Evolutionary Behavioural Ecology: practical 1 - Modelling Evolution

Bram Kuijper

2021-10-01

Contents

Chapter 1

Practicals on evolutionary models using R

During the following two practicals, we will try to implement some basic evolutionary models in R. The first practical will focus on the evolution of aggressive vs evasive behaviour. The second practical will focus on the evolution of cooperation and conflict. Along the way you get exposed to working with R, which is never a bad thing as you are likely to use this program a lot of times during this year.

1.1 Learning objectives

After this practical, you…

- should be able to use R to implement short programs
- understand how we can develop simple population genetics models in R
- should be able to explain why models necessarily include many limiting assumptions
- should be able to inspect your model and draw conclusions from it

1.2 Opening the R environment

To provide you with a standardized environment during this practical session, you will have to log into to the University of Exeter's Rstudio server. Of course, you are welcome to try to do the same using your installation of R on your own computer, but we might run into problems regarding the installation of some packages, hence we ask you to stick to the rstudio server during the practical.

Chapter 2

A brief R programming crash course

To get started with making our own evolutionary model, we need to make a simple program in R that tracks an evolving population.

To this end, you need to know something about the ingredients that simple programs use, namely variables, vectors/sequences and for-loops. Some of this may have already featured in your statistics or introduction to data science modules, but at this stage, getting some repetition about the sometimes bamboozling inner workings of R certainly does not hurt. Even the most field-oriented among you will use R extensively during your research projects, so the more exposure you can get to this environment, the better.

Try to type along with the examples below and see whether they work. Give me a shout if something looks odd.

2.1 Variables

A variable provides us with a named storage of data, allowing you to retrieve that bit of data when you want it. A variable in R can contain any type of data (e.g., a single number, a string of text, a vector of multiple values, etc).

We assign data to a variable using a `=` or `<-` operator (both can be used interchangeably) For example:

```
# create a variable named var.1 and assign a value to it using the <- operator
var.1 <- 25000

# create a variable named var.2 and assign a vector, c(), of numbers to it using the = operator
var.2 = c(0,1,9.5,200)
```

```
# a bit of text, surrounded by single '' or double "" quotes
some.text <- "Swanpool beach"
```

2.1.1 Naming variables

Variables can have any name, but cannot start with a number and can only contain dot `.` and underscore `_` characters as special characters.

```
# this fails
2times_song_release_year <- 1999
## Error: <text>:2:2: unexpected symbol
## 1: # this fails
## 2: 2times_song_release_year
##     ^
```

```
# this is fine
release_year_2times <- 1999
```

```
# this fails too
release%year <- 1999
## Error: <text>:2:8: unexpected input
## 1: # this fails too
## 2: release%year <- 1999
##            ^
```

2.1.2 Changing the value stored by variables

You can easily change values of variables, simply by assigning them a new one! The new value can be of a completely different type.

```
# print the value of a variable, showing its current value
print(var.1)
## [1] 25000

# change the value
var.1 <- "some text instead!"

# print the new value
print(var.1)
## [1] "some text instead!"
```

You can always check the type of your variable using the `typeof()` function:

```
typeof(var.1)
## [1] "character"
typeof(var.2)
## [1] "double"
```

As you see, `typeof` does not tell you `var.2` is a vector of multiple values, but just focuses on the type of the values contained in them (in this case a `double`, which is another word for floating point value). If you want to find out if something is a vector of values, you have to look at the length of the variable (see below).

2.2 Vectors and element selection

As you have seen in the example of `var.2` above, variables can point to collections of multiple values (vectors). To create a vector, we use the `c()` function. We can subsequently get values of (groups of) individual elements by using the `[[]]` operator:

```
# 1. create a vector of character elements using c()
animals <- c("Opossum","Dog","Pallas's leaf warbler")

# 2. select a single element using the double square brackets [[]]
print(animals[[3]])
## [1] "Pallas's leaf warbler"


# 3. obtain a selection of
# multiple values using [] and
# a vector of elements you want to select
print(animals[c(2,3)])
## [1] "Dog"                   "Pallas's leaf warbler"


# 4. assign a new value to an existing element
animals[[1]] <- "Click beetle"
print(animals)
## [1] "Click beetle"          "Dog"                   "Pallas's leaf warbler"
```

2.2.1 Exercises

- What happens to the vector `animals` if you assign the name of your favorite animal (in quotes `""`, because text) to element `40`?

