

Simulation normalisierter BPDFL Prozesse und TSS Prozesse

Boris Prochnau

Geboren am 22. Dezember 1989 in Tartu

22. 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	2
2	Modell	3
2.1	Grundlagen	3
2.2	BPDL Prozess	5
3	Eigenschaften des BPDL Prozesses	
	- anderer Name?	7
3.1	Normalisierung des BPDL Prozesses	7
3.2	Monomorphes Gleichgewicht	8
3.3	Der TSS Grenzwertprozess	8
3.4	Die Fitnessfunktion	10
3.5	Gleichgewicht im dimorphen Fall	11
4	Simulation	13
4.1	Implementierung	13
4.1.1	Raten berechnen	13
4.1.2	Ereignis und Zeit bestimmen	15
4.1.3	Übersicht	18
4.2	Pseudocode	18
4.3	Optimierung für viele Merkmale	20
5	Das Programm	21
5.1	Aufgabenteilung und Flexibilität	21
5.2	Layout	21
6	Verhaltenstest - Korrektheit der Implementation	26
6.1	Unit Tests der Algorithmusmodule	27
7	TSS Prozesse	28
7.1	Optimierung	30
7.2	Aufwand	30
7.3	Algorithmus	30
8	Ausblick	31
9	Fragen	33
10	Wozu hat es nicht mehr gereicht?	33

1 Einleitung

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

Alle Simulationen basieren auf dem Modell, dass jedes Lebewesen einer Population (z.B. Pflanzen) ein bestimmtes Merkmal trägt. Bestimmt wird ein Merkmal durch Todesraten und einer Fortpflanzungsrate. Die Todesraten sind eine einfache natürliche Todesrate und eine durch Wettbewerb mit jedem anderem Individuum. Das heißt diese Rate 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 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 Modell wurde in [1, 2, 3] eingeführt. Dieses nutzt 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 BPDFL Prozess (nach Bolker, Pacala, Dieckmann und Law) genannt.

Ziel wird es sein zwei spezielle BPDFL-Grenzwert-Prozesse simulieren zu können.

2.1 Grundlagen

Sei X der Raum der Merkmale. Jedes Individuum hat genau ein solches Merkmal $x \in X$. Der Einfachheit halber sei X eine Indexmenge: $X = \{1, \dots, n\}$ repräsentativ für eine Durchzählung der Merkmale und im folgenden seien $x, y \in X$ zwei solche Merkmale. Außerdem gilt:

- Jedes Individuum kann sich asexuell fortpflanzen oder sterben.
- Fortpflanzungs- und Todeszeitpunkte können durch sogenannte exponentielle Uhren beschrieben werden (wie in [4, 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.
- Die Fortpflanzung eines Individuums mit Merkmal x kann jedoch auch in der Geburt eines Individuums mit Merkmal y resultieren. Das wird durch die Mutationswahrscheinlichkeit kontrolliert.

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 zwischen Individuen mit Merkmal x und y .

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 $N_t := \#$ Individuen zur Zeit t mit Merkmal x und x_i das Merkmal des i -ten Individuums).

$d(x) + \sum_{y=1}^n c(x, y) \cdot n_t(y)$ Wie oben, nur mit $n := \#$ Merkmale, und $n_t(y) := \#$ Individuen zur Zeit t mit Merkmal y

Weil die Simulation die Entwicklung der Merkmale und nicht die Ereignisse der Individuen darstellt, 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:

- Fortpflanzungsrate des Merkmals x :

$$B_1(x) = b(x) \cdot n_t(x) \cdot \boxed{1-\mu}$$

Oder alternativ die arteigene Geburtenrate (Wachstumsrate) des Merkmals x :

$$\begin{aligned} B_2(x) &= (1 - \mu) \cdot b(x) \cdot n_t(x) \\ &+ \frac{\mu}{2} \cdot b(x+1) \cdot n_t(x+1) \cdot \mathbb{1}_{x < 1} \\ &+ \frac{\mu}{2} \cdot b(x-1) \cdot n_t(x-1) \cdot \mathbb{1}_{x > 1} \end{aligned}$$

- Todesrate des Merkmals x :

add the cases $x=1$ and $x=n$

$$D(x) = d(x) \cdot n_t(x) + n_t(x) \cdot \sum_{y=1}^n c(x, y) \cdot n_t(y)$$

Das entspricht 2 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 Zufallsvariablen) 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 BPDFL Prozess

Die auf dem vorhergehenden Modell basierende Population wird durch die Zufallsvariable

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

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 Raum:

$$\nu_t \in M_F(X) = \left\{ \sum_{i=1}^n \delta_{x_i}, n \in \mathbb{N}, x_1, \dots, x_n \in X \right\}$$

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 BPDFL 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 $\phi : M_F \rightarrow \mathbb{R}$

Mit unserem diskreten Raum X und der konstanten Mutationswahrscheinlichkeit zu Nachbarn vereinfacht sich der Generator. Der für unser Modell angepasster Generator hat somit folgende Form:

$$\begin{aligned} L_{\phi(\nu)} = & \sum_{i=1}^{\infty} b(x_i)(1 - \mu)[\phi(\nu + \delta_{x_i}) - \phi(\nu)] \cdot n(x_i) \\ & + \sum_{i \sim j} n(x_i) \cdot b(x_i) \cdot \mu \cdot [\phi(\nu + \delta_{x_j}) - \phi(\nu)] \cdot n(x_i) \\ & + \sum_{i=1}^{\infty} (d(x) + \sum_{j=1}^{\infty} c_{i,j} \cdot n(x_j)) [\phi(\nu - \delta_x) - \phi(\nu)] \cdot n(x_i) \end{aligned}$$

mit $\phi : M_F \rightarrow \mathbb{R}$ ~~und~~ $n(x_i) :=$ Anzahl Individuen mit Merkmal x_i .

