Simulation normalisierter BPDL Prozesse und TSS Prozesse

Boris Prochnau

Geboren am 22. Dezember 1989 in Tartu

31. Juli 2014

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Anton Bovier,
Dipl. Martina Baar und Dr. Loren Coquille

Institut für Angewandte Mathematik

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Inhaltsverzeichnis

1 Einleitung

Ziel dieser Bachelorarbeit ist die Entwickelung eines Programms, das die Dynamik einer Population simulieren kann. Darüber hinaus wird die Erweiterung zu einem zweiten Programm vorgestellt welches besonders interessante Prozesse, genannt "Trait Substitution Sequences", simulieren kann.

Alle Simulationen basieren auf dem Modell, dass jedes Lebewesen einer Population (z.B. Pflanzen oder Zelle) ein Merkmal trägt. Bestimmt wird ein Merkmal durch Todesraten und einer Fortpflanzungsrate. Die Todesrate besteht aus einer natürliche Todesrate und eine durch Wettbewerb mit jedem anderem Individuum. Das heißt die wettbewerbliche Todesrate steigt mit der Anzahl der konkurrierenden Individuen.
Schließlich ist es jedoch die Entwicklung der Population und nicht der Individuen die simuliert werden soll. Deshalb kann man im simulierten Prozess zwar Tode und die Geburten verfolgen, aber nicht welches Individuum dieses Ereignis auslöst. Der Übergang zu dieser Sichtweise wird näher im 2. Kapitel beschrieben.

Das Programm sollte Parameter einlesen können und eine Oberfläche bieten auf der die Simulation graphisch angezeigt wird. Tatsächlich wird das Programm mehr bieten und flexible Erweiterungsmöglichkeiten beinhalten.
Die graphische Darstellung des Prozesses soll die Möglichkeit bieten beobachten zu können ob sich ein Merkmal unter anderen Durchsetzten kann, es einen stabilen Zustand annimmt oder sicher dem Tod entgegen strebt.
Näheres zum erwarteten Verhalten der Simulation fehlt noch

Im letzten Teil werde ich noch kurz die Erweiterung auf TSS Prozesse vorstellen. Darin soll nicht nur das besonders interessante Verhalten vom Wechsel des dominanten Merkmals behandelt werden, sondern es wird eine verbesserte Laufzeit durch Interpolation vorgestellt die eine effiziente Simulation trotz sehr großer Zeit und besonders präziser Betrachtung von Aktionsreichen Gebieten anbietet.

2 Modell

Das verwendete Model wurde in [1, 2, 3] 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 wird es sein zwei spezielle BPDL-Grenzwert-Prozesse simulieren zu können.

2.1 Grundlagen

Sei X der endliche diskrete Raum der Merkmale. Jedes Individuum hat genau ein solches Merkmal $x \in X$ und ist vollständig durch dieses charakterisiert. 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 [4].
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 [5, 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 entscheiden ist um die Sichtweise von der Ebene des Individuums auf die der gesamten Population zu heben.
Diese Todes und Fortpflanzungs- Ereignisse eines Individuums haben feste Raten die das dazugehörige Merkmal beschreiben.

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

$d(x)$: Ist die natürliche Todesrate.

$c(x, y)$: Ist die Todesrate durch Wettbewerb die ein Individuum y auf x ausübt. Diese Interpretation orientiert sich an [4], während u.a. in [6] ein symmetrischer Wettbewerbskern verwendet wird.

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

Schließlich lassen sich durch 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 Geburtenraten 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:

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

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 Ereignis Rate (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 zwei 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 weitere Berechnungen machen zu müssen.

2.2 BPDL Prozess

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

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

beschrieben. Sie 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 $\mu(x)$. Und der Mutant hat dann 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 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 Model

angepasster 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 oft bei der Simulation sichtbar sein sollen. Zunächst wird dabei die Normalisierung eingeführt die es auch einfacher macht 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 [5] 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 das selbe 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 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

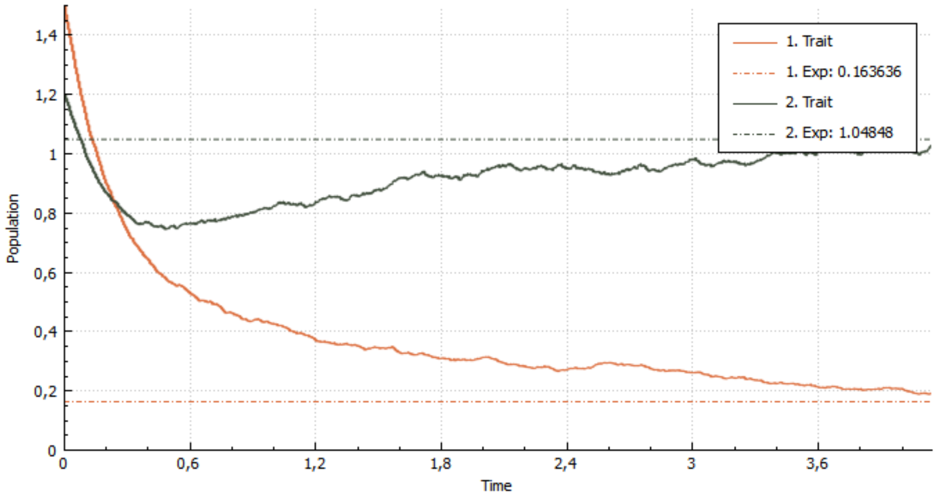

Abbildung 2: LPA Normalisierung mit K=10000

Was in beiden Abbildungen jetzt schon auffällt ist dass die Population für kleine K fast sofort aussterben würde, weil sich die Merkmale auf sehr geringen Populationen in ein 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:

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

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 $\nu_t^K \xrightarrow{K} n_t$. Ein solches deterministisches System muss folgende Differentialgleichung erfüllen:

$$\begin{pmatrix} \dot{n}(x) \\ \vdots \end{pmatrix} = \begin{pmatrix} n(x) \cdot (b(x) - d(x) - \sum_{y \in X} c(x,y) \cdot n(y) \\ \vdots \end{pmatrix}, \qquad (3)$$
$$n(0) = n_0$$

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

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

dementsprechend ist:

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

Diese Übersetzungen lassen sich leicht anhand des Generators nachvollziehen, wobei unser Generator (1), nur ohne Mutation, das selbe ergeben soll wie:

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

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 eigentlich Theorem.

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

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

und es existiert ein $M_V > 0$, so dass

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

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

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

Dann gilt für alle $t > 0$,

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

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

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

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

Endliche Raten:

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 haben wir stets eine endliche Summe über endliche Raten, welche natürlich wieder endlich ist.

Lipschitz-Stetigkeit:

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

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

Zunächst wählen wir $\varepsilon := |n_1 - n_2| = \sqrt{|n_1(x) - n_2(x)|^2 + |n_1(y) - n_2(y)|^2}$, daraus folgt:

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

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

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

So folgt wegen

$$\begin{aligned} |F(n_1) - F(n_2)| &= \sqrt{(F(n_1)_1 - F(n_2)_1)^2 + (F(n_1)_2 - F(n_2)_2)^2} \\ &\leq \sqrt{(\varepsilon \cdot c)^2 + (\varepsilon \cdot c)^2} \\ &= \sqrt{2} \cdot \varepsilon \cdot c < \infty \Rightarrow \text{Behauptung} \end{aligned} \tag{11}$$

Also bleibt nur noch (10) zu prüfen. Dabei benötigen wir, dass $|n_1(x)+n_2(x)|$ beschränkt ist. Das ergibt sich aus der Voraussetzung, dass V kompakt ist und wir $n_1, n_2 \in V$ wählen. Diese Wahl ist für unser Modell sinnvoll, weil unsere Population durch eine Population ohne Todesraten zu jedem Zeitpunkt endlich beschränkt wäre.

