

Simulation normalisierter BPDFL Prozesse

Boris Prochnau

Geboren am 22. Dezember 1989 in Tartu

16. Juli 2014

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Anton Bovier

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	Normalisierung und Eigenschaften des BPDL Prozesses	7
3.1	Normalisierung des BPDL Prozesses	7
3.2	Fitness	8
3.3	Equilibrium	8
4	Algorithmus zur Simulation eines normalisierten BPDL Prozesses	9
4.1	Implementierung	9
4.2	Pseudocode	9
4.3	Optimierung für viele Merkmale	11
5	Simulation und Programmablauf	12
5.1	Aufgabenteilung und Flexibilität	12
5.2	Layout	12
6	Verhaltenstest - Korrektheit der Implementation	17
6.1	Unit Tests der Algorithmusmodule	18
7	TSS Prozesse	19
7.1	Algorithmus	20
8	Ausblick	21
9	Beweise	23
10	Fragen	23

1 Einleitung

Das Ziel dieser Bachelorarbeit ist es ein Programm zu entwickeln welches den zeitlichen Verlauf von sogenannten normalisierten BPDFL Prozessen graphisch darstellt.

Diese BPDFL (von Bolker, Pacala, Dieckmann und Law) Prozesse basieren auf einem stochastischen Populationsmodell, welches im nächsten Kapitel näher vorgestellt wird.

Dem Programm sollte es möglich sein die Parameter einer Population festlegen zu können und die Entwicklung dieser Population schließlich zeitlich verfolgen zu können. Auf diese Weise soll man beobachten können ob sich ein Merkmal gegenüber einem anderen durchsetzen kann oder sich ein stabiler Zustand einpendelt. Insbesondere kann der Einfluss der Normalisierung auf das vorzeitige Aussterben eines Prozesses deutlich dargestellt werden.

Alle Simulationen basieren auf einem Modell, dass jedes Lebewesen einer Population (z.B. Pflanzen) ein bestimmtes Merkmal trägt. Diese Merkmale werden durch Wettbewerb zu jeder existenten Gruppe, Geburten und Todesraten klassifiziert. Schließlich ist es jedoch die Entwicklung der Population und nicht der Individuen die simuliert werden soll, weshalb man im simulierten Prozess zwar den Tod und die Geburt von Individuen verfolgen kann, aber nicht die Entwicklung spezieller Individuen. Der Übergang zu dieser Sichtweise wird näher im 2. Kapitel beschrieben und krönt in der Konvergenz zu einer deterministischen Funktion.

Um zu Prüfen ob und wie schnell der Prozess gegen seinen stabilen Zustand konvergiert, werden in der Simulation auch stabile Zustände für diverse Situationen automatisch dazu gestellt. Anhand dieser können individuelle und dynamische Abbruchkriterien formuliert werden.

Schließlich werde ich noch kurz die TSS Prozesse und ein weiteres Programm vorstellen, welches die bisher betrachteten BPDFL Prozesse erweitert auf TSS-Prozesse. Dabei sollen nicht nur das besonders interessante Verhalten vom Wechsel des dominanten Merkmals simuliert werden, sondern es wird eine verbesserte Laufzeit durch Interpolation vorgestellt die eine effiziente Simulation trotz sehr großer Zeit und besonders präzise Betrachtung von Aktionsreichen Gebieten anbietet.

2 Modell

Das verwendete Model lehnt sich an das Model aus [1] an. 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.

Jedoch wurden für meine Simulation einige kleine Änderungen gewünscht die im Anschluss erläutert werden.

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 (wie in [2, S. 3]) beschrieben werden. Diese Uhren haben exponentiell verteilte Weckzeiten. Durch die Gedächtnislosigkeit der Exponentialverteilung, können alle Uhren nach dem ersten Klingeln neu gestellt werden.
- Die Fortpflanzung eines Individuums aus x kann jedoch auch in der Geburt eines Individuums in y resultieren.

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 Superposition z.B. die beiden Todesraten zu einer gemeinsamen Todesrate zusammenfassen oder die arteigene Geburtenrate abtrennen.