3 Eigenschaften des BPDL Prozesses

- anderer Name?

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 [4] 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.

precise $n_0^K/K \rightarrow n_0$

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 $c^K = \frac{c}{K}$ gelten, da sonst die K-fach erhöhte Population mit einem intensives Aussterben den Vergleich verfälschen würde.

Ein Beispiel für eine LPA-Normalisierung sieht folgendermaßen aus:

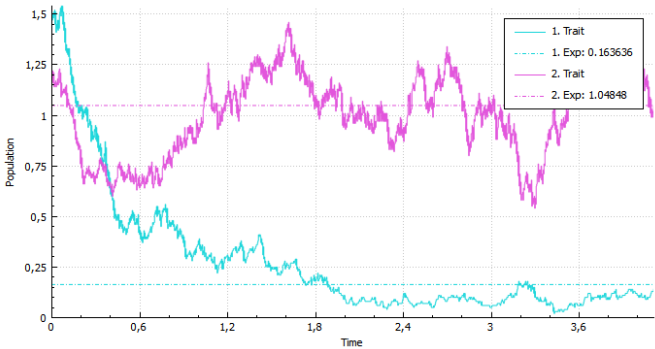


Abbildung 1: LPA Normalisierung mit K=100

Here (after the figures) write the generator for the rescaled process, and the statement of the convergence (with hypotheses) towards the general non-linear system of differential equations.

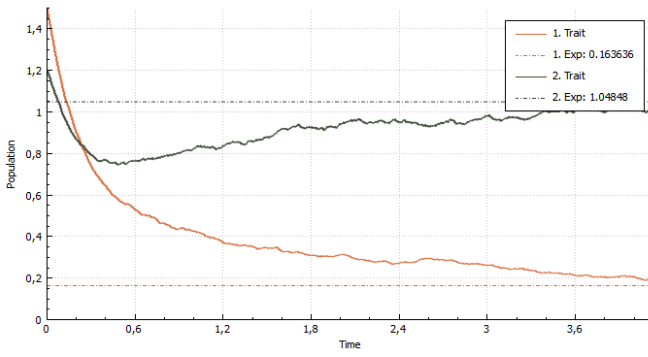


Abbildung 2: LPA Normalisierung mit $K=10000$

Ein weiteres Beispiel dazu ist Abbildung (4).

Für eine Anfangspopulation die aus zwei Merkmalen besteht, konvergiert der Prozess ohne Mutation gegen die Lösung der Konkurrenz Lotka-Volterra Gleichung für zwei Spezies (Siehe Kapitel 11 -Law of Large Numbers- von Ethien und Kurtz [5]).

3.2 Monomorphes Gleichgewicht

Wir stellen fest dass im Falle der monomorphen Population für $K \rightarrow \infty$, ν_t gegen eine Funktion konvergiert die folgende Gleichung erfüllt:

$$\begin{aligned} \dot{n} &= (b(x) - d(x) - n \cdot c(x, x)) \cdot n \\ n(0) &= n_0 \end{aligned} \quad (1)$$

Wir wollen hieraus einen stabilen Zustand für die Population ermitteln indem sich die Populationsgröße nicht mehr ändern darf:

$$\begin{aligned} 0 &= \dot{n} = (b(x) - d(x) - nc(x, x))n \\ \Rightarrow 0 &= b(x) - d(x) - nc(x, x) \\ \Rightarrow \bar{n} &= \frac{b(x) - d(x)}{c(x, x)} \end{aligned} \quad (2)$$

\bar{n} ist somit das Gleichgewicht einer monomorphen Population. Zudem gilt dass stets eine Konvergenz der Population gegen \bar{n} für beliebige Startwerte vorliegt.

3.3 Der TSS Grenzwertprozess

Die bei der LPA-Normalisierung erwähnte Konvergenz gilt auch für d Merkmale mit Mutation mit dem selben Argument, da die Mutationen nach rechts und links begrenzt sind. Dadurch hat man immer nur maximal d Subpopulationen.

Die Differenzialgleichung zu diesem Prozess konvergiert d -dimensional und ähnelt bis auf den Mutationsanteil der d -dimensionalen Konkurrenz Lotka-Volterra Gleichung.

for big populations and rare mutations

Genau wie bei der LPA-Normalisierung ergeben sich TSS-Prozesse (Trait Substitution Sequence) als Grenzprozesse von BPDFL-Prozessen. Zu der LPA-Normalisierung sollten jedoch mit größer werdendem K die Mutationen seltener werden

$$\left(\frac{1}{e^{\sqrt{K}}} \ll \mu_K \ll \frac{1}{K \log(K)} \right),$$

also die Mutationswahrscheinlichkeit gegen 0 streben.

$K \mu_K$

Frage: Warum soll μ_K in dem obigen Bereich gewählt werden?

Antwort: Dank Freidlin and Wenzell [6] erwarten wir dass unsere dominante Spezies $\exp(cK)$ Zeiteinheiten 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 erste Schranke rechtfertigt. Um die zweite 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 x ist fitter ($f(x, y) > 0$), so wird es mit positiver Wahrscheinlichkeit eine Invasion auslösen. 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. Diese Invasion wird eine Zeit von der Ordnung $\log(K)$ benötigen. Das lässt sich folgern indem man Branching Prozesse mit

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 [7].

you must explain that this TSS is a very particular limit of the BPDFL process, in which the population can jump from one monomorphic population at trait x to another monomorphic population at trait y whenever the $f(y, x) > 0$. The time scale separation is such that at any fixed time we have at most 2 coexisting populations. This simplifies a lot the analysis because the proof involves only 2-species Lotka-Volterra systems.

