Simulation einer stochastischen Populationsdynamik

Boris Prochnau

Geboren am 22. Dezember 1989 in Tartu

4. August 2014

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Anton Bovier

Institut für Angewandte Mathematik

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Inhaltsverzeichnis

1 Einleitung

Diese Bachelorarbeit behandelt die Entwicklung eines Programms zur Simulation einer Populationsdynamik. Dazu wird erst ein grundlegendes Modell vorgestellt, welches den zu simulierenden BPDL-Prozess beschreibt. Darüber hinaus wird das Programm auch in der Lage sein die Approximation von Grenzwertprozessen zu simulieren, die aus dem BPDL-Prozess gewonnen werden. Dazu zählt auch der sogenannte "TSS-Prozess". Die Simulation einer Approximation des "TSS-Prozesses" ist besonders interessant und wird in Kapitel 3 eingeführt und in Kapitel 7 genauer beschrieben. Alle Simulationen sollen auf demselben Modell basieren.
Jedes Lebewesen einer Population (z.B. Pflanzen, Zellen, Tiere) wird durch ein Merkmal beschrieben. Diese Merkmale erzeugen Dynamik in der Population, in dem sie z.B. Größe, Essensaufnahme, Fortpflanzungsrate oder Konkurrenz zu anderen Merkmalen angeben.
In unserem Modell besteht ein Merkmal aus einer asexuellen Fortpflanzungsrate und zwei Todesraten. Die Todesraten bestehen aus einer natürlichen Todesrate und einer Todesrate, die durch Wettbewerb zu jedem anderen Lebewesen entsteht. Man bemerkt schon, dass ein Individuum in diesem Modell nur sterben, töten (durch Wettbewerb) oder sich fortpflanzen kann. Da die wettbewerblichen Todesraten der Individuen von der Populationsgröße abhängen, ist die resultierende Dynamik nicht trivial und ihr Studium von besonderem Interesse.
Des Weiteren umfasst das Modell eine Mutationswahrscheinlichkeit welche sich nicht auf bereits in der Population lebende Wesen, sondern auf neugeborene Individuen bezieht.
Da die Entwicklung der Population, und nicht die der Individuen, das Ziel unserer Simulation ist (im Gegensatz zu [1]), simulieren wir viele Individuen auf wenige Merkmale verteilt. Deshalb wird man im simulierten Prozess zwar Tode und Geburten in Merkmalen verfolgen können, aber nicht welches Individuum dieses Ereignis auslöst. Der Übergang zu dieser Sichtweise wird näher im 2. Kapitel beschrieben.
Hauptsächlich wird in dieser Arbeit beschrieben, wie sich dieses Modell und seine Eigenschaften auftrennen oder zusammenfassen lassen um möglichst effizient und sicher ein Programm zur Simulation eines entsprechenden Prozesses zu implementieren. Dabei wird oft auf moderne Methoden zurückgegriffen, die eine möglichst unabhängige Strukturierung der Schritte während eines Ablaufs voraussetzen.
Schließlich wird auch beschrieben, was eine Simulation leisten kann, sollte und was diese Simulation tatsächlich tut. Z.B. sollte die graphische Darstellung des Prozesses die Möglichkeit bieten, beobachten zu können, ob sich ein Merkmal unter anderen durchsetzten kann, es einen stabilen Zustand annimmt oder dem Tod entgegen strebt.

2 Modell

Das verwendete Model wurde in [2, 3, 4] eingeführt. Bei asexueller Vermehrung nutzt das Modell die drei grundlegenden Mechanismen von Darwins Evolutionslehre: Vererbung, Variation (Mutationen) und Selektion durch Wettbewerb, um eine Menge von Merkmalen für Individuen zu beschreiben. Diese bestimmen die Fähigkeit des Individuums zu überleben und sich fortzupflanzen. Der daraus resultierende zeitstetige Sprung-Prozess wird BPDL-Prozess (nach Bolker, Pacala, Dieckmann und Law) genannt.
Ziel ist es, zwei spezielle BPDL-Grenzwert-Prozesse zu simulieren.

2.1 Grundlagen

Sei X der endliche diskrete Raum der Merkmale. Jedes Individuum hat genau ein solches Merkmal $x \in X$, wodurch es vollständig charakterisiert ist. Es ist hilfreich sich X als Indexmenge $X = \{1, \ldots, n\}$ vorzustellen, die abgezählte Merkmale enthält. Das entspricht auch der Interpretation von X aus Sicht der Simulation. Der Übersicht halber werden Elemente aus X jedoch mit $x, y \in X$ angesprochen. Ein allgemeineres Modell findet sich in [5].
Für jedes Individuum mit Merkmal $x \in X$ gilt:

- Jedes Individuum kann sich nur asexuell fortpflanzen oder sterben.
- Fortpflanzungs- und Todeszeitpunkte können durch sogenannte exponentielle Uhren beschrieben werden (wie in [1, S. 3]). Diese Uhren haben exponentiell verteilte Weckzeiten. Durch die Gedächtnislosigkeit der Exponentialverteilung und wegen des Wettbewerbs können und müssen alle Uhren nach dem ersten Klingeln neu gestellt werden. Durch den Einfluss des Wettbewerbs ist jede Todesrate abhängig von der Anzahl an Konkurrenten, die durch das zuerst eintretende Ereignis beeinflusst wird.
- Bei einer Fortpflanzung kann eine Mutation auftreten d.h. die Fortpflanzung des Individuums mit Merkmal $x \in X$ kann in der Geburt eines Individuums mit Merkmal $y \in X$ resultieren. Die Häufigkeit dieser Ereignisse wird durch die Mutationswahrscheinlichkeit beschrieben.

Später wird deutlich, dass die Zurückstellbarkeit der Uhren entscheidend ist, um die Sichtweise von der Ebene des Individuums auf die der gesamten Population zu heben.
Diese Todes- und Fortpflanzungsereignisse eines Individuums haben feste Raten, die das dazugehörige Merkmal beschreiben.

$b(x)$: Ist die Geburtenrate durch ein Individuum mit Merkmal x.

$d(x)$: Ist die natürliche Todesrate. Im Folgenden wird stets vorausgesetzt, dass ein Merkmal überlebensfähig ist. Also $b(x) - d(x) > 0$.

$c(x, y)$: Ist die Todesrate durch Wettbewerb, die ein Individuum y auf x ausübt. Diese Interpretation orientiert sich an [5], während u.a. in [6] ein symmetrischer Wettbewerbskern verwendet wird. Im Gegensatz zu vorher kann der Wettbewerb ein konkurrierendes Merkmal zum Aussterben zwingen.

μ: Ist die Mutationswahrscheinlichkeit "auf die Nachbarn" mit je $\frac{\mu}{2}$ pro Nachbar.

Alle erwähnten Raten sind wie in [1] endlich und positiv.
Schließlich lassen sich durch das Superpositionsprinzip der Exponentialverteilung die beiden Todesraten zu einer gemeinsamen Todesrate zusammenfassen oder die arteigene Geburtenrate beschreiben.

$b(x) \cdot (1 - \mu)$: Ist die arteigene Geburtenrate eines Individuums mit Merkmal x, also mutationsfreie Geburten.

$d(x) + \sum_{i=1}^{N_t} c(x, x_i)$: Ist die gesamte Todesrate eines Individuums mit Merkmal x (mit $N_t \hat{=}$ #Individuen zur Zeit t mit Merkmal x und x_i das Merkmal des i-ten Individuums).

$d(x) + \sum_{y \in X} c(x, y) \cdot n_t(y)$: Ist auch die gesamte Todesrate, diesmal jedoch über die Merkmale summiert, mit $n_t(x) \hat{=}$#Individuen zur Zeit t mit Merkmal x.

Im Unterschied zu [6] sind wir an der Entwicklung einer großen Population mit wenigen Merkmalen interessiert. Deswegen ist es unpraktisch weiterhin die Raten jedes Individuums zu berechnen.
Die letzte Darstellung der Todesrate ist z.B. praktischer für die Betrachtung der Population durch den Fokus auf die Merkmale. Ähnlich können weitere Ereignisse zusammengefasst werden, so dass man z.B. eine Todesrate und eine arteigene Geburtenrate der Merkmale erstellen kann:

- Die **Fortpflanzungsrate** des Merkmals x:

$$\tilde{B}(x) = b(x) \cdot n_t(x)$$

  Diese beschreibt die Rate, mit der Fortpflanzungen innerhalb des Merkmals x stattfinden (nicht die Geburten innerhalb x!).

- Die arteigene Geburtenrate (**Wachstumsrate**) des Merkmals x ist von besonderem Interesse und ist das, womit wir folgend hauptsächlich

arbeiten werden:

$$\begin{aligned} B(x) = &(1-\mu)\cdot b(x)\cdot n_t(x) \\ &+ \frac{\mu}{2}\cdot \underbrace{b(x+1)\cdot n_t(x+1)}_{\text{Mutation von rechts}} \cdot \mathbb{1}_{x<n} \\ &+ \frac{\mu}{2}\cdot \underbrace{b(x-1)\cdot n_t(x-1)}_{\text{Mutation von links}} \cdot \mathbb{1}_{x>1} \end{aligned}$$

Hierbei ist zu beachten, dass: $\sum_{x\in X} B(x) = \sum_{x\in X} \tilde{B}(x)$, da die Mutationen von rechts und links per Teleskopsumme den Faktor $(1-\mu)$ ausgleichen.

- Die **Todesrate** des Merkmals x:

$$D(x) = \underbrace{n_t(x)\cdot d(x)}_{\text{intrinsische Todesrate}} \underbrace{+ n_t(x)\cdot \sum_{y=1}^{n} c(x,y)\cdot n_t(y)}_{\text{wettbewerbliche Todesrate}}$$

Das entspricht zwei wesentlichen exponentiellen Uhren pro Merkmal: eine für Tod und eine für Geburt innerhalb des Merkmals.
Für die Simulation ist eine Gesamtrate für das Eintreten eines Ereignisses praktischer. Auf diese Weise wird nur auf das Eintreffen einer Uhr gewartet.

- Ereignisrate des Merkmals x (Trait Rate):

$$TR(x) = B(x) + D(x)$$

- Totale Ereignisrate (Total Event Rate):

$$TER = \sum_{x\in X} TR(x)$$

Mit der Totalen Ereignisrate gibt es eine Rate, die es erlaubt eine Zufallsvariable für das Eintreffen einer Variable zu ziehen. Anschließend ist es nur noch erforderlich (mit der Ziehung zweier weiterer Zufallsvairablen) festzustellen, welchem Merkmal welches Ereignis zukommt. Das Zusammenfassen der Raten vereinfacht es dem Programm, spätere Auswertungen und Funktionen bereitzustellen. So lässt sich z.B. aus der Geburtenrate (Wachstumsrate) eines ausgestorbenen Merkmals die Mutationsrate ablesen, ohne dass weitere Berechnungen gemacht werden müssen.

2.2 BPDL-Prozess

Eine auf diesem Modell basierende Population wird durch folgendes Punktmaß beschrieben:

$$\nu_t = \sum_{i=1}^{N_t} \delta_{x_i}, \quad x_i \hat{=} \text{Das Merkmal des i-ten Individuums}$$

Es bildet Merkmale auf die Anzahl ihrer Repräsentanten ab.
Mit Zeitpfaden ist ν_t ein stochastischer Prozess, genauer ein Markov Sprung Prozess auf dem Maßraum:

$$\nu_t \in M(X) = \left\{ \sum_{i=1}^{m} \delta_{x_i}, m \in \mathbb{N}, x_1, \ldots, x_n \in X \right\}$$

Man erkennt leicht die Sprungeigenschaft:

$$\int_X 1 \ \nu_t(dx) = N_t \text{ und } \int_X \mathbb{1}_y(x) \ \nu_t(dx) = n_t(y)$$

Normalerweise gehört zum Model des BPDL-Prozesses, dass die Mutationen auf einem beliebigen Merkmal (nicht nur den Nachbarn) landen können und der Raum der Merkmale nicht unbedingt diskret sein muss. Eine Mutationswahrscheinlichkeit hängt in diesem Fall vom Merkmal ab, also $\mu(x)$. Der Mutant hat folglich Merkmal $x + h$, wobei h eine zentrierte Zufallsvariable mit Dichte $m(x, dh)$ auf $(X - x)$ ist. Für einen solchen Prozess wäre der Generator definiert als:

$$\begin{aligned} L(\phi(\nu)) = & \int_X b(x)(1-\mu)[\phi(\nu + \delta_x) - \phi(\nu)]\nu(dx) \\ & + \int_X \int_{\mathbb{R}^d} b(x) \cdot \mu[\phi(\nu + \delta_{x+z}) - \phi(\nu)]m(x, dz)\nu(dx) \\ & + \int_X d(x)[\phi(\nu - \delta_x) - \phi(\nu)]\nu(dx) \\ & + \int_X \left(\int_X c(x, y)\nu(dy) \right) [\phi(\nu - \delta_x) - \phi(\nu)]\nu(dx) \end{aligned}$$

mit $\phi : M \to \mathbb{R}$.
Dieser beschreibt die erwartete Änderung von ν zur Zeit t. Man erkennt, dass der Generator unabhängig von t ist, da er nur mit den zeitunabhängigen Parametern b, d, c, μ konstruiert wurde. Natürlich ist der verwendete Prozess ν_t abhängig von t, weshalb man $\frac{d}{dt}\mathbb{E}\phi(\nu_t) = L\phi(\nu_t)$ schreiben kann.
Mit unserem diskreten Raum X und der konstanten Mutationswahrscheinlichkeit zu Nachbarn vereinfacht sich der Generator. Der für unser Modell

angepasste Generator hat somit folgende Form:

$$\begin{aligned} L(\phi(\nu)) = &\sum_{x \in X} b(x)(1-\mu)[\phi(\nu + \delta_x) - \phi(\nu)] \cdot n(x) \\ &+ \sum_{y \sim x} b(x) \cdot \frac{\mu}{2} \cdot [\phi(\nu + \delta_y) - \phi(\nu)] \cdot n(x) \\ &+ \sum_{x \in X} \left(d(x) + \sum_{y \in X} c(x,y) \cdot n(y) \right) [\phi(\nu - \delta_x) - \phi(\nu)] \cdot n(x) \end{aligned} \tag{1}$$

mit $\phi : M \to \mathbb{R}$.

3 Eigenschaften des BPDL-Prozesses

In diesem Kapitel werden Eigenschaften des Prozesses näher untersucht, die später bei der Simulation sichtbar sein sollen. Zunächst wird dabei die Normalisierung eingeführt, die es erleichtert Aussagen über das erwartete Verhalten des Prozesses bzw. der Population zu machen.

3.1 Normalisierung des BPDL-Prozesses

Wie schon zuvor erwähnt ist es für uns wichtig, die Tode und Geburten nicht auf der Ebene des Individuums, sondern der gesamten Population zu betrachten. Dazu wird die LPA (Large Population Approximation) Normalisierung aus [1] eingeführt.
Hierfür wird der Prozess mit einem Parameter K skaliert und es ergibt sich eine neue Zufallsvariable:

$$\nu_t^K := \frac{1}{K} \nu_t$$

Um für ν_t^K dasselbe Verhalten wie für ν_t zu erhalten, müssen einige Anpassungen vorgenommen werden.
Zunächst wird die Anfangsgröße n_0^K der Population proportional zu K gewählt. Die Raten für Geburten und natürliche Tode der Individuen bleiben unverändert. Da die Populationsgröße jedoch quadratisch in die Wettbewerbsrate einfließt, sollte $c_K(x,y) = \frac{c(x,y)}{K}$ gelten, da sonst die K-fach erhöhte Population mit einem intensiven Aussterben den Vergleich verfälschen würde.
Ein Beispiel für eine LPA-Normalisierung sieht folgendermaßen aus:

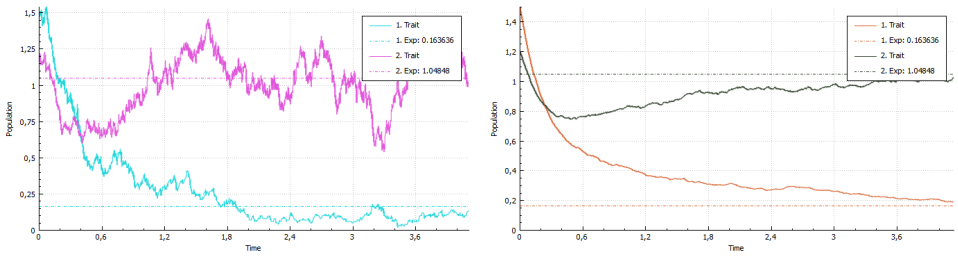


Abbildung 1: LPA Normalisierung mit K=100 und K = 1000

Was bereits in beiden Abbildungen auffällt, ist dass die Population für kleine K fast sofort aussterben würde, weil die Merkmale sich auf ein gefährlich geringes Gleichgewicht einpendeln wollen.
Für den Prozess ν_t^K ändert sich der Generator ganz einfach zu:

$$\begin{aligned} L^K(\phi(\nu^K)) = &\int_X b(x)(1-\mu)\left[\phi\left(\nu^K + \frac{\delta_x}{K}\right) - \phi(\nu^K)\right] K\nu^K(dx) \\ &+ \int_X \int_{\mathbb{R}^d} b(x)\cdot\mu\left[\phi\left(\nu^K + \frac{\delta_{x+z}}{K}\right) - \phi(\nu)\right] m(x,dz) K\nu^K(dx) \\ &+ \int_X d(x)\left[\phi\left(\nu^K - \frac{\delta_x}{K}\right) - \phi(\nu^K)\right] K\nu^K(dx) \\ &+ \int_X \left(\int_X c_K(x,y) K\nu^K(dy)\right)\left[\phi(\nu^K - \frac{\delta_x}{K}) - \phi(\nu^K)\right] K\nu^K(dx) \end{aligned}$$

und in unserem Fall zu:

$$\begin{aligned} L^K(\phi(\nu^K)) = &\sum_{x\in X} b(x)(1-\mu)\left[\phi\left(\nu^K + \frac{\delta_x}{K}\right) - \phi(\nu^K)\right] K\cdot n(x) \\ &+ \sum_{y\sim x} b(x)\cdot\mu\cdot\left[\phi\left(\nu^K + \frac{\delta_y}{K}\right) - \phi(\nu^K)\right] K\cdot n(x) \\ &+ \sum_{x\in X}\left(d(x) + \sum_{y\in X} c_K(x,y) K\cdot n(y)\right) \\ &\cdot\left[\phi\left(\nu^K - \frac{\delta_x}{K}\right) - \phi(\nu^K)\right] K\cdot n(x) \end{aligned} \tag{2}$$

3.2 LPA für zwei Merkmale ohne Mutation

Falls mit $K\to\infty$ auch $n_0^K \to n_0$ folgt, dann lässt sich beweisen, dass das System gegen ein deterministisches System konvergiert $\nu_t^K \xrightarrow{K} n_t$. Exemplarisch gehen wir von einem Fall von zwei Merkmalen ohne Mutation aus, jedoch lässt es sich auch auf d Merkmale auch mit Mutation erweitern. Dieses Beispiel ist für den TSS-Fall besonders interessant.

