:n
149     crossoverSite(i) = randi(2);
150   end
151
152   % Crossover
153   temp1 = M( mate(rnd1), : );
154   temp2 = M( mate(rnd2), : );
155
156   for i=1:n
157     if crossoverSite(i) == 2
158       M_new( index, i ) = temp2(i);
159       M_new( index+1, i ) = temp1(i);
160     end
161   end
162
163
164   end
165
166   index += 2;
167
168   % abgearbeitete Indizes aus mate loeschen
169   if rnd1 > rnd2
170     mate(rnd1) = [];
171     mate(rnd2) = [];
```

```
172     end
173     if rnd1 < rnd2
174       mate(rnd2) = [];
175       mate(rnd1) = [];
176     end
177
178 end
179
180
181 end
```

Listing A.6 : RouletteWheelSelection.m

```
1  function [M_new] = RouletteWheelSelection(M)
2
3  % ---------------------------------------
4  % Erschaffe aus der alten Population eine
5  % neue. Jedem Individuum wird eine Wk
6  % zugeordnet (Anteil der Fitness des
7  % Individuums an der kumulierten Fitness).
8  % Je hoeher diese WK, desto hoeher die Wk,
9  % dass dieses Individuum in der neuen
10 % Population wieder vorkommt.
11 % Die Individuen koennen auch mehrfach
12 % in der neuen Population vorkommen.
13 % ---------------------------------------
14
15 % Groesse der Population bestimmen
16 populationSize = length( M(:,1) );
17
18 % Laenge der Binaerzahlen
19 n = length( M(1,:) );
20
21 % Fitness der einzelnen Individuen
22 for i=1:populationSize
23   fitness(i) = fkt( Bin2Dec( M(i,:) ) );
24 end
25
26 % Summe der Fitness der gesammten Population
27 sumFitness = sum(fitness);
28
29 % Anteil der einzelnen Individuen an der ges. Fitness
```

```
for i=1:populationSize
  pselect(i) = fitness(i) / sumFitness;
end

% Roulette Wheel berechnen
rouletteWheel = zeros(1,populationSize);

for i=1:populationSize
  for j=1:i
    rouletteWheel(i) += pselect(j);
  end
end
rouletteWheel = [0 rouletteWheel];

% Zufallszahlen bestimmen
random = zeros(populationSize,1);

for i=1:length(random)
  random(i) = randi(100)/100;
end

% neue Population bestimmen
M_new = zeros(populationSize,n);

for i=1:length(random)

  for j=1:length(rouletteWheel)-1
    if random(i)>rouletteWheel(j) && random(i)<=rouletteWheel(j+1)
      M_new(i,:) = M(j,:);
    end
  end

end


end
```

Listing A.7 : OnePointCrossover.m

```
function [M] = OnePointCrossover(M)

% ----------------------------------------
```

```
% Erschaffe mit Hilfe von OnePointCrossover
% aus der alten Generation eine neue.
% ------------------------------------

% Groesse der Population bestimmen
populationSize = length( M(:,1) );

% Laenge der Binaerzahlen
n = length( M(1,:) );

% Hilfsvektor mit Indizes fuer Crossover
mate = zeros(populationSize,1);
for i=1:populationSize
  mate(i) = i;
end

while(length(mate)>0)

  % bestimme 2 Zufallszahlen
  rnd1 = randi(length(mate));
  rnd2 = randi(length(mate));

  % Zufallszahl nach der fuer Crossover abgeschnitten wird
  crossoverSite = randi(n-1)+1;

  % Crossover
  temp = M( mate(rnd1), crossoverSite:n );
  M( mate(rnd1), crossoverSite:n ) = M( mate(rnd2), crossoverSite:n );
  M( mate(rnd2), crossoverSite:n ) = temp;

  % abgearbeitete Indizes aus mate loeschen
  if rnd1 > rnd2
    mate(rnd1) = [];
    mate(rnd2) = [];
  end
  if rnd1 < rnd2
    mate(rnd2) = [];
    mate(rnd1) = [];
  end
  if rnd1 == rnd2
    mate(rnd1) = [];
  end

```

```
47 end
48
49
50 end
```

Listing A.8 : TwoPointCrossover.m

```
1  function [M] = TwoPointCrossover(M)
2
3  % ---------------------------------------
4  % Erschaffe mit Hilfe von TwoPointCrossover
5  % aus der alten Generation eine neue.
6  % ---------------------------------------
7
8  % Groesse der Population bestimmen
9  populationSize = length( M(:,1) );
10
11 % Laenge der Binaerzahlen
12 n = length( M(1,:) );
13
14 % Hilfsvektor mit Indizes fuer Crossover
15 mate = zeros(populationSize,1);
16 for i=1:populationSize
17   mate(i) = i;
18 end
19
20 while(length(mate)>0)
21
22   % bestimme 2 Zufallszahlen
23   rnd1 = randi(length(mate));
24   rnd2 = randi(length(mate));
25
26   % 2 Zufallszahlen nach denen fuer Crossover abgeschnitten wird
27   crossoverSite1 = randi(n-2);
28   crossoverSite2 = randi(n-2)+1;
29
30   % crossoverSite2 muss groesser sein
31   if crossoverSite2 < crossoverSite1
32     temp = crossoverSite1;
33     crossoverSite1 = crossoverSite2;
34     crossoverSite2 = temp;
35   end
```

```
36
37   % duerfen nicht gleich sein
38    while crossoverSite1+1 > crossoverSite2
39      crossoverSite1 = randi(n-2);
40      crossoverSite2 = randi(n-2)+1;
41    end
42
43   % Crossover
44    string11 = M( mate(rnd1), 1:crossoverSite1 );
45    string12 = M( mate(rnd1), crossoverSite1+1:crossoverSite2 );
46    string13 = M( mate(rnd1), crossoverSite2+1:n );
47
48    string21 = M( mate(rnd2), 1:crossoverSite1 );
49    string22 = M( mate(rnd2), crossoverSite1+1:crossoverSite2 );
50    string23 = M( mate(rnd2), crossoverSite2+1:n );
51
52   M( mate(rnd1), : ) = [string11 string22 string13];
53   M( mate(rnd2), : ) = [string21 string12 string23];
54
55   % abgearbeitete Indizes aus mate loeschen
56    if rnd1 > rnd2
57      mate(rnd1) = [];
58      mate(rnd2) = [];
59    end
60    if rnd1 < rnd2
61      mate(rnd2) = [];
62      mate(rnd1) = [];
63    end
64    if rnd1 == rnd2
65      mate(rnd1) = [];
66    end
67
68 end
69
70
71 end
```

Listing A.9 : UniformCrossover.m

```
1 function [M] = UniformCrossover(M)
2
3 % ----------------------------------------
```

```
% Erschaffe mit Hilfe von UniformCrossover
% aus der alten Generation eine neue.
% ------------------------------------

% Groesse der Population bestimmen
populationSize = length( M(:,1) );

% Laenge der Binaerzahlen
n = length( M(1,:) );

% Hilfsvektor mit Indizes fuer Crossover
mate = zeros(populationSize,1);
for i=1:populationSize
  mate(i) = i;
end

while(length(mate)>0)

