Junior Placement

Inferring demographic events in structured populations

Author :
Max Halford

Under :
M. Olivier Mazet
M. Lounès Chikhi

CMI SID

Gulbenkian Institute

INSA Toulouse

Due the 28th of August 2015

Contents

List of Figures

Acknowledgements

I would like to thank Olivier Mazet for being an overall great supervisor but also a genuine human being and having a clear-sighted mind.

I wish to express my gratitude towards Lounès Chikhi for his encouragements and all his advice.

My special thanks go to Willy Rodriguez, with whom I shared many interesting conversations.

I would also like to thank Bertrand Raquet, without him and a fortunate turn of events none of this would have happened.

I sincerely thank all the mathematicians I met for putting up with my questions.

Finally I am grateful to my friends and family for their proofreading.

Chapter 1

Introduction

1.1 Context

The context of my internship is atypical because it takes place between two research facilities. Lounès Chikhi is a population geneticist who works between the Laboratoire Evolution et Diversité Biologique at the Université Paul Sabatier and the *Instituto Gulbenkian de Ciência* in Oeiras (Portugal). Olivier Mazet works at the *Génie Mathématique et Modélisation* (GMM) department part of the INSA in Toulouse. Their collaboration is relatively recent but it has proven fruitful. They co-supervise one PhD student and have already published one scientific article, they have submitted another one and they are working on several additional papers. During 2014 they made a breakthrough and developed a novel way of explaining population changes.

1.2 Topic

The focus of the research team around Chikhi and Mazet is on modeling population genetics with mathematics. As with any research, models have to be validated against real data. In any domain there is never a "perfect" model that will take every possibility into account, a more reasonable question is to ask how well the model performs. An unavoidable number of simulations have to be run in order to decide.

1.3 Objectives

In population genetics research teams build models to explain genealogical phenomenons. These models can be *unstructured* and not take into account geographical properties of a population (such as internal migrations and secluded mating). *Structured* models try to be more realistic by accounting for these traits.

Models take inputs and return outputs. In this case the outputs are functions of a variable that enable to understand the properties of a population, for example it's size. As will be explained in the next chapter the *coalescent* allows to do just so by modeling the rate at which individuals find their common ancestor.

One can also look at this the other way round and wish to know what were the properties of a population if one knew what happened to it. In other terms my work is to find inputs given predetermined outputs. It turns out that this a very difficult problem. The problem's analytical form is complex. As such, numerical optimization algorithms are required.

My task is to find a way of inferring the good parameters for the model given a genealogical history (represented by a function) to fit. I have previous experience with genetic algorithms and have decided to give them a try. Genetic algorithms (GAs) are part of the heuristic optimization algorithms, they produce random candidate solutions and try to improve through even more randomness. The GAs proved to do reasonably well at first and after some tuning were performing very well.

Moreover, the solution that I produce has to be easily reusable by researchers. Indeed a parallel objective is to make a tidy package containing all the code necessary to use the method and a lot of explanations in order to allow other users to maintain the code.

1.4 Methodology

The researchers at both labs are all their own bosses, thus they have more flexibility then, say, the private sector. This *modus vivendi* undeniably affected the way I worked. In a sense I didn't apply many of the principles used in industry and taught in class. Instead I discovered a new way of working, which is a double-edged sword. On the one hand I had a lot of freedom and was able to ask an unlimited amount of questions, I had also had access to specialists on different mathematical topics. On the other hand I had a lot of responsibility and no defined work flow. In any case it was very interesting to discover the world of mathematical research.

Chapter 2

Coalescent Theory

2.1 Introduction

The goal of population genetics is to create a model which takes into account various phenomenons, such as

- Natural selection
- Genetic drift
- Mutation
- Gene flow and structure
- Recombination

Building a mathematical model to take all of these into account is one of the most important goals in modern population genetics.

The process known as *the Coalescent* has played a central role in modeling these phenomena for the past 30 years. Great advances have been, are and will be made into understanding population changes by building models based on the coalescent.

Indeed, having built a model, it can be compared to observed data and one can infer the right parameters. In this case we do so by comparing the output of the model to a curve (the PSMC).

In other words our goal is to build an alternative model that involves changes in gene flow instead of changes in population size. We also want to show that model can "mimick" population size models because of *fake signals*. In order to start doing this one has to understand the underlying statistical properties of biological genealogy.

2.2 The coalescent and different representations

By studying the distribution of a gene's versions (called *alleles*) in a population, one can avoid studying the whole genetic code of every individual of a population.

Two individuals are said to "have the same parent" if they inherit their allele from the said parent, who is called the *common ancestor*. The event where two individuals from a population find their common ancestor is called a *coalescence*. The object that is being studied is called the time to coalesce, denoted T_n. For the sake of simplicity we will only be studying coalescence times for two individuals, we call these T_2.

Before delving into the mathematics, it is important to get the intuition behind the coalescent. Consider the following figure.

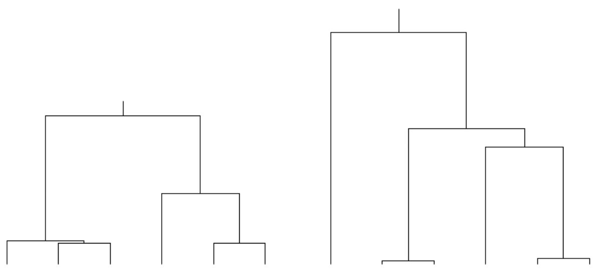


Figure 2.1: Genealogical history examples

In the figure every leaf represents an individual and every node represents a coalescence. In other words when two leaves join into one node they have the same parent. The intuition to have is that **the time to coalescence largely depends on the size of the population**. Indeed, under certain statistical hypothesises, if there are a lot of individuals in the previous generation, the chance that two randomly chosen individuals in the current generation have the same parent is low. The other way round is what interests us: if the time to coalesce is low then the population is low and *vice-versa*. In other (crude) words if we know T_n then we can infer the size of the population.

2.2.1 Wright-Fisher Model

Say we have a group of individuals and we want to trace back their *Most Recent Common Ancestor* (MRCA). The chain of links between children and parents is called *lineage*. By continuing the process back in time the number of distinct lineages will decrease and eventually there will only remain a single one and we will have found the MRCA of all the initial individuals. We consider that a particular allele doesn't influence the ability to survive of an individual, which is of course debatable however is also a necessary simplification.

If every generation has the same size n, every individual inherits their genotype from a particular parent from the previous generation with probability $\frac{1}{n}$. It is fairly easy to see that if we take a population of size n and sample two individuals, the probability that they coalesce is

$$\approx \binom{2}{2} \times \frac{1}{n} = \frac{1}{n} \tag{2.1}$$

where the first factor is the number of ways of picking two individuals from a group of size 2 and the second factor is the probability given previously. The first factor is not important here but it is for samples of three individuals or more. The reason why there is an approximation symbol is that we ignore the cases where more than two pairs of individuals coalesce or that a group of three or more individuals coalesce, indeed these events are of order $\frac{1}{n^2}$.

It follows the probability that no pair of individuals coalesce is

$$1 - \frac{1}{n} \tag{2.2}$$

Thus the probability of **no** coalescence happening for m generations is

$$(1 - \frac{1}{n})^m \tag{2.3}$$

When x is small, $(1 - x) \approx e^{-x}$ (*cf. Taylor series*). This enables us to write (2.3) as

$$\approx e^{-\frac{m}{n}} \tag{2.4}$$

The enables us to pass into continuous time. We can recall that an exponential distribution with rate λ is defined by $P(T > t) = e^{-t\lambda}$ and has mean $1/\lambda$. In this case the mean is $\frac{1}{n}$.

The Wright-Fisher (WF) is a basic model and is anterior to the coalescent model. It is a good introduction and gives an idea of what is being dealt with. However it does not take into account phenomenons listed above. A more in-depth presentation can be read in *Durrett 2008, Chapter 1*.

2.2.2 Symmetrical islands model

The basic Wright-Fisher model doesn't take into account population structure. In the natural world it often occurs that populations live in colonies or *islands*.

For example two islands of mammals can exchange individuals at certain rates and mating only occurs in the islands.

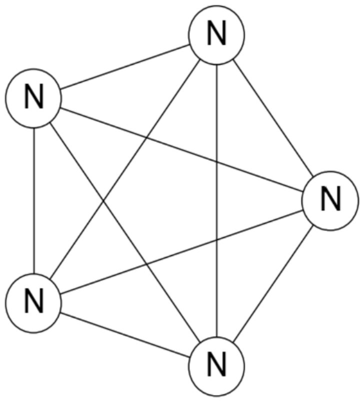


Figure 2.2: Islands configuration example

For the sake of simplicity we consider that each island exchanges individuals with other islands at the same rate (thus the term *symmetrical*) and that all the islands are connected. In this model the distribution of T_2 is harder to model because it depends on where are the individuals at certain time steps. For example they can be in different islands etc.

The mathematics for this model are slightly more complicated. *Mazet et al* have used a model based on this [4], thus it will be covered in Chapter 3.

A (silly) remark has to be made: there has to be at least two islands, else migrations wouldn't be possible.

2.2.3 Other models

There are many other models that try to integrate genetic phenomenons so as to be as realistic as possible. The object that each model studies is the distribution of the time to reach the MRCA for k individuals, denoted T_MRCA. A very popular model named *PSMC* was presented in a landmark paper called *Inference of human population history from individual whole-genome sequences* (Li & Durbin 2010 [3]). This method is widely used throughout the population genetics community.

The *Pairwise Sequentially Markovian Coalescent* (PSMC) is a method and a software l that aims at identifying population size changes over time through Hidden Markov Chains and the Expectation-Maximization algorithm.

For simulations we will be using MS [1], a C software with a shell wrapper

[1]MS download and documentation: `http://home.uchicago.edu/rhudson1/`

written by Richard Hudson. It is well known among geneticists. It can do many things related to genealogy. A more in depth description is available in the appendix.

2.3 Objectives

First of all the model has to be mathematically detailed and implemented with a programming language. After having validated the implementation it can be used to fit given data sets in order to infer parameters. Finally a tool has to be built to do so automatically.

Chapter 3

Symmetrical Island model with Changes in the Migration Rates

3.1 A continuous time Markov Chain approach

Consider a *Symmetrical Island Model* (Wright 1941 [5], Durrett 2008 [1]). This model is ruled by two parameters: the number of islands n and the migration rate M. We are interested in the study of the coalescence time of two genes (individuals), denoted T_2. The history, going backwards in the time, of two genes sampled in the present can be modeled as a *continuous time Markov chain* (Herbots 1994 [2]). When tracing the history of two lineages backwards in time, three scenarios (states) are possible:

- They are in the **s**ame island - we denote this state 1 or s.
- They are in **d**ifferent islands - we denote this state 2 or d.
- They have already **c**oalesced - we denote this state 3 or c.

Note that the state c is **absorbing**. Let Q be the *infinitesimal generator* (rate matrix) of this process. The transition semigroup of the process can be computed via the matrix exponentiation:

$$P_t = e^{tQ}, \ \forall t > 0. \tag{3.1}$$

The coefficient $P_t(i,j)$ represents the conditional probability of being in state j at time t, knowing that the process started in state i at time 0.

Let us now compute the coefficients of the Q matrix.

If at time $t = 0$ we are in the state s (both lineages in the same island), a coalescence event happens with rate 1 and a migration event with rate M. Any of these events will make the process change it's state. This implies that the first row of our Q matrix will be $(-(M+1), M, 1)$ (the row cells have to add up to 0 in a infinitesimal generator).