- Assign the number 0.5 to element `10` of the `animals` vector. Try to multiply this value by 10, by typing

```
animals[[10]]*10
```

what happens? Use `typeof(animals[[10]])` to see what is going on.

2.3 Information about your vector

You can obtain information about your vector by using a range of different functions, such as

- `sum()`: sums all values (only works with numbers obvz)
- `unique()` lists all unique values
- `sort()` sorts values. and there are lots more.

Most often, however, you will use the `length()` function to obtain its size:

```
length(x=animals)
## [1] 40
```

2.3.1 Exercise

Using the standard `iris` dataset (available within in `R` by typing `iris` on the command line), find out the number of unique values for the column `Sepal.Width`. If you see `iris` but don't know how you can access the column, try to search for it on the web.

2.4 Sequences

Vectors are also used to contain sequences of numbers, which can either be generated with the from-to operator `:` or with the more explicit `seq()` function:

```
# sequence from 1 to 10
some.seq <- 1:10

# sequence from 10 to 1
reverse.seq <- 10:1

# same, but using the sequence in some.seq and the rev() 'reverse' function:
reverse.seq.2 <- rev(some.seq)

# sequence from 10 to 1, using the seq() function, with a negative increment
another.seq <- seq(from=10,to=1,by=-1)

# the seq() function is great if you want to take bigger steps:
steps.of.5 <- seq(from=3,to=29,by=5)
```

We can also make vectors using repetitions of values, through `rep()`:

```
# a vector of 300 zeros
only.zeros <- rep(x=0,times=300)
```

```
# a vector of 0,1 values, repeated until a length of 13 is achieved
zero.one <- rep(x=c(0,1),length.out=13)
```

2.5 The important bit: `for` loops

The point about programming is that you want to repeat actions multiple times. To this end, R offers you two different looping constructs: the `while()` loop and the `for` loop.

In this practical, we focus on the `for` loop as it is more commonly used than `while()`. Best to begin by an example:

```
# you need some vector of something
# otherwise there will
# be nothing to loop over
# Here, I use a vector of numbers from 5 up to (& including) 10
some.vec <- 5:10

# component 1: the for statement within
# parentheses ()
for (iterator in some.vec)
{
    # component 2: the for body within curly braces {}
    # in which tasks
    # are performed
    # repeatedly for each value of iterator
    print(iterator)

    # I am also printing something else
    # for the sake of illustration
    print("Some text.")
}
## [1] 5
## [1] "Some text."
## [1] 6
## [1] "Some text."
## [1] 7
## [1] "Some text."
## [1] 8
## [1] "Some text."
## [1] 9
## [1] "Some text."
## [1] 10
## [1] "Some text."
```

From the printed output, you see that the `iterator` gets a new value every time

the loop goes round. The message `"Some text"` is also repeatedly printed, but this value does not change.

2.5.1 Dissecting the for loop

The loop starts with the keyword `for`, followed by parentheses `()`. Within those parentheses, we always start by declaring a new variable, which – in this case – I have called `iterator`, but you could have given it *any* valid name, like `index` or `boring.practical`.

Each time the `for` loop goes round, the `iterator` variable gets assigned a new value. Which value? An element from a vector, which we have called here `some.vec`. The `in` keyword in between `iterator` and `some.vec` performs the assigning of these consecutive elements from `some.vec` to `iterator`.

Hence, the first time the loop goes round, `iterator` receives the value of `5`, as this is the first element of `some.vec`. Subsequently, the `for` loop enters the bit in between curly braces `{}`. The code in this part will be executed every time `iterator` gets a new value. In this case, the code is `print(iterator)`, meaning it prints the *current* value of the `iterator` variable (which is `5`) is printed to the screen. After this the subsequent `print()` statement is executed, which prints `"Some text"` to the screen.

Next, the loop goes round again. Now the value of `iterator` will be 6. Again, `6` is printed by the `print()` statements in the `{}`-part and this goes on until all elements from `some.vec` have been assigned to `iterator`. Hence, the last value printed will be `10`, after which the `for` loop exits.