Ein solches deterministisches System muss folgende Differentialgleichung erfüllen:

$$\begin{aligned} \dot{n}(x) &= n(x)\left(b(x) - d(x) - c(x,x)n(x) - c(x,y)n(y)\right), \quad n_0(x) = n_{0,x} \\ \dot{n}(y) &= n(y)\left(b(y) - d(y) - c(y,y)n(y) - c(y,x)n(x)\right), \quad n_0(y) = n_{0,y} \end{aligned} \tag{3}$$

Um die Konvergenz zeigen zu können, verwenden wir ein Theorem aus [7, Kapitel 11, Thm 2.1].
Dafür wird zunächst erläutert, ob unser Modell die Bedingungen aus [7] erfüllt. Zu diesem Zweck wird unser mutationsfreies Modell in eine passende Notation aus [7] übersetzt.
Sei $l \in \{\binom{1}{0}, \binom{-1}{0}, \binom{0}{1}, \binom{0}{-1}\}$ und $\beta_l : \mathbb{R}^2 \to \mathbb{R}_+$. Mit l kann man das Merkmal und Ereignis auffassen, während β_l eine Ratenfunktion ist, welche die Raten eines Ereignisses für ein Merkmal der Population darstellt.
In unserem Fall kann allerdings nur einem Merkmal ein Ereignis widerfahren. Deswegen werden unsere l stets Einheitsvektoren sein, die auf das Merkmal verweisen, mit Vorzeichen, die auf das Ereignis deuten. Z.B. $l = \binom{0}{-1}$ meint einen Tod im zweiten Merkmal.
Da β_l eine Population als Vektor erwartet, werden wir unsere Population mit $n_t = \binom{n_t(x)}{n_t(y)}$ beschreiben. Daraus ergibt sich für das β mit obigem Beispiel:

$$\beta_{\binom{0}{-1}}(n_t) = \beta_{\binom{0}{-1}}\begin{pmatrix} n_t(x) \\ n_t(y) \end{pmatrix} = d(y) \cdot n_t(y) + \left(\sum_{x \in X} c(y,x) n_t(x)\right) \cdot n(y)$$

Dementsprechend ist:

$$\beta_{\binom{1}{0}}(n_t) = b(x) n_t(x)$$

Diese Raten lassen sich auch für ν_t^K formulieren [7, Kapitel 11 - (1.12)]:

$$\begin{aligned} q_l : \frac{\mathbb{N}}{K} &\longrightarrow \mathbb{R}_+ \\ q_l : \nu_t^K &\longmapsto K\beta_l\left(\frac{\nu_t}{K}\right) \end{aligned}$$

Daran erkennt man, dass sich auch die Wettbewerbsrate zu unserer verändert:

$$\begin{aligned} K\beta_{\binom{-1}{0}}\left(\frac{\nu_t}{K}\right) &= K \cdot d(y)\frac{\nu_t(y)}{K} + K \cdot \left(\sum_{x \in X} c(y,x) \cdot \frac{n_t(x)}{K}\right) \cdot \frac{n_t(y)}{K} \\ &= d(y)\nu_t(y) + \left(\sum_{x \in X} \frac{c(y,x)}{K} \cdot n_t(x)\right) \cdot n_t(y) \\ &= d(y)\nu_t(y) + \left(\sum_{x \in X} \boldsymbol{c^K(y,x)} \cdot n_t(x)\right) \cdot n_t(y) \end{aligned}$$

Die vorherigen Übersetzungen lassen sich leicht anhand des Generators nachvollziehen, wobei unser Generator (1), nur ohne Mutation, dasselbe ergeben soll wie:

$$\sum_l \beta_l(n_t)(f(n_t+l)-f(n_t))$$

Als nächstes kommen wir zur Definition des F, welche sich aus der Gleichung [7, Kapitel 6 - (2.2)] ergibt:

$$F(n_t)=\sum_l l\beta_l(n_t)$$

Wenn man die Summe für $l=\begin{pmatrix}1\\0\end{pmatrix}, l=\begin{pmatrix}-1\\0\end{pmatrix}$ betrachtet, so beschränkt man sich auf die erste Zeile der Funktion, also:

$$F(n_t)_1 = 1\cdot \underbrace{b(x)n(x)}_{\beta_{\binom{1}{0}}(n_t)} + (-1)\cdot \underbrace{(d(x)n(x)+\sum_{y\in X} c(x,y)n(x))}_{\beta_{\binom{-1}{0}}(n_t)},$$

was mit (3) übereinstimmt. Also gilt $F_k(n_t)=\dot{n}_t(x_k)$, wobei $x_k\hat{=}$k-te Merkmal.
Kommen wir nun zu dem eigentlichen Theorem.

Satz 3.1 ([7], Kapitel 11 - Theorem 2.1)**.** *Sei $V\subset\mathbb{R}^2$ kompakt,*

$$\sum_l |l| \sup_{n_t\in V} \beta_l(n_t) < \infty \tag{4}$$

und es existiert ein $M_V>0$, sodass

$$|F(n_t)-F(\tilde{n}_t)| \le M_V|n_t-\tilde{n}_t|, \qquad n_t,\tilde{n}_t\in V \tag{5}$$

Angenommen ν_t^K erfüllt [7, Kapitel 11 - (2.3)] und $\lim_{K\to\infty}\nu_0^K=n_0$, und n erfüllt

$$n_t = n_0+\int_0^t F(n_s)ds, \qquad t\ge 0 \tag{6}$$

Dann gilt für jedes $t>0$,

$$\lim_{K\to\infty}\sup_{s\le t}|\nu_t^K-n_t|=0 \quad f.s. \tag{7}$$

Es bleibt also zu zeigen, dass unser Modell die Bedingungen aus Satz 3.1, bzw. aus [7, Kap. 11 - **Theorem 2.1**] erfüllt.

Satz 3.2. *Unser mutationsfreies Modell erfüllt die Bedingungen von [7, Kap. 11 - **Theorem 2.1**].*

Beweis. Wir gehen zunächst von einer dimorphen Population $X = \{x, y\}$ aus. Seien

$$n_1 = \begin{pmatrix} n_1(x) \\ n_1(y) \end{pmatrix}, \quad n_2 = \begin{pmatrix} n_2(x) \\ n_2(y) \end{pmatrix}$$

zwei Lösungen der Differentialgleichung

$$F(n) = \begin{pmatrix} \dot{n}(x) \\ \dot{n}(y) \end{pmatrix} = \begin{pmatrix} n(x)(b(x) - d(x) - c(x,x)n(x) - c(x,y)n(y)) \\ n(y)(b(y) - d(y) - c(y,y)n(y) - c(y,x)n(x)) \end{pmatrix} \tag{8}$$

ausgewertet zu einem Zeitpunkt $s \in \mathbb{R}_+$.

Bedingung (4) bzw. [7, Kapitel 11 - **Thm 2.1** (2.6)] zu prüfen ist in unserem Fall sehr einfach. Unser Merkmalsraum und die verwendeten Raten sind endlich. Damit liegt stets eine endliche Stumme über endliche Raten vor, die natürlich wieder endlich ist. Das gilt für jedes $n_t \in V$, da wir wie in [1], nur endliche Raten zulassen.

Bedingung (5) bzw. [7, Kapitel 11 - **Thm 2.1** (2.7)] fordert

$$\left| F \begin{pmatrix} n_1(x) \\ n_1(y) \end{pmatrix} - F \begin{pmatrix} n_2(x) \\ n_2(y) \end{pmatrix} \right| < M_V \left| \begin{pmatrix} n_1(x) \\ n_1(y) \end{pmatrix} - \begin{pmatrix} n_2(x) \\ n_2(y) \end{pmatrix} \right|, \quad M_V \in \mathbb{R}_+$$

für $n_1, n_2 \in V$. Es ist klar, dass

$$\begin{aligned} |n_1(x) - n_2(x)| &\leq |n_1 - n_2| \\ |n_1(y) - n_2(y)| &\leq |n_1 - n_2| \end{aligned} \tag{9}$$

Falls es ein $c_V \in \mathbb{R}_+$ gibt mit

$$\begin{aligned} |F(n_1)_1 - F(n_2)_1| &\leq |n_1 - n_2| \cdot c_V \\ |F(n_1)_2 - F(n_2)_2| &\leq |n_1 - n_2| \cdot c_V, \end{aligned} \tag{10}$$

so folgt wegen

$$\begin{aligned} |F(n_1) - F(n_2)| &= \sqrt{(F(n_1)_1 - F(n_2)_1)^2 + (F(n_1)_2 - F(n_2)_2)^2} \\ &\leq \sqrt{(|n_1 - n_2| \cdot c_V)^2 + (|n_1 - n_2| \cdot c_V)^2} \\ &= |n_1 - n_2| \cdot \underbrace{\sqrt{2} \cdot c_V}_{<\infty} = |n_1 - n_2| \cdot M_V \Rightarrow (5) \end{aligned} \tag{11}$$

Also bleibt nur noch (10) zu prüfen. Dabei wird benötigt, dass $|n_1(x)| + |n_2(x)|$ beschränkt ist. Das ergibt sich aus der Voraussetzung, dass V kompakt ist und $n_1, n_2 \in V$ gewählt wurden. Diese Wahl ist für unser Modell sinnvoll, weil unsere Population mit einer endlichen Anfangsbedingung startet und bis zu einem festen Zeitpunkt $t > 0$ stets endliche Werte annimmt. Die Endlichkeit zu jedem Zeitpunkt $t > 0$ kann dadurch begründet werden,

dass unsere Population durch eine identische ohne Todesraten zu jedem Zeitpunkt von dieser beschränkt wäre (also $n_t(x) = b(x) \cdot t$).
Für F_1 und F_2 ist dabei das Vorgehen analog, daher wird nur F_1 vorgestellt:

$$\begin{aligned}
|F(n_1)_1 - F(n_2)_1| &= |(n_1(x) - n_2(x))(b(x) - d(x)) \\
&\quad - ((n_1(x))^2 - (n_2(x))^2) \cdot c(x,x) \\
&\quad - ((n_1(y))^2 - (n_2(y))^2) \cdot c(x,y)| \\
&\leq |\underbrace{(n_1(x) - n_2(x))}_{\leq |n_1 - n_2|}(b(x) - d(x))| \\
&\quad + |(n_1(x) - n_2(x))(n_1(x) + n_2(x)) \cdot c(x,x)| \\
&\quad + |(n_1(y) - n_2(y))(n_1(y) + n_2(y)) \cdot c(x,y)| \\
&\leq |n_1 - n_2| \cdot |b(x) - d(x)| \\
&\quad + |n_1 - n_2| \cdot |\underbrace{n_1(x) + n_2(x)}_{\text{beschränkt}}| c(x,x) \\
&\quad + |n_1 - n_2| \cdot |n_1(y) + n_2(y)| \cdot c(x,y) \\
&\leq |n_1 - n_2| \cdot (c_1 + c_{2,V} \cdot c(x,x) + c_{3,V} \cdot c(x,y)) \\
&= |n_1 - n_2| \cdot c_V
\end{aligned}$$

Wie schon erwähnt folgt durch analoges Vorgehen für y, dass (9) für unser Modell gilt.
Tatsächlich können für Fälle mit mehr als 2 Merkmalen durch analoges Vorgehen dieselben Abschätzungen gemacht werden, die ebenso (5) bestätigen.

Um *Annahme [7, Kapitel 11 - (2.3)]* nachzuweisen, verwenden wir zunächst aus dem selben Abschnitt die Darstellung von ν_t^K als unskalierten Prozess $K \cdot \nu_t^K = \hat{\nu}_t{}^K \in \mathbb{N}^2$ durch

$$\hat{\nu}_t{}^K = \hat{\nu}_0^K + \sum_l lY\left(n \int_0^t \beta_l\left(\frac{\hat{\nu}_s{}^K}{K}\right) ds\right)$$

Dabei sind die Y_l unabhängige standard Poisson Prozesse, die unsere Tode und Geburten auf jedem Merkmal bestimmen. Es ist erkennbar dass die Summe tatsächlich den Verlauf unseres Markov Prozesses darstellt.
Schließlich wird für Teil (2.3) aus [7, Kapitel 11] lediglich gefordert, dass unser Prozess ν_t^K durch $\nu_t^K = \frac{\hat{\nu}_t{}^K}{K}$ die Gleichung

$$\nu_t^K = \nu_0^K + \sum_l \frac{l}{K} \widetilde{Y}_l\left(n \int_0^t \beta_l(\nu_s^K) ds\right) + \int_0^t F(\nu_s^K) ds,$$

erfüllt, wobei $\widetilde{Y}_l(u) = Y_l(u) - u$ ein am Erwartungswert zentrierter Poisson Prozess ist.

Wir bemerken, dass hier neben der Skalierung $\frac{1}{K}$, nur das Superpositionsprinzip Anwendung gefunden hat. Statt mit dem Poisson Prozess die Entwicklung der Geburten und Tode zu beschreiben, wird hier die skalierte Abweichung der Geburten und Tode beschrieben und zum erwarteten Wert ergänzt. Also ist die Annahme (2.3) für unseren Prozess zutreffend.

Und *Bedingung (6)* folgt direkt aus unserer Definition

$$n_t = n_0 + \int_0^t \dot{n}_s ds = n_0 + \int_0^t F(n_s) ds$$

womit alle Bedingungen für (3.1) erfüllt sind und wir die Konvergenz (7) nachgewiesen haben. □

3.3 Monomorphes Gleichgewicht

Wir stellen fest, dass im Falle der monomorphen Population, d.h. $X = \{x\}$, für $K \to \infty$, ν_t gegen eine Funktion konvergiert, die folgende Gleichung erfüllt:

$$\begin{aligned} \dot{n} &= (b(x) - d(x) - n \cdot c(x,x)) \cdot n \\ n(0) &= n_0 \end{aligned} \tag{12}$$

Wir wollen hieraus einen stabilen nicht trivialen Zustand für die Population ermitteln, in dem sich die Populationsgröße nicht mehr ändern darf:

$$\begin{aligned} 0 = \dot{n} &= (b(x) - d(x) - nc(x,x))n \\ \Rightarrow 0 &= b(x) - d(x) - nc(x,x) \\ \Rightarrow \bar{n} &= \frac{b(x) - d(x)}{c(x,x)} \quad \wedge \quad \bar{n} = 0 \end{aligned} \tag{13}$$

$\bar{n}$ ist somit das Gleichgewicht einer monomorphen Population, falls sie nicht zuvor ausstirbt. Eine ausgestorbene Population hat natürlich keine Änderungsrate mehr und erfüllt somit jede Gleichgewichtsgleichung. Zudem gilt, dass stets eine Konvergenz der Population gegen $\bar{n}$ für beliebige Startwerte vorliegt. Ab jetzt wird mit $\bar{n}_x$ der monomorphe Gleichgewichtszustand aus (13) für das Merkmal x beschrieben.