  % bestimme 2 Zufallszahlen
  rnd1 = randi(length(mate));
  rnd2 = randi(length(mate));

  % Zufallszahlen fuer Crossover
  for i=1:n
    crossoverSite(i) = randi(2);
  end

  % Crossover
  temp1 = M( mate(rnd1), : );
  temp2 = M( mate(rnd2), : );

  for i=1:n
    if crossoverSite(i) == 2
      M( mate(rnd1), i ) = temp2(i);
      M( mate(rnd2), i ) = temp1(i);
    end
  end

  % abgearbeitete Indizes aus mate loeschen
  if rnd1 > rnd2
    mate(rnd1) = [];
    mate(rnd2) = [];
  end
```

```
47    if rnd1 < rnd2
48      mate(rnd2) = [];
49      mate(rnd1) = [];
50    end
51    if rnd1 == rnd2
52      mate(rnd1) = [];
53    end
54  end
55
56
57  end
```

Listing A.10 : GA.m

```
1  function [maxFitness] = GA(MethodOut, Ausgabe)
2  tic
3  % ---------------------------------------
4  % Genetischer Algorithmus zur Maximierung
5  % der Funktion fkt(x):
6  % Zu gegebener Laenge der Binaerzahlen n
7  % und Groesse der Population populationSize
8  % erschaffe eine zufaellige Population
9  % und wende die Funktionen selection,
10 % crossover und mutation solange auf die
11 % Population an, bis die maximale Anzahl
12 % Iterationen maxIt erreicht ist und
13 % gebe den maximalen Fitnesswert zurueck.
14 % Falls Ausgabe==true gebe fuer jede
15 % Generation den maximalen Fitnesswert und
16 % die durchschnittliche Fitness aus.
17 % Falls MethodOut==true wird die genaue
18 % Methode (selection, crossover und
19 % mutation) ausgegeben.
20 % ---------------------------------------
21
22 % ---------------------------------------
23 % Auswahl der genauen Methode
24
25 % Selection:
26 % 1 = ElitistSelection
27 % 2 = RouletteWheelSelection
28 % 3 = TournamentSelection
```

```
29 selection = 1;
30
31 % Crossover:
32 % 1 = OnePointCrossover
33 % 2 = TwoPointCrossover
34 % 3 = UniformCrossover
35 crossover = 3;
36
37 % Mutation:
38 % 1 = alle Bits gleiche feste WK
39 % 2 = Bevorzugung der niederwertigeren Bits
40 mutation = 1;
41
42 % ---------------------------------------
43
44 % Laenge der Binaerzahlen
45 n = 10;
46
47 % Groesse der Population
48 populationSize = 50;
49
50 % Anzahl der Iterationen
51 maxIt = 50;
52
53 % Ausgabe in jeder Generation
54 %Ausgabe = false;
55
56 % Ausgabe der genauen Methode
57 if MethodOut == true
58
59 out = 'Methode:'
60 if selection == 1
61   out = ' Selection: ElitistSelection'
62 end
63 if selection == 2
64   out = ' Selection: RouletteWheelSelection'
65 end
66 if selection == 3
67   out = ' Selection: TournamentSelection'
68 end
69 if crossover == 1
70   out = ' Crossover: OnePointCrossover'
71 end
```

```
72 if crossover == 2
73   out = ' Crossover: TwoPointCrossover'
74 end
75 if crossover == 3
76   out = ' Crossover: UniformCrossover'
77 end
78 if mutation == 1
79   out = ' Mutation : feste WK'
80 end
81 if mutation == 2
82   out = ' Mutation : Bevorzugung der niederwertigeren Bits'
83 end
84   out = ['Populationsgroesse: ',num2str(populationSize)]
85   out = ['Iterationen: ',num2str(maxIt)]
86
87 end
88
89 % erstmal Nullmatrix erstellen
90 M = zeros(populationSize,n);
91
92 % zufaellige Binaerzahlen erstellen
93 for i=1:populationSize
94   for j=1:n
95     M(i,j) = randi(2)-1;
96   end
97 end
98
99 if Ausgabe
100   % Start-Fitness der einzelnen Individuen
101   for i=1:populationSize
102     fitness(i) = fkt( Bin2Dec( M(i,:) ) );
103   end
104 end
105
106 % bool fuer Abbruchkriterium
107 cond = true;
108
109 % zum Zaehlen der Iterationen
110 It = 0;
111
112 % Ausgabe
113 if Ausgabe
114   out = ['Generation: ',num2str(It),' Fittest: ',num2str(StartmaxFitness)]
```

```
115     out = [' ']
116 end
117
118 % Algorithmus
119 while(cond)
120
121   % Selection
122   if selection == 1
123     M = ElitistSelection(M, crossover);
124   end
125   if selection == 2
126     M = RouletteWheelSelection(M);
127   end
128   if selection == 3
129     M = TournamentSelection(M);
130   end
131
132   % Crossover
133   if crossover == 1 && selection ~= 1
134     M = OnePointCrossover(M);
135   end
136   if crossover == 2 && selection ~= 1
137     M = TwoPointCrossover(M);
138   end
139   if crossover == 3 && selection ~= 1
140     M = UniformCrossover(M);
141   end
142
143   % Mutation
144   M = Mutation(M, mutation);
145
146   % Fitness der einzelnen Individuen
147   for i=1:populationSize
148     fitness(i) = fkt( Bin2Dec( M(i,:) ) );
149   end
150
151   % maximale Fitness
152   maxFitness = max(fitness);
153
154   if Ausgabe
155     % durchschnittliche Fitness der gesammten Population
156     avg = sum(fitness)/populationSize;
157   end
```

```
158
159     It += 1;
160
161     if It >= maxIt
162       cond = false;
163     end
164
165
166     if Ausgabe
167       out = ['Generation: ',num2str(It)]
168       out = ['Fittest: ',num2str(maxFitness)]
169       out = ['Average: ',num2str(avg)]
170       out = [' ']
171     end
172
173 end
174
175 toc
176 end
```

Listing A.11 : test.m

```
1  function [] = test()
2
3  % ---------------------------------------
4  % Test zur Auswertung des Algorithmus:
5  % Lasse den Algorithmus Anz-mal durchlaufen
6  % und gebe die Quote (zu wie viel Prozent das
7  % Ergebnis des Algorithmus gleich dem exakten
8  % Ergebnis war), sowie die durchschnittliche
9  % Abweichung von der exakten Loesung aus.
10 % ---------------------------------------
11
12 % Anzahl der Tests
13 Anz = 100;
14
15 % Variable fuer Anzahl/Quote der "richtigen" Ergebnisse
16 quote = 0;
17
18 % Variable fuer durchschnittliche Abweichung vom "richtigen" Ergebnis
19 abw = 0;
20
```

```
for i=1:Anz

  % Fuehre Algorithmus aus und speichere Ergebnis
   if i==1
     fitness(i) = GA(true,false);
   else
     fitness(i) = GA(false,false);
   end

  % Ausgabe des Ergebnisses von Test i
  %out = num2str(fitness(i))

  % Zaehle Anzahl "richtiger" Ergebnisse
   if fitness(i) > 0.99999
     quote += 1;
   end