2.5.2 Another example

Just to throw in another example that is slightly different, let us calculate the product of elements of two vectors and sum those products:

```
# generate two vectors with numbers;
# vectors have the same length
vec.1 <- c(0.3,0.9,1.2,0.1,3.5)
vec.2 <- c(5,8,12,3,7)

# check whether the length is indeed the same
stopifnot(length(vec.1) == length(vec.2))

sum <- 0

# loop not over the elements of the list
# but over the element indices
# (we can do so by creating a list from
# 1 to the length of one of the vectors,
# i.e., 1,2,3,...)
```

```
for (idx in 1:length(vec.1))
{
    # use idx to obtain elements of both vectors
    # and add them to the total
    sum <- sum +
        vec.1[[idx]] * vec.2[[idx]]
}
print(sum)
## [1] 47.9
```

2.5.3 Exercises

- Why we don't we write `for (idx in vec.1)`, as in the previous example?
- Could you change the code to store the sum that you obtain in each iteration in a list? I.e., that list should have the values

```
print(cumul.list)
```

```
## [1]  1.5  8.7 23.1 23.4 47.9
```

Now that we have some tools under our belts, let's get going with evolution!!

Chapter 3

An evolutionary model of the hawk-dove game

Why do we often find that competitors exhibit ritualized displays (e.g., threats such growling, baring one's teeth) to decide fights with competitors, rather than engaging in escalated fights in order to kill the opponent?

This question has been addressed by a simple evolutionary model, called the Hawk-Dove game. This influential model aims to sketch out how aggressive versus more restrained behaviours evolve when individuals compete with each other over resources.

Here we analyze a deterministic version of this model, by focusing on a population consisting of i) aggressive hawks (denoted by H) who always fight and do not retreat; and ii) timid Doves (denoted by D), who may perform some threat display, but eventually will always retreat if the other individual starts a fight.

3.1 Payoff matrices

We can summarize these interactions in a so-called payoff matrix, where the rows reflect the payoff to a focal player who either plays strategy H or D, when it encounters other Hs or Ds (columns).

focal ↓ opponent →	H	D
H	$\frac{v-c}{2}$	v
D	0	$\frac{v}{2}$

3.2 Exercise

Can you explain in your words what each of the payoffs mean? Try to search for descriptions of the Hawk Dove game online if unclear. We will discuss the answer in class.

3.3 A haploid population genetical model of the Hawk-Dove game

Although the Hawk-Dove game is often used as an example of evolutionary game theory, here we start from the ground up by developing a more rigorous model that is based on population genetics. Population genetics deals with genetic detail, so is more complete than evolutionary game theory. However, analyzing the formulas of the population genetics model is slightly more challenging, so this is why we will use R to simulate the model instead.

By comparing whether predictions from evolutionary game theory match our population genetics model, we can see whether genetic detail matters, if at all. If predictions do not match, this may tell us that lack genetic assumptions in evolutionary game theory can lead to erroneous conclusions. Such exercises are worthwhile if you want to predict when a phenotypic gambit may be an accurate vs inaccurate description of the evolution of behaviour.

3.4 Lots of crazy assumptions

For the sake of simplicity, we assume that being a hawk or dove is the result of **just two alleles at a single haploid locus**! Moreover, reproduction occurs asexually. The population sizes are assumed to be infinite. This is crazy of course, hardly any animals are asexual, let alone haploid. Moreover, even fewer are chiefly the result of variation at a single locus only.

The question is whether we would learn a similar amount from a hugely convoluted model in which we track the actual population size, have 1000s of loci or deal with sexual reproduction and all the resulting products of recombination.

3.5 Allele frequencies

Let p_t be the frequency of the allele coding for H in the population at generation t, while $1-p_t$ is the frequency of the D allele in the population at generation t. We assume that strategies are *pure*, meaning that an individual having the H or D allele will always be a hawk or dove respectively. Later on, we will consider a case where strategies are *mixed*, so that p_t does not reflect the frequency of the H allele, but the probability with which each individual plays H at each interaction.

3.6 Life cycle

We assume that generations are discrete, implying that all individuals die after reproduction. Upon birth, a newborn individual fights or displays with other individuals a single time. From this interaction, it will collect a payoff as given in the payoff matrix above. The payoff is then proportional to the number of offspring the individual produces.