If we are in state d at $t = 0$ (the lineages are in different islands), it is not possible to go to the state c because the two lineages must be in the same island in order to coalesce. Hence, the only state the process can move to is s and this happens at rate $\frac{M}{n-1}$ (a migration event happens at rate M and we have a probability of $\frac{1}{n-1}$ that the migrating lineage migrates exactly to the island where the other lineage actually is).

We thus have:

$$Q = \begin{pmatrix} -(M+1) & M & 1 \\ \frac{M}{n-1} & -\frac{M}{n-1} & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{3.2}$$

We consider two individuals chosen at random from the population, and we denote A_1 and A_2 the events: "the two individuals are from the same island", respectively "the two individuals are from different islands".

For $i = 1, 2$ we have $P(T_2 \leq t | A_i) = P_t(i, 3)$.

The distribution function of the coalescence time T_2 for the two individuals is then given by:

$$\begin{aligned} F_{T_2}(t) = P(T_2 \leq t) &= P(A_1)P(T_2 \leq t|A_1) + P(A_2)P(T_2 \leq t|A_2) \\ &= \frac{1}{n} P_t(1,3) + \frac{n-1}{n} P_t(2,3) \\ &= \frac{1}{n} P(T_2^s \leq t) + \frac{n-1}{n} P(T_2^d \leq t). \end{aligned}$$

3.1.1 Conditioning

In order to compute the distribution function of T_2 when the migration rate changes, we will use some kind of "memoryless" property. As we will see this can be very handy for computational purposes. More precisely, we will write the probability $P(T_2 \leq t)$ in terms of $P(T_2 \leq u)$ and $P(T_2 \leq t-u)$ for any $u \leq t$. To do that, we will use the transition semigroup P_t.

Let us fix some $u > 0$ such that $u \leq t$. At time u the probability of being in a given state, knowing the initial state, is one of the entries of the matrix $P_u = e^{uQ}$.

Let us consider that the process started in state s (two haploid individuals where sampled from the same island). Using the Markov property of the process, we can compute the distribution function of their coalescent time T_2^s by conditioning on the state of the process at time u:

$$P(T_2^s \leq t) = P_u(1,1)P(T_2^s \leq t-u)+P_u(1,2)P(T_2^d \leq t-u)+P_u(1,3). \quad (3.3)$$

It is possible to write this probability more compactly by using just the transition semigroup:

$$P(T_2^s \leq t) = P_u(1,1)P_{t-u}(1,3) + P_u(1,2)P_{t-u}(2,3) + P_u(1,3). \quad (3.4)$$

The above formula equals the scalar product of the first row of P_u with the third column of P_{t-u}.

In a similar manner, we can compute the distribution function of the coalescent time T_2^d for two individuals sampled from two different islands:

$$P(T_2^d \leq t) = P_u(2,1)P_{t-u}(1,3) + P_u(2,2)P_{t-u}(2,3) + P_u(2,3). \quad (3.5)$$

The relations 3.4 and 3.5 come from the following property of the transition semigroup P_t:

$$P_t = P_u P_{t-u},\ \forall 0 \leq u \leq t.$$

3.2 Changing the migration rate

Equations 3.4 and 3.5 give a way to “stop" and “restart" the process at any given time u. This means that, using the Markov property of the process, we are able to compute the distribution function of T_2 at any time t, by conditioning on the state of the process at any time u between 0 and t. This property turns to be very useful for computing the distribution of T_2 under the assumption that migration rate changes from $M = M_0$ to $M = M_1$ at time $t = T$ backwards in time.

We can use equations 3.4 and 3.5 to “stop" the process at time $t = T$ and restart it again with a modified infinitesimal generator.

From time 0 to T (backwards in time) the process evolves like a symmetrical island model with n islands and migration rate M_0. From time T to infinity the process is still a symmetrical n-island model, but the migration rate changes to M_1.

We thus use one infinitesimal generator Q^0, given by 3.2 with $M = M_0$ (and the corresponding P_t^0 and $F_{T_2}^0$) in the time interval $[0, T[$, and a modified infinitesimal generator Q^1, given by 3.2 with $M = M_1$ (and the corresponding P_t^1 and $F^0_{T_2}$) in the time interval $[T, \infty[$.

This leads us to write the distribution function of T_2 in this symmetrical island model with migration rate change as:

$$F_{T_2^s}(t) = \begin{cases} P_t^0(1,3), & \text{if } t \leq T \\ P_T^0(1,1)P^1_{t-T}(1,3) + P_T^0(1,2)P^1_{t-T}(2,3) + P_T^0(1,3) & \text{otherwise,} \end{cases} \tag{3.6}$$

or equivalently

$$F_{T_2^s}(t) = \begin{cases} F^0_{T_2^s}(t), & \text{if } t \leq T \\ P_T^0(1,1)F^1_{T_2^s}(t-T) + P_T^0(1,2)F^1_{T_2^d}(t-T) + P_T^0(1,3) & \text{otherwise.} \end{cases} \tag{3.7}$$

We also have

$$F_{T_2^d}(t) = \begin{cases} P_t^0(2,3), & \text{if } t \leq T \\ P_T^0(2,1)P^1_{t-T}(1,3) + P_T^0(2,2)P^1_{t-T}(2,3) + P_T^0(2,3) & \text{otherwise,} \end{cases} \tag{3.8}$$

or equivalently

$$F_{T_2^d}(t) = \begin{cases} F^0_{T_2^d}(t), & \text{if } t \leq T \\ P_T^0(2,1)F^1_{T_2^s}(t-T) + P_T^0(2,2)F^1_{T_2^d}(t-T) + P_T^0(2,3) & \text{otherwise.} \end{cases} \tag{3.9}$$

By the Markov property, the transition semigroup $\tilde{P}_t$ of the new markovian jump process is given for $t \geq T$ by the product

$$\tilde{P}_t = P_T^0 P^1_{t-T},$$

where P_t^0 (respectively P_t^1) is the transition semigroup corresponding to a migration rate $M = M_0$ (respectively $M = M_1$). Obviously, for $t \leq T$, we have $\tilde{P}_t = P_t^0$.

This can be extended to an arbitrary number of changes in the migration rate.

$$\tilde{P}_t = P_t^{(1)}\mathbb{1}_{t<T_1} + P_{T_1}^{(1)}P_{t-T_1}^{(2)}\mathbb{1}_{T_1 \leq t < T_2} + P_{T_1}^{(1)}P_{T_2-T_1}^{(2)}P_{t-T_2}^{(3)}\mathbb{1}_{T_2 \leq t < T_3} + \dots$$
$$\dots + \prod_{i=1}^{k-1} P^{(i)}_{T_i - T_{i-1}} P^{(k)}_{t-T_{k-1}} \mathbb{1}_{T_{k-1} \leq t}. \tag{3.10}$$

We are interested in the *Inverse Instantaneous Coalescence Rate* (IICR), which is given by

$$IICR(t) = \frac{1 - F_{T_2}(t)}{f_{T_2}(t)} \tag{3.11}$$

Because of the matrix representation this boils down to

$$IICR(t) = \frac{1 - P_t[0][2]}{P_t[0][0]} \tag{3.12}$$

where P_t is the exponential of the infinitesimal generator at time t.

3.3 Implementing the model

Now that the mathematics are defined the model has to be transcribed into computer code. The resulting script is called `model.py`. As with all the pieces of code they are quite long so they are not detailed, however they are all available on GitHub (details in the appendix). Here is the pseudo-algorithm:

Listing 1: Model

```
n = Integer
T = list of increasing positive floats
M = list of positive floats
S = list of time steps
Q = infinitesimalGenerator(n, m) for each m in M
D = diagonalize(q) for each q in Q
E = exponential(d) for each d in D
evaluate(E, s) for each s in S
extract each PDF and each CDF
```

By running a profiler on the code (in this case the project[1] written by Robert Kern) on the code, it is fairly obvious that computing the exponential of the matrix is very hefty, we can sidestep this problem by diagonalizing Q. Indeed after verification the algorithm used by *numpy* is the *Padé approximant*. This algorithm is of complexity $O(n^3 log(n))$, which is *very* bad. It is possible to remove this bottleneck by noticing that if

$$e^{tQ} = \sum_{k=1}^{\infty} \frac{(tQ)^k}{k!} \tag{3.13}$$

[1]Python profiler:`https://github.com/rkern/line_profiler`

it follows that by diagonalizing Q

$$\sum_{k=1}^{\infty} \frac{(tQ)^k}{k!} = \sum_{k=1}^{\infty} \frac{P(tD)^k P^{-1}}{k!} \tag{3.14}$$

which is

$$\sum_{k=1}^{\infty} \frac{P(tD)^k P^{-1}}{k!} = P e^{tD} P^{-1} \tag{3.15}$$

However it follows from Sylvester's formula that the exponential of a diagonal matrix is the matrix with each element of the diagonal elevated to the exponential, which is very easy to compute. What is nice in this case is that P, D and P^{-1} are reusable because they don't depend on t anymore. Indeed the algorithm is now much faster because the diagonalization of Q only has to be computed once for each model.

Furthermore in the algorithm the matrix exponential has to be computed for each t part of a given list of positive increasing times. We can use 3.4 to avoid computing it for every migration rate by performing the following dodge: instead of computing e^{tD} for each M we exhibit the memoryless property of the exponential law.

We don't have to compute the matrix exponential for each migration rate, we just have to compute it for the latest migration rate. Algorithmically we store the matrix exponentials for each migration rate. Then when evaluating the model at one time we only have to compute the matrix exponential for the latest migration rate. This greatly reduces the dot products to be computed. Indeed, at worst there will only be one product to be computed, at best none. If we didn't do this there would at worst be k, where k is the number of different migration rates.

3.4 Example

The model is evaluated at given steps, hence the discrete plotting. The dotted lines are the times at which the migration rate change. In this example there are four changes in migration rates, which is equivalent to saying that there are five different migration rates. The script for generating these is `plotting.py`.

In this particular example the IICR goes down and then up again going back in time. This is called a *bottleneck* and happens a lot with mammal populations. In layman's terms it is a sudden reduction in population size, often due to diseases or bad climatic conditions.