  % Abweichung vom "richtigen" Ergebnis
   abw += abs( 1 - fitness(i) );

end

% Berechne Quote + Ausgabe
quote = quote/Anz;
out = ['Quote: ',num2str(quote*100),'%']

% Berechne durchschnittliche Abweichung vom exakten Ergebnis + Ausgabe
abw = abw/Anz;
out = ['durchschnittliche Abweichung: ',num2str(abw)]

end
```

Listing A.12 : GA1D.cpp

```
//
// ------------------------------------------------------------
//
//  Genetischer Algorithmus
//
//  zur Berechnung des Maximums einer Funktion
//  f : R -> R
//
```

```
9  //  mit:
10 //
11 //  Selection: Tournament Selection
12 //  Ein Teilnehmerfeld wird erstellt, in welchem
13 //  jedes Individuum zwei mal vorkommt.
14 //  Anschliessend werden jeweils zufaellig zwei
15 //  Individuen aus dem Teilnehmerfeld gewaehlt.
16 //  Das mit der hoeheren Fitness wird ausgewaehlt
17 //  (fuer die neue Population bzw. fuer das Crossover).
18 //
19 //  Recombination: Uniform Crossover
20 //  Zwei Individuen werden zu zwei neuen rekombiniert.
21 //  Bei jeder Stelle bekommt Nachfahre 1 zufaellig von
22 //  ersten oder zweiten Vorfahren das Bit, der andere
23 //  Nachfahre das andere.
24 //
25 //  Mutation: Jedes Bit wird mit der Wahrscheinlichkeit
26 //  1 zu 10*Bitlaenge invertiert (0 zu 1 bzw. 1 zu 0).
27 //
28 //          Manuel Waidhas, 2014
29 //
30 // ------------------------------------------------------------
31
32 #include <iostream>
33 #include <cstdlib>
34 #include <cmath>
35 #include <vector>
36 #include <iomanip>
37 using namespace std;
38
39 int random(int lowerbounds, int upperbounds);
40 double diffclock(clock_t clock1, clock_t clock2);
41
42 class GA
43 {
44 private:
45 int popSize; // Populationsgroesse
46 int bitLength; // Bitlaenge
47 int maxIt; // Anzahl Iterationen
48 int** M; // Populationsmatrix
49 int** M_new; // Hilfsmatrix
50 double* fitness; // Fitnessvektor
51 int* crossoverSite; // Hilfsvektor fuer Crossover
```

```
52 double PI; // Pi
53 double arg1;
54 double arg2;
55 public:
56 GA() { PI = 3.14159265359; }; // Konstruktor
57 ~GA() {}; // Destruktor
58 double fkt(int arg[]); // Zielfunktion
59 double startGA(int psize, int blength, int maxit); // startet den
       Algorithmus
60 void selection(); // Selektion
61 void crossover(); // Crossover
62 void mutation(); // Mutation
63 };
64
65
66 int main (int argc, char const* argv[])
67 {
68   // beginne Zeitmessung
69   clock_t begin = clock();
70
71   double erg = 0;
72
73   // Starte den GA
74   GA Genetic;
75   for(int i=0; i<50; i++)
76   {
77   erg += Genetic.startGA(500,20,100);
78   }
79
80   erg = erg / 50;
81
82   cout << setprecision(8) << "Erg: " << erg << endl;
83
84   // beente Zeitmessung
85   clock_t end = clock();
86
87   // benoetigte Zeit
88   double time = diffclock(end,begin);
89   cout << setprecision(4) << "Zeit: " << time/50 << endl;
90
91   return 0;
92 }
93
```

```
94
95  double GA::fkt(int arg[])
96  {
97
98    // aus dem String die beiden Dezimalzahlen berechnen
99    arg1 = 0;
100   arg2 = 0;
101   for(int i=0; i<bitLength; i++)
102   {
103     arg1 += pow(2.0,bitLength-i-1) * arg[i];
104   }
105
106   // auf Intervall [0,1] normieren
107   double x = arg1 / ( pow(2.0,bitLength) - 1.0);
108
109   // auf Intervall [-2,2]
110   x = x * 4.0 - 2.0;
111
112   for(int k=0; k<1000; k++)
113   {
114     arg2 += ( pow(2.0,k) * sin( pow(2.0,k) * x ) ) / ( pow(3.0,k) );
115   }
116
117   return arg2;
118 }
119
120
121 double GA::startGA(int psize, int blength, int maxit)
122 {
123   // Populationsgroesse
124   popSize = psize;
125   // Bitlaenge
126   bitLength = blength;
127   // Anzahl Iterationen
128   maxIt = maxit;
129
130   // weitere Variablen
131   int It = 0;
132   double maxFitness = 0;
133   crossoverSite = new int[bitLength];
134   fitness = new double[popSize];
135
136   // Populationsmatrix
```

```
137   M = new int*[popSize];
138   for(int i=0; i<popSize ; i++)
139   {
140     M[i] = new int[bitLength];
141   }
142
143   // Hilfsmatrix
144   M_new = new int*[popSize];
145   for(int i=0; i<popSize ; i++)
146   {
147     M_new[i] = new int[bitLength];
148   }
149
150   // Zufallszahl initialisieren
151   srand(static_cast<int>(time(NULL)));
152
153   // zufaellige Population
154   for(int i=0; i<popSize ; i++)
155   {
156     for(int j=0; j<bitLength ; j++)
157     {
158       M[i][j] = random(0,1);
159     }
160   }
161
162   // Iteration
163   while(It<maxIt)
164   {
165     selection();
166
167     crossover();
168
169     mutation();
170
171     // Fitness der Individuen bestimmen
172     maxFitness = 0;
173     for(int i=0; i<popSize ; i++)
174     {
175       // Fitness bestimmen
176       fitness[i] = fkt( M[i] );
177
178       // maximale Fitness
179       if(fitness[i]>maxFitness)
```

```
180         {
181           maxFitness = fitness[i];
182         }
183       }
184
185       // It um einen erhoehen
186       It++;
187     }
188
189     // Speicher wieder freigeben
190     delete[] fitness;
191     delete[] crossoverSite;
192     for(int i=0; i<popSize ; i++)
193     {
194       delete[] M[i];
195       delete[] M_new[i];
196     }
197     delete[] M;
198     delete[] M_new;
199
200     return maxFitness;
201 }
202
203
204 void GA::selection()
205 {
206     // Zufallszahl initialisieren
207     srand(static_cast<int>(time(NULL)));
208
209     // Fitness bestimmen
210     for(int i=0; i<popSize ; i++)
211     {
212       fitness[i] = fkt( M[i] );
213     }
214
215     // Teilnehmerfeld bestimmen
216     vector <int> participants;
217     for(int i=0; i<popSize ; i++)
218     {
219       participants.push_back(i);
220     }
221     for(int i=0; i<popSize ; i++)
222     {
```

