

Small summery

Boris Prochnau

2. Juni 2014

Inhaltsverzeichnis

1 First meeting (14.04): Model of interest	1
2 Second meeting (22.04): Pseudocode	2
2.1 Calculating total-event rates	3
2.2 Sampling the next event-time	5
2.3 Changing a trait	6
3 Fifth meeting (Fr 30.05): Improvements - afterwards	8
3.1 Graphische Änderungen	8
3.2 Interne Änderungen	8
3.3 bekannte Bugs	9
4 Sixth meeting (03.06): Introducing TSS	9

1 First meeting (14.04): Model of interest

These are some main points of the model we are using for the simulation. They are the product of the first meeting with Loren Coquille and Martina Baar.

- We have N traits and $(X_{i=1,\dots,N})$ are the amount of members of the traits.
- We use a constant mutation rate that does not depend on the traits μ .
- but the other rates are depending on traits and change with time (by as the amount of traitmembers rises)
 - b_i , [intrinsic birth-rate]
 - d_i , [intrinsic death-rate]
 - $\left(\sum_{j=1}^N \frac{c_{i,j}}{K} x_j\right)_i$, [competition death]
 - $\mu \left(\frac{b_{i-1}+b_{i+1}}{2}\right)$, [extrinsic birth-rate]

- We use 3 groups of PPP's where every group represents an event.

$$\begin{aligned} & - N_t^{b_i} \\ & - N_t^{d_i} \\ & - N_t^{\bar{c}_i} \end{aligned}$$

where $\bar{c} = \left(\sum_{j=1}^N \frac{c_{\cdot,j}}{K} x_j \right)_i$. Therefore we get 3N processes running that compete about the first one occurring with each one triggering an death/birth for an trait.

- With respect to the fact that the distribution of the N_t changes with the size of the population we can think of an reset of the parameters of N_t due to the fact that the increments are exponentially distributed and therefore are memoryless (markov property)
- The use of coloring one trait was (only \rightarrow ask later) to explain the superposition of the PPP associated with the simulation to extract the occurred event from the PPP's

$$\rightarrow N_t^{total_i} = N_t^{b_i+d_i+\bar{c}_i} = N_t^{b_i} + N_t^{d_i} + N_t^{\bar{c}_i}$$

Therefore we would decide:

$$X_i^{total} = \begin{cases} \text{coloring d} & \text{with prob. p} \\ \text{coloring b} & \text{with prob. q} \\ \text{coloring } \bar{c} & \text{with prob. 1-p-q} \end{cases} \quad (1)$$

with:

$$p = \frac{d_i}{d_i + b_i + \bar{c}_i} \quad q = \frac{b_i}{d_i + b_i + \bar{c}_i} \quad 1 - p - q = \frac{\bar{c}_i}{d_i + b_i + \bar{c}_i}$$

2 Second meeting (22.04): Pseudocode

First thing to mention is that the Pseudocode snippets are marked as „Algorithm“, but in fact they are just functions and I don't know yet how to change the name in this particular LaTeX environment. Also I use a „=“ instead of „ \leftarrow “ because I think it provides better readability.

The following Pseudocode will contain many function-calls. Every function can be identified as a verb, and objects or local variables are nouns. There will also be one Boolean „isBorn“ which is an adjective.

Generally the functions have no return values because in this situation we tell functions to do something (not ask). Means we pass a task to them. There will be no requests for return values.

One other thing to mention is that the function names can get long, but they will (or should) always say what the function does. Not more or less.

The function-body should always follow a rule called „pretty much what you expected“. Some functions do violent this rule slightly. They are usually more than 6 lines and will be changed later.

First we start with the main Step of the Algorithm and continue with further explaining of the used functions. When a new function appear, it will be explained directly below it. Also we separete this section in 3 parts, each one is dedicated to one of the functions used in this following EvolutionStep():

Algorithm 1 EvolutionStep()

Require: -

Ensure: A full evolution Step happened

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

This function does a full evolution step. First thing to do is to calculate the Event rates with „calculateEventRates“, than we sample an exponential time with our current rate parameters with „sampleEventTime“ and finish the Step with changing a Trait with „changeATrait“.

