SAPIENZA
UNIVERSITÀ DI ROMA

The Battle of the Sexes

Simulation report

Programming - Unit 2
Prof. Pietro Cenciarelli
Team: Gennarelli Christian, Milone Giovanni, Obradovic Andela
Applied Computer Science and Artificial Intelligence

Introduction

The Battle of the Sexes is the focal theme of chapter 9 of the book entitled *The Selfish Gene* published in 1976 and written by biologist Richard Dawkins. The chapter in particular talks about a population model composed of 4 types of individuals: faithful men who are willing to a long courtship and to the raising of a child, philanderer men who do not have the patience to wait for a long period of courtship or for having to look after a child, coy females who need a long period of courtship before engaging in a relationship, and fast women who mate with any male who makes a proposal.

Dawkins argues that considered a population composed of these four types contextualised in an isolated system, with the progress of time (and consequently of generations), each of the subpopulations composed of the four types can reach a precise state of quantitative stability in relation to the totality of individuals.

The purpose of this report is to verify whether by building a program that respects the specifications of the model of the aforementioned book, the development of the population and the achievement of a possible stability of the system can occur in a simulated environment.

The rules.

The model proposed by Dawkins includes specifications for its operation. First, each pairing results in the earning of a payoff P for both individuals i_0, i_1 involved. The payoff is computed using 3 variables a, b and c. Respectively a is evolutionary gain, b is parenting cost and c is courtship cost. In particular, the laws representing P as a function of a specific pair can be formalised according to the following 2x2 matrix.

	F	P
C	(a - b/2 - c, a - b/2 - c)	(0, 0)
S	(a - b/2, a - b/2)	(a - b, a)

In particular, Dawkins argues that chosen $a = 15, b = 20$ and $c = 3$, the population p reaches a state of stability in which 5/8 of the men are faithful and 5/6 of the women are coy.

In order to build a proper simulation engine, it is important to establish precise rules of operation. In particular, chapter 9 already provides six rules:

1. The initial number of individuals must be the same for each subpopulation.
2. Philanderer individuals cannot mate with Coy individuals.
3. Encounters between subpopulations that can mate with individuals belonging to more than just one subpopulation must be random.
4. Children inherit the type of the same-sex parent.
5. Individuals cannot change their type.
6. It is the males who make mating proposals to the females, who consequently can decide whether to accept or reject.

However the model, as it is conceived, fails to provide fundamental information relating to the continuation of reproductive relationships over time and the longevity of the individual. In particular, the points that are not defined are:

- Relationship between payoffs and resources dedicated to future couplings.
- Maximum reproduction limit for an individual.

First of all, it is essential to underline that the chapter often refers to the concept of *time*, in particular when it comes to the time spent in courtship and raising the child. Since these parameters are considered within the calculation of the payoff, it can be deduced that the payoff represents, at least in part, a quantification of time resources dedicated to the couplings still available to the individual. Therefore, it is possible to assume that the amount of mating that an individual can make over the course of his life can be established on the basis of the sum of the payoffs of his previous relationships.

Hence, we define h_i as the happiness of an individual i, equal to the sum of all the payoffs of his previous relationships r, $(P(r_0) + P(r_1) + \ldots + P(r_{n-1}))$ and we define $R(h_i) \to \mathbb{N}$ as a class of functions that given h_i as input returns 1 if h is greater than some threshold $t \in \mathbb{N}$. To build the simulator, we will opt for the use of two interchangeable $R(h_i)$ functions, which we define as linearR(h_i) and exponentialR(h_i):

$$\text{linearR}(h_i) = \begin{cases} \text{if } h \geq ct \to 1 \\ \text{if } h < ct \to 0 \end{cases} \qquad \text{exponentialR}(h_i) = \begin{cases} \text{if } h \geq t^c \to 1 \\ \text{if } h < t^c \to 0 \end{cases}$$

where the value of the t threshold will be equal to 2 (the lowest positive payoff attributable to the 2x2 matrix). The $R(h_i)$ function will be called iteratively after each reproduction of i, to increment a resources$_i$ variable. Consequently we add the following rules:

7. If i has completed its nth pairing, h_i += $P(r_n)$.
8. Whenever a $R(h_i)$ function is called using a new h_i, initially $c = 1$.
9. After a $R(h_i)$ function is called for a specific i, c += 1.
10. If i has finished pairing and h_i has been updated, the resources$_i$ += $R(h_i)$ operation will have to be repeated until $R(h_i) = 0$.
11. Before i mates with another individual, the value of resources$_i$ is checked. If resources$_i \geq 1$, 1 point is removed and the individual can begin to mate. Otherwise i cannot reproduce, so it dies.

It will also be interesting to verify the alteration of the system due to the choice of one of the two functions $R(h_i)$ on the basis of similar settings.