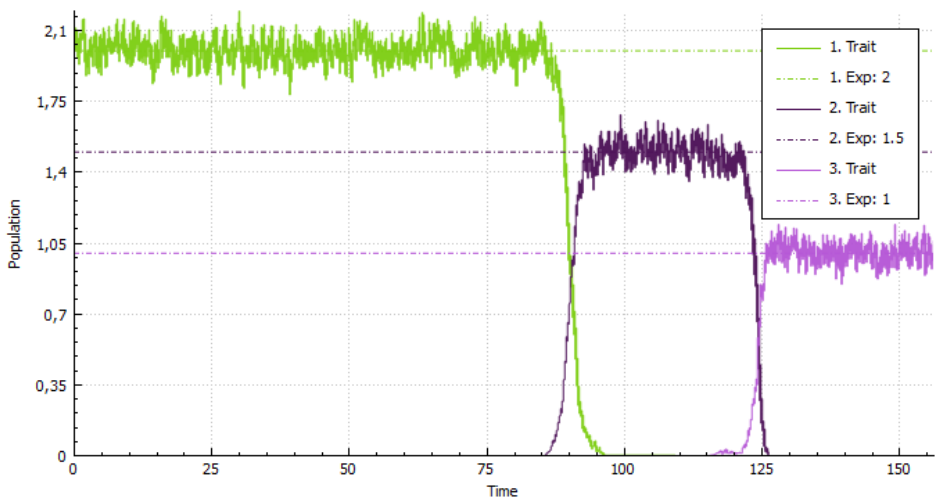


Abbildung 3: TSS Prozess mit: $K = 1000$ und $4 \cdot 10^6$ Sprüngen

3.4 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 in einem Gleichgewichtssystem durchsetzen kann.

Man erkennt dass die Fitnessfunktion die Geburtenrate eines Mutanten $b(x)$ dem zu erwarteten Widerstand, in Form der Todesrate $-d(x) - c(x, y)\bar{n}_y$, gegenüberstellt. 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.

Man kann in dieser Stelle bereits erahnen dass die Fitnessfunktion bzw. die Wachstumsrate eines Mutanten die Möglichkeit bietet aussagen über die Invasionswahrscheinlichkeit bietet. Und tatsächlich wird in [8] eine Konvergenz für $K \rightarrow \infty$ von einer Konvergenz gegen

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

gesprochen. Begründet wird das durch [...]

look at thm 4 of
Champagnat 2006,
and reference there,
i.e. the book of
Athreya and Ney p109

3.5 Gleichgewicht im dimorphen Fall

Wir wissen mittlerweile dass die Population für $K \rightarrow \infty$ gegen eine deterministische Funktion konvergiert. Angenommen es gibt wieder keine Mutation, dann gilt:

$$\begin{aligned}\dot{n}_x &= n_x(b(x) - d(x) - c(x, x)n_x - c(x, y)n_y) & n_x(0) &= n_{x,0} \\ \dot{n}_y &= n_y(b(y) - d(y) - c(y, x)n_x - c(y, y)n_y) & n_y(0) &= n_{y,0}\end{aligned}\quad (3)$$

pay attention to the consistency of notation : here $n(x)$, $n(y)$

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:

$$\begin{aligned}n_x &= \frac{(b(x) - d(x))c(y, y) - (b(y) - d(y))c(x, y)}{c(y, y)c(x, x) - c(y, x)c(x, y)} \\ n_y &= \frac{(b(y) - d(y))c(x, x) - (b(x) - d(x))c(y, x)}{c(y, y)c(x, x) - c(y, x)c(x, y)}\end{aligned}\quad (4)$$

Die BPDFL 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 [8, 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

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 :

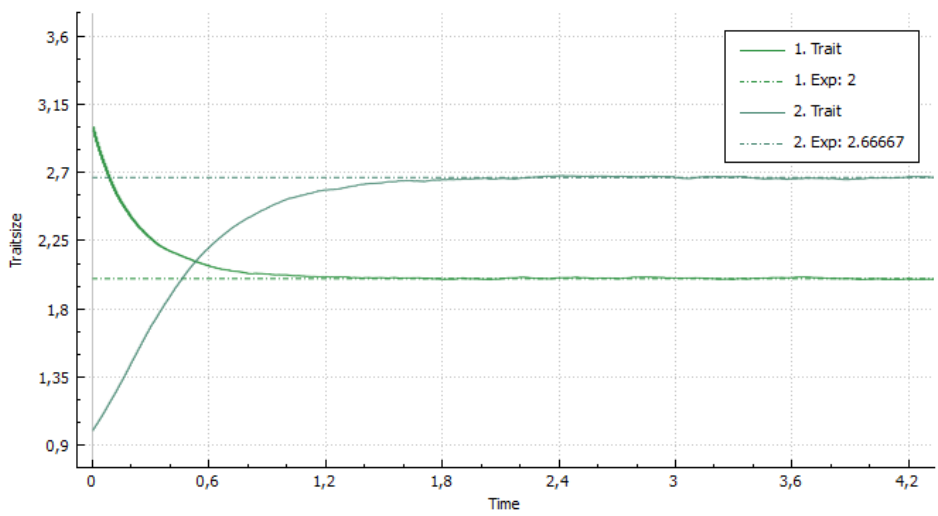


Abbildung 4: Konvergenz mit $K=100000$ und $15 \cdot 10^6$ Sprüngen

4 Simulation

In diesem Kapitel wird der Kern der Simulation algorithmisch näher untersucht. Dieser Kern besteht im Wesentlichen aus einem Sprung des BPDFL 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 [4] weil...