Für F_1 und F_2 ist dabei das Vorgehen analog, daher wird nur F_1 vorgestellt:

$$\begin{aligned}
|F(n_1)_1 - F(n_2)_1| &= |(n_1(x) - n_2(x))(b(x) - d(x)) - ((n_1(x))^2 - (n_2(x))^2) \cdot c(x,x) \\
&\quad - ((n_1(y))^2 - (n_2(y))^2) \cdot c(x,y)| \\
&\leq |\underbrace{(n_1(x) - n_2(x))}_{\leq \varepsilon}(b(x) - d(x))| \\
&\quad + |(n_1(x) - n_2(x))(n_1(x) + n_2(x)) \cdot c(x,x)| \\
&\quad + |(n_1(y) - n_2(y))(n_1(y) + n_2(y)) \cdot c(x,y)| \\
&\leq \varepsilon \cdot (|b(x) - d(x)| + |\underbrace{n_1(x) + n_2(x)}_{\text{beschränkt}}|c(x,x) + |n_1(y) + n_2(y)| \cdot c(x,y)) \\
&\leq \varepsilon \cdot (c_1 + c_2 \cdot c(x,x) + c_3 \cdot c(x,y)) \\
&= \varepsilon \cdot c
\end{aligned}$$

wie schon erwähnt folgt durch analoges Vorgehen für y, dass (9) für unser Modell gilt.
Tatsächlich kann für Fälle mit mehr als 2 Merkmalen durch analoges Vorgehen die selben Abschätzungen gemacht werden die alle zum gleichen Ergebnis führen.
Schließlich folgt für alle Fälle durch (11) die Lipschitz-Stetigkeit.

[7, Kapitel 11 - **(2.3)**] bleibt dem Leser überlassen, folgt aber aus [7, Kapitel 6 - (2.1)].
Und (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. □

3.2 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 indem 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.

3.3 Die Fitnessfunktion

Spätestens jetzt wird die Fitnessfunktion interessant:

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

Die Fitness-Funktion 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 können ($b(x) - d(x) > 0$) und dazu noch dem Konkorruenzdruck widerstehen können ($b(x) - d(x) \underline{- c(x, y)\bar{n}_y} > 0$).
Hierbei wird $c(x, x)$ nicht berücksichtigt weil es im Grenzwert mit K immer geringeren Einfluss hat, und 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 Konkurrenz der nahezu konstanten $c(x, y)\bar{n}_y$.
Die Fitness-Funktion 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. 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 wird in [6] eine Konvergenz für $K \to \infty$ von einer Konvergenz gegen

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

gesprochen.

<<< ↓ Todo ↓ >>>

Begründet wird das durch [...]

3.4 Dimorphes Gleichgewicht

Wir wissen mittlerweile dass falls $n_0^K \to n_0$, eine dimorphe Population $\nu_0^K = n_0^K(x)\delta_x + n_0^K(y)\delta_y$ für $K \to \infty$ gegen ein deterministisches System $(n(x), n(y))$ konvergiert. Angenommen es gibt wieder keine Mutation, dann 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 sieht man bereits leicht dass $(\bar{n}(x), 0)$, $(0, \bar{n}(y))$ und $(0, 0)$ stabile Zustände sind. Jedoch gibt es in diesem Fall auch einen Zustand indem 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 $(\bar{n}_x, 0)$ ist 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

---

$<<< \quad \downarrow$ Todo $\downarrow \quad >>>$

Die Idee des Beweises ist [...]

Das folgende Bild zeigt sowohl die Konvergenz gegen das eben berechnete Gleichgewicht, als auch das deterministische Verhalten für sehr große K:

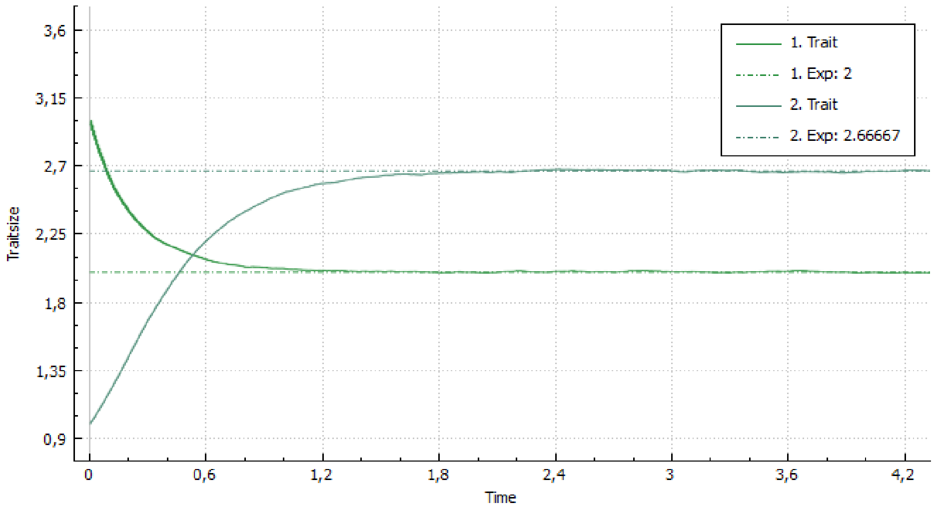

Abbildung 3: Konvergenz mit K=100000 und $15 \cdot 10^6$ Sprüngen

3.5 Der TSS Grenzwertprozess

Genau wie bei der LPA-Normalisierung ergeben sich 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 springen falls $f(y,x) > 0$ und $f(x,y) < 0$.
In diesem Fall haben wir zu jeder festen Zeit höchstens 2 konkurrierende Merkmale haben. Diese Eigenschaft vereinfacht die Analyse des Prozesses sehr und ist der Wahl von μ_K zu verdanken.

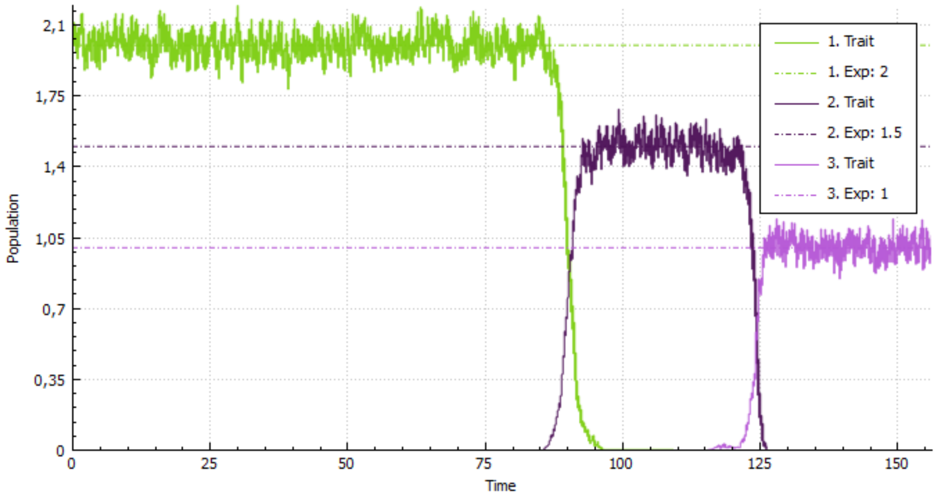

Abbildung 4: TSS Prozess mit: K = 1000 und $4 \cdot 10^6$ Sprüngen

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

Antwort: Dank Freidlin and Wenzell [8] erwarten wir dass unsere dominante Spezies eine Zeit von der Ordnung $\exp(cK)$ im Gleichgewicht bleibt. Schließlich können wir so kontrollieren wie lange uns eine dominante Spezies die für eine mutative Geburt in Frage kommt erhalten bleibt. Und dadurch dass die Mutationen exponentiell verteilt sind benötigt man ein eine Rate $\mu_K \gg \frac{1}{e^{cK}}$ um eine Zeit von $\exp(cK)$ nicht zu überschreiten, was die obere Schranke rechtfertigt.
Um die untere 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 so viel 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 es mit positiver Wahrscheinlichkeit eine Invasion auslösen (näheres dazu im Kapitel 7 - TSS Prozesse). 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 [4].