```
223       participants.push_back(i);
224     }
225
226     int i1;
227     int i2;
228
229     // das eigentliche Tunier
230     for(int i=0; i<popSize ; i++)
231     {
232       // 2 zufaellige Individuen
233       i1 = random(0,participants.size()-1);
234       i2 = random(0,participants.size()-1);
235
236       // sollen nicht gleich sein
237       while(i1==i2)
238       {
239         i1 = random(0,participants.size()-1);
240         i2 = random(0,participants.size()-1);
241       }
242
243       // das staerkere Individuum kommt in die naechste Generation
244       if( fitness[ participants[i1] ] >= fitness[ participants[i2] ] )
245       {
246         for(int j=0; j<bitLength ; j++)
247         {
248           M_new[i][j] = M[participants[i1]][j];
249         }
250       }
251       else
252       {
253         for(int j=0; j<bitLength ; j++)
254         {
255           M_new[i][j] = M[participants[i2]][j];
256         }
257       }
258
259       // beide aus dem Teilnehmerfeld loeschen
260       if(i1>i2)
261       {
262         participants.erase(participants.begin()+i1);
263         participants.erase(participants.begin()+i2);
264       }
265       else
```

```
266       {
267         participants.erase(participants.begin()+i2);
268         participants.erase(participants.begin()+i1);
269       }
270
271     }
272
273     // neue Population uebertragen
274     for(int i=0; i<popSize ; i++)
275     {
276       for(int j=0; j<bitLength ; j++)
277       {
278         M[i][j] = M_new[i][j];
279       }
280     }
281
282 }
283
284
285 void GA::crossover()
286 {
287     // Zufallszahl initialisieren
288     srand(static_cast<int>(time(NULL)));
289
290     // Hilfsvektor fuer crossover fuer die Indices
291     vector <int> mate;
292     for(int i=0; i<popSize ; i++)
293     {
294       mate.push_back(i);
295     }
296
297     int rnd1;
298     int rnd2;
299     int temp;
300
301     while(mate.empty()==false)
302     {
303       // 2 Zufallszahlen bestimmen
304       rnd1 = random(0,mate.size()-1);
305       rnd2 = random(0,mate.size()-1);
306
307       // Zufallszahlen fuer Crossover
308       for(int i=0; i<bitLength ; i++)
```

```
309     {
310       crossoverSite[i] = random(0,1);
311     }
312
313     // Crossover
314     for(int i=0; i<bitLength ; i++)
315     {
316       if(crossoverSite[i]==1)
317       {
318         temp = M[mate[rnd1]][i];
319         M[mate[rnd1]][i] = M[mate[rnd2]][i];
320         M[mate[rnd2]][i] = temp;
321       }
322     }
323
324     // abgearbeitete Indices aus mate loeschen
325     if(rnd1>rnd2)
326     {
327       mate.erase( mate.begin()+rnd1 );
328       mate.erase( mate.begin()+rnd2 );
329     }
330     if(rnd1<rnd2)
331     {
332       mate.erase( mate.begin()+rnd2 );
333       mate.erase( mate.begin()+rnd1 );
334     }
335     if(rnd1==rnd2)
336     {
337       mate.erase( mate.begin()+rnd1 );
338     }
339   }
340
341 }
342
343
344 void GA::mutation()
345 {
346   // Zufallszahl initialisieren
347   srand(static_cast<int>(time(NULL)));
348
349   // WK fuer Mutation
350   int p = 10 * bitLength;
351
```

```
352   int rnd;
353
354   for(int i=0; i<popSize ; i++)
355   {
356     for(int j=0; j<bitLength ; j++)
357     {
358       // Zufallszahl zw. 1 und p
359       rnd = random(1,p);
360
361       if(rnd==1)
362       {
363         if( M[i][j] == 0 )
364           M[i][j] = 1;
365         if( M[i][j] == 1 )
366           M[i][j] = 0;
367       }
368     }
369   }
370 }
371
372
373 int random(int lowerbounds, int upperbounds)
374 {
375   if(upperbounds==lowerbounds)
376     return upperbounds;
377   return lowerbounds + std::rand() % (upperbounds - lowerbounds + 1);
378 }
379
380
381 double diffclock(clock_t clock1, clock_t clock2)
382 {
383   double diffticks = clock1 - clock2;
384   double diff = diffticks / CLOCKS_PER_SEC;
385   return diff;
386 }
```

Listing A.13 : GA2D-HC.cpp

```
1
2 // ------------------------------------------------------------
3 //
4 //  Genetischer Algorithmus
```

```
5  //
6  //  zur Berechnung des Maximums einer Funktion
7  //  f : R^2 -> R
8  //
9  //  kombiniert mit dem Hill Climbing Verfahren
10 //
11 //  mit:
12 //
13 //  Selection: Tournament Selection
14 //  Ein Teilnehmerfeld wird erstellt, in welchem
15 //  jedes Individuum zwei mal vorkommt.
16 //  Anschliessend werden jeweils zufaellig zwei
17 //  Individuen aus dem Teilnehmerfeld gewaehlt.
18 //  Das mit der hoeheren Fitness wird ausgewaehlt
19 //  (fuer die neue Population bzw. fuer das Crossover).
20 //
21 //  Recombination: Uniform Crossover
22 //  Zwei Individuen werden zu zwei neuen rekombiniert.
23 //  Bei jeder Stelle bekommt Nachfahre 1 zufaellig von
24 //  ersten oder zweiten Vorfahren das Bit, der andere
25 //  Nachfahre das andere.
26 //
27 //  Mutation: Jedes Bit wird mit der Wahrscheinlichkeit
28 //  1 zu 10*Bitlaenge invertiert (0 zu 1 bzw. 1 zu 0).
29 //
30 //          Manuel Waidhas, 2014
31 //
32 // ------------------------------------------------------------
33
34 #include <iostream>
35 #include <cstdlib>
36 #include <cmath>
37 #include <vector>
38 #include <iomanip>
39 using namespace std;
40
41 int random(int lowerbounds, int upperbounds);
42 double diffclock(clock_t clock1, clock_t clock2);
43
44 class GA
45 {
46 private:
47 int popSize; // Populationsgroesse
```

```
48 int bitLength; // Bitlaenge
49 int maxIt; // Anzahl Iterationen
50 int** M; // Populationsmatrix
51 int** M_new; // Hilfsmatrix
52 double* fitness; // Fitnessvektor
53 int* crossoverSite; // Hilfsvektor fuer Crossover
54 double PI; // Pi
55 double solX;
56 double solY;
57
58
59 int ItHC;
60 double h;
61 double value;
62 double value1;
63 double value2;
64 double value3;
65 double value4;
66 double value5;
67 double value6;
68 double value7;
69 double value8;
70 double maxvalue;
71 public:
72 GA() { PI = 3.14159265359; }; // Konstruktor
73 ~GA() {}; // Destruktor
74 double fkt(int arg[]); // Zielfunktion
75 double f(double x, double y);
76 double Bin2Dec(int arg[]);
77 double startGA(int psize, int blength, int maxit); // startet den
    Algorithmus
78 void selection(); // Selektion
79 void crossover(); // Crossover
80 void mutation(); // Mutation
81 double HC(double x, double y);
82 };
83
84
85 int main ()
86 {
87   // beginne Zeitmessung
88   clock_t begin = clock();
89
```