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 (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] · Members[i]

Diese Funktion addiert die Wettbewerbs-Todesrate zur aktuellen Todesrate.

Algorithm 2 addTotalCompDeathRateOf(TraitIndex: i)

Ensure: addiert zur Todesrate die Wettbewerbs-Todesrate

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

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

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

```
1: TotalEventRate = 0;
2: for i=0 to 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 aktuellen Raten berechnet wurden

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

```
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 $\frac{\text{TotalTraitRate}[i]}{\text{TotalEventRate}}$ 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.

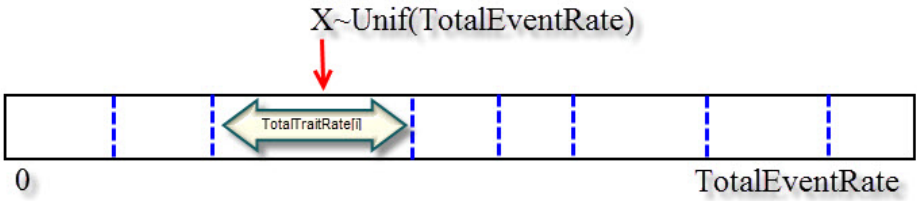


Abbildung 5: Auswahl des Merkmals nach Anteil an der TotalEventRate

Im Code wird dafür iterativ geprüft ob " $X \sim \text{Unif}(\text{TotalEventRate})$ " im ersten Intervall der Länge $\text{TotalTraitRate}[0]$ 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 Intervalllänge $\text{TotalTraitRate}[0]$ reduziert um schließlich erneut mit dem ersten relevanten Intervall verglichen zu werden (jetzt $\text{TotalTraitRate}[1]$). Auf diese weise nähert man sich immer weiter dem getroffenen Merkmal:

Algorithm 8 choseTraitToChange()

Ensure: wählt ein Merkmal zum Ändern aus

```

1:  $X = \text{rollUnifDist}(\text{TotalEventRate});$ 
2: for  $i=0$  to  $n-1$  do
3:   if  $X \leq \text{TotalTraitRate}[i]$  then
4:     ChosenTrait =  $i$ ;
5:     return;
6:   end if
7:    $X -= \text{TotalTraitRate}[i];$ 
8: 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,

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

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 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

```
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.

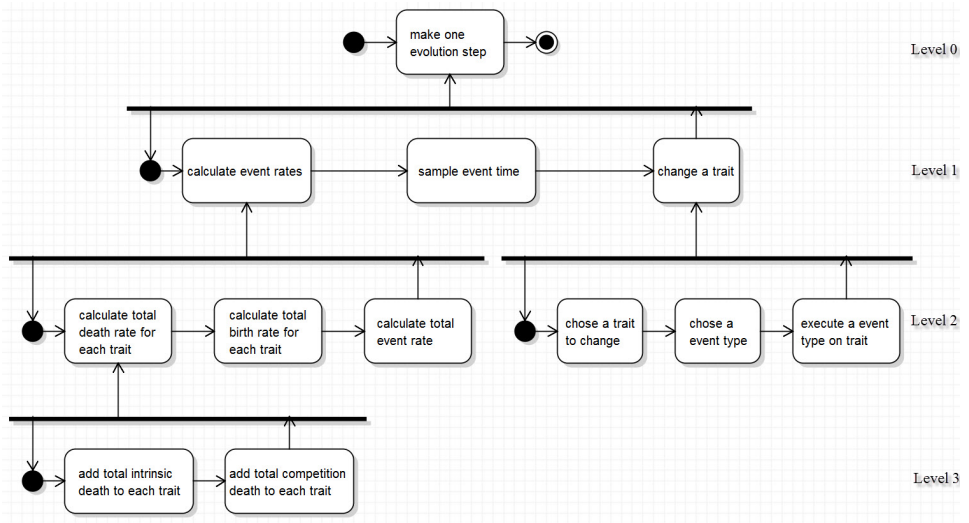


Abbildung 6: Diagramm mit Funktionsaufrufen und ihren Tiefenebenen

Normalisierung [...]

4.2 Pseudocode

Natürlich lässt sich der Ablauf eines Sprunges auch durch Pseudocode in eine Funktion zusammenfassen:

Algorithm 13 EvolutionStep()