4 Simulation

In diesem Kapitel wird der Kern der Simulation algorithmisch näher untersucht. Dieser Kern besteht im Wesentlich 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 werden und im 5. Kapitel weiter verwendet werden.
Die hier verwendete Vorgehensweise unterscheidet sich von der aus [5] weil...

4.1 Implementierung

Die Simulation durchläuft mehrere Schritte bis ein vollständiger Sprungs 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 (6) 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 Implementation gegeben. So kann hier z.B. angenommen werden dass man mit der Variable "Members[i]" Zugriff auf die Anzahl der Individuen des i-ten Merkmals hat, was im jedoch Code komplexer realisiert werden musste.

4.1.1 Raten berechnen

Zunächst müssen wir die Raten wissen nach der die exponentiellen Uhren gestellt werden bevor ein Merkmal und Ereignis ausgewählt werden kann.
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] $\cdot$ Members[i]

Diese Funktion addiert die Wettbewerbs-Todesrate zur aktuellen Todesrate.

Algorithm 2 addTotalCompDeathRateOf(TraitIndex: i)

Ensure: addiert zur Todesrate die Wettbewerbs-Todesrate

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

Auch wenn es vielleicht so erscheint dass man zu stark trennt, so ist doch für das verwendete Programmierkonzept entscheidend dass jede Funktion nach Möglichkeit eine genau Aufgabe hat, weshalb diese beiden Funktionen getrennt wurden. Wie man vermuten kann werden diese von einer Funktion aufgerufen die für das Berechnen der gesamten Todesrate zuständig ist.

Algorithm 3 calculateTotalDeathRates()

Ensure: berechnet die gesamten Todesraten aller Merkmale

```
for i=0 to n-1 do
    TotalDeathRate[i] = 0;
    addTotalIntrinsicDeathRateOf(i);
    addTotalCompDeathRateOf(i);
end for
```

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

Algorithm 4 calculateTotalBirthRates()

Ensure: berechnet die gesamten Geburtsraten aller Merkmale

```
        ↓ intrinsische Geburtenrate ↓
for i=0 to n-1 do
    TotalBirthRate[i] = Members[i] · BirthRate[i] · (1 - Mutation);
end for
        ↓ Mutationsraten ↓
for i=1 to n-2 do
    TotalBirthRate[i] += Members[i-1] · BirthRate[i-1] · Mutation · 0.5;
    TotalBirthRate[i] += Members[i+1] · BirthRate[i+1] · Mutation · 0.5;
end for
TotalBirthRate[0] += Members[1] · BirthRate[1] · Mutation · 0.5;
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 Total Eventrate

```
TotalEventRate = 0;
for i=0 to n-1 do
    TotalTraitRate[i] = TotalBirthRate[i] + TotalDeathRate[i];
    TotalEventRate += TotalTraitRate[i];
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

```
calculateTotalDeathRates();
calculateTotalBirthRates();
calculateTotalEventRate();
```

4.1.2 Ereignis und Zeit bestimmen

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

Algorithm 7 sampleEventTime()

Ensure: Zieht die nächste Ereigniszeit

```
EventTime = rollExpDist(TotalEventRate);
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 i Merkmalsraten damit gemeint ist. Abbildung (5) illustriert den Auswahlprozess.

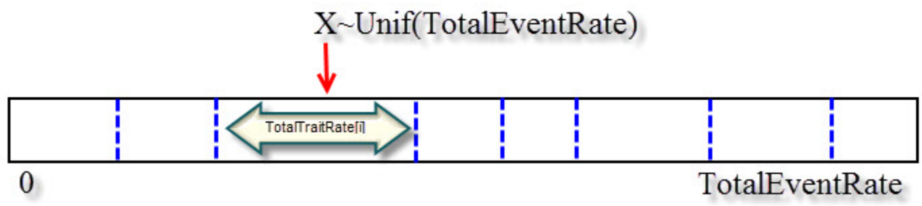

Abbildung 5: Auswahl des Merkmals nach Anteil an der TotalEventRate

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

```
X = rollUnifDist(TotalEventRate);
for i=0 to n-1 do
  if X ≤ TotalTraitRate[i] then
    ChosenTrait = i;
    return;
  end if
  X -= TotalTraitRate[i];
end for
```

Auf die selbe Weise wird entschieden welches Ereignis eintrifft und speichern diese Entscheidung in "isBirth". 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

```
X = rollUnifDist(TotalTraitRate[ChosenTrait]);
if X ≤ TotalBirthRate[ChosenTrait] then
    isBirth = true;
else
    isBirth = false;
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

```
X = rollUnifDist(TotalTraitRate[ChosenTrait]);
if isBirth then
    Members[ChosenTrait] += 1;
end if
if ¬isBirth & Members[ChosenTrait] > 0 then
    Members[ChosenTrait] -= 1;
end if
```

Zum Schluss wir noch eine Funktion erstellt welches das Ausführen eines Ereignisses in richtiger Reihenfolge ausführt und in einem Schritt aus gegebenen Raten die eine Veränderung der Population durchführt.

Algorithm 11 changeATrait()

Ensure: lässt ein Ereignis ein Merkmal treffen

```
choseTraitToChange();
choseEventType();
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

```
calculateEventRates();
sampleEventTime();
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.

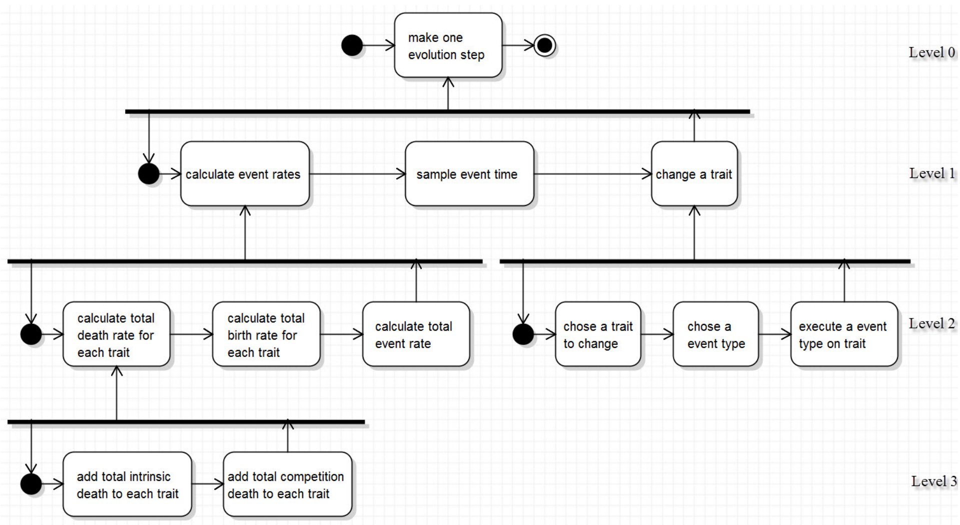

Abbildung 6: Diagramm mit Funktionsaufrufen und ihren Tiefenebenen

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()