3.7 Fitness

An individual reproduces according to the payoffs it has collected. Let us denote the fitness of a hawk and dove as W_H and W_D respectively, which we can interpret as the number of offspring produced by each of these types.

Starting with the fitness of a focal dove $W_{D,t}$ at generation t, we have

$$W_{D,t} = w_0 + p_t(0) + (1 - p_t)\,\frac{v}{2} \tag{3.1}$$

- the first element on the right-hand side of the equation above reflects so-called 'baseline fitness', which is the fitness that this individual accrues during other activities than displaying. It is assumed that baseline fitness is similar for hawks and doves.
- the second part reflects the probability that a dove meets a hawk (which has frequency p_t). It will then immediately retreat, hence collecting no payoff (but also incurring no costs), reflected by the (0).
- the third part reflects the probability that a dove meets another dove (which has frequency $1 - p_t$). In that case they will split the value of the resource, hence $v/2$.

3.8 Exercise

1. Based on similar reasoning as for $W_{D,t}$ above, can you write down the fitness expression $W_{H,t}$ for a focal hawk in generation t? Again, assume that there is some baseline fitness w_0 accrued from other activities than fighting:

$$W_{H,t} = w_0 + ... + ... \tag{3.2}$$

2. Can fitnesses $W_{H,t}$ and $W_{D,t}$ be negative?

3.9 Evolutionary dynamics

Now that we have derived fitness expressions of the dove and hawk alleles, let us consider how the frequencies of these alleles change over time. The frequency p_{t+1} of the hawk allele at time $t + 1$ is given by a surprisingly simple equation:

$$p_{t+1} = p_t \frac{W_H}{\bar{W}_t} \tag{3.3}$$

This is a so-called recursion equation, tracking an allele frequency *recursively* from one generation to the next. Keep an eye on this equation, as we will need it later on. The term $\bar{W}_t$ in the expression above is the average fitness in the population in generation t, given by

$$\bar{W} = p_t W_H + (1 - p_t) W_D \tag{3.4}$$

The expression for p_{t+1} above tells you that if hawks do better than the average ($W_H > \bar{W}$), the frequency of the hawk allele will increase from one generation to the next, as $W_H/\bar{W} > 1$. Similarly, if hawks do worse than average, we have $W_H/\bar{W} < 1$ and the frequency of the hawk allele decreases.

One advantage of our super-simple model is that we do not have derive a separate expression for the change in frequency of the dove allele!! This is simply $1 - p_{t+1}$. However, if we would consider more than two alleles per locus (or multiple loci), matters would be (far) more difficult.

3.10 Relation to Δp in lecture slides (only if interested)

In the lecture slides we have seen that the change in allele frequency was given by

$$\Delta p = p_{t+1} - p_t = p_t (1 - p_t) \frac{W_H - W_D}{\bar{W}} \tag{3.5}$$

If you want to know how the above relates to the equation we have seen in the lecture slides, read this! If not interested, please skip ahead to the next section.

How do we get from our recursion equation \$p_{t+1} = …\$ to the one above? The clue is in the $\Delta p = p_{t+1} - p_t$. If we substitute here for our recursion equation of p_{t+1}, we get:

$$\begin{aligned}
\Delta p &= p_{t+1} - p_t && (3.6)\\
&= p_t \frac{W_H}{\bar{W}_t} - p_t \quad \text{substitute for } p_{t+1} && (3.7)\\
&= p_t \frac{W_H}{\bar{W}_t} - p_t \frac{\bar{W}_t}{\bar{W}_t} \quad \text{common demoninator} && (3.8)\\
&= p_t \frac{W_H}{\bar{W}_t} - p_t \frac{p_t W_H + (1-p_t) W_D}{\bar{W}_t} \quad \text{expanding } \bar{W} && (3.9)\\
&= p_t (1-p_t) \frac{W_H}{\bar{W}_t} - p_t \frac{(1-p_t) W_D}{\bar{W}_t} \quad \text{clubbing terms of } W_H && (3.10)\\
&= p_t (1-p_t) \frac{W_H - W_D}{\bar{W}_t} \quad \text{clubbing terms of } p_t (1-p_t) && (3.11)\\
& && (3.12)
\end{aligned}$$

and we are done. Quite a bit of algebra to show that p_{t+1} and Δp are just two sides of the same coin, the first one is a *recursion* equation, the second one a *difference* equation.