- $b(x) \cdot (1 - \mu)$: ist die arteigene Geburtenrate eines Individuums mit Merkmal x
- $d(x) + \sum_{i=1}^{N_t} c(x, x_i)$: 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).
- $d(x) + \sum_{y=1}^n c(x, y) \cdot n_t(x)$: wie oben, nur mit $n := \#$ Merkmale, und $n_t(x) := \#$ Individuen zur Zeit t mit Merkmal x

Die letzte Darstellung der Todesrate ist praktischer für die Betrachtung der Population. Ähnlich können weitere Ereignisse zusammengefasst werden, so dass man z.B. eine Todesrate und eine arteigene Geburtenraten der Merkmale erstellen kann:

- Arteigene Geburtenrate des Merkmals x :

$$b(x) \cdot (1 - \mu) + (b(x+1) \cdot \mathbb{1}_{x < n} + b(x-1) \cdot \mathbb{1}_{x > 1}) \cdot \frac{\mu}{2}$$

- Gesamte Todesrate des Merkmals x :

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

Die Simulation soll die Entwicklung der Merkmale und nicht die Ereignisse der Individuen darstellen, daher ist es unpraktisch weiterhin die Raten jedes Individuums zu berechnen. Alternativ können Ereignisse zu denen von Merkmalen zusammengefasst werden:

- Fortpflanzungsrate des Merkmals x :

$$B_1(x) = b(x) \cdot n_t(x)$$

- Oder alternativ die 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 :

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

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 jetzt 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 Bereit zu stellen. So lässt sich z.B. aus der Geburtenrate (Wachstumsrate) eines ausgestorbenen Merkmals die Mutationsrate ablesen, ohne weitere Berechnungen machen zu müssen.

2.2 BPD L Prozess

Die Population wird durch die Zufallsvariable

$$\nu_t = \sum_{i=1}^{N_t} \delta_{x_i}$$

beschrieben.

Damit 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 BPD L Prozesses, dass die Mutationen auf einem beliebigen Merkmal (nicht nur den Nachbarn) landen können. Für

einen solchen Prozess ist 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 $\phi : M_F \rightarrow \mathbb{R}$

Der für unser Model angepasster Generator müsste folgende Form haben:

$$\begin{aligned}
L_{\phi(\nu)} &= \int_X b(x)(1 - \mu)[\phi(\nu + \delta_x) - \phi(\nu)]\nu(dx) \\
&+ \int_X b(x) \cdot \mu \cdot \mathbb{1}_{x>0}[\phi(\nu + \delta_{x-1}) - \phi(\nu)]\nu(dx) \\
&+ \int_X b(x) \cdot \mu \cdot \mathbb{1}_{x<n}[\phi(\nu + \delta_{x-1}) - \phi(\nu)]\nu(dx) \\
&+ \int_X d(x)[\phi(\nu - \delta_x) - \phi(\nu)]\nu(dx) \\
&+ \int_X \left(\int_X c(x, y)\nu(dy) \right) [\phi(\nu - \delta_x) - \phi(\nu)]\nu(dx)
\end{aligned}$$

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

3 Normalisierung und Eigenschaften des BPDFL Prozesses

3.1 Normalisierung des BPDFL 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 eingeführt.

Dafür 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 $c^K = \frac{c}{K}$ gelten, da sonst wegen der hohen Population ein intensives Aussterben den Vergleich verfälschen würde. Ein Beispiel dafür sieht folgendermaßen aus:



Abbildung 1: LPA Normalisierung mit $K=100$

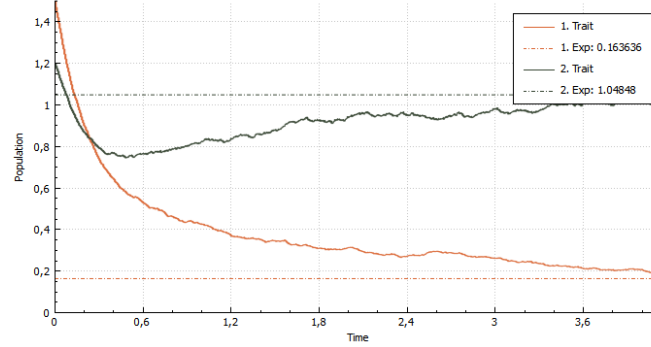


Abbildung 2: LPA Normalisierung mit $K=10000$

3.2 Fitness

3.3 Equilibrium

Man erkennt dass die Merkmale für $K \rightarrow \infty$ eine Konvergenz gegen eine Funktion $\xi_t = n_t \delta_x$ aufweist. Diese Funktion wiederum konvergiert gegen einen stabilen Zustand (im folgenden Equilibrium genannt), worin sich die Populationsgröße nicht mehr ändert. Diese sehen für monomorphe und dimorphe Populationen ohne Mutation folgendermaßen aus:

- Monomorpher Population:

$$0 = \dot{n} = (b(x) - d(x) - \bar{n}c(x, x))\bar{n}$$

$$\bar{n}_x = \frac{[b(x) - d(x)]_+}{c(x, x)}$$

- Dimorphe Population:

Auf die selbe Weise lässt sich ein dimorpher stabiler Zustand ermitteln:

$$n_x = \frac{(b(x) - d(x))c(y, y) - (b(y) - d(y))c(x, y)}{c(y, y)c(x, x) - c(y, x)c(x, y)}$$

$$n_y = \frac{(b(y) - d(y))c(x, x) - (b(x) - d(x))c(y, x)}{c(y, y)c(x, x) - c(y, x)c(x, y)}$$

Also sind (n_x, n_y) , $(\bar{n}_x, 0)$, $(0, \bar{n}_y)$ oder $(0, 0)$ mögliche stabile Zustände im dimorphen Fall.

Die BPDFL Simulationen erkennen dimorphe und monomorphe Populationen und stellen stets einen passenden stabilen Zustand n_x , bzw. \bar{n}_x dar. Um zu entscheiden unter welchen Voraussetzungen welcher stabile Zustand erreicht wird der folgende Satz helfen:

<<< ↓ In Arbeit ↓ >>>

4 Algorithmus zur Simulation eines normalisier- ten BPDFL Prozesses

4.1 Implementierung

Der Simulation liegt ein Algorithmus zugrunde der einen Sprung des Markov Sprung Prozesses durchführt. Im Code wird dazu die "EvolutionStep()" Funktion aufgerufen. (Es gibt auch Jump statt Step, aber mehrere Schritte)

Algorithm 1 EvolutionStep()

Ensure: A full evolution Step happened

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

Von dieser werden folgende Berechnungen angestoßen:

Algorithm 2 EvolutionStep()

Ensure: A full evolution Step happened

- 1: —>calculateEventRates();
 - 2: calculateTotalDeathRates()
 - 3: calculateTotalBirthRates()
 - 4: calculateTotalEventRate()
 - 5: —>sampleEventTime();
 - 6: sampleEventTime();
 - 7: —>changeATrait();
 - 8: choseTraitToChange();
 - 9: choseEventType();
 - 10: executeEventTypeOnTrait();
-

4.2 Pseudocode

Schließlich der Ablauf der tatsächlichen Berechnung:

Algorithm 3 EvolutionStep()