Ensure: A full evolution Step happened
Require: $t, X = \{0, \ldots, n-1\}$

$\downarrow$ calculateEventRates() $\downarrow$
for $x \in X$ **do**
  $D(x) := n_t(x) \cdot \Big(d(x) + \sum_{y \in X} c(x,y) \cdot n_t(y)\Big)$
  $B(x) := \underbrace{b(x) \cdot (1-\mu) \cdot n_t(x)}_{arteigene}$
  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
$TotalEventRate := \sum_{x \in X} TotalTraitRate(x)$
$\downarrow$ sampleEventTime() $\downarrow$
sample $Z \sim exp(TotalEventRate)$
$t += Z$
$\downarrow$ choseTraitToChange() $\downarrow$
sample $Y \sim U(0, TotalEventRate)$
for $x \in X$ **do**
  if $Y \leq TotalTraitRate(x)$ **then**
    $ChosenTrait := x$
    break
  end if
  $Y -= TotalTraitRate(x)$
end for
$\downarrow$ choseEventType() $\downarrow$
sample $Y \sim U(0, TotalTraitRate(ChosenTrait))$
if $Y \leq B(ChosenTrait)$ **then**
  isBirht := true
else
  isBirth := false
end if
$\downarrow$ executeEventTypeOnTrait() $\downarrow$
if isBirth **then**
  $n_t(ChosenTrait) += 1$
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 (wenn auch hoher) linearer Aufwand zu erwarten ist.
Zwar ist der nachvollziehbar am Modell gehalten worden, jedoch ist durch die Wettbewerbsrate ein quadratischer Aufwand in der Anzahl der Merkmale nicht vermieden worden.
In unserer Situation werden wir immer eine überschaubare Menge an Merkmalen haben, aber der Nutzten meiner Optimierung ist bereits ab einem Merkmal spürbar sein (praktisch wurde er erst ab 2 gemessen vgl. Abb. 20).
Nach meinen Tests, die ich später im Kapitel "Verhaltenstest" einführe, ergab sich der größte zeitliche Aufwand in der Berechnung der Raten (genauer der Todesraten), was auch zu erwarten war.
Diese Optimierung vermeidet es die Raten komplett neu zu berechnen und möchte sie stattdessen anpassen.
Dazu werden die Raten nicht zu Beginn berechnet wie zuvor in Algorithmus 12 "makeEvolutionStep()" in der 1. Zeile. Als erstes 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 kann zusammen mit dem ausgewählten Merkmal "chosenTrait" folgende Anpassungen gemacht werden:

- Die **intrinsische Todesrate** von "chosenTrait" wird um die das geborene Individuum erhöht:
  TotalDeathRate[chosenTrait] += DeathRate[chosenTrait];
- Die **Todesrate durch Wettbewerb** wird bei jedem Merkmal um das geborene Individuum erhöht:
  TotalDeathRate[i] += CompDeathRate[i][chosenTrait]; $\forall$ i $\in X$
- Die **Geburtsraten** werden genauso wie die Todesraten behandelt:
  TotalBirthRate[chosenTrait] += BirthRate[chosenTrait];
- Und passend die Mutationsraten:
  TotalBirthRate[i] += Mutation $\cdot$ 0.5 $\cdot$ BirthRate[chosenTrait];
  $\forall$ i $\sim$ chosenTrait
- Zum Schluss noch die Totale Ereignisrate:
  Hier kommt man nicht herum alle Totalen Raten erneut auszurechnen.

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

4.4 Normalisierung

Vielleicht ist aufgefallen dass weder in der Implementierung noch im Pseudocode die Normalisierung erwähnt wurde. Diese Option wurde vom Programmkern getrennt, weil sie den evolutionären Mechanismus nicht beeinflusst, sondern nur die Sichtweise verändert.
Um das zu realisieren wurde an zwei Stellen Veränderungen vorgenommen.

1. Beim Einlesen der Parameter. Hier wurde unmittelbar nach dem Lesen 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 einfach durch wachsendes K verfeinern, ohne Rücksicht auf anzupassende Parameter zu nehmen und man kann mit wachsendem K die Konvergenz wie in Abbildung 3 verfolgen.

2. Zuletzt geht es darum dass die Ergebnisse zwar ohne Rücksicht auf ein vorhandenes K berechnet wurden, aber die Darstellung sollte natürlich reskaliert erfolgen ($\nu_t^K = \frac{\nu_t}{K}$). Deswegen ist es außerhalb des Programmkerns nur über einen sogenannten "getter", eine selbstgebaute Zugriffsfunktion, möglich auf die Mitglieder eines Merkmals zuzugreifen. Dieser "getter" liefert stehts den reskalierten Wert zurück.

4.4.1 Sehr viele Sprünge

Gerade bei der Normalisierung [...]

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 im Programm verwendet wurde, welcher Codestil und wie das Layout der Programme gewählt wurde und 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 [9, 14 - Grafische Benutzungsschnittstellen] kennt Standard C++ keine Elemente für grafische Benutzungsoberflächen (englisch *graphical user interfaces - GUI*). Weil es für unseren Fall jedoch nicht wegzudenken ist, wird hier kurz vorgestellt warum welche Variante der GUI-Programmierung ausgewählt werden sollte und welche es gibt.

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 verwendete, an einem Institut was überwiegend mit Ubuntu arbeitet. Deswegen kam MFC nicht in Frage.

GTK$_+$ (the GIMP Toolkit)
Das ist eine weitere Bibliothek zur Erstellung von GUI's. GTK$_+$ wurde Hauptsächlich für GIMP, ein mächtiges Bildbearbeitungsprogramm, entwickelt. GTK$_+$ wird zur Entwicklung des GNOME-Desktops benutzt, welche auf vielen Linux Systemen zu finden. 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 oder der sehr ausführlichen Dokumentation. Obendrein ist es eine Open Source Software.

"Zusammengefasst: Qt ist die ausgereifteste und umfangreichste Open Source Software für die portable Entwicklung grafischer Benutzungsoberflächen mit C++"[9, S.452]

5.2 Architektur und Module

Modul heißt möglichst unabhängige Sammlung [...]
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. 7, rechts).
   Das besondere an diesem Bereich ist dass er mit Standard c++11 implementiert wurde und somit überall wiederverwendbar ist. Er ist leicht zu verändern und sehr flexibel, weshalb z.B. einen anderen "Population Kernel" der ein anderes Modell beschreibt, ohne Veränderungen an anderen Modulen vornehmen zu müssen.
   Natürliche müsste der neue Kernel weiterhin die Funktion "makeEvoultionStep" implementieren.

2. "Graph Management"
   Der zweite Bereich organisiert alle Daten die die Simulation anfragen kann. Hier liegt 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 dem "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. Um für die Analyse der Ergebnisse möglichst viel Übersicht zu haben, verwendet das Hauptfenster eine weitere Klasse "PlotWindow" welche ein extra Fenster mit der Graphen öffnet (Plot-Fenster). Wie man vermuten kann wird hier bereits etwas zum Darstellen eines Graphen und 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 selber verantwortlich.

Die folgende Abbildung 7 stellt zusammenfassend die obigen Module und die Klassenhierarchie dar:

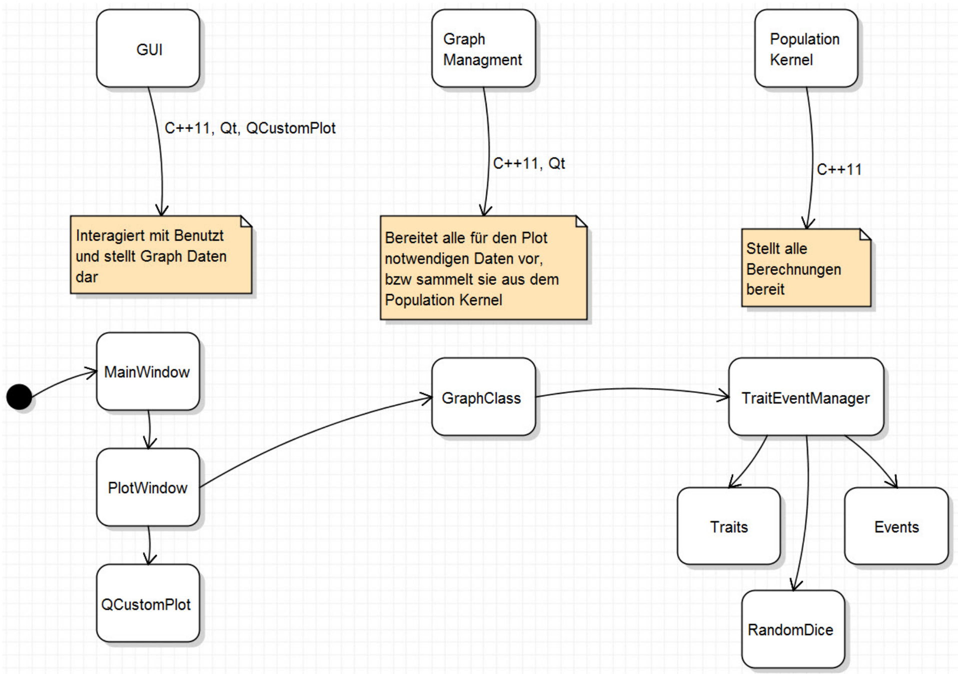

Abbildung 7: Arbeitsmodule und Klassenabhängigkeiten

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-Responsability-Prinzip" und wurde von Robert C. Martin in [11] 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 Parameter

