Programmieren von Algorithmen
Hausarbeit

Paul Heskamp

Kurs: Programmieren von Algorithmen im Master Life Science

September 2022

Inhaltsverzeichnis

1 Einleitung **2**

2 Theoretischer Hintergrund **2**
2.1 Bioprozessmodellierung 2
2.1.1 Wachstumsmodelle 3
2.2 Algorithmus zur Lösung von Differentialgleichungen 5
2.3 Programmstruktur 6
2.4 Genetischer Algorithmus 7
2.4.1 Generischer Genetischer Algorithmus 8
2.5 Simplex Algorithmus nach Nelder-Mead / Nelder-Mead Methode 9

3 Parameteroptimierung und Modellierung **10**
3.1 Verwendeter Code 10
3.2 Ergebnisse und Diskussion 18
3.2.1 Modellierung des Bioprozesses ohne Suchraumeinschränkung und ohne Gewichtung der Zielfunktion bei der Parameteroptimierung. 20
3.2.2 Modellierung des Bioprozesses mit Suchraumeinschränkung und ohne Gewichtung der Zielfunktion bei der Parameteroptimierung. 22
3.2.3 Modellierung des Bioprozesses mit Suchraumeinschränkung und Gewichtung der Zielfunktion bei der Parameteroptimierung. 24
3.2.4 Analyse der Parameterwerte 26

Literatur **28**

1 Einleitung

Ziel dieser Hausarbeit ist es, ein Programm in der Programmiersprache C# zu erstellen, welches einen Bioprozess modellieren kann. Als Programmierumgebung wird Visual Studio benutzt und als Projektart der Typ Windows Forms App.
Um den Bioprozess zu modellieren, muss das den Prozess beschreibende Differentialgleichungssystem gelöst werden. Ein weiterer Bestandteil der Modellierung ist die Parameteroptimierung vor der Lösung des Differentialgleichungssystems, um den Bioprozess möglichst genau nachzubilden. Hierzu soll ein genetischer Algorithmus programmiert werden, der in der Lage ist, mit verschiedenen Datentypen zu arbeiten. Abschließend sollen die Ergebnisse des Programms graphisch dargestellt werden.

2 Theoretischer Hintergrund

In den Nachfolgenden Abschnitten werden die theoretischen Grundlagen behandelt.

2.1 Bioprozessmodellierung

Bioprozesse sind hochkomplexe Verfahren, bei denen zumeist Mikroorganismen oder deren Bestandteilen verwendet werden, um ein bestimmtes biologisches Produkt zu erhalten. Bei der Modellierung durch ein mathematisches Modell müssen daher meistens einige Bestandteile des Prozesses stark vereinfacht beziehungsweise weggelassen werden. Trotzdem ist es möglich aus den Daten wertvolle Erkenntnisse abzuleiten und in vielen Fällen ist die Modellierung eine sehr gute Lösung, um einen Prozess besser zu verstehen beziehungsweise um vor einem Prozess im größeren Maßstab zu überprüfen, welche Faktoren berücksichtigt werden müssen. Eine häufige Vereinfachung bei Bioprozessmodellen ist zum Beispiel die Art der Modellierung des Stoffumsatzes. Statt die intrazellulären Stoffflüsse zu simulieren, wird dann nur eine Konstante benutzt, die beschreibt, aus wie viel Edukt eine gewisse Menge Produkt entsteht. Häufig wird auch nur die Gesamtkonzentration von Substraten in die den Prozess beschreibenden Gleichungen aufgenommen, statt zu berücksichtigen, dass die Aufnahme der Substrate durch Mikroorganismen limitiert ist. Diese Vereinfachungen mindern zwar die Genauigkeit des Modells, jedoch ist die so wesentlich geringere Modellkomplexität häufig entscheidender als der daraus resultierende Genauigkeitsverlust.[1,2]
In Bioprozessmodellen werden verschiedene Parameter eingesetzt, die zum Beispiel den Umsatz von Stoffen oder das Wachstum von Mikroorganismen beschreiben. Damit das Modell den eigentlichen Prozess möglichst gut nachbildet, müssen die Parameter korrekt gewählt werden. Für diese Anpassung werden Daten benötigt. Diese können generiert werden, indem ein Experiment im kleinen Maßstab durchgeführt wird, welches auf den später in größerem Maßstab durchgeführten Prozess übertragbar ist. Anhand dieser Daten kann anschließend mithilfe von Optimierungsverfahren eine Parameteranpassung durchgeführt werden. Eine Variante von Optimierungsverfahren ist zum Beispiel der im späteren Verlauf der Arbeit noch erwähnte genetische Algorithmus und der Simplex Algorithmus nach Nelder-Mead. Mit diesen Verfahren können die Parameter optimiert werden, indem sie so gewählt werden, dass die Modelldaten

eine möglichst geringe Abweichung zu denen im Experiment generierten Validierungsdaten besitzen..[1,3]

2.1.1 Wachstumsmodelle

Bioprozessmodelle können auf verschiedene Arten modelliert werden. Neben dem Verfahren selber (numerisches, maschinelles Lernen) stehen für die Simulation des Zellwachstums unterschiedliche Modelle zur Verfügung, die verwendet werden können. Ein solches Wachstumsmodell, welches bei numerischen Verfahren angewendet werden kann, ist die Monod-Kinetik. Hier wird die Wachstumsgeschwindigkeit in Abhängigkeit von der Substratkonzentration beschrieben. Bei der Monod-Kinetik ist die Wachstumsrate durch die Substratkonzentration limitiert.Die Monod Kinetik ist in Gleichung (1) zu sehen.

$$\mu = \mu_{max} \times \frac{C_S}{K_S + C_S} \tag{1}$$

Der Parameter μ_{max} beschreibt die maximal erreichbare Wachstumsrate, μ die aktuelle Wachstumsrate, C_S die Substratkonzentration und der Paramter K_S beschreibt die Substratkonzentration, bei der die Wachstumsrate halbmaximal ist. Da bei der Monod-Kinetik keine Substratinhibierung und keine Substrataufnahmerate berücksichtigt wird, ist das Wachstum von diesen Faktoren unabhängig. Bei einer Modellierung eines Satzreaktors mit der Monod-Kinetik wird der Wachstumsprozess nur durch die im Reaktor herrschenden Bedingungen beeinflusst.[4,5]
Die aktuelle Biomassekonzentration $C_X(t)$ ist von der Wachstumsrate und der aktuellen Biomassekonzentration abhängig (Gleichung (2)).

$$\frac{dC_X(t)}{dt} = \mu \times C_X(t) \tag{2}$$

Die aktuelle Wachstumsrate ist wie in Gleichung (1) beschrieben abhängig von C_S und K_S. C_S ergibt sich aus μ, einem Ausbeutekoeffizienten Y_{XS} und der aktuellen Biomassekonzentration (Gleichung (3)).

$$\frac{dC_S(t)}{dt} = -\frac{\mu}{Y_{XS}} \times C_X(t) \tag{3}$$

Als letztes kann noch die Produktkonzentration betrachtet werden. Diese ergibt sich aus μ, einem Ausbeutekoeffizienten Y_{XP} und der aktuellen Biomassekonzentration (Gleichung (4)).

$$\frac{dC_P(t)}{dt} = -\frac{\mu}{Y_{XP}} \times C_X(t) \tag{4}$$

Wird der Reaktor im Fed-Batch betrieben, müssen weitere Faktoren berücksichtigt werden. Während der Kultivierung ändern sich durch den Zulauf das Flüssigkeitsvolumen und die Konzentrationen im Reaktor. Den Gleichungen für die Biomassekonzentration und die Produktkonzentration müssen somit jeweils Terme hinzugefügt werden, die in Abhängigkeit von der Zulaufrate und dem Flüssigkeitsvolumen diese Verdünnung beschreiben. Für die Substratkonzentration muss gleichzeitig ein Term hinzugefügt werden, der den Zulauf von Substrat beschreibt.

Weiterhin benötigt es eine weitere Differentialgleichung, die die Veränderung des Flüssigkeitsvolumens in Abhängigkeit von der Zulaufrate berücksichtigt.
Eine weitere Möglichkeit besteht darin, einen Reaktor kontinuierlich zu betreiben. Hierbei entspricht die Zulaufrate der Abflussrate, beide können somit mit einer Durchflussrate beschrieben werden und das Flüssigkeitsvolumen bleibt konstant. Wie bei dem Betrieb im Fed-Batch müssen jedoch die Differentialgleichungen für die Biomassekonzentration, die Substratkonzentration und die Produktkonzentration ergänzt werden. Diesmal um die Abnahme der Biomassekonzentration durch den Abfluss von Biomasse und die Abnahme der Produktkonzentration durch den Abfluss von Produkt zu berücksichtigen. Weiterhin muss die Gleichung für die Substratkonzentration erweitert werden, um den konstanten Zulauf und Ablauf von Substrat zu berücksichtigen.[4,6,7]
Weitere Ansätze um das Wachstum von Mikroorganismen zu beschreiben sind das Contois Modell (siehe Gleichung (5)) und das Mason Millis-Modell (siehe Gleichung (6))

$$\mu = \mu_{max} \times \frac{C_S}{K_S \times C_X + C_S}) \tag{5}$$

Bei dem Contois Modell wird ein zusätzliche Parameter C_X eingeführt. Dadurch wird die Biomassekonzentration und die durch diese beeinflusste spezifische Wachstumsrate mitberücksichtigt.