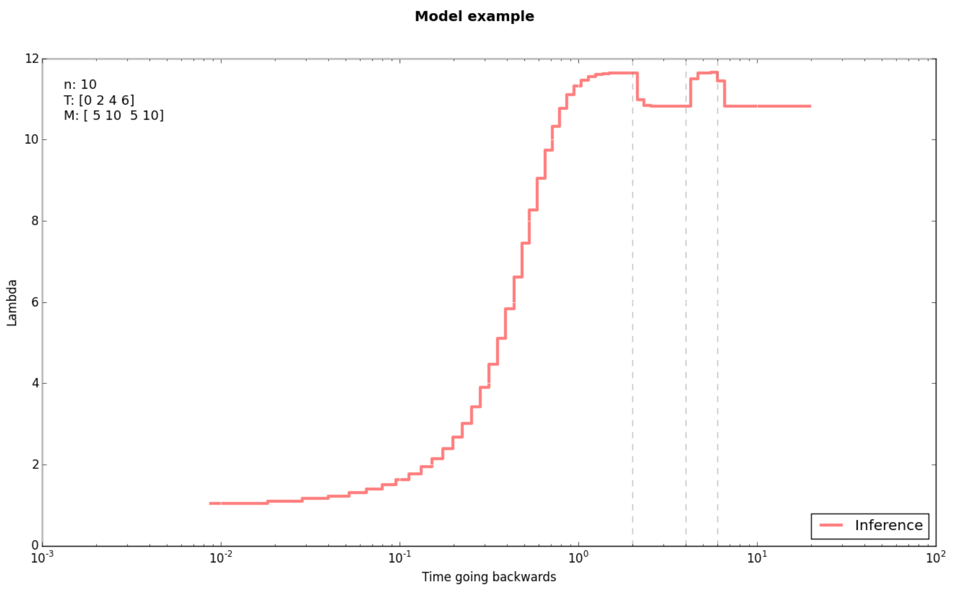


Figure 3.1: Model output example

3.5 Validation

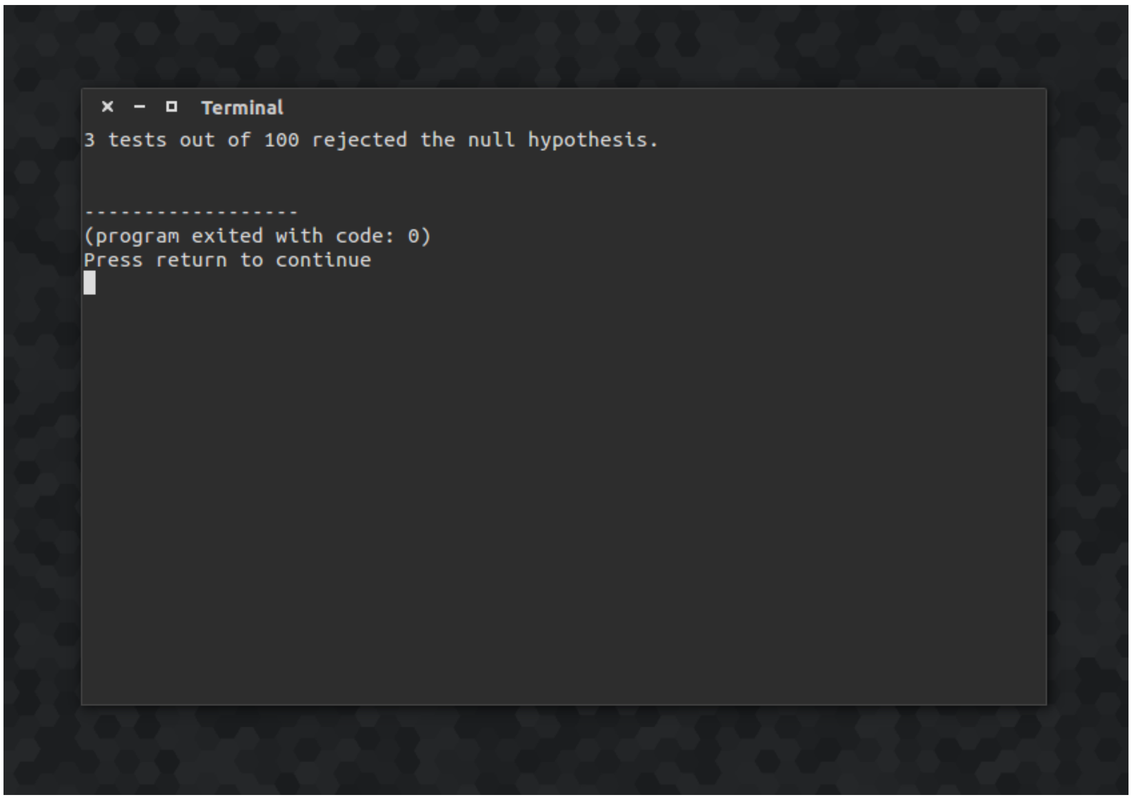
The way we modeled the problem is elegant and practical but we have to make sure it returns correct distributions. For example there is no mention of Wright-Fisher or any hypothesis for that matter.

In order to validate the model it can be compared to simulated data under a symmetrical island model produced by MS (which bases itself on the Wright-Fisher model described in Chapter 2). In our case we can ask it to produce T_2 values for given parameters. These will serve as benchmarks so as to see if the implementation is correct. For this one can define an artificial scenario, for example

- n: 10
- T: [0, 10, 20]
- M: [1, 10, 5]

The T2s given by MS is a random variable whose expected value is the true distribution which we should obtain with the model.

In order to verify that the models returns correct results we can use a *Kolmogorov-Smirnoff* test for $\alpha = 0.05$ one hundred times and check how many times the test rejected the null hypothesis (both observations originate from the same distribution).



```
3 tests out of 100 rejected the null hypothesis.


------------------
(program exited with code: 0)
Press return to continue
```

Figure 3.2: Result of Kolmogorov-Smirnoff tests

The results of these KS tests are very satisfactory. There was not much worth in pursuing the KS tests for other scenarios, mainly because it can't be a coincidence that the implementation only works for this randomly chosen scenario, if it was badly implemented the KS tests would have detected it. All the code for the KS tests in the `testmodel.py` script. A simple interface is available in the `model_validation.py` script.

We can safely say that the model predicts the outcome of a genealogical scenario.

3.6 Further objectives

We now have something of the shape

$$f(X) = Y \tag{3.16}$$

which in biological terms is equivalent to saying: **"Based on a set of parameters (X) what will be the IICR (Y) over time according to the model?"**

That is the easy part, it doesn't serve any great purpose. What we would now want is

$$f^{-1}(Y) = X \tag{3.17}$$

which is like saying **"If we know the size of a population over time then what are the parameters according to the model?"**. This is a far more difficult question and is the main concern of my internship.

This question boils down to an optimization problem. Indeed we want to find the X that minimizes the distance between $f(X)$ and Y.

As mentioned in the introduction classic "linear" optimization methods didn't do well in solving the problem. In light of this I decided to use the tool I know best for complicated optimization problems: **Genetic Algorithms**.

Chapter 4

Building a Genetic Algorithm

4.1 Reasoning

Genetic algorithms (**GA**) are a good way of abstracting oneself from a difficult problem. GAs look at how the inputs to a problem affect the output. The general idea behind GAs and heuristic algorithms in general is to **find a clever way of exploring the search space**. The main advantage of GAs is that their implementation doesn't change much for every problem, in fact the only variable that changes is the problem itself. Indeed heuristic algorithms only need to measure and compare the effect of variables (see 4.2). Finally this allows us to have much more power of the algorithm; using a black box is not a good idea.

I decided to code my own genetic algorithm. First of all it's a good learning experience. Secondly the way the problem is put requires shaping the search space in a particular shape. Finally a homemade implementation enables much more power over the optimization process; on the contrary a black box is limiting.

4.2 Variables

As a reminder these are the variables the model takes into account:

Variable	Description	Search space
n	Number of islands	$[2; +\infty[$
s	Number of flow rates	$\mathbb{N}^*$
T	Times of flow rate changes	$\mathbb{R}^{s+}$
M	Flow rates	$\mathbb{R}^{s+}$

Obviously there has to be at least two islands, hence the 2 in the search space. The flow rates nor their number can be negative and the list of times of flow rate changes has to be monotonous (because we are going backwards in time). The first flow rate time is always 0. Also there has to be at least one flow rate (the initial one).

A computer doesn't understand infinity, hence we have to feed it boundaries. For mammals, the number of islands doesn't generally exceed 200. The only reason one would go above would be for insect colonies or for bacteria strains.

The number of flow rates makes the search space explode. Here lies the greatest difficulty of the problem: we don't know how many switches occurred, hence the number of variables is itself a variable. Explicitly the number of variables to infer is $2s - 1 + 1 = 2s$.

For the sake of simplicity the number of flows can be fixed. In this case the size of T and M do not change, which hugely reduces the search space. In a sense this isn't an unreasonable decision. When a PSMC curve goes up, down and then up gain this can be reproduced with three different migration rates; adding more migration rates will allow better fits (because it gives more flexibility to the model) but it is overkill and not elegant.

Genetic algorithms need a heuristic on which they base themselves to sort the best candidates. In genetic algorithms terms this is called a *fitness function*. In this case the objective is to fit a curve on a set of points, thus we can use least squares,

$$fitness = \sum_{i=1}^{n} (y_i^{obs} - y_i^{mod})^2 \tag{4.1}$$

Of course this is a naive first guess, if the GA doesn't do well there might be progress to be made with the choice in fitness function.

For this particular problem the fitness has to be minimized (indeed the GA has to fit a curve), thus the difference between y_i^{obs} (the PSMC points) and y_i^{mod} (the model points) has to be the smallest possible. The n points are given with the curve to fit. At every time step there is a matching value, which we call the *lambda* value (or IICR). The problem is now the following: given s, find the n, T, M which minimize fitness.

On a sidenote, this problem is nice in the way that the end result is a comparison between two curves, thus we can understand where the algorithm struggles by observing the evolution of the curve and by using a bit of imagination.

4.3 Process

The general flow of a genetic algorithm is the following.

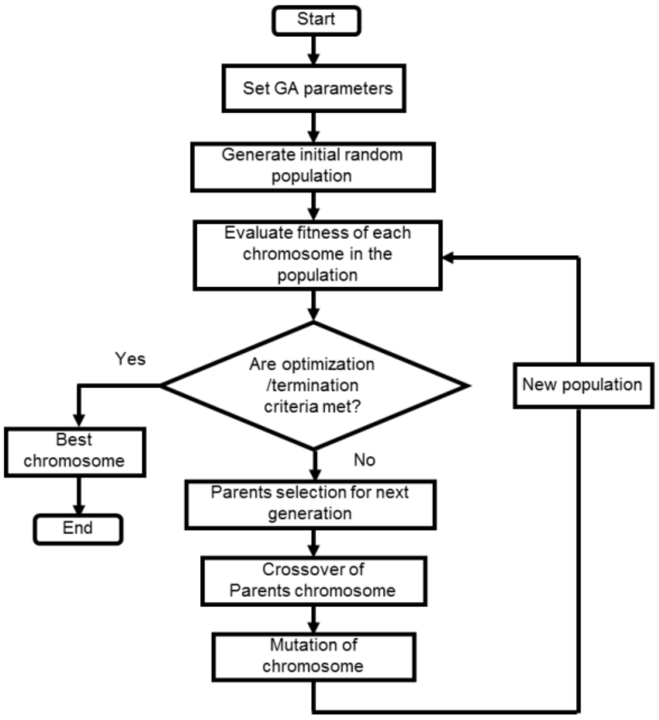

Start
Set GA parameters
Generate initial random population
Evaluate fitness of each chromosome in the population
Are optimization /termination criteria met?
Yes
Best chromosome
End
No
Parents selection for next generation
Crossover of Parents chromosome
Mutation of chromosome
New population

Figure 4.1: Genetic algorithm flow

However every genetic algorithm is specific to a given problem. The algorithm's pseudo-code is the following.

Listing 2: Genetic algorithm routine

```
population = generateNaiveModels()
population.evaluate()
population.findBest()
for i in range(nbIterations):
        for individual in population.best:
                individual.enhance(variable)
        population.evaluate()
        population.findBest()
```

4.3.1 Instantiation

The first step is to generate random sets of variables; we call these *individuals* (a group of individuals is called a *population*). There is no prior as to what the optimum we are looking for resembles; thus the variables are distributed in a uniform manner. Concretely upper bounds have to be given, one for the number of islands, one for the migration rate times and one for the migration rates. As mentioned previously 100 islands is a solid upper boundary for the cases the algorithm will face. The boundary for the migration rates isn't meaningful because the way in which the search space will be explored relative to the migration rates will be unbounded. Because the model has to fit a curve the time boundary is simply the highest time of the curve to fit.