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 8):

- m: Anzahl der Merkmale
- K: Parameter
- μ: Mutationswahrscheinlichkeit
- $c(1,1)\ \dots\ c(1,m)$: 1. Zeile der Wettbewerbsraten
  $\vdots$
  $c(m,1)\ \dots\ c(m,m)$: 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
  $\vdots$
  $b(m)$: Geburtenrate des m.Merkmals

- $d(1)$: Todesrate des 1.Merkmals

  $\vdots$

  $d(m)$: Todesrate des m.Merkmals

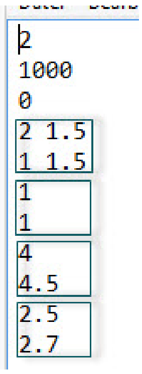
2
1000
0
2 1.5
1 1.5
1
1
4
4.5
2.5
2.7

Abbildung 8: Datei

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

Nebenstehend in Abb. 8 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.

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 9, gelb markiert).
Sobald man einen Namen in das einzige möglich Feld eingegeben hat, kann man zwischen den Schaltflächen "load File" und "create File" wählen (Abbildung 9, rot markiert).

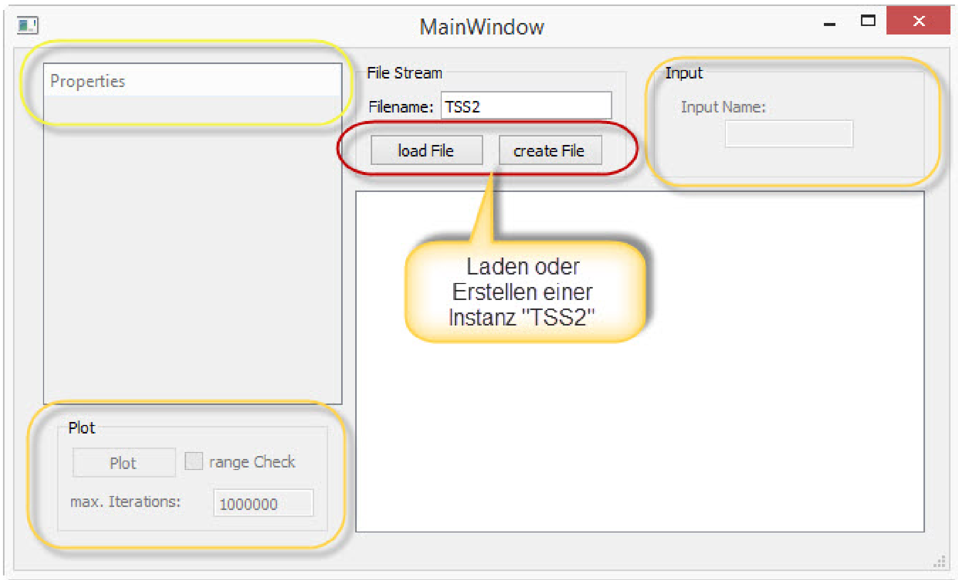

Abbildung 9: MainWindow nach dem Start

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 den selben Umfang (Abbildung 10) und der gewünschte Ast lässt sich einfach erweitern (Abbildung 11).

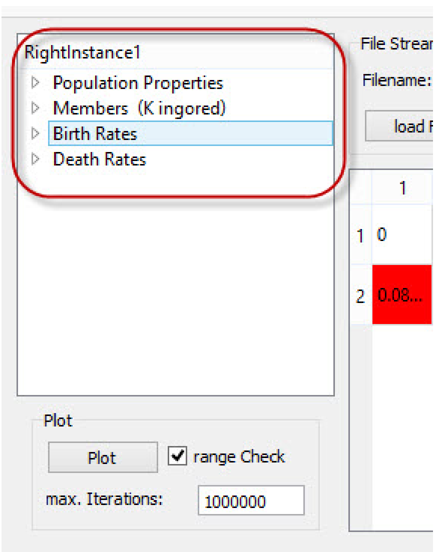

Abbildung 10: Baumstruktur - geschlossen

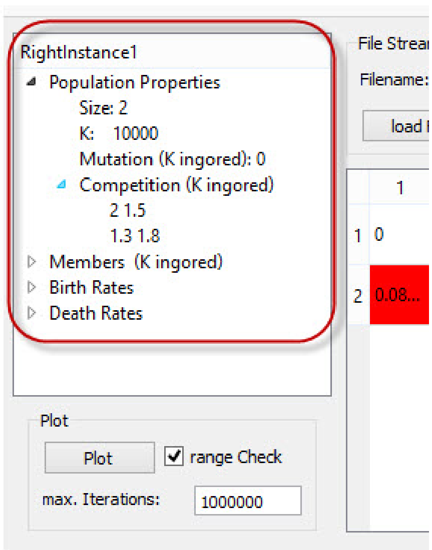

Abbildung 11: Verzweigte Baumstruktur - geöffnet

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" [9].
Während zuvor die Reihenfolge der Programmschritte vorbestimmt war, so gilt das nicht mehr für graphische Benutungsoberflächen. Einfach weil die Reihenfolge der Interaktionsschritte eines Benutzters, z.B. Anklicken oder Mausbewegung, nicht vorhersehbar ist und daher auch nicht die Reihenfolge der auszuführenden Programmschritte.
Es geht also darum Funktionen zu schreiben, die nicht von 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, wurde für diesen Fall nur eine Eingabemöglichkeit offen gelassen (Abbildung 12) und die Parameter der Reihe nach Abgefragt. Die Eingabeaufforderung wird durch das über der Textbox liegende Label gegeben vermittelt und dient dem Programm als "Lesezeichen" um den gelesenen Parameter einzuordnen.