3.4 Die Fitnessfunktion

Spätestens jetzt wird die Fitnessfunktion interessant:

$$f(x,y) = b(x) - d(x) - c(x,y)\bar{n}_y$$

Die Fitnessfunktion gibt an, wie gut sich ein Mutant eines ausgestorbenen Merkmals x gegen ein Merkmal y im Gleichgewicht $\bar{n}_y$ (13) durchsetzten

kann.
Wenn man die Fitnessfunktion genauer untersucht, bemerkt man, dass für ein durchsetzungsfähiges Individuum ($f(x,y) > 0$) bereits die Geburtenrate größer sein muss als die eigene interne Todesrate zusammen mit der wettbewerblichen Todesrate des konkurrierenden Merkmals. Es muss also erstmal selbstständig überleben ($b(x) - d(x) > 0$) und dazu noch dem Konkorruenzdruck widerstehen können ($b(x) - d(x) - c(x,y)\bar{n}_y > 0$).
Hierbei wird $c(x,x)$ nicht berücksichtigt, weil es im Grenzwert mit K immer geringeren Einfluss hat. Gleichermaßen haben wenige x kaum Einfluss auf den Gleichgewichtszustand $\bar{n}_y$ von y. Damit begründet sich der Widerstand gegen den Mutanten durch die eigene Todesrate und die nahezu konstante Konkurrenz durch $c(x,y)\bar{n}_y$.
Die Fitnessfunktion ist also die asymptotische Wachstumsrate von x, wenn y sich in einem Gleichgewichtszustand befindet und nur wenige Individuen von Typ x in der Population vorhanden sind.
Wenn in einer monomorphen Population ein Mutant eine Verdrängung des bis dahin dominanten Merkmals auslöst, so nennt man diesen Vorgang Invasion. Näheres zur Invasion findet sich in Kapitel 7 (TSS-Prozesse).

3.5 Dimorphes Gleichgewicht

Wir wissen mittlerweile, dass, falls $n_0^K \to n_0$, eine dimorphe Population $\nu_0^K = n_0^K(x)\delta_x + n_0^K(y)\delta_y$ ohne Mutation für $K \to \infty$ gegen ein deterministisches System $(n(x), n(y))$ konvergiert. Für diesen Fall gilt:

$$\begin{aligned} \dot{n}(x) &= n(x)(b(x) - d(x) - c(x,x)n(x) - c(y,x)n(y)) \quad n_0(x) = n_{0,x} \\ \dot{n}(y) &= n(y)(b(y) - d(y) - c(y,y)n(y) - c(x,y)n(x)) \quad n_0(y) = n_{0,y} \end{aligned} \tag{14}$$

Hier ist leicht zu erkennen, dass $(\bar{n}(x), 0)$, $(0, \bar{n}(y))$ und $(0,0)$ stabile Zustände sind. Jedoch gibt es in diesem Fall auch einen Zustand, in dem eine Koexistenz beider Merkmale herrschen kann:

$$\begin{aligned} n_x &= \frac{(b(x) - d(x))c(y,y) - (b(y) - d(y))c(x,y)}{c(y,y)c(x,x) - c(y,x)c(x,y)} \\ n_y &= \frac{(b(y) - d(y))c(x,x) - (b(x) - d(x))c(y,x)}{c(y,y)c(x,x) - c(y,x)c(x,y)} \end{aligned} \tag{15}$$

Die BPDL Simulationen erkennen dimorphe und monomorphe Populationen und stellen stets einen passenden stabilen Zustand n_x, bzw. $\bar{n}_x$ dar.
Um im dimorphen Fall zu entscheiden, unter welchen Voraussetzungen zu welchem Gleichgewicht konvergiert wird, benötigen wir [6, Proposition 3]. Darin werden die Gleichgewichte $(\bar{n}_x, 0)$ und $(0, \bar{n}_y)$ untersucht:

> Falls $f(y,x) < 0$, so ist $(\bar{n}_x, 0)$ ein stabiler Zustand.

Falls jedoch $f(y,x) > 0$ und $f(x,y) < 0$, so ist $(0, \bar{n}_y)$ stabil und $(\bar{n}_x, 0)$ ist instabil. In diesem Fall konvergiert jede Lösung von (14) mit $n_{0,y} > 0$ gegen $(0, \bar{n}_y)$.

Der Beweis dazu findet sich in [8] und unterscheidet die Konvergenz in einem Linearen System (aus 14) in der Nähe der kritischen Punkte zu den Gleichgewichtszuständen.

Das folgende Bild zeigt sowohl die Konvergenz gegen das eben berechnete Gleichgewicht (15), als auch das deterministische Verhalten für sehr große K mit positiver Fitness auf beiden Seiten:

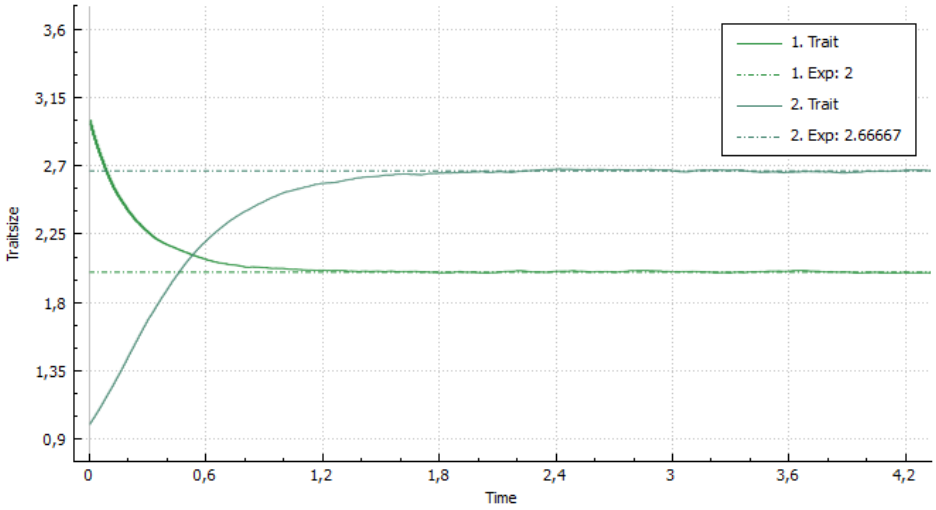


Abbildung 2: Konvergenz mit K=100000 und $15 \cdot 10^6$ Sprüngen

3.6 Der TSS-Grenzwertprozess

Wie zuvor bei der LPA-Normalisierung erhalten wir TSS-Prozesse (Trait Substitution Sequence) als Grenzprozesse von BPDL-Prozessen mit großen Populationen und seltenen Mutationen. Mit wachsendem K soll für die Mutationswahrscheinlichkeit durch die Vorschrift

$$\frac{1}{e^{cK}} \ll \mu_K \ll \frac{1}{K \log(K)}, \qquad \forall c > 0, \tag{16}$$

eine schnellere Konvergenz gegen 0 stattfinden, als die Geburten pro fester Zeiteinheit gegen unendlich streben. D.h. mit wachsendem K werden Mutationen zunehmend seltener.
Der TSS-Prozess ist ein besonders interessanter Grenzprozess des BPDL-Prozesses, weil es ihm möglich ist, von einer monomorphen Population im Merkmal x zu einer anderen monomorphen Population mit Merkmal y zu

springen, falls $f(y,x) > 0$ und $f(x,y) < 0$ (mehr dazu in Kapitel 7).
In diesem Fall haben wir zu jeder festen Zeit höchstens zwei konkurrierende Merkmale. Diese Eigenschaft vereinfacht die Analyse des Prozesses sehr und ist der Wahl von μ_K zu verdanken.

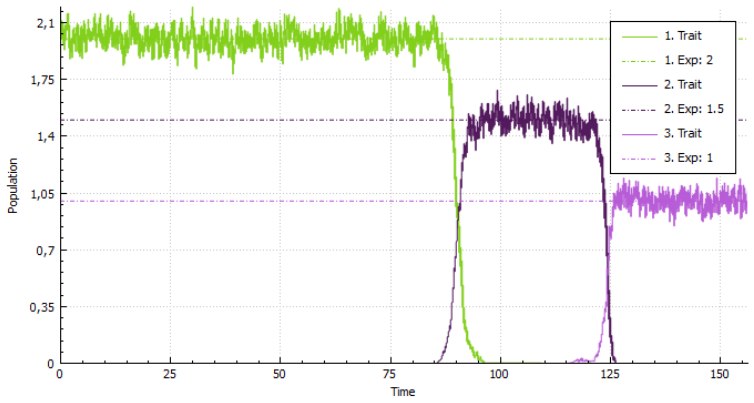


Abbildung 3: TSS-Prozess mit: K = 1000 und $4 \cdot 10^6$ Sprüngen

Frage: Warum bietet μ_K im Bereich (16) diese Eigenschaften?

Antwort: Dank Freidlin und Wenzell [9] erwarten wir, dass unsere dominante Spezies eine Zeit von der Ordnung $\exp(cK)$ im Gleichgewicht bleibt. Schließlich können wir so kontrollieren, wie lange eine dominante Spezies, die für eine mutative Geburt in Frage kommt, erhalten bleibt. Dadurch, dass die Mutationen exponentiell verteilt sind, benötigt man eine Rate $\mu_K \gg \frac{1}{e^{cK}}$ um eine Zeit von $\exp(cK)$ nicht zu überschreiten, was die *untere Schranke* rechtfertigt.
Um die *obere Schranke* zu rechtfertigen, betrachten wir nun die Zeit, die ein Mutant braucht, um ein dominantes Merkmal aussterben zu lassen. Das wird uns die Möglichkeit geben, einer neuen Mutation genug Zeit zu lassen bis das derzeit benachteiligte Merkmal ausgestorben ist.
Angenommen es ereignet sich eine Mutation und der Mutant in y ist fitter ($f(y,x) > 0, f(x,y) < 0$), so wird mit positiver Wahrscheinlichkeit eine Invasion (näheres dazu im Kapitel 7 - TSS-Prozesse) ausgelöst. Wenn man Branching-Prozesse mit dem Lotka-Volterra-System vergleicht, kommt man darauf, dass die benötigte Zeit zum Verdrängen und Aussterben des ursprünglich dominanten Merkmals von der Ordnung $\log(K)$ ist. Skaliert man nun noch zusätzlich die Zeit, so führt dies dazu, dass der Prozess ausreichend Zeit zwischen zwei Mutationen hat, um ein benachteiligtes Merkmal zu verdrängen. Somit erlaubt uns die LPA-Annahme von einer deterministischen Populationsdynamik zwischen zwei Mutationen auszugehen [5].

4 Simulation

In diesem Kapitel wird der Kern der Simulation algorithmisch näher untersucht. Dieser Kern besteht im Wesentlichen aus einem Sprung des BPDL-Prozesses. Dabei wird zwischen der Implementierung und dem Pseudocode unterschieden, weil bei der Implementierung sorgfältig auf die Trennung der Aufgabenbereiche geachtet wurde, welche später beim Verhaltenstest sehr wichtig und im 5. Kapitel weiter verwendet werden.
Die hier verwendete Vorgehensweise unterscheidet sich von der aus [1], weil wir ein System mit vielen Individuen und wenigen Merkmalen betrachten, was uns ermöglicht, die individuellen Raten zu berücksichtigen.

4.1 Implementierung

Die Simulation durchläuft mehrere Schritte bis ein vollständiger Sprung von ν_t abgeschlossen ist. Hier wird beschrieben in welcher Reihenfolge welche Schritte durchlaufen werden und welche Aufgabe diese erfüllen. Am Ende werden alle Funktionsaufrufe (Schritte) und Zusammenhänge in Abbildung 5 als ein Ablauf-Tiefen-Diagramm illustriert.
Im Code wird dabei objektorientiert mit Klassen und Objekten gearbeitet. Da diese Details nicht besonders von Interesse sind, wird eher ein heuristischer Überblick der Implementierung gegeben. So kann hier z.B. angenommen werden, dass die Variable "Members[i]" Zugriff auf die Anzahl der Individuen des i-ten Merkmals bietet, was im Code jedoch komplexer realisiert werden musste.

4.1.1 Raten berechnen

Bevor ein Merkmal und Ereignis ausgewählt werden kann, müssen die Raten bekannt sein, nach denen die exponentiellen Uhren gestellt werden.
Die Todesrate setzt sich aus der intrinsischen Todesrate und der durch Wettbewerb zusammen und ist zu Beginn 0.

Die folgende Funktion addiert die intrinsische Todesrate zur aktuellen Todesrate. Dabei wird direkt das Superpositionsprinzip genutzt, um die gesamte intrinsische Todesrate des Merkmals in "TotalDeathRate[i]" aufzuaddieren.

Algorithm 1 addTotalIntrinsicDeathRateOf(TraitIndex: i)

Ensure: addiert zur Todesrate die intrinsische-Todesrate

1: TotalDeathRate[i] = DeathRate[i] · Members[i]

Diese Funktion addiert die wettbewerbliche Todesrate zur aktuellen Todesrate.

Algorithm 2 addTotalCompDeathRateOf(TraitIndex: i)

Ensure: addiert zur Todesrate die Wettbewerbs-Todesrate

```
1: for j=0 → n-1 do
2:     TotalDeathRate[i] += CompDeathRate[i,j] · Members[i] · Members[j];
3: end for
```

Die auf den ersten Blick übertrieben erscheinende Trennung der beiden Funktionen ist für das verwendete Programmierkonzept entscheidend, nach dem jeder Funktion eine möglichst eindeutige Aufgabe zukommen soll [10]. Darauf wird näher im nächsten Kapitel eingegangen.
Schließlich erfolgt aus obigem die Berechnung der Totalen Todesraten:

Algorithm 3 calculateTotalDeathRates()

Ensure: berechnet die gesamten Todesraten aller Merkmale

```
1: for i=0 → n-1 do
2:     TotalDeathRate[i] = 0;
3:     addTotalIntrinsicDeathRateOf(i);
4:     addTotalCompDeathRateOf(i);
5: end for
```

Danach kommen wir zur Berechnung der Geburtsrate pro Merkmal. Auch hier sollten Mutationen und intrinsische Geburten gesondert berechnet werden. Zusammengefasst:

Algorithm 4 calculateTotalBirthRates()

Ensure: berechnet die gesamten Geburtsraten aller Merkmale

```
1:      ↓ intrinsische Geburtenrate ↓
2: for i=0 → n-1 do
3:     TotalBirthRate[i] = Members[i] · BirthRate[i] · (1 - Mutation);
4: end for
5:      ↓ Mutationsraten ↓
6: for i=1 → n-2 do
7:     TotalBirthRate[i] += Members[i-1] · BirthRate[i-1] · Mutation · 0.5;
8:     TotalBirthRate[i] += Members[i+1] · BirthRate[i+1] · Mutation · 0.5;
9: end for
10: TotalBirthRate[0] += Members[1] · BirthRate[1] · Mutation · 0.5;
11: TotalBirthRate[n-1] += Members[n-2] · BirthRate[n-2] · Mutation · 0.5;
```

Jetzt sind wir bereit eine Funktion aufzurufen, die aus den vorher berechneten Geburts- und Todesraten pro Merkmal durch Superposition eine totale

Eventrate berechnet, nach der wir eine exponentielle Uhr stellen können, die schließlich das Klingeln der ersten aller Merkmalsuhren simuliert.

Algorithm 5 calculateTotalEventRate()

Ensure: berechnet die totale Ereignisrate

```
1: TotalEventRate = 0;
2: for i=0 → n-1 do
3:     TotalTraitRate[i] = TotalBirthRate[i] + TotalDeathRate[i];
4:     TotalEventRate += TotalTraitRate[i];
5: end for
```

Hier fällt auf, dass wir auch die *TotalTraitRate* oder Totale Merkmalsrate gespeichert haben. Diese repräsentiert die gesamte Ereignisrate eines Merkmals.
Zum Schluss sollte es eine Funktion geben, die alle bisherigen Funktionen in der richtigen Reihenfolge ausführt und so die Berechnung aller Ereignisraten sichert:

Algorithm 6 calculateEventRates()

Ensure: stellt sicher, dass alle akutellen Raten berechnet wurden

```
1: calculateTotalDeathRates();
2: calculateTotalBirthRates();
3: calculateTotalEventRate();
```

4.1.2 Ereignis und Zeit bestimmen

Mit den zuvor berechneten Raten ist es einfach, die Dauer bis zum nächsten Ereignis zu bestimmen. An dieser Stelle verwende ich die Funktion *rollExpDist(Parameter)* zum Ziehen einer exponentiell verteilten Zufallsvariable, die nicht weiter interessant ist und deshalb nicht erläutert wird.

Algorithm 7 sampleEventTime()

Ensure: Zieht die nächste Ereigniszeit

```
1: EventTime = rollExpDist(TotalEventRate);
2: Timeline += EventTime;
```

Jetzt bleibt zu bestimmen, wem was passiert, also welches Ereignis welches Merkmal treffen wird. Dafür wenden wir das Superpositionsprinzip in anderer Richtung an als bisher:
Zum Bestimmen des auserwählten Merkmals beachten wir den Anteil der Merkmale an der Totalen Eventrate. Dieser ist klar erkennbar durch die

Summe:

$$\text{TotalTraitRate} = \sum_{i=0}^{n-1} \text{TotalTraitRate[i]}$$

Also hat das *i-te* Merkmal mit Wahrscheinlichkeit $\frac{TotalTraitRate[i]}{TotalEventRate}$ das Ereignis ausgelöst. Um also das verantwortliche Merkmal auszuwählen, können wir eine uniform verteilte Zufallsvariable ziehen und entscheiden, welche der Merkmalsraten damit gemeint ist. Abbildung 4 illustriert den Auswahlprozess.