$$\mu = \mu_{max} \times \frac{C_S}{K_S + C_S}) + K_1 \times C_S \tag{6}$$

Auch bei dem Mason-Millis Modell wird ein weiterer Parameter eingeführt. Mithilfe von K_1 wird der Transport- und Diffusionsprozess bei der Substrataufnahme mit berücksichtigt .
Mikroorganismen wachsen nicht zwangsläufig nur auf einem Substrat. Es kann sein, dass zwei verschiedene verwertbare Substrate die den gleichen Makronährstoff liefern zur Verfügung stehen. In diesem Fall können beide Substrate für das Wachstum verwendet werden. Sind zwei verwertbare Substrate vorhanden, tritt häufig das Phänomen der Diauxie auf. Der Prozess des diauxischen Wachstums beschreibt dabei das Verhalten von einigen Mikroorganismen zuerst ausschließlich auf dem einen Substrat zu wachsen und anschließend, wenn dieses verbraucht ist, den Stoffwechsel umzustellen und auf dem anderen Substrat zu wachsen.
Ein Beispiel für Diauxisches Wachstum ist das Wachstum von *Escherichia coli* auf Glucose und Lactose. Erst wächst *E. coli* ausschließlich auf Glucose. Ist diese vollständig umgesetzt wird der Stoffwechsel umgestellt und Lactose verwertet. Hierbei wird von *E. coli* unter anderem das Enzym β-Galactosidase hergestellt, welches für die Umsetzung von Lactose benötigt wird. Dieses Verhalten resultiert in einer markanten Wachstumskurve, da das Wachstum sich zum Zeitpunkt der Stoffwechselumstellung verlangsamt. Dies ist in Abb. 1 zu sehen.[8,9] Um diauxisches Wachstum zu modellieren werden meist sogenannte strukturierte Modelle mit mehreren Zuständen benutzt. Beim diauxischen Wachstum wären das zum Beispiel die Zustände: Wachstum auf Substrat 1 und Wachstum auf Substrat 2. Optimalerweise könnte noch ein dritter Zustand eingeführt werden, der den Übergangszustand und die notwendigen Stoffwechselumstellung beschreibt. Neben diauxischem Wachstum ist auch die simultane Verwertung (co-utilization) von zwei verschiedenen Nährstoffen möglich.

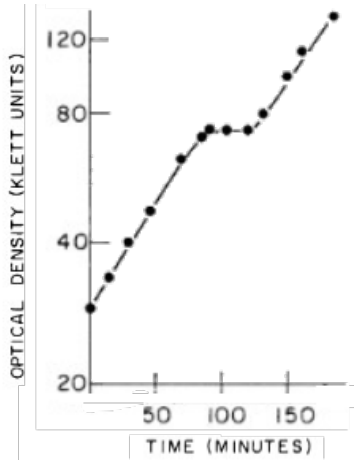

Abbildung 1: Diaxuisches Wachstum von *Escherichia coli*.[10]

Glucose-Laktose Diauxie. Wachstum auf Laktose bis ca. 90 Minuten, anschließend Stoffwechselumstellung und Wachstum auf Lactose.

Ein Beispiel für simultanes Wachstum auf zwei Nährstoffen ist das in dieser Hausarbeit betrachtete Wachstum des Stäbchenbakteriums *Proteus vulgaris*. Dieses baut Biomasse auf, indem es zwei verfügbare Substrate – Zitronensäure und Glucose – verwertet. Hierbei werden beide Substrate gleichzeitig genutzt.
Meistens wird davon ausgegangen, dass Mikroorganismen entweder diauxisch wachsen oder Substrate simultan verwerten. Gerade außerhalb des Labors bei komplexeren Nährstoffzusammensetzungenn schließen sich diese beiden Wachstumsstrategien jedoch nicht aus, wie zum Beispiel Perrin et al. gezeigt hat. Es sind somit auch Mischformen dieser Wachstumsarten möglich.[9,11]

2.2 Algorithmus zur Lösung von Differentialgleichungen

Differentialgleichungen lassen sich analytisch oder numerisch lösen. Dabei ist nicht jede Differentialgleichung analytisch lösbar. Generell funktioniert die numerische Lösung von Differentialgleichungen immer nach dem ähnlichen Prinzip: Anhand eines Anfangswertes überprüft, wie sich der Wert der Differentialgleichung verändert und anhand dieser Änderung der weitere Verlauf extrapoliert. Für die numerische Lösung ist dabei die Schrittweite mit der extrapoliert wird von zentraler Bedeutung. Würde eine unendlich kleine Schrittweite gewählt, wäre die durch numerische Lösung identisch zu der analytischen. Da es jedoch nicht möglich ist eine unendlich kleine Schrittweite zu wählen und mit kleinerer Schrittweite die Rechenzeit zunimmt, muss in der Praxis eine angemessene Schrittweite gefunden werden, die ein ausreichend genaues Ergebnis liefert. Auch Differentialgleichungssysteme können numerisches gelöst werden. Dabei muss jedoch beachtet werden, dass Differentialgleichungssysteme steif sein können. Das bedeutet, dass bei ihnen bei der Anwendung bestimmter numerische Methoden die Lösung der Gleichung instabil ist (also starke Schwenkungen aufweist), sofern die Schrittweite nicht extrem

klein gewählt wird. Eine genaue Definition ist dabei schwierig, generell treten steife Differentialgleichugnssysteme jedoch auf, wenn einige Terme der Gleichung zu einer schnellen und großen Veränderung der Lösung führen können.[12,13]
Eine Möglichkeit um Differentialgleichungen numerisch zu Lösen sind die Runge-Kutta-Methoden. Dabei handelt es sich um eine Reihe von iterativen Methoden zur numerischen Lösungen von Gleichungen verschiedener Ordnung, die durch Carl Runge und Wilhelm Kutta entwickelt wurde. Besonders das Runge-Kutta-Verfahren 4. Ordnung wird dabei häufig verwendet, da es sehr präzise Lösungen bei angemessener Schrittweite liefert. Weiterhin ist sie zum lösen steifer Differentialgleichungssysteme bei vertretbaren Rechenaufwand geeignet. Bei diesem Verfahren wird der Verlauf der Lösungsfunktion in einzelne Bereiche beziehungsweise Intervalle aufgeteilt, in denen der Funktionsverlauf durch eine Gerade approximiert wird. Die Geradensteigung in jedem Teilintervall ist dabei ein Mittelwert aus verschiedenen Steigungen an den Rändern und in der Intervallmitte. Die einzelnen Gleichungen für die verschiedenen Hilfsteigungen sind nachfolgend dargestellt.

$$\begin{aligned} k_1 &= \Delta t f(t_{i-1}, x_{i-1}) \\ k_2 &= \Delta t f(t_{i-\frac{1}{2}}, x_{i-1} + \frac{k_1}{2}) \\ k_3 &= \Delta t f(t_{i-\frac{1}{2}}, x_{i-1} + \frac{k_2}{2}) \\ k_4 &= \Delta t f(t_{i-1} + \Delta t, x_{i-1} + k_3) \end{aligned} \tag{7}$$

Der Mittelwert wird anhand von Gleichung (8) bestimmt.

$$x_i = x_{i-1} + \frac{1}{6} \times (k_1 + 2k_2 + 2k_3 + k_4) \tag{8}$$

2.3 Programmstruktur

Die Programmiersprache C# ist eine objektorientierte Programmiersprache. Das bedeutet, dass mit dieser in der Regel objektorientiert programmiert wird und nicht wie in einigen anderen Programmiersprachen prozedural. Die Objektorientierte Programmierung (OOP) unterscheidet sich dadurch von der prozeduralen Programmierung, dass die Struktur auf Klassen und Objekten beruht, die nahezu wie Bausteine immer wieder verwendet und einzeln modifiziert werden können. Eine Klasse ist dabei ein abstraktes Grundgerüst auf welchem die einzelnen darin enthaltenen Objekte aufbauen können. Objekte und Klassen können dabei beliebig miteinander kommunizieren und interagieren.[14]
Bei der prozeduralen Programmierung wird im Gegensatz zur Objektorientierten Programmierung ein Top-Down-Ansatz verwendet, bei der eine Reihe von Prozeduren beziehungsweise Unterprogramme die Hauptbausteine darstellen. Diese werden in einer bestimmten Reihenfolge ausgeführt um eine Problemstellung zu lösen. Zum Beispiel der Programmiersprache C ist dieser Programmablauf in einem bestimmten Programmteil festgelegt, der sogenannten Main Datei. Diese bestimmt den gesamten Ablauf der Programmierung.[15]
Klassen strukturieren bei der objektorientierten Programmierung das Programm und unterteilen das Grundproblem in viele kleinere Teilprobleme beziehungsweise kleinere Teilaufgaben. Ist

nun eine Teilaufgabe in einer Klasse vollständig gelöst und deckt alle nötigen Funktionalitäten ab, kann der Fall auftreten, dass im späteren Verlauf weitere Funktionalitäten benötigt werden. Hierbei ist es häufig nicht sinnvoll, die weiteren Funktionalitäten der ursprünglichen Klasse hinzuzufügen, da diese an sich schon für einen Aufgabenbereich komplett ist. Für dieses Problem gibt es in der Objektorientierten Programmierung eine Lösung mit dem Namen Vererbung.
Bei der Vererbung wird eine neue Klasse erstellt, die die Funktionalitäten der ursprünglichen Klasse erbt. Die ursprüngliche Klasse wird dabei Basisklasse und die erbende Klasse als abgeleitete Klasse beziehungsweise Subklasse bezeichnet. Vererbung ist dabei mehrfach möglich und es kann eine weitere Klasse von einer Subklasse abgeleitet werden kann. Der Funktionsbereich kann so beliebig strukturiert und erweitert werden.[14]

2.4 Genetischer Algorithmus

Genetische Algorithmen sind Methoden zur Lösung von Optimierungsproblemen. Der Fokus bei dem genetischen Algorithmus liegt auf dem auffinden eines globalem Minimums (oder Maximums). Die Grundidee hinter dem Algorithmus ist die Nachbildung einiger Prozesse aus der natürlichen Evolution. Ein genetischer Algorithmus ist ein iterativer, heuristischer Prozess. Er ist sehr flexibel einsetzbar, da er relativ einfach an viele verschiedene Problemstellungen angepasst werden kann, allerdings kann eine komplexe Problemstellung leicht zu sehr langen Rechenzeiten führen und es ist nicht garantiert, dass der Algorithmus das optimale oder auch nur ein brauchbares Ergebnis liefert[16,17]
Der generelle Ablauf eines genetischen Algorithmus ist in Abb. 2 dargestellt.