4.3.2 Evaluation

Once an initial population has been generated it has to be evaluated in order to sort the individuals. The evaluation assigns a *fitness* to each individual. In our case the fitness is a least squares evaluation. Explicitly, for each individual, we create a model described in Chapter 3 with the individual's parameters and evaluate it for each time step of the curve that has to be fitted. Evaluating the least squares is then a simple task, ie. we simply have to apply 4.1.

4.3.3 Selection

Once the population has been sorted according to it's fitness a "best sample" has to be chosen in order to generate new individuals. The definition of best sample is complex and is a matter of good judgement and experience. A simple method (called *elitism*) is to select the n best individuals of a population and that's it. Intuitively this is a good idea if the space on which the optimization is done is convex. However, because the selected individuals are going to generate new individuals, they will propagate and "influence" further generations. In other terms they will converge to a solution. If the search space is hilly, then maybe "weak" solutions have to be considered in order to reach better ones. Thus I propose to rather used *tournament selection*:

- Choose n individuals at random. - Keep the best. - Repeat m times.

Although this seems simple it is rather powerful because it allows selecting bad individuals. Indeed the n chosen individuals can all be weak and then *the strongest of the weak* will *win the tournament*. What is interesting is that if n is increased then strong individuals have a higher chance to participate and thus to win. Indeed if n is the size of the population then tournament selection becomes elitism. The output sample will comprise m individuals, which are the winners of m tournaments.

4.3.4 Breeding

Once a sample of individuals has been chosen we can begin generating new individuals, called *offspring*. There are two ways of generating a new individual. The first one is called *crossover* but I am not a firm believer of it. In most GA tutorials/papers crossover is presented as the main breeding method. For implementing crossover one has to devise a way of "mixing" two individuals. In my experience a good way of performing a mix isn't always obvious and sometimes new individuals can take a turn for the worst. The second one is called *mutation*. The idea of mutation is to modify a variable to a new value in it's neighborhood. This implies a lot of possibilities. In the literature mutation is only seen as way of not getting stuck local optimums, however I use it as the main breeding method. The mutation procedure is different for every kind of variable:

- n: the number of islands is discrete and has to be superior or equal to 2. The mutation procedure is not too complex, the algorithm simply increases or decreases by 1.
- T: as mentioned the times have to be increasing ($t_1 < t_2 < ... < t_i < ... < t_s$). A first initial mutation procedure is that a chosen t takes a random value between it's adjacent values. The first time is always 0. An artificial upper adjacent value can be set for the last time ($t_{s-1} < t_s < t_s + (t_s - t_{s-1}) = t_{s-1} < t_s < 2t_s - t_{s-1}$).
- M: the migration rates can take any random value in their neighborhood. Explicitly $m' \in [m - \alpha, m + \alpha]$ where α is a chosen parameter. The parameter doesn't really matter because this is all a question of convergence. The higher the parameter the faster the algorithm will converge. The smaller it is the more it will be, at the expense of computational cost. The migration rates have to be positive.

The number of offsprings that each individual generates is of course another parameter that has to be specified. If m individuals were selected to breed and each individuals generates c children then the new population will comprise $m \times c$ individuals.

4.3.5 Iterating

Once a new population has been generated the process loops. The obvious question is "*Till when?*". The general motto is "*The more, the better, the costlier.*" The algorithm will converge at one point, checking if it has and stopping the algorithm is a good idea.

4.4 First results

Now that the skeleton of the algorithm is coded (`genalg.py`) we can start evaluating it's performance. To get a glance of how well it is doing we can test with a very simple scenario. A PhD. student at the laboratory has some PSMC files available so I decide to use them. PSMC files are always in the same format, parsing them can thus be automated (`psmcfit.py`). The interface to the algorithm is quite neat (it's arranged in classes to make the most of object-oriented programming). Here is an example of reading PSMC data and fitting it with the GA:

Listing 3: PSMC fitting example

```python
from lib.inference import genalg
from lib import model, psmcfit, plotting

psmc = psmcfit.get_psmc_history('simulations/
0_switch/experiment_2.psmc')

times = psmc['times']
lambdas = psmc['lambdas']

individuals = genalg.Population(model.StSICMR,
times, lambdas, maxIslands=100, switches=0,
size=500)

individuals.enhance(150)

plotting.plotModel(individuals.best.model, times,
lambdas, logScale=True)
```

4.4.1 No switches

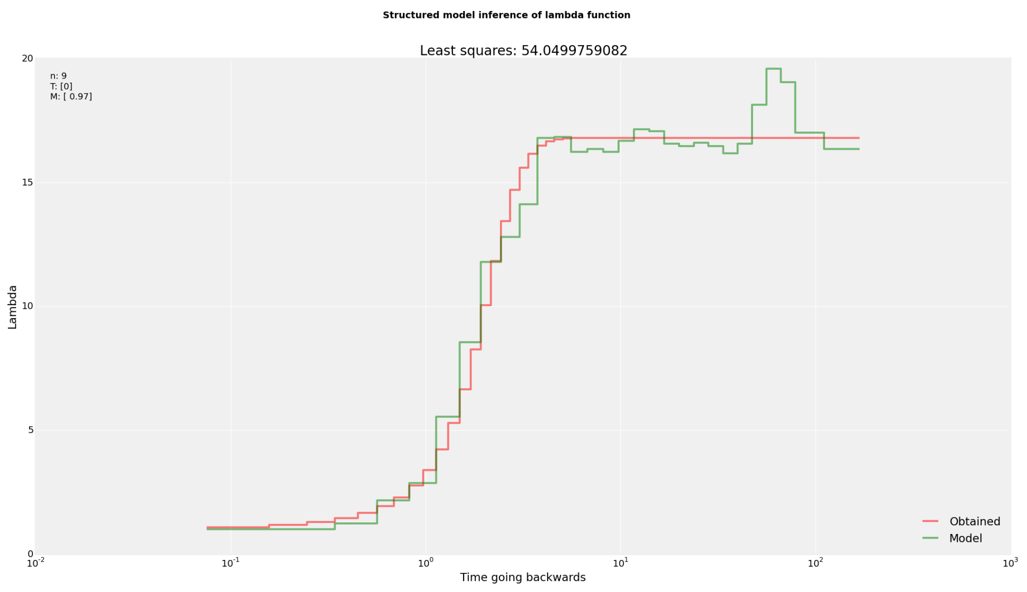


Figure 4.2: No switches PSMC fitting

4.4.2 Four switches

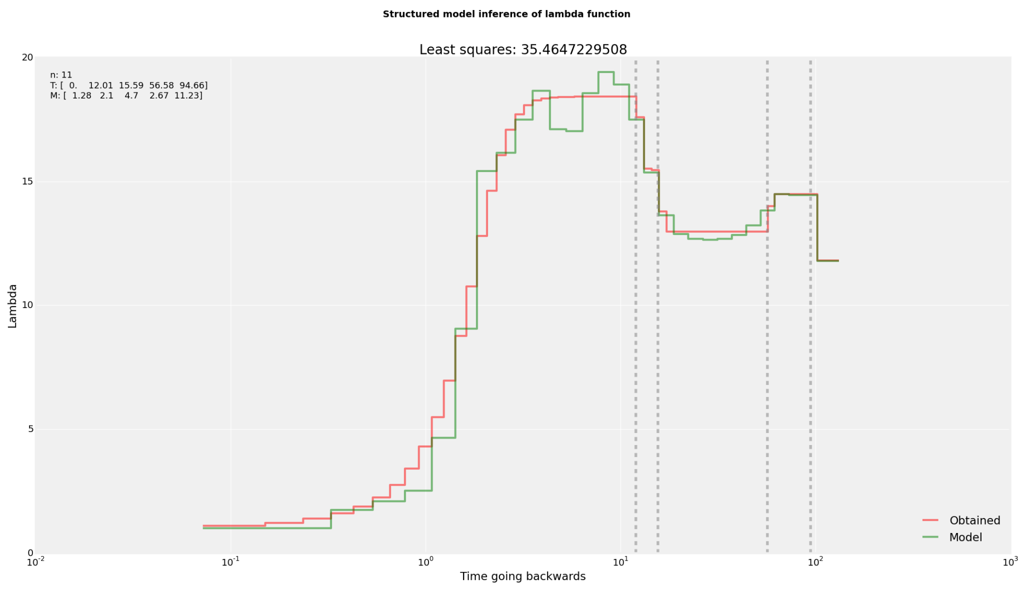


Figure 4.3: Four switches PSMC fitting

4.4.3 Remarks

The results shown above are very satisfactory. Of course the first scenario doesn't incorporate any switch in migration rate and so only has two parameters. Nevertheless the algorithm does really well. In the second scenario the algorithm also did extremely well. However a remark has to be made on the stability of the algorithm. At this point around 2 out of 5 algorithm runs do well, the other are utterly wrong; this is a problem to be worked upon.

At this point of the internship I had to consult my supervisors to know how well I did. They were happy and impressed, they gave me two new objectives:

- Making the algorithm more robust and less dependent on randomness.
- Validating the algorithm by applying it to many more scenarios.

4.5 Tuning the algorithm

4.5.1 Finding n quickly

After a few iterations it is fairly obvious that the scale of the curve is highly correlated with the number of islands. Thus to set the algorithm on the right track it is a huge improvement to generate the most initial individuals possible to cover the most number of islands possible. The idea is to generate many individuals whose number of islands follows a uniform law between 2 and the specified upper boundary. By doing this the algorithm knows very quickly that the number of islands is, say for example, between 58-62 or 8-12 because n is the parameter that affects the output most. In other words, finding the number of islands gives the algorithm a good idea of where the curve should be.

4.5.2 Changing the mutation method

The method for mutating T and M is a matter of concern. The times can change uniformly in an interval between their neighbours. This seems a bit constrained and there is no basis using this method. The same goes for the mutation rates, they take a new value in a closed interval. This doesn't seem like a problem because the intervals will "stack" as the generations go through and the mutation rates will reach their goal. However, the migration rate has to change a lot in order to obtain a better fit. This is a classic problem, ie. getting stuck in a local optimum. This is illustrated in the following image.

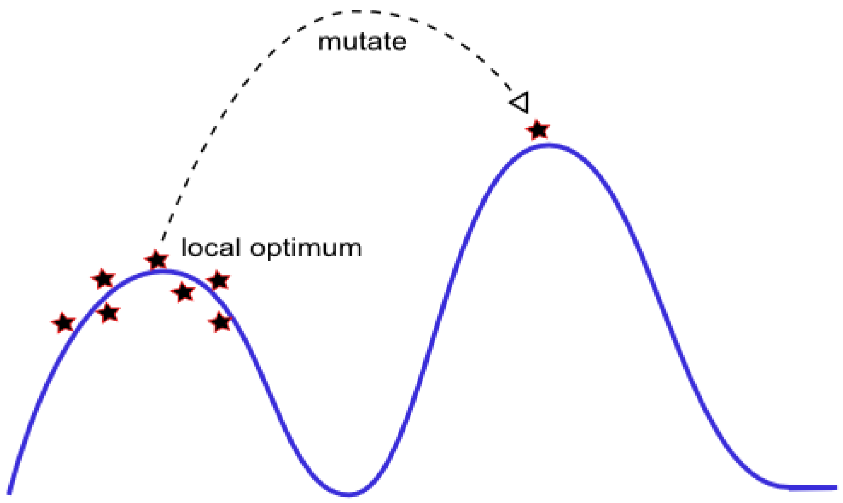


Figure 4.4: Escaping a local optimum

A better mutation method would be to allow any new possible value, assigning more weight to closer values and less to further values. I call this the *miracle mutation*. In some ways it mimics natural evolution where once in a while an unexpected mutation occurs and turns out to be a step forward. After fiddling with different possibilities I thought of using a *normal law* for the mutation of both the times and the rates.