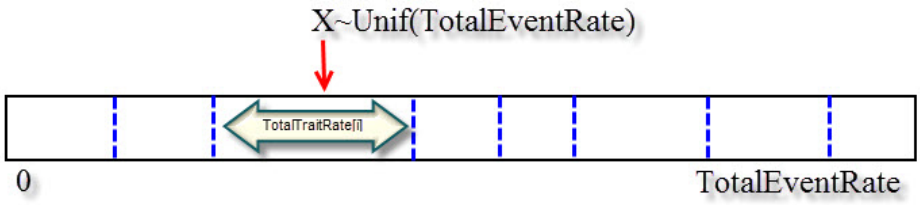


Abbildung 4: Auswahl des Merkmals nach Anteil an der TotalEventRate

Im Code wird dafür iterativ geprüft, ob "X $\sim$ Unif(TotalEventRate)" im ersten Intervall der Länge TotalTraitRate[0] liegt.

> Falls ja, so wird dieses Merkmal gewählt und die Funktion wird verlassen.
>
> Falls nicht, so wird das Merkmal 0 aus den relevanten Merkmalen entfernt und X wird um die Intervallänge TotalTraitRate[0] reduziert, um schließlich erneut mit dem ersten relevanten Intervall verglichen zu werden (jetzt TotalTraitRate[1]). Auf diese Weise nähert man sich immer weiter dem getroffenen Merkmal:

Algorithm 8 choseTraitToChange()

Ensure: wählt ein Merkmal zum Ändern aus

```
1: X = rollUnifDist(TotalEventRate);
2: for i=0 → n-1 do
3:     if X ≤ TotalTraitRate[i] then
4:         ChosenTrait = i;
5:         return;
6:     end if
7:     X -= TotalTraitRate[i];
8: end for
```

Auf dieselbe Weise wird entschieden, welches Ereignis eintrifft und anschließend in *isBirth* gespeichert. Da wir hier jedoch nur Geburt und Tod zur

Auswahl haben, würde sich natürlich eine Bernoulli verteilte Zufallsvariable ergeben mit:

$$\text{isBirth} \sim \text{Ber}(\text{TotalBirthRate[ChosenTrait]})$$

Im Code wurde *isBirth* folgendermaßen gezogen:

Algorithm 9 choseEventType()

Ensure: wählt ein Ereignis für das entsprechende Merkmal aus

```
1: X = rollUnifDist(TotalTraitRate[ChosenTrait]);
2: if X ≤ TotalBirthRate[ChosenTrait] then
3:     isBirth = true;
4: else
5:     isBirth = false;
6: end if
```

Danach muss noch das Ereignis aus Alg. 9 auf das Merkmal aus Alg. 8 angewendet werden.

Algorithm 10 executeEventTypeOnTrait()

Ensure: wendet das gewählte Ereignis auf das gewählte Merkmal an

```
1: X = rollUnifDist(TotalTraitRate[ChosenTrait]);
2: if isBirth then
3:     Members[ChosenTrait] += 1;
4: end if
5: if ¬isBirth & Members[ChosenTrait] > 0 then
6:     Members[ChosenTrait] -= 1;
7: end if
```

Zum Schluss wird noch eine Funktion erstellt, welche das Ausführen eines Ereignisses in richtiger Reihenfolge realisiert und in einem Schritt aus gegebenen Raten die Veränderung der Population durchführt.

Algorithm 11 changeATrait()

Ensure: lässt ein Ereignis ein Merkmal treffen

```
1: choseTraitToChange();
2: choseEventType();
3: executeEventTypeOnTrait();
```

Aus diesen 3 wesentlichen Schritten

- Raten berechnen
- Ereigniszeit ziehen

- Ereignis eintreten lassen

kann schließlich eine sehr übersichtliche Funktion konstruiert werden, die einen kompletten Sprung des Prozesses durchführt.

Algorithm 12 makeEvolutionStep()

Ensure: lässt ein Ereignis ein Merkmal treffen

```
1: calculateEventRates();
2: sampleEventTime();
3: changeATrait();
```

4.1.3 Übersicht

Hier ist eine Übersicht aller Funktionen, ihrer Reihenfolge und Aufruftiefe. Die Funktionen wurden auf Englisch beschrieben, weil sie damit eine Referenz zu der im Quellcode beschriebenen Funktion darstellen. Z.B. "make one evolution step" verweist auf die Funktion "makeEvoultionStep", oder "calculate event rates" → "calculateEventRates" etc.

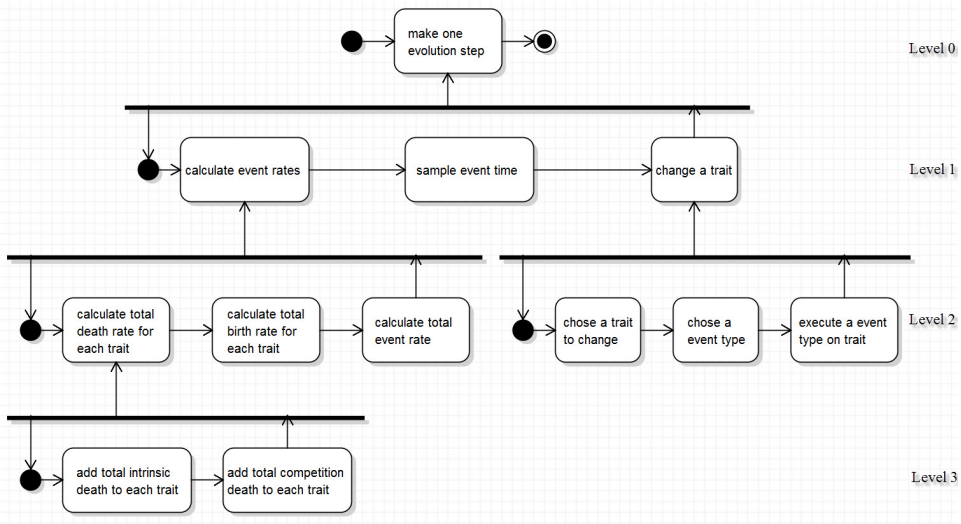


Abbildung 5: Diagramm mit Funktionsaufrufen und ihren Tiefenebenen

4.2 Pseudocode

Natürlich lässt sich der Ablauf eines Sprunges auch durch Pseudocode in eine Funktion "makeEvoultionStep" zusammenfassen.
Im Pseudocode finden sich in blau einige Verweise auf die zuvor beschriebene Implementierung.

Algorithm 13 makeEvolutionStep() - Part 1

Ensure: A full evolution Step happened

Require: $t, X = \{0, \dots, n-1\}$

for $x \in X$ **do** ▷ ↓ calculateEventRates() ↓

  $D(x) := n_t(x) \cdot \left(d(x) + \sum_{y \in X} c(x,y) \cdot n_t(y)\right)$

  $B(x) := \underbrace{b(x) \cdot (1-\mu) \cdot n_t(x)}_{arteigene}$

  if $x > 0$ **then**

    $B(x) += \underbrace{b(x-1) \cdot n_t(x-1)}_{MutationLinks} \cdot \frac{\mu}{2}$

  end if

  if $x < n-1$ **then**

    $B(x) += \underbrace{b(x+1) \cdot n_t(x+1)}_{MutationRechts} \cdot \frac{\mu}{2}$

  end if

  $TotalTraitRate(x) = B(x) + D(x)$

end for

$TotalEventRate := \sum_{x \in X} TotalTraitRate(x)$

sample $Z \sim exp(TotalEventRate)$ ▷ ↓ sampleEventTime() ↓

$t += Z$

sample $Y \sim U(0, TotalEventRate)$ ▷ ↓ choseTraitToChange() ↓

for $x \in X$ **do**

  if $Y \leq TotalTraitRate(x)$ **then**

    $ChosenTrait := x$

    break

  end if

  $Y -= TotalTraitRate(x)$

end for

sample $Y \sim U(0, TotalTraitRate(ChosenTrait))$

if $Y \leq B(ChosenTrait)$ **then** ▷ ↓ choseEventType() ↓

  isBirht := true

else

  isBirth := false

end if

if isBirth **then** ▷ ↓ executeEventTypeOnTrait() ↓

  $n_t(ChosenTrait) += 1$

else

  if $n_t(ChosenTrait) \geq 0$ **then**

    $n_t(ChosenTrait) -= 1$

  end if

end if

4.3 Optimierung für viele Merkmale

Für eine Simulation ist klar, dass in Abhängigkeit der Sprünge (ggf. hoher) linearer Aufwand zu erwarten ist. Zwar ist dieser am Modell nachvollziehbar, jedoch konnte wegen der Wettbewerbsrate ein quadratischer Aufwand in der Anzahl der Merkmale nicht vermieden werden. Die diesbezüglich von mir vorgenommene Optimierung ist bereits ab dem ersten Merkmal von Nutzen (praktisch wurde dieser jedoch erst ab dem 2. gemessen, vgl. Abb. 19)), und somit trotz der stets überschaubaren Menge an Merkmalen in unserer Simulation relevant.
Nach meinen Tests, die später im Kapitel "Verhaltenstest" eingeführt werden, ergab sich, wie zu erwarten, der größte zeitliche Aufwand in der Berechnung der Raten (genauer: der Todesraten). Die Optimierung vermeidet die komplette Neuberechnung der Raten, sondern führt stattdessen zu deren Anpassung.
Dazu werden die Raten nicht zu Beginn berechnet, wie zuvor in Algorithmus 12 "makeEvolutionStep()" in der 1. Zeile. Stattdessen wird mit den aktuellen Raten eine Ereigniszeit gezogen "sampleEventTime()" und anschließend ein Ereignis ausgelöst "changeATrait()", welches es ermöglicht die nächsten Raten anzupassen "adjustNewEventRates()". Dafür wird zunächst unterschieden, ob ein Tod oder eine Geburt eingetreten ist. Das lässt sich leicht mit der in Algorithmus 9 erwähnten "isBirth" Variable entscheiden.
Angenommen es ereignet sich eine Geburt. Damit können zusammen mit dem ausgewählten Merkmal "chosenTrait" folgende Anpassungen gemacht werden:

- Die **intrinsische Todesrate** von "chosenTrait" wird um die des geborenen Individuums erhöht:
  TotalDeathRate[chosenTrait] += DeathRate[chosenTrait];
- Die **Todesrate durch Wettbewerb** wird bei jedem Merkmal um das geborene Individuum erhöht:
  TotalDeathRate[i] += CompDeathRate[i][chosenTrait]; $\quad \forall\, i \in X$
- Die **Geburtsraten** werden genauso wie die Todesraten behandelt:
  TotalBirthRate[chosenTrait] += BirthRate[chosenTrait];
- Entsprechend auch die **Mutationsraten**:
  TotalBirthRate[i] += Mutation $\cdot$ 0.5 $\cdot$ BirthRate[chosenTrait];
  $\forall$ i $\sim$ chosenTrait
- Zum Schluss noch die **Totale Ereignisrate**:
  Eine erneute Berechnung aller Totalen Raten kann hier nicht vermieden werden.

Man erkennt, dass hier keine quadratische Abhängigkeit der Merkmale mehr zu finden ist.

4.4 Normalisierung

Weder in der Implementierung noch im Pseudocode war bisher von Normalisierung die Rede. Dies liegt darin begründet, dass sie vom Programmkern getrennt wurde, da sie den wesentlichen evolutionären Mechanismus nicht beeinflusst, sondern nur die Sichtweise darauf verändert.
Um diese Trennung zu realisieren wurde an zwei Stellen Veränderungen vorgenommen.

1. Unmittelbar nach dem Einlesen der Parameter wurde sowohl die Wettbewerbsrate durch K geteilt als auch die Startpopulation um K hoch skaliert. Auf diese Weise kann der Programmkern ohne Kenntnis eines K die Evolution durchführen.
   Dadurch lässt sich ein Prozess ohne Rücksicht auf anzupassende Parameter durch wachsendes K verfeinern, sodass die Konvergenz mit wachsendem K wie in Abbildung 2 verfolgt werden kann.

2. Trotz der Berechnung der Ergebnisse ohne Rücksicht auf ein vorhandenes K sollte die Darstellung natürlich reskaliert erfolgen ($\nu_t^K = \frac{\nu_t}{K}$). Deswegen ist es außerhalb des Programmkerns nur über einen sogenannten "getter" (eine spezielle Zugriffsfunktion) möglich auf die Mitglieder eines Merkmals zuzugreifen. Dieser "getter" liefert stets den reskalierten Wert zurück.

4.4.1 Sehr viele Sprünge

Gerade bei der Normalisierung wird es mit wachsendem K notwendig, besonders viele Sprünge zu machen. Die Anzahl der Sprünge übersteigt schnell das Fassungsvermögen eines Arrays. Um auch mehr als 100 Millionen Punkte speichern zu können, wurden zwei Lösungen implementiert:

1. Die "erweiterten Sprünge" sind eine Lösung, die zur Zeit angewendet wird. Zu Beginn der Simulation wird anhand der maximalen Iterationen entschieden wie viel Speicher benötigt wird und eine "Jumps"-Variable gesetzt. Diese "Jumps" Variable enthält die Anzahl der Sprünge bis ein Punkt gespeichert wird. So wird bei 100 Millionen Iterationen nur jeder zehnte Punkte gespeichert.

2. Eine weitere Lösung ist die "storePoint()" Funktion. Sie erstellt zu Beginn der Rechnung eine Datei und hängt immer weiter Punkte an bis die Berechnung vorbei ist.
   Schließlich wird die Graphklasse einzeln jeden Punkt an den Graphen anhängen.

Diese Variante ist implementiert und wurde getestet. Sie ist auch in einem Unit Test enthalten, der aber nicht in Kapitel 6 erläutert wird.

5 Das Programm

In diesem Kapitel werden die Programme und ihre Entwicklung vorgestellt. Dabei wird darauf eingegangen, welche Möglichkeiten es gibt eine GUI zu entwickeln, welche Architektur und welcher Codestil im Programm verwendet wurden, wie das Layout gewählt wurde und wie das Programm zu bedienen ist.
Zunächst sei gesagt, dass die Programme zum Zeitpunkt der Erstellung der Bachelorarbeit noch nicht völlig fertig gestellt sind. Es gibt noch überflüssige Funktionen im TSS-Simulator und noch einige Wünsche zur erweiterten Verwendung des Programms.

5.1 GUI - Entwicklung

Ziel war es, ein Programm mit einer graphischen Oberfläche in C++ zu schreiben. Nach [11, 14 - Grafische Benutzungsschnittstellen] kennt Standard C++ keine Elemente für grafische Benutzungsoberflächen (englisch *graphical user interfaces - GUI*). Weil in unserem Fall aber nicht auf eine solche verzichtet werden konnte, wird hier kurz vorgestellt, welche Varianten der GUI-Programmierung es gibt und wie unsere Auswahl begründet ist.

MFC
Sehr bekannt sind die Microsoft Foundation Classes (MFC) bzw. ihre Nachfolger in .NET für Windows Betriebssysteme. Aber genau hier liegt bereits das Problem. In unserem Fall wurden die Programme für zwei Mitarbeiter entwickelt, welche jeweils MAC verwendeten, an einem Institut, welches überwiegend mit Ubuntu arbeitet. Deswegen kam MFC nicht in Frage.

GTK$_+$ (the GIMP Toolkit)
GTK$_+$ ist eine weitere Bibliothek zur Erstellung von GUI's. Sie wurde hauptsächlich für das Bildbearbeitungsprogramm GIMP, entwickelt. GTK$_+$ wird zur Entwicklung des GNOME-Desktops benutzt, welche auf vielen Linux Systemen zu finden sind. Aber auch dazu gibt es eine bessere Alternative.

Qt
Die portable Qt-Bibliothek ist sehr bekannt und lässt sich sehr einfach auf Windows-, Mac- und Unix-Betriebssysteme portieren. Die aus manchen Linux-Systemen bekannte Benutzungsoberfläche KDE wird mit Qt entwickelt. Damit bietet sich Qt bereits jetzt als eine gute Option zur Entwicklung für alle Betriebssysteme an.

Hinzu kommen noch viele weitere Vorteile, wie die große Auswahl an Hilfsprogrammen und Bibliotheken, die sehr ausführliche Dokumentation , und dass es sich um eine Open Source Software handelt.
"Zusammengefasst: Qt ist die ausgereifteste und umfangreichste Open Source Software für die portable Entwicklung grafischer Benutzungsoberflächen mit C++"[11, S.452].

5.2 Architektur und Module

Zuerst möchte ich die grobe Architektur des Programmcodes vorstellen. Diese kann in drei Module zusammengefasst werden, welche möglichst wenige Schnittstellen untereinander verwenden und damit viel Unabhängigkeit bieten.

1. "Population Kernel"
   Der Programmkern besteht aus dem "Population Kernel", welcher ausschließlich für die Berechnungen zuständig ist. Hier werden keine historischen Daten gespeichert, womit dieser Bereich immer nur die Daten der Population zur aktuellen Berechnung bereit hält. Also enthält dieser nur eine Sammlung von Funktionen, die das Modell befragen können.
   Die einzige Klasse dieses Moduls, mit der eine Kommunikation nach außen möglich ist, ist der "TraitEventManager", der, wie der Name schon sagt, die Ereignisse und Merkmale verwaltet (Abb. 6, rechts).
   Das Besondere am "Population Kernel" ist, dass er mit Standard C++11 implementiert wurde und somit überall wiederverwendbar ist. Er ist leicht zu verändern und sehr flexibel, weshalb man unseren Kern z.B. durch einen anderen ersetzten könnte, ohne Anpassungen an anderen Modulen vornehmen zu müssen. So kann man die simulierten Modelle variieren, solange jedes Modell in der Lage ist, einen unabhängigen Sprung durchzuführen.