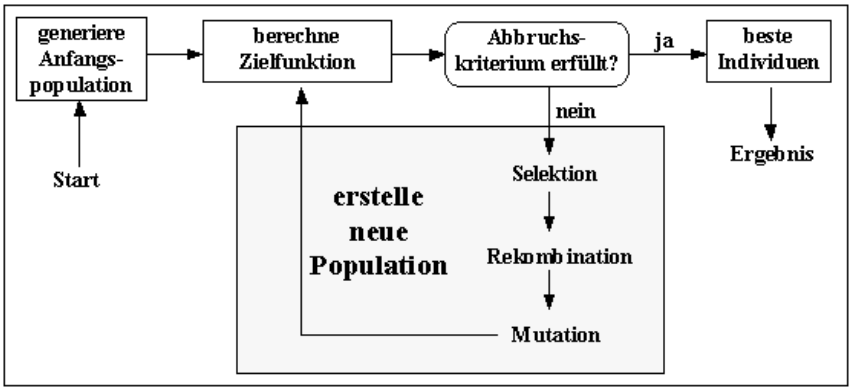

Abbildung 2: Genereller Ablauf von einem genetischen Algorithmus.[18]

Zuerst wird eine Startpopulation mit zufälligem Genom generiert. Das Genom bezeichnet hierbei die Werte, die die einzelnen Individuen enthalten und welche die möglichen Lösungswerte für das Optimierungsproblem darstellen. Hierbei kann der Bereich in welchem die zufälligen Werte liegen beschränkt werden beziehungsweise die Art wie diese generiert werden kann generell verschieden modifiziert werden.
Nach dem erzeugen der Startpopulation wird die Fitness der Individuen bestimmt, indem die Werte in die Fitness- beziehungsweise die Zielfunktion eingegeben wird. Anschließend wird

überprüft ob das Abbruchkriterium erfüllt wurde. Dieses kann zum Beispiel ein bestimmter Fitnesswert sein oder eine bestimmte Anzahl von Iterationen. Ist das Abbruchkriterium erfüllt wurde ein Ergebnis erreicht und der Algorithmus ist zu Ende. Ist es nicht erfüllt, folgt die erste beziehungsweise nächste Iteration, in der eine neue Population aus der aktuellen Population erstellt wird.
Eine Iteration besteht in der Regel aus drei Schritten: der Selektion, der Rekombination und der Mutation. Zuerst werden die Eltern für die neue Population selektiert. Hierfür gibt es verschiedene Methoden. Eine Variante ist die Tournament Selection. Hierbei wird eine bestimmte Anzahl an Individuen zufällig aus der aktuellen Population ausgewählt und das jeweils beste wird als Elternteil für die neue Population ausgewählt. Diese Art der Auswahl hat den Vorteil, dass auch Individuen die nicht so gute Fitnesswerte besitzen ihre Gene weitergeben können und so die genetische Vielfalt erhalten bleibt. Gleichzeitig wird weiterhin nach Fitness selektiert und nicht rein zufällig die Elterngeneration ausgewählt. Mit dem Parameter der Tournament Size, also der Anzahl der am Tournament Teilnehmenden Individuen kann entweder genetische Vielfalt oder eine srengere Selektion priorisiert werden.
Zwischen den so ausgewählten Eltern findet mit einer vorher festgelegten Wahrscheinlichkeit Rekombination statt. Das bedeutet, dass sie Teile ihres Genoms untereinander austauschen. Nach der Rekombination findet die Mutation statt. Hier wird durch eine vorher festgelegte Wahrscheinlichkeit bestimmt, ob ein oder mehrere Gene des Genoms der Individuen zufällig neu generiert werden. Dieser Schritt ist essentiell, da durch ihn neue Werte in den Algorithmus eingehen. Ansonsten wäre es möglich, dass nur mit den Werten der Startpopulation kein brauchbares Ergebnis erreicht werden kann. Mit der neuen Population beginnt der Algorithmus wieder bei der Berechnung der Fitnessfunktion mit anschließender Überprüfung des Abbruchkriteriums.[16–18]
Eine Problem bei dem gerade erklärten Ablauf ist ein möglicher Informationsverlust durch Mutation. Werden bei einem sehr vorteilhaften Individuum ein oder mehrere Gene mutiert, ist es möglich das die maximale erhaltene Fitness von einer Generation auf die nächste abnimmt. Dem kann entgegengewirkt werden, indem eine festgelegte Anzahl von Individuen mit der höchsten Fitness unverändert in die neue Population eingehen. Dieses Prinzip wird Elitismus genannt. Neben der Vermeidung eines möglichen Informationsverlustes hat diese Vorgehensweise noch den Vorteil, dass die Ausführungszeit des Genetischen Algorithmus im allgemeinen verkürzt wird. Allerdings ist es auch möglich, dass sich das Prinzip negativ auf das Ergebnis auswirkt, wenn zum Beispiel eine unverhältnismäßig hohe Anzahl von Elite Individuen unverändert an die nächste Generation weitergegeben wird, da dies die genetische Vielfalt zu sehr einschränkt.[17]

2.4.1 Generischer Genetischer Algorithmus

Es gibt verschiedene Möglichkeiten einen genetischen Algorithmus in C# zu Implementieren. Soll der Algorithmus flexibel für verschiedene Datentypen benutzt werden können, bietet es sich an den Mechanismus der Vererbung oder den generischen Datentyp in C# zu verwenden. In diesem Abschnitt wird das Prinzip hinter dem generischen genetischen Algorithmus erklärt. Der generische Datentyp oder Generics in C# bezeichnen Variablen, die nicht als Platzhalter für Daten eines bestimmten Datentyps, sondern als Platzhalter für den Datentyp selber dienen.

So lässt sich ein Objekt in der Programmierung so gestalten, dass der Anwender oder ein anderes Objekt in dem Objektaufruf den Datentyp festlegen kann. Zum Beispiel könnte ein Objekt welches ein Array zu einer Liste konvertiert sowohl ein String-Array, Integer-Array oder jede weitere Art von Array zu einer Liste konvertieren.[14]
Am Beispiel des genetischen Algorithmus funktioniert das ganze ähnlich. Hier wird statt eines auf einen Datentypen festgelegten Genoms ein generisches Genom verwendet. Weiterhin werden diverse Objekte per Delegate übergeben, da auch die Umsetzung von Einzelnen Objekten innerhalb der Klasse des Genetischen Algorithmus abweichend implementiert werden. Hierzu zählt zum Beispiel die Berechnung der Fitness oder der Ablauf der Mutation. Die Berechnung der Fitness ist hier jedoch sowieso ein Sonderfall, da diese auch bei einem Genetischen Algorithmus mit einem festen Datentyp variabel gehalten werden sollte, ansonsten wäre der Algorithmus nur für eine beim programmieren festgelegte Problemstellung geeignet. Sonst müsste die Klasse selber müsste durch den Nutzer umprogrammiert werden, was generell nicht erwünscht ist.

2.5 Simplex Algorithmus nach Nelder-Mead / Nelder-Mead Methode

Der Simplex Algorithmus nach Nelder-Mead ist eine numerische heuristische Methode zur Lösung von linearen und nicht linearen Optimierungsproblemen. Im Gegensatz zu dem im vorherigen Abschnitt beschriebenen genetischen Algorithmus liegt der Fokus beim Simplex Algorithmus auf der Optimierung eines lokalen Minimums mit hoher Genauigkeit. Somit ergänzen sich die beiden Methoden sehr gut, da zuerst eine globale Suche mit dem genetischen Algorithmus und dann eine lokale Optimierung mit dem Simplex Algorithmus durchgeführt werden kann. Der Simplex Algorithmus wird auch als Nelder-Mead Methode bezeichnet. Dies dient der namentlichen Abgrenzung von der Simplex Methode von Dantzig zur Lösung linearer Optimierungsprobleme.[13,19]
Mit der Nelder-Mead Methode können n-dimensionale Problemstellungen gelöst werden und es ist keine Ableitung der Zielfunktion notwendig, sie muss nur im Verlauf des Algorithmus öfter ausgewertet werden, um die Funktionswerte und somit die Güte der einzelnen Punkte zu bestimmen. Dies kann sich gerade bei komplexeren Zielfunktionen negativ auf die Rechenzeit auswirken. Da der Simplex Algorithmus nach Nelder-Mead nur an bestimmten Punkten im n-dimensionalen Raum einige Funktionswerte benötigt und keine Information über den Funktionsgradienten, zählt er zu den direct Search Methoden.
Bei dem Algorithmus wird der sogenannte Simplex verwendet. Bei diesem handelt es sich um ein begrenztes n-dimensionales Polygon mit einem Innenraum und exakt n + 1 Eckpunkten. Im zweidimensionalen Raum handelt es sich bei einem Simplex um ein Dreieck. Weiterhin ist der Ablauf des Algorithmus zur Minimierung einer Funktion im zweidimensionalen Raum relativ anschaulich zu erklären. Ohne Anfangswerte werden 3 zufällig generierte Punkte generiert. Mit Anfangswerten werden mit einem Punkt zwei zusätzliche Punkte generiert, indem zu jeweils einer Koordinate ein Abstand addiert wird. In beiden Fällen werden diese 3 Punkte im nächsten Schritt mit der Zielfunktion ausgewertet und sortiert. Anschließend wird ein neuer Punkt generiert in dem der Punkt mit dem höchsten Funktionswert an dem Mittelpunkt der anderen beiden Punkte gespiegelt wird.