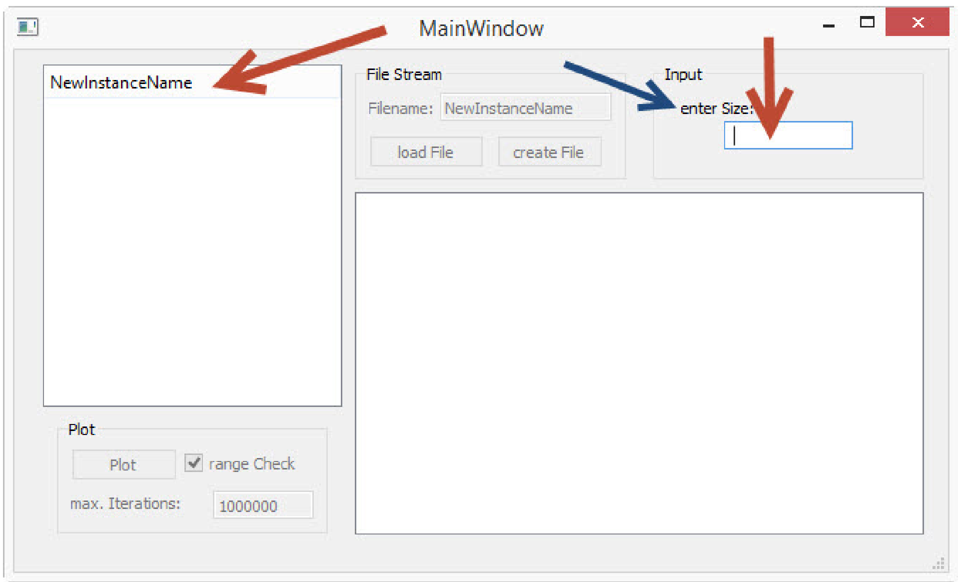

Abbildung 12: Nach Klick auf "create File" werden die neuen Parameter einzeln abgefragt

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

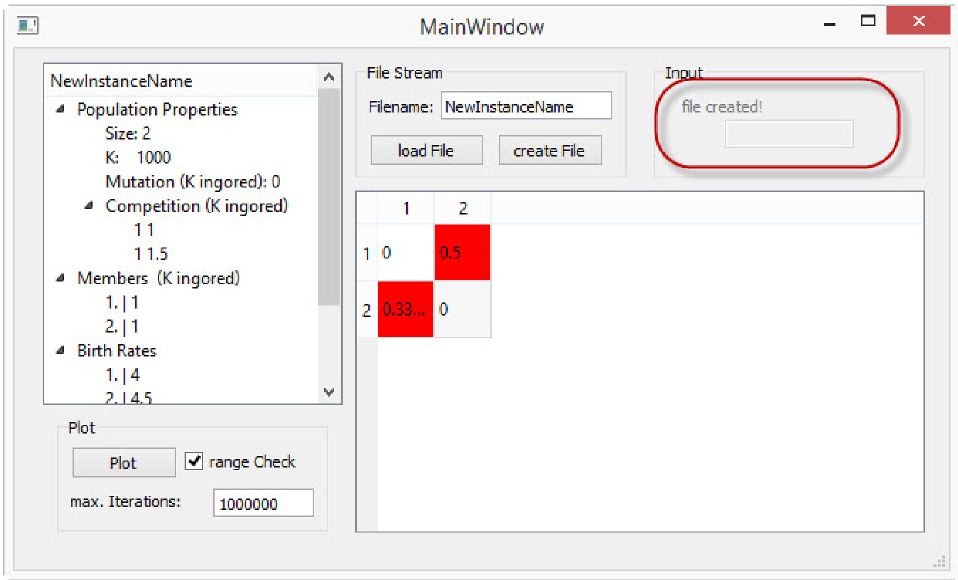

Abbildung 13: Nach Eingabe des letzten Parameters

5.4.3 Darstellung des Graphen

Nachdem Erstellen oder Laden einer Testinstanz ist es möglich die Simulation zu starten. Das erkennt man an dem aktivierten "Plot" Bereich (vgl. Abb. 9, zu 10, 11, 13) links unten.
Wie man schon auf vielen Abbildungen zuvor beobachten kann gibt es im "Plot" Bereich, eine Eingabe Feld 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 dem 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 ist und etwas zusätzliche Zeit verstrichen ist. Die frühzeitigen Abbruchbedingungen 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 sollte den zeitlichen Verlauf der Populationsgrößen beobachten können.
- Um die Prozesse besser analysieren zu können sollte es möglich sein,

Stellen des Graphen näher betrachten zu können (zoom).

- Um Simulationen vergleichen zu können sollten außerdem Ausschnitte als Bilder gespeichert werden können.
- Und natürlich sollte 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 die selben Ressourcen zugreifen. Das zwangsläufige Ergebnis hat wahrscheinlich jeder schon erlebt:

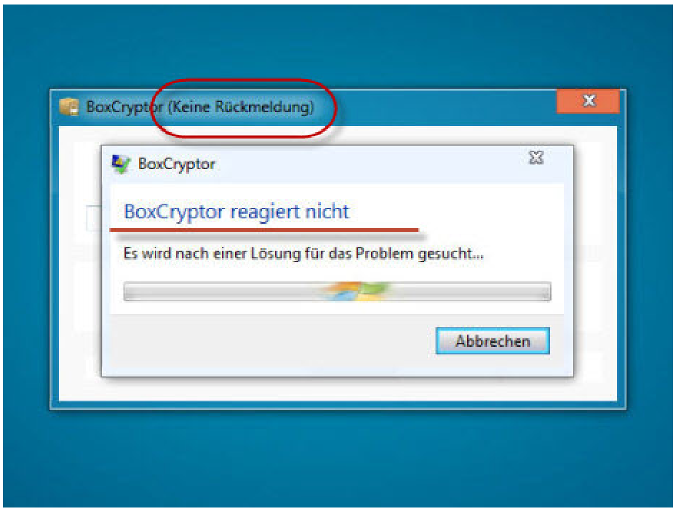

Abbildung 14: Hauptthread wurde überlastet

Die Lösung dieses Problems ist es 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 dass 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 15)

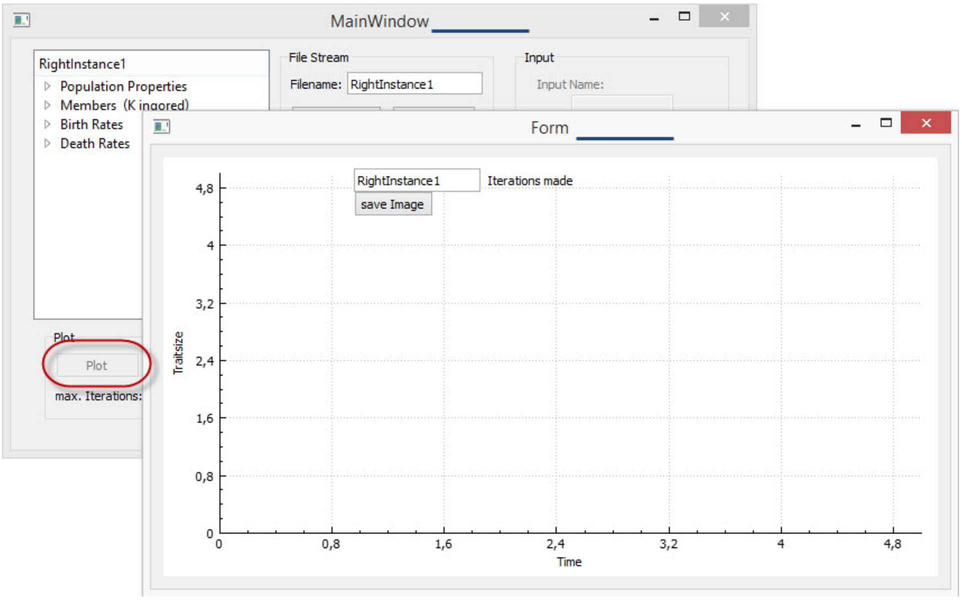

Abbildung 15: Start des PlotWindow

In blau sieht man dass, das Programm nicht eingefroren ist als die Fenster übereinander gelegt wurden.

Wenn die Simulation einen günstigen Zustand erreicht hat, oder die maximale Anzahl an gewünschten Iterationen absolviert hat, werden anschließend maximal 10mio Punkte auf dem Koordinatensystem zu Graphen verbunden (Abbildung 16).