2. "Graph Management"
   Der zweite Bereich organisiert alle Daten, die die Simulation anfragen kann. Hier liegen der eigentliche Speicher und die Kommunikation zwischen GUI und Algorithmus.
   Es ist eine Art Hilfsklasse, die eine Kommunikation zwischen den Modulen ermöglicht. Zu diesem Zweck wurde hier bereits die Qt-Bibliothek verwendet, um Daten für die GUI abholbereit anzubieten. Sie erhält die Wünsche des Users in Form von Anweisungen aus der "GUI" und verwendet die Möglichkeiten des "Population Kernel", um entsprechende Daten abrufbereit zu erstellen.

3. "GUI"
Das letzte und unflexibelste Modul umfasst die grafische Benutzungsoberfläche. Hier kommen zwei wichtige Klassen zum Einsatz.
Die erste, "MainWindow", ist das Hauptfenster, in dem der Nutzer die Parameter festlegen und, wenn alles stimmt, auch die Simulation starten kann. Für eine möglichst große Übersicht bei der Analyse der Ergebnisse, verwendet das Hauptfenster eine weitere Klasse, "PlotWindow", welche ein extra Fenster mit Graphen öffnet (Plot-Fenster). Wie man vermuten kann, wird hier bereits etwas zum Darstellen eines Graphen, also fast ausschließlich Qt-Programmierung verwendet.
Dieses Plot-Fenster verwendet ein "GraphClass"-Objekt, um einen Graphen erstellen zu können. Natürlich ist es für die Verwendung der Daten selbst verantwortlich.

Die folgende Abbildung 6 stellt zusammenfassend die obigen Module und die Klassenhierarchie dar:

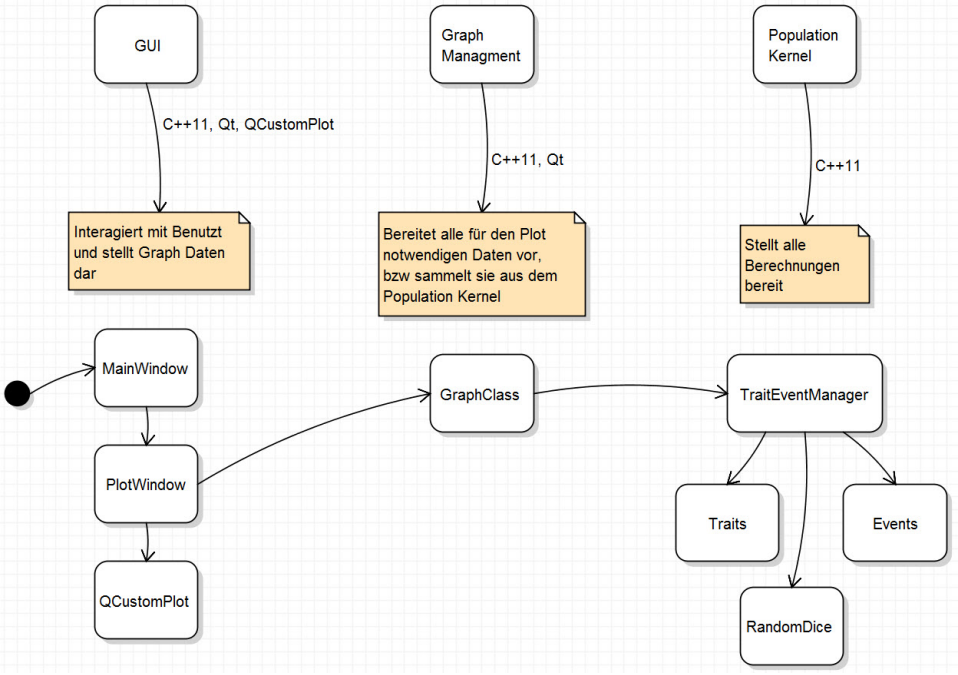


Abbildung 6: Arbeitsmodule und Klassenabhängigkeiten

5.3 Flexibilität und agile Softwareentwicklung

Die Idee der getrennten Aufgabenbereiche geht darauf zurück, dass eine möglichst große Unabhängigkeit zwischen Arbeitsschritten notwendig ist, um das Programm flexibel zu halten und sogenannten "Coderot/Softwareerosion" (faulen Code) zu verhindern [10]. Dieser bezeichnet die zunehmende

Entropie einer Software. Das heißt sie führt mit zunehmender Weiterentwicklung der Software zu einer Verringerung der Leistung, Erschwernissen bei der Anpassbarkeit bzw. Flexibilität und in zunehmendem Maße zu undefiniertem Verhalten.
In der Softwareentwicklung ist undefiniertes Verhalten schwer zu behandeln, da man keine Fehler beim Compilieren oder Ausführen erhält. Es wird lediglich ein nicht nachvollziehbares Verhalten des Programms festgestellt, welches sich im obigen Falle nur noch sehr schwer im Code eingrenzen lässt.

Außerdem wurde darauf geachtet, dass jede Klasse nur eine möglichst fest definierte Aufgabe zu erfüllen hat. In einer Klasse sollten lediglich Funktionen vorhanden sein, die zur Erfüllung dieser Aufgabe beitragen. Dieses Prinzip trägt den Namen "Single-Responsibility-Prinzip" und wurde von Robert C. Martin in [12] eingeführt. Die dort beschriebene "Agile Softwareentwicklung" fand auch Anwendung bei der Neuorganisation von Zielen und Wünschen mit Loren Coquille und Martina Baar, wird aber hier nicht weiter erläutert.

5.4 Layout

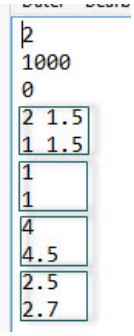
Hier wird die Oberfläche des Programms vorgestellt.

5.4.1 Lesen und Anzeigen von Parametern

Die Bedienung des Programms sollte das Lesen und Anzeigen der Merkmals-Parameter bereitstellen. Da es viele Parameter gibt und die Anzahl der Parameter quadratisch mit der Anzahl der betrachteten Merkmale steigt, bietet sich das Lesen aus zuvor erstellten Dateien an.
Dabei werden die Daten aus den Dateien in folgender Reihenfolge zeilenweise ausgelesen (Abbildung 7):

- m: Anzahl der Merkmale
- K: Parameter
- μ: Mutationswahrscheinlichkeit
- $c(1,1)$ ... $c(1,m)$: 1. Zeile der Wettbewerbsraten
  $\vdots$
  $c(m,1)$ ... $c(m,m)$: m. te Zeile der Wettbewerbsraten
- $n_0(1)$: Populationsgröße des 1. Merkmals:
  $\vdots$
  $n_0(m)$: Populationsgröße des m. Merkmals:

- $b(1)$: Geburtenrate des 1. Merkmals
  ⋮
  $b(m)$: Geburtenrate des m. Merkmals
- $d(1)$: Todesrate des 1. Merkmals
  ⋮
  $d(m)$: Todesrate des m. Merkmals



Dabei wird μ und c natürlich ohne Skalierung mit K angegeben. Die Gründe dafür wurden in Kapitel 4.4 beschrieben.

Nebenstehend sieht man ein Beispiel einer Instanz mit zwei Merkmalen, $K = 1000$, $\mu = 0$ usw., wobei die Markierungen die Wettbewerbsmatrix, die Startpopulation, die Geburtenraten und die Todesraten enthalten.

Abbildung 7:
Datei

Das Programm muss also die Eingabe eines gültigen Dateinamens fordern, bevor eine Simulation gestartet werden kann. Deshalb werden bis zum Zeitpunkt gelesener Parameter alle nicht relevanten Schaltflächen deaktiviert und nur ausgegraut angezeigt Abbildung 8, gelb markiert.
Sobald man einen Namen in das einzig mögliche Feld eingegeben hat, kann man zwischen den Schaltflächen "load File" und "create File" wählen (Abbildung 8, rot markiert).

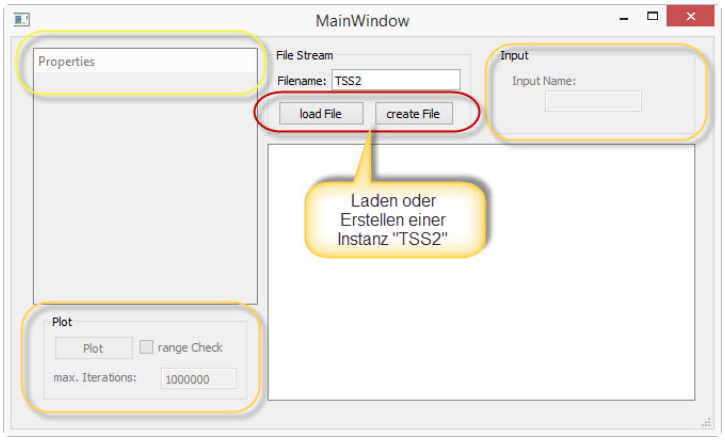


Abbildung 8: MainWindow nach dem Start

Wie schon zuvor erwähnt, ist dem Programm vorab unbekannt, welchen Umfang die gelesenen Parameter haben werden, weshalb die Entscheidung der Darstellung auf eine Baumstruktur fiel.
Sie hat bei der Initialisierung immer denselben Umfang (Abbildung 9) und der gewünschte Ast lässt sich einfach erweitern (Abbildung 10).

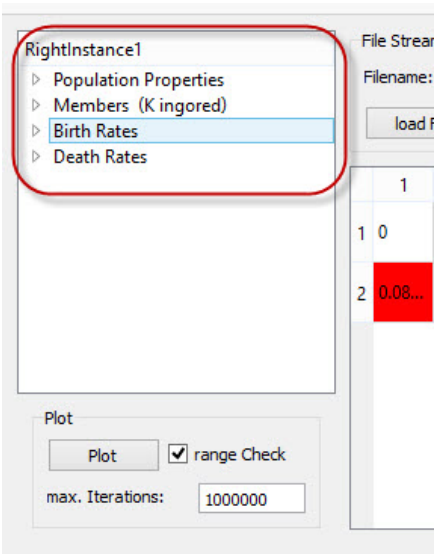


Abbildung 9: Baumstruktur - geschlossen

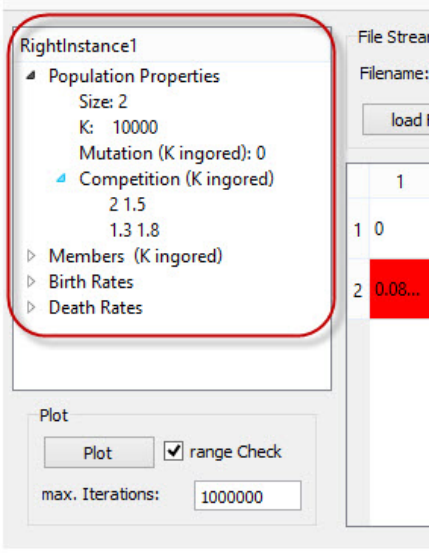


Abbildung 10: Verzweigte Baumstruktur - geöffnet

5.4.2 Schreiben neuer Testinstanzen

Die letzte Herausforderung bestand darin, eine Instanz durch das Programm geleitet erstellen zu können. Während diese Aufgabe bei einer Konsolenanwendung (bekannt aus den klassischen C-Programmen) denkbar einfach mit "printf" und "scanf" erledigt werden konnten, stößt man hier auf das Problem der sogenannten "Ereignisgesteuerten Programmierung" [11].
Während zuvor die Reihenfolge der Programmschritte vorbestimmt war, so gilt das nicht mehr für graphische Benutzungsoberflächen. In unserem Fall ist die Reihenfolge der Interaktionsschritte eines Benutzers, z.B. Anklicken oder Mausbewegung, nicht vorhersehbar und damit auch nicht die Reihenfolge der auszuführenden Programmschritte.
Es geht also darum, Funktionen zu schreiben, die nicht an anderer Stelle vom Programm, sondern bei Eintreffen eines Ereignisses aufgerufen werden und die gewünschte Reaktion liefern.
Da diese feste Reihenfolge jedoch beim Schreiben neuer Testinstanzen unbedingt eingehalten werden muss, wurden für diesen Fall nur eine Eingabemöglichkeit offen gelassen (Abbildung 11) und die Parameter der Reihe nach abgefragt. Die Eingabeaufforderung wird durch das über der Textbox

liegende Label vermittelt und dient dem Programm als "Lesezeichen", um den gelesenen Parameter einzuordnen.

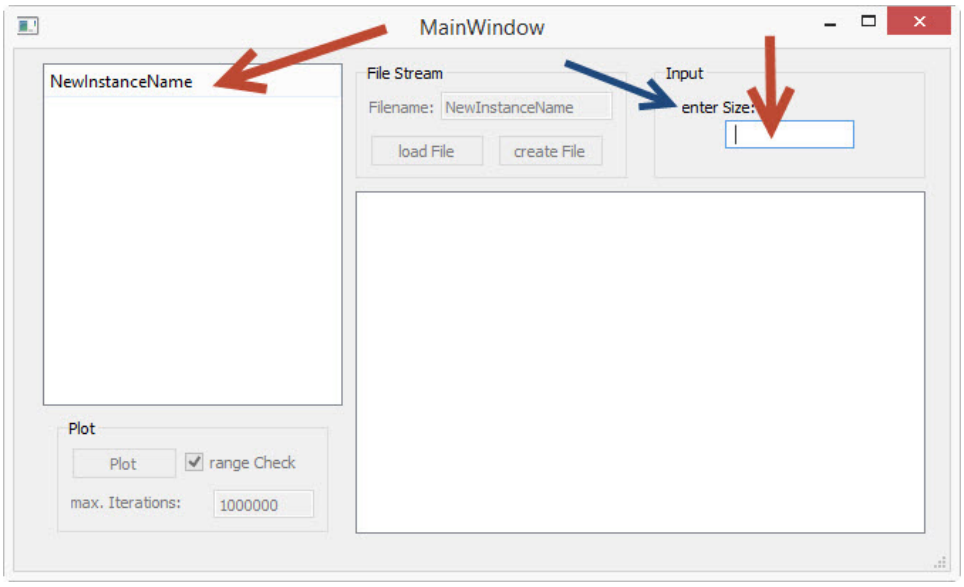


Abbildung 11: Nach Klick auf "create File" werden die neuen Parameter einzeln abgefragt

Nach erfolgreichem Einlesen aller Parameter wird die neue Instanz geladen und nebenstehend angezeigt.

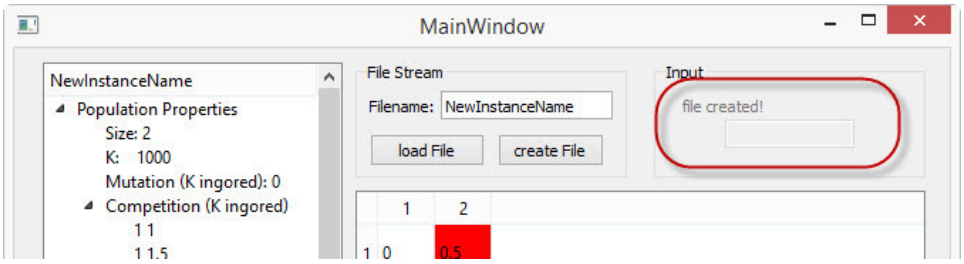


Abbildung 12: Nach Eingabe des letzten Parameters

5.4.3 Darstellung des Graphen

Nach dem Erstellen oder Laden einer Testinstanz ist es möglich, die Simulation zu starten. Das erkennt man an dem aktivierten "Plot"-Bereich (vgl. Abb. 8, zu 9, 10, 12) links unten.
Wie man schon auf vielen Abbildungen zuvor beobachten kann, gibt es im "Plot"-Bereich ein Eingabefeld mit dem Titel "max. Iterations:" und ein

Auswahlkästchen für "range Check".
Das Feld mit den "max. Iteraterions" gibt der Simulation vor, wie viele Sprünge zugelassen sind bis die gesammelten Populationsgrößen und die dabei vergangene Zeit gezeichnet werden.
Mit dem Kästchen "range Check" erlaubt man der Simulation in günstigen Situationen schon vor Erreichen der maximalen Iterationszahl abzubrechen. Eine Situation wird als günstig eingestuft, sobald die Entfernung jedes Merkmals zu seinem Gleichgewicht an einem Punkt gering genug gewesen und etwas zusätzliche Zeit verstrichen ist. Die frühzeitigen Abbruchbedigungen skalieren mit der bereits verstrichenen Zeit und dem verwendeten K.

Nach Absprache mit Loren Coquille und Martina Baar wurde von der graphischen Darstellung folgendes gewünscht:

- Man soll den zeitlichen Verlauf der Populationsgrößen beobachten können.
- Um die Prozesse besser analysieren zu können, soll es möglich sein, Stellen des Graphen näher betrachten zu können (zoom).
- Um Simulationen vergleichen zu können, sollen außerdem Ausschnitte als Bilder gespeichert werden können.
- Und natürlich soll das Programm bei der Berechnung nicht abstürzen.

