Evolutionary Behavioural Ecology: practicals 1 & 2 - Modelling Evolution

Bram Kuijper

2022-10-12

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 `class()` function:

```
class(var.1)
## [1] "character"
class(var.2)
## [1] "numeric"
```

As you see, `class()` does not tell you that `var.2` is a vector of multiple values, but just focuses on the type of the values contained in them (in this case a `numeric`, which is R's way of saying that we are dealing with a real number). 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 `class(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

- In the first example of a `for` loop (right below section 2.5), we used the statement `for (iterator in some.vec)`. The statement in the second example of a `for` loop is slightly more complicated, however: `for (idx in 1:length(vec.1))`. Can you explain the difference between the two statements?
- 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).

Table 3.1: The payoff matrix of the Hawk-Dove game.

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

3.2 Exercise

The parameter v clearly has something to do with gaining a resource, while c is a cost. Knowing that, can you explain the biological meaning of each of the payoffs in the table?

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

There we go: 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? Sure, it would be more realistic! But is it really full-blown reality we are after with this model? If we would be, why don't we study the real world instead? (Last thing I have heard is that the real world is a far more realistic model of reality). Perhaps the idea of a model is not to replicate reality, but to get a sharper idea about a particular phenomenon (the evolution of ritualization) while isolating lots of unnecessary detail.

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 absolute fitness of a hawk and a dove as W_H and W_D respectively, which we can interpret as the expected number of surviving 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, w_0 reflects the 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 focal dove meets a hawk (hawks are encountered with a frequency of p_t). The focal dove 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 focal dove meets another dove (doves are encountered with a frequency of $1 - p_t$). 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 you say something about the possible values that fitnesses of hawks $W_{H,t}$ and doves $W_{D,t}$ can attain? For example, can an individual have a fitness of 0.5 (or can fitness values only be integers)? And a fitness value of -1, would that be possible? Or a fitness value of $1e08$?

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 arithmetic 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 (this section is optional)

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. 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 Exercise: implementing your first evolutionary algorithm in R

In order to asses whether hawks or doves win out in the long term, in this exercise we are going to build our own evolutionary model! The goal is to iterate the population genetics recursion equation $p_{t+1} = p_t W_H / \bar{W}$ that you encountered before. We do so over a large bunch of time steps, so that we can see what happens with evolving populations like these in the long term: Will hawks win or doves? Will both happily coexist? Will frequencies fluctuate over time? Do evolutionary changes happen rapidly or slowly? All pretty important questions if we want to say something about how animals adapt to the behaviours of others.

The `for` loop that we encountered before is just the ideal tool for such a task! For example, by having a `for` statement like `for (t in 1:max_time)` we can make a variable `t` that takes the value $t = 1$, $t = 2$, $t = 3$...,$t =$`max_time` in steps of 1. This is a great way to keep track of subsequent generations. Moreover, using the recursion equation, we can then calculate the frequency of the hawk allele in these different generations.

To get you started, I provide a part of an R-script below. The task at hand is to update the recursion equation $p_{t+1} = p_t W_H / \bar{W}$ during every iteration of the `for` loop:

```
# 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

# parameters coding the payoffs
c <- 2.0

# baseline fitness
w0 <- 10

# maximum time the simulation should run for
max_time <- 1000

# 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]] <- 0.5

# a for loop to iterate the recursion equation
# over multiple timesteps
for (time_idx in )
{
    # calculate fitness of a dove
    wD <- w0 + p[[time_idx]] * 0 + (1-p[[time_idx]]) * v/2

    # calculate mean fitness
    wBar <- wD * (1 - p[[time_idx]]) + wH *

    # recursion equation
}
```