How to compute stability?

Let's consider a subset of individuals $T \subseteq p$. Then, let's define state(T) as a function which, taken as input T, returns $\frac{|T|}{|p|}$. We can also define two populations p_a and p_b as close if state(T_{p_a}) − state(T_{p_b}) $\cong 0$ $\forall T_{p_a} \in \{T \mid T \subseteq p_a\}, T_{p_b} \in \{T \mid T \subseteq p_b\}$. An infinite sequence $< \text{p0,p1,...} > \mid \text{p}_{n+1} = \text{Rules}(p_n)$ is defined as evolution trace E. Therefore, once the right premises are made, we can provide a correct definition of stability.
We define p_n a stable generation if $p_n, p_m \in E$ are close $\forall m > n$.

The goal of the simulation is to monitor the stability of the system as it progresses over time. The statistical tool best suited to the task is variance. The variance Var(x) provides a measure of the quadratic variability with respect to the arithmetic mean of the values assumed by a function.

$$\text{Var}(x) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}$$

Therefore, chosen n states belonging to contiguous populations, the lower the variance between the states, the more stable the system will be. In order to be able to evaluate variations in stability in each phase of the simulation and to eventually be also able to establish if more than one state

of temporary stability should occur, the variance of the system will be computed by the simulator on an evolution trace of fixed length which will be constantly updated with new data.

The Simulator.

High-level structure.

The simulator was built with the aim of respecting the 11 previously established rules as carefully as possible. It was made through the Java programming language. The source code includes 4 classes (EvolutionStage, Generation, IndividualPool and UserInterface) plus an Individual package containing 7 further classes. The program foresees the creation of 4 queues of type IndividualPool for the respective containment of objects of type Coy, Fast, Faithful and Philanderer. Faithful and Philanderer are subclasses of the abstract Male class, while Coy and Fast are subclasses of the Abstract Female class. In turn Female and Male are subclasses of the abstract Individual class.
Once the data structures are filled with a number of objects consistent with the user's initial specifications, two ActivePool objects (EvolutionStage's generic static nested-class) are instantiated. These objects receive as parameter an IndividualPool parameterised on objects of type subclass of Male. The ActivePools work on separate threads, and exploit the IndividualPool taken as a parameter to extract objects of the Faithful or Philanderer type, which will carry out coupling proposals to objects of Coy or Fast type, taken from their respective pools (2), (3), and will generate new individuals in accordance with rule 4 of the model, drawing on the resources represented by a field reproductionResources. Once the coupling is complete, the parents perceive their payoff, and consequently reproductionResources is updated. Finally, the parents are evaluated by the static killOrPush method, which will reinsert individuals in their pools only if they still have reproduction resources. This process runs in an endless loop for each ActivePool object.
In the meanwhile, an instance of the UserInterface class running on a separate thread, takes care of obtaining data from the simulation such as numerical dimensions of the subpopulations, percentages and variance rates, and displaying them in real time. In addition, the UserInterface object keeps the generation count by increasing the nGenerations field at each iteration.
The simulation ends when the number of generations set by the user is reached.

Individual

Individuals in the Dawkins model are represented by the abstract class Individual. As such, this class and its two abstract subclasses Males and Females do not allow the creation of direct instances, but can be used as references for using generic objects and methods, and they also contain protected fields and methods that will be shared with all the subclasses. Each object of a subtype of individual has fields to keep track of happiness, reproductive resources, number of children produced and threshold t value for the assignment of reproductive resources. In addition, there are three static fields containing the values of a, b, c. In addition there are two further static fields, respectively a Boolean signal that can be set to activate a maximum reproduction limit for a single individual, and another which contains the value of the maximum reproduction limit.
Finally, there are methods for increasing the count of offspring produced, for the deduction of reproductive resources prior to mating (11), for the inspection of preventive resources for mating (11) and for the increase of the same after a reproduction (10). There is also an abstract method for calculating the payoff, which must be implemented in each non-abstract subclass of Individual. Whenever this method is called on an Individual, it takes as a parameter another Individual object, checks its type, and based on it increases the field happiness with a specific value computed using the parameters a, b, c and the formulas proposed by the 2x2 matrix (7). After that, the method ends up with a method call to possibly increase reproductive resources.

IndividualPool

The IndividualPool class is simple in its structure. It is designed to function as a parameterized queue for Individual objects. The IndividualPools must contain Individuals, they must have capabilities to withdraw them from the beginning of the queue, to insert from the end, to keep track of them and to be able to tell if the queue is empty. The field within which the Individual objects are stored is a LinkedList. Taking into consideration that two threads will operate concurrently on the same IndividualPool, in order to make the IndividualPool thread-safe all the methods of the class are synchronized. In this way, the thread that uses the IndividualPool object will be able to keep the lock until it has finished using it.