Ensure: A full evolution Step happened

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

```
1:   ↓ calculateEventRates() ↓
2: for  $x \in X$  do
3:    $D(x) := n_t(x) \cdot \left( d(x) + \sum_{y \in X} c(x, y) \cdot n_t(y) \right)$ 
4:    $B(x) := \underbrace{b(x) \cdot (1 - \mu) \cdot n_t(x)}_{\text{arteigene}}$ 
5:   if  $x > 0$  then
6:      $B(x) += \underbrace{b(x-1) \cdot n_t(x-1)}_{\text{MutationLinks}} \cdot \frac{\mu}{2}$ 
7:   end if
8:   if  $x < n-1$  then
9:      $B(x) += \underbrace{b(x+1) \cdot n_t(x+1)}_{\text{MutationRechts}} \cdot \frac{\mu}{2}$ 
10:  end if
11:   $TotalTraitRate(x) = B(x) + D(x)$ 
12: end for
13:  $TotalEventRate := \sum_{x \in X} TotalTraitRate(x)$ 
14:   ↓ sampleEventTime() ↓
15: sample  $Z \sim \exp(TotalEventRate)$ 
16:  $t+ = Z$ 
17:   ↓ choseTraitToChange() ↓
18: sample  $Y \sim U(0, TotalEventRate)$ 
19: for  $x \in X$  do
20:   if  $Y \leq TotalTraitRate(x)$  then
21:      $ChosenTrait := x$ 
22:     break
23:   end if
24:    $Y - = TotalTraitRate(x)$ 
25: end for
26:   ↓ choseEventType() ↓
27: sample  $Y \sim U(0, TotalTraitRate(ChosenTrait))$ 
28: if  $Y \leq B(ChosenTrait)$  then
29:   isBirht := true
30: else
31:   isBirth := false
32: end if
33:   ↓ executeEventTypeOnTrait() ↓
34: if isBirth then
35:    $n_t(ChosenTrait) + = 1$ 
36: else
37:   if  $n_t(ChosenTrait) \geq 0$  then
38:      $n_t(ChosenTrait) - = 1$ 
39:   end if
40: 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 Nutzen meiner Optimierung ist bereits ab einem Merkmal spürbar sein (praktisch wurde er erst ab 2 gemessen).

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:
$$\text{TotalDeathRate}[\text{chosenTrait}] += \text{DeathRate}[\text{chosenTrait}];$$
- Die **Todesrate durch Wettbewerb** wird bei jedem Merkmal um das geborene Individuum erhöht:
$$\text{TotalDeathRate}[i] += \text{CompDeathRate}[i][\text{chosenTrait}]; \quad \forall i \in X$$
- Die **Geburtsraten** werden genauso wie die Todesraten behandelt:
$$\text{TotalBirthRate}[\text{chosenTrait}] += \text{BirthRate}[\text{chosenTrait}];$$
- Und passend die Mutationsraten:
$$\text{TotalBirthRate}[i] += \text{Mutation} \cdot 0.5 \cdot \text{BirthRate}[\text{chosenTrait}];$$

$$\forall i \sim \text{chosenTrait}$$
- 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.

5 Das Programm

Nun kommen wir zur eigentlichen Simulation. Damit ist sowohl die Darstellung als auch die Programmarchitektur gemeint.

5.1 Aufgabenteilung und Flexibilität

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" - "faulen Code" zu verhindern. Das bedeutet dass das Programm mit steigender Komplexität zunehmend unflexibler wird, also das Hinzufügen weiterer Features oder das Ändern/Verbessern zu sogenanntem "undefiniertem Verhalten" führt. (kurze Erklärung zum Begriff).

Die Architektur des Programms kann grob in drei Module gefasst werden.

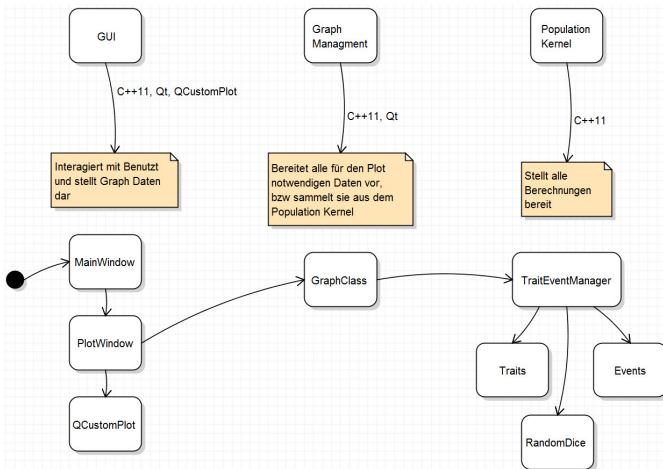


Abbildung 7: Arbeitsmodule und Klassenabhängigkeiten

Innerhalb der Arbeitsschritte sollte dabei genau die selbe Regel der Unabhängigkeit gelten wie bei den Modulen.

5.2 Layout

- 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 beschriebenen Dateien an.

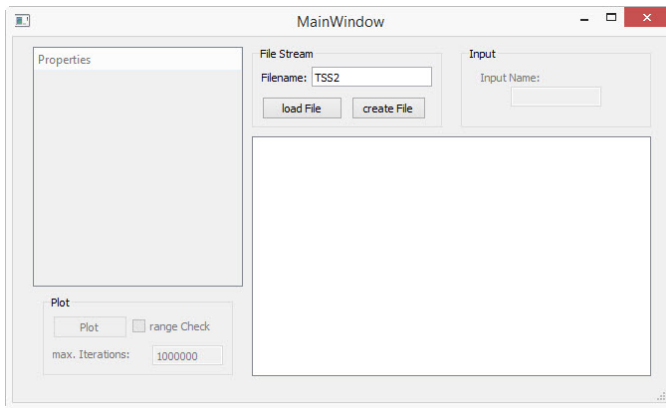


Abbildung 8: MainWindow nach dem Start

Zur Darstellung der gelesenen Parameter habe ich ein Baumstruktur gewählt.

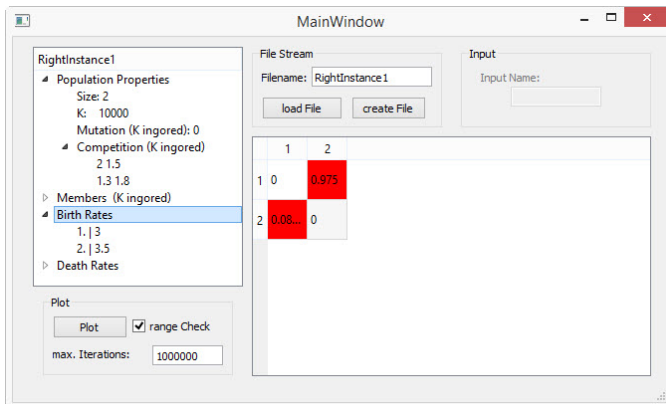


Abbildung 9: MainWindow mit geladenen Parametern

- 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 konnte, sollte bei einem GUI eine Lösung her die Inputkonflikte verhindert und das Einlesen der Daten denkbar einfach macht. Dafür war es sinnvoll "Enter" als Bestätigung abzufangen und sicherzustellen dass der Cursor nur innerhalb des gewünschten Feldes bleibt.

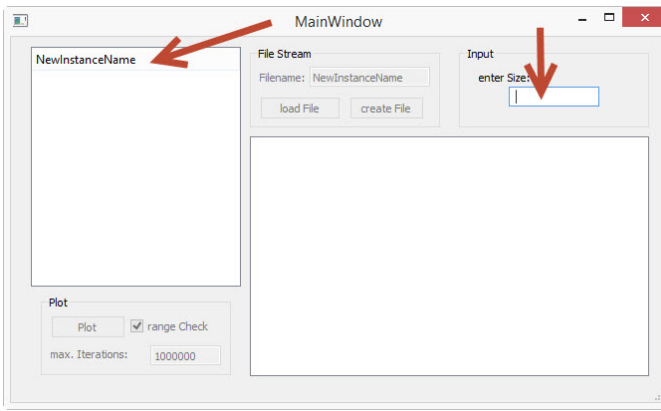


Abbildung 10: Nach Klick auf "create File" werden die neuen Parameter einzeln abgefragt

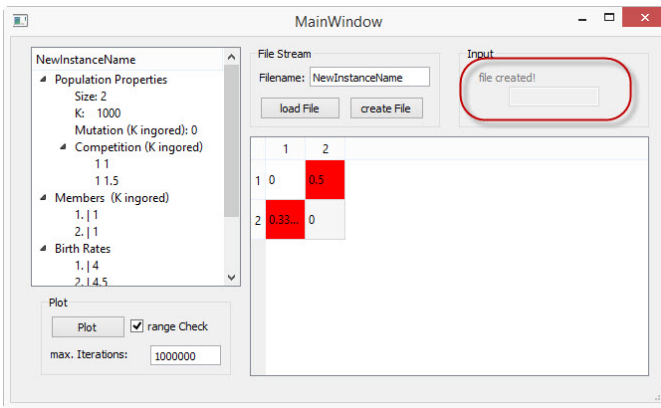


Abbildung 11: Nach Eingabe des letzten Parameters

Was die Darstellung der Graphen angeht, hat sich ein einfaches Bild des fertigen Plots durchgesetzt, wobei das Zoomen und ziehen des Bildes notwendige Elemente zur Untersuchung des Graphen sind. Zusätzlich ist es notwendig viele Bilder vergleichen zu können. Hierfür wurde das Abspeichern des Bildes gewünscht.