Der letzte Punkt scheint vielleicht absurd, jedoch ist er bei der "Ereignisgesteuerten Programmierung" eine Hürde, die gemeistert werden muss.
Im Gegensatz zur Konsolenanwendung, die alle Abläufe in einer festen Reihenfolge bearbeitet, entstehen Konflikte, wenn während einer Berechnung auf ein laufendes Programm zugegriffen wird. Die Reaktionen können nicht nebeneinander laufen, weil sie oft auf dieselben Ressourcen zugreifen. Das zwangsläufige Ergebnis hat wahrscheinlich jeder schon erlebt:

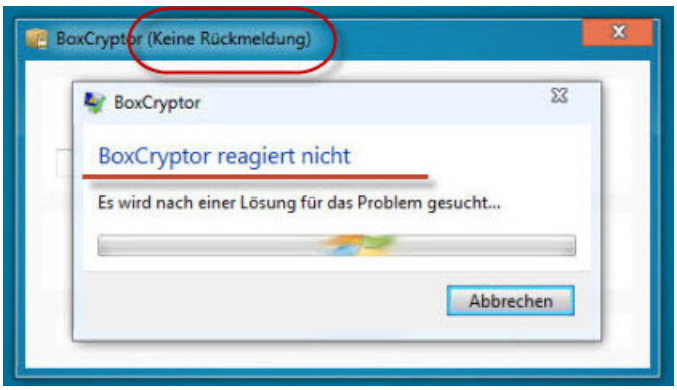


Abbildung 13: Hauptthread wurde überlastet

Die Lösung dieses Problems ist, die aufwändige Berechnung auf einen getrennten Prozess auszulagern. Hier kommt die Unabhängigkeit der Module besonders gelegen, denn man kann die Verwendung der Ressourcen leicht organisieren.
Die Auslagerung von Prozessen fällt unter den Begriff "Multithreading" und erstellt an geeigneter Stelle einen "Thread", um ihm Aufgaben zuzuteilen. Sobald der Thread seine Arbeit erfüllt hat (Simulation des Prozesses), sendet er ein Signal, das vom Programm als Ereignis (wie Benutzereingabe) interpretiert wird, um (in unserem Fall) das Zeichnen der Daten zu initialisieren.
Auf diese Weise nimmt das Programm (beide Fenster) weiter Benutzereingaben entgegen ohne seine Berechnungen unterbrechen zu müssen.
Das Drücken der "Plot"-Schaltfläche öffnet das "Plot-Fenster" (Abbildung 14).

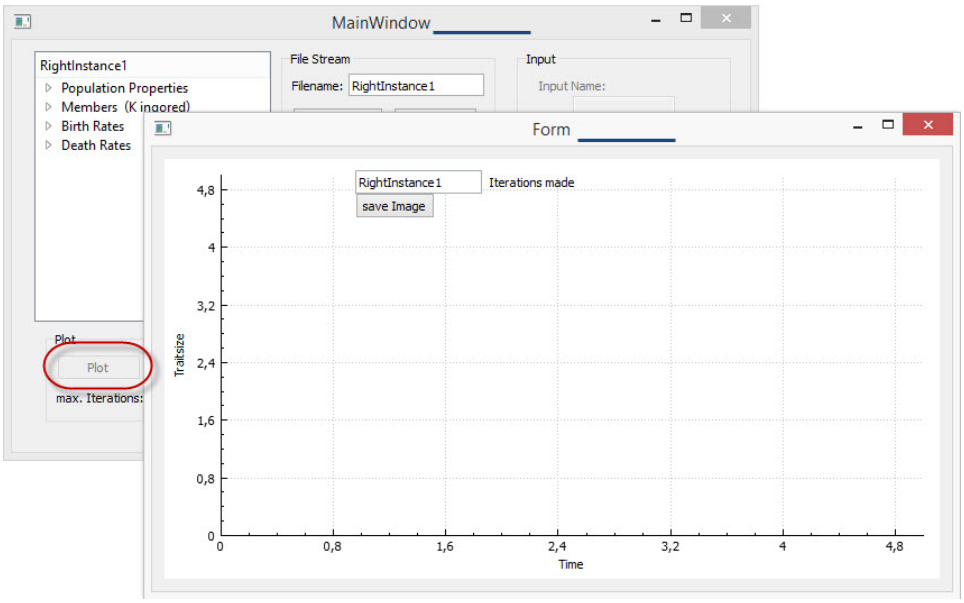


Abbildung 14: Start des PlotWindow

In blau sieht man, dass das Programm durch Übereinanderlegen während der Berechnung weiterhin ansprechbar ist statt keine Rückmeldung zu liefern (vgl. Abb. 13).
Wenn die Simulation einen günstigen Zustand erreicht oder die maximale Anzahl an gewünschten Iterationen absolviert hat, werden anschließend maximal 10mio Punkte auf dem Koordinatensystem zu Graphen verbunden (Abbildung 15).

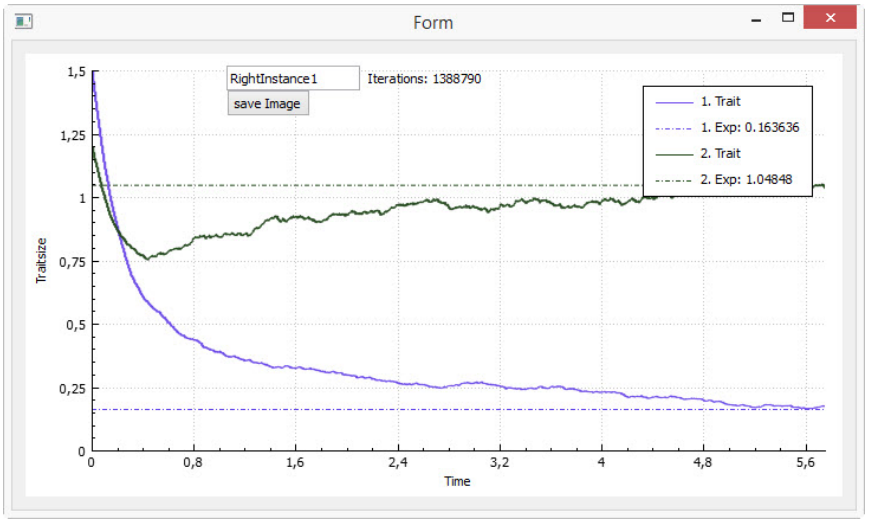


Abbildung 15: PlotWindow mit Dimorpher Population

Welche Optionen finden sich auf diesem Fenster?

- Natürlich kann man wie gewünscht in einem Koordinatensystem die zeitliche Entwicklung der Population verfolgen.
- Darüber hinaus sieht man in gestrichelten Linien die stabilen Zustände der Population und kann die Konvergenz dahin verfolgen.
- Außerdem ist bei aktiviertem "range Check" nicht klar, wie viele Iterationen (Sprünge) tatsächlich gemacht wurden, um den aktuellen Zustand zu erreichen. Zu diesem Zweck ist mittig im Bild ein Label "Iterations: ", worin der Wert 1388790 zu sehen ist.
- In der rechten oberen Ecke findet sich außerdem eine Legende der dargestellten Graphen. Dort sind im Wechsel die Merkmale mit ihrem erwarteten Gleichgewicht. "Exp" steht für "Expected" und kennzeichnet den Gleichgewichtszustand.
- Des weiteren findet sich eine Schaltfläche "save Image" und eine Textbox, in die man den gewünschten Bildnamen eintragen kann. Damit lässt sich das Bild mit Legende als .pdf und .jpg abspeichern.

Was man zwar nicht direkt in Abbildung 15 sehen kann, aber gut ausgearbeitet wurde, ist die Bewegungsfreiheit auf dem Bild.

- Mann kann in das Bild hineinzoomen.
- Die Messgitter werden beim zoomen automatisch angepasst.

- Auf dem Koordinatensystem kann man sich durch "Ziehen" bewegen.
- Das Fenster lässt sich durch Strecken skalieren, wobei sich der Graph automatisch anpasst.

Als Beispiel dient Abbildung 16, welche durch Heranzoomen und Bewegen sowie maximierte Fenstergröße (vgl. kleine Legende) erstellt wurde.

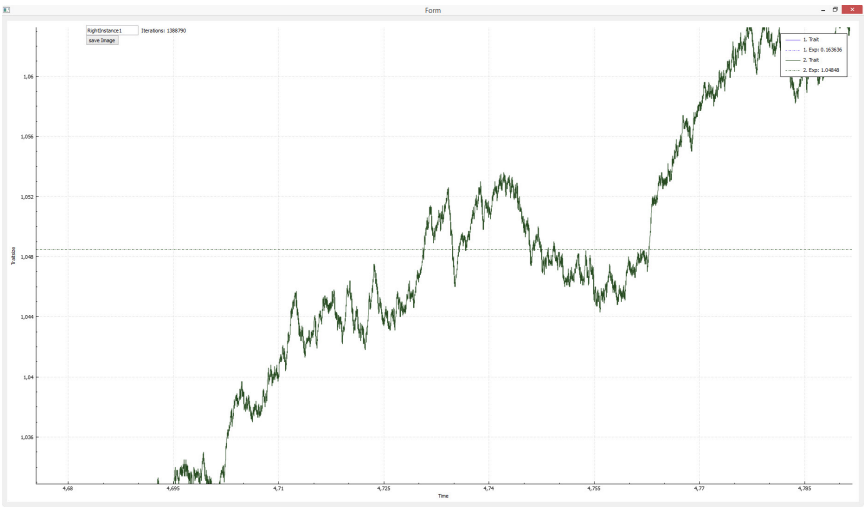


Abbildung 16: PlotWindow mit Dimorpher Population

Man kann sogar soweit skalieren, bis die Ereignisse lokal zählbar werden und der Plot eine Größe hat, in der das Messgitter eine Sprungmaschenweite hat:

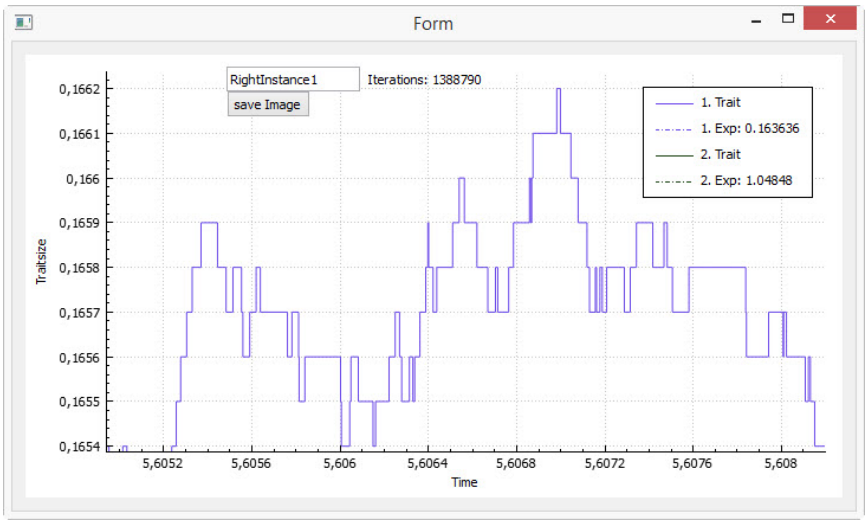


Abbildung 17: PlotWindow mit Dimorpher Population

6 Verhaltenstests und Korrektheit der Implementierung

Um beurteilen zu können, ob das erstellte Simulationsprogramm tatsächlich geeignet ist, ist es wichtig, die Korrektheit der Implementierung zu überprüfen. Dies gestaltet sich bei steigender Komplexität generell immer schwieriger (besonders bei zufallsbedingten Simulationen) und ist deswegen ein besonders interessantes Thema. Daher habe ich nach dem Prinzip der "testgetriebenen Entwicklung" (Test Driven Develeopment - TDD) gearbeitet [13].
Wie der Name schon verrät, geht es darum seine Entwicklung durch Tests, sogenannte "Unit Tests" anzutreiben. Diese Tests gewährleisten kontrollierte Bedingungen, um Änderungen und Funktionalität so souverän wie möglich zu gestalten.
Das Konzept des TDD kann durch die Zusammenarbeit dreier Punkte aus [10] gut beschrieben werden:

1. Produktiver Code sollte nur geschrieben werden, um einen fehlgeschlagenen Unit Test bestehen zu lassen.

2. Ein Unit Test sollte nur soweit entwickelt werden, bis er fehlschlägt.

3. Es sollte nur so viel produktiver Code geschrieben werden, um einen Unit Test bestehen zu lassen.

6.1 Unit Tests

Was ist ein Unit Test und was tut er?
Ein Unit Test ist nichts anderes als eine Funktion, die speziell dafür geschrieben wird, um ein implementiertes Verhalten bzgl. dem erwarteten Verhalten zu testen.
Auf diese Weise würde nach den obigen drei Punkten z.B. zunächst eine Funktion (ein Test) geschrieben werden, die unser Modell mit Parametern initialisiert und prüft, ob die Geburtenrate für diese Parameter gleich dem erwarteten Wert ist. Erst dann würde eine Geburtenraten-Funktion geschrieben werden, die den Test bestehen lässt.
Auf diese Weise lässt sich oft eine deutlich effizientere Programmstruktur modellieren, als man es ursprünglich geplant hatte. Das Planen ist natürlich trotzdem ein wesentlicher Schritt. Mehr Einzelheiten zur Effizienz finden sich in [10, The Bowling Game: An example of test-first pair programming].
In der Abbildung 18 ist ein Beispiel für eine Implementierung eines einfachen Tests, der prüft, ob alle Parameter korrekt aus der Datei in die Objekte geschrieben werden.
Dazu werden in einer Schleife erst 1.000.000 mal immer wieder neu die Todesraten berechnet und schließlich geprüft ob alle Raten trotzdem dem er-

warteten Wert entsprechen (roter Kasten). Das ist dank der modularen Implementierung, die in Kapitel 4 vorgestellt wurde, möglich.

```
void TraitEventManagerTest::verifyTotalDeathRate()
{
    qDebug()<<"verify total death rates ...";
    Manager.initWithFile("ValidateTests.txt");

    time_t start = clock();
    for(int k = 0; k < 1000000; ++k)
        Manager.calculateTotalDeathRates();
    qDebug()<<"elapsed time:"<< clock()- start<<"ms";

    QCOMPARE(Manager.Trait[0].TotalDeathRate, 30000.+500.);
    QCOMPARE(Manager.Trait[1].TotalDeathRate, 25000.+500.);
    QCOMPARE(Manager.Trait[2].TotalDeathRate, 40000.+500.);

    Manager.clearData();
}
```

Abbildung 18: UnitTest versichert korrekte Berechnung der Todesraten

Startet man nun das Testprogramm, so werden der Reihe nach alle implementierten Tests gestartet. Die Ausgabe enthält Erfolge, Fehlschläge und zusätzliche Debug-Ausgaben. Das ist in Abbildung 19 für die ersten Tests des BPDL Programms vorgeführt worden. In rot sieht man den zuvor erwähnten Test:

```
********* Start testing of TraitEventManagerTest *********
Config: Using QtTest library 5.2.1, Qt 5.2.1
PASS   : TraitEventManagerTest::initTestCase()
PASS   : TraitEventManagerTest::readAndClearStandardInput()
PASS   : TraitEventManagerTest::verifyTotalIntrinsicDeathRate()
PASS   : TraitEventManagerTest::verifyTotalCompDeathRate()
QDEBUG : TraitEventManagerTest::verifyTotalDeathRate() verify total death rates ...
QDEBUG : TraitEventManagerTest::verifyTotalDeathRate() elapsed time: 291 ms
PASS   : TraitEventManagerTest::verifyTotalDeathRate()
QDEBUG : TraitEventManagerTest::verifyTotalBirthRate() verify total birth rates ...
```

Abbildung 19: Ergebnisse einiger Tests

In blau sieht man, dass auch der praktische Aufwand gemessen wurde um Schwachstellen in der Implementierung aufzudecken (z.B. auch beim Ziehen von Zufallsvariablen).

6.1.1 Korrektheit durch Unit Tests

Warum sollten ein paar Tests für spezifische Situationen den Anspruch erheben Korrektheit des Programms zu gewährleisten?

Das Besondere an unserer Simulation ist, dass zufälliges Verhalten vorliegt und daher kleinere Fehler oft nicht so einfach aufgedeckt werden können. Z.B. würde es in einer dimorphen BPDL Simulation nicht auffallen, wenn die Mutation unberücksichtigt bleibt, solange die Wahrscheinlichkeit gering ist, oder wenn die Zeiten zwischen Sprüngen nicht richtig sind. Gerade deswegen wurde die "Testgetriebene Entwicklung" so interessant.
Die Korrektheit eines Programmkerns nachzuweisen kann sich als schwierig herausstellen, weil man die Korrektheit nicht für eine spezifische Instanz, sondern flexibel für alle nachweisen möchte. Doch ist es nicht notwendig die Flexibilität des Programms zu testen, wenn man stattdessen versucht, die Korrektheit jedes Arbeitsschrittes zu prüfen. Ein Arbeitsschritt erhebt in unserem Fall keinen Anspruch auf Komplexität. Tatsächlich sind unsere Arbeitsschritte alle möglichst einfach und erfüllen nur eine Aufgabe. D.h. die Korrektheit der Arbeitsschritte zu verifizieren lässt sich denkbar einfach mit einem Test abdecken.
Wenn nun jeder Arbeitsschritt seine Aufgabe nachweisbar richtig erfüllt, dann kann davon ausgegangen werden, dass durch richtige Verwendung der Arbeitsschritte das erwartete Verhalten eintritt. Die richtige Verwendung der Arbeitsschritte wird dadurch einfach nachzuweisen, dass wir in Kapitel 4 den Algorithmus für einen Evolutionsschritt vorgestellt haben. Dass dieser eine Schritt richtig ausgeführt wird, ist anhand des Pseudocodes leicht ersichtlich. Doch ist der nächste Schritt nichts anderes als der erste Schritt, nur mit leicht veränderten Startwerten. Wenn also die Startwerte korrekt sind, dann wird auch der nächste Schritt wieder richtig ausgeführt.
Auf diese Weise wird die Komplexität der Simulation vereinfacht auf die der Ablaufpunkte, die im Wesentlichen sehr einfach sind.
Was allerdings nicht geprüft wurde, ist der Umgang mit den erhobenen Daten innerhalb des "GUI"-Moduls. Aufgrund des zeitlichen Rahmens der Arbeit war es nicht mehr möglich, eine Methode zum Test von ereignisgesteuertem Code zu recherchieren und anzuwenden, sodass nur zwei von den drei in Abbildung 6 beschriebenen Modulen getestet werden konnten.