Ensure: A full evolution Step happened

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

```
1:  $\rightarrow \text{calculateEventRates}();$ 
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:      $B(x)+ = \underbrace{b(x-1) \cdot n_t(x-1) \cdot \frac{\mu}{2}}_{\text{MutationLinks}}$ 
7:   end if
8:   if  $x < n-1$  then
9:      $B(x)+ = \underbrace{b(x+1) \cdot n_t(x+1) \cdot \frac{\mu}{2}}_{\text{MutationRechts}}$ 
10:  end if
11:   $TotalTraitRate(x) = B(x) + D(x)$ 
12: end for
13:  $TotalEventRate := \sum_{x \in X} TotalTraitRate(x)$ 
14:  $\rightarrow \text{sampleEventTime}();$ 
15: sample  $Z \sim \text{exp}(TotalEventRate)$ 
16:  $t+ = Z$ 
17:  $\rightarrow \text{choseTraitToChange}();$ 
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:  $\rightarrow \text{choseEventType}();$ 
27: sample  $Y \sim U(0, TotalTraitRate(ChosenTrait))$ 
28: if  $Y \leq B(ChosenTrait)$  then
29:   isBirth := true
30: else
31:   isBirth := false
32: end if
33:  $\rightarrow \text{executeEventTypeOnTrait}();$ 
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

5 Simulation und Programmablauf

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.

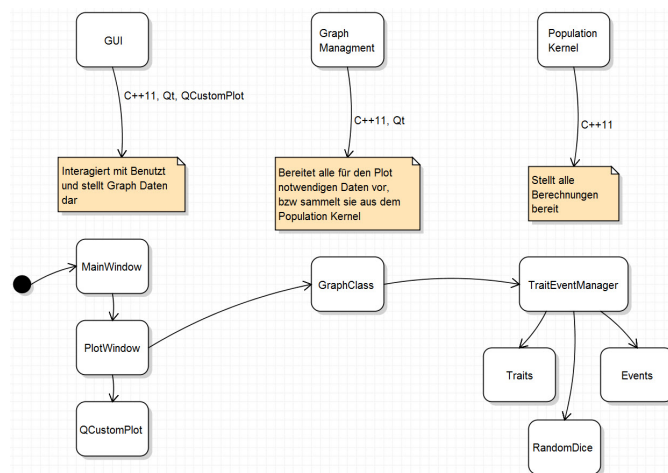


Abbildung 3: Arbeitsmodule und Klassenabhängigkeiten

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.

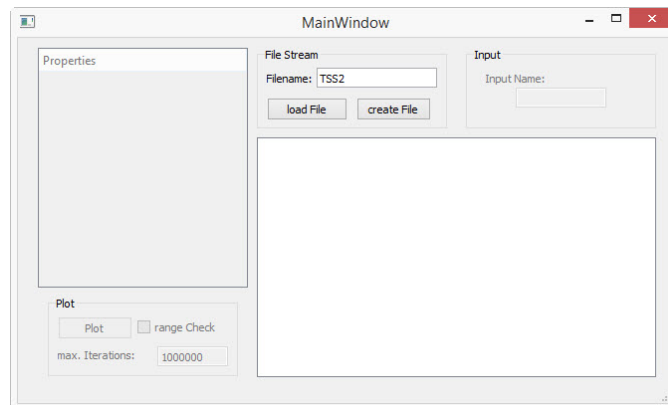