EvolutionStage

This in effect is the main class of the simulator. This class contains only static members. In total it consists of 4 IndividualPool fields (respectively for Coy, Fast, Faithful and Philanderer objects) plus a startingPoolSize field which contains the number of objects of which each pool must be filled with at the beginning.

As for methods, initPools is the first. When called, it initialises the pools by pushing in a number of Individuals equal to that of the startingPoolSize. The second is generateIndividual, and is a generic method that takes as parameters two objects mother and father which must be respectively of a subtype of Female and Male, and generates a random Boolean value. If the value is true, it instantiates a new object of the same type of mother and pushes it into the respective pool, if instead it is false, it generates a new object of the same type of father and pushes it into the respective pool. Then there is killOrPush, also this generic parameterised on subtypes of Individual, which calls the method for controlling the resources on the object taken as a parameter, and if the individual still has reproductive resources, the type is checked and pushed into his own pool. Then there is the drawMother method, which generates a random Boolean value. If this value is true and the coyPool is not empty, the method pops a Coy object from the coyPool and returns it. If, on the other hand, the value is false and the fastPool is not empty, the method pops a Fast object and returns it. Then we have getPopulationSize method, which returns an array of integers containing the current size of all pools and the throwDisease method, which when called empties each pool by 50%. This method is called having exceeded a certain threshold for the overall size of all pools to avoid slowing down the simulation and saturating the JVM heap space. As for the main method, when started, it calls a static method in UserInterface to allow the user to choose the initial simulation settings. Then it calls the initPools method and instantiates three objects (two of type ActivePool parametrised to types Faithful and Philanderer and one of type UserInterface) and calls the start method on all three. Finally, main waits for the UserInterface instance to finish and then ends the execution of the main thread.

ActivePool

ActivePool is a static generic nested class parametrised by subtypes of Male, and inherits directly from Thread in order to make each instance work concurrently. The constructor of this class can take an IndividualPool<Faithful> or IndividualPool<Philanderer> object as a parameter to use it as a base pool. The only method present in this class is the overridden run method which pops an object from the ActivePool and invokes the drawMother method of EvolutionStage, assigning the reference of the two returned objects to two new objects (parent and mother) and they are both used as parameters to call the generateIndividual method (of course if father is a philanderer, a fast mother will be chosen). Then methods are invoked on both objects to increase the number of children produced, to deduct reproductive resources and to increase the payoff. Finally, both parents are passed as parameters in the killOrPush method. Each of these operations is performed in an infinite loop until the thread is stopped by the interruption of the main thread.

Generation

The purpose of this class is to pack into a single object the distribution percentages of each of the subpopulations together with the number of the generation in which the values were sampled. The structure of this class is also relatively simple: there is only one static method that takes as parameters two integers (subPopulation and totIndividuals) and returns the distribution percentage of subPopulation. This method is called by the constructor (which receives as a parameter the current size of all pools, the total number of individuals and the number of the sampling generation) for each of the pool sizes taken as a parameter, to then assign the returned values to the fields of the Generation object, which can be accessed through the appropriate get methods.

UserInterface

The task of this class is to collect simulation data and display it to the user in real time. This class extends the Thread class, so that an instance of it can run parallel to the ActivePools. It contains static fields to store the value of the dimensions (both purely numerical and percentages in relation to the totality of individuals) of the pools, to store the value of the variance of the pools in the evolutional trace, to define the length of the evolutional trace itself, to keep track of the number of total simulation samples, for the refresh interval and for the threshold of total individuals to invoke the EvolutionStage throwDisease method. The evolutional trace is also stored into a static field of the class, represented by an ArrayDeque parametrised for Generation objects. The run method of this class is composed of a loop that is executed as many times as there are samples to be performed, which initially invokes another static method of the class that receives the size of the pools from EvolutionStage and computes the percentage frequency by invoking the special static method of the Generation class, assigning all the aforementioned data to the fields of the class. Part of this data is used to generate a new Generation type object in a separate method, which will serve to update the evolutional trace. Each time a newer element is pushed into the evolutional trace, the oldest is discarded. Then the variance is computed for each object contained in it, in particular for each of the subpopulations, and these data are also stored in the appropriate fields. Then a static method is invoked which will show in the console all the data obtained, also represented with histograms.