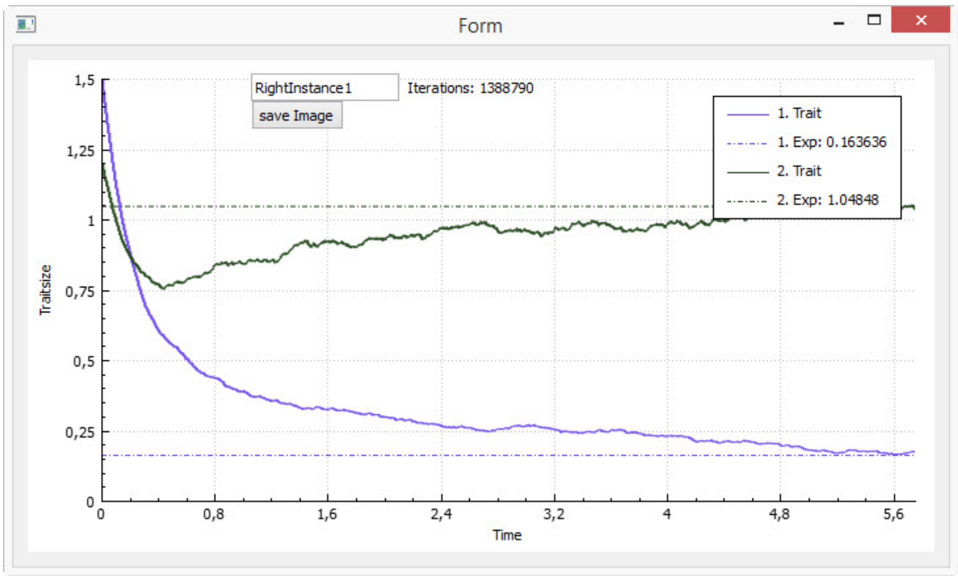

Abbildung 16: PlotWindow mit Dimorpher Population

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.
- 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 16 sehen kann, aber gut ausgearbeitet wurde, ist die Bewegungsfreiheit auf dem Bild.

- Mann kann in das Bild hineinzoomen.
- Auf dem Koordinatensystem kann man sich durch "Ziehen" bewegen.
- Das Fenster lässt sich durch strecken skalieren, wobei sich der Graph automatisch anpasst.
- Automatisches Anpassen der Messgitter.

Als Beispiel dient Abbildung 17, welche durch Heranzoomen, Bewegen zum gewünschten Bereich und maximale Fenstergröße (vgl. kleine Legende) erstellt wurde.

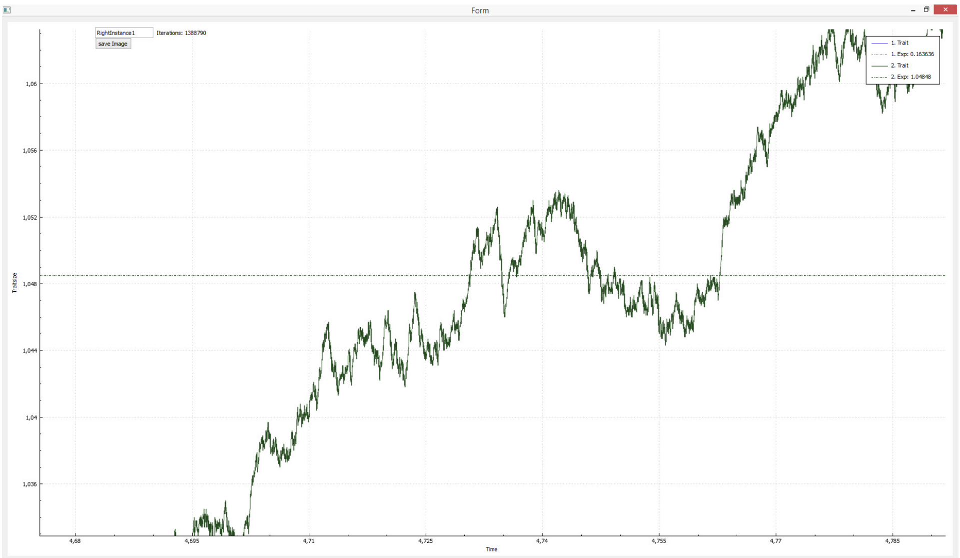

Abbildung 17: PlotWindow mit Dimorpher Population

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

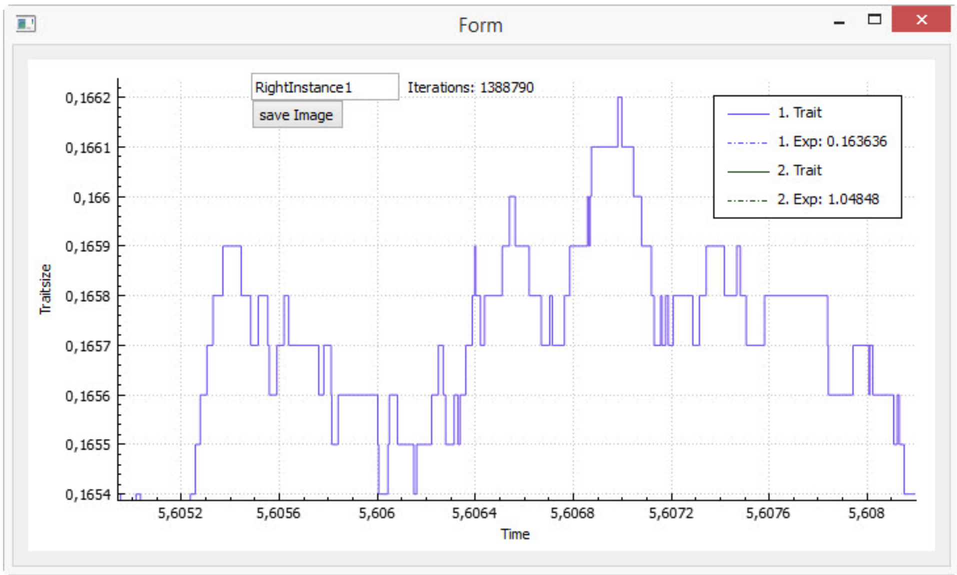

Abbildung 18: PlotWindow mit Dimorpher Population

6 Verhaltenstests und Korrektheit der Implementation

Ein ganz besonders interessantes Thema ist die Korrektheit der Implementation. Diese ist generell mit steigender Komplexität schwerer zu prüfen (besonders bei Zufallsbedingten Simulationen).
Daher habe ich mich dem Prinzip der "Testgetriebene Entwicklung" (Test Driven Develeopment - TDD) zugewendet [12].
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 anders 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 3 Punkten z.B. zunächst eine Funktion (ein Test) geschrieben werden die unser Modell mit Parametern initialisiert und z.B. 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 findet sich in [10, The Bowling Game: An example of test-first pair programming].

In der Abbildung 19 ist ein Beispiel für eine Implementation 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 von neu die Todesraten berechnet und prüft schließlich ob alle Raten trotzdem dem erwarteten Wert entsprechen (roter Kasten). Das ist dank der modularen Implementierung die in Kapitel 4 - Simulation 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 19: 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 20 für die ersten Tests des BPDL Programms vorgemacht worden. In Rot sieht man den zuvor erwähnten Test:

```
Starte D:\Thesis\FinalRegulatedPopulation\build-TraitEventManager-Desktop_Qt_5_2_1_
********* 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 ...
QDEBUG : TraitEventManagerTest::verifyTotalBirthRate() elapsed time: 61 ms
```

Abbildung 20: 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).

6.2 Unit Tests des Programmkerns

Hier werden Tests vorgestellt mit denen die korrekte Berechnung der Raten verifiziert werden soll. Dazu wird die folgende Testinstanz mit 3 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 3 Merkmale gewählt weil es die minimale Anzahl ist um die Auswirkungen von "inneren" Merkmalen auf Rand-Merkmale (und umgekehrt) 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 20 zu sehen.