Most functions-calls should not need to be explained, that’s why I leave this explanations out in following comments.

This function will be improved later with not calculating the Rates in every new step, rather than updating them with the previous changes.

2.1 Calculating total-event rates

We will encounter 2 classes called „Trait“ and „Events“. They are so called Data Classes and usually mainly store data.

They are visible at all time and can be used like global variables.

In this first part of calculating total-event rates we only need to introduce the „Trait“ class which will be used as an array of traits „Trait[$i \leq n$]“ with n beeing the maximum number of traits. An Trait Object has attributes that can be accessed through a dot operator like „Trait[i].Members“ what would be „the Members of Trait i“.

This is a listing of all attributes in one Trait Object:

class Trait

- BirthRate
- TotalBirthRate
- DeathRate
- TotalDeathRate
- TotalTraitRate

- TotalEventRate - [static]
- CompDeathRate[i][j] - [static]

Here is much work with Superposition of the PPP included. We summarize the sum of birth-rates from members within a trait to the „TotalBirthRate“ (mutation included), same for „TotalDeathRate“ (competition included). The „TotalTraitRate“ is the summed „TotalBirthRate“ and „TotalDeathRate“ and means the total rate of events for a specific trait. „TotalEventRate“ and „CompDeathRate“ are static, this means that they are the same for all initialized Trait objects. They (all items from the array) access the same variable for this. „TotalEventRate“ is the same as usual and „CompDeathRate“ is the competition-matrix with according rates.

Algorithm 2 calculateEventRates()

Require: -

Ensure: All (total)Rates will be set

- 1: **for** i=0 **to** n-1 **do**
 - 2: calculateTotalDeathRateOf(i)
 - 3: **end for**
 - 4: calculateTotalBirthRates(0);
 - 5: calculateTotalEventRate();
-

In the next function we will manipulate attributes of the Trait Objects. The mutation will be accessed without and Traitindex: „Trait.Mutation“ because it is static and therefore the same for all Objects.

Algorithm 3 calculateTotalBirthRates(StartIndex: i)

Require: int i

Ensure: Total birthrate of Trait „i“ will be set (recursively)

- 1: Trait[i].TotalBirthRate = (Trait[i].Members)·(Trait[i].BirthRate)
 - 2: **if** $i < n - 1$ **then**
 - 3: calculateTotalBirthRates(i+1)
 - 4: Trait[i].TotalBirthRate += $\frac{\text{Trait.Mutation}}{2} \cdot \text{Trait}[i + 1].\text{TotalBirthRate}$
 - 5: **end if**
 - 6: **if** $i > 0$ **then**
 - 7: Trait[i].TotalBirthRate += $\frac{\text{Trait.Mutation}}{2} \cdot \text{Trait}[i - 1].\text{TotalBirthRate}$
 - 8: **end if**
-

In Algorithm 3 in line 3 is used recursion, because this improves the calculation speed a lot, although it slightly makes code less intuitive.

Algorithm 4 calculateTotalDeathRateOf(TraitIndex: i)

[H]

Require: int i

Ensure: Total deathrate of Trait „i“ will be set

- 1: Trait[i].TotalDeathRate = 0;
 - 2: addTotalIntrinsicDeathRateOf(i);
 - 3: addTotCompetitionDeathRateOf(i);
-

Algorithm 5 addTotalIntrinsicDeathRateOf(TraitIndex: i)

- 1: Trait[i].TotalDeathRate = (Trait[i].DeathRate) · (Trait[i].Members)
-

Algorithm 6 addTotalCompetitionDeathRateOf(TraitIndex: i)

- 1: **for** j=0 **to** n-1 **do**
 - 2: Trait[i].TotalDeathRate += (Trait.CompDeathRate[i,j]) · (Trait[j].Members);
 - 3: **end for**
-

Algorithm 7 calculateTotalEventRate()

Require: -

Ensure: Current Totaleventrate is set

- 1: **for** i=0 **to** n-1 **do**
 - 2: Trait[i].TotalTraitRate = Trait[i].TotalBirthRate
 + Trait[i].TotalDeathRate;
 - 3: Trait.TotalEventRate += Trait[i].TotalTraitRate;
 - 4: **end for**
-

2.2 Sampling the next event-time

Here will appear a, not yet mentioned, object that will not be explained further, called Dice. The Dice Object will provide a uniform or exponential random Variable.