Before reaching the next iteration, run will check if the total number of individuals is lower than the one predetermined by the maximum threshold set by the user, if this is not the case, then it will

```
Generation: 15
Tot individuals: 3905367 | Females percentage: 30,90 | Males percentage: 69,10)
Variance: Green = Stable | Red = Unstable

Coys: 679679 | Percentage: 17,40
Variance over 10 generations: 1,87
[#######                                ]

Fasts: 526986 | Percentage: 13,49
Variance over 10 generations: 5,68
[######                                 ]

Faithfuls: 1736161 | Percentage: 44,46
Variance over 10 generations: 5,68
[##################                     ]

Philanderers: 962541 | Percentage: 24,65
Variance over 10 generations: 0,20
[##########                             ]
```

invoke the throwDisease method. Also note the presence of two static methods: one that allows the user to enter initial settings (number of starting individuals for each type, the values of parameters a, b, and c, the maximum threshold of total individuals to throw a disease , the possibility of giving a maximum number of reproductions to each individual, the total number of sampling iterations and the length of the evolutional trace), and the other that sets the simulation according to the specifications received from the first method.

How to collect data?

According to the operating specifications of the code provided so far, the data obtained from the simulation can only be shown in real time, but they are not permanently collected, and this represents a problem when studying the results produced by the simulation. To overcome this problem, the Apache POI APIs have been imported which are used directly in the UserInterface run method to compile spreadsheets with data that is updated at each iteration.

Results

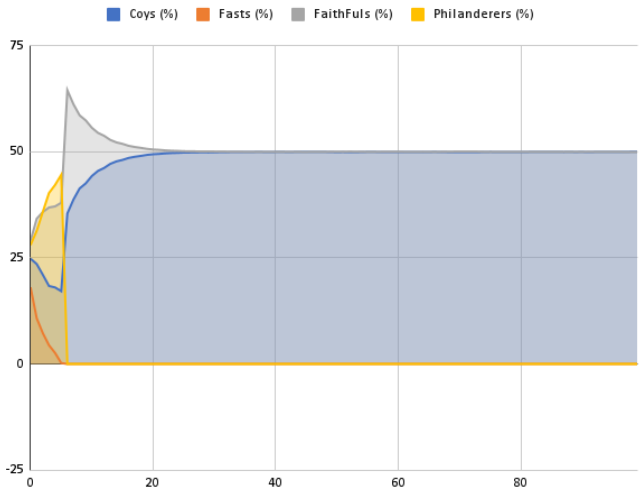
First of all, unless otherwise specified, all tests were performed with the following settings: 500,000 initial individuals for each pool, 5,000,000 maximum individuals before throwing disease, no limit to reproductions, 10-element evolutional trace, 1000ms of di sampling interval and 1000 total sampling generations. All tests were performed twice, using first the exponentialR(h_i) function and then the linearR(h_i) function. Respectively the tests are the following:

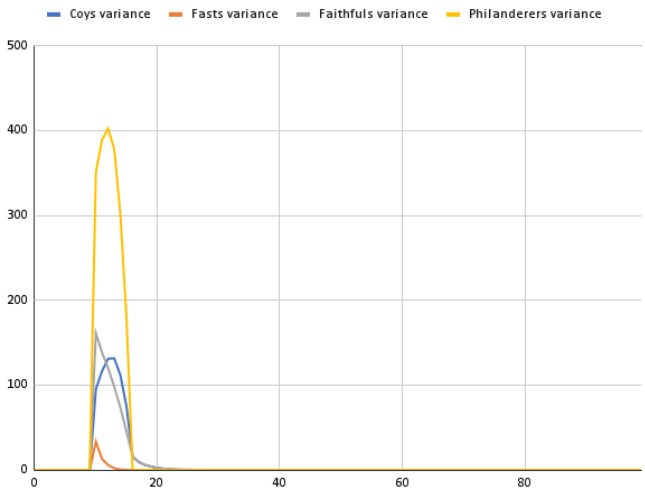
- $a = 15, b = 20, c = 3$ (default)
- $a = 20, b = 20, c = 3$
- $a = 20, b = 15, c = 3$
- $a = 15, b = 20, c = 5$

exponentialR(h_i)

- $a = 15, b = 20, c = 3$

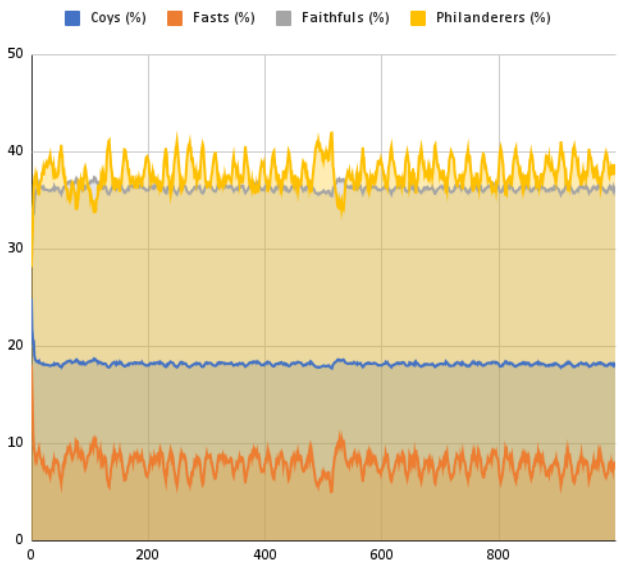
This test was the only one to have been performed with different settings from the others, i.e. 7,500,000 as a disease threshold, 5 children that can be produced at most and 100 sampling generations. Notice that starting from the eighth generation there is a total extinction of the Fast type individuals and consequently of the Philanderer individuals, with a consequent surge in variance in correspondence (second graph). After the disappearance of the Fast and the Philanderer both populations of Coys and Faithfuls stabilize on an average of 50% both with a variance rate of 0.023.