6.2 Unit Tests des Programmkerns

Hier werden Tests vorgestellt, mit denen die korrekte Berechnung der Raten verifiziert werden soll. Dazu wird die folgende Testinstanz mit drei Merkmalen ($X = \{x, y, z\}$) verwendet:

$c(\cdot,\cdot)$	x	y	z
x	2	1	0
y	0	2	0.5
z	0	2	2

	$b(\cdot)$	$d(\cdot)$	$n(\cdot)$	μ
x	10	5	100	0.1
y	10	5	100	0.1
z	10	5	100	0.1

Es wurden drei Merkmale gewählt, weil dies die minimale Anzahl ist, um die Auswirkungen von "inneren" Merkmalen auf Rand-Merkmale (und um-

gekehrt) zu testen.
Vor den Kerntests wurden noch Tests gemacht, die versichern, dass kein Fehler beim Lesen, Schreiben, Neulesen, Verändern etc. der Raten und Parameter auftritt. Diese sind z.B. in grün in Abbildung 19 zu sehen.

6.2.1 Raten

Das Testen der Todesraten läuft auf den finalen Todesraten-Test aus Abbildung 18 hinaus. Diese zeigt die Implementierung des entsprechenden Tests. Jede weitere Implementierung kann im Quellcode nachgesehen werden, jedoch werden hier nur heuristisch die Tests aufgelistet, um die Qualitätstests nachvollziehbar zu machen. In den Tests selbst werden stets jedes Merkmal und dessen Werte auf Korrektheit geprüft, obwohl folgend immer nur ein Merkmal vorgerechnet wird.

Intrinsische Todesrate:
Exemplarisch an y erwarten wir für diese Instanz folgende Berechnung:

$$\begin{aligned} \underline{\text{Erwartet}}: \quad & d(y) \cdot n(y) \\ & = 5 \cdot 100 = 500 \end{aligned}$$

$$\underline{\text{Testergebnis}}: \quad \begin{pmatrix} 500 \\ 500 \\ 500 \end{pmatrix}$$

Der Vergleich mit dem Testergebnis bestätigt somit das Verhalten der intrinsischen Todesraten-Berechnung.

Todesrate durch Wettbewerb:
Erneut an y erwarten wir für diese Instanz folgende Berechnung:

$$\begin{aligned} \underline{\text{Erwartet}}: \quad & n_0(y) \cdot \textstyle\sum_{w \in X} c(y,w) \cdot n_0(w) \\ & = 100 \cdot (2 \cdot 100 + 0.5 \cdot 100) = 25000 \end{aligned}$$

$$\underline{\text{Testergebnis}}: \quad \begin{pmatrix} 30000 \\ 25000 \\ 40000 \end{pmatrix}$$

Wie erwartet bestätigt sich das Ergebnis für y. Durch den großen Abstand zum Equilibrium beider Funktionen entsteht hier natürlich eine große wettbewerbliche Todesrate. Außerdem lässt sich beobachten, dass für das letzte Merkmal z die größte Rate berechnet wurde, was mit der Zeilensumme der Wettbewerbsmatrix begründet werden kann.

Totale Todesraten:
Hier muss nicht mehr viel vorbereitet werden. Wir haben bereits die intrinsische und wettbewerbliche Todesrate ausgerechnet.

$$\begin{aligned}
\underline{\text{Erwartet}}: \quad & D(y) = d(y) \cdot n(y) + n_0(y) \cdot \textstyle\sum_{w \in X} c(y,w) \cdot n_0(w) \\
& = 500 + 25000 = 25500 \\
\underline{\text{Testergebnis}}: \quad & \begin{pmatrix} 30500 \\ 25500 \\ 40500 \end{pmatrix}
\end{aligned}$$

Wie erwartet bestätigt sich das Ergebnis für y. In Abbildung 19 kann man die Ausführung aller drei Tests verfolgen.
In anderen Tests (des TSS Programms) wurden dieselben Prüfungen noch für unterschiedliche Instanzen gemacht.

Totale Geburtenraten:
Hier wird exemplarisch erneut das mittlere Merkmal y verwendet, welches durch die Mutation von beiden Seiten eine höhere Geburtenrate erwartet.

$$\begin{aligned}
\underline{\text{Erwartet}}: \quad & B(y) = (1-\mu) \cdot b(y) \cdot n_0(y) + \tfrac{\mu}{2} \cdot \left(\textstyle\sum_{w \in X} b(w) \cdot n_0(w)\right) \\
& = 10 \cdot 100 + 0.05 \cdot (10 \cdot 100 + 10 \cdot 100) = 1000 \\
\underline{\text{Testergebnis}}: \quad & \begin{pmatrix} 950 \\ 1000 \\ 950 \end{pmatrix}
\end{aligned}$$

Die Geburtenraten unterscheiden sich nur zwischen Rand und Innerem um 50, da wir die mutative Geburtenrate $0.05 \cdot 10 \cdot 100 = 50$ von links und rechts an Randmerkmalen natürlich nur einmal erhalten.

Totale Raten:
Dank der vorherigen Schritte lässt sich die Totale Merkmalsrate leicht berechnen:

$$\begin{aligned}
\underline{\text{Erwartet}}: \quad & B(y) + D(y) = \\
& 1000 + 25500, \text{ bzw. } \begin{pmatrix} 950 \\ 1000 \\ 950 \end{pmatrix} + \begin{pmatrix} 30500 \\ 25500 \\ 40500 \end{pmatrix} = \begin{pmatrix} 31450 \\ 26500 \\ 41450 \end{pmatrix} \\
\underline{\text{Testergebnis}}: \quad & \begin{pmatrix} 31450 \\ 26500 \\ 41450 \end{pmatrix}
\end{aligned}$$

Schließlich folgt daraus auch die Totale Ereignisrate:

Erwartet: $31450 + 26500 + 41450 = 99400$

Testergebnis: 99400

Das beendet die wesentlichen Tests zur Berechnung der Raten.

6.2.2 Ziehen einer Ereigniszeit

In diesem zweiten Teil der Tests werden Ereigniszeiten gezogen und mit dem Gesetz der großen Zahlen der Mittelwert mit dem erwarteten Mittelwert verglichen.
Es gab noch weitere Tests, die hier nicht erwähnt werden, weil sie bloß testen, ob der verwendete "Würfel" wirklich unabhängige Ziehungen macht und ob die Verteilung das tut, was sie soll.

Geburts- und Todeszeitpunkte:
Um diese Ziehung zu verifizieren wurden 1.000.000 Ereigniszeitpunkte mit denselben Raten gezogen und gemittelt.
Der Erwartungswert dieser Ziehung entspricht $\frac{1}{\mathrm{TotalEventRate}}$, was auch der Referenzwert für die Fehlerberechnung ist. Für das Testergebnis wird eine absolute Genauigkeit von 99.995% gefordert und der Fehler erreicht meistens einen Wert um 0.0008%.

TSS-Mutationszeiten:
Für die korrekte Ziehung der Mutationszeitpunkte in einer TSS-Simulation wird eine etwas andere Testinstanz verwendet, die nicht entscheidend ist, aber in allen TSS-Testinstanzen verwendet wird. Diese Instanz ist unter dem Namen "ValidateTSSTests.txt" im Programmordner zu finden und stellt sicher, dass die Fitness einen Wechsel der Merkmale garantiert.
Um die Mutationszeiten zu validieren wird 100.000 mal der folgende Ablauf durchgegangen:

1. Zunächst werden alle bis auf ein Merkmal getötet.
2. Das übrige Merkmal wird auf sein Gleichgewicht gehoben.
3. Dann wird eine Funktion aufgerufen, die im nächsten Kapitel vorgestellt wird und den Zeitpunkt der nächsten Mutation aus den Mutationsraten zieht.
4. Dieser Zeitpunkt wird zu vorhergehenden Zeiten summiert.

Nach der Schleife wird das Mittel gezogen und auf eine absolute Genauigkeit von 99.95% geprüft. Die Testergebnisse lagen in der Regel bei einem Fehler um 0.002%.

6.2.3 Veränderung der Population

Hier wird der letzte Bereich aus Abbildung 5, Ebene 1, vorgestellt.

Merkmal auswählen
Um zu verifizieren, dass die Merkmale korrekt den Ereignissen zugeteilt werden, wurde ein Histogramm der Wahlen erstellt. Genau gesagt wurde 100.000 mal entschieden, welches Merkmal ein Ereignis auslösen wird.
Wie schon in Kapitel 4 erwähnt, ist es sehr nützlich gewesen die Raten getrennt zu berechnen und zu speichern. Von diesem Vorteil kann jetzt Gebrauch gemacht werden, um zu unterscheiden welches Merkmal wie viele der 100.000 Ereignisse ausgelöst hätte.
Die getestete Genauigkeit war 99.95% und wie zuvor haben die Ergebnisse sie stets ausreichend überschritten.

Ereignistyp wählen
Die Idee dieses Tests ist praktisch identisch zum Vorherigen und wird nicht näher beschrieben, ist aber natürlich ausführlich im Quellcode vorhanden und hat ebenso bestanden.

Ereignis ausführen
Hier wurde getestet, ob das Ausführen eines gewählten Ereignisses auf ein gewähltes Merkmal korrekt abläuft. Dabei wurde lediglich geprüft, ob die Population bei einer Geburt wirklich anwächst und bei Tod bis zu einem Minimum von 0 sinken kann.

6.3 Unit Tests der Konvergenz

Hier wurde zunächst geprüft, ob das vom Programm ermittelte Gleichgewicht wirklich richtig berechnet wurde.
Schließlich wurde für dimorphe und monomorphe Populationen getestet, ob und wie gut die Konvergenz für wachsende K zum Gleichgewicht abläuft. Hierzu wurden 1.000.000 Sprünge des Prozesses mit $K = 10$ und $K = 10.000$ gemacht. Für $K = 10$ bewegt sich der Fehler um 0.1% und für $K = 10.000$ um 0.001%. Besonders bei großem K konnte man beobachten, dass der Fehler oft mit der dimorphen Population korreliert, d.h. beide vergleichbar gering oder hoch sind.

7 TSS-Prozesse

In diesem Kapitel geht es um TSS-Prozesse und wie eine Annäherung dieser durch unsere Simulation gelöst wurde.

7.1 Invasion

Wie schon im Rahmen der Fitnessfunktion angesprochen, beschreibt die Invasion den Vorgang, durch den ein neu auftretendes Merkmal (dank Mutation) ein bis dahin dominantes Merkmal verdrängt.
In Kapitel 3.2 haben wir weiterhin gesehen, dass der BPDL-Prozess für $K \to \infty$ gegen ein deterministisches System konvergiert. Jedoch soll unser TSS-Prozess ein nicht deterministischer Prozess sein, der eine monomorphe Population simuliert, die zu zufälligen Zeiten ihr dominantes Merkmal austauscht.
Um dies zu erreichen wird der Grenzwert nicht nur mit $K \to \infty$, sondern gleichzeitig $\mu \to 0$ gebildet. Auf diese Weise wird das sonst so deterministische Verhalten aus $K \to \infty$ durch weiterhin zufällige Ereignisse beeinflusst. Dabei wird der aus Kapitel 3.6 bekannte Bereich (16) für die Konvergenz von μ gewählt. Er garantiert, dass eine Invasion ausreichend Zeit zur Verdrängung hat, bis ein weiterer Mutant entsteht und nicht zu viel Zeit vergeht, sodass vor der nächsten Mutation die Population ausstirbt.
Des Weiteren wollen wir sicherstellen, dass sich auf lange Sicht nur ein Merkmal durchsetzt. Dies können wir erreichen, in dem wir ausnutzen, dass stets zwei Merkmale um Dominanz streiten, und die Aussagen aus Kapitel 3.5 (Dimorphes Gleichgewicht) auf mehrere Merkmale ausweiten.
Wenn für beliebiges $x \in X$ und für alle $y \in (X \backslash x)$ entweder $f(y,x) < 0$ oder $f(y,x) > 0$ und $f(x,y) < 0$, so ist sichergestellt, dass sich auf lange Sicht ein Merkmal in der Population durchsetzen wird und keine zwei Merkmale nebeneinander existieren können. Unter diesen Umständen können weiterhin die stabilen monomorphen Zustände, die in Kapitel 3.5 beschrieben wurden, angenommen werden, nur eben für mehrere Merkmale.

Welche Phasen durchläuft eine Invasion?
Damit ein Merkmal y ein dominantes Merkmal x verdrängen kann, muss nach obigen Erkenntnissen zunächst einmal $f(y,x) > 0$ und $f(x,y) < 0$ gelten.
Wie aus [8] beschrieben, lässt sich die Invasion in drei Teilen beschreiben.

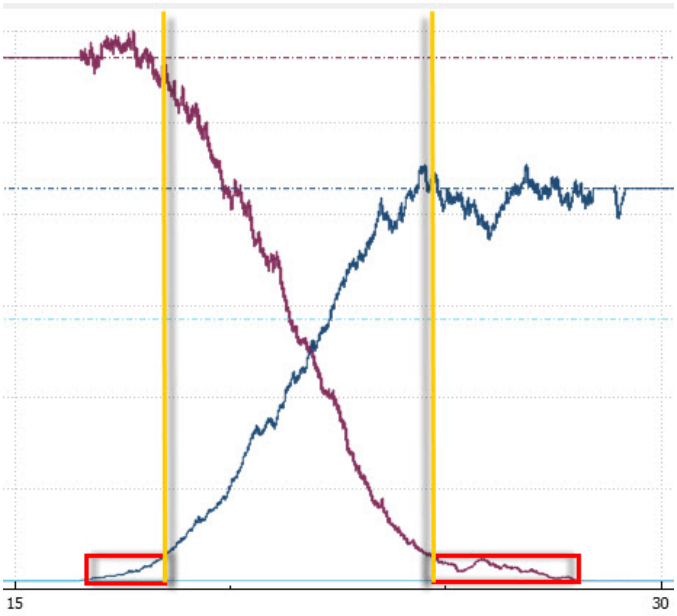
1. Fixierung: Zu Beginn einer Invasion werden die Individuen des Merkmals x kaum von y beeinflusst, da $c(x,y)n(y)$ unbedeutend klein ist. Jedoch ist dieser Zustand zeitlich durch die positive Wachstumsrate des Merkmals y für x im Gleichgewicht $\bar{n}_x$ begrenzt. Ab einer bestimmten Populationsgröße ε, kann der Einfluss des Eindringlings x von y nicht mehr ignoriert werden. Die Dauer, bis ein erster Mutant das

Merkmal zu diesem relevanten Niveau bringt, hat die Größenordnung $\log K$.

2. Invasion: Hier ereignet sich der Fall des dominanten Merkmals x und der Anstieg des neuen Merkmals y. Dank $f(x, y) < 0$ und $f(y, x) > 0$ beobachten wir die Konvergenz gegen das neue Gleichgewicht $(0, \bar{n}_y)$.

3. Aussterben: Hier greift der umgekehrte Fall der Fixierung. Sobald $n(x) < \varepsilon$ geworden ist, kann zunächst der Einfluss von x auf y vernachlässigt werden. Dank der negativen Wachstumsrate wird das Merkmal x in einer Zeit von der Ordnung $\log K$ aussterben.

Beispielhaft kann man das an folgenden Abbildungen beobachten:



In rot sind die Fixierungsphasen gekennzeichnet und in orange erkennt man die Zeitpunkte, an denen die Übergänge beobachtbar sind.

Abbildung 20: Eine Invasion für K = 1000

Man kann in dieser Stelle bereits erahnen, dass die Fitnessfunktion bzw. die Wachstumsrate eines Mutanten die Möglichkeit bietet, Aussagen über die Invasionswahrscheinlichkeit zu machen. Und tatsächlich erhalten wir aus [6], dass die Invasionswahrscheinlichkeit mit $K \to \infty$ gegen

$$\frac{[f(y, x)]_+}{b(y)}$$

konvergiert.