Die Simulation wird gestartet nachdem man auf den "Plot" Button drückt. Dabei wird ein neues Fenster geöffnet.

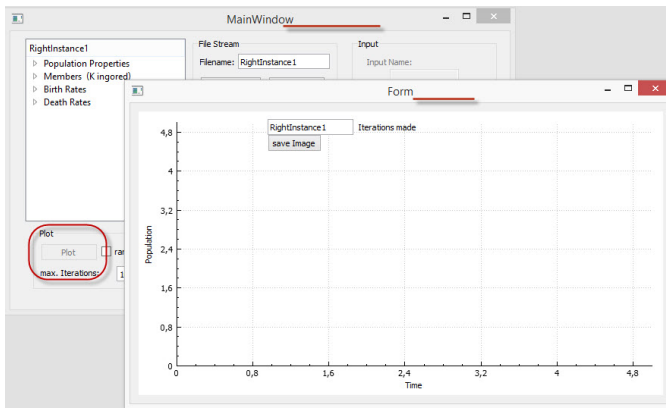


Abbildung 12: Start des PlotWindow

Sobald der Plot Button gedrückt wurde und das Fenster angezeigt wird, arbeitet die Simulation bereits im Hintergrund in einem eigenen Thread. Dort werden alle notwendigen Iterationen bzw. Sprünge der Population durchgeführt ohne den Hauptthread damit zu belasten. Während dieser Arbeiterthread aktiv arbeitet wird der "Plot" Button ausgegraut um mehrfaches Auslösen zu vermeiden und um anzuzeigen dass die Rechnung im Gange ist. Der Arbeiterthread gewährleistet nicht nur eine flüssige Interaktion mit dem Programm, sondern verhindert auch effektiv dass das Betriebssystem denkt das Programm wäre Abstürzt oder würde nicht mehr ordnungsgemäß funktionieren. Das würde sonst folgendes evtl. bekannte Bild hervorrufen:

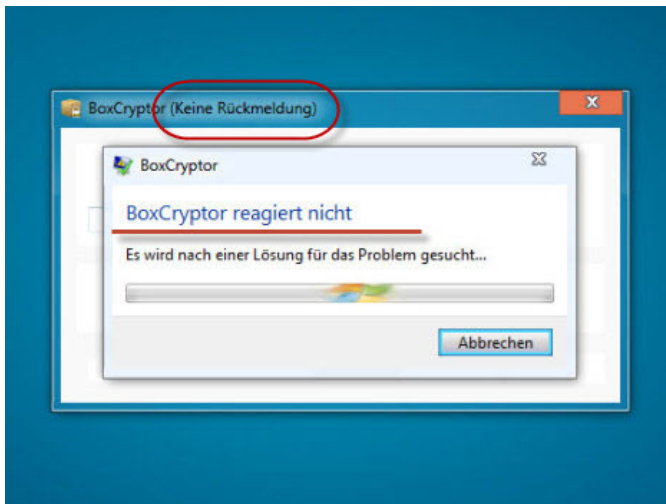


Abbildung 13: Hauptthread wurde überlastet

Wenn die Simulation einen gewünschten Zustand erreicht hat, oder die

maximale Anzahl an gewünschten Iterationen absolviert hat, werden anschließend maximal 10mio Punkte auf dem Koordinatensystem zu Graphen verbunden. Das kann so aussehen:

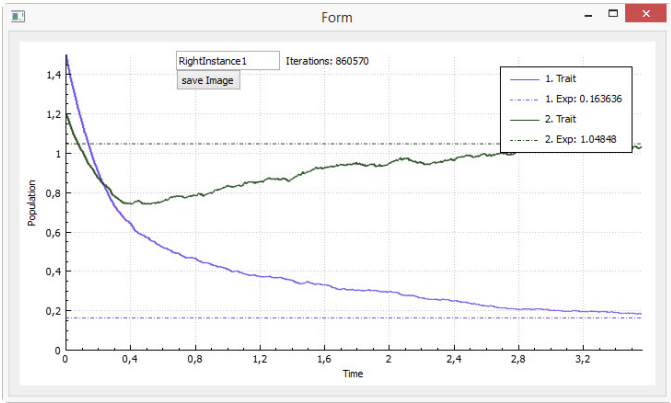


Abbildung 14: PlotWindow mit Dimorpher Population

6 Verhaltenstest - Korrektheit der Implementation

(Korrektheit des Algorithmus nicht notwendig bzw möglich) Ein ganz besonders interessantes Thema ist die Korrektheit der Implementation. Diese ist generell mit steigender Komplexität schwerer zu prüfen (besonders bei Zufallsbedingten Simulationen).

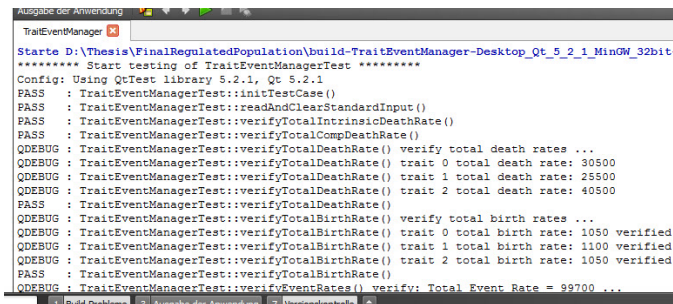
Daher habe ich das Prinzip der "Testgetriebene Entwicklung" (Test Driven Developeopment) verwendet.

Dabei werden Funktionen des Programms unter vorher festgelegten Bedingungen laufen gelassen und mit einem erwarteten Verhalten verglichen. Das Ergebnis ist eine Ausgabe für Erfolg oder Misserfolg des Tests. Folgend ein Beispiel für eine Implementation eines einfachen Tests der prüft ob alle Parameter korrekt aus der Datei in die Objekte geschrieben werden.