Algorithm 8 sampleEventTime()

Require: -

Ensure: First ringing Eventclock has been sampled

- 1: double Parameter = Trait.TotalEventRate;
 - 2: double newEvent = this.Dice.RollExpDice(Parameter);
 - 3: Events.EventTimes.push(newEvent);
-

Here we use `Dice.RollExpDice(λ)` to get $X \sim \exp(\lambda)$. The same is possible for `Dice.RollUnifDice(λ)` to get $X \sim \text{Unif}[0, \lambda]$.

2.3 Changing a trait

Here we will work with the actual Events taking place. For this purpose i introduce the Events class like before the Trait class:

class Events

- Dice
- EventTimes[i]
- ChosenTrait[i]
- isBirth[]

The EventTime, ChosenTrait and isBirth are so called vectors. They are dynamic containers and we need to push an item into them to append it to the last entry.

Algorithm 9 `changeATrait()`

Require: -

Ensure: make a change to the Population with current Parameters

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

Algorithm 10 `choseTraitToChange()`

Require: -

Ensure: Trait is chosen for changing

- 1: double Parameter = Trait.TotalEventRate;
 - 2: double HittenTrait = Dice.rollUnif(Parameter);
 - 3: **for** i = 1 **to** n-1 **do**
 - 4: **if** HittenTrait \leq Trait[i].TotalEvent **then**
 - 5: this.ChosenTrait.push(i);
 - 6: break;
 - 7: **end if**
 - 8: HittenTrait -= Trait[i].TotalEvent;
 - 9: **end for**
-

Algorithm 11 choseEventType()

Require: -**Ensure:** Decision for Birth or Death is made

```
1: int i = Events.ChosenTrait.lastentry();
2: double EventType = Dice.rollUnif(Trait[i].TotalTraitRate);
3: if EventType ≤ Trait[i].TotalBirthRate then
4:   Events.isBirth.push(true);
5: else
6:   Events.isBirth.push(false);
7: end if
```

Algorithm 12 executeEventTypeOnTrait()

Require: -**Ensure:** Chosen event will occur on chosen trait

```
1: if isBirth then
2:   Trait[ChosenTrait.lastentry()] += 1;
3: else
4:   if ChosenTrait.Members > 0 then
5:     Trait[ChosenTrait.lastentry()] -= 1;
6:   end if
7: end if
```

List of Algorithms

1	EvolutionStep()	3
2	calculateEventRates()	4
3	calculateTotalBirthRates(StartIndex: i)	4
4	calculateTotalDeathRateOf(TraitIndex: i)	5
5	addTotalIntrinsicDeathRateOf(TraitIndex: i)	5
6	addTotalCompetitionDeathRateOf(TraitIndex: i)	5
7	calculateTotalEventRate()	5
8	sampleEventTime()	5
9	changeATrait()	6
10	choseTraitToChange()	6
11	choseEventType()	7
12	executeEventTypeOnTrait()	7

3 Fifth meeting (Fr 30.05): Improvements - afterwards

An diesem Tag habe bloß einige Änderungen vorgestellt und sichergestellt dass alle Funktionen ordnungsgemäß laufen:

3.1 Graphische Änderungen

Das waren hoffentlich alle optisch merklichen Änderungen:

- Ich habe eine Checkbox hinzugefügt die eine Option für sinnvollen Abbruch bietet. Diese Funktion prüft ob sich beide Populationen in einer */epsilon* Umgebung um den erwarteten Zustand befindet und stoppt anschließend die Iterationen. Zu Gunsten der Übersicht werden noch mal 1/3 der Anzahl bisher gemachten Iterationen angefügt.
Zu diesem Zeitpunkt ist $\epsilon = \frac{5}{\sqrt{K}}$.
- Es wurde ein Anzeigefehler behoben der für Geburts- und Todesraten immer nur die Populationsgrößen der Eigenschaften angezeigt hat.
- Auf dem Plotfenster habe ich ein Label mit den gemachten Iterationen hinzugefügt. Weiterhin findet sich dort jetzt auch ein Feld wo man einen Dateinamen eingeben kann der einem gespeicherten Bild dienen soll welches man mit dem neuen "save Image" Button erstellen kann. Das Bild wird zur Zeit als .png und .pdf gespeichert.
- Die x-Achsenbeschriftung lautet nun nicht mehr Millisekunden sondern "Time". Grund dafür ist dass ich nicht sicher bin welche Zeit unsere exponentielle Uhr ausgibt. Jedoch habe ich sie der Übersicht halber nicht mehr durch 1000 geteilt, da bei den gewünschten Instanzen ausreichend große Zeiten angenommen werden.
- Zuletzt wurde noch bei der Legende der für die k-te Erwartete Eigenschaft "k. Expected" zusätzlich der konvergierte Wert angezeigt "k. Exp: 1.048".

3.2 Interne Änderungen

- Ich habe mehrere Arten der Datenspeicherung ausprobiert. Zunächst habe ich die Daten alle nacheinander in eine "Storate" Datei geschrieben um schließlich einzeln die Daten mit "addData" zum Graphen hinzuzufügen. Das schien mir jedoch an der "addData" Stelle und am Datastream zu aufwendig. Danach bin ich auf eine alternative mit "Jumped Steps" umgestiegen. Die Jumped Steps basieren darauf das maximal 10.000.000 Punkte gespeichert werden und auch wenn viel mehr Iterationen gemacht werden, so werden nur Äquidistante 10 mio Stützstellen geplottet. Der Gedanke dahinter war die nicht Unterscheidbarkeit des Graphen für das menschliche Auge.

- Ich habe eine "GraphClass" hinzugefügt die alle Daten für das "PlotWindow" vorbereitet, so dass das Fenster nur noch darauf zugreifen muss. Damit habe ich eine Brücke zwischen den Fenstern und dem Plot geschlagen.
- Eine der wichtigsten Änderungen ist das Hinzufügen des QThreads. Dieser gewährleistet dass der Plot die Fenster nicht einfrieren lässt. Die Berechnung kann für mehr als 1mio Punkte bereits merkliche Zeit (3s) beanspruchen.
- ...to do...

3.3 bekannte Bugs

Es gibt hier noch einen Bug der mit der Beschriftung der x-Achse zu tun hat. Ich weiß noch nicht wie man ihn reproduziert, aber er könnte die Achse um einen großen Faktor zu groß skalieren. Der Plot weißt jedoch das geplante Verhalten auf und dieser Bug tritt nur bei replots; also dann auf wenn ein neuer Plot den alten überschreibt.

4 Sixth meeting (03.06): Introducing TSS

Einführung in TSS

- Das erste wäre das Einfügen einer Tabelle oder Grafik die einen guten Überblick über die Fitness der Population bietet. Diese gibt Aufschluss darüber ob TSS ordnungsgemäßes Verhalten zeigen wird. Allgemein ist es aber auch gut zu sehen welche Population wie stark auf eine andere wirkt, daher vielleicht schon ohne TSS sinnvoll. Dieses Feature wird in beiden Programmen vorhanden sein.
- Wenn das geschafft ist sollte ein umfassenderes Wissen über TSS angeeignet werden. Dazu wurde mir aus [2] S.1135 empfohlen.
- Anschließend wird alles notwendige organisiert und hier nachgetragen was notwendig ist um eine erste TSS Simulation laufen zu lassen.
- Wenn es nicht schon im Schritt darüber passiert ist, so sollte jetzt entschieden werden ob und wie man die x-Achse skaliert (logscale).
- Wenn ein ordentliches Bild produziert wird dann sollte anschließend eine gute Möglichkeit geboten werden Daten und Schranken für TSS Prozess Parameter zu setzen.