7.1.1 Praktische Umsetzung

Normalerweise lassen sich die Parameter einer Simulation so wählen, dass bei mindestens drei Merkmalen ein Kreislauf der Invasionen erzwungen werden

kann. D.h. es wird nicht dazu kommen, dass ein Merkmal nicht verdrängt werden kann, also: $x \xleftarrow{verdrängt} y \xleftarrow{verdrängt} z \xleftarrow{verdrängt} x \ldots$
In unserem Modell jedoch haben wir nur Mutationen zu den Nachbarn erlaubt. Diese Einschränkung verhindert den Kreislauf. Wenn ein Merkmal y ein x verdrängt, so kann es nicht mehr von selbigem x verdrängt werden, sondern muss vom anderen Nachbarn verdrängt werden.
Auf diese Weise wandert die Vorherrschaft in der Population, bis ein Merkmal dominant ist, welches keine fitteren Nachbarn hat oder am Rand angekommen ist, was zwangsläufig wieder dem vorherigen Fall entspricht.
Die Darstellung der Fitnessmatrix im Programm lässt sich an folgendem Bild leicht nachvollziehen:

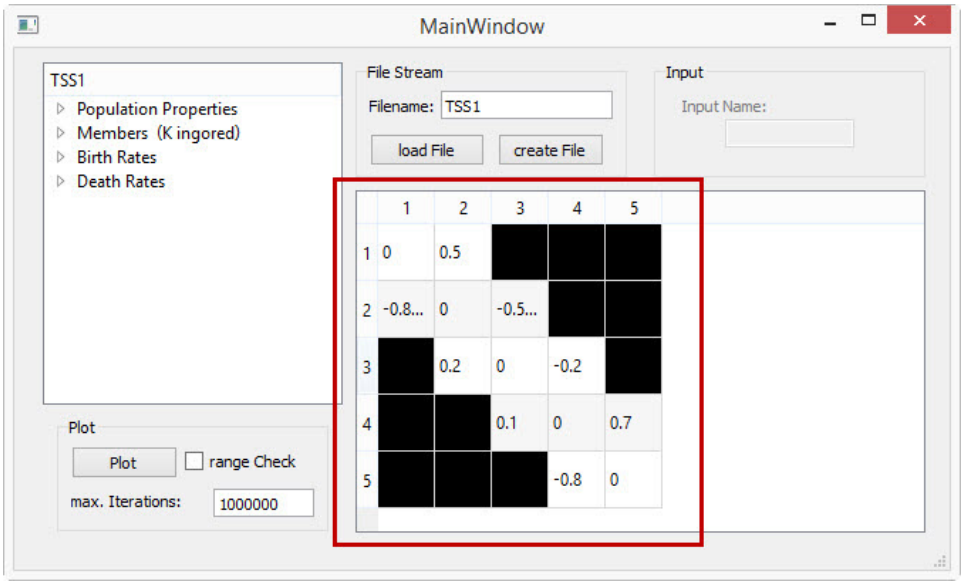


Abbildung 21: Fitness Bandmatrix

Da die Fitness nicht die einzige nützliche Information bei Invasionen ist, wurde dem Programm noch die Möglichkeit gegeben, farblich darauf hinzuweisen, ob Invasion möglich und ob sie besonders wahrscheinlich ist.
Dazu folgendes Bild:

Abbildung 22: Fitness Matrix mit roten und grünen Akzenten

Dabei werden drei Farben unterschieden.

Rot werden beide Merkmale gefärbt, falls wir einen Zustand der Koexistenz erwarten. In diesem Fall sind $f(x,y)\&f(y,x) > 0$.

Weiß wird ein Merkmal gefärbt, wenn es verdrängen oder verdrängt werden kann.

Grün wird ein Merkmal gefärbt, wenn die Verdrängungswahrscheinlichkeit über 60% ist.

7.2 Optimierung

Einfach den BPDL-Simulator zu verwenden, um diesen Grenzwertprozess zu approximieren, würde darin resultieren, dass sehr viel Rechenzeit im Gleichgewicht eines Merkmals verbracht werden würde.
Die Beobachtung des Prozesses im Gleichgewicht ist nicht besonders interessant und es lässt sich dabei nicht gut nachvollziehen, wann vielleicht Mutanten Invasionsversuche gestartet haben, die sich nicht behaupten konnten.
Diese Simulation wurde bereits in Abbildung 3 vorgestellt.
Alternativ wird hier eine andere Variante dieser Simulation vorgestellt. Diese nutzt eine lineare Interpolation als eine Optimierung um die Übersichtlichkeit, Analysefähigkeit und Laufzeit deutlich zu verbessern.
Diese lineare Interpolation macht es notwendig, stets zu wissen in welcher der drei Phasen der Prozess sich zur Zeit bewegt. Sobald er die Aussterben-Phase verlassen hat und schließlich der Prozess sein Equilibrium erreicht,

wird eine Interpolation des Prozesses bis zu seiner nächsten Mutation durchgeführt.
Die Sprünge bis zum nächsten Mutanten werden nicht berechnet, stattdessen wird anhand der Geburtenrate der toten Nachbarn eine Mutationsrate zusammengefasst, welche uns die Zeit für die nächste Mutation liefert. Danach wird nach dem selben Vorgehen, wie in Kapitel 4 beschrieben, entschieden, welcher Nachbar die Mutation ausgelöst hat.

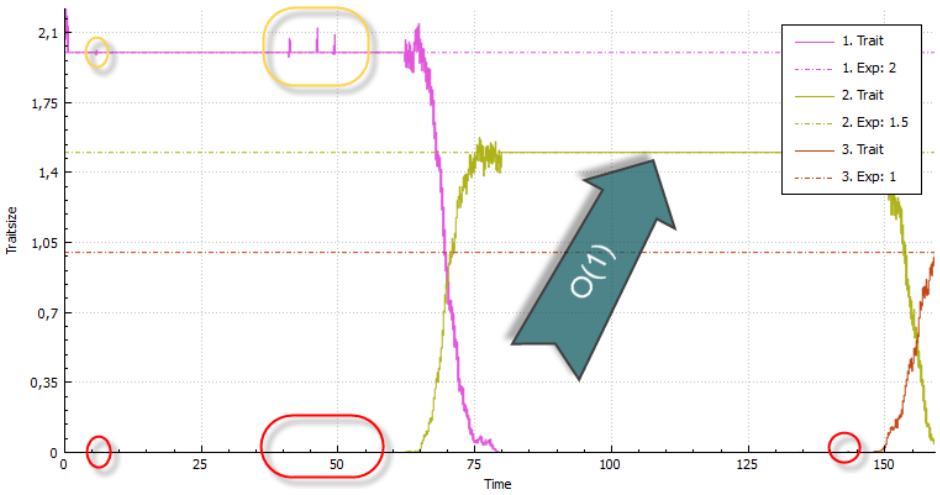
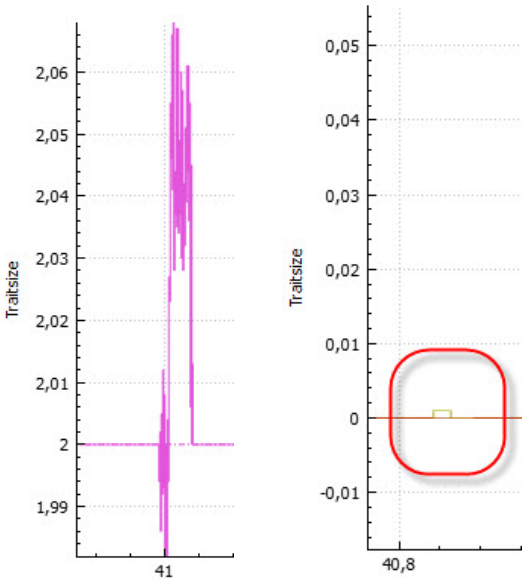


Abbildung 23: optimierte Simulation einer TSS-Approximation

Dass sich dadurch die größte Rechenzeit auf $O(1)$ verkürzt und die Übersichtlichkeit verbessert, ist sehr deutlich und war zu erwarten. Die Analysemöglichkeiten jedoch verbessern sich dank des großen Ratenunterschiedes der Merkmale. Der eindringende Mutant bietet dem neuen Merkmal nur eine sehr geringe Ereignisrate. Während dieser Zeit sind große Bewegungen im dominanten Merkmal zu erwarten. Diese Beobachtung lässt sich leicht in Abbildung 23 nachvollziehen, wo man in rot niemals einen Mutanten entdecken würde, jedoch dank der Ausschläge in der gelben Markierung ist es einfach die Mutanten aufzuspüren.
Folgend werden beide Effekte gegenübergestellt:



Dank der Beweglichkeit auf dem Koordinatensystem ist es möglich, die Entwicklung des Mutanten näher zu betrachten (nebenstehendes Bild). Oder man kann die Invasionsversuche aufspüren und auf eine deutliche Skala bringen (unteres Bild).

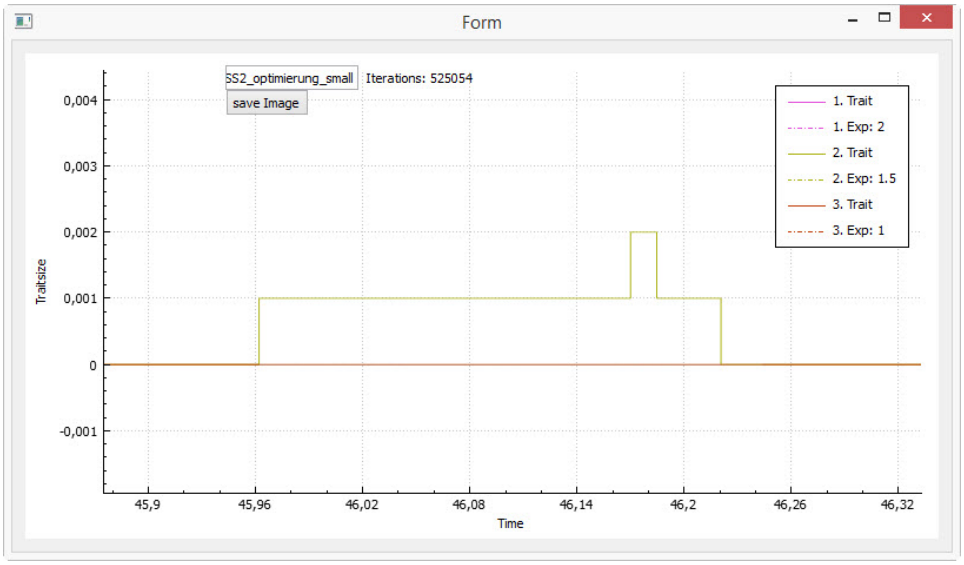


Abbildung 24: Dritte Mutation von Links

7.3 Implementierung

Welche Form die Implementierung und der Algorithmus annehmen, wurde bereits in Kapitel 4 ausreichend erläutert. Deswegen wird hier nur kurz erklärt, wie sich der Algorithmus ändern würde. Die tatsächliche Implementierung weicht etwas ab und verwendet technisch unwichtige Details.

Die Implementierung benötigt jetzt, bevor sie einen gewöhnlichen Sprung macht, zusätzlich eine Überprüfung, ob wirklich ein Prozesssprung oder eine Interpolation ausgeführt werden soll.
Dazu wird nach jedem Sprung eine Abfrage "isNear()"gestartet, die prüft, ob es nur ein lebendes Merkmal gibt und ob dieses im Gleichgewicht ist:

Algorithm 14 isNear()

Ensure: Bedingungen für eine Interpolation

```
1: for  i = 0 < n-1 do
2:     if i ≠ ChosenTrait & Members[i] > 0 then
3:         return false;
4:     end if
5: end for
6: if |getKMembersOf[i] − Expected[ChosenTrait]| > 0.5/K then
7:     return false;
8: end if
9: return true;
```

Hier sorgt Zeile 2 in der Schleife dafür, dass außer dem zuletzt aktiven kein weiteres Merkmal aktiv sein könnte.
Danach wird in Zeile 6 sicher gestellt, dass der Abstand zum Gleichgewicht minimal ist. Das ist notwendig, da der minimale Abstand nicht immer 0 ist. Falls das Gleichgewicht einen Wert ergibt, der kein Vielfaches von $\frac{1}{K}$ ist, so kann das Gleichgewicht nicht durch den Prozess angenommen werden. In diesem Fall wird der Prozess jedoch stattdessen einen Sprungpunkt im Radius $\frac{0.5}{K}$ um das Gleichgewicht haben.
Als Abbruchkriterium wird nicht mehr einfach die maximale Anzahl erlaubter Sprünge verwendet, sondern auch ein Interpolationspunkt des Merkmals, an dem keine fitteren Nachbarn mehr vorhanden sind. Die Implementierung des Abbruchkriteriums sollte intuitiv sein und wird deshalb nicht weiter beschrieben.

Aufwand

Eine Besonderheit lässt sich zum Schluss noch betonen. Durch die Interpolation ist der Aufwand der Berechnung nicht mehr allein von K abhängig, sondern besonders von der Anzahl der erwarteten Mutationen und der Anzahl der erwarteten Invasionen.

8 Schlusswort

Schon früh war klar, dass noch viele Möglichkeiten offen bleiben werden, wie die Simulation erweitert werden kann. Einige größere Anpassungen, von denen auch welche nach dem Erstellen dieser Arbeit umgesetzt werden, werden hier kurz vorgestellt:

1. Dank der hier verwendeten Art der Implementierung ist das Erweitern des Programmkerns auf alternative Modelle denkbar einfach. So wäre es sehr einfach die Mutationswahrscheinlichkeit nicht nur mit gleichen Teilen auf die Nachbarn zu verteilen, sondern eine individuelle Wahrscheinlichkeit auf beliebige Merkmale. Dazu müssen nur die Einlesefunktion der Parameter und die Berechnung der mutativen Geburtenrate verändert werden. Alle anderen Teile des Programms wären davon unbeeinflusst. Insbesondere garantieren die bereits geschriebenen Tests, dass bei einer Veränderung am Programmkern keine unbemerkten negativen Effekte auftreten. Natürlich wäre damit das Abbruchkriterium der TSS-Simulation hinfällig, aber es würde keine Fehler verursachen.

2. Weiterhin kann die Darstellung des Graphen im TSS-Simulator optimiert werden. So kann z.B. die Zeit zwischen den Mutationen gestaucht (z.B. logarithmisch) werden, so dass es einfach wird, nur die Übergänge und Mutationen genauer zu betrachten.
   Außerdem könnte es eine Möglichkeit geben, die Bilder des Programms als Vektorgraphiken abzuspeichern oder an einer Stelle des Prozesses mit geänderten Parameter weiter zu simulieren.

3. Man könnte einen Popupcontainer implementieren, der zusätzliche Anzeigedaten für den Plot ermöglicht. Z.B. könnte man so die Legende mit weiteren Parametern, wie μ_K, K oder anderen, füllen. Außerdem könnte er eine Option enthalten, die Anzahl der relevanten Invasionsversuche eines Merkmals anzuzeigen.

Da die Implementierung des Programms eine flexible Weiterentwicklung ermöglicht, wurde diese Liste stetig erweitert.

8.1 Danksagung

An dieser Stelle möchte ich meinen tiefen Dank bei all jenen aussprechen, die zum Gelingen dieser Bachelorarbeit beigetragen haben. Ein ganz besonderer Dank geht an Dipl. Martina Baar und Dr. Loren Coquille für ihre regelmäßige Betreuung, der ich die präzise Umsetzung der Simulation verdanke, ihre Geduld, der ich die reibungslose Zusammenarbeit verdanke und ihre Bereitschaft stets mit Rat erreichbar zu sein. Ohne ihre Aufsicht hätte ich nie so viele Erfolge während der Bachelorarbeit erzielen können.

Literatur

[1] Nicolas Fournier and Sylvie Méléard. A microscopic probabilistic description of a locally regulated population and macroscopic approximations. Ann. Appl. Probab., 14(4):1880–1919, 2004.

[2] B. Bolker and Pacala. Spatial moment equations for plant competition: Understanding spatial strategies and the advantages of short dispersal. The American Naturalist, 153:575–602, 1999.

[3] Benjamin Bolker and Stephen W Pacala. Using moment equations to understand stochastically driven spatial pattern formation in ecological systems. Theoretical Population Biology, 52(3):179 – 197, 1997.

[4] Ulf Dieckmann and Richard Law. The dynamical theory of coevolution: a derivation from stochastic ecological processes. Journal of Mathematical Biology, 34(5-6):579–612, 1996.

[5] Nicolas Champagnat and Sylvie Méléard. Polymorphic evolution sequence and evolutionary branching. Probability Theory and Related Fields, 151(1-2):45–94, 2011.

[6] Nicolas Champagnat. A microscopic interpretation for adaptive dynamics trait substitution sequence models. Stochastic Processes and their Applications, 116(8):1127 – 1160, 2006.

[7] Stewart N Ethier and Thomas G Kurtz. Markov processes: characterization and convergence, volume 282. John Wiley & Sons, 2009.

[8] Silke Völk. Stochastische modelle der populationsdynamik. Bachelorarbeit, August 2010. Bachelorarbeit.

[9] Mark I Freidlin, Joseph Szücs, and Alexander D Wentzell. Random perturbations of dynamical systems, volume 260. Springer, 2012.

[10] Robert C Martin. Clean code. A Handbook of Agile Software Craftsmanship. PrenticeHall, page 464, 2008.

[11] Ulrich Breymann. Der C++-Programmierer. Carl Hanser Verlag GmbH & Co. KG, 2011.

[12] Robert Cecil Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[13] Robert C Martin. Professionalism and test-driven development. Ieee Software, 24(3):32–36, 2007.