3.11 Implementing an evolutionary algorithm in R

In order to asses whether hawks or doves win out, we need to iterate the population genetics recursion equation $p_{t+1} = p_t W_H / \bar{W}$ for a number of timesteps until there is no further change.

The `for` loop that we briefly encountered is an ideal tool to do this. To get you started, I provide a brief part of an R-script - containing mistakes and omissions - that you can modify so that it can update the recursion equation over multiple time steps until evolution stops.

```
# broken code on the evolution of hawks vs doves
# we intend to iterate the haploid population genetics
# recursion p(t+1) = p(t)*...
# but quite some things are missing or are wrong.
# Can you spot the errors / omissions?

# parameters coding the payoffs
c <- 2.0

# baseline fitness
w0 <- -100

# maximum time the simulation should run for
```

```
max_time <- 10

# a vector containing allele frequencies
# for all time steps
p <- rep(x=NA, times=max_time)

# set the initial frequency of the hawks
# at time step 1
p[[1]] <- 0.01


wD <- w0 + p[[time_idx]] * 0

# a for loop to iterate the recursion equation
# over multiple timesteps
for (time_idx  )
{
    # recursion equation
    p[[time_idx + 1]] <- p[[time_idx]] * wD
}
```

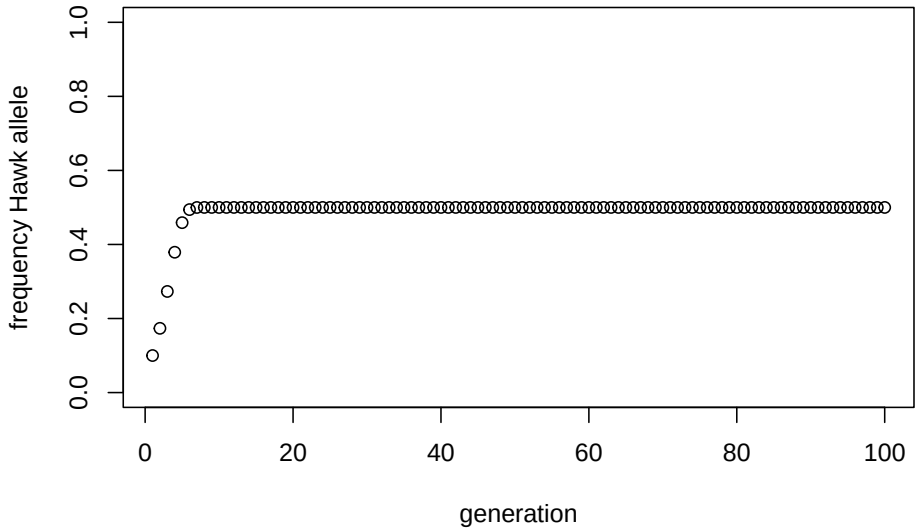
3.12 Exercise

Can you improve the code above to obtain the change in frequency of the hawk allele over `max_time` timesteps? If successful, try to explore the parameter values v and c.

3.13 Exercise

Can you try to run your script multiple times, each time for a different initial value of the frequency of hawks? Does it matter where you start? Can you plot the output from your simulation either in R or in Excel? Plotting in R can be done with

```
plot(x=1:max_time
     ,y=p
     ,ylim=c(0,1.0)
     ,xlab="generation"
     ,ylab="frequency Hawk allele")
```



3.14 Exercise

From evolutionary game theory, we know that the equilibrium frequency $\hat{p}$ of hawks (i.e., the frequency at which there is no further change in the frequency of hawks) is $\hat{p} = v/c$. Do you get the same answer here? What would you conclude about the importance of genetic detail?

Chapter 4

Individual-based simulations and drift

So far, the model that we have used was deterministic, where multiple runs for the same values of the parameters v and c always result in the same outcome. In reality, however, evolution does not work like that: lots of processes are the result of chance. This does not only apply to things like mutation, or whether or not you happen to encounter a hawk rather than a dove, but also to the sampling of newborns in the next generation. This process is called genetic drift. To include such chance processes in models of evolution, we need to work with a different model, namely a model that is stochastic!