The idea is to consider a time/rate as a random variable following a normal law. The mean is the current value and the variance is a parameter.

$$x_{i+1} \sim \mathcal{N}(x_i, \sigma) \tag{4.2}$$

First of all with this method x_{i+1} can take *any* value because a normal law is not bounded. However it will assign more weight to closer values and less to further ones, which is exactly the aim. Second of all the variance (σ) can be used as the mutation rate of the genetic algorithm, specifically the search space exploration breadth can easily be modified just by changing σ.

4.5.3 Allowing more varied mutations

At the moment the algorithm picks a variable at random and mutates it. It would be nice to give some more freedom to the algorithm. First of all instead of picking one variable it should pick a random subset of random size. Also it should be able to pick a time and a migration rate couple and mutate them both at the same time. These are not too hard to implement in Python and are implemented in the `mutate` procedures in `genalg.py`.

4.5.4 Further results

The changes implemented in section 4.5 prove to be very good. Maybe they are not the optimal way of proceeding but the GA results are much more satisfying. Indeed the algorithm is much more stable. Of course the fits are not always the same but none of them are aberrant. In the first version of the GA the solutions were most probably getting stuck on local optimums with no way of escaping.

4.6 Validating the algorithm

The results above may seem good, however concluding on the efficiency of the algorithm is a bold assumption. In order to validate the algorithm a series of *unit tests* have to be put in place. Concretely the algorithm has to "face" various curve typologies, generally it has to be pushed to it's limits.

The curve typologies have to be different and of varied complexity. The `model.py` script can be used to generate curves. Doing this will enable us to numerically measure how well the algorithm does because the underlying parameters of the curves will be known.

4.6.1 Procedure

The following setups will be tested:

- Migration rate switches: $[2, 4, 8]$
- Highest time: $[1, 10, 50, 100]$
- Islands: $[2, 5, 10, 50, 100]$

Moreover, for every number of migration rates the shapes of the curves can be different. For example they can go UP-DOWN-UP or UP-UP-DOWN-UP etc.

- 2 switches
  - $[1, 10, 1]$
  - $[10, 1, 10]$
- 4 switches
  - $[1, 10, 1, 10, 1]$
  - $[10, 1, 10, 1, 10]$
  - $[1, 5, 10, 5, 1]$

    - $[10, 5, 1, 5, 10]$

- 8 switches
    - $[1, 10, 1, 10, 1, 10, 1, 10, 1]$
    - $[10, 1, 10, 1, 10, 1, 10, 1, 10]$
    - $[1, 5, 3, 10, 15, 10, 3, 5, 1]$
    - $[15, 10, 3, 5, 1, 5, 3, 10, 15]$
    - $[1, 10, 1, 10, 1, 10, 5, 3, 1]$
    - $[1, 3, 5, 10, 1, 10, 5, 10, 1]$

This gives a total of 240 scenarios $((2 + 4 + 6) \times (4 \times 5))$. At each scenario the genetic algorithm takes 3 "shots" and the best fit is saved. At every shot the algorithm iterates 100 times. Every iteration takes at most 2 minutes. In the worst case scenario the whole test takes 24 hours $((2 \times 3 \times 240)/60)$. The whole script is called `genalg_validation.py` and the results go to the `genalg_results` folder.

4.6.2 Results

There is a graph associated with each scenario. Even for the appendix this is too big to be presented on paper. All the results can be found at both of the following links:

- `http://maxhalford.com/StSICMR/GA_trials/`
- `http://imgur.com/a/21vqQ`

As can be seen in the following images, the results can be impressive but also quite disastrous. They are a great source for understanding how the algorithms fails. For an objective assessment I referred to my supervisors. For the moment we haven't thought of a mathematical way of judging if a fit is good or not. Luckily this can be decided upon graphically as the graphs are in two dimensions. The team was very happy and the algorithm can be considered sound.

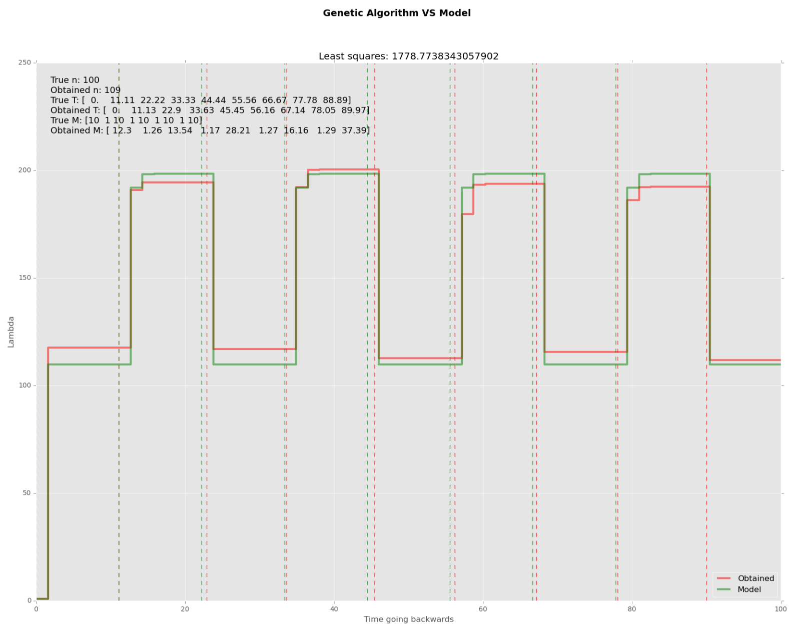


Figure 4.5: A good test

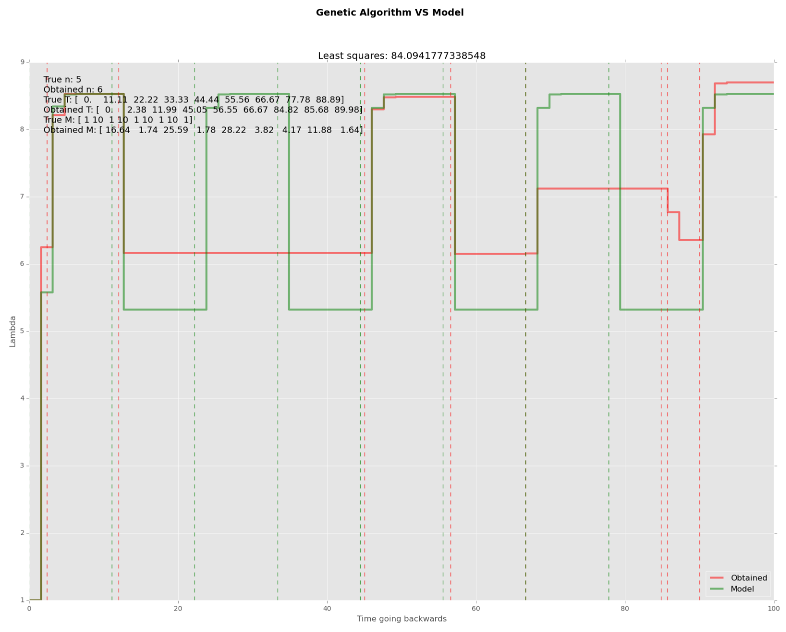


Figure 4.6: A bad test

Chapter 5

Applying the method to real data

5.1 Summary

The research team has built a model. The model is supposed to reproduce a genetic scenario given some parameters. There is enough evidence to suggest that some scenarios that were explained by population size changes can also be interpreted as changes in population structure. If this is the case many population genetics papers would be "wrong". To disprove these papers it has be shown that there exists a set of variables that can reproduce a genetic scenario with the model. To do this the genetic algorithm can be put to use. This provides an example of such a use.

5.2 Li & Durbin 2011

As mentioned in the introduction Heng Li and Richard Durbin produced a landmark giving the details of their PSMC algorithm in 2011. In it they showed how to explain a genetic scenario with changes in population sizes. However they did mention that it was possible that this wasn't necessarily the sole explanation.

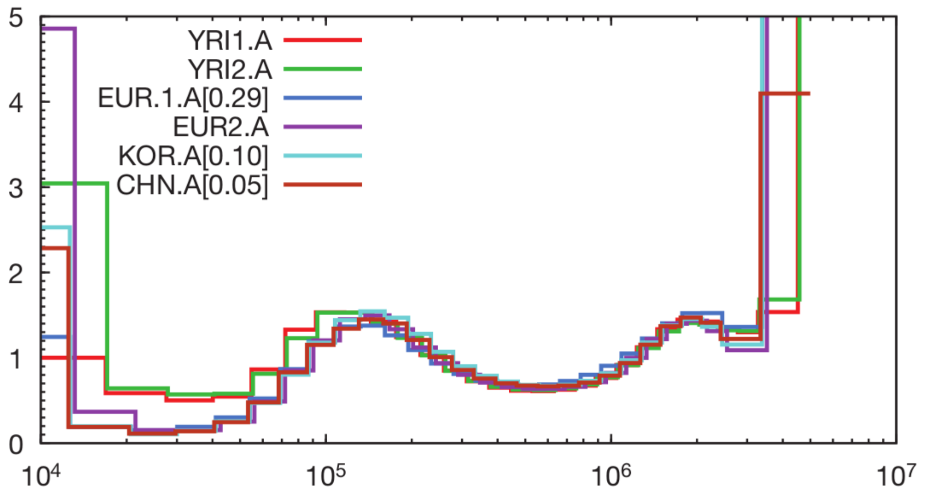


Figure 5.1: Li & Durbin PSMC

5.2.1 Fitting

In this case the `.psmc` files are not available, however this can be circumvented. In the appendix they gave the MS command to reproduce the graph they produced:

```
   ms 2 100 -t 65130.39 -r 10973.82 30000000 -eN 0.0055 0.0832
-eN 0.0089 0.0489  -eN 0.0130 0.0607 -eN 0.0177 0.1072 -eN 0.0233
0.2093 -eN 0.0299 0.3630  -eN 0.0375 0.5041 -eN 0.0465 0.5870 -eN
0.0571 0.6343 -eN 0.0695 0.6138  -eN 0.0840 0.5292 -eN 0.1010
0.4409 -eN 0.1210 0.3749 -eN 0.1444 0.3313  -eN 0.1718 0.3066 -eN
0.2040 0.2952 -eN 0.2418 0.2915 -eN 0.2860 0.2950  -eN 0.3379
0.3103 -eN 0.3988 0.3458 -eN 0.4701 0.4109 -eN 0.5538 0.5048  -eN
0.6520 0.5996 -eN 0.7671 0.6440 -eN 0.9020 0.6178 -eN 1.0603 0.5345
-eN 1.4635 1.7931
```

To be able to use the algorithm the times have to be scaled accordingly. In this case the individuals were considered diploid whereas the model is for haploid individuals. Coalescence is twice as fast for diploid individuals, thus the times that are given have to be multiplied by two to reach a haploid scenario (intuitively the times are "stretched") (Durrett 2008 [1]). The model also normalizes the lambda function so that it always starts at 1 (because $e^0 = 1$); the lambdas extracted from the PSMC thus have to be normalized, this is easily done by dividing them all by the first lambda value.