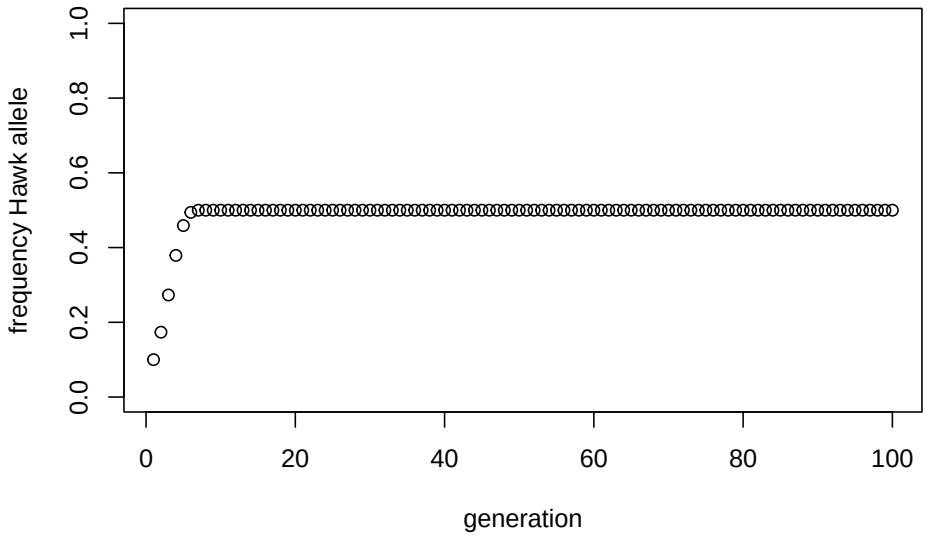
3.12 Exercise

If your model works, try to explore what happens when you vary the parameter values v and c. And what happens when you vary the value of the baseline fitness w_0?

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

Previously, the model of the Hawk-Dove game 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, as quite a number of biological processes are the result of **chance**. To give some examples: chance plays a big role in determining whether a mutation occurs or not, whether or not you happen to encounter a hawk rather than a dove, which gamete actually contributes to the next generation or which offspring manages to survive in order to reproduce.

Introducing chance in models of evolution makes these models **stochastic**, meaning that any two runs of the same model (with the same values of the parameters) does not necessarily result in the same outcome. Stochasticity in evolution introduces a phenomenon called 'genetic drift' which we will explore in this practical. Genetic drift can – particularly in small populations – dramatically affect the course of evolution!

4.1 Before you start this practical

First, please be reminded to use one of the Rstudio servers rather than your local installation of R. This is because we will be using some custom-built R packages, which themselves require extensions like Rtools that can be a bit of a pain to install on some computers. Luckily, this should all have been done for you on the Rstudio server.

Next, when logged in on the Rstudio server, please use R version `4.1.1`.

And finally, it may be worthwhile to remove variables from our previous session, which we do by typing in the following

```
# clean all R variables
rm(list=ls())
```

4.2 Installing the `driftSim` package

Now, let's get started. In this part of the practical, we will explore a model that includes stochasticity. Such a model is provided in the R package `driftSim`. To install this package, do the following:

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

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

If you get a message like `Skipping install of 'driftSim' from a github remote, the SHA1 (27e9c52c) has not changed since last install. Use`force = TRUE`to force installation`, R is simply telling you the package has already been installed. Consequently, you can just move on to the next step.

Then we can load the library using:

```
library("driftSim")
```

4.3 Individual-based models

This package contains what we call an *individual-based simulation*, which is nothing more than a computer program that models individuals and their actions (e.g., surviving, fighting, reproducing). By contrast, in the previous set of exercises, we did not focus on individuals too much, but rather on an *analytical model* which tracked the population-wide frequency p_t of hawks. Indeed, keeping track of a single population-wide frequency p_t is a lot simpler than keeping track of thousands or even millions of individuals. However, such simplicity also comes also at a price, which will be the focus of this chapter.

While the simulation package `driftSim` 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 run 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 of N of these individuals. You might remember that the analytical model in the previous chapter assumed infinitely many individuals, so while the individual-based model is an absolute loser when it comes to speed, it is already a winner when it comes to increased realism!

4.4 Life cycle

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 we used in the deterministic model in the previous chapter. 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.

4.5 Exercise: limiting assumptions

Individual-based simulations are often used to relax many of the limiting assumptions present in deterministic models, as writing computer code allows for much more flexibility than when using mathematical formulas. 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 three other limiting assumptions from the deterministic population genetics model that you would like to relax? 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 how addressing it will change outcomes!)

4.6 Running the `driftSim` package

The `driftSim` package contains a single function, called `runSimulation()`. Before running this function, please inspect the documentation by typing `?runSimulation()` in your R console, after which a help page should show up.