Die Spiegelung beziehungsweise das voranschreiten des Simplex kann dabei unterschiedlich durchgeführt werden. Durch Multiplikation mit unterschiedlichen Konstanten kann der Simplex entweder eine normale, größere, kleinere oder rückwärtige Schrittweite besitzen. In diesem Zusammenhang wird von einem normalen, expandierten, kontrahierten oder negativ kontrahierten Simplex gesprochen. So kann verhindert werden, dass der Algorithmus bei einer festen Schrittweite nur bis zu einem bestimmten Punkt konvergieren kann, da er über das eigentliche Minimum hinaus schreitet.[13,19,20]
Nach diesem Schritt wird der neue Punkt mit der Zielfunktion ausgewertet und die Punkte werden wieder sortiert. An dieser Stelle wird das Abbruchkriterium getestet und der Algorithmus kann bei entsprechendem Ergebnis beendet werden. Mögliche Abbruchkriterien sind hierbei eine maximale Anzahl von Iterationen, eine maximal abgelaufene Zeit oder ein Test um festzustellen ob der Simplex konvergiert. Das konvergieren des Simplex kann überprüft werden, indem getestet wird wie weit die Eckpunkte des Simplex auseinander liegen. Unterscheiden sie sich nur noch minimal wird der Algorithmus beendet.[19]

3 Parameteroptimierung und Modellierung

Es soll ein Bioprozessmodell für das Wachstum des Stäbchenbakteriums *Proteus vulgaris* erstellt werden. Dieses wächst auf zwei Substraten und es sind Daten aus einer vorangegangenen Kultivierung vorhanden. Für die Parameteroptimiertung und Modellierung des Bioprozesses werden mehrere Programmkomponenten beziehungsweise Algorithmen benötigt. Eine besonders wichtige Komponente ist dabei das Differentialgleichungssystem, welches den Bioprozess selber beschreibt.

3.1 Verwendeter Code

In Gleichung (9) ist das den Bioprozess beschreibende Differentialgleichungssystem zu sehen.

$$
\begin{aligned}
\mu_1 &= \mu_{max1} \times \frac{C_{S1}}{K_{S1} + C_{S1}} \\
\mu_2 &= \mu_{max2} \times \frac{C_{S2}}{K_{S2} + C_{S2}} \\
\frac{dC_X(t)}{dt} &= (\mu_1(C_{S1}) + \mu_2(C_{S2})) \times C_X(t) \\
\frac{dC_{S1}(t)}{dt} &= \frac{\mu_1(C_{S1}) \times C_X(t)}{Y_{XS1}} \\
\frac{dC_{S2}(t)}{dt} &= \frac{\mu_2(C_{S2}) \times C_X(t)}{Y_{XS2}}
\end{aligned}
\tag{9}
$$

Umgewandelt als Methode im Code ist es in Codebeispiel 1 dargestellt.

Codebeispiel 1: Differentialgleichungen in C#.

```
/// <summary>
/// Differential Equation System that models the bioprocess
```

```
    /// </summary>
    /// <param name="x">double with process parameters</param>
    /// <param name="t">time if needed</param>
    /// <returns></returns>
    private double[] ODESystem(double[] x, double t) {
        double[] dxdt = new double[3];
        ///  CS1 = 0, CS2 = 1, CX = 2,
        double mue1 = mueMax1 * x[0] / (kS1 + x[0]);
        double mue2 = mueMax2 * x[1] / (kS2 + x[1]);
        dxdt[0] = -(mue1 * x[2]) / yXS1;
        dxdt[1] = -(mue2 * x[2]) / yXS2;
        dxdt[2] = (mue1 + mue2) * x[2];
        return dxdt;
    }
```

Die Methode bekommt ein Double Array und eine Zeit als Eingangsparameter und gibt als Ausgabe die berechneten Funktionswerte zurück.
Für die Lösung des Differentialgleichungssystems benötigt es einen weiteren Methode. Hierzu wird ein Algorithmus zur Lösung von Differentialgleichungen benötigt. In diesem Fall wird das Runge-Kutta-Verfahren 4. Ordnung verwendet. Die dazu gehörigen Gleichungen wurden bereits in den Gleichungen (7) und (8) beschrieben. In Codebeispiel 2 ist eine Zusammenfassung der wichtigsten Methodenbestandteile gezeigt.

Codebeispiel 2: Beschreibung der RungeKutta4 Methode in C#.

```
    /// <summary>Numerically solve the system of ordinary differential equations
        given in equations using the Runge Kutta method of the 4th degree.</summary>
...
    /// <returns>A list of arrays (time and values of process variables) for each
        output time point.</returns>
    public static List<double[]> RungeKutta4(double[] InitialValues, double t0,
        double tEnd, double deltaT, ODE equations,
    double dtOutput, double[] LowerBounds = null, double[] UpperBounds = null){
...
            // get derivatives
            double[] dxdt = equations(x, t);
            //Runge Kutta 4 Algorithm
            for (int i = 0; i < x.Length; i++) {
                k1[i] = deltaT * dxdt[i];
                k2[i] = deltaT * (dxdt[i] + 0.5 * k1[i]);
                k3[i] = deltaT * (dxdt[i] + 0.5 * k2[i]);
                k4[i] = deltaT * (dxdt[i] + k3[i]);
                x[i] += (1.0 / 6.0) * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]);
            }
...
    }
```

Als zu optimierendes Differentialgleichungssystem (ODE equations) wird der in Codebeispiel 1 gezeigte Code genutzt. Weitere zu erwähnende Eingabeparameter sind die Initialwerte für die nummerische Lösung , die Anfangszeit, Endzeit, die Schrittweite und das Ausgabezeitintervall (InitialValues, t0, tEND, deltaT, dtOutput). Die Ausgabe der Methode ist eine Liste mit den Zeitintervallen und den dazugehörigen Werten. In diesem Fall sind das C_{S1}, C_{S2} und C_x.

Das Bioprozessmodell besitzt sechs Parameter, welche so angepasst werden müssen, dass das Modell für den aktuellen Bioprozess brauchbare Daten liefert. Die Parameter sind μ_{max1}, μ_{max2}, KS_1, KS_2, Y_{XS1} und Y_{XS2}. In dieser Hausarbeit werden die Parameter zuerst mit einem genetischen Algorithmus und anschließend mit dem Simplex Algorithmus nach Nelder-Mead optimiert. Der hier verwendete genetische Algorithmus ist wie oben bereits beschrieben mithilfe des generischen Datentyps programmiert. In Codebeispiel 3 ist der Konstruktor für die Individuen des genetischen Algorithmus gezeigt.

Codebeispiel 3: Individuum für den genetische Algorithmus.

```
public class Individual<T> : IComparable<Individual<T>> {
...
    /// <summary>
    /// Constructor for an Individual
    /// </summary>
    /// <param name="genomeSize">Size of the Genome Array</param>
    /// <param name="genomeLowerBound">Lower bounds for genome generation</param>
    /// <param name="genomeUpperBound">// upper bounds for genome generation</param>
    /// <param name="random">random object</param>
    /// <param name="getRandomGene">Function for generating random gene</param>
    /// <param name="fitnessFunction">Function for calculating fitness</param>
    /// <param name="initialiseGenome">Should Genome be randomly initilized</param>
    public Individual(int genomeSize, T[] genomeLowerBound, T[] genomeUpperBound,
        Random random, Func<T, T, T, T> getRandomGene, Func<T[], double>
        fitnessFunction, bool initialiseGenome) {
    ...
        }
```

Zu erwähnende Eingabeparameter sind hier die Genomgröße (genomeSize), welche in diesem Fall sechs beträgt. Weiterhin ist es möglich die Ober- und Untergrenzen für die einzelnen Gene des Genoms zu setzen (genomeUpperBound, genomeLowerBound). Außerdem wird die in dem Algorithmus verwendete Fitnessfunktion und eine Methode zur Erzeugung eines zufälligen Gens übergeben, da diese Funktionen je nach Datentyp und Anwendung abweichen. Für den Datentyp Double ist die Methode in Codebeispiel 4 gezeigt.

Codebeispiel 4: Methode zur Erzeugung eines zufälligen Gens. Normalverteilt oder gleich verteilt mit Ober- und Untergrenze.

```
    /// <summary>
    /// Gets a random double, if mue != null, generates normal distributed doubles
        with mue as the Median.
    /// </summary>
    /// <param name="max">max value</param>
    /// <param name="min">min value</param>
    /// <param name="mue">mue for normal distribution</param>
    /// <returns></returns>
    private double GetRandomDouble(double max, double min, double mue) {
        if (mue == 0) {
            return random.NextDouble() * (max - min) + min;
        }
        double number;
        do {
```

```
            number = myRandom.RandomNormalDistribution(mue, sigma, random);
        } while (number < min || number > max);
        return number;
    }
```

Wird der Funktion dabei μ übergeben werden zufällige normalverteilte Doubles mit μ als Median und σ als Streuwert der Normalverteilung erzeugt. Ist μ nicht gegeben wird ein gleichverteilter Double erzeugt. Die Erzeugung der Zufallswerte erfolgt dabei immer innerhalb der gegebenen Ober- und Untergrenzen.
Die Methode für die Fitnessfunktion ist in Codebeispiel 5 zu sehen.