The script to apply the GA to Li & Durbin's data is the following. The parameter `logScale` can be set to `True` to plot the time values on a logarithmic scale due to the output of the PSMC.

Listing 4: Li & Durbin

```
import sys
sys.path.append('../../lib/')
from inference import genalg
import model
import plotting
import numpy as np

# We take the ms command from the
Supplementary Material multiplied by 2
liDurbin_tk = np.array([0.011, 0.0178, 0.026,
0.0354, 0.0466, 0.0598, 0.075, 0.093, 0.1142,
0.139, 0.168, 0.202, 0.242, 0.2888, 0.3436,
0.408, 0.4836, 0.572, 0.6758, 0.7976, 0.9402,
1.1076, 1.304, 1.5342, 1.804, 2.1206])

liDurbin_lk = np.array([0.0244, 0.0489,
0.0607, 0.1072, 0.2093, 0.363, 0.5041,
0.587, 0.6343, 0.6138, 0.5292, 0.4409,
0.3749, 0.3313, 0.3066, 0.2952, 0.2915,
0.295, 0.3103, 0.3458, 0.4109, 0.5048,
0.5996, 0.644, 0.6178, 0.5345])

l0 = 1 / liDurbin_lk[0]
liDurbin_tk *= l0
liDurbin_lk *= l0

pop = genalg.Population(model.StSICMR,
liDurbin_tk, liDurbin_lk, maxIslands=100,
switches=3, size=1000, repetitions=1)

pop.enhance(200)

plotting.plotModel(pop.best.model,
liDurbin_tk, liDurbin_lk, logScale=True)
```

By running the script the results are surprisingly good.

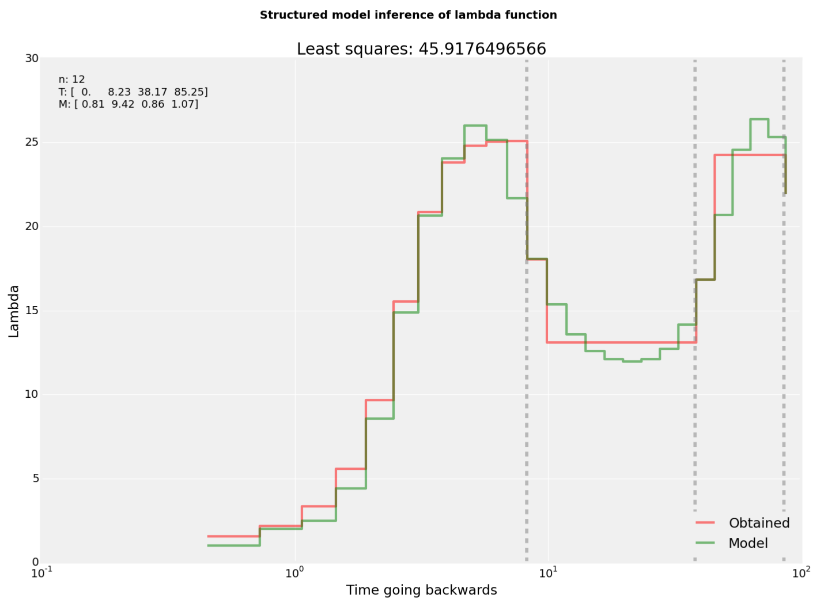


Figure 5.2: Fitting Li & Durbin data

5.2.2 Questioning the least squares method

First of all it has to be said that the algorithm did very well considering that the second bump isn't defined by many points. Indeed on some occasions the algorithm finds a good least squares that doesn't fit the second bump. Although it isn't a critical problem it casts a shadow of doubt over the least squares measure. The least squares method doesn't take into account the x axis. In PSMC files times are distributed exponentially, which means that there are less points in the past. In a sense the algorithm is assigning less and less value to areas in the past as there are less and less points to compute the least squares. Instead of computing the least squares over the vectors we could do it for the functions:

$$fitness = \sum_{i=1}^{n} (y_i^{obs}(x_{i+1} - xi) - y_i^{mod}(x_{i+1} - xi))^2 \tag{5.1}$$

This integration method works well because we are dealing with step functions, the difference between two of these being a rectangle. An artificial x_{n+1} has to be added to the tip of the x axis to be able to consider y_n.

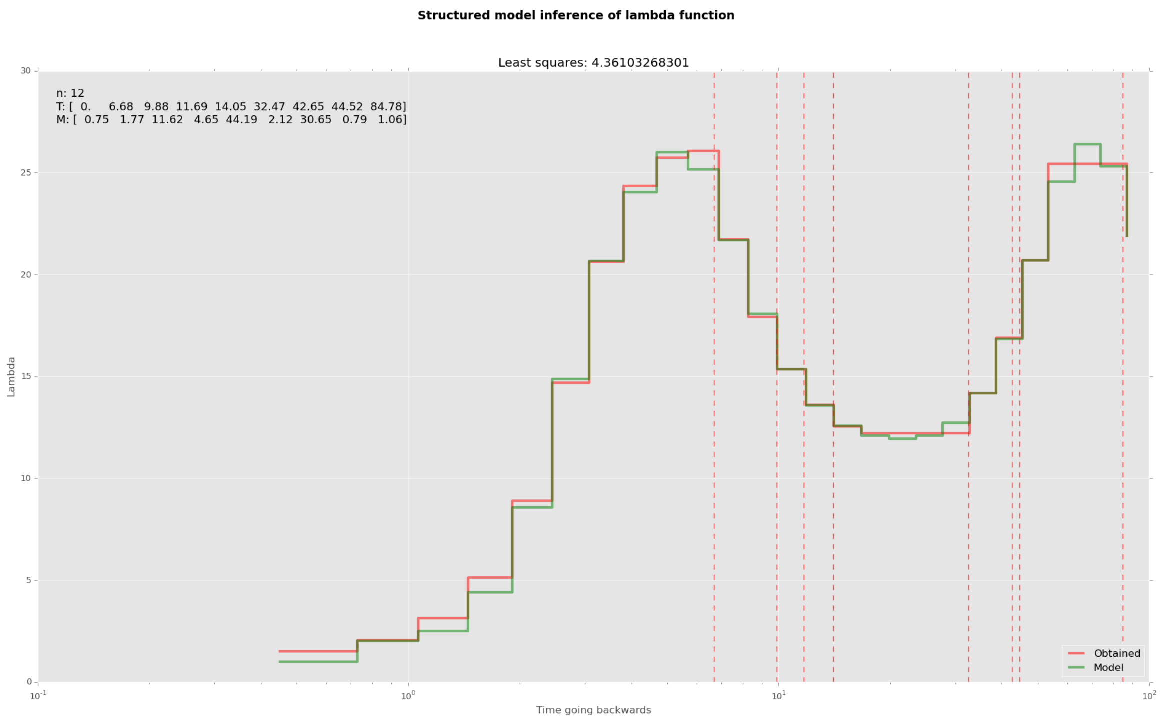
It turns out that this method assigns *too much* weight to further areas. Indeed the differences between further away times and current times are so large that shifting a migration rate will disrupt further fittings and local optimums will be attained too fast. To counteract this effect but still preserve the integration method we can do:

$$fitness = \sum_{i=1}^{n} (y_i^{obs}(\frac{x_{i+1} - x_i}{x_i}) - y_i^{mod}(\frac{x_{i+1} - x_i}{x_i}))^2 \tag{5.2}$$

$\frac{x_{i+1}-x_i}{x_i}$ is the *relative increase* of the x axis, it is the same order of magnitude at every time interval. Although it cannot be said that results have improved in quality, they are however much more stable. Sadly I haven't found any mathematical background for this method. In a sense all that matters is that the heuristic is sound and makes sense. Indeed the 5.2 equation really translates "*Give as much weight to current time areas as to further away areas.*" mathematically.

5.2.3 Studying n

For the "fun of it" I tried to go "overkill" and fit the data with a model allowing 8 changes in migration rate.



```
n: 12
T: [  0.     6.68   9.88  11.69  14.05  32.47  42.65  44.52  84.78]
M: [  0.75   1.77  11.62   4.65  44.19   2.12  30.65   0.79   1.06]
```

Figure 5.3: Overkill fitting

What the team and I found interesting is that the number of islands (ie. 12 in this case) seems to be stable when the number of migrations rates changes. It would be interesting to understand the influence of n over all the fits. To do this we can study the distribution of the fitness for different numbers of switches and different number of islands. To do so we can plot a list of boxplots, where every boxplot corresponds to a number of islands and regroups inferences. Then we can do this for different numbers of switches.

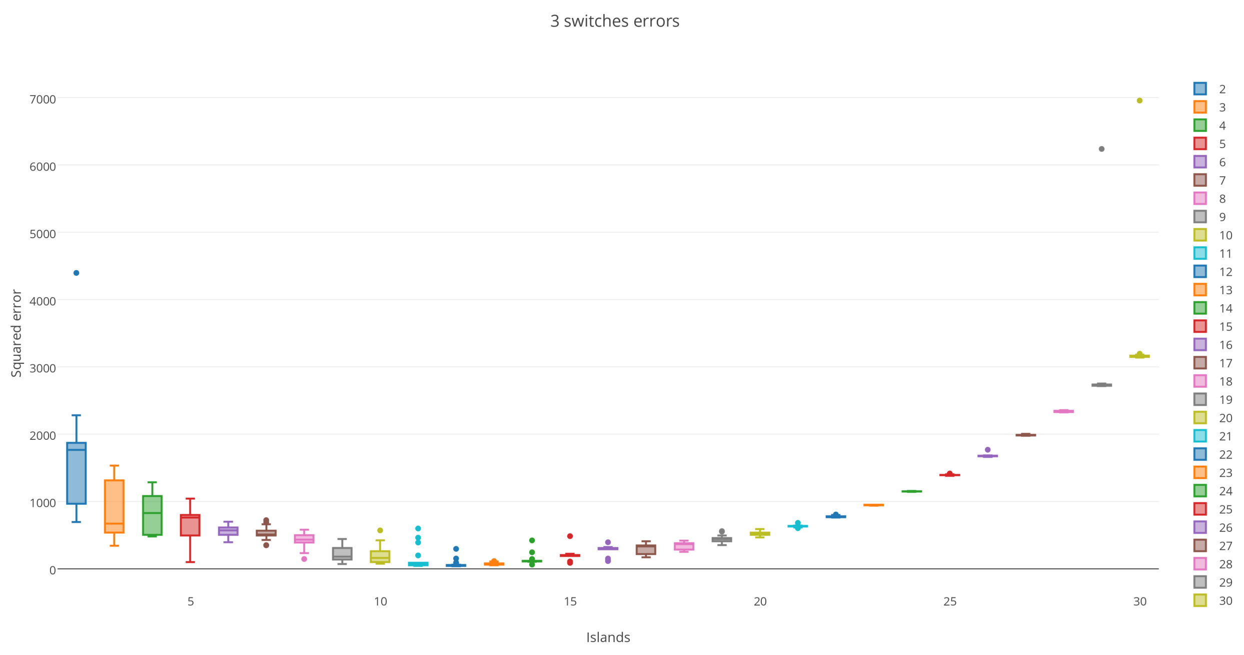


Figure 5.4: 3 switches error distribution

The previous is an example of such a graph for 3 switches (ie. 4 migration rates). Clearly a parabola seems to appear, reaching it's nadir for 12 islands. What is even more intriguing is that the same parabola appears for other number of switches! They are all available at `http://maxhalford.com/StSICMR/li_durbin/`. The ramifications of this observation are vast and more importantly are off topic. Indeed this would mean that although different numbers of migrations rates can enable to fit a genetic scenario the number of islands is always the same.