```
90    double abw = 0;
91
92    // Start
93    GA Genetic;
94    for(int i=0; i<50; i++)
95    {
96      abw += 2 - Genetic.startGA(50,30,100);
97    }
98
99    cout << "abw: " << setprecision(8) << abw/50 << endl;
100
101   // beente Zeitmessung
102   clock_t end = clock();
103
104   // benoetigte Zeit
105   double time = diffclock(end,begin);
106   cout << setprecision(4) << "Zeit: " << time/50 << endl;
107
108   return 0;
109 }
110
111
112 double GA::HC(double x, double y)
113 {
114
115   value = f(x,y);
116   maxvalue = value;
117
118   value1 = f(x+h,y);
119   if(value1>value)
120     maxvalue = value1;
121
122   value2 = f(x,y+h);
123   if(value2>maxvalue)
124     maxvalue = value2;
125
126   value3 = f(x+h,y+h);
127   if(value3>maxvalue)
128     maxvalue = value3;
129
130   value4 = f(x-h,y);
131   if(value4>value)
132     maxvalue = value4;
```

```
133
134   value5 = f(x,y-h);
135   if(value5>maxvalue)
136     maxvalue = value5;
137
138   value6 = f(x-h,y-h);
139   if(value6>maxvalue)
140     maxvalue = value6;
141
142   value7 = f(x+h,y-h);
143   if(value7>maxvalue)
144     maxvalue = value7;
145
146   value8 = f(x-h,y+h);
147   if(value8>maxvalue)
148     maxvalue = value8;
149
150   if(maxvalue==value)
151   {
152     ItHC++;
153     return value;
154   }
155   else if(maxvalue==value1)
156   {
157     ItHC++;
158     return HC(x+h,y);
159   }
160   else if(maxvalue==value2)
161   {
162     ItHC++;
163     return HC(x,y+h);
164   }
165   else if(maxvalue==value3)
166   {
167     ItHC++;
168     return HC(x+h,y+h);
169   }
170   else if(maxvalue==value4)
171   {
172     ItHC++;
173     return HC(x-h,y);
174   }
175   else if(maxvalue==value5)
```

```
176     {
177       ItHC++;
178       return HC(x,y-h);
179     }
180     else if(maxvalue==value6)
181     {
182       ItHC++;
183       return HC(x-h,y-h);
184     }
185     else if(maxvalue==value7)
186     {
187       ItHC++;
188       return HC(x+h,y-h);
189     }
190     else
191     {
192       ItHC++;
193       return HC(x-h,y+h);
194     }
195 }
196 
197 
198 double GA::f(double x, double y)
199 {
200     if(x>=0 && x<=1 && y>=0 && y<=1 )
201       return sin(2*PI*x)+y;
202     return 0;
203 }
204 
205 double GA::Bin2Dec(int arg[])
206 {
207     int fkt_no = 2;
208 
209     // aus dem String die beiden Dezimalzahlen berechnen
210     double arg1 = 0;
211     double arg2 = 0;
212     for(int i=0; i<bitLength; i++)
213     {
214       arg1 += pow(2.0,bitLength-i-1) * arg[i];
215       arg2 += pow(2.0,bitLength-i-1) * arg[bitLength+i];
216     }
217 
218     // auf Intervall [0,1] normieren
```

```
219   double x = arg1 / ( pow(2.0,bitLength) - 1.0);
220   double y = arg2 / ( pow(2.0,bitLength) - 1.0);
221
222   /*
223     // auf Intervall [-5,5]x[-5,5]
224     x = x * 10 - 5;
225     y = y * 10 - 5;
226
227     solX = x;
228     solY = y;
229   */
230
231   solX = x;
232   solY = y;
233 }
234
235
236 double GA::fkt(int arg[])
237 {
238   int fkt_no = 3;
239
240   // aus dem String die beiden Dezimalzahlen berechnen
241   double arg1 = 0;
242   double arg2 = 0;
243   for(int i=0; i<bitLength; i++)
244   {
245     arg1 += pow(2.0,bitLength-i-1) * arg[i];
246     arg2 += pow(2.0,bitLength-i-1) * arg[bitLength+i];
247   }
248
249   // auf Intervall [0,1] normieren
250   double x = arg1 / ( pow(2.0,bitLength) - 1.0);
251   double y = arg2 / ( pow(2.0,bitLength) - 1.0);
252
253   if(fkt_no==1)
254   {
255     // auf Intervall [-5,5]x[-5,5]
256     x = x * 10 - 5;
257     y = y * 10 - 5;
258
259     return 50.0 - pow(x,2.0) - pow(y,2.0);
260   }
261
```

```
262   if(fkt_no==2)
263   {
264     // auf Intervall [-2,2]x[-2,2]
265     x = x * 4 - 2;
266     y = y * 4 - 2;
267
268     return x * exp( (-1)*pow(x,2.0) + (-1)*pow(y,2.0) );
269   }
270
271   if(fkt_no==3)
272   {
273     // auf Intervall [0,1]x[0,1]
274     return sin(2*PI*x)+y;
275   }
276 }
277
278
279 double GA::startGA(int psize, int blength, int maxit)
280 {
281   // Populationsgroesse
282   popSize = psize;
283   // Bitlaenge
284   bitLength = blength;
285   // Anzahl Iterationen
286   maxIt = maxit;
287
288   // weitere Variablen
289   int It = 0;
290   double maxFitness = 0;
291   crossoverSite = new int[2*bitLength];
292   fitness = new double[popSize];
293   int maxIndex = 0;
294
295   // Populationsmatrix
296   M = new int*[popSize];
297   for(int i=0; i<popSize ; i++)
298   {
299     M[i] = new int[2*bitLength];
300   }
301
302   // Hilfsmatrix
303   M_new = new int*[popSize];
304   for(int i=0; i<popSize ; i++)
```

```
305   {
306     M_new[i] = new int[2*bitLength];
307   }
308
309   // Zufallszahl initialisieren
310   srand(static_cast<int>(time(NULL)));
311
312   // zufaellige Population
313   for(int i=0; i<popSize ; i++)
314   {
315     for(int j=0; j<2*bitLength ; j++)
316     {
317       M[i][j] = random(0,1);
318     }
319   }
320
321   // Iteration
322   while(It<maxIt)
323   {
324     selection();
325
326     crossover();
327
328     mutation();
329
330     // Fitness der Individuen bestimmen
331     maxFitness = 0;
332     for(int i=0; i<popSize ; i++)
333     {
334       // Fitness bestimmen
335       fitness[i] = fkt( M[i] );
336
337       // maximale Fitness
338       if(fitness[i]>maxFitness)
339       {
340         maxFitness = fitness[i];
341         maxIndex = i;
342       }
343     }
344
345     // It um einen erhoehen
346     It++;
347   }
```

```
348
349   // x und y bestimmen
350   Bin2Dec( M[maxIndex] );
351
352   // Speicher wieder freigeben
353   delete[] fitness;
354   delete[] crossoverSite;
355   for(int i=0; i<popSize ; i++)
356   {
357     delete[] M[i];
358     delete[] M_new[i];
359   }
360   delete[] M;
361   delete[] M_new;
362
363   // Hill Climbing
364   h = pow(10.0,-4.0);
365   ItHC = 0;
366   double solution = HC(solX,solY);
367
368   //double solution = maxFitness;
369
370   //std::cout << "GA: " << maxFitness << std::endl;
371   //std::cout << "HC Iterationen: " << ItHC << std::endl;
372
373   return solution;
374 }
375
376
377 void GA::selection()
378 {
379   // Zufallszahl initialisieren
380   srand(static_cast<int>(time(NULL)));
381
382   // Fitness bestimmen
383   for(int i=0; i<popSize ; i++)
384   {
385     fitness[i] = fkt( M[i] );
386   }
387
388   // Teilnehmerfeld bestimmen
389   vector <int> participants;
390   for(int i=0; i<popSize ; i++)
```