Abbildung 4: MainWindow nach dem Start

Zur Darstellung der gelesenen Parameter habe ich ein Baumstruktur gewählt.

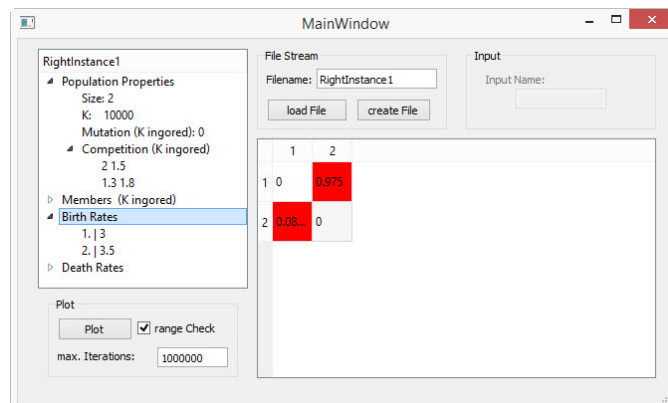


Abbildung 5: MainWindow mit geladenen Parametern

- 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.

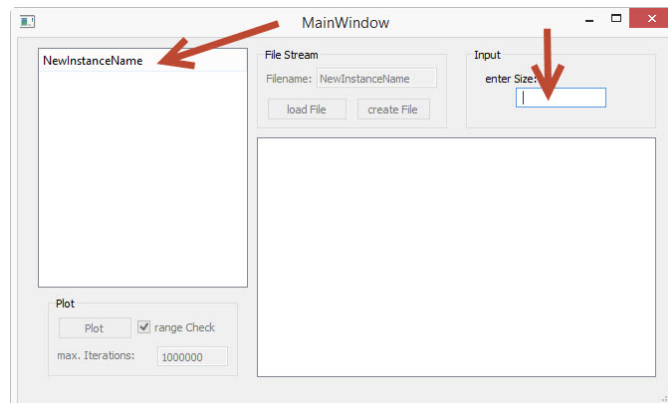


Abbildung 6: Nach Klick auf "create File" werden die neuen Parameter einzeln abgefragt

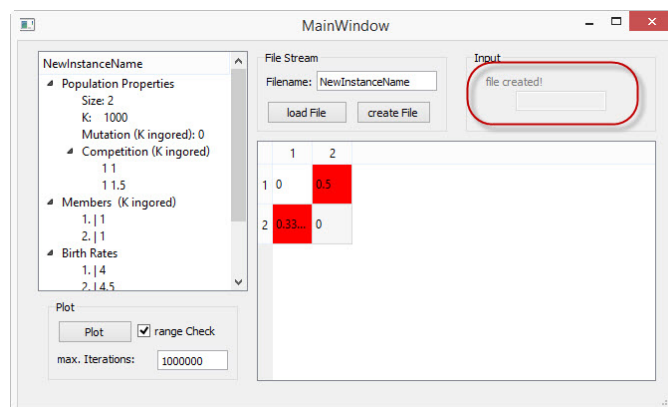


Abbildung 7: Nach Eingabe des letzten Parameters

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.

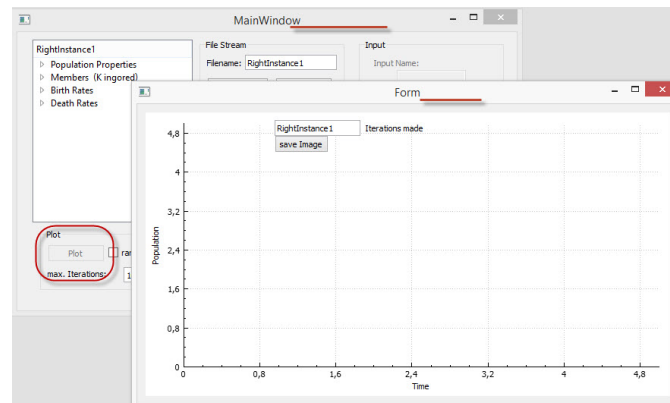


Abbildung 8: Start des PlotWindow

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:

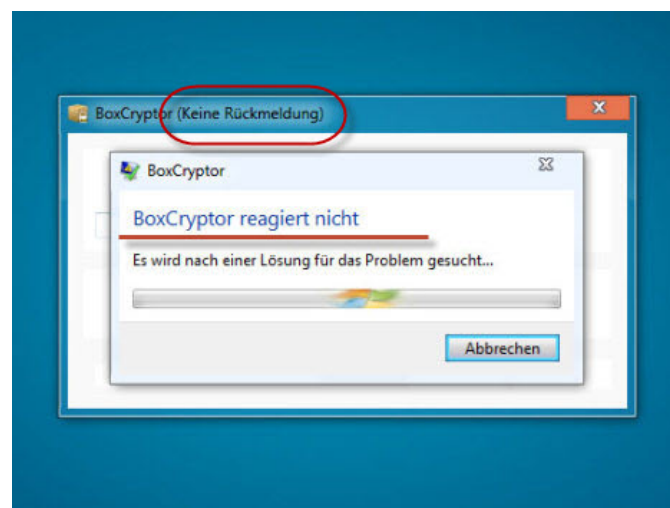


Abbildung 9: Hauptthread wurde überlastet

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:

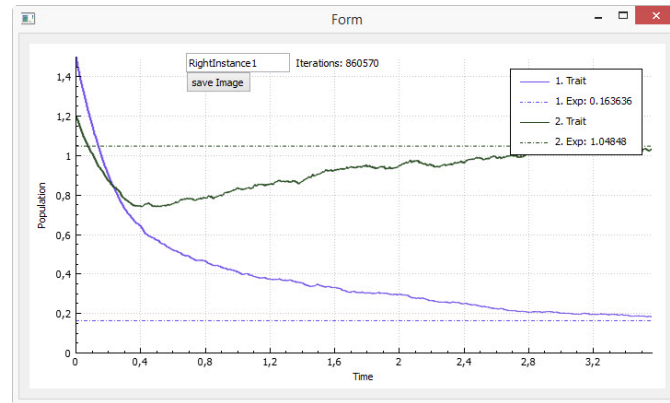


Abbildung 10: PlotWindow mit Dimorpher Population

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).

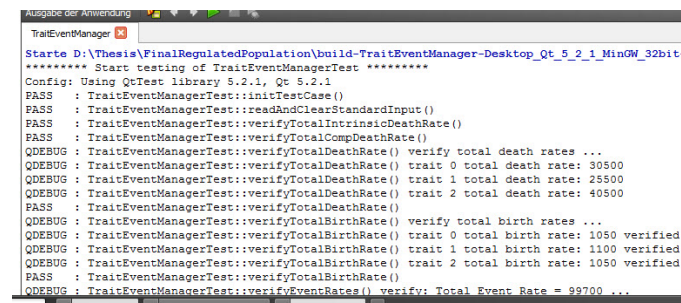
Daher habe ich das Prinzip der "Testgetriebene Entwicklung" (Test Driven Developeopment) verwendet.

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         QCOMPARE(Manager.Trait[i].BirthRate,10.);
57         QCOMPARE(Manager.Trait[i].DeathRate,5.);
58         QCOMPARE(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 }
75
```