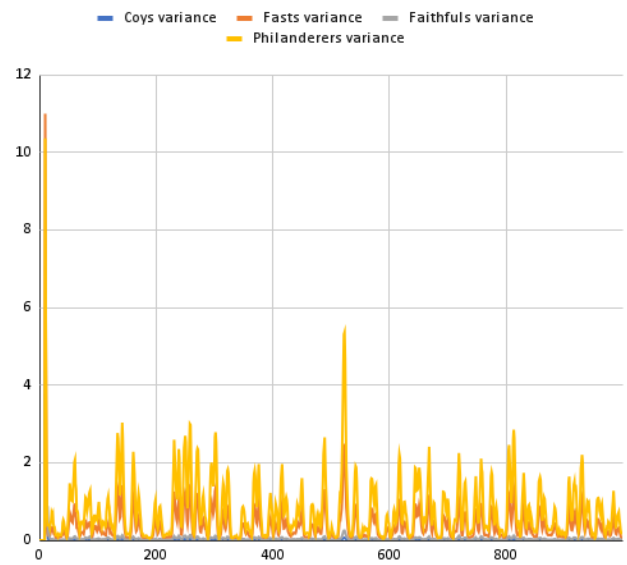




- $a = 20, b = 20, c = 3$

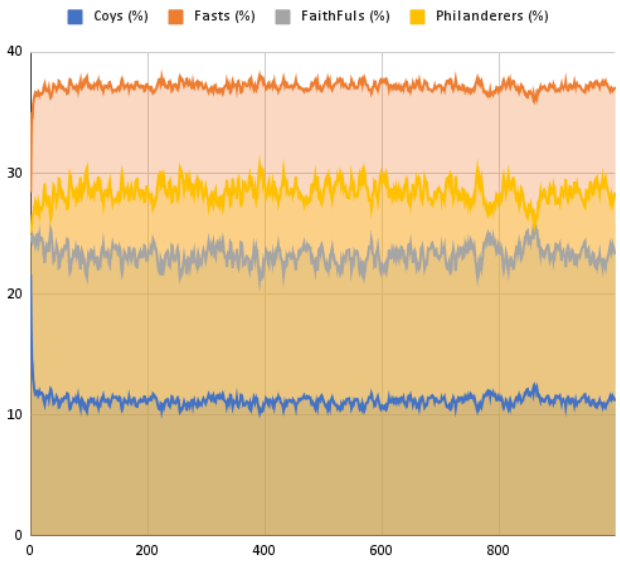
The purpose of this test is to verify whether by assigning an evolutionary gain value equal to the parenting cost, the populations that tend to receive a lower payoff are able to survive and stabilise. Indeed it turns out that Philanderers and Fasts this time manage to survive. However, both populations have continuous fluctuations that do not allow the achievement of total stability. In particular, the Fast fluctuate between 5-10% of the total population with an average variance rate of 0.37, while the Philanderers are in a range of 35-45% of the total population with an average variance rate of 0.76. Also this time Coy with a mean percentage of 18.15% and a mean variance rate of 0.016 and Faithfuls with a mean percentage of 36.24% and a mean variance rate of 0.043 turn out to be the most stable populations. It should also be noted that on average 73.93% of the population is made up of males.