Codebeispiel 5: Zielfunktion für den Genetischen Algorithmus. Berechnet die Fehlerquadratsumme zwischen den gemessenen Werten der 3 Prozessgrößen (Validierungsdaten) und den simulierten Daten.

```
    /// <summary>
    /// This fitness function uses the MeanSquaredError of the modeled bioprocess
        data and the validationData.
    /// </summary>
    /// <param name="index">Index of the Individum of the genetic algorithm
        Population</param>
    /// <returns></returns>
    public double FitnessFunction(double[] genome) {
        double cS10 = validationData[0][1];
        double cS20 = validationData[0][2];
        double cX0 = validationData[0][3];
        double mueMax1 = genome[0];
        double mueMax2 = genome[1];
        double KS1 = genome[2];
        double KS2 = genome[3];
        double YXS1 = genome[4];
        double YXS2 = genome[5];
        BioprocessModel bioprocess = new BioprocessModel(cS10, cS20, cX0, mueMax1,
            mueMax2, KS1, KS2, YXS1, YXS2);
        dataList = bioprocess.Run();
        //CS1 = 0, CS2 = 1, CX = 2,
        double[] weights = new double[] { 1, 1, 1.2 };
        double error = Error.SquaredError(dataList, validationData, true, weights);
        if (Double.IsNaN(error)) {
            error = double.PositiveInfinity;
        }
        return error;
    }
```

Als Eingabe erhält die Methode das Genom eines Individuums aus dem genetischen Algorithmus. Ausgabe ist die Fehlerquadratsumme der Validierungsdaten und der Simulationsdaten. Innerhalb der Fitnessfunktion werden die relevanten Daten an das Bioprozessmodell übergeben um die Simulationsdaten zu erhalten. Diese werden anschließend mit einer Methode die die Fehlerquadratsumme berechnet verarbeitet. Bei der resultierenden Fehlerquadratsumme wird zusätzlich überprüft, ob der Wert definiert ist (kein NaN/NotaNumber). Ist er es nicht, wird der Fehler gleich unendlich gesetzt, damit Individuuen die nicht durch den Bioprozess model-

lierbare Parameter enthalten aussortiert werden.
In Codebeispiel 6 ist die Methode zur Bestimmung der Fehlerquadratsumme gezeigt.

Codebeispiel 6: Methode zur Berechnung der Fehlerquadratsumme. Optional mit Gewichtung

```
    /// <summary>
    ///Calculates the squared Error of a data and validation data pair. Both
        datasets must have the same length and width.
    ///The first column must be the time and the rest data points.
    /// Optionally weights the differents parameters with given values
    /// </summary>
    /// <param name="data"></param>
    /// <param name="validationData"></param>
    /// <param name="useWeights">(optional) use weights or not</param>
    /// <param name="weights">must be the same size as the double arrays of the
        data lists</param>
    /// <returns></returns>
    public static double SquaredError(List<double[]> data, List<double[]>
        validationData, bool useWeights = false, double[] weights = null) {
...ArgumentExceptions
    int arrayLength = data[0].Length;
    double[] sum = new double[arrayLength-1];
    for (int i = 0; i < sum.Length; i++) sum[i] = 0;
    double difference;
    for (int k = 0; k < data.Count; k++) {
        for (int i = 1; i < arrayLength; i++) { //skips time
            difference = data[k][i] - validationData[k][i];
            sum[i-1] += difference * difference;
        }
    }
    double total = 0;
    for (int i = 0; i < sum.Length; i++) {
        if (useWeights) sum[i] *= weights[i];
        total += sum[i];
    }
    return total;
}
```

Die Methode Vergleicht zwei Listen aus Double Arrays und berechnet die jeweilige Differenz zwischen den Werten, quadriert diese und addiert sie zu einer Fehlerquadratsumme. Sie ermöglicht die Gewichtung von einzelnen Teilsummen.
Im genetischen Algorithmus werden die eben beschriebenen Individuuen verwendet, um die im Genom enthaltenen Parameter zu optimieren, indem die Zielfunktion minimiert wird. Der Ablauf des genetischen Algorithmus wurde bereits in Abschnitt 2.4 beschrieben. In Codebeispiel 7 ist der While-Loop in dem der genetische Algorithmus ausgeführt wird erkennbar.

Codebeispiel 7: Methode zur Benutzung des generischen genetischen Algorithmus mit dem Datentyp Double mit Zustandsaktualisierungen.

```
    /// <summary>
    /// Runs the genetic algorithm.
    /// </summary>
    /// <param name="Message"></param>
```

```
public void Run(Action<string> Message) {
    if (Message != null) Message("Start of genetic algorithm!");
    geneticAlgorithmDouble = new GeneticAlgorithm<double>(Generations,
        PopulationSize, genomeSize, genomeLowerBound, genomeUpperBound, random,
        GetRandomDouble,
        FitnessFunction, EliteSize, MutationRate, TournamentSize,
            CrossoverPoint, CrossoverProbability);
    while (geneticAlgorithmDouble.Iteration <
        geneticAlgorithmDouble.Generations) {
        geneticAlgorithmDouble.NewGeneration();
        if (Message != null) Message($"Generation:
            {geneticAlgorithmDouble.Iteration} von
            {geneticAlgorithmDouble.Generations}.");
        if (Message != null) Message($"BestFitness:
            {geneticAlgorithmDouble.BestIndividual.Fitness}.");
        if (Message != null) Message($"Genom: {string.Join(";",
            geneticAlgorithmDouble.BestIndividual.Genome)}.");
        if (geneticAlgorithmDouble.BestIndividual.Fitness == 0) break;
        sigma = 0.99 * sigma;
        this.BestIndividual = geneticAlgorithmDouble.BestIndividual;
    }
    if (Message != null) Message("Genetic algorithm finished!");
}
```

Dabei werden jeweils neue Generationen erzeugt und der Explorationswert $sigma$ aktualisiert bis die maximale Anzahl an Iterationen erreicht ist oder der Fitnesswert beziehungsweise die Fehlerquadratsumme null erreicht. Während der Algorithmus ausgeführt wird werden Zustandsaktualisierungen ausgegeben (aktueller Fitnesswert, Iteration etc.).
In dem gerade beschriebenen Code wird unter anderem eine Methode zur Erzeugung einer neuen Generation benutzt. Diese ist in Codebeispiel 8 zu sehen.

Codebeispiel 8: Methode zur Erzeugung einer neuen Generation aus dem Bereich des generischen genetischen Algorithmus.

```
/// <summary>
/// Creates a new Generation from the old Generation via Selection, Crossover &
    Mutation
/// </summary>
public void NewGeneration() {
    if (Population.Count == 0) return; //failsafe
    if (Iteration == 0) {
        ComputeFitness();
        Population.Sort();
    }
    newPopulation.Clear();
    for (int i = 0; i < Population.Count; i++) {
        if (i < eliteSize) {
            newPopulation.Add(new Individual<T>(Population[i]));
        } else {
            Individual<T> individual1 = TournamentSelection(tournamentSize);
            Individual<T> individual2 = TournamentSelection(tournamentSize);
            Individual<T> newIndividual = new
                Individual<T>(individual1.OnePointCrossover(individual2,
```

```
                crossoverPoint, crossoverProbability));
            newIndividual.Mutation(mutationRate);
            newPopulation.Add(newIndividual);
        }
    }
    //reuse List instead of instatiating new one every time
    List<Individual<T>> temporaryList = Population;
    Population = newPopulation;
    newPopulation = temporaryList; //needed, otherwise Population and
        newPopulation contain the same pointer
    ComputeFitness();
    Population.Sort();
    this.BestIndividual = new Individual<T>(Population[0]); //saves current
        best individual
    Console.WriteLine($"BestFitness = {BestIndividual.Fitness}");
    Iteration++;
}
```

Dabei wird die Tournament Selection bneutzt, um die Individuen auszuwählen zwischen denen der Crossover durchgeführt und die anschließend mutiert werden, um die neue Generation zu erhalten. Ein Teil der Individuen bleibt unverändert als Elite Individuen erhalten. Außerdem wird für die neue Generation die Fitness berechnet.
Der Code für den Crossover ist in Codebeispiel 9 zu sehen.

Codebeispiel 9: Methode für Crossover zwischen zwei Organismen.

```
/// <summary>
/// Performs a one Point Crossover. The point of Crossover and the probability
    can be defined.
/// </summary>
/// <param name="otherIndividual">other Individual partaking in the
    crossover</param>
/// <param name="crossoverPoint">location of the crossover</param>
/// <param name="crossoverProbability">probability of the crossover
    happening</param>
/// <returns>new Individual with possibly new genome</returns>
public Individual<T> OnePointCrossover(Individual<T> otherIndividual, int
    crossoverPoint, double crossoverProbability) {

    Individual<T> newIndividual = new Individual<T>(Genome.Length,
        genomeLowerBound, genomeUpperBound, random, getRandomGene,
        fitnessFunction, initialiseGenome: false);
    if (random.NextDouble() < crossoverProbability) { //Chance of crossover
        Array.Copy(Genome, newIndividual.Genome, crossoverPoint);
        Array.Copy(Genome, crossoverPoint, otherIndividual.Genome,
            crossoverPoint, Genome.Length - crossoverPoint);
    } else Genome.CopyTo(newIndividual.Genome, 0); //no crossover
    return newIndividual;
}
```

Dabei werden mit einer gewissen Wahrscheinlichkeit Teile des Genoms von einem Individuum mit Teilen des Genoms eines anderen Individuums kombiniert. Der Punkt an dem der Crossover

stattfindet kann dabei bestimmt werden.
In Codebeispiel 10 ist die Methode für die Mutation gezeigt.