```
391   {
392     participants.push_back(i);
393   }
394   for(int i=0; i<popSize ; i++)
395   {
396     participants.push_back(i);
397   }
398
399   int i1;
400   int i2;
401
402   // das eigentliche Tunier
403   for(int i=0; i<popSize ; i++)
404   {
405     // 2 zufaellige Individuen
406     i1 = random(0,participants.size()-1);
407     i2 = random(0,participants.size()-1);
408
409     // sollen nicht gleich sein
410     while(i1==i2)
411     {
412       i1 = random(0,participants.size()-1);
413       i2 = random(0,participants.size()-1);
414     }
415
416     // das staerkere Individuum kommt in die naechste Generation
417     if( fitness[ participants[i1] ] >= fitness[ participants[i2] ] )
418     {
419       for(int j=0; j<2*bitLength ; j++)
420       {
421         M_new[i][j] = M[participants[i1]][j];
422       }
423     }
424     else
425     {
426       for(int j=0; j<2*bitLength ; j++)
427       {
428         M_new[i][j] = M[participants[i2]][j];
429       }
430     }
431
432     // beide aus dem Teilnehmerfeld loeschen
433     if(i1>i2)
```

```
434     {
435       participants.erase(participants.begin()+i1);
436       participants.erase(participants.begin()+i2);
437     }
438     else
439     {
440       participants.erase(participants.begin()+i2);
441       participants.erase(participants.begin()+i1);
442     }
443
444   }
445
446   // neue Population uebertragen
447   for(int i=0; i<popSize ; i++)
448   {
449     for(int j=0; j<2*bitLength ; j++)
450     {
451       M[i][j] = M_new[i][j];
452     }
453   }
454
455 }
456
457
458 void GA::crossover()
459 {
460   // Zufallszahl initialisieren
461   srand(static_cast<int>(time(NULL)));
462
463   // Hilfsvektor fuer crossover fuer die Indices
464   vector <int> mate;
465   for(int i=0; i<popSize ; i++)
466   {
467     mate.push_back(i);
468   }
469
470   int rnd1;
471   int rnd2;
472   int temp;
473
474   while(mate.empty()==false)
475   {
476     // 2 Zufallszahlen bestimmen
```

```
477     rnd1 = random(0,mate.size()-1);
478     rnd2 = random(0,mate.size()-1);
479 
480     // Zufallszahlen fuer Crossover
481     for(int i=0; i<2*bitLength ; i++)
482     {
483       crossoverSite[i] = random(0,1);
484     }
485 
486     // Crossover
487     for(int i=0; i<2*bitLength ; i++)
488     {
489       if(crossoverSite[i]==1)
490       {
491         temp = M[mate[rnd1]][i];
492         M[mate[rnd1]][i] = M[mate[rnd2]][i];
493         M[mate[rnd2]][i] = temp;
494       }
495     }
496 
497     // abgearbeitete Indices aus mate loeschen
498     if(rnd1>rnd2)
499     {
500       mate.erase( mate.begin()+rnd1 );
501       mate.erase( mate.begin()+rnd2 );
502     }
503     if(rnd1<rnd2)
504     {
505       mate.erase( mate.begin()+rnd2 );
506       mate.erase( mate.begin()+rnd1 );
507     }
508     if(rnd1==rnd2)
509     {
510       mate.erase( mate.begin()+rnd1 );
511     }
512   }
513 
514 }
515 
516 
517 void GA::mutation()
518 {
519   // Zufallszahl initialisieren
```

```
520   srand(static_cast<int>(time(NULL)));
521
522   // WK fuer Mutation
523   int p = 10 * bitLength;
524
525   int rnd;
526
527   for(int i=0; i<popSize ; i++)
528   {
529     for(int j=0; j<2*bitLength ; j++)
530     {
531       // Zufallszahl zw. 1 und p
532       rnd = random(1,p);
533
534       if(rnd==1)
535       {
536         if( M[i][j] == 0 )
537           M[i][j] = 1;
538         if( M[i][j] == 1 )
539           M[i][j] = 0;
540       }
541     }
542   }
543 }
544
545
546 int random(int lowerbounds, int upperbounds)
547 {
548   if(upperbounds==lowerbounds)
549     return upperbounds;
550   return lowerbounds + std::rand() % (upperbounds - lowerbounds + 1);
551 }
552
553
554 double diffclock(clock_t clock1, clock_t clock2)
555 {
556   double diffticks = clock1 - clock2;
557   double diff = diffticks / CLOCKS_PER_SEC;
558   return diff;
559 }
```

Listing A.14 : GA2D.cpp

```
1
2  // ------------------------------------------------------------
3  //
4  //  Genetischer Algorithmus
5  //
6  //  zur Berechnung des Maximums einer Funktion
7  //  f : R^2 -> R
8  //
9  //  mit:
10 //
11 //  Selection: Tournament Selection
12 //  Ein Teilnehmerfeld wird erstellt, in welchem
13 //  jedes Individuum zwei mal vorkommt.
14 //  Anschliessend werden jeweils zufaellig zwei
15 //  Individuen aus dem Teilnehmerfeld gewaehlt.
16 //  Das mit der hoeheren Fitness wird ausgewaehlt
17 //  (fuer die neue Population bzw. fuer das Crossover).
18 //
19 //  Recombination: Uniform Crossover
20 //  Zwei Individuen werden zu zwei neuen rekombiniert.
21 //  Bei jeder Stelle bekommt Nachfahre 1 zufaellig von
22 //  ersten oder zweiten Vorfahren das Bit, der andere
23 //  Nachfahre das andere.
24 //
25 //  Mutation: Jedes Bit wird mit der Wahrscheinlichkeit
26 //  1 zu 10*Bitlaenge invertiert (0 zu 1 bzw. 1 zu 0).
27 //
28 //          Manuel Waidhas, 2014
29 //
30 // ------------------------------------------------------------
31
32 #include <iostream>
33 #include <cstdlib>
34 #include <cmath>
35 #include <vector>
36 #include <iomanip>
37 using namespace std;
38
39 int random(int lowerbounds, int upperbounds);
40 double diffclock(clock_t clock1, clock_t clock2);
41
42 class GA
```

```
43 {
44 private:
45 int popSize; // Populationsgroesse
46 int bitLength; // Bitlaenge
47 int maxIt; // Anzahl Iterationen
48 int** M; // Populationsmatrix
49 int** M_new; // Hilfsmatrix
50 double* fitness; // Fitnessvektor
51 int* crossoverSite; // Hilfsvektor fuer Crossover
52 double PI; // Pi
53 public:
54 GA() { PI = 3.14159265359; }; // Konstruktor
55 ~GA() {}; // Destruktor
56 double fkt(int arg[]); // Zielfunktion
57 double startGA(int psize, int blength, int maxit); // startet den
       Algorithmus
58 void selection(); // Selektion
59 void crossover(); // Crossover
60 void mutation(); // Mutation
61 };
62
63
64 int main (int argc, char const* argv[])
65 {
66   // beginne Zeitmessung
67   clock_t begin = clock();
68
69   double abw = 0;
70
71   // Starte den GA
72   GA Genetic;
73   for(int i=0; i<50 ; i++)
74   {
75   abw += 2 - Genetic.startGA(500,30,500);
76   }
77
78   abw = abw / 50.0;
79
80   cout << setprecision(8) << "Abw: " << abw << endl;
81
82   // beente Zeitmessung
83   clock_t end = clock();
84
```