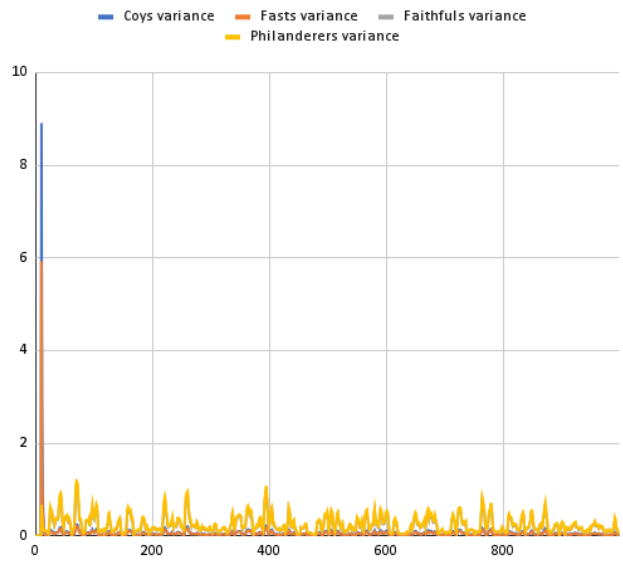




- $a = 20, b = 15, c = 3$

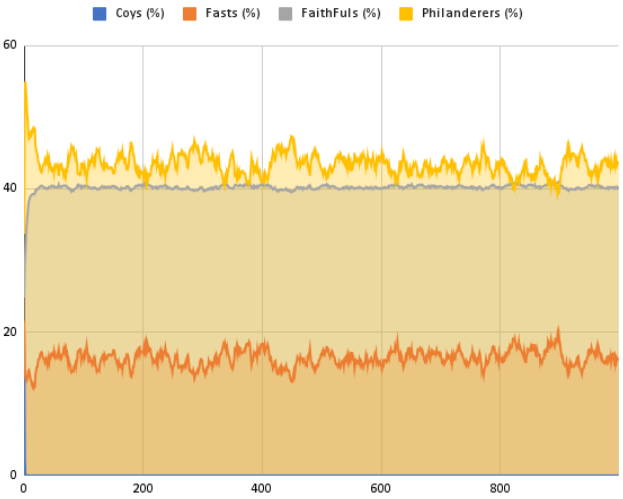
In this test the value of evolutionary gain and parenting cost were exchanged in order to verify the consequences on the system. We can observe how in this case the best performing population is the Fast one (the most disadvantaged in the previous tests) with an average distribution percentage of 37.16% and an average variance rate of 0.051. Philanderers follow with a mean distribution of 28.43% and a mean variance rate of 0.254. Then there are the Faithfuls who make up an average of 23.23% of the population with an average variance of 0.23. The least successful population is the Coy, spread on average at 11.18% and with an average variance of 0.066.

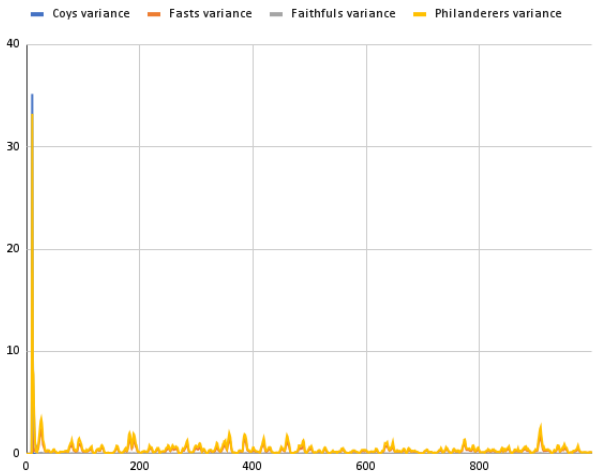




• $a = 15, b = 20, c = 5$

The goal of this test is to verify the consequences of the increased courtship cost. The most widespread population is that of the Philanderer (43.45%) and at the same time it is also the most unstable (0.492 of variance). The most stable population, on the other hand, is the Faithful (distribution of 40.17% and 0.057 variance). The penultimate by extension is the Fast spread at 16.36% with 0.313 of variance. The population that suffers most from the increase in the cost of courtship is that of the Coy, which is completely extinct after 2 generations.

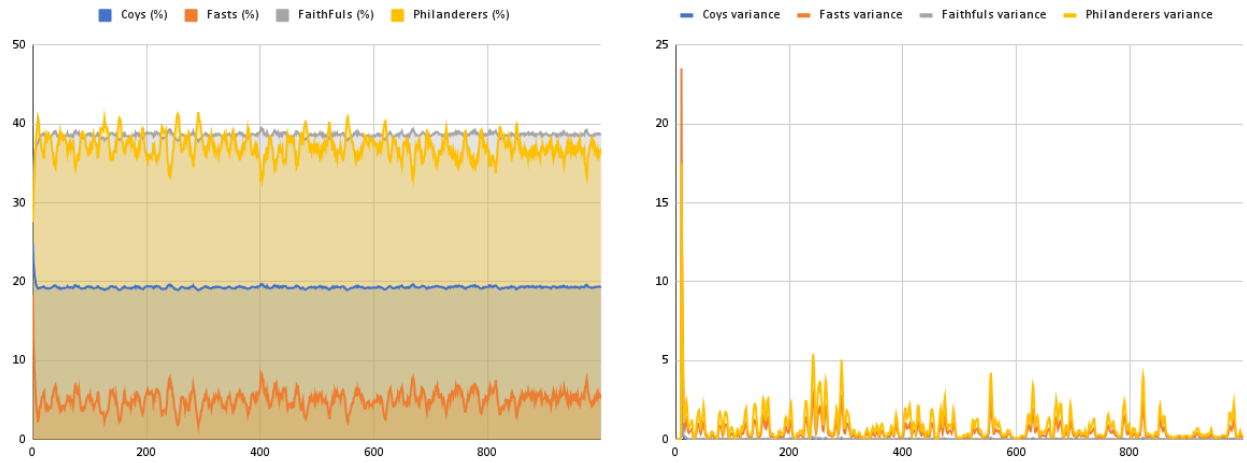




linearR(h_i)