Codebeispiel 10: Methode für Crossover zwischen zwei Organismen.

```
    /// <summary>
    /// Performs a mutation for the Individual if the random number that is
        generated is bigger then the mutation Rate.
    /// </summary>
    /// <param name="mutationRate">Defines how likely Mutation occurs</param>
    public void Mutation(double mutationRate) {
        for (int i = 0; i < Genome.Length; i++) {
            if (random.NextDouble() < mutationRate) {
                Genome[i] = getRandomGene(genomeUpperBound[i], genomeLowerBound[i],
                    Genome[i]);
            }
        }
    }
```

Diese erzeugt mit einer gewissen Wahrscheinlichkeit ein neues mutiertes Individuum. Dies geschieht mit Hilfe der Methode zur Erzeugung eines zufälligen Gens (Codebeispiel 4), da je nach Datentyp eine unterschiedliche Methode benötigt wird.
Der genetische Algorithmus ist als globaler Suchalgorithmus nicht besonders genau was lokale Minima angeht. Daher werden die besten Individuuen aus dem genetischen Algorithmus mit dem Simplex Algorithmus nach Nelder-Mead optimiert. Der Ablauf des Algorithmus ist bereits in Abschnitt 2.5 genauer beschrieben worden. In Codebeispiel 11 ist der dazugehörige Code zu sehen.

Codebeispiel 11: Simplex Algorithmus nach Nelder-Mead.

```
        /// <summary>
        /// Performs a variable simplex algorithm according to Nelder-Mead (also
            Downhill Simplex, Nelder Mead method). It is an iterative algorithm for
            linear and non-linear Optimization problems.
        /// </summary>
        /// <param name="targetFunction">Target function to evaluate the
            points</param>
        /// <param name="dimensions">Dimensions of the array for the target
            function</param>
...
        /// <returns>optimized double[]</returns>
        public static double[] VariableSimplex(Func<double[], double>
            targetFunction, int dimensions, int MaxIterations = 1000, double
            distance = 0.1, bool convergence = true, double[] initialValues = null)
            {
...
            double[][] simplex = new double[dimensions + 1][];
            int counter = -1;
            // generate initial arrays dim + 1
...
            counter = 0;
            // Loop until the simplex is convergeing or the Maximum iterations are
                reached
```

```
...
                // Sort
                Array.Sort(functionValues, indices); //minimize
...
                // Find geometric center of the simplex excluding the vertex/edge
                    with highest function value.
...
                // Check for convergence
                if (convergence) {
                    if (Math.Abs(targetFunction(geometricCenter) -
                        functionValues[0]) < 1e-13) break;
                }

                //Reflect the Simplex
...
                // Expand the Simplex
...
                // Contract the Simplex
...
                //Shrink the Simplex
...
                counter++;
            }
            return simplex[0];
        }
```

Die Methode benötigt als Eingabeparameter eine Zielfunktion (targetFunction) und die Anzahl an Parametern die optimiert werden müssen (dimensions). Zusätzlich kann optional definiert werden, ob der Simplex nach einer bestimmten Zahl an Iterationen oder wenn er konvergiert stoppt. Eine weitere wichtige optionale Angabe sind die initialen Parameter (initialValues), da hier das Genom eines Individuums aus dem genetischen Algorithmus eingesetzt werden kann. Der generelle Ablauf des Simplex kann anhand der Code Kommentare betrachtet werden. Als Ausgabe gibt der Simplex ein optimiertes Double Array zurück, welches in diesem Fall die Prozessparameter enthält.

3.2 Ergebnisse und Diskussion

Es wurden drei verschiedene Durchläufe des Programms durchgeführt. Im ersten Durchlauf wurde der genetische Algorithmus ohne Einschränkung des Suchraums verwendet. Im zweiten Durchlauf wurde der Suchraum mit biologisch Sinnvollen Ober- und Untergrenzen eingeschränkt und im letzten Durchlauf wurde die Gewichtung der Zielfunktion angepasst. In Tabelle 1 sind die Parameter für die Durchläufe des genetischen Algorithmus zu erkennen.
In Tabelle 2 ist der Validierungsdatensatz gezeigt.

Tabelle 1: Parameter für den genetischen Algorithmus.

Parameter	Wert
Anzahl Individuuen	400
Iterationszahl	100
Anzahl Elite Individuuen	16
Anzahl Individuuen Tournament Selection	8
Wahrscheinlichkeit für Mutation	0.8
Wahrscheinlichkeit für Crossover	0.5
Explorationswert	20

Tabelle 2: Validationsdatensatz für das Wachstum von *P. vulgaris* (C_x) auf 2 Substraten in einer Batch Kultivierung. Die Substrate sind Glukose (C_{S2}) und Zitronensäure (C_{S1}).

t [h]	S1 [kg/m³]	S2 [kg/m³]	X [kg/m³]
0	0.5	1.6	0.001
2.5	0.48	1.7	0.005
5	0.45	1.8	0.02
7.5	0.39	1.8	0.025
10	0.32	1.8	0.05
12.5	0.2	1.7	0.08
15	0.05	1.6	0.1
17.5	0	1.5	0.15
20	0	1.5	0.2
22.5	0	1.4	0.24
25	0	1.1	0.3
27.5	0	0.9	0.36
30	0	0.7	0.5
32.5	0	0.5	0.6
35	0	0.3	0.68
37.5	0	0.1	0.75
40	0	0	0.82

3.2.1 Modellierung des Bioprozesses ohne Suchraumeinschränkung und ohne Gewichtung der Zielfunktion bei der Parameteroptimierung.

Nach der ersten Parameteroptimierung durch den genetischen Algorithmus und den Simplex Algorithmus nach Nelder-Mead wurden die in Tabelle 3 gezeigten Parameter generiert. Dabei wurde weder der Suchraum eingeschränkt, noch wurde die Zielfunktion gewichtet. Die erreichte minimale Fehlerquadratsumme beträgt 3.467.

Tabelle 3: Parameteroptimierung ohne Suchraumeinschränkung und ohne Gewichtung der Zielfunktion.

Parameter	μ_{max1}	μ_{max2}	KS_1	KS_2	Y_{XS1}	Y_{XS2}
Wert	387.177	798.234	8842.036	6877.406	2585.814	0.329

Es ist zu erkennen, das die einzelnen Parameter relativ hohe Werte besitzen. Weiterhin resultieren sie in einer relativ hohen Fehlerquadratsumme.
In Abb. 3 ist das Ergebnis des mit den Parametern durchgeführten Bioprozessmodells dargestellt.

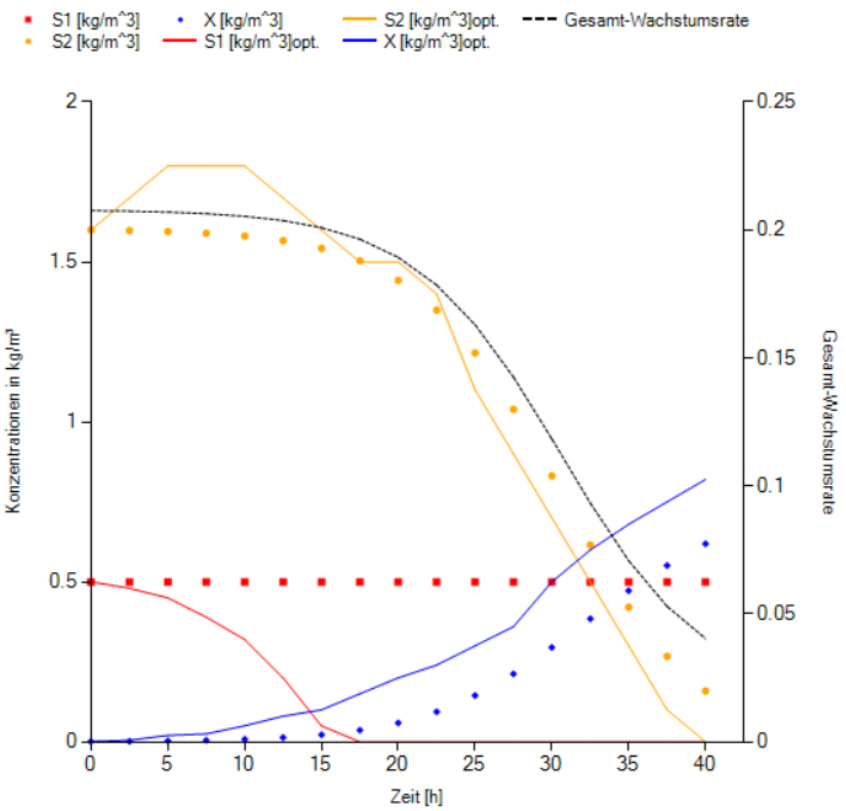

Abbildung 3: Modellierung des Bioprozesses ohne Suchraumeinschränkung und ohne Gewichtung der Zielfunktion bei der Parameteroptimierung.