Abbildung 11: UnitTest versichert korrektes lesen von Parametern aus Datei

Anschließend ein Beispiel für einen Durchlauf der Testfunktionen:



```
Starte D:\Thesis\FinalRegulatedPopulation\build-TraitEventManager-Desktop_Qt_5_2_1_MinGW_32bit-
***** 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() 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 12: 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

7 TSS Prozesse

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 BPDF-Prozessen. Zu der LPA-Normalisierung sollten jedoch mit größer werdendem K die Mutationen seltener werden ($\frac{1}{e^{\sqrt{K}}} \ll \mu_K \ll \frac{1}{K \log(K)}$), 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) Equilibrium 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.

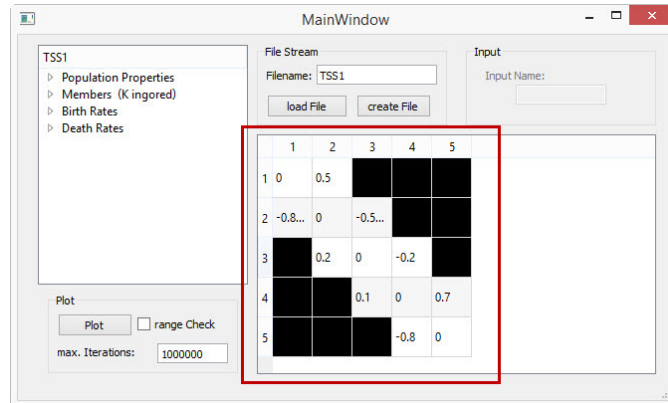


Abbildung 13: Fitness Bandmatrix

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.

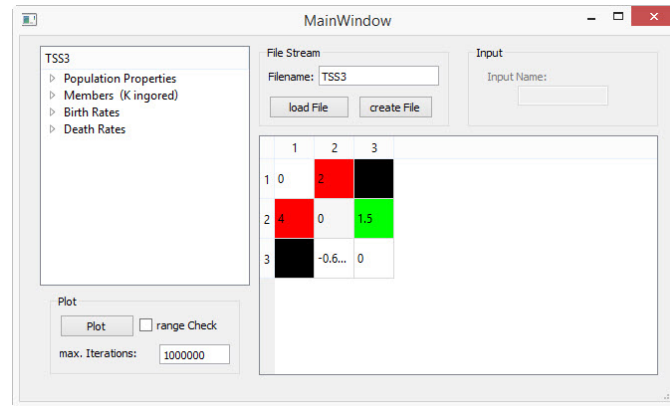


Abbildung 14: Fitness Matrix mit roten und grünen akzenten

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

7.1 Algorithmus

8 Ausblick

- Weiteres Abbruchkriterium = Zeit : sehr einfach zu implementieren.

Literatur

- [1] Nicolas Champagnat. A microscopic interpretation for adaptive dynamics trait substitution sequence models. *Stochastic Processes and their Applications*, 116(8):1127 – 1160, 2006.
- [2] 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.

9 Beweise

10 Fragen

- Normalisierung, Equilibrium und mono/dimorphe Fälle als einzelne Kapitel?
- Soll ich monomorphe BPDF Prozesse einführen? Schließlich kann ich damit das fast sichere Aussterben
- Beweis der LPA-Normalisierung per Differentialgleichungen?