6.2.1 Raten

Das testen der Todesraten läuft auf den finalen Todesraten Test aus Abbildung 19 hinaus. Diese zeigt die Implementation des entsprechenden Tests. Jede weitere Implementation kann im Quellcode nachgesehen werden, jedoch werden hier nur heuristisch die Tests aufgelistet um die Qualitätstests nachvollziehbar zu machen. In den Tests selber wird 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,

Erwartet: $d(y) \cdot n(y)$
$= 5 \cdot 100 = 500$

Testergebnis: $\begin{pmatrix} 500 \\ 500 \\ 500 \end{pmatrix}$

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,

Erwartet: $n_0(y) \cdot \sum_{w \in X} c(y,w) \cdot n_0(w)$
$= 100 \cdot (2 \cdot 100 + 0.5 \cdot 100) = 25000$

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

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 einfach an 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.

$$\underline{\text{Erwartet}}: \quad D(y) = d(y) \cdot n(y) + n_0(y) \cdot \textstyle\sum_{w \in X} c(y,w) \cdot n_0(w)$$
$$= 500 + 25000 = 25500$$

$$\underline{\text{Testergebnis}}: \quad \begin{pmatrix} 30500 \\ 25500 \\ 40500 \end{pmatrix}$$

Wie erwartet bestätigt sich das Ergebnis für y. In Abbildung 20 kann man die Ausführung aller 3 Tests noch verfolgen.
In anderen Tests (des TSS Programms) wurden die selben Tests noch für unterschiedliche Instanzen gemacht. Das ermöglicht es davon auszugehen dass die Todesraten korrekt berechnet werden.

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

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

$$\underline{\text{Testergebnis}}: \quad \begin{pmatrix} 950 \\ 1000 \\ 950 \end{pmatrix}$$

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.

$$\underline{\text{Erwartet}}: \quad B(y) + D(y) =$$

$$1000 + 25500, \text{ bzw. } \begin{pmatrix} 950 \\ 1000 \\ 950 \end{pmatrix} + \begin{pmatrix} 30500 \\ 25500 \\ 40500 \end{pmatrix} = \begin{pmatrix} 31450 \\ 26500 \\ 41450 \end{pmatrix}$$

$$\underline{\text{Testergebnis}}: \quad \begin{pmatrix} 31450 \\ 26500 \\ 41450 \end{pmatrix}$$

Schließlich folgt daraus auch die Totale Ereignisrate.

$$\underline{\text{Erwartet}}: \quad 31450 + 26500 + 41450 = 99400$$

$$\underline{\text{Testergebnis}}: \quad 99400$$

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 gesetzt 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 den selben Raten gezogen und gemittelt.
Der Erwartungswert dieser Ziehung entspricht $\frac{1}{\text{TotalEventRate}}$ 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% herum.

TSS Mutationszeiten:
Um die korrekte Ziehung der Mutationszeitpunkte in einer TSS Simulation wird eine etwas andere Testinstanz verwendet, die nicht entscheiden 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 eine Merkmale getötet.

2 Das übrige Merkmal wird auf sein Gleichgewicht gehoben.

3 Dann wird eine Funktion aufgerufen die später 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 natürlich das Mittel gezogen und auf eine absolute Genauigkeit von 99.95% geprüft. Die Testergebnisse lagen in der Regel bei einem Fehler um 0.002% herum.

6.2.3 Veränderung der Population

Wie vielleicht bemerkt wurde kommen wir zum letzten Bereich der Aufteilung aus Abbildung 6 Level 1.

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 - "Simulation" erwähnt ist es sehr nützlich gewesen die Raten getrennt zu berechnen und 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.

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

6.3 Unit Tests der Konvergenz

Hier wurde erstmal geprüft ob das vom Programm ermittelte Gleichgewicht wirklich richtig berechnet wurde.
Schließlich habe wurde noch 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 der dimorphen Population oft korreliert d.h. oft waren beide gering oder hoch.

$<<<\quad \downarrow$ Reserviert $\downarrow \quad >>>$

7 TSS Prozesse

Seite 21 - Invasion

$<<<\quad \uparrow$ Reserviert $\uparrow \quad >>>$

Bei TSS Prozessen beobachten wir wie sich Merkmale gegeneinander durchsetzten und sich verdrängen. (Tafelbild)
Genau wie bei der LPA-Normalisierung ergeben sich TSS-Prozesse(Trait Substitution Sequence) als Grenzprozesse von BPDL-Prozessen. Zu der LPA-Normalisierung sollten jedoch mit größer werdendem K die Mutationen seltener werden ($\frac{1}{e^{VK}} << \mu_K << \frac{1}{Klog(K)}$) , also die Mutationswahrscheinlichkeit gegen 0 streben. Skaliert man nun noch zusätzlich die Zeit, so führt dies dazu, dass die Zeit, die ein Merkmal benötigt, um sich gegenüber einem anderen durchzusetzen und dieses zu verdrängen, infinitesimal klein wird. Somit simulieren die TSS-Prozesse eine Population, die zu jedem Zeitpunkt monomorph ist und sich im entsprechenden (für $K < \infty$ angepassten) Gleichgewicht befindet.
Spätestens jetzt wird die Fitness-Funktion interessant:

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

Diese Fitness-Funktion gibt an, wie gut sich ein Merkmal gegenüber einem anderen durchsetzen kann. Sie ist die asymptotische Wachstumsrate von y, wenn x sich im Gleichgewichtszustand $\bar{n}_x$ befindet und nur wenige Individuen von Typ y in der Population vorhanden sind. In der Simulation sieht man die Fitness der Merkmale zueinander in einer Matrix nach dem Laden der Parameter.

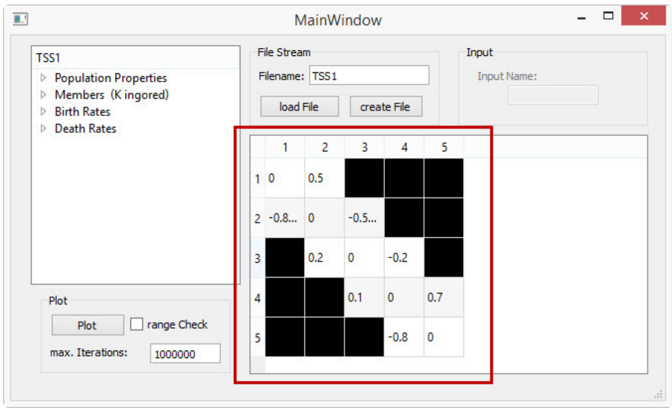

Abbildung 21: Fitness Bandmatrix

Mit der Fitness kann man eine Konvergenz der Wahrscheinlichkeit für das überleben einer Mutation vorhersagen:

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

Da diese Wahrscheinlichkeit gerne bereits beim einlesen der Parameter angezeigt werden will, habe ich vor sie als farblich ansteigenden Akzent den Elementen der Zelle hinzuzufügen. Bisher wird in der Matrix etwas grün oder rot markiert. Rot falls eine Koexistenz vorliegt und grün wenn die Wahrscheinlichkeit für Dominanz des Mutanten über 50% liegt.

Abbildung 22: Fitness Matrix mit roten und gruenen akzenten

Wenn der einfache BPDL Simulator für die Simulation dieses Prozesses verwendet werden würde, würde durch die seltenen Mutationen

7.1 Implementierung

7.2 Optimierung

Hier stelle ich die lineare Interpolation als eine Optimierung vor die Übersichtlichkeit, Analyse und Laufzeit deutlich verbessert. Dann beschreibe ich anhand von Bildern wie gut man Mutationen und deren Auswirkungen verfolgen kann und wie ich geprüft habe ob auch hier alles Korrekt läuft. Schließlich erkläre ich wie ich die Mutationszeitpunkte ermittelt habe.

7.3 Aufwand

Hängt nicht mehr nur von K ab...

7.4 Algorithmus

8 Ausblick

- Weiteres Abbruchkriterium = Zeit : sehr einfach zu implementieren.

Literatur

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

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

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

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

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

9 Fragen

- Fast sicheres Aussterben...

10 Wozu hat es nicht mehr gereicht?

Instanzbrowser oder Dateisuche.
einfaches Bearbeiten erstellter Instanzen.