Hier zeigt sich, dass die Bioprozessmodellierung mit den Parametern aus Tabelle 3 den Prozess noch nicht besonders gut beschreiben. Während der Verlauf von C_{S2} und C_x noch einigermaßen im Bereich der Validationsdaten liegt, sind die simulierten Werte für C_{S1} konstant. Es besteht, bis auf die Anfangskonzentration, fast kein Zusammenhang zwischen den simulierten Werten und den Validationsdaten für C_{S1}.
In Tabelle 4 sind die zu der Bioprozessmodellierung zugehörigen Daten gezeigt. Auch hier ist zu erkennen, dass die in Abb. 3 gezeigten simulierten Werte für C_{S1} konstant sind. Alles in allem entspricht der mit diesen Parametern durchgeführte Bioprozess eher einem Wachstumsprozess bei dem nur ein Substrat verwertet wird, während ein anderes zwar dauerhaft vorhanden ist, aber nie verbraucht wird. Dieses Verhalten resultiert aus der konstanten Konzentration des ersten Substrates C_{S1}, welches aufgrund des extrem hohen Ausbeutekoeffizienten Y_{XS1} dauerhaft zu der Wachstumsrate beiträgt und dabei nur minimal verwertet wird.
Würde dieses Verhalten auf reale Substrate übertragen, die zum Beispiel Energie liefern, würde Substrat C_{S1} eine extrem hohe Energiedichte besitzen und könnte so effizient verwertet werden, sodass von dem Mikroorganismus nur extrem geringe Mengen davon verwertet werden müssten. Alles in allem weicht das mit den optimierten Parametern durchgeführte Bioprozessmodell stark von dem realen Modell ab.

Tabelle 4: Daten aus der Modellierung des Bioprozesses ohne Suchraumeinschränkung und ohne Gewichtung der Zielfunktion.

Zeit[h]	S1 [kg/m³]	S2 [kg/m³]	X [kg/m³]	Gesamt Wachstumsrate [kg/m³]
0	0.50	1.60	0.00	0.21
2.5	0.50	1.60	0.00	0.21
5	0.50	1.60	0.00	0.21
7.5	0.50	1.59	0.00	0.21
10	0.50	1.58	0.01	0.21
12.5	0.50	1.57	0.01	0.20
15	0.50	1.54	0.02	0.20
17.5	0.50	1.50	0.04	0.20
20	0.50	1.44	0.06	0.19
22.5	0.50	1.35	0.09	0.18
25	0.50	1.22	0.14	0.16
27.5	0.50	1.04	0.21	0.14
30	0.50	0.83	0.29	0.12
32.5	0.50	0.62	0.38	0.09
35	0.50	0.42	0.47	0.07
37.5	0.50	0.27	0.55	0.05
40	0.50	0.16	0.62	0.04

3.2.2 Modellierung des Bioprozesses mit Suchraumeinschränkung und ohne Gewichtung der Zielfunktion bei der Parameteroptimierung.

Nach der zweiten Parameteroptimierung durch den genetischen Algorithmus und den Simplex Algorithmus nach Nelder-Mead wurden die in Tabelle 5 gezeigten Parameter generiert.

Tabelle 5: Parameteroptimierung mit Suchraumeinschränkung und ohne Gewichtung der Zielfunktion.

Parameter	μ_{max1}	μ_{max2}	KS_1	KS_2	Y_{XS1}	Y_{XS2}
Wert	0.435	0.153	0.529	0.452	0.099	0.485
Minimum	0.15	0.15	0.01	0.01	0.10	0.10
Maximum	0.50	0.50	5.00	5.00	0.80	0.80

Während der Optimierung wurde der Suchraum durch die in Tabelle 5 gezeigten Minima und Maxima eingeschränkt. Die Zielfunktion wurde nicht gewichtet. Die erreichte minimale Fehlerquadratsumme beträgt hier 0.271.
In Abb. 4 ist die aus den Parametern resultierende Bioprozessmodellierung dargestellt.

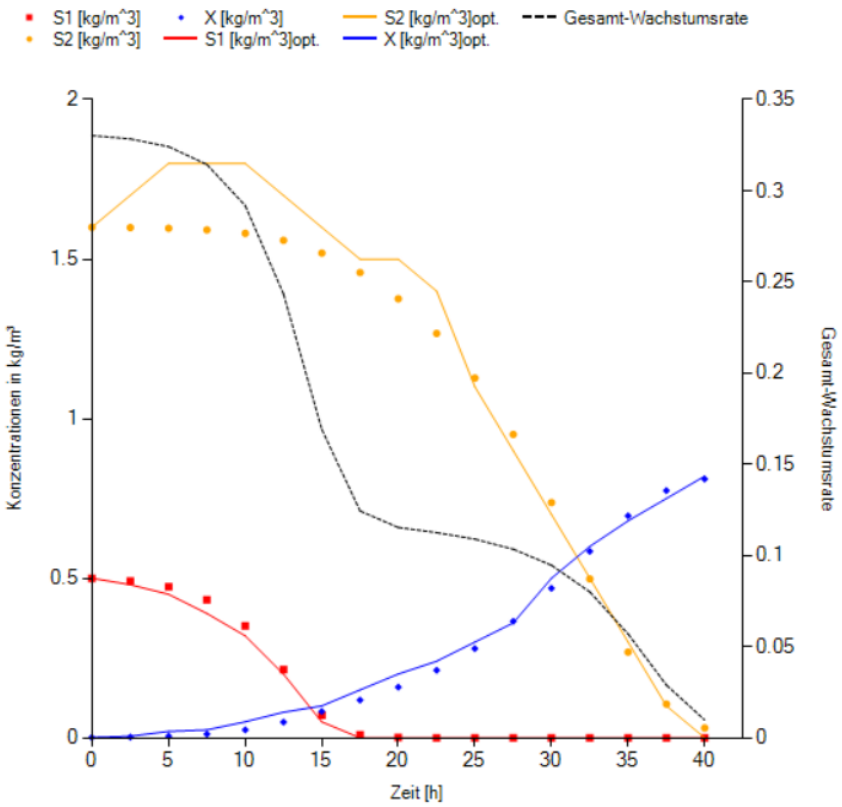

Abbildung 4: Modellierung des Bioprozesses mit Suchraumeinschränkung und ohne Gewichtung der Zielfunktion bei der Parameteroptimierung.

Es zeigt sich, dass die Daten aus der Bioprozessmodellierung mit den Parametern aus Tabelle 5 den Prozess – im Vergleich zur ersten Parameteroptimierung – bereits wesentlich besser beschreiben. Die simulierten Verläufe von C_{S1}, C_{S2} und C_x liegen alle bereits relativ dicht bei den Validationsdaten. Weiterhin zeigt der Verlauf der Gesamt-Wachstumsrate den zu erwartenden Verlauf. Die Gesamt-Wachstumsrate ist am Anfang sehr hoch, da hier beide Substrate gleichzeitig verwertet werden und so zum Zellwachstum beitragen. Nachdem das erste Substrat nahezu verbraucht ist, sinkt die Gesamt-Wachstumsrate stark ab und bleibt, solange die Konzentration des zweiten Substrates noch verhältnismäßig hoch, ist auf einem relativ konstanten Niveau. Nachdem die Konzentration des zweiten Substrates stärker absinkt, beginnt auch die Gesamt-Wachstumsrate stärker zu sinken. Als letztes geht sie nahezu gegen null, nachdem die Konzentration des zweiten Substrates gegen null geht.
In Tabelle 6 sind die Daten zu der Bioprozessmodellierung gezeigt. Wie in Abb. 4 bereits gezeigt sinkt mit abfallenden Werten für Substrat C_{S1} und Substrat C_{S2} auch die Wachstumsrate. Weiterhin ist zu erkennen, dass die Zeitpunkte an denen die Substrate auf null fallen ungefähr mit den Zeitpunkten im Validationsdatensatz übereinstimmen. Der Endwert für die Biomassekonzentration C_x ist zudem nahezu identisch.

Tabelle 6: Daten aus der Modellierung des Bioprozesses mit Suchraumeinschränkung und ohne Gewichtung der Zielfunktion.

Zeit[h]	S1 [kg/m³]	S2 [kg/m³]	X [kg/m³]	Gesamt Wachstumsrate [kg/m³]
0	0.50	1.60	0.00	0.33
2.5	0.49	1.60	0.00	0.33
5	0.47	1.60	0.01	0.32
7.5	0.43	1.59	0.01	0.31
10	0.35	1.58	0.02	0.29
12.5	0.21	1.56	0.05	0.24
15	0.07	1.52	0.08	0.17
17.5	0.01	1.46	0.12	0.12
20	0	1.38	0.16	0.12
22.5	0	1.27	0.21	0.11
25	0	1.13	0.28	0.11
27.5	0	0.95	0.37	0.10
30	0	0.74	0.47	0.09
32.5	0	0.50	0.58	0.08
35	0	0.27	0.70	0.06
37.5	0	0.11	0.77	0.03
40	0	0.03	0.81	0.01

3.2.3 Modellierung des Bioprozesses mit Suchraumeinschränkung und Gewichtung der Zielfunktion bei der Parameteroptimierung.

Nach der dritten Parameteroptimierung durch den genetischen Algorithmus und den Simplex Algorithmus nach Nelder-Mead wurden die in Tabelle 7 gezeigten Parameter generiert.

Tabelle 7: Parameteroptimierung mit Suchraumeinschränkung und mit Gewichtung der Zielfunktion.

Parameter	μ_{max1}	μ_{max2}	KS_1	KS_2	Y_{XS1}	Y_{XS2}
Wert	0.398	0.160	0.476	0.483	0.090	0.490
Minimum	0.15	0.15	0.01	0.01	0.10	0.10
Maximum	0.50	0.50	5.00	5.00	0.80	0.80

Während der Optimierung wurde der Suchraum durch die in Tabelle 7 gezeigten Minima und Maxima eingeschränkt und die Zielfunktion wurde gewichtet. Die Gewichtung ist in Tabelle 8 gezeigt.

Tabelle 8: Parametergewichtung bei der Fehlerberechnung.