5.2.4 Conclusion

This study will enable the research team to propose a different interpretation of Li & Durbin's results; indeed it illustrates how structured populations can give fake size change signals. Moreover, it will help illustrate that the model can "mimic" population size models.

5.3 Sheep data

The previous data was human, the algorithm can also be applied to animal data. For example we can study sheep PSMC files that were collected by Willy Rodriguez (PhD. under Lounès Chikhi). The following graph is the best result out of 20 iterations, each one with 1000 generations. We can safely assume that each one has reached it's limit (hopefully a good local optimum) and that the following graph is the best the model can do.

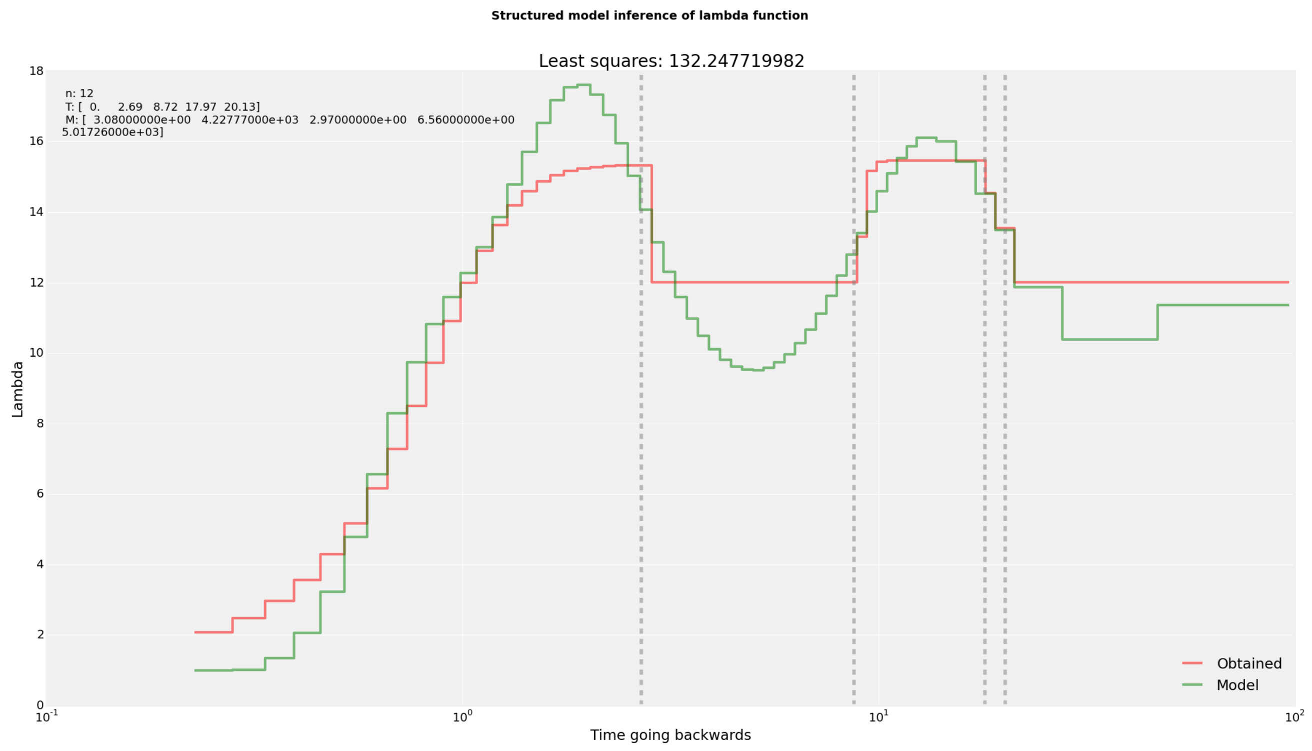


Figure 5.5: Fitting sheep data

Interestingly, the model doesn't seem to be able to fit the first bump very well. By looking at the other graphs, none of them did! Moreover, the migration rates for when the curve goes down are ludicrously high, suggesting that the model has reached it's limit. Intuitively, the PSMC (green) increases too quickly for the model to catch up. This would mean that certain genetic scenarios are not just due to changes in structure but are necessarily due to population size changes. Indeed a leap if population size would be the only explanation to such a sudden increase.

Chapter 6

Distributing the algorithm

Now that the algorithm is in place, the next step is to make it easily reusable. The ultimate goal is to produce a software which anyone can use without much help. This has a more software engineering side to it than previously. Concretely the sub-goals are:

- Flexible installation
- Easy to use
- Readable code

I decided to distribute the code on GitHub. At it's core GitHub is a website where you can store code projects into folders which are commonly called *repositories* (or *repos*). I chose it because first of all it's very popular and a lot of scientific code is stored there (for example Li's PSMC algorithm). Secondly it is possible to create private repositories, which is currently the case for the software because it contains unpublished mathematics. Finally it is *extremely* rewarding and easy to use once the basics are understood, especially after installing a GUI client.

GitHub repositories always contain a `READ.md` file. The `.md` extension designates a Markdown file, which uses a markup language to produce organized and readable documents. When navigating to a GitHub repository this is the first file that appears, thus I decided to add all of the documentation into the `READ.md` file.

6.1 Installation

The people who will be using the software will mostly be geneticists and mathematicians. It is a not a far fetched assumption that fiddling with the Python code will be enjoyable for them. In order to make the installation as smooth as possible I describe three installations methods in the texttttREAD.md.

6.1.1 Basic

The user can use his own Python installation. Once he has navigated to the directory containing the code he has to type `pip install -r requirements.txt` and all the necessary modules will be installed.

6.1.2 With a virtual environment

A virtual environment can be thought of as a sandbox which contains only interacts with it's components and not the users. The user only has to type some simple shell commands and then type `source venv/bin/activate` in order to get on track. This is recommended for most projects as the downloaded modules will only affect the folder and not the computer's Python version.

6.1.3 With Anaconda

Anaconda is basically a big sandbox which contains many useful Python modules out of the box. It can be downloaded here: http://continuum.io/downloads#py34. Once it is installed there is nothing else to do, the Python interpreter will use the one contained in Anaconda, however contrary to the virtual environment this is system wide.

6.2 Usage

I decided to make the software accessible through a command line. First of all most of the scientific community is accustomed to this method and so the usage should be easy to get used to. Secondly it's more maintainable than building a fully-fledged GUI. Finally users will be able to add on top of it by using homemade shell scripts. An example use case could be:

Listing 5: Command line example

```
python convert.py examples/example1.psmc
python infer.py examples/example1.csv -n 100 -s 0
-p 1000 -r 1 -g 25 -m least_squares -k True
```

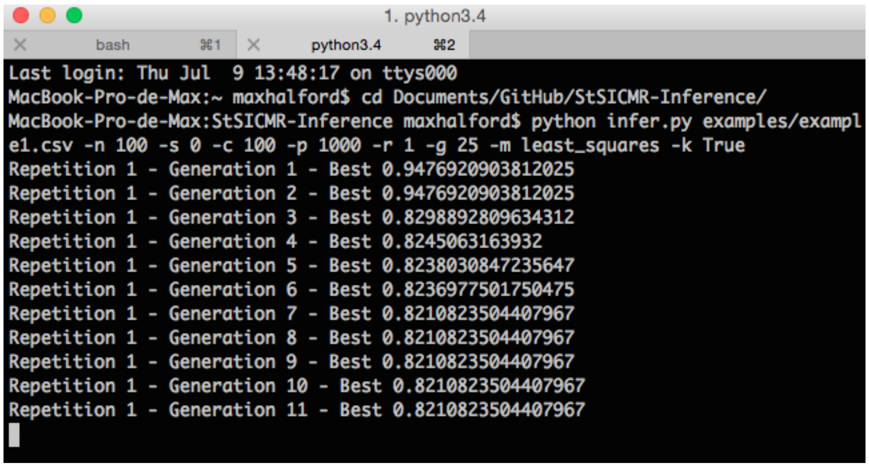
Where the parameters are the following.

Argument	**Parameter**	**Description**
-*s*	Switches	Number of switches for the model
-*r*	Repetitions	Number of times to repeat the process
-*g*	Generations	Number of iterations for each population
-*m*	Method	Method for evaluating the fits
-*k*	Keep	Set to True to save the inference as a plot and a JSON file
-*o*	Outfile	Override name of output files

At first the `convert.py` tool can be used to convert a `.psmc` file into a `.csv` file. Using CSV files as inputs to the algorithm makes it much more general and different scenarios are then easier to fake. People who have produced data sets with the PSMC algorithm would certainly agree.

Of course the number of switches (-*s*) has to be given, as does the number of iterations of the algorithm (-*g*). For convenience the repetitions argument (-*r*) is added so the algorithm repeats, enabling users to run the algorithms for a few hours and coming back to see the best result. Of course there are parameters for saving the results (-*k* and -*o*).

When run the algorithm will print in the console outputs it's current state, which is a nice feature for impatient or nervous users.

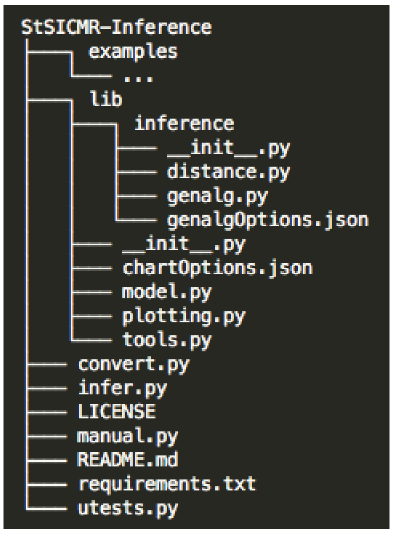


```
Last login: Thu Jul  9 13:48:17 on ttys000
MacBook-Pro-de-Max:~ maxhalford$ cd Documents/GitHub/StSICMR-Inference/
MacBook-Pro-de-Max:StSICMR-Inference maxhalford$ python infer.py examples/exampl
e1.csv -n 100 -s 0 -c 100 -p 1000 -r 1 -g 25 -m least_squares -k True
Repetition 1 - Generation 1 - Best 0.9476920903812025
Repetition 1 - Generation 2 - Best 0.9476920903812025
Repetition 1 - Generation 3 - Best 0.8298892809634312
Repetition 1 - Generation 4 - Best 0.8245063163932
Repetition 1 - Generation 5 - Best 0.8238030847235647
Repetition 1 - Generation 6 - Best 0.8236977501750475
Repetition 1 - Generation 7 - Best 0.8210823504407967
Repetition 1 - Generation 8 - Best 0.8210823504407967
Repetition 1 - Generation 9 - Best 0.8210823504407967
Repetition 1 - Generation 10 - Best 0.8210823504407967
Repetition 1 - Generation 11 - Best 0.8210823504407967
```

Figure 6.1: Command-line output

6.3 Code organization

Although the structure of the folder is not important for the user, I made an effort to organize it in the most modular fashion possible. Later on someone



```
StSICMR-Inference
├── examples
│   └── ...
├── lib
│   ├── inference
│   │   ├── __init__.py
│   │   ├── distance.py
│   │   ├── genalg.py
│   │   └── genalgOptions.json
│   ├── __init__.py
│   ├── chartOptions.json
│   ├── model.py
│   ├── plotting.py
│   └── tools.py
├── convert.py
├── infer.py
├── LICENSE
├── manual.py
├── README.md
├── requirements.txt
└── utests.py
```

Figure 6.2: Software architecture

might want to pick this project up, either for enhancing it or for picking some of the code. Readability and minimalism are of the essence.