```
51 void TraitEventManagerTest::verifyWrittenData()
52 {
53     QCOMPARE(TraitClass::Size,3.);
54     QCOMPARE(TraitClass::Mutation,0.1);
55     for(int i = 0; i < TraitClass::Size; ++i){
56         QCMPARE(Manager.Trait[i].BirthRate,10.);
57         QCMPARE(Manager.Trait[i].DeathRate,5.);
58         QCMPARE(TraitClass::CompDeathRate[i][i],2.);
59     }
60 }
61
62 // ----- section 1: Rates -----
63 /// Unit Tests for INPUT VALIDATION
64
65 void TraitEventManagerTest::readAndClearStandardInput()
66 {
67     Manager.initWithFile("ValidateTests.txt");
68     verifyWrittenData();
69     Manager.clearData();
70     QVERIFY(TraitClass::Size == 0.);
71     QVERIFY(Manager.Trait.size() == 0.);
72     QVERIFY(TraitClass::CompDeathRate.size() == 0.);
73     QVERIFY(Manager.Trait.size() == 0.);
74 }
```

Abbildung 15: UnitTest versichert korrektes lesen von Parametern aus Datei

Anschließend ein Beispiel für einen Durchlauf der Testfunktionen:



```
Ausgabe der Anwendung
TraitEventManager
Starte D:\Thesis\FinalRegulatedPopulation\build-TraitEventManager-Desktop_Qt_5_2_1_MinGW_32bit-
***** Start testing of TraitEventManagerTest *****
Config: Using QTest 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() trait 0 total death rate: 30500
QDEBUG : TraitEventManagerTest::verifyTotalDeathRate() trait 1 total death rate: 25500
QDEBUG : TraitEventManagerTest::verifyTotalDeathRate() trait 2 total death rate: 40500
PASS : TraitEventManagerTest::verifyTotalDeathRate()
QDEBUG : TraitEventManagerTest::verifyTotalBirthRate() verify total birth rates ...
QDEBUG : TraitEventManagerTest::verifyTotalBirthRate() trait 0 total birth rate: 1050 verified
QDEBUG : TraitEventManagerTest::verifyTotalBirthRate() trait 1 total birth rate: 1100 verified
QDEBUG : TraitEventManagerTest::verifyTotalBirthRate() trait 2 total birth rate: 1050 verified
PASS : TraitEventManagerTest::verifyTotalBirthRate()
QDEBUG : TraitEventManagerTest::verifyEventRates() verify: Total Event Rate = 99700 ...
```

Abbildung 16: Ergebnisse einiger Tests

Die Testfunktionen zeigen auch dass eine komplexe Verwendung der Simulationen über die graphische Darstellung hinaus einfach realisierbar ist.

6.1 Unit Tests der Algorithmusmodule

<<< ↓ Reserviert ↓ >>>

Frage: Warum soll μ_K in dem Bereich

$$\left(\frac{1}{e^{\sqrt{V}K}} << \mu_K << \frac{1}{K \log(K)} \right),$$

gewählt werden?

Antwort: Dank Freidlin and Wenzell [6] erwarten wir dass unsere dominante Spezies $\exp(cK)$ Zeiteinheiten im Equilibrium 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 eine unsere Mutationen exponentiell verteilt sind benötigt man ein eine Rate $\mu_K \gg \frac{1}{e^{\sqrt{V}K}}$ um eine Zeit von $\exp(cK)$ nicht zu überschreiten, was die erste Schranke rechtfertigt.

Um die zweite 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 x ist fitter ($f(x, y) > 0$), so wird es mit positiver Wahrscheinlichkeit eine Invasion auslösen. 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. Diese Invasion wird eine Zeit von der Ordnung $\log(K)$ benötigen. Das lässt sich folgern indem man Branching Prozesse mit

<<< ↑ Reserviert ↑ >>>

Bei TSS Prozessen beobachten wir wie sich Merkmale gegeneinander durchsetzen und sich verdrängen. (Tafelbild)

Genau wie bei der LPA-Normalisierung ergeben sich TSS-Prozesse (Trait Substitution Sequence) als Grenzprozesse von BPDFL-Prozessen. Zu der LPA-Normalisierung sollten jedoch mit größer werdendem K die Mutationen seltener werden ($\frac{1}{e^{\sqrt{V}K}} << \mu_K << \frac{1}{K \log(K)}$), 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.

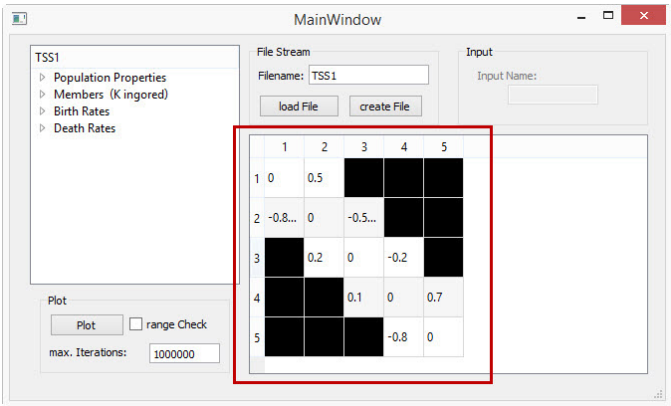


Abbildung 17: Fitness Bandmatrix

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.

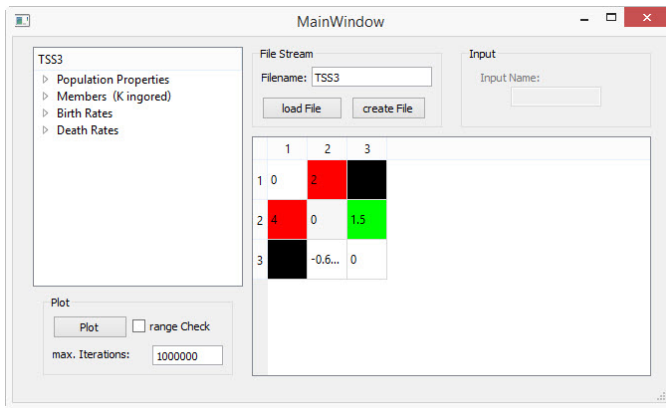


Abbildung 18: Fitness Matrix mit roten und gruenen akzenten

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

7.1 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.

7.2 Aufwand

Hängt nicht mehr nur von K ab...

7.3 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 Fournier and Sylvie Méléard. A microscopic probabilistic description of a locally regulated population and macroscopic approximations. *Ann. Appl. Probab.*, 14(4):1880–1919, 2004.
- [5] Stewart N Ethier and Thomas G Kurtz. *Markov processes: characterization and convergence*, volume 282. John Wiley & Sons, 2009.
- [6] Mark I Freidlin, Joseph Szücs, and Alexander D Wentzell. *Random perturbations of dynamical systems*, volume 260. Springer, 2012.
- [7] Nicolas Champagnat and Sylvie Méléard. Polymorphic evolution sequence and evolutionary branching. *Probability Theory and Related Fields*, 151(1-2):45–94, 2011.
- [8] Nicolas Champagnat. A microscopic interpretation for adaptive dynamics trait substitution sequence models. *Stochastic Processes and their Applications*, 116(8):1127 – 1160, 2006.

9 Fragen

- Normalisierung, Gleichgewicht und mono/dimorphe Fälle als einzelne Kapitel?
- Soll ich monomorphe BPDFL Prozesse einführen? Schließlich kann ich damit das fast sichere Aussterben

10 Wozu hat es nicht mehr gereicht?

Instanzbrowser oder Dateisuche.
einfaches Bearbeiten erstellter Instanzen.