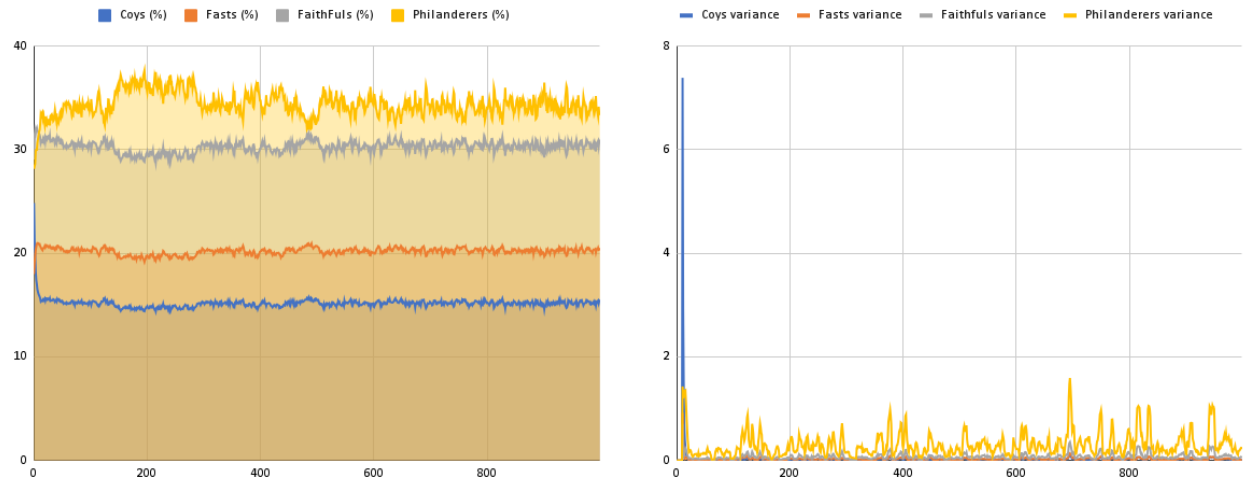
- $a = 15, b = 20, c = 3$

Also in this situation the most stable populations are the Faithfuls (38.56% spread and 0.038 mean variance) and Coy (19.31% spread and 0.012 mean variance). Unlike the previous test, here both Philanderer and Fasts manage to survive (with 37.06% diffusion and 0.876 mean variance for Philanderers and 5.07% diffusion and 0.499 mean variance for Fast, respectively)



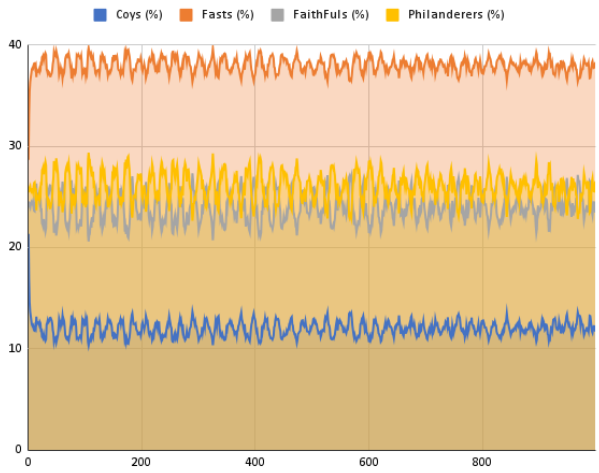
- $a = 20, b = 20, c = 3$

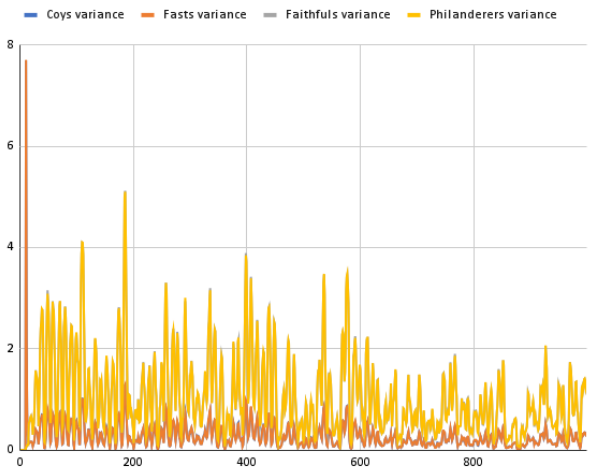
The populations from the most to the least performing are the following: Philanderers, who are also the most unstable, with a range of diffusion that varies from 32.0% to 37.6% and 0.298 of mean variance, the Faithfuls diffused at 30.27% and with mean variance of 0.072, Fast spreads at 20.18% and with mean variance of 0.026 and Coy spreads at 15.16% with mean variance of 0.031.