4.1 Installing the `driftSim` package

In the following exercise we explore such a model, which is provided in an R package `driftSim`. To install this, you need to do the following

```
if (!require("devtools")) install.packages("devtools")

library("devtools")
devtools::install_github("bramkuijper/driftSim")

library("driftSim")
```

This package contains what we call an individual-based simulation, which is nothing more than a computer program that explicitly models individuals and their actions. While the package runs within R, under the hood it is coded in the low-level programming language C++. This is because large simulations tend to be very slow in R, but are much faster in C++. Below you see a little snippet from this package, reflecting the properties of single individual:

```
struct Individual {
    bool is_hawk; // TRUE/FALSE individual is hawk or dove
    double payoff; // the value of the payoff
    double prob_hawk; // the probability that this individual develops as a hawk in ea
};
```

The program then simulates a finite population (take that, population genetics) of N of these individuals. At the start of each generation, each individual interacts with a randomly chosen other individual and either attacks (hawk) or displays (dove). It gets its payoffs based on the same payoff matrix as in the deterministic model. Subsequently, we then produce N newborn offspring from this population. Parents of each of these offspring are sampled from the previous generation, with individuals who have a larger payoff being more likely to be selected than individuals which have a lower payoff.

4.2 Exercise: limiting assumptions

Individual-based simulations are often used to relax many of the limiting assumptions present in deterministic models, as writing computer code is typically much easier than devising a simple and tractable mathematical model. We have already seen how individual-based simulations have relaxed one assumption of the popoulation genetics mode, namely the fact that population sizes can now be finite, rather than infinite.

Can you list at least three limiting assumptions from the deterministic population genetics model? For each assumption, can you think how relaxing the assumption could potentially affect results? (It is often much easier to spell out what is wrong with a model than to spell out why it matters…)

4.3 Running the `driftSim` package

The `driftSim` package contains a single function, called `runSimulation()`. In order to run it, inspect the documentation by trying out `?runSimulation()`. Let us start with a simple example run with a population size of $N = 10$, $v = 1.0$, $c = 2.0$ and a bunch of different parameters. Most are not so important, except that the initial frequency $p_{t=0}$ of Hawks which is set at $p_{t=0} = 0.25$.

```
runSimulation(N=10,v=1.0,c=2.0,is_pure=T,mu=0,max_time=10,pHawk_init=0.25,output_nth_g
##    generation freq_Hawk mean_pHawkMixed sd_pHawkMixed
## 1           1       0.1               0             0
## 2           2       0.2               0             0
## 3           3       0.1               0             0
## 4           4       0.0               0             0
## 5           5       0.0               0             0
## 6           6       0.0               0             0
```

```
## 7           7       0.0                0                0
## 8           8       0.0                0                0
## 9           9       0.0                0                0
## 10         10       0.0                0                0
```

In this way, however, we cannot use the data from the `runSimulation()` function. To make sure that the resulting data will be contained in a `data.frame` by assigning the return value of the `runSimulation()` function to a variable, like this:

```
my.data <- runSimulation(N=10,v=1.0,c=2.0,is_pure=T,mu=0,max_time=10,pHawk_init=0.25,output_nth_g
```

The `my.data` variable points to a `data.frame` with the following columns: 1. `generation`: generation time point 2. `freq_Hawk`: the frequency of hawks in the population, the variable of interest 3. `mean_pHawkMixed`: in case hawks and doves are the result of a mixed strategy, this is the (evolving) probability that any individual will develop as Hawk at the start of its life. This is 0 in case `is_pure=T`, when we consider pure strategies of Hawks and Doves 4. `sd_pHawkMixed`: variation among individuals in their ability to develop as Hawks (only nonzero when `is_pure=F`)

4.4 Plotting the output from the `driftSim` package

Make a plot where plot `generation` on the x-axis and `freq_Hawk` on the y-axis. Re-run your simulation at least 10 times. What happens (typically) with the hawk allele?

4.5 Exercise: change the initial frequency of hawks

Now, change the initial frequency of Hawks by changing the value of `pHawk_init` to `pHawk_init=0.75`. Run the simulation for a long time, for example `max_time=500`. What happens now with the frequency of Hawks? What does that tell you about the phenotypic gambit, do we still reach the predicted equilibrium frequency of hawks $\hat{p} = v/c$?

4.6 Exercise: increase the population size

Now, try the same with a population size of $N = 500$. What happens now?