```
85    // benoetigte Zeit
86    double time = diffclock(end,begin);
87    cout << setprecision(4) << "Zeit: " << time/50 << endl;
88
89    return 0;
90  }
91
92
93  double GA::fkt(int arg[])
94  {
95    int fkt_no = 3;
96
97    // aus dem String die beiden Dezimalzahlen berechnen
98    double arg1 = 0;
99    double arg2 = 0;
100   for(int i=0; i<bitLength; i++)
101   {
102     arg1 += pow(2.0,bitLength-i-1) * arg[i];
103     arg2 += pow(2.0,bitLength-i-1) * arg[bitLength+i];
104   }
105
106   // auf Intervall [0,1] normieren
107   double x = arg1 / ( pow(2.0,bitLength) - 1.0);
108   double y = arg2 / ( pow(2.0,bitLength) - 1.0);
109
110   if(fkt_no==1)
111   {
112     // auf Intervall [-5,5]x[-5,5]
113     x = x * 10 - 5;
114     y = y * 10 - 5;
115
116     return 50.0 - pow(x,2.0) - pow(y,2.0);
117   }
118
119   if(fkt_no==2)
120   {
121     // auf Intervall [-2,2]x[-2,2]
122     x = x * 4 - 2;
123     y = y * 4 - 2;
124
125     return x * exp( (-1)*pow(x,2.0) + (-1)*pow(y,2.0) );
126   }
127
```

```
128   if(fkt_no==3)
129   {
130     // auf Intervall [0,1]x[0,1]
131     return sin(2*PI*x)+y;
132   }
133
134   if(fkt_no==4)
135   {
136     return cos( pow(x-0.5,2)+pow(y-0.5,2) ) / exp( pow(x-0.5,2)+pow(y
      -0.5,2) );
137   }
138 }
139
140
141 double GA::startGA(int psize, int blength, int maxit)
142 {
143   // Populationsgroesse
144   popSize = psize;
145   // Bitlaenge
146   bitLength = blength;
147   // Anzahl Iterationen
148   maxIt = maxit;
149
150   // weitere Variablen
151   int It = 0;
152   double maxFitness = 0;
153   crossoverSite = new int[2*bitLength];
154   fitness = new double[popSize];
155
156   // Populationsmatrix
157   M = new int*[popSize];
158   for(int i=0; i<popSize ; i++)
159   {
160     M[i] = new int[2*bitLength];
161   }
162
163   // Hilfsmatrix
164   M_new = new int*[popSize];
165   for(int i=0; i<popSize ; i++)
166   {
167     M_new[i] = new int[2*bitLength];
168   }
169
```

```
170   // Zufallszahl initialisieren
171   srand(static_cast<int>(time(NULL)));
172
173   // zufaellige Population
174   for(int i=0; i<popSize ; i++)
175   {
176     for(int j=0; j<2*bitLength ; j++)
177     {
178       M[i][j] = random(0,1);
179     }
180   }
181
182   // Iteration
183   while(It<maxIt)
184   {
185     selection();
186
187     crossover();
188
189     mutation();
190
191     // Fitness der Individuen bestimmen
192     maxFitness = 0;
193     for(int i=0; i<popSize ; i++)
194     {
195       // Fitness bestimmen
196       fitness[i] = fkt( M[i] );
197
198       // maximale Fitness
199       if(fitness[i]>maxFitness)
200       {
201         maxFitness = fitness[i];
202       }
203     }
204
205     // It um einen erhoehen
206     It++;
207   }
208
209   // Speicher wieder freigeben
210   delete[] fitness;
211   delete[] crossoverSite;
212   for(int i=0; i<popSize ; i++)
```

```
213   {
214     delete [] M[i];
215     delete [] M_new[i];
216   }
217   delete [] M;
218   delete [] M_new;
219 
220   return maxFitness;
221 }
222 
223 
224 void GA::selection()
225 {
226   // Zufallszahl initialisieren
227   srand(static_cast<int>(time(NULL)));
228 
229   // Fitness bestimmen
230   for(int i=0; i<popSize ; i++)
231   {
232     fitness[i] = fkt( M[i] );
233   }
234 
235   // Teilnehmerfeld bestimmen
236   vector <int> participants;
237   for(int i=0; i<popSize ; i++)
238   {
239     participants.push_back(i);
240   }
241   for(int i=0; i<popSize ; i++)
242   {
243     participants.push_back(i);
244   }
245 
246   int i1;
247   int i2;
248 
249   // das eigentliche Tunier
250   for(int i=0; i<popSize ; i++)
251   {
252     // 2 zufaellige Individuen
253     i1 = random(0,participants.size()-1);
254     i2 = random(0,participants.size()-1);
255 
```

```
256     // sollen nicht gleich sein
257     while(i1==i2)
258     {
259       i1 = random(0,participants.size()-1);
260       i2 = random(0,participants.size()-1);
261     }
262
263     // das staerkere Individuum kommt in die naechste Generation
264     if( fitness[ participants[i1] ] >= fitness[ participants[i2] ] )
265     {
266       for(int j=0; j<2*bitLength ; j++)
267       {
268         M_new[i][j] = M[participants[i1]][j];
269       }
270     }
271     else
272     {
273       for(int j=0; j<2*bitLength ; j++)
274       {
275         M_new[i][j] = M[participants[i2]][j];
276       }
277     }
278
279     // beide aus dem Teilnehmerfeld loeschen
280     if(i1>i2)
281     {
282       participants.erase(participants.begin()+i1);
283       participants.erase(participants.begin()+i2);
284     }
285     else
286     {
287       participants.erase(participants.begin()+i2);
288       participants.erase(participants.begin()+i1);
289     }
290
291   }
292
293   // neue Population uebertragen
294   for(int i=0; i<popSize ; i++)
295   {
296     for(int j=0; j<2*bitLength ; j++)
297     {
298       M[i][j] = M_new[i][j];
```

```
299       }
300     }
301
302 }
303
304
305 void GA::crossover()
306 {
307   // Zufallszahl initialisieren
308   srand(static_cast<int>(time(NULL)));
309
310   // Hilfsvektor fuer crossover fuer die Indices
311   vector <int> mate;
312   for(int i=0; i<popSize ; i++)
313   {
314     mate.push_back(i);
315   }
316
317   int rnd1;
318   int rnd2;
319   int temp;
320
321   while(mate.empty()==false)
322   {
323     // 2 Zufallszahlen bestimmen
324     rnd1 = random(0,mate.size()-1);
325     rnd2 = random(0,mate.size()-1);
326
327     // Zufallszahlen fuer Crossover
328     for(int i=0; i<2*bitLength ; i++)
329     {
330       crossoverSite[i] = random(0,1);
331     }
332
333     // Crossover
334     for(int i=0; i<2*bitLength ; i++)
335     {
336       if(crossoverSite[i]==1)
337       {
338         temp = M[mate[rnd1]][i];
339         M[mate[rnd1]][i] = M[mate[rnd2]][i];
340         M[mate[rnd2]][i] = temp;
341       }
```