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_generation=1
##    generation freq_Hawk mean_pHawkMixed sd_pHawkMixed
```

```
## 1           1        0.1                0                0
## 2           2        0.2                0                0
## 3           3        0.2                0                0
## 4           4        0.2                0                0
## 5           5        0.2                0                0
## 6           6        0.2                0                0
## 7           7        0.1                0                0
## 8           8        0.2                0                0
## 9           9        0.1                0                0
## 10         10        0.2                0                0
```

In this way, however, we cannot use the data from the `runSimulation()` function, as it will simply be printed out to the console, but it is not stored anywhere. To make sure that the resulting data will be contained in a `data.frame`, we assign the return values from 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,ou
```

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`)

4.7 Exercise: 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?

4.8 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 differences between the analytical model and the current one? Do we still reach the predicted equilibrium frequency of hawks $\hat{p} = v/c$?

4.9 Exercise: increase the population size

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

Chapter 5

Hawk Dove games with relatives

So far, we used the Hawk Dove game to study the evolution of fighting behaviour in well-mixed populations. We found that aggression can be limited by costs of fighting c.

In this practical, we take the Hawk Dove game a step further by considering how interactions among relatives affect the evolution of aggression. Indeed, one could argue that a Hawk could be analogous to a defector that exploits cooperative Doves.

In the simple model we used previously, interactions were entirely driven by *random* encounters, so that the probability of encountering one type vs the other was given by the frequencies of Hawks p_t and Doves $1 - p_t$ in generation t.

Now we study the case where interactions are not random, but some assortment takes place. Hence, we need to change the model in the same way the prisoner's dilemma model is changed to account for non-random interactions.

While we will use the same payoff matrix as before (see the table in chapter 2), let us now change the interaction probabilities between the individuals. Recall the previous fitness expressions W_H and W_D for Hawks and Doves respectively:

$$W_H = w_0 + p_t \frac{v-c}{2} + (1-p_t)\, v \tag{5.1}$$

$$W_D = w_0 + p_t(0) + (1-p_t)\, \frac{v}{2} \tag{5.2}$$

$$\tag{5.3}$$

Below you find the R code that we used to iterate the system of equations.

```
# code to iterate a hawk-dove game


# parameters coding the payoffs
c <- 1.0
v <- 0

# baseline fitness
w0 <- 10

# maximum time the simulation should run for
max_time <- 1000000

# 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]] <- 0.1

# a for loop to iterate the recursion equation
# over multiple timesteps
for (time_idx in 1:(max_time-1))
{
  wH <- w0 + p[[time_idx]] * (v-c)/2 + (1-p[[time_idx]]) * v
  wD <- w0 + p[[time_idx]] * 0 + (1-p[[time_idx]]) * v/2

  wBar <- wH * p[[time_idx]] + wD * (1-p[[time_idx]])

  # recursion equation
  p[[time_idx + 1]] <- p[[time_idx]] * wH / wBar
}
```

5.1 Exercise: running the model

Just to get yourself familiar with the model again, try to run it for different values of v and c and see what happens.

5.2 Interactions among relatives

Rather than using the frequencies p_t and $1 - p_t$, we will now modify the Hawk Dove game by using *conditional probabilities* reflecting non-random encounter rates, replacing the random encounter rates.

- $$\Pr(H \mid H)$$
  probability of encounter between a focal Hawk and non-focal Hawk.
- $$\Pr(D \mid H)$$
  probability of encounter between a focal Hawk and a Dove (note again, that $\Pr(H \mid H) + Pr(D \mid H) = 1$).
- $$\Pr(H \mid D)$$
  probability of encounter between a focal Dove and a Hawk.
- $$\Pr(D \mid D)$$
  probability of encounter between a focal Dove and a non-focal Dove ($\Pr(H \mid D) + \Pr(D \mid D) = 1$).

Incorporating these conditional probabilities into the fitness expressions above yields

$$W_H = w_0 + \Pr(H \mid H)\frac{v-c}{2} + \Pr(D \mid H)v \tag{5.4}$$

$$W_D = w_0 + \Pr(H \mid D)(0) + \Pr(D \mid D)\frac{v}{2} \tag{5.5}$$

$$\tag{5.6}$$

5.3 Hamilton's rule in a Hawk Dove game?

If we would follow the example of the Prisoner's dilemma in the lecture slides, would we expect to find something similar to Hamilton's rule when we assess the condition in which more cooperative Doves do better than defecting Hawks, i.e., $W_D > W_H$?