All of the code, the most important part of the folder, is contained in the `lib/` folder. It all starts with the `model.py` script, which was initially coded by Willy Rodriguez and to which I added/removed pieces of code to make it very readable. The idea behind the architecture is to use *Object Oriented Programming* (OOP) to make things editable.

1. `model.py` spits out a `Model` object.
2. `inference/genalg.py` takes a `Model` object as a parameter, together with a CSV file and spits out another `Model` object (hopefully with better parameters!).
3. `plotting.py` takes a `Model` object and plots it, with or without a CSV file.

Although this doesn't seem like much, it's a good thing. Using OOP makes it possible to change a script without having to worry about the scripts further down or upper up the algorithm. In a sense this is a "pretty" organization if you are into code organization.

The charts are easily editable thanks to the `chartOptions.json` file, there is no need to fiddle with the code because it directly reads from the JSON file.

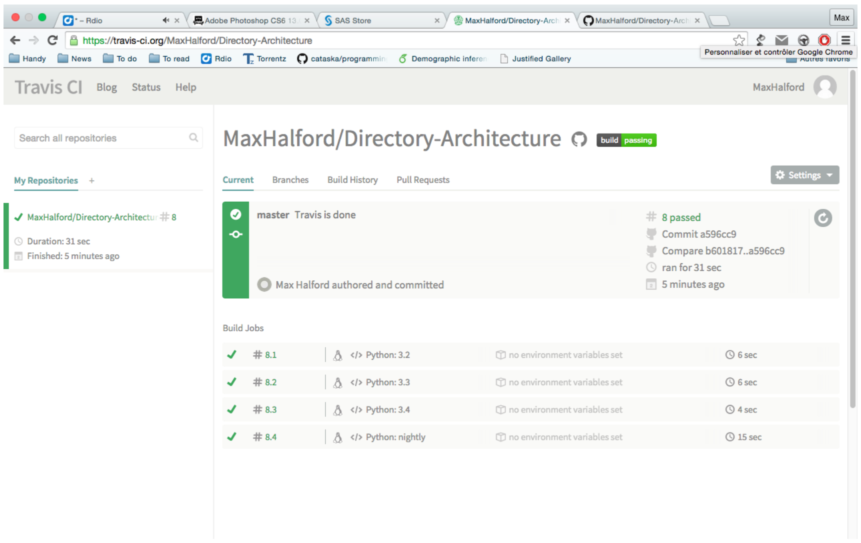
The same goes for the genetic algorithm. Indeed the `genalgOptions.json` contains settings for the mutation rates and the tournament selection method. These are settings that are important, putting them in a file makes it possible to memorize what they were on a good run of the algorithm. That is also why these parameters are not in the command-line parameters, along with the fact that the command-line instruction would become too verbose.

6.4 Code validation

Distributed code has to be reliable and has to work. In the case it doesn't or a bug is found users need to have access to a channel to file their issues.

6.4.1 Unit tests

First of all I decided to make a tests script (called `tests.py`) in the `lib/` folder to make sure everything is working as intended. In other words the `tests.py` script contains a bunch of *unit tests*. To go even further I decided to automatize them by using *Travis*, a web tool that can run the tests every time the code changes on GitHub. What's more Travis can be set up so as to run the tests on different Python versions, to ensure that a wide audience can use the code.



MaxHalford/Directory-Architecture

✓ MaxHalford/Directory-Architectur # 8
Duration: 31 sec
Finished: 5 minutes ago

master Travis is done
Max Halford authored and committed

8 passed
Commit a596cc9
Compare b601817..a596cc9
ran for 31 sec
5 minutes ago

Build Jobs

✓	# 8.1	Python: 3.2	no environment variables set	6 sec
✓	# 8.2	Python: 3.3	no environment variables set	6 sec
✓	# 8.3	Python: 3.4	no environment variables set	4 sec
✓	# 8.4	Python: nightly	no environment variables set	15 sec

Figure 6.3: Automatic unit tests with Travis

All the unit tests are in the `utests.py` script. Of course users are advised to run the script after installation to make sure it was successful.

Verifying the installation

This simply checks that the required modules are properly installed.

Verifying the model

This runs a dummy module with basic parameters just to make sure the mathematics are correct.

Verifying the plotting

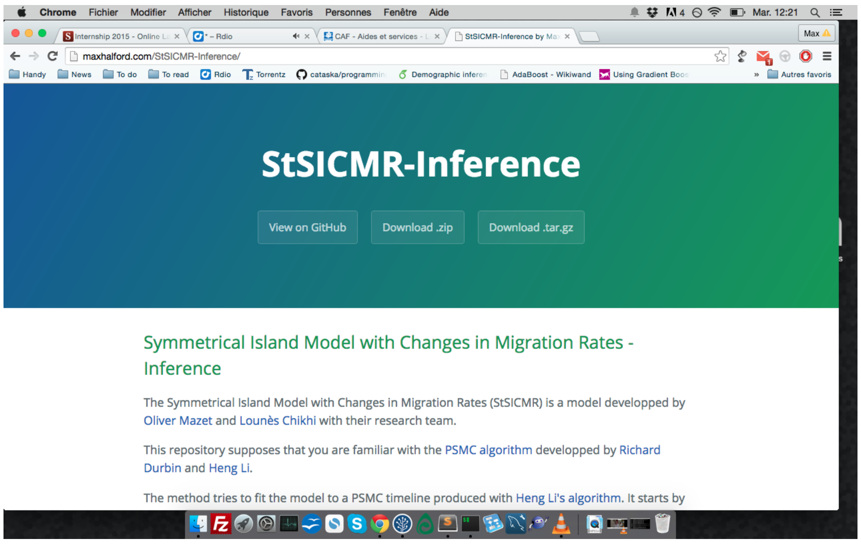
This takes the previously generated model and assures that a plot file can be generated from it.

Verifying the genetic algorithm

This makes sure the genetic algorithm runs smoothly in the console.

6.4.2 Webpage

Because the project is stored on GitHub, it is quite easy to generate a webpage based on the README.md file in the repository.


Chrome Fichier Modifier Afficher Historique Favoris Personnes Fenêtre Aide
Mar. 12:21
maxhalford.com/StSICMR-Inference/
View on GitHub
Download .zip
Download .tar.gz

StSICMR-Inference

Symmetrical Island Model with Changes in Migration Rates - Inference

The Symmetrical Island Model with Changes in Migration Rates (StSICMR) is a model developped by Oliver Mazet and Lounès Chikhi with their research team.

This repository supposes that you are familiar with the PSMC algorithm developped by Richard Durbin and Heng Li.

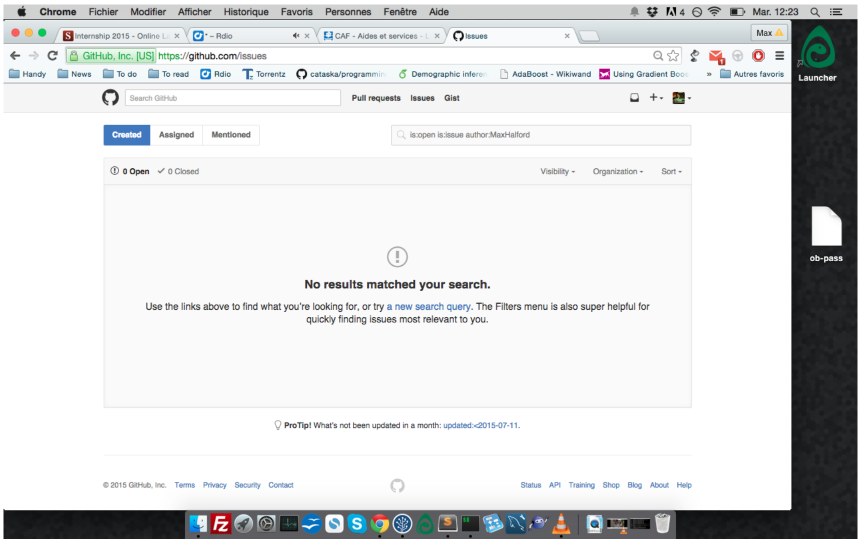
The method tries to fit the model to a PSMC timeline produced with Heng Li's algorithm. It starts by

Figure 6.4: Webpage

6.4.3 Feedback

First of all Willy Rodriguez's and my email addresses are indicated at the end of the GitHub page if users or researchers have questions.

Secondly GitHub enables users to be able to send *issues* to the the creator of a repository. This is a normalized way of suggesting improvements and reporting bugs. The creator gets sent a mail so that issues can be sorted as soon as possible.


Chrome Fichier Modifier Afficher Historique Favoris Personnes Fenêtre Aide
Mar. 12:23
https://github.com/issues
Search GitHub
Pull requests
Issues
Gist
Created
Assigned
Mentioned
is:open is:issue author:MaxHalford
0 Open
0 Closed
Visibility
Organization
Sort
Launcher
ob-pass

No results matched your search.

Use the links above to find what you're looking for, or try a new search query. The Filters menu is also super helpful for quickly finding issues most relevant to you.

ProTip! What's not been updated in a month: updated:<2015-07-11.

© 2015 GitHub, Inc. Terms Privacy Security Contact Status API Training Shop Blog About Help

Figure 6.5: GitHub issues

Chapter 7

Conclusion

To summarize, I discovered a subject that was entirely new to me. I familiarized myself with the underlying mathematics, which enabled me to understand the approach on how to formulate a model. I also enhanced my software engineering skills by producing an ergonomic application allowing to explore the model's potential. Moreover I discovered the world of research and that was priceless. I really enjoyed the work atmosphere and the way of "doing" things.

Being independent and open-minded are valued qualities, which I very much appreciated. I also had the opportunity to enhance my coding skills, more precisely I got better at structuring code and making it shareable/comprehensible. In general it was not so much the topic but rather the way of working and communicating with fellow workers that I found engaging and rewarding, I would without a doubt wish to work in such a way on other problems.

I sincerely enjoyed the relationships with the people around, never have I seen people so thrilled about mathematics. It definitely will make me consider pursuing a part of my career in academia. I have already decided that I want to pursue my work with the research team and keep adding some new features to software to make it even more supple and user-friendly.

I am also quite excited to see how many people will use this software and how I can improve on it. For the moment it is private but when it will be open sourced it will be easy to track the number of users.

Bibliography

[1] Rick Durrett. Probability models for dna sequence evolution. http://www.math.duke.edu/~rtd/Gbook/PM4DNA_0317.pdf, 2008.

[2] Hilde Maria Jozefa Dominiek Herbots. *Stochastic models in population genetics: genealogy and genetic differentiation in structured populations*. PhD thesis, 1994.

[3] Heng Li and Richard Durbin. Inference of human population history from individual whole-genome sequences. http://www.nature.com/nature/journal/v475/n7357/full/nature10231.html, 2010.

[4] Oliver Mazet, Willy Rodríguez, and Lounès Chikhi. Demographic inference using genetic data from a single individual: Separating population size variation from population structure. http://www.sciencedirect.com/science/article/pii/S0040580915000581, 2015.

[5] Sewall Wright. Evolution in Mendelian populations. *Genetics*, 16(2):97, 1931.