```
342     }
343
344     // abgearbeitete Indices aus mate loeschen
345     if(rnd1>rnd2)
346     {
347       mate.erase( mate.begin()+rnd1 );
348       mate.erase( mate.begin()+rnd2 );
349     }
350     if(rnd1<rnd2)
351     {
352       mate.erase( mate.begin()+rnd2 );
353       mate.erase( mate.begin()+rnd1 );
354     }
355     if(rnd1==rnd2)
356     {
357       mate.erase( mate.begin()+rnd1 );
358     }
359   }
360
361 }
362
363
364 void GA::mutation()
365 {
366   // Zufallszahl initialisieren
367   srand(static_cast<int>(time(NULL)));
368
369   // WK fuer Mutation
370   int p = 10 * bitLength;
371
372   int rnd;
373
374   for(int i=0; i<popSize ; i++)
375   {
376     for(int j=0; j<2*bitLength ; j++)
377     {
378       // Zufallszahl zw. 1 und p
379       rnd = random(1,p);
380
381       if(rnd==1)
382       {
383         if( M[i][j] == 0 )
384           M[i][j] = 1;
```

```
385           if( M[i][j] == 1 )
386             M[i][j] = 0;
387         }
388       }
389     }
390 }
391
392
393 int random(int lowerbounds, int upperbounds)
394 {
395   if(upperbounds==lowerbounds)
396     return upperbounds;
397   return lowerbounds + std::rand() % (upperbounds - lowerbounds + 1);
398 }
399
400
401 double diffclock(clock_t clock1, clock_t clock2)
402 {
403   double diffticks = clock1 - clock2;
404   double diff = diffticks / CLOCKS_PER_SEC;
405   return diff;
406 }
```

Literaturverzeichnis

[1] Windisch A. *Vergleichende Analyse von Genetischen Algorithmen und der Particle Swarm Optimization für den Evolutionären Strukturtest.* Diplomarbeit, Technische Universität, Berlin., 2007.

[2] S. Abdullah and H. Turabieh. Generating university course timetable using genetic algorithms and local search. In *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, volume 1, pages 254–260, Nov 2008.

[3] Marcos Alvarez-Diaz and Alberto Alvarez. Forecasting exchange rates using genetic algorithms. *Applied Economics Letters*, 10(6):319–322, 2003.

[4] Jasmina Arifovic. Genetic algorithm learning and the cobweb model. *Journal of Economic Dynamics and Control*, 18(1):3–28, 1994. Special Issue on Computer Science and Economics.

[5] Halim Ceylan and Harun Kemal Ozturk. Estimating energy demand of Turkey based on economic indicators using genetic algorithm approach. *Energy Conversion and Management*, 45(15–16):2525–2537, 2004.

[6] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Natural Computing Series. Springer, 2003.

[7] Michael J. Fogel, Lawrence J. Owens, and Alvin J. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, Chichester, WS, UK, 1966.

[8] Sylvie Geisendorf. Genetic Algorithms in Resource Economic Models. Working Papers 99-08-058, Santa Fe Institute, August 1999.

[9] E. E. George. Intrasystem transmission losses. *American Institute of Electrical Engineers, Transactions of the*, 62(3):153–158, March 1943.

[10] I. Gerdes, F. Klawonn, and R. Kruse. *Evolutionäre Algorithmen:*. Computational Intelligence. Vieweg+Teubner Verlag, 2004.

[11] A. F. Glimn, L. K. Kirchmayer, and J.J. Skiles. Improved method of interconnecting transmission loss formulas. *Power Apparatus and Systems, Part III. Transactions of the American Institute of Electrical Engineers*, 77(3):755–760, April 1958.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[13] José Fernando Gonçalves, Jorge José de Magalhães Mendes, and Maurício G. C. Resende. A hybrid genetic algorithm for the job shop scheduling problem. *EUROPEAN JOURNAL OF OPERATIONAL RESEARCH*, 167:2005, 2002.

[14] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[15] Daniel Howard and Joseph Kolibal. Solution of differential equations with genetic programming and the stochastic bernstein interpolation, 2005.

[16] Daniel Howard and Simon C. Roberts. Genetic programming solution of the convection-diffusion equation. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, page 34–41, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[17] Z. Oplatkova Lars Nolle I. Zelinka. Analytical programming – symbolic regression by means of arbitrary evolutionary algorithms.

[18] I. A. Ismail, N. A. Elramly, M. M. Elkafrawy, and M. M. Nasef. Game theory using genetic algorithms.

[19] James Kennedy and Russell C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[20] Kenneth E. Kinnear and Jr. Generality and difficulty in genetic programming: Evolving a sort, 1993.

[21] L. K. Kirchmayer and G.W. Stagg. Analysis of total and incremental losses in transmission systems. *American Institute of Electrical Engineers, Transactions of the*, 70(2):1197–1205, July 1951.

[22] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[23] John R. Koza. Genetic programming ii, 1998.

[24] Jana Juriová Marian KĺúcikMiroslav KĺúcikTime series modelling with genetic programming relative to arima models. *Eurostat*, 2009.

[25] R. Ouiddir, M. Rahli, and L. Abdelhakem-Koridak. Economic dispatch using a genetic algorithm: Application to western algeria's electrical power network. *J. Inf. Sci. Eng.*, 21(3):659–668, 2005.

[26] Robert Pereira. Genetic Algorithm Optimisation for Finance and Investments. MPRA Paper 8610, University Library of Munich, Germany, February 2000.

[27] Hong-Chan Chang Po-Hung Chen. Large-scale economic dispatch by genetic algorithm. *Power Systems, IEEE Transactions on*, 10(4):1919–1926, Nov. 1995.

[28] I.C. Report. Present practices in the economic operation of power systems. *Power Apparatus and Systems, IEEE Transactions on*, PAS-90(4):1768–1775, July 1971.

[29] Massimo Santini and Andrea Tettamanzi. Genetic programming for financial time series prediction. In Julian Miller, Marco Tomassini, PierLuca Lanzi, Conor Ryan, AndreaG.B. Tettamanzi, and WilliamB. Langdon, editors, *Genetic Programming*, volume 2038 of *Lecture Notes in Computer Science*, page 361–370. Springer Berlin Heidelberg, 2001.

[30] H.-P. Schwefel. *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. Diplomarbeit, Technische Universität, Berlin., 1965.

[31] PhilipD. Truscott and MichaelF. Korns. Detecting shadow economy sizes with symbolic regression. In Rick Riolo, Ekaterina Vladislavleva, and Jason H. Moore, editors, *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, pages 195–210. Springer New York, 2011.

[32] I.G. Tsoulos and I.E. Lagaris. Solving differential equations with genetic programming. *Genetic Programming and Evolvable Machines*, 7(1):33–54, 2006.

[33] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.