Let us look again at the Prisoner's dilemma: altruists (A) do better than nonaltruists NA (originally called 'Defect' in the lecture slides but I change notation to avoid confusion with Dove) when $W_A > W_{NA}$, which eventually results in (see lecture slides):

$$(\Pr(A \mid A) - \Pr(A \mid NA))\, b > c \tag{5.7}$$

$$\tag{5.8}$$

where the term $\Pr(A \mid A) - \Pr(A \mid NA)$ can be shown to reflect the coefficient of relatedness r.

In case Hamilton's rule would apply to the Hawk Dove game, we would thus expect an expression of the form

$$(\Pr(D \mid D) - \Pr(D \mid H))\, x > y \tag{5.9}$$

$$\tag{5.10}$$

where x and y would be coefficients that would result from combinations of payoffs in the Hawk Dove game.

Yet, actually working out the same tricks on the Hawk Dove game as was done on the Prisoner's Dilemma in the preceding lecture, we obtain something rather different, namely

$$[\Pr(D \mid D) - \Pr(D \mid H)]\, v - \Pr(D \mid H)v > \Pr(H \mid H)\,(v - c) \tag{5.11}$$

Clearly, coefficients are not simply v on the left hand side and c on the right, but that is obvious as costs are not carried by the cooperators, but rather by pairs of interacting Hawks. But beyond this, there is something more important to note in the expression above, namely that next to the relatedness coefficient $\Pr(D \mid D) - \Pr(D \mid H)$ there is an additional coefficient on the left-hand side (and the right-hand side is multiplied by $\Pr(H \mid H)$).

Hence, rather than anything like $rx > y$ we now have $rx + kz > gy$ (i.e., lots more terms)!! And that despite it being a super simple 2 player game! What is going on??

5.4 Exercise: the downfall of Hamilton's rule

Can you think of why a simple expression like $rx > y$ does not apply to the Hawk Dove game? Specifically, compare the payoff matrix for the Prisoner's dilemma (see lecture slides) with the one of the Hawk dove game (see the table in chapter 2). What happens if you move from the top row to the bottom row in the payoff matrix, does the payoff increase / decrease by the same amount in the Hawk Dove game vs the Prisoner's Dilemma? What happens if you move from the left column to the right column? Discuss.

5.5 Evolution of aggression when relatives interact in R

Now that we know we cannot really use Hamilton's rule to predict when Doves will outlive Hawks, we have to resort to our previously used method of iterating a population genetics recursion equation in R to predict how interactions among relatives affects aggression. It turns out, relatedness r comes into this in another way, namely directly through the conditional probabilities. If we interpret r as being the chance of interacting with an individual that shares an allele identical by descent, we have

- $$\Pr(H \mid H) = r \times 1 + (1 - r)p_t$$

  probability of encounter between a focal Hawk and non-focal Hawk.

- $$\Pr(D \mid H) = r \times 0 + (1 - r)(1 - p_t)$$

  probability of encounter between a focal Hawk and a Dove (note that $\Pr(H \mid H) + Pr(D \mid H) = 1$).

- $$\Pr(H \mid D) = r \times 0 + (1 - r)p_t$$

  probability of encounter between a focal Dove and a Hawk

- $$\Pr(D \mid D) = r \times 1 + (1 - r)(1 - p_t)$$

  probability of encounter between a focal Dove and a non-focal Dove ($\Pr(H \mid D) + \Pr(D \mid D) = 1$).

5.6 Exercise: explaining these expressions?

Can you try to explain what is going on in the expressions above? Why do allele frequencies p_t still feature here?

5.7 Exercise: implementing it in the numerical model

1. Copy-paste the code of the correct numerical iterations of the hawk dove game above and modify the encounter rates to allow for nonrandom interactions. While p_t is still the allele frequency of the Hawk allele, you have to add r as a constant parameter, which you can give any value between $0 \leq r \leq 1$ before you start the iteration. Start with $r = 0.5$, for example, what does this mean about the relatives any individual is most likely to interact?

2. Is it reasonable to think as r as a constant parameter?

5.8 Exercise: varying the values of r

Plot your results for different values of r, either in Excel or by using one of the plotting packages in R. Is the relationship between aggression and r in line with your expectations?

Chapter 6

Spatially explicit models of the Hawk Dove game

In the previous set of exercises, we discussed our assumption of r being constant. While this allows you to assess the role of specific values of r, in reality r emerges as a result of the ecology of a population, allowing us to focus on the ecological factors that drive the evolution of sociality. Models in which r is not treated at a constant, but *emerges* are called *demographically consistent*.