Parameter	C_{S1}	C_{S2}	C_x
Gewichtung	1	1.1	1.5

Die erreichte minimale Fehlerquadratsumme beträgt hier 0.239. Diese ist jedoch nur teilweise mit den vorherigen Fehlerquadratsummen vergleichbar. Im Gegensatz zu den anderen Fehlerquadratsummen wurden hier die Differenzen zwischen den simulierten Daten und den Validationsdaten für das Substrat C_{S2} und die Biomassekonzentration C_x stärker gewichtet.
In Abb. 5 ist die Daten der aus den Parametern resultierenden Bioprozessmodellierung dargestellt.

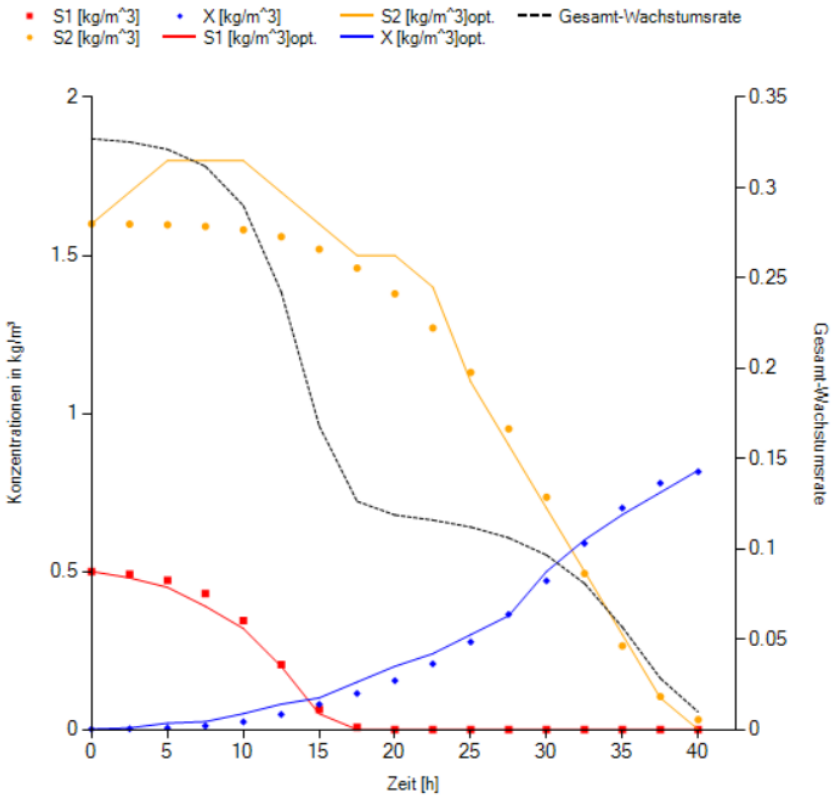

Abbildung 5: Modellierung des Bioprozesses mit Suchraumeinschränkung und mit Gewichtung der Zielfunktion bei der Parameteroptimierung.

Es zeigt sich, dass die Daten aus der Bioprozessmodellierung mit den Parametern aus Tabelle 7 den Prozess – im Vergleich zur zweiten Parameteroptimierung – noch ein wenig besser beschreiben. Die Differenz zwischen den beiden Modellen ist jedoch bereits wesentlich kleiner Die simulierten Verläufe von C_{S1}, C_{S2} und C_x liegen ähnlich dicht bei den Validationsdaten, wie im zweiten Durchlauf. Auch Verlauf der Gesamt-Wachstumsrate ist ähnlich. Die etwas stärkere

Gewichtungen von dem Substrat C_{S2} und der Biomassekonzentration C_x führen dazu, dass diese hier etwas genauer abgebildet werden, während der Verlauf von Substrat C_{S1} nur unwesentlich mehr abweicht, als im zweiten Durchlauf. Alles in allem ist der Unterschied jedoch minimal.
In Tabelle 9 sind die Daten zu der Bioprozessmodellierung gezeigt. Auch hier zeigen sich Par-

Tabelle 9: Daten aus der Modellierung des Bioprozesses mit Suchraumeinschränkung und mit Gewichtung der Zielfunktion.

Zeit[h]	S1 [kg/m³]	S2 [kg/m³]	X [kg/m³]	Gesamt Wachstumsrate [kg/m³]
0	0.50	1.60	0.00	0.33
2.5	0.49	1.60	0.00	0.33
5	0.47	1.60	0.01	0.32
7.5	0.43	1.59	0.01	0.31
10	0.34	1.58	0.02	0.29
12.5	0.21	1.56	0.05	0.24
15	0.06	1.52	0.08	0.17
17.5	0.01	1.46	0.11	0.13
20	0	1.38	0.15	0.12
22.5	0	1.27	0.21	0.12
25	0	1.13	0.28	0.11
27.5	0	0.95	0.36	0.11
30	0	0.74	0.47	0.10
32.5	0	0.49	0.59	0.08
35	0	0.26	0.70	0.06
37.5	0	0.10	0.78	0.03
40	0	0.03	0.82	0.01

allelen zum zweiten Durchlauf. Es wieder ist zu erkennen, dass die Zeitpunkte an denen die Substrate auf null fallen ungefähr mit den Zeitpunkten im Validationsdatensatz übereinstimmen und Endwert für die Biomassekonzentration C_x ist nahezu identisch.

3.2.4 Analyse der Parameterwerte

Der zweite und der dritte Durchlauf haben als Resultat relativ ähnliche Parameterwerte (Tabellen 5 und 7). Im folgenden wir vor allem auf die Parameter aus dem dritten Durchlauf (Tabelle 7 Bezug genommen. Es zeigt sich, dass der Ausbeutekoeffizient von dem Substrat C_{S1} (Zitronensäure) wesentlich niedriger ist, als der von Substrat C_{S2} (Glucose). Das bedeutet, dass aus Zitronensäure im Verhältnis weniger Biomasse resultiert als aus Glucose. Dieser Zusammenhang ist biologisch sinnvoll, da leichter mehr Energie aus Glucose gewonnen werden kann. Die Monod-Parameter für Zitronensäure und Glucose sind ähnlich. Das heißt, bei beiden ist die halbmaximale Wachstumsrate ungefähr bei einer Konzentration von circa 0.5 kg/m³ erreicht.

Auch dieser Wert liegt in einem biologisch sinnvollen Bereich. Die spezifische Wachstumsrate für Zitronensäure beträgt 0.398 und die für Glucose 0.160. Daraus ergibt sich eine maximale Gesamt-Wachstumsrate von circa 0.46. Damit ist der Mikroorganismus in der Lage nur auf Zitronensäure mit einer höheren Wachstumsrate zu wachsen, als nur auf Glucose. Alles in allem liegen damit alle Parameter und die daraus resultierenden Daten in einem sinnvollen Bereich. Aus der Bioprozessmodellierung können daher mit einer hohen Wahrscheinlichkeit brauchbare Daten gewonnen werden.

Literatur

(1) Roeva, O. in *Large-Scale Scientific Computing*; Springer Berlin Heidelberg: 2008, S. 601–608.

(2) Strigul, N.; Dette, H.; Melas, V. B. *Environmental Modelling and Software* **2009**, *24*, 1019–1026.

(3) Carrillo-Ureta, G.; Roberts, P.; Becerra, V. in *Proceeding of the 2001 IEEE International Symposium on Intelligent Control (ISIC 01) (Cat. No.01CH37206)*, IEEE.

(4) *Bioprozesstechnik*; Chmiel, H., Hrsg.; Spektrum Akademischer Verlag: 2011.

(5) Monod, J. *Annual Review of Microbiology* **1949**, *3*, 371–394.

(6) Moser, A. in *Bioprozeßtechnik*; Springer Vienna: 1981, S. 128–168.

(7) Muloiwa, M.; Nyende-Byakika, S.; Dinka, M. *South African Journal of Chemical Engineering* **2020**, *33*, 141–150.

(8) Steinbüchel, A.; Oppermann-Sanio, F. B.; Ewering, C.; Pötter, M., *Mikrobiologisches Praktikum*; Springer Berlin Heidelberg: 2013.

(9) Perrin, E.; Ghini, V.; Giovannini, M.; Patti, F. D.; Cardazzo, B.; Carraro, L.; Fagorzi, C.; Turano, P.; Fani, R.; Fondi, M. *Nature Communications* **2020**, *11*, DOI: 10.1038/s41467-020-16872-8.

(10) Loomis, W. F.; Magasanik, B. *Journal of Bacteriology* **1967**, *93*, 1397–1401.

(11) Narang, A.; Pilyugin, S. S. *Journal of Theoretical Biology* **2007**, *244*, 326–348.

(12) Kallenrode, M.-B., *Rechenmethoden der Physik*; Springer-Verlag GmbH: 2006; 386 S.

(13) Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P., *Numerical Recipes 3rd Edition, The Art of Scientific Computing*; Cambridge University Press: 2007, S. 1256.

(14) Kühnel, A., *C# 8 mit Visual Studio 2012*; Rheinwerk Verlag GmbH: 2012; 1478 S.

(15) Hernández García, R., *ANSI C Grundlagen der Programmierung*; HERDT-Verlag: 2015; 1478 S.

(16) Heistermann, J., *Genetische Algorithmen*; Teubner Verlag: 1994.

(17) Sivanandam, S. N., *Introduction to genetic algorithms*; Springer: 2007, S. 442.

(18) Pohlheim, H., *Evolutionäre Algorithmen*; Springer Berlin Heidelberg: 2000.

(19) Audet, C.; Hare, W., *Derivative-Free and Blackbox Optimization*; Springer International Publishing: 2017.

(20) Nelder, J. A.; Mead, R. *The Computer Journal* **1965**, *7*, 308–313.