• $a = 20, b = 15, c = 3$

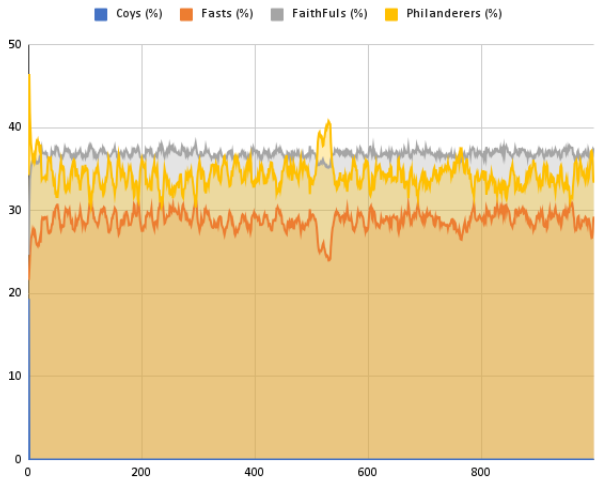
It can be seen that the system presents a trend similar to that seen in the exponentialR(h_i) one, although it is less stable in general. The most widespread population is the Fast (at 37.99% with 0.275 of mean variance), followed by Philanderers (at 26% with 1.05 of mean variance), then by Faithfuls (at 23.98% with 1.053 of mean variance, which is the highest variance recorded so far) and finally by the Coys (at 12% with 0.275 mean variance).

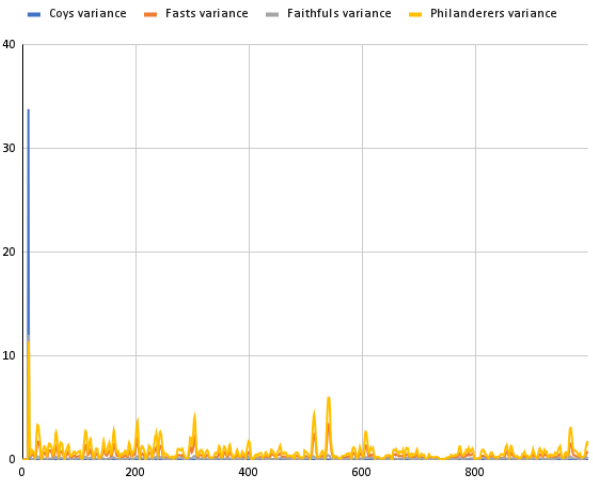




• $a = 15, b = 20, c = 5$

Even using linearR(h_i), the Coy population becomes extinct in a single generation. In this case, the most widespread and at the same time most stable population is Faithful (36.83% of average diffusion and 0.102 of mean variance), followed by Philanderers (34.30% of mean diffusion and 0.787 of mean variance) and subsequently by Fast (28.84% average diffusion and 0.385 of mean variance).





Conclusions

Considering the simulations discussed above, we can actually see how the system in each of the cases represented is able to achieve almost immediate stability with an average variance rate ranging from 0 to 1, which demonstrates how the fluctuations of the populations in the system are never drastic. According to the tests, the least stable population turns out to be that of the Philanderers. This may be justified by the fact that Philanderers are uniquely reproductively dependent on Fast individuals. It should be noted in fact how often in the graphs a decrease in the Philanderer type population corresponds to an increase in the Fast population and vice versa.
The populations that generally perform better in the simulation are Faithfuls and Philanderers, indeed, it appears that males tend to be happier, and consequently able to reproduce longer. Contrariwise the generally more disadvantaged population is Coy, which in the event of an increase in the courtship cost tends to extinction, as at the same time a Faithful-Fast mating would be more convenient for both peers involved.
The only case in which Coy is not the most disadvantaged population is that of using the standard values proposed by Dawkins. In particular, this type of test allows us to observe how the choice of a function R(h_i) drastically influences the performance of the system. In fact, the choice of a linear threshold for the assignment of reproductive resources allows a naturally greater quantity of reproductions compared to an exponentially increasing threshold, which instead, as it has been noted, would have caused the total extinction of the Fast and Philanderer individuals.
However, in none of the cases presented, the populations reach a similar distribution rate to that proposed by Dawkins.