One major ecological factor that affects r is the probability of dispersal: if few individuals disperse, while most just stick around in their natal environment, relatedness among randomly sampled individuals is likely to be much higher than when most individuals disperse and go elsewhere.

6.1 Exercise

Can you think of other ecological factors that influence r?

6.2 Investigating how dispersal affects aggression: adding space to the Hawk Dove game

Given the results from the numerical model of the Hawk Dove game among relatives, we would thus predict that reducing the rate of dispersal (which results in an increase in r) should reduce the evolution of aggression.

To investigate this prediction, we need a model that includes some notion of space, as individuals need to disperse from one patch to another. There lots of different ways to include space in models. Most commonly used are:

1. **two-patch models**: in which a population distributed across (lo-and-behold) two sites, with migration occurring between them (example: island vs mainland). In this case, at least one of the environments typically contains lots of individuals (the mainland), making it unlikely that relatives encounter each other in a large proportion of the population. If this is the ecological setting, interactions between relatives are likely to be less important anyway.
2. **lattice models**: individuals migrate around on a grid of patches, moving from one patch to an adjacent one per timestep. If this is the ecology, then interactions among relatives are much more likelyt to occur, as even when individuals migrate, they are still close to the natal site and hence likely to interact with other relatives that have dispersed. However, at the same time, this means that what happens in one patches correlates with what happens in another, which makes these models much more tedious to model. Also there are loads of additional assumptions involved (e.g., what do you do at the edge of a lattice? Do you move to the other side (i.e., a torus world) or do you bounce back?
3. **island models**: in this flavour of a spatial model, once individuals migrate, they move to a randomly chosen site in the population. This may well describe taxa such as many insects where individuals move relatively away from their natal patch, while the non-dispersers stay at their natal site. Next to the two-patch case, this is the easiest version of a spatial model, so that is what we will be focusing on here.

6.3 Exercise

Discuss the pros and cons of each model - what approach has your preference and why?

The life cycle of the island model is similar to that of the numerical one before: 1. individuals are born; 2. they disperse or not and compete for one of N_{bp} breeding positions with native and immigrant juveniles; 3. they interact with a randomly chosen patch mate and obtain a payoff; 4. the number of offspring that are produced are then proportional to the payoff they have received.

6.4 A spatial model of the hawk dove game

In `R` (make sure you use version 4.1.1), we now proceed to install the package `hdIslandModel` which is an individual based simulation of an island model with Hawks and Doves, allowing us to vary the rate of dispersal.

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

library("devtools")
```

```
devtools::install_github("bramkuijper/hdIslandModel")
```

Then we can load the library using:

```
library("hdIslandModel")
```

The model contained in the package can be called by the command `runHDSpace()` which spits out a `data.frame` containing a bunch of statistics resulting from the simulation that we can save in a variable, say `sim.standard`:

```
sim.standard <- runHDSpace()
```

6.5 Exercise

1. Take a look at the documentation of the function `runHDSpace()` by typing `?runHDSpace()`.
2. Let's run the island model for a payoff in which we would expect an evolutionarily stable strategy (ESS) to occur at $\hat{p} = 0.5$ (frequency Hawks is 50%) when interactions among relatives are absent. This happens happens when $\hat{p} = v/c = 1/2$. To run the model, focus on a pure strategy (do you remember what this means?) with a small mutation rate of $mu = 0.01$ and $N = 10$ breeders patch. Set the dispersal rate to something low, like $d = 0.1$, which typically results in quite high relatedness coefficients of $r \approx 0.3 - 0.4$ among randomly sampled patch mates. The other parameters can be left at their default values.
3. In case dispersal indeed increases r, what would we expect to see happen with the proportion of Hawks over time, should it be smaller or larger than 0.5? What is the proportion of Hawks at the end of the simulation? (Run multiple runs, as drift may cause outcomes to vary).
4. What happens if you set the dispersal rate to $d = 0.8$? Does this make a difference?

6.6 Exercise

What can we conclude about the relationship between dispersal and social behaviour from the above? Why does this happen, you think?

6.7 Exercise

Perhaps something different happens if we allow dispersal to evolve? Run your simulation for $max_time = 20000$ and see what happens with the dispersal rate. Does dispersal evolve in a predictable way?

6.8 Exercise

Now incorporate the modulated